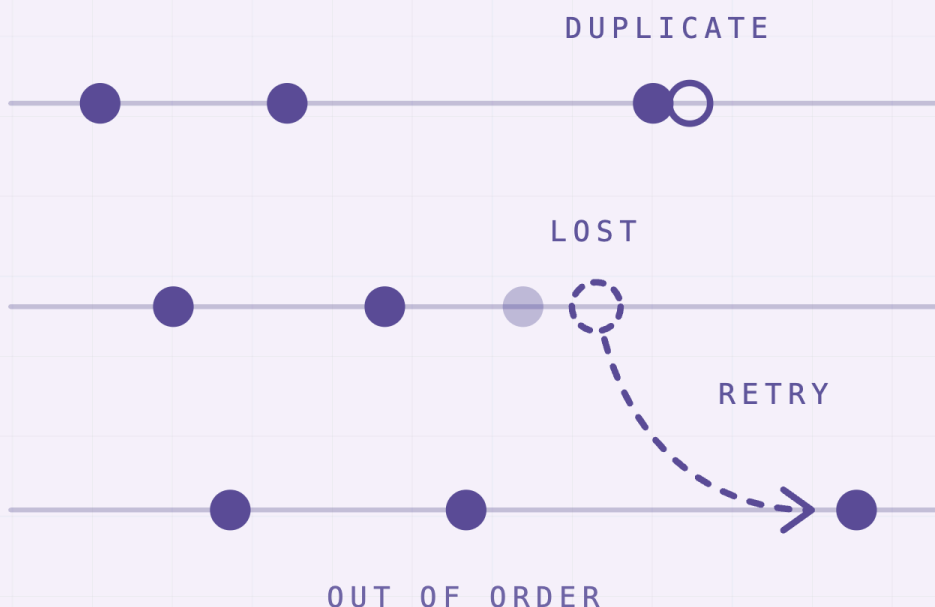




Projetando sistemas confiáveis orientados a eventos

Projetando sistemas confiáveis orientados a eventos

Mensageria, retries, idempotência, outbox, ordenação e evolução de esquemas

Muhammet Şafak

Sumário

Prefácio	5
PARTE I — FUNDAMENTOS	12
Capítulo 1 — A mentira do happy path	13
Capítulo 2 — Semântica de mensageria	26
Capítulo 3 — Um modelo de falhas prático	44
PARTE II — PUBLICAR COM CONFIABILIDADE	63
Capítulo 4 — O problema do dual write e o transactional outbox	64
Capítulo 5 — Relays, polling e change data capture	84
PARTE III — CONSUMIR COM CONFIABILIDADE	106
Capítulo 6 — Retries bem-feitos	107
Capítulo 7 — Idempotência	132
Capítulo 8 — Poison messages e dead letters	155
Capítulo 9 — Ordenação	178
PARTE IV — FLUXOS DE TRABALHO E CONTRATOS	200
Capítulo 10 — Sagas e fluxos de trabalho de longa duração	201
Capítulo 11 — Design de eventos	223
Capítulo 12 — Evolução de esquemas	243
PARTE V — OPERAR E COMPROVAR	265
Capítulo 13 — Observabilidade e reconciliação	266
Capítulo 14 — Recuperação, desastres e migração de brokers	289
Capítulo 15 — Como testar sistemas orientados a eventos	315
Capítulo 16 — Um checklist de confiabilidade e reflexões finais	333
Apêndice: Glossário e referência rápida de padrões	346

Prefácio

Imagine uma tarde de terça-feira na Parcelly, um marketplace que não existe. O fluxo de checkout está saudável há meses. Um comprador faz um pedido, o serviço Orders grava o pedido no banco de dados e um evento `OrderPlaced` é enviado ao broker. O Payments autoriza o cartão. O Inventory reserva o estoque. O Shipping agenda a encomenda. Todos os dashboards estão verdes.

Então um deploy passa por Orders. Cada instância trata um pedido da mesma forma: faz o commit do pedido no banco de dados e, em seguida, publica `OrderPlaced` no broker. Durante alguns segundos, instâncias estão sendo encerradas enquanto outras sobem. Uma instância faz o commit de um pedido e é interrompida antes da chamada de publicação. O pedido está guardado com segurança, e o evento nunca sai. Nada falha com estardalhaço e nenhum alerta dispara. Na manhã seguinte, o comprador está diante de um pedido marcado como realizado, mas o Payments nunca autorizou um cartão e o Inventory nunca reservou um único item. O Orders acredita que o trabalho está em andamento, e o resto do sistema nunca ouviu falar dele.

Ninguém escreveu um bug no sentido habitual. Cada linha de código fez o que dizia. O sistema falhou no espaço entre as linhas: entre um commit no banco de dados e uma chamada de rede, entre uma mensagem ser entregue e ser compreendida, entre “nós enviamos” e “eles fizeram”.

Esse espaço é o assunto deste livro.

Por que este livro existe

A arquitetura orientada a eventos tem um problema de reputação, e ele é merecido nas duas direções. Seus defensores descrevem serviços fracamente acoplados, que escalam de forma independente e evoluem sem co-

ordenação. Seus críticos descrevem um espaguete distribuído, em que ninguém sabe dizer o que acontece quando um pedido é feito. Ambos descrevem o mesmo sistema em dias diferentes.

O que separa os dois dias raramente é a escolha do broker, do framework ou mesmo do diagrama de arquitetura. É uma lista curta de perguntas sem glamour. O que acontece se esta mensagem for entregue duas vezes? O que acontece se ela nunca chegar? O que acontece se ela chegar depois da mensagem que deveria vir depois dela? O que acontece se o consumidor falhar todas as vezes? O que acontece quando o produtor precisa mudar um campo e cem consumidores ainda executam o código antigo?

Essas perguntas são a tese deste livro. Sistemas confiáveis orientados a eventos são construídos em torno da **semântica de falhas**, não dos message brokers: para uma dada junção, o que pode falhar, como você observaria essa falha e qual garantia sobrevive a ela. Uma junção é qualquer ponto em que um componente entrega trabalho a outro, como um commit no banco de dados seguido de uma publicação, ou uma entrega seguida de um efeito. Os padrões aqui existem por causa da rede e do banco de dados, não por causa de um produto específico, e um broker é só mais um componente com junções próprias.

As equipes costumam encontrar essas perguntas uma de cada vez, em produção, na ordem em que o sistema por acaso falha. Cada resposta é aprendida separadamente, e as lições raramente acabam reunidas em um só lugar. A documentação de um broker diz como publicar e consumir. Fala muito menos sobre como é um serviço correto em torno desses dois verbos.

Este livro tenta ser esse lugar. É um guia dos padrões que tornam confiáveis os sistemas orientados a eventos: o transactional outbox, consumidores idempotentes, retries com backoff e jitter, backpressure, dead-letter queues, estratégias de ordenação, sagas, design cuidadoso de eventos, evolução de esquemas, observabilidade, recuperação de dados perdidos e de uma troca de broker, e testes. Procurei tratar cada um como uma decisão com um preço, em vez de uma regra a seguir. Um padrão que é adequado para os pagamentos da Parcelly pode ser um desperdício para suas notificações, e o livro dirá isso quando fizer diferença.

Também quero ser direto sobre o que o livro não é. Não é um tutorial de nenhum broker específico. Não é um catálogo de todos os padrões de integração corporativa já nomeados. Não promete que segui-lo tornará o seu sistema correto. Confiabilidade não é uma propriedade que se acrescenta; é um conjunto de falhas cujo tratamento você decidiu, de forma deliberada, e um conjunto que você decidiu, também de forma deliberada, aceitar. Meu objetivo é tornar essas decisões visíveis.

Para quem é este livro

O livro foi escrito para **engenheiros de backend e de plataforma** que constroem ou operam serviços que se comunicam por eventos, e para os **tech leads** que precisam decidir como esses serviços devem se comportar quando as coisas dão errado. Se você já ficou olhando para uma dead-letter queue que cresceu durante a noite, já discutiu se um consumidor precisa ser idempotente ou herdou um topic cujos eventos ninguém consegue explicar por completo, você é a pessoa leitora que tenho em mente.

Você não precisa ser especialista em mensageria. Deve estar à vontade com o seguinte:

- Bancos de dados relacionais e transações: o que é um commit, o que faz uma restrição de unicidade e o que significa isolamento, em linhas gerais.
- Serviços que conversam pela rede e o vocabulário básico de timeouts e de requisição e resposta no estilo HTTP.
- A ideia geral de um message broker: algo que aceita mensagens de produtores e as entrega a consumidores.

Você não precisa de experiência com nenhum produto em particular.

Quando um capítulo cita Kafka, RabbitMQ ou Amazon SQS, faz isso como exemplo de uma família de sistemas e descreve um comportamento que vale em geral, não em uma versão específica. Se você não usou nenhum deles, o livro ainda fará sentido. Se você usou algum a fundo, vai perceber que parte do que sabe é específica dele, e o livro tentará indicar quais partes.

Se você é um tech lead que não escreve código todos os dias, pode ler o cenário de abertura, a seção de trade-offs e os pontos-chave de cada capítulo e extrair a maior parte do valor para a tomada de decisão. O pseudo-código está ali para quem precisa ver onde exatamente fica a fronteira da transação.

Como o livro está organizado

O livro tem cinco partes, e a ideia é que sejam lidas em ordem na primeira vez e usadas em qualquer ordem depois.

Parte I, Fundamentos (capítulos 1 a 3). Começamos desmontando o happy path. O Capítulo 1 mostra como um fluxo que parece simples esconde uma dúzia de decisões. O Capítulo 2 define com precisão o que as garantias de mensageria realmente significam e por que a entrega at-least-once é a linha de base realista. O Capítulo 3 constrói um modelo de falhas prático, um pequeno vocabulário para as formas como as coisas quebram, de timeouts e crashes a mensagens que sempre falham e à perda de um armazenamento inteiro. O restante do livro o usa como linguagem comum.

Parte II, Publicar com confiabilidade (capítulos 4 e 5). O Capítulo 4 aborda o problema do dual write, exatamente a lacuna da cena de abertura, e apresenta o transactional outbox. O Capítulo 5 trata da engrenagem que leva as linhas do outbox até o broker, comparando relays com polling e change data capture, e o que cada um custa.

Parte III, Consumir com confiabilidade (capítulos 6 a 9). É a parte mais longa, porque os consumidores são o lugar onde acontece a maioria dos efeitos duplicados, das filas travadas e das atualizações perdidas. O Capítulo 6 trata dos retries: quando fazer um retry, quando não fazer, por que o backoff precisa de jitter e como impedir que um consumidor seja sobrecarregado pelos próprios retries. O Capítulo 7 trata de idempotência e do inbox. O Capítulo 8 lida com poison messages e dead letters. O Capítulo 9 se volta para a ordenação e para o quão pouco dela se pode, em geral, exigir.

Parte IV, Fluxos de trabalho e contratos (capítulos 10 a 12). O Capítulo 10 apresenta as sagas, para fluxos de trabalho que atravessam serviços e não podem compartilhar uma transação. O Capítulo 11 trata do design

dos eventos em si: o que entra neles, o que fica de fora, como são nomeados e como um envelope sobrevive a ser gravado e lido em várias linguagens. O Capítulo 12 trata da evolução de esquemas, a disciplina discreta que permite a produtores e consumidores mudar em velocidades diferentes.

Parte V, Operar e comprovar (capítulos 13 a 16). O Capítulo 13 trata de enxergar o que o seu sistema está fazendo e da reconciliação, o hábito de conferir a visão de um sistema com a de outro. O Capítulo 14 trata de se recuperar de dados perdidos, de uma região perdida e de uma troca de broker. O Capítulo 15 mostra como testar código cujos piores bugs só aparecem sob duplicação, atraso e reordenação. O Capítulo 16 reúne tudo em um checklist de confiabilidade e encerra o livro.

Um apêndice traz um glossário e uma referência rápida dos padrões, para que você consulte algo sem reler um capítulo.

Todo capítulo de 1 a 15 segue o mesmo formato. Abre com um cenário curto, explica por que o problema é difícil, apresenta o padrão e então passa a duas seções que considero as mais valiosas: **Trade-offs e custos**, que diz quando não usar o padrão, e **Erros comuns**. Cada capítulo termina com cinco a sete pontos-chave. Se o seu tempo é curto, essas três partes levarão você mais longe do que se imagina.

Você pode ler os capítulos de forma seletiva. Se a sua dor hoje é uma dead-letter queue crescendo, vá direto ao Capítulo 8 e siga as referências de volta aos capítulos 6 e 7 conforme precisar. Se você está projetando um serviço novo, leria a Parte II antes de escrever a primeira linha do código de publicação, porque o outbox é muito mais fácil de adotar no início do que de introduzir depois.

Conheça a Parcelly

Ao longo do livro, um sistema fictício sustenta os exemplos: a **Parcelly**, um marketplace online em que vendedores independentes anunciam mercadorias e compradores as pedem. Ela tem seis serviços.

- **Orders** é responsável pelo ciclo de vida do pedido e é onde começa a maioria dos fluxos de trabalho.

- **Payments** autoriza e captura o dinheiro, e conversa com um provedor de pagamento externo.
- **Inventory** controla o estoque e mantém reservas temporárias.
- **Shipping** agenda os envios e entrega as encomendas às transportadoras.
- **Notifications** envia e-mails, mensagens push e SMS, e nunca deve bloquear nada mais.
- **Ledger** mantém um registro append-only, de partidas dobradas, das movimentações de dinheiro.

Cada serviço tem o próprio banco de dados relacional e não lê o de ninguém mais. Eles se comunicam por um broker, trocando eventos com nomes como `OrderPlaced`, `PaymentAuthorized`, `StockReserved`, `PaymentFailed` e `ShipmentScheduled`. Os eventos seguem um formato consistente, um envelope que carrega um identificador, um tipo, uma versão de esquema, campos de correlação e um payload, para que os capítulos possam falar deles a uma só voz.

Escolhi um marketplace de propósito. Receber dinheiro, reservar mercadorias e movimentar encomendas é um domínio em que o custo de uma mensagem perdida ou de um efeito duplicado se sente rápido. Um e-mail duplicado é um incômodo. Uma captura duplicada é um reembolso, um ticket de suporte e um pedido de desculpas. Uma reserva perdida é um item vendido além do estoque. Esse gradiente, do inofensivo ao caro, é exatamente o que você precisa para decidir quanta maquinaria de confiabilidade cada fluxo merece.

A Parcelly é ficção, e seus incidentes também. Os cenários que abrem cada capítulo são inventados para ilustrar o formato de uma falha, e sempre são apresentados como ficção. Vou usar a primeira pessoa para julgamentos, como “adicionaria uma restrição de unicidade aqui” ou “na minha opinião isso não vale a complexidade”, mas não vou enfeitar raciocínio com histórias de guerra. Você deve poder confiar que, quando o livro diz que algo acontece, é por causa de como esses sistemas funcionam, e não por causa de uma anedota.

O livro também é deliberadamente **agnóstico quanto ao broker**. Quando um ponto depende de o seu broker ser baseado em log, com consumidores controlando um offset em uma partição ordenada, ou baseado em fila,

com mensagens que recebem ack e são reentregues uma a uma, direi qual é o caso. Quando não faz diferença, não vou fingir o contrário.

Como ler o código

O código neste livro é **pseudocódigo**. Ele não pertence a nenhuma linguagem, não compila e nada nele chama uma biblioteca real. Isso é intencional. O objetivo de cada trecho é mostrar onde uma transação começa e termina, onde uma mensagem recebe o ack e o que acontece quando um passo falha no meio. Esses fatos são os mesmos em todas as linguagens, e a sintaxe só atrapalharia.

Os blocos são curtos, com no máximo 30 linhas. Onde um esquema de tabela ajuda, ele é escrito em uma notação parecida com SQL. Os diagramas usam Mermaid, e cada um é introduzido por uma frase que diz o que observar. Se você traduzir um padrão para a sua própria stack, trate o pseudocódigo como uma especificação de ordem e atomicidade, não como uma implementação.

Uma nota sobre a certeza

Farei recomendações neste livro e mantereí algumas delas com firmeza. Não publicaria eventos com um dual write simples, por exemplo, quando a perda de um evento importa para o negócio. Outras recomendações são questão de gosto ou de contexto, e tentarei dizer qual é qual.

Haverá também lugares em que a resposta honesta é “depende, e aqui está do quê”. Retry budgets, retenção de dead letters, número de partições e o nível certo de ordenação dependem de fatos sobre o seu negócio e a sua equipe que não consigo ver. Nesses lugares, vou nomear as variáveis em vez de inventar um número. Se você terminar um capítulo sabendo quais perguntas fazer no seu próprio sistema, considerarei isso um sucesso.

O trabalho de confiabilidade tem uma propriedade infeliz: quando dá certo, nada visível acontece. Os pedidos são atendidos, os livros-razão fecham e ninguém escreve uma mensagem para agradecer. Espero que este livro ajude você a construir esse tipo de silêncio.

Vamos começar pelo happy path e pelos motivos para não confiar nele.

PARTE I

Fundamentos

Capítulo 1 — A mentira do happy path

Imagine a Parcelly em seu primeiro ano. O Orders chama o Payments por HTTP, espera a resposta, depois chama o Inventory, espera de novo, depois chama o Shipping. O código se lê de cima para baixo, uma falha é uma exceção, e o comprador que vê uma página de erro pode tentar de novo. É lento quando o provedor de pagamento está lento e cai quando o Inventory está fora do ar, mas ninguém fica confuso sobre o que aconteceu.

Então a equipe decide migrar para eventos. O Orders publicará `OrderPlaced` em um broker, e os outros serviços reagirão no seu próprio ritmo. A demonstração é linda. O comprador clica em “Fazer pedido” e, em um segundo, o estoque está reservado, o cartão foi capturado, o pedido está confirmado e um envio está agendado. Todos aplaudem.

Agora imagine o primeiro mês em produção. O comprador é cobrado, mas o pedido fica “pendente” para sempre. Outro pedido é enviado duas vezes. Um terceiro tem um lançamento no ledger sem o pagamento correspondente, e ninguém sabe dizer qual serviço está certo. O suporte pergunta: “o que aconteceu com o pedido 81724?”, e a resposta honesta é que quatro serviços conhecem, cada um, um pedaço dele e nenhum conhece tudo.

Nada na demonstração estava errado. Ela estava incompleta, porque uma demonstração é uma história em que todo passo dá certo. A pergunta que este capítulo responde é o que a equipe abriu mão ao deixar o modelo de requisição e resposta, e o que “confiável” precisa significar antes que alguém comece a reconstruir isso.

O problema e por que ele é difícil

O happy path é a sequência de eventos que ocorre quando todo componente está de pé, toda chamada de rede chega, toda mensagem é entregue uma vez e em ordem, e todo consumidor termina o que começa. É o cami-

nho que você desenha no quadro branco e o caminho que uma demonstração exercita. É também o caminho mais raro em um sistema que roda por meses.

As falhas não são exóticas. Deploys reiniciam processos. Conexões caem no meio de uma gravação. Um broker faz failover para uma réplica. Um consumidor leva mais tempo do que o esperado e sua mensagem é entregue a outro. Cada uma dessas coisas acontece em algum lugar de um sistema movimentado todos os dias. O formato habitual de um mau mês em produção é um punhado de eventos comuns encontrando código escrito para o happy path.

O que torna isso difícil é que os sistemas orientados a eventos removem justamente aquilo que tornava as falhas visíveis. Em uma chamada síncrona, quem chama espera, e o resultado volta na mesma conexão: um retorno ou um erro, ao lado do código que o causou. Em um design orientado a eventos, o produtor entrega uma mensagem ao broker e segue em frente. Se algo dá errado três serviços e dez segundos depois, o produtor já se foi há muito tempo, e a evidência está espalhada por logs que ninguém correlacionou ainda.

A resposta óbvia é tornar cada peça confiável: um bom broker, serviços bem testados, retries em toda chamada. Cada um ajuda, e nenhum basta, porque as falhas que machucam moram nas lacunas entre os componentes. Uma mensagem gravada no banco de dados mas nunca publicada, ou processada mas que nunca recebeu ack, não é um bug em nenhum dos componentes. É uma propriedade da junção. Boa parte deste livro trata de junções.

Para cada junção, o livro faz as mesmas três perguntas: o que pode falhar, como você observaria isso e qual garantia sobrevive. Vou chamar as respostas de **semântica de falhas** da junção. A documentação de um broker dá a semântica das junções do próprio broker. As junções ao redor dele, entre o seu banco de dados e a sua chamada de publicação ou entre uma entrega e o seu efeito, cabem a você descobrir, e o Capítulo 3 mostra como.

Acoplamento síncrono e assíncrono

A migração para eventos costuma ser descrita como “desacoplamento”, uma palavra que promete mais do que entrega. É mais exato dizer que o acoplamento muda de forma.

Uma chamada síncrona acopla dois serviços no **tempo**, na **disponibilidade** e no **conhecimento**. Quem é chamado precisa estar rodando agora, a latência de quem chama inclui a de quem é chamado, e quem chama sabe quem é o outro e o que ele vai responder. O benefício é um modelo de falhas simples: quem chama sabe que a chamada falhou e pode decidir de imediato fazer um retry, recorrer a um fallback ou informar um erro. O custo é que a cadeia só é tão disponível quanto o seu elo mais fraco. Uma API lenta de transportadora vira um checkout lento.

Uma mensagem que passa por um broker remove o acoplamento no tempo e na disponibilidade. O Orders pode aceitar um pedido enquanto o Inventory reinicia, porque o broker guarda a mensagem até o Inventory voltar. Um novo consumidor, como um serviço de analytics, pode começar a escutar sem que o Orders mude. Esse ganho é real, e é a razão para adotar o estilo.

Mas o acoplamento mudou de lugar, não desapareceu. Os serviços agora estão acoplados pelo **contrato do evento**: seu nome, seus campos e seu significado. Estão acoplados pelo **comportamento do broker**: o que ele promete sobre durabilidade, ordenação e entrega. E estão acoplados por suposições sobre o timing uns dos outros que ninguém escreveu. O Capítulo 11 e o Capítulo 12 tratam do contrato, e o Capítulo 2 trata do broker.

Em resumo, os sistemas síncronos falham com estardalhaço e em cadeia. Os sistemas assíncronos falham em silêncio e em pedaços.

O preço do desacoplamento temporal

Desacoplamento temporal significa que o produtor e o consumidor não precisam estar ativos ao mesmo tempo. É a principal coisa que se compra com eventos, então vale ser exato sobre o preço.

O produtor deixa de saber o resultado

Quando o Orders chamava o Payments de forma síncrona, ficava sabendo se o pagamento funcionou antes de responder ao comprador. Com eventos, o Orders publica `OrderPlaced` e responde “recebido”. Se o pagamento tem sucesso é uma pergunta para depois, e a resposta chega, se chegar, como outro evento. O pedido passa a ter um estado intermediário, “realizado mas ainda não confirmado”, e o sistema precisa decidir o que o comprador vê enquanto ele dura e o que acontece se durar mais do que o previsto.

Ninguém é responsável pelo fluxo inteiro

Na versão síncrona, uma função continha a sequência, e dava para ler o processo de negócio em um único lugar. Na versão orientada a eventos, o processo é a soma de reações espalhadas por serviços. O Payments reage a `OrderPlaced`, o Orders reage a `PaymentAuthorized` e `StockReserved`, o Shipping reage a `OrderConfirmed`. O fluxo existe apenas como propriedade emergente, e ninguém consegue apontar onde ele está definido.

Isso não é problema quando tudo funciona e é caro quando algo não funciona, porque “onde este pedido ficou preso?” não tem um único lugar para olhar. O Capítulo 10 volta a isso com as sagas, e o Capítulo 13 trata do que você precisa observar para conseguir responder a essa pergunta.

Falhas deixam de ser exceções

Em uma cadeia de chamadas, um erro se propaga pela pilha até que algo o trate. Em um fluxo de mensagens não há pilha. Um consumidor que falha pode dar o ack na mensagem e perdê-la, recusá-la e tê-la reentregue, ou estacioná-la para uma pessoa resolver. Cada uma é uma decisão de política com consequências para serviços que nunca veem o erro. Na minha opinião, o tratamento de erros em um sistema síncrono é uma questão de fluxo de controle, e em um sistema assíncrono é uma questão de design, porque o tratamento fica espalhado por tempo, processos e, às vezes, equipes.

O tempo vira uma variável

Entre “publicado” e “processado” existe uma lacuna sem limite superior, a menos que você lhe dê um. Durante a lacuna, um produto pode esgotar, um comprador pode cancelar, um preço pode mudar. Um consumidor precisa decidir se um evento que era verdadeiro quando foi publicado ainda faz sentido quando é tratado.

Nota: É por isso que os eventos são nomeados no passado.

`OrderPlaced` é um fato sobre um momento que já passou. Um consumidor que o trata como uma instrução para agir agora mesmo supôs, sem perceber, que nenhum tempo se passou. Um comando, como `CapturePayment`, pede a um serviço que faça algo e pode ser recusado; um evento informa algo que já aconteceu. O Capítulo 11 desenvolve a distinção.

O que “com o tempo” realmente significa

A maioria das explicações de **consistência eventual** cabe em uma frase: se nenhuma atualização nova for feita, todas as réplicas convergem para o mesmo valor. Isso é correto e quase inútil para quem está construindo um fluxo de checkout, porque deixa de fora quanto tempo dura o “com o tempo” e o que fazer enquanto se espera.

“Com o tempo” não é uma duração

Um sistema que converge em 200 milissegundos e um sistema que converge depois de um lote noturno têm, ambos, consistência eventual. Para o comprador que olha a página de confirmação, são produtos diferentes. Minha recomendação é pedir um número para cada salto que importa, mesmo que aproximado. Digamos que o `Orders` deva mostrar um pedido confirmado poucos segundos depois da captura e que o `Notifications` possa levar um minuto. Esses valores são ilustrações e os seus serão diferentes, mas transformam o “com o tempo” em um requisito que pode ser testado e monitorado com alertas. Um atraso sem limite não é uma escolha de design. É a ausência de uma.

“Com o tempo” não é uma garantia

A convergência só acontece se toda mensagem que importa for, no fim, entregue, se todo consumidor a processar e se nenhum passo a descartar em silêncio. Se uma mensagem se perde entre um commit no banco de dados e uma publicação no broker, as visões não convergem. Ficam diferentes até que uma pessoa ou um job de reparo perceba. “Com o tempo” é uma promessa condicional, e a condição é que o sistema tenha mecanismos que tornem a perda detectável e reparável. Escolher um broker não os fornece. As partes II e III deste livro evitam a maior parte das perdas, e o Capítulo 13 detecta e repara o resto.

O que fazer enquanto se espera

Durante a janela de inconsistência, outros leitores veem um estado intermediário. Considere este instantâneo de um pedido, um momento depois de o comprador ser cobrado:

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

Cada serviço está certo sobre os próprios dados. A combinação parece inconsistente e é normal por alguns segundos. Ela só se torna um problema quando continua assim. A pergunta operacional útil não é “os serviços estão consistentes agora?”, mas “há quanto tempo este pedido está neste estado, e isso é mais do que permitimos?”

Uma definição prática de confiabilidade

“Confiável” é usado de forma tão frouxa que dois engenheiros podem concordar que um sistema deve ser confiável e querer dizer coisas opostas. Quero uma definição que possa ser verificada. Para sistemas orientados a eventos, uso quatro propriedades. Elas dizem o que está sendo protegido; as três perguntas acima são a forma de verificar uma junção em relação a elas.

Nenhuma perda. Todo evento que representa um fato de negócio chega a todo consumidor que precisa dele. A perda é a falha mais danosa porque é silenciosa: o sistema continua rodando, e o único sintoma é uma ausência que ninguém percebe até que um comprador reclame. Nenhuma perda é uma propriedade do caminho inteiro, não do broker. Um broker durável não ajuda se o produtor nunca conseguiu armazenar a mensagem, ou se o consumidor deu o ack antes de terminar o trabalho. O Capítulo 2 examina o que os brokers prometem, e o Capítulo 4 e o Capítulo 5 tratam do lado do produtor.

Nenhuma duplicação danosa. Algumas mensagens serão entregues mais de uma vez. Isso não é um defeito de um broker em particular, mas o preço de não perder mensagens: quando quem envia não consegue saber se uma mensagem chegou, o movimento seguro é enviá-la de novo. Um sistema confiável não tenta impedir duplicados no transporte. Ele os torna inofensivos. A palavra “danosa” importa. Dois e-mails idênticos são um incômodo, enquanto uma segunda captura do mesmo pagamento é um incidente financeiro. O Capítulo 7 trata da idempotência, a principal ferramenta.

Atraso limitado. Todo evento é processado dentro de um tempo que o negócio tolera, ou o sistema avisa em alto e bom som que isso não aconteceu. Uma mensagem entregue corretamente com três dias de atraso é, para muitos fins, uma mensagem perdida. Limitado não significa rápido. Significa que existe um limite, alguém o escolheu e uma violação gera um alerta. A confirmação de um pagamento pode precisar de segundos; um extrato semanal pode tolerar horas. O Capítulo 6 trata de retries e backoff, e o Capítulo 13 trata de atraso e alertas.

Explicabilidade. Para qualquer pedido, a qualquer momento, alguém consegue reconstruir o que aconteceu: quais eventos foram produzidos e consumidos, o que cada serviço decidiu e por quê. É a propriedade menos glamourosa e é ela que decide se as outras três podem ser mantidas. Um sistema que perde uma mensagem mas consegue mostrar exatamente onde é reparável. Um sistema que se comporta mal de um jeito que ninguém consegue rastrear não é. O Capítulo 13 trata dos identificadores, registros e comparações de que isso precisa.

As quatro puxam em direções opostas, e é por isso que uma definição com apenas uma delas é perigosa. Perseguir nenhuma perda com retries ilimitados ameaça o atraso limitado. Descartar mensagens lentas para limitar o atraso ameaça nenhuma perda. Recusar-se a fazer retries para evitar duplicados traz a perda de volta. Confiabilidade é um equilíbrio deliberado, escolhido por fluxo e registrado por escrito.

Por que o happy path falha em produção

Com a definição em mãos, volte ao primeiro mês da Parcelly e classifique as falhas por propriedade. Alguns mecanismos respondem pela maior parte do dano.

O que a Parcelly viu	Propriedade em risco	Causa subjacente	Capítulo
Pedido aceito, mas nada acontece depois dele	Nenhuma perda	Evento com commit no banco de dados mas nunca publicado	4
Pedido enviado duas vezes	Nenhuma duplicação danosa	Mensagem reentregue processada de novo	7
Uma mensagem ruim bloqueia uma partição	Atraso limitado	Uma mensagem falha todas as vezes e sofre retries para sempre	8
Pedido “confirmado” antes de “realizado”	Explicabilidade	Eventos relacionados processados fora de ordem	9
O ledger discorda dos pagamentos	Explicabilidade	Sem identificador comum, sem como comparar as visões	13

A quarta linha é a incômoda. A reordenação não duplica nada. Ela danifica o estado, por exemplo quando uma atualização antiga sobrescreve uma mais nova, e nenhuma das quatro propriedades a nomeia diretamente. Classifico essa propriedade em explicabilidade porque o sintoma é um histórico que ninguém consegue explicar, e o Capítulo 9 trata do dano ao estado em si.

A primeira linha merece um passo a passo, porque é a falha que mais costuma pôr fim à lua de mel.

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

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

Este bloco não é seguro, e é mostrado como a linha de base quebrada. O commit no banco de dados e a publicação no broker são dois sistemas sem transação compartilhada, o que o livro chama de **dual write**. Se o processo para entre as duas linhas, o pedido existe e ninguém fica sabendo. Se as linhas forem invertidas, um evento pode ser publicado para um pedido que nunca sofreu commit. Nenhuma das duas ordens está correta, e ambas passam em todos os testes que não matam o processo exatamente no pior momento.

Esse não é um bug raro. É o comportamento padrão de um design que nunca teve um modelo de falhas. O Capítulo 3 dá o modelo, e o Capítulo 4 dá a correção.

Um mapa do livro

O restante do livro responde às perguntas que a definição levanta, na ordem em que uma equipe costuma encontrá-las.

Pergunta que você vai fazer	Capítulo
O que o broker realmente promete, e o que não promete?	2
O que pode quebrar, e o que vou ver quando quebrar?	3
Como altero meu banco de dados e publico um evento sem perder nenhum?	4
Como os eventos publicados chegam do banco de dados ao broker?	5
Como faço retries sem piorar as coisas?	6
Como torno inofensivo o processamento repetido?	7
O que faço com uma mensagem que nunca pode ter sucesso?	8
Quando a ordem importa, e como a mantenho?	9
Como coordeno trabalho entre serviços e o desfaço?	10

Pergunta que você vai fazer	Capítulo
O que um evento deve conter e como deve se chamar?	11
Como altero um evento sem quebrar os consumidores?	12
Como vejo, meço e reparo o que está acontecendo?	13
Como me recupero de dados perdidos, de uma região perdida ou de uma troca de broker?	14
Como testo os caminhos de falha?	15
O que verifico antes de entrar no ar?	16

A Parte I, que este capítulo abre, constrói o vocabulário e o modelo de falhas. A Parte II trata de publicar com confiabilidade, a Parte III de consumir com confiabilidade, a Parte IV de fluxos de trabalho e contratos, e a Parte V de operar e comprovar o sistema. Leia as partes em ordem numa primeira passada. Se você está lutando contra um problema específico hoje, a tabela é um índice.

Trade-offs e custos

Migrar para eventos não é um erro, e este capítulo não é um argumento contra isso. É um argumento a favor de fazer a migração de olhos abertos, porque o custo é real e é pago em lugares específicos.

Você paga em complexidade de estado. O fluxo síncrono tinha dois resultados: concluído ou falho. O fluxo orientado a eventos tem muitos estados intermediários, e cada um precisa ser nomeado, mostrado aos usuários, monitorado e, no fim, resolvido. Alguém precisa ser responsável por essa máquina de estados.

Você paga em habilidade operacional. Depurar uma cadeia de chamadas exige um stack trace. Depurar um fluxo de mensagens exige correlacionar ids, dashboards de atraso e a capacidade de raciocinar sobre timing. Uma equipe sem isso descobre que o tempo para entender um incidente cresce mesmo que os incidentes fiquem menos frequentes. O custo recai sobre quem está de on-call.

Você paga em esforço de teste. O happy path dá pouco trabalho para testar. Os caminhos de falha exigem mensagens duplicadas, mensagens reordenadas e reinícios em momentos inoportunos. O Capítulo 15 mostra como fazer isso sem uma suíte lenta e instável, mas o esforço não é zero.

Você paga em latência por coisas simples. Uma operação que era uma chamada de função vira vários saltos por um broker. Quando quem chama precisa da resposta agora, como ao validar um código de desconto antes de mostrar um preço, uma chamada síncrona é a melhor ferramenta.

Não usaria eventos quando dois passos precisam ter sucesso ou falhar juntos e vivem no mesmo serviço e no mesmo banco de dados. Uma transação local é mais simples, mais forte e mais barata do que qualquer padrão deste livro. Também hesitaria quando a equipe não tem capacidade de operar um broker e o monitoramento em torno dele, ou quando o sistema é pequeno o bastante para que uma única unidade implantável com um banco de dados resolva. Muitos bons sistemas misturam os dois estilos: uma chamada síncrona para uma pergunta que precisa de resposta, um evento para um fato que outros podem querer saber. Escolheria por interação, não por sistema.

A definição de quatro propriedades também tem um limite. É uma lente, não uma especificação. Ela não diz qual é o limite de atraso certo para o seu negócio nem quais duplicados são danosos para um dado consumidor. Não conheço os seus números, e essas respostas raramente vêm só da engenharia. O que posso oferecer é o hábito de pedi-los.

Erros comuns

- **Testar apenas o happy path.** É tentador porque o pedido da funcionalidade o descrevia e ele passa rápido. Trate cada seta do diagrama de fluxo como uma pergunta sobre falha e escreva ao menos um teste por resposta (Capítulo 15).
- **Tratar o broker como a camada de confiabilidade.** Os brokers anunciam durabilidade, o que torna isso tentador. Um broker cobre o trecho entre a aceitação e a entrega, e as falhas que mais machucam ficam antes e depois dele. Liste o que o broker promete e depois liste as lacunas em torno da promessa (Capítulo 2 e Capítulo 3).

- **Usar “com o tempo” como design.** Soa como uma decisão e evita a pergunta mais difícil de quanto tempo. Atribua um limite e um alerta a cada salto assíncrono que importa e diga o que o usuário vê enquanto espera.
- **Supor que duplicados são raros demais para tratar.** Em um ambiente de teste tranquilo eles nunca aparecem. Projete os consumidores supondo que toda mensagem pode chegar mais de uma vez (Capítulo 7).
- **Fazer o commit no banco de dados e publicar no broker como dois passos simples.** Lê-se com naturalidade e funciona em toda demonstração. A correção é o transactional outbox do Capítulo 4.
- **Definir confiabilidade como uptime.** O uptime é barato de medir, mas um serviço pode ficar no ar o mês inteiro e ainda assim perder mensagens ou processá-las com atraso. Meça diretamente as quatro propriedades: o que foi perdido, o que foi duplicado, quão atrasadas ficaram as coisas e se você conseguiria explicá-las.

Pontos-chave

- O happy path é uma história em que todo passo dá certo, e um sistema testado apenas contra ele foi mostrado funcionando no único caso que raramente se sustenta por muito tempo.
- Passar de chamadas síncronas para eventos não remove o acoplamento; muda-o de tempo e disponibilidade para contratos, comportamento do broker e suposições de timing não escritas.
- As falhas orientadas a eventos são silenciosas e se espalham por serviços, então um design sem modelo de falhas tende a descobrir suas junções pela produção.
- “Com o tempo” é uma promessa condicional sem duração associada, então todo salto assíncrono que importa precisa de um limite, de um alerta e de um estado definido, visível ao usuário, para o período de espera.
- Um sistema confiável tem nenhuma perda, nenhuma duplicação danosa, atraso limitado e explicabilidade, e, como essas propriedades se contrapõem, cada fluxo precisa de um equilíbrio deliberado e registrado por escrito.

- As junções entre componentes, como um commit no banco de dados seguido de uma publicação no broker, causam mais dano do que os próprios componentes, e a maior parte deste livro trata dessas junções, examinadas com três perguntas: o que pode falhar, como você observaria isso e qual garantia sobrevive.
- Eventos nem sempre são a ferramenta certa: quando os passos podem compartilhar uma transação local, ou quando quem chama precisa de uma resposta imediata, uma chamada síncrona é mais simples e mais forte.

O próximo capítulo examina o próprio broker. Antes de projetar em torno de suas lacunas, precisamos saber exatamente o que uma fila ou um log promete quando diz que uma mensagem foi entregue, e o que ele deixa discretamente por sua conta.

Capítulo 2 — Semântica de mensageria

Imagine uma sexta-feira à noite na Parcelly, em plena promoção. O serviço Notifications envia há meses e-mails de “seu pedido foi confirmado” sem nenhuma reclamação. Conforme o backlog cresce, o autoscaling adiciona consumidores e, poucos minutos depois, remove alguns. No fim da noite, o suporte tem dois tickets. Um comprador recebeu o mesmo e-mail de confirmação três vezes. Outro não recebeu nenhum, embora o pedido esteja confirmado.

A equipe abre o dashboard do broker e descobre que nada está quebrado. Todo offset teve o commit feito. Todo consumidor está saudável. Os gráficos estão planos. Mesmo assim, o mesmo sistema produziu um duplicado para um comprador e um silêncio para outro, e os dois resultados são coerentes com a documentação.

Alguém diz que o broker prometeu “entrega at-least-once”, e outra pessoa diz que supunha que o broker cuidava disso. As duas afirmações são verdadeiras, e nenhuma explica o que aconteceu com os dois compradores. Suponha que o Notifications leia um topic baseado em log como um único grupo de consumidores, com o cliente fazendo commit dos offsets automaticamente, e que um rebalanceamento tenha ocorrido a cada mudança de tamanho do grupo. Esses dois fatos podem produzir os dois tickets, e o capítulo mostra como.

Para explicar isso, a equipe precisa de um quadro preciso do que um broker promete, do que não promete e de quais partes da vida da mensagem pertencem à aplicação. O que uma mensagem realmente atravessa entre um produtor e um efeito colateral concluído, e onde cada garantia começa e termina?

O problema e por que ele é difícil

A maioria dos engenheiros conhece um broker pela descrição mais simples: você publica uma mensagem, e um consumidor a recebe. Essa descrição esconde três transferências distintas, e toda pergunta de confiabilidade deste livro mora em uma delas.

A primeira transferência vai do produtor ao broker. A segunda, do broker ao consumidor. A terceira, do consumidor ao efeito: a escrita no banco de dados, o e-mail, a chamada a uma transportadora. Um broker tem muito a dizer sobre as duas primeiras e quase nada sobre a terceira, porque a terceira acontece em código e em sistemas que o broker não enxerga.

A dificuldade é que cada transferência termina com um acknowledgment, e um acknowledgment é uma declaração sobre o passado feita por uma parte a outra através de uma rede. Ele diz “recebi isto” ou “terminei isto”. Não diz “e nada mais pode dar errado”. Quando um acknowledgment se perde, quem enviou não consegue saber se o trabalho aconteceu. Essa incerteza é a raiz das três semânticas de entrega, e voltaremos a ela depois de termos o vocabulário.

Uma segunda dificuldade é que a palavra “broker” abrange duas famílias que se comportam de modo diferente, de maneiras que importam. Uma **fila** é uma estrutura do broker em que cada mensagem é entregue a um consumidor e recebe um ack individualmente. Um **log de eventos** é uma sequência durável, somente de acréscimo, de eventos, em que os consumidores leem no próprio ritmo e registram uma posição. O que é verdadeiro para uma família costuma ser falso para a outra. O livro nomeia a família sempre que a distinção importa, e este capítulo é onde a distinção é construída.

Uma terceira dificuldade é que as garantias na documentação de um produto têm escopo. “Entregue pelo menos uma vez” é uma afirmação sobre a fronteira do próprio broker. Ela não cobre o que o consumidor faz depois de receber a mensagem, e depende de configurações que o produtor escolheu ao publicar. Ler a garantia sem o seu escopo é como uma equipe acaba vivendo a sexta-feira à noite descrita acima.

Filas e logs

O modelo de fila

Em uma fila, o broker acompanha o estado de cada mensagem. Uma mensagem está esperando, entregue a um consumidor, ou com ack dado e removida. Vários consumidores podem ler da mesma fila; esse arranjo se chama **consumidores concorrentes**, porque cada mensagem é entregue a um deles por vez e eles competem pela próxima.

Quando um consumidor dá ack em uma mensagem, o broker a apaga. A retenção é, portanto, curta por projeto: uma mensagem vive até ser processada ou até expirar. Não há como pedir à fila uma mensagem que já recebeu ack, porque a fila não a guarda.

A consequência é que uma fila se encaixa bem em trabalho que deve ser feito uma vez e esquecido. Um job da Parcelly como “gerar o PDF desta fatura” é um candidato natural: faz-se o trabalho, e depois a mensagem não tem mais utilidade. O custo é que o histórico não está disponível. Se um bug envia o texto errado durante uma hora, as mensagens que carregavam os fatos desapareceram, e o reparo precisa começar de outro lugar.

O modelo de log

Em um log de eventos, o broker não acompanha um estado por mensagem. Ele armazena as mensagens em ordem, em partições, e as mantém segundo uma política de retenção independente de quem já as leu. Uma **partição** é uma subsequência ordenada de um topic, e um **topic** é um stream nomeado de mensagens. Cada mensagem tem um **offset**, sua posição na partição.

Os consumidores registram até onde leram. Essa posição também se chama offset, e quem a registra é o consumidor ou o broker em nome dele. Um **grupo de consumidores** é um conjunto de consumidores que dividem o trabalho de um topic: cada partição é atribuída a um consumidor do grupo por vez. Grupos diferentes leem o mesmo topic de forma independente, cada um com os seus offsets.

A propriedade definidora do log é que ler não consome. Dez grupos podem ler o mesmo topic, e um grupo novo pode começar do início. **Replay**, a releitura deliberada do histórico, é o termo que este livro reserva para

esse ato, e ele só é possível porque o log mantém os dados. O Ledger da Parcelly é um bom exemplo de consumidor que se beneficia: se um bug é encontrado na forma como os lançamentos são montados, o serviço Ledger pode ser corrigido e apontado de volta para eventos antigos.

O custo é que o log empurra a contabilidade para o lado do consumidor. No modelo clássico de log, o broker não consegue reentregar uma mensagem que falhou enquanto avança para a próxima, porque não acompanha as mensagens individualmente. Um consumidor que não consegue processar a mensagem no offset 500 precisa decidir o que fazer antes de poder avançar com segurança para o 501.

Escolhendo entre elas

Aspecto	Família de filas	Família de logs
Unidade de acompanhamento	Cada mensagem	Uma posição por partição
Retenção	Até receber ack ou expirar	Por tempo ou tamanho, independente dos leitores
Vários leitores independentes	Exige uma cópia da mensagem por leitor	Natural, uma posição por grupo
Replay do histórico	Indisponível	Disponível dentro da retenção
Uma mensagem com falha	Reentregue isoladamente, as demais continuam	Bloqueia a partição, a menos que seja tratada
Ordenação	FIFO no armazenamento, na melhor das hipóteses; perda entre consumidores concorrentes e na reentrega	Forte dentro de uma partição

Não escolheria uma família pela popularidade da ferramenta. Começaria por três perguntas: alguém precisa reler esse histórico, vários consumidores precisam dos mesmos fatos, e a ordenação importa para o mesmo agregado? Um sim a qualquer uma delas aponta para um log. Uma fila de trabalho pura, sem necessidade de histórico e com necessidade de retries independentes de mensagens individuais, aponta para uma fila.

Muitas equipes acabam operando as duas. Eventos a partir dos quais outros serviços constroem estado vão para um log, e tarefas que devem acontecer uma única vez vão para uma fila. Esse é um design legítimo, e vem com o custo de operar e entender dois conjuntos de semântica. O Capítulo 9 retoma o que ordenação significa em cada família.

Push, pull e o fluxo da entrega

Os brokers também diferem em quem inicia a entrega. No modelo **push**, o broker envia as mensagens aos consumidores à medida que ficam disponíveis. No modelo **pull**, o consumidor pede as mensagens quando está pronto.

Os brokers baseados em filas costumam fazer push (o RabbitMQ entrega aos consumidores, enquanto os consumidores do Amazon SQS fazem polling, de modo que até isso varia dentro da família), com um limite de quantas mensagens sem ack um consumidor pode manter ao mesmo tempo. Esse limite costuma ser chamado de prefetch count ou de janela. Os brokers baseados em logs costumam fazer pull: o consumidor pede o próximo lote a partir do seu offset atual. Os dois modelos podem ser encontrados em qualquer das famílias, então trataria isso como uma tendência e não como uma lei.

A diferença importa sobretudo para o que acontece quando um consumidor é lento. No modelo push sem limite, o broker continua enviando, a memória do consumidor se enche de mensagens em que ele ainda não começou a trabalhar, e essas mensagens ficam em um estado em que não estão disponíveis para outro consumidor nem concluídas. No modelo pull, o consumidor controla o próprio ritmo, e um consumidor lento aparece como atraso do consumidor crescente, não como um processo sobrecarregado. O **atraso do consumidor** é o quanto um consumidor está atrás da ponta de uma partição ou fila, em mensagens ou em tempo.

A regra prática é que um consumidor nunca deve manter mais trabalho do que consegue concluir antes que sua reivindicação sobre o trabalho expire. O Capítulo 3 trata consumidores lentos e backpressure como falhas por si só, e o Capítulo 6 cobre como limitar a carga. Por ora, lembre-se de que “entregue” e “em processamento” são estados diferentes, e o intervalo entre eles é onde o tempo passa.

Acknowledgment, reentrega e offsets

Acknowledgment em uma fila

Um **acknowledgment**, em geral abreviado como **ack**, é a declaração do consumidor de que terminou com uma mensagem e o broker pode esquecê-la. Um **nack**, um acknowledgment negativo, diz que o consumidor não conseguiu processar a mensagem. Dependendo do broker e das configurações, um nack devolve a mensagem à fila, a descarta ou a envia para uma dead-letter queue (DLQ).

A pergunta de design central é quando fazer o ack. Há apenas duas posições razoáveis.

- **Ack antes de processar.** O broker esquece a mensagem imediatamente. Se o consumidor cai no meio do trabalho, a mensagem se perde. Isso produz comportamento at-most-once.
- **Ack depois de processar.** O broker mantém a mensagem até o trabalho terminar. Se o consumidor cai depois de terminar, mas antes de dar ack, a mensagem volta e o trabalho é feito duas vezes. Isso produz comportamento at-least-once.

Não existe posição que dê as duas propriedades, porque o acknowledgment e o trabalho são duas ações separadas em dois sistemas separados, e uma queda pode ocorrer entre elas.

Visibility timeout e reentrega

Como uma fila decide que um consumidor caiu? Ela não consegue observar isso diretamente. Alguns brokers monitoram a conexão e tratam uma conexão fechada como falha. Outros usam um **visibility timeout**: quando uma mensagem é entregue, ela fica invisível para os outros consumidores por um tempo determinado. Se o consumidor dá ack antes de o tempo acabar, a mensagem é apagada. Se não, a mensagem volta a ficar visível e é entregue a outro consumidor. Essa segunda entrega é uma **reentrega**.

O timeout é um palpite sobre quanto tempo o processamento leva, e errar o palpite para qualquer lado tem um custo. Se for curto demais, um consumidor lento, mas vivo, terá a mensagem entregue a um segundo consumidor enquanto ainda está trabalhando nela. Agora dois consumidores processam a mesma mensagem, e o ack do primeiro, quando enfim vier,

pode falhar, ser ignorado ou, dependendo do broker, ter sucesso e apagar a mensagem enquanto o segundo consumidor ainda trabalha nela. Nesse último caso, uma falha da segunda tentativa perde a mensagem. Se for longo demais, uma queda real deixa a mensagem escondida durante todo esse período, e o atraso aparece como latência para o comprador.

Digamos que o consumidor do Inventory normalmente conclua uma reserva em bem menos de um segundo, e alguém defina o timeout em 30 segundos. Esse número é ilustrativo. Um soluço do banco de dados que faça uma reserva levar 40 segundos agora produzirá uma segunda entrega do mesmo `OrderPlaced`, e o Inventory tentará reservar o estoque duas vezes. Nada está quebrado no broker. O timeout codificava uma suposição, e a suposição deixou de valer.

Alguns sistemas permitem que o consumidor estenda o timeout enquanto trabalha. Isso ajuda em jobs legitimamente longos, e acrescenta algo novo que pode dar errado: um consumidor que estende para sempre o timeout de uma mensagem que nunca vai terminar.

Offsets em um log

Em um log, não há ack por mensagem. O consumidor faz, em vez disso, um **commit de offset**: registra “processei tudo o que vem antes desta posição”. Depois de uma reinicialização ou de uma reatribuição, o próximo consumidor lê a partir do offset cujo commit foi feito por último.

As mesmas duas posições existem aqui. Se o commit do offset é feito antes de processar, uma queda perde as mensagens posteriores à última concluída, até a posição do commit. Se o commit é feito depois de processar, uma queda repete as mensagens entre o último commit e a queda. Consumidores reais fazem commit em lotes, então o número de mensagens repetidas após uma queda é um lote, não uma.

Muitas bibliotecas de cliente conseguem fazer o commit automaticamente. Isso é conveniente e é também a forma menos explícita de definir a sua semântica de entrega, porque, dependendo do cliente, o commit pode acontecer em um momento sem relação com o fim do seu processamento. Se o processamento é repassado a outra thread, o cliente pode fazer o commit

de um trabalho que nunca foi concluído. Na minha visão, um consumidor que importa deve fazer o commit de forma explícita, em um ponto que a equipe consiga apontar no código e sobre o qual consiga raciocinar.

Grupos de consumidores e rebalanceamento

Quando consumidores entram em um grupo de consumidores ou saem dele, o broker redistribui as partições entre eles. Isso se chama **rebalanceamento**. Durante um rebalanceamento, uma partição que estava sendo lida por um consumidor é reatribuída a outro, e o novo proprietário começa no último offset cujo commit foi feito.

Os grupos baseados em logs fazem o mesmo palpite que o visibility timeout faz em uma fila, sob outra forma. Um consumidor que não volta ao broker dentro de um intervalo configurado é presumido morto, suas partições são reatribuídas, e o trabalho dele ainda sem commit é lido de novo pelo novo proprietário. Um consumidor lento que ainda está vivo é a causa cotidiana dos duplicados de rebalanceamento.

É daqui que vêm os duplicados do cenário de abertura. Considere um consumidor que processou as mensagens 100 a 150, mas só fez o commit até a 120. Um deploy o reinicia, e a partição passa para um segundo consumidor, que lê a partir da 120. As mensagens 120 a 150 são processadas duas vezes, e nenhuma parte do sistema funcionou mal. O rebalanceamento é um evento normal que pode acontecer a cada deploy, scale-up e queda, de modo que a reentrega é uma ocorrência cotidiana, não um acidente raro.

O silêncio tem a causa espelhada. Se o cliente fez o commit de offsets de mensagens que o consumidor tinha apenas repassado a um emissor em segundo plano, e o processo foi interrompido antes de o emissor terminar, o próximo proprietário começa depois dessas mensagens. Elas nunca são lidas de novo, e nenhuma linha de log diz isso. O mesmo rebalanceamento pode, portanto, repetir a mensagem de um comprador e descartar a de outro.

Há um risco relacionado: por pouco tempo, o antigo proprietário pode ainda não saber que perdeu a partição e continuar trabalhando nela. Dois consumidores podem então processar a mesma partição ao mesmo tempo. Clientes bem projetados tentam evitar isso, mas não projetaria um consumidor que dependa de essa prevenção ser perfeita.

Semânticas de entrega

Com a mecânica estabelecida, as três semânticas de entrega são consequências do lugar onde você põe o acknowledgment.

At-most-once

At-most-once significa que cada mensagem é entregue zero ou uma vez. Obtém-se dando ack antes de processar ou não fazendo retry de uma publicação que falhou. Uma mensagem pode ser perdida, e nunca pode ser duplicada.

É uma escolha razoável quando perder uma mensagem custa menos que tratar duplicados. Um ping periódico de “localização atual” de uma transportadora da Parcelly é um caso: o ping seguinte torna irrelevante o que se perdeu. É a escolha errada para `PaymentCaptured`, em que uma mensagem perdida significa dinheiro movimentado sem que ninguém tenha registrado.

At-least-once

At-least-once significa que cada mensagem é entregue uma ou mais vezes. Obtém-se dando ack depois de processar e fazendo retry até que um acknowledgment chegue. Uma mensagem nunca se perde enquanto o broker a mantém, e pode ser entregue de novo.

Esta é a semântica sobre a qual o livro se constrói. É a única das três em que a falha é visível e recuperável: um duplicado é algo que você consegue detectar e ignorar, e uma mensagem perdida é algo que você pode não descobrir por dias. O preço é que todo consumidor precisa tolerar um **duplicado**, isto é, uma mensagem processada ou entregue mais de uma vez. O Capítulo 7 trata de como.

“Exactly-once”

A entrega é um ato físico através de uma rede não confiável, e o acknowledgment do receptor pode se perder como qualquer outra mensagem. Quem enviou e não recebeu um ack tem duas opções: enviar de novo, o que arrisca um duplicado, ou não enviar, o que arrisca uma perda. Nenhum protocolo elimina essa escolha só entre as duas partes.

Ainda assim, *exactly-once* é um resultado real e útil, e é um efeito, não uma garantia de entrega. Quando uma entrega *at-least-once* encontra um consumidor cujo processamento repetido tem o mesmo efeito que processar uma vez, o sistema combinado se comporta como se cada mensagem tivesse sido tratada exatamente uma vez. A receita é entrega *at-least-once* mais um **consumidor idempotente**.

Alguns brokers anunciam processamento *exactly-once*, e a alegação é legítima dentro de uma fronteira. Um broker baseado em logs pode tornar atômico um ciclo de ler, processar e escrever quando a entrada, a saída e o commit de offset estão todos dentro desse mesmo broker. Dentro dessa fronteira, uma queda desfaz tudo em conjunto. A garantia também exige que todo leitor a jusante leia apenas dados com commit feito, e depende de uma identidade estável do produtor para que uma instância antiga seja barrada após uma reinicialização. A fronteira termina onde começam os seus efeitos colaterais. No momento em que o consumidor escreve em um banco de dados relacional, chama um provedor de pagamentos ou envia um e-mail, a transação do broker deixa de cobrir o efeito, e o problema dos duplicados retorna.

Por isso, o padrão do livro continua sendo *at-least-once* mais consumidores idempotentes. Funciona com qualquer broker e qualquer efeito colateral, não depende de um recurso estar habilitado e sobrevive a uma migração para outro produto, tema que o Capítulo 14 cobre. Se você consegue manter um pipeline inteiro dentro da fronteira transacional de um único broker, o recurso pode reduzir o trabalho, e mesmo assim projetaria os consumidores para que não dependessem dele.

Nota: Os produtores também podem criar duplicados, antes mesmo de um consumidor entrar em cena. Um produtor que expira esperando o acknowledgment de uma publicação normalmente publica de novo, e o broker pode passar a ter duas cópias de um evento. Ambas as cópias carregam o mesmo `event_id`, desde que o produtor o tenha atribuído uma única vez, quando o evento foi criado, e não tenha gerado um novo para o retry. Alguns brokers eliminam duplicados dos retries do próprio produtor, por número de sequência dentro de uma sessão do produtor ou por um id de deduplicação dentro de uma janela de tempo. Isso cobre um retry dentro de uma conexão. Não cobre um processo que reinicia e publica de novo, como um relay de outbox, e é por isso que o consumidor ainda verifica o `event_id`. O Capítulo 4 cobre como publicar sem perder eventos, e o Capítulo 7 cobre o reconhecimento de duplicados.

O que o broker realmente garante

Tudo o que foi descrito acima trata do lado broker-consumidor. O lado produtor-broker tem os seus próprios limites, e eles dependem mais da configuração do que do nome do produto.

Acknowledgments do produtor

Quando um produtor publica, o broker pode dar ack em momentos diferentes, e a maneira útil de distinguir as configurações é pelo número de nós que guardam a mensagem quando o produtor ouve “pronto”. Sem acknowledgment, o produtor não fica sabendo nada sobre o armazenamento. Com acknowledgment pelo nó receptor, um nó aceitou a mensagem, tenha ou não feito o flush dela para o disco. Com acknowledgment após a replicação, a mensagem existe em um número exigido de nós. O flush para o disco é uma questão à parte, tratada mais adiante.

A escolha do produtor importa para o que uma queda pode causar. Com as duas primeiras configurações, um nó do broker que morre logo depois de dar ack pode levar a mensagem junto. Com o acknowledgment replicado, a resposta chega mais tarde e a janela de perda é menor. O preço é uma latência maior em toda publicação e a dependência de haver nós suficientes disponíveis para completar a escrita.

Os brokers que oferecem publisher confirms, na família de filas, e níveis de acknowledgment, na família de logs, descrevem a mesma ideia. Verifique o que o acknowledgment que você espera significa de fato na sua configuração. Na minha visão, essa é uma das páginas de documentação mais valiosas que uma equipe pode ler uma vez e registrar por escrito.

Replicação e fsync

A replicação protege contra a perda de um nó: a mensagem existe em vários. Não protege contra uma falha simultânea de todos, e sua garantia depende de quantas cópias precisavam existir antes de dizerem “pronto” ao produtor. Verifique tanto o nível de acknowledgment quanto o número mínimo de réplicas que ele exige, porque “todas as réplicas” pode significar uma só quando as outras ficaram para trás. Verifique também se um failover pode promover uma réplica que não estava atualizada, pois isso pode perder dados que o produtor foi informado de que estavam seguros.

Há um ponto relacionado, e frequentemente mal compreendido, sobre o disco. Escrever em um arquivo normalmente coloca os dados no cache do sistema operacional, e os dados chegam ao disco físico mais tarde, a menos que o programa force isso com um flush síncrono, comumente chamado de fsync. Forçar o flush a cada mensagem é lento, então muitos brokers não o fazem a cada escrita. Eles contam com a replicação para cobrir a lacuna e fazem o flush em intervalos programados. Se o acknowledgment de um determinado broker implica que os dados estão no disco depende da configuração dele, e não suporia que sim.

Não sei qual dessas configurações serve ao seu sistema, e isso depende do custo de perder um evento. Um `PaymentCaptured` perdido e um “página visualizada” perdido são problemas diferentes, e seria razoável configurá-los de modo diferente.

O limite da garantia

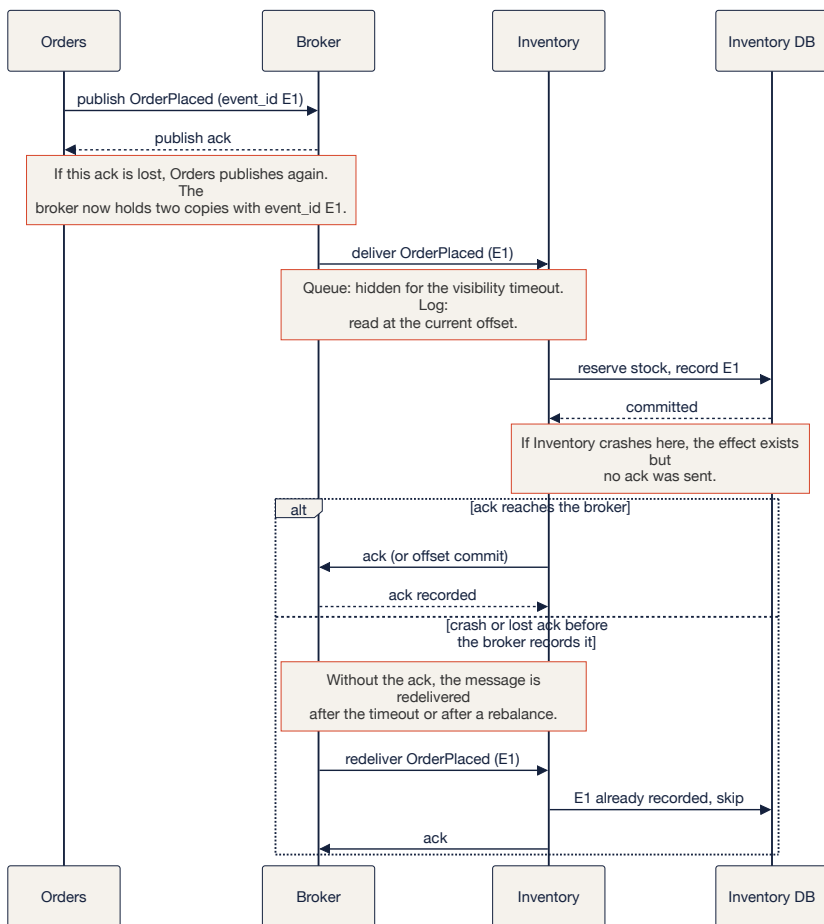
Junte tudo isso e a fronteira fica clara. Um broker pode prometer que uma mensagem que aceitou será mantida e oferecida aos consumidores, dentro dos seus limites de retenção ou expiração, e apenas tão duravelmente quanto as configurações de acknowledgment e replicação acima permitirem. Ele não pode prometer que uma mensagem chegue ao broker, para

começar, porque isso depende do que o produtor fez quando a sua publicação expirou. Também não pode prometer que o efeito de um consumidor tenha acontecido exatamente uma vez.

Duas lacunas ficam nas mãos da aplicação. A primeira está no lado do produtor: um pedido salvo no banco de dados do Orders e um evento `OrderPlaced` publicado no broker são duas escritas, e uma queda pode ocorrer entre elas. O Capítulo 1 chamou isso de dual write, e o Capítulo 4 o elimina. A segunda está no lado do consumidor, entre o efeito e o acknowledgment. O Capítulo 7 fecha essa lacuna. Vistas como as três transferências que abriram este capítulo, o broker declara a semântica de falhas das duas primeiras, e as duas lacunas são o que resta em suas mãos: o dual write antes da primeira transferência e o intervalo entre o efeito e o acknowledgment na terceira.

A vida de uma mensagem

O diagrama abaixo acompanha uma mensagem `OrderPlaced` do serviço Orders até um efeito concluído no Inventory. Procure os dois pontos em que uma falha deixa quem enviou sem resposta, marcados nas notas: o acknowledgment de publicação perdido e o acknowledgment do consumidor perdido. Tudo o que o capítulo descreveu acontece dentro desta única sequência.



O ramo `else` do diagrama é o ponto do capítulo. A reentrega não é um erro e nenhum componente se comportou mal. A única coisa que impede o estoque de ser reservado duas vezes é que o **Inventory** registrou `E1` na mesma transação da reserva e o verificou de novo na segunda entrega.

Veja como fica em pseudocódigo um consumidor com essa propriedade. Ele usa um **inbox**, uma tabela de valores de `event_id` processados no próprio banco de dados do consumidor, e coloca o acknowledgment depois do commit.

```

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

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

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

```

Este é o design que torna o at-least-once seguro, e ele é apresentado aqui apenas para mostrar onde o acknowledgment pertence. O Capítulo 7 o desenvolve por completo, incluindo os casos em que a verificação do inbox sozinha não basta.

Trade-offs e custos

Escolher a semântica é escolher qual falha você prefere gerenciar. Nenhuma das opções elimina a falha, e cada uma a leva para outro lugar.

O at-least-once custa trabalho de idempotência em todo consumidor.

Cada consumidor precisa de um modo de reconhecer trabalho repetido, em geral uma tabela de inbox no próprio banco de dados. Isso é armazenamento, uma restrição de unicidade no caminho crítico e uma política de limpeza. Para uma equipe com uma dúzia de consumidores, é um padrão a aplicar uma dúzia de vezes. Quando o efeito de um consumidor é naturalmente idempotente, como definir um status com um valor fixo, o inbox pode ser desnecessário, e mesmo assim registraria por escrito o porquê.

O at-most-once custa completude. É simples e rápido, e só é aceitável quando alguém decidiu de antemão que uma mensagem perdida é mais barata que um duplicado. Essa decisão deve ser explícita e registrada. A

forma habitual do erro é uma equipe que o escolheu por padrão, por meio de configurações de auto-acknowledgment, sem perceber.

O processamento transacional dentro de um broker custa portabilidade. Ele pode reduzir o tratamento de duplicados dentro da sua fronteira, e amarra o design aos recursos de um único produto, à permanência de todos os participantes dentro dessa fronteira e ao esforço operacional de mantê-lo. Quando você cruza para um banco de dados relacional ou uma API externa, precisa de idempotência de qualquer jeito. Não rejeitaria o recurso, e não construiria a correção apenas sobre ele.

Acknowledgments fortes do produtor custam latência e disponibilidade. Esperar por várias réplicas torna cada publicação mais lenta e faz a escrita falhar quando poucos nós estão acessíveis. Para eventos que não podem ser perdidos, esse trade-off costuma ser o certo. Para telemetria de alto volume e baixo valor, pode não ser.

Os logs custam contabilidade no lado do consumidor, e as filas custam histórico. Se você escolhe um log, fica com a política de commit de offset e com o tratamento de uma mensagem que bloqueia uma partição. Se escolhe uma fila, abre mão do replay. Quando você precisa de retries independentes de mensagens individuais e nenhum histórico, um log pode ser mais maquinário do que o trabalho merece.

Quando não aplicar o conselho deste capítulo: um sistema com um único produtor, um único consumidor e tolerância a perdas ocasionais, como um feed interno de métricas, não precisa de inbox nem de replicação forte. O custo das garantias deve corresponder ao custo da falha. O erro não é escolher uma garantia mais fraca; o erro é não saber que você a escolheu.

Erros comuns

- **Tratar “entregue” como “processado”.** É tentador porque o dashboard mostra entregas e acknowledgments, e esses são os números que parecem saudáveis. Entregue significa que o broker repassou a mensagem. Processado significa que o commit do efeito foi feito. Acompanhe os dois e faça seu acknowledgment vir depois do efeito.
- **Dar ack antes de o trabalho estar feito.** É tentador porque evita a re-entrega e deixa a fila com aparência limpa. Converte toda queda de

consumidor em uma mensagem perdida. Dê ack depois que o commit do efeito for feito e trate os duplicados que resultarem disso.

- **Deixar o auto-commit de offset ligado por padrão.** É tentador porque funciona em uma demo e não exige código. Dependendo do cliente, ele pode fazer o commit em um momento sem relação com o seu processamento, de modo que uma queda pode perder ou repetir uma fatia arbitrária. Faça o commit de forma explícita, depois do efeito.
- **Definir o visibility timeout no olho.** É tentador porque o padrão costuma parecer bom. Um timeout mais curto que uma execução lenta, mas saudável, causa processamento duplicado concorrente. Baseie-o no tempo de processamento legítimo mais lento que você consiga observar, mais uma margem, e estenda-o de forma deliberada para jobs longos.
- **Acreditar que o recurso exactly-once de um broker cobre o seu banco de dados.** É tentador porque a documentação usa a expressão. A garantia tem o escopo da fronteira do próprio broker. Tudo o que fica fora dela, como uma linha no seu banco de dados ou um e-mail, precisa de idempotência.
- **Supor que mensagem publicada é mensagem armazenada.** É tentador porque a chamada de publicação retornou. Dependendo do nível de acknowledgment, a mensagem pode existir apenas na memória de um nó. Saiba o que o seu produtor espera e registre por escrito.
- **Supor que o rebalanceamento é raro.** É tentador porque acontece em silêncio. Deploys, eventos de escala e quedas podem todos disparar um, dependendo do protocolo do grupo e das configurações de membros, e cada rebalanceamento pode repetir um lote de mensagens. Projete como se a reentrega acontecesse todos os dias, porque o custo de errar é um efeito duplicado.

Pontos-chave

- Uma mensagem passa por três transferências, do produtor ao broker, do broker ao consumidor e do consumidor ao efeito, e as garantias de um broker cobrem apenas as duas primeiras.
- As filas acompanham cada mensagem e a apagam no acknowledgment, enquanto os logs retêm as mensagens independentemente dos

leitores e deixam cada grupo de consumidores manter o seu próprio offset, o que torna o replay possível.

- O lugar onde você põe o acknowledgment ou o commit de offset decide a semântica: antes de processar dá at-most-once, depois de processar dá at-least-once, e nenhuma posição evita ao mesmo tempo perdas e duplicados.
- A reentrega é rotina, não exceção, porque os visibility timeouts expiram, os consumidores caem e os rebalanceamentos (que os deploys costumam causar) repetem lotes.
- Exactly-once é um efeito construído a partir de entrega at-least-once e um consumidor idempotente, e o processamento exactly-once de um broker termina na fronteira desse broker.
- O acknowledgment de um produtor é tão forte quanto as configurações de replicação e flush por trás dele, então a equipe deve saber e registrar o que “publicado” significa para cada evento importante.
- Cada opção semântica move a falha em vez de eliminá-la, e a escolha certa é aquela cuja falha restante você decidiu que tem condições de gerenciar.

A semântica de mensageria diz o que o broker fará e não fará em um dia normal. O Capítulo 3 se volta para os dias anormais e constrói um modelo prático do que de fato quebra: partições de rede, quedas, consumidores lentos, relógios e pessoas.

Capítulo 3 — Um modelo de falhas prático

Imagine o Payments da Parcelly no meio de uma captura. O Orders enviou um comando `CapturePayment` ao Payments, e o Payments chamou o provedor de pagamentos externo para capturar o dinheiro. A conexão com o provedor fica muda. Trinta segundos depois, o cliente HTTP do Payments desiste e levanta um timeout.

O Payments agora precisa responder a uma pergunta que não consegue responder. O provedor capturou o pagamento antes de a conexão morrer, ou nunca recebeu a requisição? As duas histórias produzem exatamente a mesma observação: silêncio, depois um timeout. Se o Payments faz um `retry` e a primeira captura tinha tido sucesso, o comprador é cobrado duas vezes. Se o Payments informa uma falha e a primeira captura tinha tido sucesso, o Orders cancela um pedido que já está pago.

Nada nesta história é exótico. Não há bug, nem relatório de indisponibilidade, nem engenheiro que tenha cometido um erro. Uma rede fez o que as redes fazem, e o sistema não tinha como saber em qual de dois mundos estava vivendo.

A maior parte do que dá errado em um sistema orientado a eventos tem esse formato: um passo aconteceu, ou não, e a evidência está faltando. Este capítulo nomeia as maneiras como essa evidência desaparece, para que você possa projetar para cada uma delas de propósito. A pergunta que ele responde é esta: o que exatamente pode quebrar em um sistema como o da Parcelly, o que você verá quando isso acontecer, e onde, neste livro, cada falha é tratada?

O problema e por que ele é difícil

Um modelo de falhas é uma lista escrita das coisas que você supõe que podem dar errado e das coisas que supõe que não podem. Todo sistema tem um, quer alguém o tenha escrito ou não. A versão não escrita costuma conter duas entradas: “o serviço cai” e “a rede está lenta”. Elas são reais, mas são as falhas que se anunciam sozinhas.

O caso simples é uma falha que você consegue detectar, como uma conexão recusada ou um erro de banco de dados. Cada uma diz que o passo não foi concluído.

O caso difícil é a falha que o deixa incerto. Um timeout não diz “a operação falhou”. Diz “parei de esperar”. A operação pode ter falhado, pode ter tido sucesso, ou pode ainda estar em execução e prestes a ter sucesso depois que você desistiu. Basta um timeout e um efeito colateral do outro lado para sentir isso.

É por isso que a abordagem óbvia, escrever código para o happy path e acrescentar um `try` e um `retry` em volta de cada chamada, não se sustenta. Fazer um `retry` é uma decisão sobre o futuro, tomada sem conhecer o passado. Se a tentativa anterior teve sucesso, o `retry` é um duplicado. Se você se recusa a fazer o `retry`, uma mensagem pode se perder. Nenhuma das escolhas está errada em si; cada uma está errada em algum mundo que você não conseguia distinguir daquele em que estava.

Há uma segunda razão. As falhas se combinam: um `deploy` reinicia um consumidor enquanto um `broker` faz `failover` e um produtor faz `retry`. Listá-las uma a uma continua sendo o começo certo, mas a produção as entrega juntas.

Assim, o trabalho deste capítulo é a classificação. Para cada falha, farei as três perguntas do Capítulo 1: o que pode falhar nesta junção, como você a observaria e qual garantia sobrevive a ela. Nas entradas abaixo, “o que é de fato verdade” responde à primeira, “o que você vê” responde à segunda, e o padrão nomeado no fim de cada entrada responde à terceira. Juntas, essas três respostas são a semântica de uma falha, e o restante do livro continua pedindo por elas.

Dez falhas que importam

A lista abaixo é o conjunto com que começaria para um sistema construído com serviços, bancos de dados relacionais e um message broker. Ela não é completa, e nenhuma lista é.

Partições de rede e a incerteza do timeout

O cenário de abertura é o caso-modelo. Duas partes estão ligadas por uma rede, e a rede descarta, atrasa ou reordena o tráfego. Uma **partição de rede** é a forma extrema, em que dois grupos de máquinas não conseguem se alcançar de jeito nenhum. Um timeout é a forma cotidiana, e é muito mais comum.

O que cada lado consegue observar é assimétrico. Quem chama vê apenas que nenhuma resposta chegou. Quem é chamado pode ter feito o trabalho e perdido a resposta, ou nunca ter visto a requisição. O mesmo vale para cada salto na Parcelly: um serviço chamando uma transportadora externa, um produtor enviando ao broker, um consumidor dando ack ao broker.

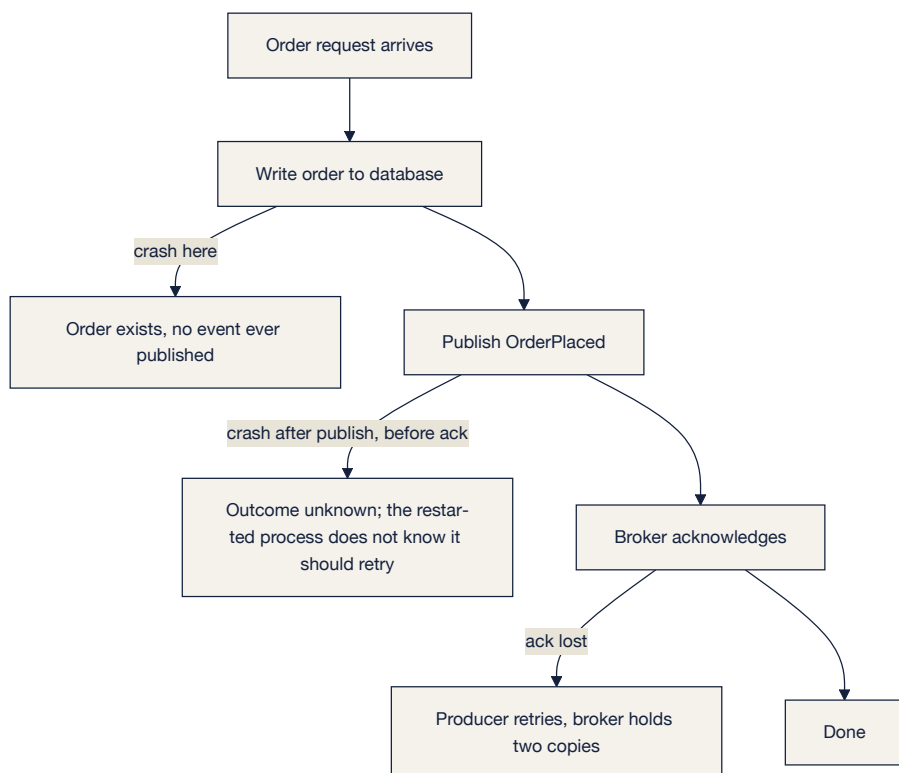
A única resposta honesta a isso é tornar a operação segura de repetir. Se o Payments envia ao provedor uma chave de idempotência estável a cada captura, e o provedor trata uma chave repetida como a mesma requisição, então o retry após um timeout se torna inofensivo. O design deixa de depender da resposta a “aconteceu?”, porque as duas respostas levam ao mesmo estado final. O Capítulo 6 cobre quando e como fazer retry, e o Capítulo 7 cobre a idempotência que torna o retry seguro.

Alguns sistemas externos não oferecem tal chave. Nesse caso, você pergunta ao sistema o que aconteceu antes de agir de novo, o que é uma etapa de reconciliação, e o Capítulo 13 a cobre.

Janelas de queda do produtor

Considere o Orders tratando um novo pedido. Ele precisa salvar o pedido no seu banco de dados e publicar `OrderPlaced` no broker. São dois sistemas, e nenhuma transação abrange os dois. Esse é o dual write do Capítulo 1, e a parte interessante é o que cada janela de queda deixa para trás.

O processo pode cair entre as duas escritas, e a ordem das escritas decide qual dano você recebe.



O diagrama desenha a ordem em que o banco de dados vem primeiro e mostra três janelas: uma queda antes da publicação, uma queda depois dela com resultado desconhecido e um acknowledgment perdido. Se o commit no banco de dados vem primeiro e o processo morre antes de publicar, o pedido existe e ninguém a jusante jamais fica sabendo dele. O Payments nunca autoriza, o Inventory nunca reserva, e o comprador espera por uma confirmação que nunca chega. Nada dá erro. O pedido fica ali parado.

A ordem inversa, que o diagrama não desenha, tem a sua própria janela. Se você publica primeiro, uma queda antes do commit deixa um estado pior: um evento anunciando um fato que nunca se tornou verdadeiro. Os serviços a jusante agem sobre um pedido que não existe. E, em qualquer das ordens, se o acknowledgment do broker se perde na volta, o produtor fará um retry e o broker poderá ter duas cópias do mesmo evento.

O que se vê de fora é silencioso: pedidos sem eventos, eventos sem pedidos e duplicados ocasionais. Nada nos logs diz “perdido”. O Capítulo 4 introduz o transactional outbox, que elimina a janela ao fazer da mudança

de estado e da intenção de publicar uma única transação local. O Capítulo 5 cobre como as linhas do outbox viajam depois até o broker.

Failover do broker

Os brokers são replicados para que uma máquina possa morrer sem levar os dados junto. Quando a máquina que guarda a cópia líder de uma partição ou fila morre, outra assume. Isso é um **failover**, e é um comportamento projetado, não um mau funcionamento.

O que importa é o que o produtor ouviu antes do failover. Se uma mensagem que recebeu ack instantes antes de um líder morrer sobrevive depende de como o broker foi configurado para dar ack: de forma alguma, depois que o nó receptor a aceitou (tenha ou não feito o flush para o disco), ou depois que um número exigido de réplicas a tem. As configurações mais fortes custam latência e throughput. As mais fracas podem perder mensagens que receberam ack em um failover. Isso difere entre famílias de brokers e entre configurações, então não afirmarei aqui um comportamento específico para um produto específico; o ponto é que a garantia tem uma fronteira, e a fronteira é uma configuração que alguém escolheu.

Durante o failover, os produtores também veem erros e timeouts, fazem retry e podem publicar duplicados de mensagens que o antigo líder já havia aceitado. O Capítulo 2 descreve onde as garantias do broker terminam. A regra aqui é tratar “ack do broker” como evidência forte e não como prova.

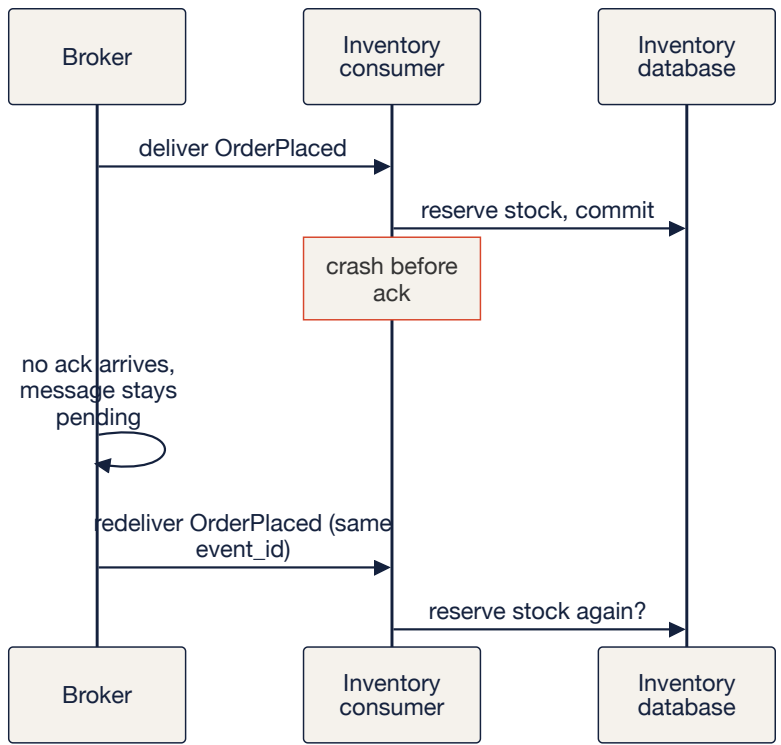
O que você vê: uma rajada de erros de publicação, uma queda no throughput dos consumidores e depois uma onda de reentregas. Se uma mensagem foi perdida, você não vê nada, e é por isso que a reconciliação do Capítulo 13 importa.

Queda do consumidor: processado, mas sem ack

Agora, o lado do consumidor. O Inventory recebe `OrderPlaced`, reserva estoque no seu banco de dados e faz o commit. Então o processo morre antes de poder dar ack na mensagem.

Do ponto de vista do broker, a mensagem nunca terminou. Em um broker baseado em filas, o broker percebe a conexão fechada ou o visibility timeout expira, e a mensagem é reentregue. Em um broker baseado em logs, o

commit do offset do consumidor nunca foi feito, então o próximo consumidor proprietário da partição começa da posição antiga e lê a mensagem de novo. Nas duas famílias, a segunda entrega chega para um trabalho que já foi feito.



A última linha do diagrama é o problema inteiro. Um consumidor que reserva estoque incondicionalmente a cada entrega o reserva duas vezes. Esta é a fonte comum de duplicados, e é a razão pela qual o padrão do livro é a entrega at-least-once combinada com consumidores idempotentes. Um consumidor pode cair em pontos diferentes, e cada ponto tem uma consequência diferente:

```

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

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

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

```

Esta é uma versão compacta do handler do Capítulo 2, reduzida para mostrar os pontos de queda. O efeito do ponto de queda A é nenhum. O efeito do ponto de queda C é nenhum. O interessante é o B, e só sobrevive a ele um consumidor que registra o que processou, na mesma transação do trabalho. O Capítulo 7 constrói isso como se deve.

O que você vê: efeitos colaterais duplicados, como duas reservas, dois e-mails, dois lançamentos no ledger, e nenhum erro em lugar algum, porque cada tentativa de processamento teve sucesso.

Consumidores lentos e backpressure

Nem toda falha é uma queda. Um consumidor que está apenas lento pode causar tanto dano quanto, e é mais difícil de notar, porque nada está quebrado.

Suponha que o Notifications chame um provedor de e-mail que passa a levar 10 segundos por requisição em vez de 100 milissegundos. O Notifications continua funcionando, mas cada mensagem agora ocupa um worker 100 vezes mais. As mensagens chegam mais rápido do que são concluídas, então o atraso do consumidor, que o Capítulo 2 definiu, cresce de forma constante.

A lentidão então dispara falhas secundárias. Em um broker baseado em filas, uma mensagem que ultrapassa o seu visibility timeout reaparece enquanto o primeiro worker ainda a processa, e um segundo worker começa

a mesma mensagem. Os duplicados somam carga a um sistema que já está lento. Em um broker baseado em logs, um consumidor que não volta ao broker a tempo pode ser declarado morto e ter suas partições reatribuídas, o que inicia a próxima falha desta lista.

Backpressure é a disciplina de limitar o trabalho em andamento em cada estágio para que a carga permaneça finita. Em um sistema mediado por broker, o broker absorve a diferença, que é o benefício do desacoplamento temporal, mas o buffer é finito e se esgota de duas maneiras diferentes. Quando o disco do broker enche, ele pode limitar ou bloquear os produtores, ou parar de atender por completo, e qual dessas coisas o seu broker faz é algo que convém descobrir antes do dia em que acontecer. Seja qual for, o produtor percebe, e isso é backpressure de verdade: ruidoso e visível. Quando um consumidor fica mais atrasado do que o período de retenção de um log, ou uma mensagem na fila sobrevive além da sua expiração, o broker apaga mensagens que nunca foram lidas. Isso não é atraso. É perda silenciosa: nenhum erro é levantado, nenhum consumidor falha, e os eventos `PaymentCaptured` não lidos desapareceram. Muito antes de qualquer um dos dois limites, um backlog já pode violar um prazo de negócio, como uma autorização de pagamento que expira antes de ser capturada.

Como o buffer do broker apenas adia o problema, a carga precisa ser limitada em vários pontos ao mesmo tempo. Um consumidor pode limitar o número de mensagens em andamento, ou o lote que ele busca antecipadamente. Pode limitar a concorrência dos seus workers, de modo que um provedor de e-mail lento prenda um número fixo de workers e não todos. Pode aplicar rate limit às suas chamadas à dependência lenta, ou pausar o consumo por completo enquanto a dependência se recupera. Uma fila pode ter um limite de comprimento com uma política explícita de overflow, para que você escolha o que é descartado ou recusado em vez de deixar o broker escolher.

O lado produtor e a topologia também impõem limites. No lado produtor, o outbox absorve um broker que recusa publicações, porque as linhas esperam no banco de dados até o relay conseguir entregá-las. E, em um broker baseado em logs, o número de partições limita quantos consumidores de um grupo podem trabalhar em paralelo, de modo que adicionar instân-

cias deixa de ajudar em um número fixado de antemão. Cada um desses limites troca throughput por estabilidade, e cada um precisa de uma métrica, ou você não saberá que é ele que o está segurando.

O que você vê: atraso e idade das mensagens em alta, um backlog crescente, contagens de reentrega subindo. O Capítulo 6 cobre backoff, limites de mensagens em andamento e a pausa do consumo, o Capítulo 8 cobre o que fazer com mensagens que não podem ser processadas a tempo, e o Capítulo 13 cobre as métricas de atraso e idade que avisam cedo.

Mensagens que falham sempre e dependências que falham para toda mensagem

Até aqui, toda falha era questão de acaso: uma queda, uma resposta perdida, um mau momento. Algumas falhas são determinísticas. Uma **poison message** é uma mensagem que falha em toda tentativa, não importa o momento. Suponha que um `OrderPlaced` chegue ao Inventory com um SKU que não existe no seu catálogo. O Inventory levanta um erro, a mensagem é reentregue, o erro se repete, e nada na próxima tentativa é diferente da anterior.

A causa também pode estar fora da mensagem. Se o provedor do Payments está fora do ar, toda captura falha, e toda mensagem parece envenenada enquanto durar a indisponibilidade. Os dois casos pedem respostas opostas. Uma mensagem que nunca terá sucesso deve sair rapidamente do fluxo principal, para uma **dead-letter queue (DLQ)**, um lugar separado onde as mensagens que não podem ser processadas esperam por uma pessoa ou por um job de reparo. Uma mensagem que falha porque uma dependência está fora do ar não deve ir para a dead letter de forma alguma, porque teria sucesso após a recuperação, e mandar um backlog inteiro para dead letters durante uma indisponibilidade transforma um problema temporário em um problema manual.

O que dá errado no meio segue um padrão. Os retries de uma poison message gastam tentativas e capacidade sem nenhuma chance de sucesso. Em um broker baseado em logs, ou em uma fila que preserva a ordem, uma única mensagem dessas pode bloquear tudo o que vem atrás. Muitos consumidores fazendo retry contra uma única dependência com falha somam carga àquilo que está tentando se recuperar, um retry storm. E um bug de

código pode falhar sem erro algum: o consumidor dá ack, retorna e escreve o valor errado, de modo que nada no caminho de falha chega a executar.

O que você vê: o mesmo `event_id` aparecendo nos logs repetidamente, contagens de reentrega que nunca diminuem, uma partição travada, uma DLQ que cresce durante uma indisponibilidade e, no caso do bug silencioso, nada, até que uma verificação de reconciliação discorde. O Capítulo 6 cobre como distinguir quais falhas valem um retry, o Capítulo 8 cobre as poison messages e o caminho de dead letters, e os capítulos 13 e 14 cobrem como encontrar e reparar o dano de um bug que não levantou erro. O preço de qualquer classificação é que ela pode estar errada, então mantenha a decisão visível e reversível.

Relógios e timestamps em que você não pode confiar

Toda máquina tem um relógio, e dois relógios nunca concordam exatamente. Eles derivam, são corrigidos por sincronização, e uma correção pode mover um relógio para trás. Uma máquina virtual pausada acorda com um relógio errado pelo tempo que dormiu.

A consequência para os eventos é específica. `occurred_at` é um fato sobre quando algo aconteceu segundo uma máquina. Comparar o `occurred_at` de um evento do Orders com o de um do Payments diz algo sobre os relógios delas tanto quanto sobre a ordem dos eventos. Se você ordena por timestamp para decidir qual atualização vence, um serviço com o relógio alguns segundos adiantado vencerá todo conflito que deveria perder.

Os timeouts têm uma armadilha parecida. Um deadline expresso como horário de relógio de parede, “esta autorização é válida até as 14h00”, é comparado por uma máquina cuja ideia de 14h00 pode diferir da do emissor. Para medir durações em uma única máquina, um relógio monotônico, que só avança, é a ferramenta certa. Para ordenar entre máquinas, não usaria o relógio de parede de forma alguma.

O que ordena eventos de forma confiável é a estrutura do sistema: em um broker baseado em logs, uma partição com chave pelo agregado, dando uma ordem total aos eventos dessa chave, desde que haja um único caminho de produtor para cada chave e que o número de partições não mude, ou um número de versão explícito no agregado que os consumidores pos-

sam comparar. O Capítulo 9 cobre a ordenação em profundidade. Trate `occurred_at` como informação para humanos e para analytics, e desconfie dele como entrada de correção.

O que você vê: atualizações aplicadas fora de ordem, sequências “impossíveis” nos logs em que um efeito parece preceder a sua causa e bugs que aparecem apenas em um host.

Deploys e rebalanceamentos

Um deploy é uma falha planejada. As instâncias são paradas e iniciadas e, no tempo entre uma coisa e outra, o trabalho é interrompido no meio. Se o seu processo de deploy envia um sinal de término e o consumidor sai com mensagens em andamento, você criou o caso “processado, mas sem ack” de propósito, repetidamente, a cada release.

Um **rebalanceamento** é a versão da família de logs, e pode ocorrer sem um deploy. Quando membros entram em um grupo de consumidores ou saem dele, ou quando um membro é julgado sem resposta, as partições são reatribuídas. Durante a transferência, dois consumidores podem acreditar por pouco tempo que são proprietários da mesma partição, e um consumidor que estava lento pode continuar processando mensagens depois que a sua partição já foi para outro lugar. O equivalente na família de filas são os consumidores concorrentes cujos visibility timeouts expiram enquanto o trabalho continua.

O padrão de dano é uma rajada de duplicados na hora do deploy e, às vezes, uma breve paralisação enquanto as partições são reatribuídas. Um deploy que percorre todas as instâncias em rápida sucessão pode disparar vários rebalanceamentos seguidos e, no pior caso, os consumidores passam mais tempo reatribuindo o trabalho do que fazendo-o. Duas mitigações reduzem isso, ao preço de mais configuração para entender: a reatribuição cooperativa, em que só as partições que precisam se mover são retiradas, e uma identidade estável de membro, para que uma instância reiniciada possa recuperar suas partições antigas sem um reembaralhamento completo.

Faça os consumidores pararem com elegância, concluindo ou liberando as mensagens em andamento antes de sair, e suponha que o caminho elegante às vezes falhará, de modo que o consumidor idempotente continue sen-

do a defesa real. Também espaçaria os rolling deploys no tempo para que não disparem reatribuições em um loop apertado. O que você vê: duplicados e picos de atraso que coincidem com os horários de release.

Erros humanos

A última categoria é a que a maioria dos documentos de arquitetura deixa de fora. Pessoas operam esses sistemas, e pessoas cometem erros sob pressão, em geral enquanto reparam uma falha anterior.

Imagine a pessoa on-call na Parcelly que encontra 40.000 dead letters e aperta o botão para fazer o redrive de todas. Se a causa foi corrigida, elas passam. Se não foi, falham de novo, imediatamente, e inundam o sistema que a pessoa estava tentando curar. Se algumas já haviam sido tratadas por um contorno manual, são duplicados de um trabalho que alguém fez à mão.

Outras formas são igualmente comuns. Alguém edita uma linha diretamente para destravar um cliente, contornando o outbox, de modo que nenhum evento é publicado e os serviços a jusante divergem. Alguém faz o deploy de um produtor com um payload alterado antes de os consumidores conseguirem lê-lo. Alguém purga uma fila no ambiente errado. É isso o que acontece quando as ferramentas de reparo são amplas e as salvaguardas faltam.

Trataria as ações dos operadores como parte do modelo de falhas e as projetaria como qualquer outro componente: redrive em lotes pequenos com uma chave de pausa, ações de reparo que passam pelo mesmo caminho das mudanças normais para que os eventos sejam emitidos, e toda ação manual registrada em uma trilha de auditoria. O Capítulo 8 cobre as dead letters e o redrive, o Capítulo 12 cobre as mudanças de payload, e o Capítulo 13 cobre a reconciliação, que é como você descobre que uma correção humana deu errado.

O que você vê: um pico repentino logo depois de alguém mexer em algo e dados que discordam entre serviços sem nenhuma mensagem com falha para explicar.

Perder um armazenamento inteiro

Toda falha até aqui supunha que os armazenamentos sobrevivem: o banco de dados guarda o que teve o commit feito, e o broker guarda o que recebeu ack. A última entrada abandona essa suposição. Um cluster ou uma região é perdido. Um broker perde dados além do que a sua replicação cobria. O banco de dados de um consumidor é restaurado a partir de um backup feito seis horas atrás. O banco de dados de um produtor é restaurado para um ponto anterior às últimas centenas de pedidos.

Essas não são uma única falha, porque o dano depende de qual armazenamento voltou no tempo. Se o banco de dados do Inventory é restaurado, ele esquece as reservas feitas depois do ponto de restauração, e o broker pode já não guardar os eventos que as recriariam. Se o banco de dados do Orders é restaurado, ele esquece pedidos sobre os quais os serviços a jusante já agiram, e o Payments guarda capturas de pedidos que o Orders não conhece mais. Nenhum dos lados levanta erro. Cada um é coerente consigo mesmo e errado sobre o outro.

Uma propriedade ajuda. Como o inbox mora no mesmo banco de dados que os efeitos, uma restauração leva os dois de volta ao mesmo momento. As mensagens processadas antes do ponto de restauração continuam no inbox, então um replay as ignora, o que está correto porque seus efeitos também sobreviveram. As mensagens processadas depois do ponto de restauração são as que correm risco, e reproduzi-las é seguro: o inbox já não se lembra delas, e seus efeitos se perderam junto.

Os efeitos externos são a exceção. Uma captura no provedor de pagamentos não é revertida junto com o seu banco de dados, então essas chamadas precisam das próprias chaves de idempotência, ou o replay cobra o comprador uma segunda vez. A regra é reconciliar antes de retomar o consumo: comparar o que cada serviço acredita, reparar as diferenças e só então deixar o fluxo correr.

O que você vê: nada, no início, e esse é o problema. Os serviços estão saudáveis e os dashboards estão verdes, enquanto os totais discordam entre eles. O Capítulo 13 cobre a reconciliação que encontra a discordância, e o Capítulo 14, Recuperação, desastres e migração de brokers, cobre como

restaurar, fazer replay e retomar. Projetar para isso custa backups que são de fato testados, retenção que dure mais que o seu tempo de recuperação e a disciplina de ensaiar uma restauração antes de precisar dela.

O mapa de falhas

Aqui está o mesmo material em um só lugar. Mantenha-o ao lado das suas revisões de design.

Falha	O que você vê	O que é de fato verdade	Onde é tratada
Partição de rede ou timeout	Quem chama não recebe resposta	A operação pode ou não ter sido executada	Capítulo 6, Capítulo 7, Capítulo 13
Queda do produtor entre a escrita e a publicação	Pedidos sem eventos, ou eventos sem pedidos	Estado e evento discordam	Capítulo 4, Capítulo 5
Acknowledgment do broker perdido	Eventos duplicados no broker	A primeira publicação teve sucesso	Capítulo 7
Failover do broker	Erros de publicação, depois re-entregas; raramente, perda silenciosa	A garantia depende das configurações de acknowledgment	Capítulo 2, Capítulo 13
Queda do consumidor após o trabalho	Efeitos colaterais duplicados, sem erros	Trabalho feito, mensagem ainda pendente	Capítulo 7
Consumidor lento	Atraso e idade das mensagens crescentes	Consumidor vivo, ficando para trás	Capítulo 6, Capítulo 8, Capítulo 13
Retenção ou expiração durante o atraso	Nada; os eventos ausentes nunca chegam	Mensagens não lidas foram apagadas pelo broker	Capítulo 8, Capítulo 13
Poison message ou dependência com falha	A mesma mensagem reentre-gue; partição travada; DLQ crescendo durante uma indisponibilidade	A falha é determinística, ou a causa está fora da mensagem	Capítulo 6, Capítulo 8
Bug do consumidor sem erro	Nada; os totais discordam depois	O handler teve sucesso e escreveu o resultado errado	Capítulo 13, Capítulo 14

Falha	O que você vê	O que é de fato verdade	Onde é tratada
Skew de relógio	Atualizações fora de ordem, sequências impossíveis	Os timestamps discordam da causalidade	Capítulo 9
Deploy ou rebalanceamento	Duplicados e atraso na hora do release	A responsabilidade pelo trabalho mudou no meio do caminho	Capítulo 7, Capítulo 13
Erro de operador	Pico após uma ação manual; divergência silenciosa	Um reparo mudou o estado fora do caminho normal	Capítulo 8, Capítulo 13
Perder um armazenamento inteiro	Nada no início; os serviços discordam após uma restauração	Um armazenamento voltou no tempo, ou seus dados se foram	Capítulo 13, Capítulo 14

Duas coisas se destacam. Primeiro, os duplicados aparecem na coluna “O que você vê” por três causas diferentes (acknowledgment perdido, queda do consumidor, deploy), e as reentregas por uma quarta. É por isso que um padrão, o consumidor idempotente, merece um capítulo inteiro. Segundo, a última coluna volta sempre aos capítulos 13 e 14. Algumas falhas não podem ser evitadas nem sequer detectadas no momento em que ocorrem, então o sistema também precisa de um meio de perceber depois que duas visões da verdade discordam.

Escrevendo o modelo de falhas

Um modelo de falhas só é útil se for escrito, atual e compartilhado. Na minha visão, a prática vale mais que qualquer entrada isolada, porque o ato de escrever força a pergunta “o que supomos aqui?”, e as suposições são onde moram as surpresas.

Manteria o modelo como um documento curto no repositório, ao lado da descrição da arquitetura, com uma entrada por falha e os mesmos campos todas as vezes. Os campos espelham as três perguntas: “where” e “we assume” dizem o que pode falhar, “observable as” diz como você a veria e “defense” diz qual garantia sobrevive. Uma entrada utilizável tem este aspecto:

```
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
```

Os campos que importam são os três últimos. “Verified by” transforma a entrada de uma crença em uma afirmação que pode ser testada, e o Capítulo 15 mostra como testá-la. “Owner” significa que alguém recebe o page. “Last reviewed” admite que o modelo apodrece: serviços são adicionados, brokers são trocados, e uma suposição que era verdadeira no ano passado pode não ser agora.

Nota: O modelo deve declarar também o que você supõe que não pode acontecer. “Supomos que o banco de dados nunca perde uma transação com commit feito” é uma entrada legítima. É melhor registrar a suposição por escrito, e saber o que você faria se ela fosse falsa, do que deixá-la implícita.

Use o modelo em três lugares. Nas revisões de design, percorra cada novo fluxo contra a lista e pergunte a quais falhas ele está exposto. Nas revisões de incidentes, encontre a qual entrada o incidente pertence e, se a nenhuma, acrescente uma. Nos testes, transforme cada entrada em um cenário. Uma falha que aparece no modelo, mas em nenhum teste, é uma esperança.

Trade-offs e custos

Escrever um modelo de falhas custa tempo e o desconforto de registrar o que você não sabe. Uma equipe pequena, com um serviço e uma fila, não precisa de todas as entradas desta lista. Comece pelas três ou quatro falhas que mais doeriam.

O modelo também tem uma fronteira. Toda defesa neste livro supõe que algumas falhas estão fora de escopo: que o banco de dados é durável, que a replicação do broker funciona como configurada. Essas suposições são razoáveis, e cada uma é um lugar onde o modelo pode estar errado. Listá-las permite que você decida se são aceitáveis, o que não se consegue enquanto estão invisíveis. Um bug no seu próprio código é diferente. Ele está no escopo, como uma falha a detectar e da qual se recuperar, porque vai acontecer, e o modelo precisa dizer como você o notaria e repararia o que ele escreveu.

Um modelo com 40 entradas não será lido, e um com três deixará escapar casos. Buscaria uma lista que um novo membro da equipe consiga ler de uma só vez.

Projetar para toda falha com força máxima também não é o objetivo. Sobreviver a um failover de broker sem perda custa latência e dinheiro, e sobreviver a uma falha dupla custa mais. Um e-mail do Notifications que às vezes é enviado duas vezes não merece a mesma defesa que um lançamento do Ledger que nunca pode ser duplicado. O trabalho do modelo é permitir que você gaste conforme o risco. Se não consigo nomear o custo de negócio de uma falha, hesitaria em pagar um custo técnico para evitá-la.

O modelo não torna o sistema confiável. Ele torna as suas suposições inspecionáveis.

Erros comuns

- **Modelar apenas quedas.** Uma queda é tentadora de modelar porque é limpa e detectável. As falhas caras são as ambíguas, como timeouts e consumidores lentos. Correção: para cada chamada, escreva o que quem chama faz quando nenhuma resposta chega.
- **Tratar um timeout como uma falha.** Parece natural ler “sem resposta” como “não aconteceu”. Correção: trate-o como desconhecido e projete a operação de modo que repeti-la seja seguro ou verificável.
- **Confiar no acknowledgment do broker como prova.** A palavra “acknowledged” soa definitiva. Correção: saiba o que a sua configuração de acknowledgment de fato garante e reconcilie para os casos em que não garante.

- **Ordenar por timestamp.** `occurred_at` está ali, no envelope, e ordenar por ele leva uma linha. Correção: ordene dentro de uma chave de partição pela versão do agregado e guarde os timestamps para humanos.
- **Ignorar os deploys.** Os releases parecem rotina, não falha. Correção: conte cada deploy como uma queda de consumidor agendada e teste o desligamento enquanto há mensagens em andamento.
- **Deixar os operadores de fora.** As ferramentas de reparo recebem menos revisão que o código de produto. Correção: ponha o redrive, o replay e as edições manuais no modelo, com limites de lote, chaves de pausa e trilhas de auditoria.
- **Tratar um banco de dados restaurado como um começo limpo.** Uma restauração parece o fim do incidente. Correção: depois de qualquer restauração, reconcilie com os outros serviços antes de o consumo ser retomado e verifique os efeitos externos que não foram revertidos.
- **Fazer retry de uma falha que não vai mudar.** Os retries parecem diligência. Correção: separe as falhas que dependem do momento daquelas que dependem da mensagem e envie as do segundo tipo cedo para o caminho de dead letters.
- **Escrever o modelo uma única vez.** Um documento escrito durante uma fase de design é um snapshot do momento. Correção: atribua um responsável, revise-o depois de incidentes e date cada entrada.

Pontos-chave

- Um timeout significa “parei de esperar”, não “falhou”, então projete operações que permaneçam corretas qualquer que seja o modo como o desconhecido se resolva.
- Cada junção tem as suas próprias janelas de queda, e o dano é silencioso. Um dual write deixa pedidos sem eventos, ou eventos sobre pedidos que não existem, e um consumidor que cai depois de fazer o commit, mas antes de dar ack, recebe um duplicado a cada reentrega, o que torna os consumidores idempotentes uma exigência.
- O acknowledgment de um broker é evidência com uma fronteira definida pelas configurações de replicação e flush, então conheça a fron-

teira e planeje reconciliar o que fica fora dela.

- Consumidores lentos e deploys causam duplicados e paralisações sem que nenhum componente esteja quebrado, então limite a carga, meça o atraso e a idade das mensagens e pare os consumidores com elegância.
- Uma mensagem que falha sempre, ou uma dependência que falha para toda mensagem, pede uma resposta diferente de uma falha casual, então classifique antes de fazer retry e reserve a dead letter para o que não vai se recuperar.
- Os timestamps de relógio de parede discordam entre máquinas, então ordene os eventos dentro de uma chave de partição pela versão do agregado e use `occurred_at` para humanos.
- Operadores e restaurações fazem parte do sistema: um armazenamento restaurado voltou no tempo, então reconcilie antes de retomar o consumo e mantenha um modelo de falhas escrito que nomeie responsáveis, testes e datas de revisão, para que os reparos não se tornem o próximo incidente.

A primeira falha a remover é a do produtor, onde uma mudança de estado e o seu evento podem se separar sem que ninguém perceba. O Capítulo 4 inicia a Parte II com o transactional outbox, que fecha essa janela.

PARTE II

Publicar com confiabilidade

Capítulo 4 — O problema do dual write e o transactional outbox

Imagine a Parcelly algumas semanas depois de lançar as promoções para vendedores. Uma engenheira de suporte abre um ticket que diz, na íntegra: “O comprador diz que concluiu o checkout do pedido 81724, mas o vendedor nunca recebeu a etiqueta de envio. O pedido aparece como realizado.” A engenheira consulta o banco de dados de Orders e encontra a linha, com status `PLACED`, criada às 14:02:11. Depois consulta Payments, e não há autorização para esse pedido. Inventory também não tem nenhuma reserva. Para o restante da Parcelly, o pedido não existe.

Ela pesquisa os logs de Orders por volta das 14:02 e encontra uma marca de deploy. O serviço foi reiniciado 200 milissegundos depois de fazer o commit do pedido e antes de publicar `OrderPlaced`. O código fez exatamente o que foi escrito para fazer: salvar e depois publicar. Nada lançou um erro, porque o processo que o lançaria já tinha desaparecido. Suponha que uma consulta rápida mostre outros 11 pedidos na mesma situação, todos dentro do mesmo minuto.

A equipe corrige os 11 à mão e acrescenta uma observação ao runbook. Ninguém se sente bem com isso. O código não está errado de nenhuma forma que um revisor apontaria, e mesmo assim ele pode perder trabalho sempre que um processo morre no momento errado. Uma máquina maior não muda isso, e um desenvolvedor mais cuidadoso também não. A pergunta deste capítulo é se um serviço consegue alterar o próprio banco de dados e avisar o resto do sistema de um modo que não possa se desfazer, sem uma transação que abranja o banco de dados e o broker.

O problema e por que ele é difícil

O Capítulo 1 definiu o dual write: dois sistemas atualizados sem uma transação compartilhada. Orders precisa gravar uma linha em seu banco de dados relacional e publicar uma mensagem no broker. O banco de dados

garante a atomicidade da primeira operação. O broker garante a durabilidade, em alguma configuração, da segunda. Nada garante que as duas aconteçam ou que nenhuma aconteça.

O Capítulo 3 esboçou as janelas de falha do produtor em um diagrama. Aqui quero percorrê-las em uma tabela, porque a ordem das duas gravações é uma decisão de design que a maioria das equipes toma sem perceber, e toda escolha tem uma falha à espera logo atrás.

Toda ordenação falha

Pegue as duas sequências óbvias e pergunte, para cada ponto de falha possível, como o mundo fica depois. É o mesmo exercício de escrever a semântica de falhas da junção entre o banco de dados e o broker: o que pode falhar, como você observaria a falha e qual garantia sobrevive a ela. A tabela abaixo é esse exercício para o produtor.

Sequência	Ponto de falha	Estado do banco de dados	Estado do broker	Dano
Commit e depois publicação	Crash após o commit, antes da publicação	Pedido salvo	Sem evento	Evento perdido. Os sistemas a jusante nunca ficam sabendo do pedido
Commit e depois publicação	A publicação falha ou expira, o processo sobrevive	Pedido salvo	Desconhecido	Quem chamou precisa fazer um retry ou desistir. Desistir perde o evento
Commit e depois publicação	O broker aceita, o ack se perde	Pedido salvo	Evento presente	O retry publica um duplicado
Publicação e depois commit	Crash após a publicação, antes do commit	Sem pedido	Evento presente	Evento fantasma. Os sistemas a jusante agem com base em um fato que nunca se tornou verdadeiro
Publicação e depois commit	O commit falha (restrição, deadlock, timeout)	Sem pedido	Evento presente	O mesmo evento fantasma, agora com um erro visível no produtor
Publicação dentro da transação, commit por último	O commit falha após a publicação	Sem pedido	Evento presente	O bloco de transação dá ao código uma falsa sensação de segurança, e o broker não faz parte dele

A última linha merece um comentário, porque é a que parece mais segura. Muitos frameworks permitem publicar de dentro de um bloco de transação do banco de dados. Parece que a publicação fizesse parte da transação. Não faz. A chamada ao broker acontece de imediato e não pode sofrer rollback, e o commit ainda acontece depois. É a ordenação de publicação e depois commit com uma indentação melhor.

A imagem espelhada é o callback “after commit” que muitos frameworks oferecem, um hook que roda depois que a transação fez o commit. As equipes costumam achar que ele é seguro, porque a publicação só acontece para dados que passaram pelo commit. É a primeira linha da tabela: commit e depois publicação, com a intenção guardada na memória do processo. Um crash entre o commit e o hook perde o evento, e nada registra que ele era devido.

Há ainda uma quarta forma que a tabela não captura: um broker lento. Se a publicação acontece dentro da transação, o banco de dados mantém bloqueios e uma conexão enquanto espera pelo broker. Um broker lento por um minuto vira um banco de dados lento por um minuto. Você acoplou a disponibilidade de dois sistemas que deveriam ser independentes.

Por que o retry não ajuda

O reparo tentador é um loop de retry em volta da publicação. Os retries ajudam na segunda linha da tabela, em que o processo sobrevive e pode tentar de novo. Não fazem nada pela primeira linha, em que o processo desapareceu, e criam o duplicado da terceira linha. O Capítulo 6 trata dos retries como se deve. Por ora, observe que um loop de retry vive na memória do processo, e a falha que nos interessa é justamente o processo desaparecer.

O segundo reparo tentador é um job periódico que compara o banco de dados com o broker e republica o que falta. Algumas equipes constroem isso, e acho uma rede de segurança razoável, à qual o Capítulo 13 volta com o nome de reconciliação. Como mecanismo principal, ele tem dois problemas. O broker raramente pode ser consultado por chave de negócio, então “o que falta” vira um palpite caro. E o próprio job faz o mesmo dual write, de modo que pode falhar das mesmas formas, só que com menos frequência e mais confusão.

Por que não uma transação distribuída

Se o problema são dois sistemas sem uma transação compartilhada, a resposta clássica é dar uma a eles. O **two-phase commit (2PC)**, padronizado para bancos de dados e gerenciadores de transação como XA, coordena vários gerenciadores de recursos. Na primeira fase, cada participante se prepara, o que significa que promete que consegue fazer o commit e mantém seus bloqueios. Na segunda fase, um coordenador diz a todos os participantes que façam o commit, ou o rollback caso algum deles não tenha conseguido se preparar.

No papel, isso fecha a janela. Na prática, raramente é a resposta para um broker e o banco de dados de um serviço, por razões que são sobretudo operacionais.

- **O suporte é irregular.** Muitos brokers populares não implementam nenhuma interface de gerenciador de transações. Onde algum implementa, as garantias e o tratamento de falhas diferem dos do banco de dados, e você herda os dois conjuntos de regras.
- **Os participantes mantêm bloqueios enquanto o coordenador decide.** Se o coordenador sofre um crash entre as duas fases, as transações preparadas ficam em dúvida. As linhas continuam bloqueadas até que alguém, ou alguma coisa, as resolva. A janela de dúvida é exatamente o tipo de falha rara e longa que ninguém ensaia.
- **O coordenador se torna um componente que você precisa operar, proteger e recuperar.** Ele precisa de logs duráveis próprios e de um procedimento de recuperação que tem de estar correto.
- **A disponibilidade se multiplica.** Uma transação que precisa que o banco de dados, o broker e o coordenador estejam saudáveis ao mesmo tempo só está disponível quando os três estão. A independência que tornava o broker atraente se perdeu.
- **O throughput e a latência sofrem.** Duas idas e vindas, gravações forçadas em cada participante e bloqueios mantidos durante chamadas de rede se somam.
- **Não se estende ao mundo externo.** Um provedor de e-mail ou de pagamentos não vai entrar na sua transação. Mesmo uma equipe que adote o 2PC para o broker ainda precisa de outra resposta para cada chamada externa.

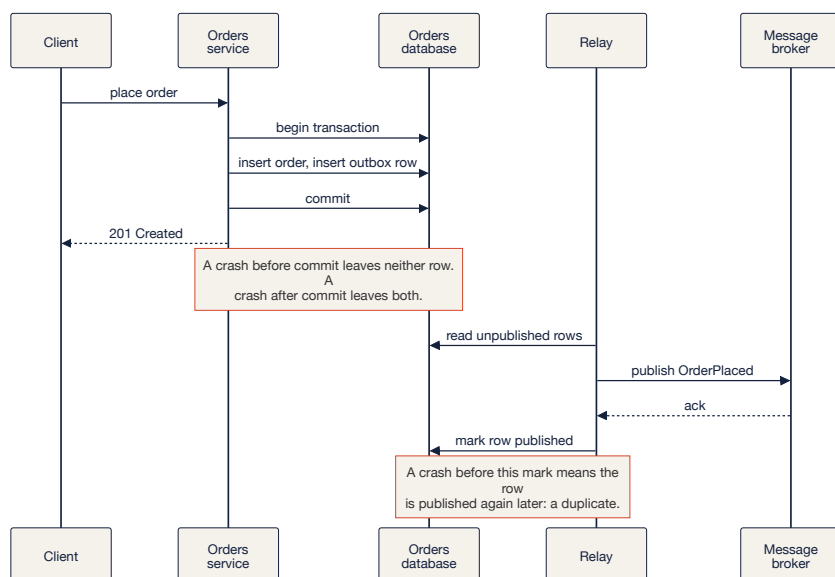
Não descartaria o 2PC em todos os casos. Uma equipe que controla os dois recursos, opera um gerenciador de transações maduro e tem as habilidades operacionais para resolver transações em dúvida consegue fazê-lo funcionar. Para a maioria das equipes que constroem um marketplace com serviços, cada um com seu banco de dados e um broker que não expõe nenhuma interface XA, o custo é alto e o benefício é estreito. O que elas querem é menor do que uma transação distribuída: querem que a mudança de estado e a intenção de publicar façam o commit juntas. Esse é um problema local, e as transações locais são bem compreendidas.

O transactional outbox

A ideia é transformar o dual write em uma única gravação mais um movimento posterior e repetível. Em vez de publicar no broker dentro da requisição, o serviço grava, em uma tabela do próprio banco de dados, uma linha que descreve o evento, na mesma transação local da mudança de estado. Essa tabela é o **transactional outbox**. A partir daqui, o livro o chama de outbox.

Um processo separado, o **relay**, lê as linhas do outbox e as publica no broker. O Capítulo 5 trata do relay. Neste capítulo, nós o tratamos como uma função que, no fim, entrega cada linha pelo menos uma vez, e examinamos o que isso nos dá.

O raciocínio é curto. O pedido e a sua linha no outbox fazem o commit juntos, ou nenhum dos dois faz, porque vivem em um só banco de dados e em uma só transação. Depois que a linha existe, a intenção de publicar é durável. Um crash após o commit não perde mais o evento, porque a linha continua lá quando o serviço, ou o seu substituto, sobe de novo. Um crash antes do commit não deixa pedido nem linha, então não existe evento fantasma.



Observe as duas notas do diagrama. A primeira mostra a janela que fechamos. A segunda mostra a janela que mantivemos: o relay pode publicar e sofrer um crash antes de registrar que publicou. O outbox não elimina os duplicados, mas leva a falha para um lugar em que ela é limitada e tratada. Volto a isso mais adiante.

A tabela

O outbox segue o formato que o resto do livro usa. Ele carrega todos os campos do envelope, para que o relay possa montar o evento sem perguntar nada ao serviço produtor. Este capítulo acrescenta quatro colunas além do envelope, cada uma por um motivo.

```

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

```

Algumas dessas escolhas precisam de defesa.

- **id e event_id são coisas diferentes.** O `event_id` é um UUID que os consumidores usarão como chave de idempotência, e ele não pode mudar quando o evento é publicado duas vezes. O `id` é um número de sequência atribuído pelo banco de dados que o relay pode usar para encontrar linhas. Ele não diz nada sobre a ordem do commit, e a próxima seção mostra por quê.
- **aggregate_version é a fonte de verdade da ordenação.** É a versão do agregado, incrementada na mesma transação que grava a linha. Nos pedidos da Parcelly, é a `order_version`. Os consumidores que precisam de ordem por agregado comparam esse número, e não o `id` nem a ordem de chegada. O Capítulo 9 mostra como um consumidor o usa. O envelope padrão não tem um campo para ele, então, quando o relay monta o evento, copia a coluna para o payload como `order_version`, e é ali que os consumidores a leem.
- **aggregate_type e partition_key são separados.** A `partition_key` é o que o broker usa para a ordenação, e é o id do agregado, como `order-81724`. O `aggregate_type` diz ao relay, e a quem lê a tabela, a

que tipo de coisa a chave se refere, e é uma forma conveniente de direcionar linhas para topics diferentes.

- **created_at é separado de occurred_at.** A primeira é o momento em que a linha foi gravada, e a segunda é o momento em que o fato aconteceu. Em geral as duas estão a milissegundos de distância. Elas divergem quando um serviço registra um fato que soube tardiamente, como uma leitura de transportadora que chega depois do evento. Uma métrica de atraso do relay deve usar `created_at`, porque o atraso é uma propriedade da tabela, não do negócio. Deixe que o banco de dados a preencha por um valor default da coluna, em vez de chamar `now()` no código da aplicação, para que o relógio seja um só. Ainda assim é o momento do insert, não o do commit: uma transação longa faz a linha parecer mais antiga do que na visão do relay, e essa parte do atraso não é culpa do relay.
- **headers guarda as questões de transporte.** O trace context e as dicas de roteamento ficam aqui, para que o payload continue sendo o fato de negócio. Uma equipe que não consegue decidir se um campo é dado ou dica de transporte deve ler o Capítulo 11 antes de acrescentá-lo ao payload.
- **published_at aceita nulo, e o nulo tem significado.** Uma linha sem `published_at` é uma linha que o relay ainda deve ao broker. Algumas equipes usam uma coluna `status`, com valores como `PENDING`, `PUBLISHED` e `FAILED`. Os dois designs carregam a mesma informação. Prefiro o timestamp porque ele diz também quando, o que é útil quando você pergunta quanto o relay se atrasou, e porque há menos a errar. Se precisar de uma forma de estacionar linhas que falham de modo permanente, acrescente uma coluna `failed_at` nesse momento, e não antes.

Esse momento chega mais cedo do que parece. Uma linha que o broker nunca vai aceitar, como um payload grande demais ou um que o serializador rejeita, bloqueia a sua chave e, no relay mais simples, a tabela inteira, até que alguém a repare. Na primeira vez que isso acontecer, você precisa de uma coluna `failed_at` (ou de um status `FAILED`, se escolheu colunas de status) e de um alerta sobre as linhas estacionadas. O Capítulo 5 mostra como o relay a usa.

Um índice sobre as linhas não publicadas, em ordem de inserção, é o único índice de que o relay precisa. Sem ele, o relay varre uma tabela que cresce para sempre. O Capítulo 5 trata da limpeza.

Gravando na mesma transação

O código produtor é curto, e a brevidade é o ponto. A função a seguir é toda a mudança no caminho de requisição de Orders.

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

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

    return order
```

Três detalhes importam mais do que parecem.

Primeiro, nada na função conversa com o broker. Uma queda do broker não faz a criação do pedido falhar, e um broker que leva um minuto para responder não mantém uma transação do banco de dados aberta. Isso é um ganho real de disponibilidade: Orders consegue aceitar pedidos enquanto o broker está fora do ar, e o outbox cresce até que o relay consiga esvaziá-lo.

Segundo, o `event_id` é criado antes do commit e guardado na linha. Toda republicação posterior desse evento carrega o mesmo id. Essa é a propriedade da qual os consumidores dependem, e é por isso que gerar o id no relay, no momento da publicação, seria um erro.

Terceiro, o payload é montado dentro da transação, a partir do estado que a transação gravou. Se o payload fosse montado depois, relendo o pedido, uma atualização concorrente poderia alterá-lo, e o evento descreveria um fato que não existia na hora. Um evento é um snapshot do que aconteceu, e o snapshot é tirado no mesmo lugar da mudança.

O pseudocódigo supõe que o insert do pedido e o insert do outbox usam a mesma conexão de banco de dados e a mesma transação. Isso se quebra por acidente com mais frequência do que parece. Uma camada de acesso a dados que abre uma segunda conexão para o outbox, ou que faz auto-commit do insert do outbox, restabelece em silêncio o dual write, agora entre duas tabelas. O teste para isso pertence à sua suíte: force uma falha depois do insert do pedido e verifique que nenhuma linha de outbox existe.

Ordem do commit, ordem do `id` e o bloqueio do agregado

Um pedido novo não tem concorrente, então `placeOrder` não precisa de bloqueio. Toda mudança posterior em um pedido existente precisa, e o bloqueio tem de vir primeiro. Eis `cancelOrder`, que emite `OrderCancelled`.

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

O bloqueio é o que torna a versão confiável, e é também a única coisa que faz a ordem do `id` significar algo para um agregado. Sem ele, tome duas transações sobre o mesmo pedido. A transação B insere sua linha de outbox e recebe o `id` 101, depois fica bloqueada na linha do pedido que a transação A mantém. A insere o `id` 102 e faz o commit, e B faz o commit

depois. A ordem do commit é A e depois B, e a ordem do `id` é B e depois A. Um relay que vê as duas linhas em um mesmo poll as publica invertidas. Com o bloqueio adquirido primeiro, B não consegue inserir até que A tenha feito o commit, então os ids de um agregado seguem a sua ordem de commit.

Mesmo assim, não faria os consumidores dependerem do `id`. As linhas de agregados diferentes se intercalam, um relay paralelo reordena entre lotes, e ninguém fora do banco de dados de Orders consegue ver o `id`. A `aggregate_version` viaja com o evento, como `order_version` no payload, e diz em que ordem os fatos aconteceram com esse pedido. O Capítulo 5 volta ao que um relay pode e não pode preservar, e o Capítulo 9 mostra o lado do consumidor.

O que o relay deve

O contrato do relay é deliberadamente pequeno. Ele seleciona as linhas não publicadas, publica cada uma no broker e a marca como publicada depois que o broker confirma. A versão mais simples está abaixo, e o Capítulo 5 trata das questões de produção que ela ignora, como lotes, bloqueios, ordenação e 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())
```

O comentário dentro do loop é toda a garantia. A linha só é marcada depois que o broker confirma, e é por isso que a chamada espera pelo ack e que uma falha interrompe o lote. Um crash entre a publicação e a marcação deixa a linha não publicada. Na passagem seguinte, o relay a publica

de novo. A alternativa, marcar primeiro e publicar depois, faria da falha um evento perdido em vez de um duplicado. O outbox escolhe a falha mais segura de propósito.

O que o outbox garante

O outbox lhe dá três coisas e tira uma.

- **Atomicidade entre o estado e a intenção.** A mudança de estado e a decisão de publicar fazem o commit juntas. Não há pedido com commit sem uma linha de outbox, nem linha de outbox sem um pedido com commit.
- **Durabilidade da intenção.** Depois que a transação faz o commit, o evento é tão durável quanto o banco de dados. O relay pode estar fora do ar, o broker pode estar fora do ar, o produtor pode ser substituído, e a linha espera.
- **Publicação at-least-once.** Toda linha com commit é publicada com o tempo, supondo que o relay e o broker se recuperem com o tempo, que o relay encontre as linhas pelo estado e não por um cursor de id, que a limpeza elimine apenas linhas publicadas e que uma linha que o broker nunca vai aceitar seja estacionada e gere alerta, em vez de ficar bloqueando para sempre (tudo isso está no Capítulo 5). Uma linha pode ser publicada mais de uma vez.
- **Não garantido: publicação exactly-once.** O relay não consegue publicar no broker e marcar uma linha no banco de dados de forma atômica, pela mesma razão que a aplicação não conseguia. A janela saiu do caminho da requisição e foi para o relay, onde é menor, observável e reparável.

Quero ter cuidado com esse resumo. O outbox não afirma que os eventos nunca se perdem. Ele afirma que a janela de perda do dual write no lado do produtor está fechada, desde que a aplicação grave pelo outbox em toda mudança de estado que deva produzir um evento. Um desenvolvedor que atualiza a tabela à mão, ou por um caminho que ignora o outbox, abre a janela de novo. O Capítulo 3 listou isso como um erro humano, e é a forma mais comum de uma equipe com outbox ainda perder eventos.

O duplicado vira problema do consumidor

Como a publicação é at-least-once, todo consumidor de um evento publicado pelo outbox precisa tolerar duplicados. Isso não é uma falha a ser eliminada por engenharia, é o preço da garantia, e vale ser claro sobre quem o paga.

A equipe produtora fica com a parte mais leve. Ela acrescenta uma tabela, grava uma linha e opera um relay. As equipes consumidoras ficam com a parte mais difícil, porque precisam reconhecer um `event_id` que já viram e evitar repetir o seu efeito. Payments não pode autorizar duas vezes. Notifications não pode enviar o mesmo e-mail duas vezes. Ledger não pode lançar o mesmo registro duas vezes.

Aviso: Adotar um outbox sem consumidores idempotentes troca um problema de evento perdido por um problema de efeito duplicado. Para alguns eventos, como uma cobrança em dobro, o segundo problema é o mais caro. Entregue os dois juntos e veja o Capítulo 7 para saber como um consumidor reconhece trabalho repetido.

Também diria isto com todas as letras às equipes produtoras: o outbox é um contrato com os seus consumidores. O `event_id` precisa permanecer estável em toda republicação do mesmo fato, e o contrato deve estar escrito na documentação do evento, não deixado implícito.

Um breve passo a passo das falhas

Para verificar o design contra as janelas do Capítulo 3, percorra de novo os crashes em Orders.

1. O processo morre antes de a transação fazer o commit. Sem pedido, sem linha de outbox, sem evento. O cliente vê um erro ou um timeout e pode fazer retry da requisição, o que é um problema à parte para a camada de API, coberto pela discussão sobre idempotência no Capítulo 7.
2. O processo morre logo após o commit. O pedido e a linha existem. O relay publica a linha na sua próxima passagem. O cliente pode não ter recebido a resposta, mas o sistema está consistente.
3. O relay morre depois de publicar e antes de marcar. O evento está no broker e a linha está não publicada. A passagem seguinte o publica de

novo. Os consumidores veem um duplicado com o mesmo `event_id`.

4. O broker está indisponível. O relay não consegue publicar, deixa as linhas não publicadas e tenta de novo mais tarde. Orders continua aceitando requisições. O atraso cresce, e um alerta também deveria crescer.
5. O serviço perde a conexão durante o commit e não sabe o resultado. Ou o pedido e a sua linha existem, ou nenhum dos dois existe, e o serviço não sabe dizer qual. Ele não deve adivinhar. A resposta segura é fazer o retry do cliente carregar uma chave de idempotência no nível da API, de modo que, seja qual for o estado do banco de dados, o retry convirja para um pedido e um evento (Capítulo 7).

Cada caso termina em consistência ou em um duplicado, e nenhum termina em evento fantasma ou perda silenciosa. Isso é tudo o que o outbox afirma, e é suficiente para justificá-lo.

Há um duplicado que fica a montante do outbox, e nenhuma defesa a jussante o enxerga. Quando um cliente faz um retry depois de uma resposta perdida e a API não tem chave própria, Orders cria um segundo pedido e um segundo `OrderPlaced` com um `event_id` novo. O inbox de cada consumidor vê então dois eventos diferentes, porque são diferentes. O outbox fez seu trabalho duas vezes. A correção pertence à API, com a abordagem de `Idempotency-Key` do Capítulo 7.

Alternativas e vizinhos

O outbox não é a única forma de impedir o dual write. Dois outros designs merecem uma comparação justa, já que uma equipe poderia preferir qualquer um deles.

Listen to yourself

No padrão **listen-to-yourself**, o serviço não grava primeiro em seu banco de dados. Ele publica o evento no broker, depois consome o próprio evento e aplica a mudança de estado no handler. O broker passa a ser a gravação única, e o banco de dados passa a ser uma projeção do que o serviço ouviu.

Isso elimina o dual write, porque há uma só gravação. Também muda a forma do serviço. A requisição do cliente retorna antes de o estado existir, então quem chamou não consegue ler a própria gravação, a menos que o sistema ofereça uma forma de esperar. A validação precisa acontecer antes da publicação, já que um evento, uma vez publicado, é um fato, e um evento inválido não pode ser rejeitado depois sem uma compensação. E o handler que aplica o evento precisa ele próprio ser idempotente, o que traz de volta o mesmo ônus do consumidor.

Consideraria isso para serviços que já pensam em eventos e toleram a consistência eventual nas próprias leituras. Para Orders, em que o navegador do comprador espera ver o pedido logo após clicar no botão, não consideraria. A lacuna de read-your-own-write é um problema de produto que a equipe acabaria resolvendo com soluções paliativas.

Event sourcing

No **event sourcing**, os eventos são a fonte de verdade. O serviço armazena a sequência de eventos por agregado, e o estado atual é derivado deles. A gravação no event store é a mudança de estado, então não há nada a manter consistente com um evento separado. A publicação no broker passa então a ler do event store, o que é um relay com outro nome.

O event sourcing resolve o dual write ao mudar o que o banco de dados é. Também lhe dá um histórico completo e a capacidade de reconstruir projeções. Custa um modelo de programação diferente, a evolução de esquemas de eventos armazenados que você nunca pode reescrever de forma casual, e uma curva de aprendizado para a equipe toda. É um compromisso arquitetural significativo, e não o adotaria só para evitar um outbox. Se você já o usa, ainda precisa de um relay, e tudo o que este capítulo diz sobre a publicação at-least-once se aplica.

Comparando as opções

Opção	O que remove	O que custa	Serve melhor quando
Commit e depois publicação	Nada	Perda silenciosa em um crash	Um evento perdido é aceitável, como métricas de melhor esforço

Opção	O que remove	O que custa	Serve melhor quando
2PC / XA	A janela, em teoria	Bloqueios, coordenador, suporte irregular	Você controla os dois recursos e executa a recuperação por conta própria
Transactional outbox	A janela do lado do produtor	Uma tabela extra, um relay, duplicados	O estado vive em um banco de dados relacional e os eventos não podem ser perdidos
Listen to yourself	A segunda gravação	Consistência eventual nas próprias leituras, validação antecipada	O serviço já é event-first
Event sourcing	A gravação de estado separada	Um modelo de estado diferente	O histórico é ele próprio um requisito

Para um serviço que já armazena seu estado em um banco de dados relacional e precisa avisar os outros sobre ele, o outbox é a menor mudança. É por isso que a maior parte deste livro o supõe.

Trade-offs e custos

O outbox é barato de descrever e não é de graça para operar. Estes são os custos que poria diante de uma equipe antes de ela se comprometer.

Mais peças em movimento. Há uma tabela, um processo relay e um atraso a monitorar. Quando algo quebra, existe um novo lugar para olhar. O relay é um componente que alguém tem de manter, e precisa de deploys, alertas e um runbook. O Capítulo 5 dá os detalhes.

Carga extra no banco de dados. Cada evento é um insert na mesma transação da gravação de negócio, e cada publicação é uma leitura e uma atualização. As transações ficam um pouco mais longas, e o volume de gravação sobe. Para a maioria dos serviços, o aumento é pequeno comparado ao resto do trabalho. Um serviço que grava volumes muito altos de eventos deve medir isso no próprio hardware, em vez de confiar no meu “pequeno”.

Latência adicional. O evento chega ao broker depois que a transação faz o commit e depois que o relay percebe a linha. Com o polling, o atraso depende do intervalo de polling. Com uma abordagem baseada em log, pode

ser menor, e o Capítulo 5 compara as duas. Em qualquer caso, o outbox não é a escolha para fluxos em que o evento precisa chegar ao broker em poucos milissegundos depois da requisição, e você deve conhecer o seu orçamento de latência antes de escolher.

Duplicados por design. Tratado acima. Se não for possível tornar um consumidor idempotente, o outbox não melhora nada para ele, e a equipe tem de decidir se esse consumidor deveria existir nessa forma.

A ordenação só é preservada em parte. O relay lê as linhas por `id`, e a ordem do `id` não é a ordem do commit, a menos que toda transação bloqueie a linha do agregado antes de inserir a linha do outbox. Esse bloqueio serializa os escritores de um agregado, o que é um custo por si só em agregados muito disputados. Relays paralelos pioram a ordem, e o bloqueio não ajuda entre agregados. A coluna `aggregate_version` dá aos consumidores algo para verificar, e custa um incremento de versão a cada gravação. O outbox lhe dá um lugar para impor a ordem, não a ordem de graça. O Capítulo 5 e o Capítulo 9 voltam a isso.

Crescimento e limpeza. As linhas publicadas se acumulam. Sem exclusão ou arquivamento, a tabela cresce até a consulta do relay ficar lenta. A retenção é uma decisão, e ela interage com a reconciliação, já que as linhas antigas também são a sua evidência do que foi publicado.

Só funciona para estado em um banco de dados que ofereça transações. Se o estado vive em um armazenamento sem transações de várias linhas, você precisa de outra abordagem, como um change stream específico do armazenamento ou um design com event sourcing. Um armazenamento que oferece atomicidade apenas dentro de um documento tem uma variante própria: manter os eventos pendentes dentro do documento do agregado e deixar o relay esvaziá-los de lá.

Quando não usar

Não acrescentaria um outbox quando:

- o evento for de melhor esforço e a sua perda for aceitável, como uma métrica ou uma dica de analytics de baixo valor
- o serviço não tiver estado durável próprio, de modo que não haja nada para fazer o commit junto com o evento

- o sistema for pequeno o bastante para que um único processo seja proprietário tanto do estado quanto dos seus consumidores, e uma chamada direta seja mais simples
- a equipe ainda não conseguir tornar os consumidores idempotentes, caso em que resolveria isso primeiro, já que o outbox aumenta o que está em jogo nessa lacuna

Nota: “Usamos um outbox” é uma afirmação sobre o caminho do produtor de um serviço. Ela não diz nada sobre se os consumidores estão seguros, se a ordenação se mantém ou se os eventos foram bem projetados. Trate-o como um padrão dentro de um conjunto, não como um selo de confiabilidade.

Erros comuns

- **Gravar a linha do outbox em uma transação separada.** É tentador porque um hook do framework ou uma abstração de event bus esconde a conexão. As duas gravações viram então um dual write entre duas tabelas do mesmo banco de dados, e a janela está de volta. Use uma conexão e uma transação, e teste o rollback.
- **Publicar a partir de um hook “after commit”.** É tentador porque ele só dispara para dados que passaram pelo commit e não precisa de tabela. É commit e depois publicação com a intenção na memória, então um crash entre os dois perde o evento. Grave a linha na transação e deixe o relay publicar.
- **Inserir a linha do outbox antes de bloquear o agregado.** O código funciona em todos os testes com um único escritor. Sob concorrência, a ordem do `id` e a ordem do commit divergem para um mesmo pedido, e o relay pode publicar os eventos dele invertidos. Bloqueie primeiro a linha do agregado, incremente a versão e só então insira.
- **Publicar a partir do caminho da requisição e manter o outbox como backup.** Parece cinto e suspensório. Produz dois publicadores, duplicados com tempos diferentes e uma ordenação sobre a qual você não consegue raciocinar. Deixe o relay ser o único publicador.
- **Gerar o `event_id` no relay.** Parece inofensivo, já que o relay é quem monta a mensagem. Uma republicação passa então a ter um id novo, e

os consumidores não conseguem reconhecê-la como duplicado. Gere o id quando a linha for gravada.

- **Marcar a linha como publicada antes de o broker confirmar.** Reduz os duplicados e parece mais arrumado. Converte um duplicado em um evento perdido, que é a pior falha. Marque após a confirmação e aceite o duplicado.
- **Montar o payload depois do commit.** Rerler o agregado para preencher o evento parece manter o código produtor simples. Outra transação pode alterar os dados nesse intervalo, e o evento deixa de corresponder ao fato. Monte o payload a partir do que a transação gravou.
- **Deixar alguns caminhos de código contornarem o outbox.** Um script de migração, uma ferramenta administrativa ou uma correção de SQL feita à mão muda o estado e não emite nada. Faça toda mudança de estado relevante passar pela mesma função, e revise os reparos manuais quanto aos eventos que deveriam ter produzido.
- **Esquecer a limpeza.** A tabela funciona por meses, e então a consulta do relay fica lenta na pior hora. Decida a retenção quando criar a tabela.
- **Tratar o outbox como o fim da história da confiabilidade.** Ele fecha uma janela. Os consumidores ainda precisam de idempotência, retries e um lugar para as mensagens que nunca têm sucesso.

Pontos-chave

- Um dual write não pode ser tornado seguro escolhendo uma ordem melhor: commit e depois publicação perde eventos, publicação e depois commit os inventa, e uma publicação dentro de um bloco de transação continua sendo a segunda.
- O two-phase commit pode fechar a janela em princípio, mas, para o banco de dados de um serviço e um broker, em geral custa mais em bloqueios, operação do coordenador e disponibilidade do que o problema vale.
- O transactional outbox transforma o dual write em uma transação local mais um movimento repetível, gravando o evento como uma linha na mesma transação da mudança de estado.

- A linha do outbox carrega o envelope completo do evento, um `event_id` estável criado no momento da gravação para que toda republicação possa ser reconhecida, e uma `aggregate_version` incrementada sob o bloqueio do agregado, porque a ordem do `id` não é a ordem do commit.
- O outbox garante a atomicidade do estado e da intenção, a intenção durável e a publicação at-least-once, e não garante que uma mensagem seja publicada exatamente uma vez.
- Os duplicados são o preço da garantia, então um outbox sem consumidores idempotentes troca uma falha por outra.
- Listen-to-yourself e event sourcing também eliminam o dual write, mas mudam a forma como o serviço lê e armazena o seu estado, então os escolheria pelo que oferecem e não apenas para evitar um relay.

Publicar é só metade do movimento: algo precisa ler essas linhas e levá-las ao broker sem perdê-las, reordená-las nem inundá-lo. O Capítulo 5 trata do relay, do polling em comparação com o change data capture e de como executar o outbox em produção.

Capítulo 5 — Relays, polling e change data capture

Imagine uma quarta-feira de manhã na Parcelly. A equipe de Orders colocou o transactional outbox no ar há duas semanas. Todo evento `OrderPlaced` agora é gravado na tabela `outbox` na mesma transação local do próprio pedido, e a janela do dual write do Capítulo 4 está fechada. A equipe considera o problema resolvido e passa para o ticket seguinte.

O relay é um processo pequeno que acorda a cada segundo, lê as linhas com `id` maior que o da última que viu, publica essas linhas e guarda o novo `id` mais alto. Ele roda como uma única instância, e ninguém pensou nele desde o dia em que foi implantado.

Então um deploy de rotina substitui o relay no pior momento. A instância antiga publicou no broker um lote de 200 linhas, mas ainda não salvou a nova posição. A instância nova começa da última posição salva, lê as mesmas 200 linhas e as publica de novo. Inventory reserva o estoque uma segunda vez para pedidos que já tinha tratado. Uma semana depois, um job de limpeza é acrescentado e apaga em silêncio linhas que o relay ainda não leu. No mês seguinte, uma transação demorada faz o commit tarde e suas linhas nunca saem da tabela.

Nenhuma dessas falhas está no padrão outbox em si. Todas estão na parte que leva as linhas ao broker. Como executar essa parte de modo que ela demore a falhar, se recupere depressa e seja honesta quanto aos duplicados que ainda pode produzir?

O problema e por que ele é difícil

O transactional outbox lhe dá uma garantia: se o pedido passou pelo commit, a intenção de publicar o seu evento passou com ele. Ele não diz nada sobre a segunda metade da jornada. Alguém precisa ler as linhas, entregá-

las ao broker e lembrar que o fez. Esse alguém é o **relay**, o processo que leva as linhas do outbox ao broker.

Um relay faz três coisas: encontra as linhas que precisam ser publicadas, publica-as e registra que o fez. São três passos separados, e o broker e o banco de dados não compartilham uma transação. O mesmo problema de dual write que removemos do produtor reaparece dentro do relay, com uma janela mais estreita.

As janelas de crash são poucas o bastante para serem listadas. A tabela é a semântica de falhas do relay: o que pode falhar, como você observaria a falha e qual garantia sobrevive a ela.

O crash acontece	Estado da linha	Resultado
Antes da publicação	Não publicada	A linha é retomada, o evento é publicado uma vez. Seguro.
Depois da publicação, antes da marcação	Não publicada, mas o evento está no broker	A linha é retomada, o evento é publicado duas vezes. Um duplicado.
Depois da marcação	Publicada	Nada a fazer. Seguro.

Nenhuma ordenação dos três passos fecha a linha do meio. Se o relay marca primeiro e publica depois, um crash entre os dois perde o evento, que é a falha que o outbox existe para impedir. Então o relay publica primeiro e marca depois, e aceita que um crash entre os dois produza um duplicado. É a mesma troca que o capítulo sobre o outbox fez: entrega at-least-once, com o tratamento de duplicados empurrado para os consumidores (Capítulo 7).

A primeira pergunta de design, portanto, não é como evitar duplicados. É como manter a janela pequena, como garantir que as linhas saiam em uma ordem sensata e como perceber o relay ficando para trás antes que um cliente perceba. A segunda pergunta de design é a escolha entre dois mecanismos para ler as linhas: fazer polling da tabela ou ler o próprio log de transações do banco de dados. Cada um leva a dificuldade para outro lugar.

O relay com polling

Um **relay com polling** consulta a tabela de outbox em um intervalo, publica o que encontra e marca essas linhas como publicadas. É o mecanismo mais simples que funciona, e começaria por ele, a menos que os números digam o contrário.

O loop básico

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

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

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

A consulta por trás de `selectUnpublished` é onde a maioria dos relays com polling erra, então vale olhá-la de perto. A forma óbvia é “dê-me as linhas com `id` maior que o da última que tratei”. Parece eficiente, e não é segura.

Por que um cursor sobre o id não é seguro

Os ids das linhas costumam ser atribuídos quando o insert roda, não quando a transação faz o commit. Imagine duas transações de Orders. A transação A insere uma linha de outbox com o id 101 e depois gasta algum tempo chamando uma função lenta antes de fazer o commit. A transação B insere o id 102 e faz o commit imediatamente. O relay faz o poll, vê o 102, publica-o e move o cursor para 102. Então A faz o commit. A linha 101 agora está atrás do cursor e nunca será lida.

A falha é silenciosa. Nada dá erro, nenhuma linha de log aparece e o evento nunca é publicado. É a mesma perda que o outbox foi construído para remover, reintroduzida por uma otimização.

A correção que prefiro é parar de lembrar uma posição e consultar pelo estado: selecionar as linhas em que `published_at` é nulo, ordenadas por `id`. Uma linha que faz o commit tarde continua não publicada quando o poll seguinte chega, então é encontrada. O custo é um índice sobre as linhas não publicadas e uma tabela que precisa continuar pequena o bastante para que essa consulta seja barata. Um índice parcial que cubra apenas as linhas não publicadas mantém a consulta rápida, qualquer que seja o histórico que a tabela guarde, se o seu banco de dados oferecer um. Se não oferecer, o trabalho de retenção mais adiante neste capítulo importa mais. Um índice parcial não remove todo o custo: nos bancos de dados que mantêm versões antigas das linhas, cada linha é inserida, atualizada e depois apagada, e essa rotatividade deixa linhas mortas para o motor limpar.

A leitura baseada em estado corrige a perda. Não corrige a ordem, que é uma questão à parte e que a seção sobre a preservação da ordem responde.

Lotes e o intervalo de polling

Duas configurações moldam o comportamento do relay: quantas linhas ele pega por poll e quanto tempo dorme quando não há nada a fazer.

Um lote maior dilui o custo da consulta e da atualização de marcação, e melhora o throughput. Também alarga a janela de crash: um relay que morre na metade de um lote de 500 deixa até 500 eventos para republicar. Um lote menor republica menos depois de um crash e, com a reserva de linhas descrita adiante, mantém bloqueios por menos tempo. Digamos que de 50 a 200 linhas seja um primeiro palpite razoável; meça antes de ajustar.

O intervalo de polling é um piso para a latência. Com um intervalo de um segundo, um evento pode esperar quase um segundo antes mesmo de o relay olhar para ele. Encurtar o intervalo reduz a latência e aumenta o número de consultas vazias que o banco de dados precisa atender. Para a Parcelly, um ou dois segundos de atraso entre o commit de `OrderPlaced` e a sua publicação são invisíveis para os compradores. Para um fluxo em que uma pessoa espera diante de uma tela, talvez não sejam.

Marcar depois de publicar

O passo de marcação merece cuidado próprio. Publique cada linha, colete o ack do broker e marque apenas as linhas que o broker confirmou. Não marque uma linha cuja publicação não foi confirmada, mesmo que a chamada tenha retornado sem erro, porque muitas bibliotecas cliente enviam de forma assíncrona e relatam a falha depois.

```
function publishAked(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
```

Essa função nunca toca a tabela. Ela devolve os ids que o broker confirmou, e todo relay neste capítulo marca exatamente esses ids e nenhum outro. Uma linha que falhou, e toda linha depois dela no lote, continua não publicada e é lida de novo no poll seguinte. A publicação e a marcação são mantidas separadas de propósito, para que a marcação possa ser protegida por um bloqueio ou um lease no único lugar em que acontece. Cada evento vai para o topic do seu tipo de agregado, com `partition_key` como chave da mensagem, o mesmo roteamento que o Capítulo 4 usa.

O `break` é uma escolha deliberada, e diz respeito à ordenação, que reaparece adiante. Pular uma linha que falhou e publicar as linhas seguintes é mais rápido, e pode reordenar eventos de um mesmo agregado. Parar é mais lento e seguro. Pararia, e deixaria o monitoramento me dizer que o relay está travado. Este esboço interrompe o lote inteiro por simplicidade; bloquear apenas a chave afetada é o refinamento descrito adiante, em preservando a ordem.

Um relay ativo ou vários

A topologia mais simples é uma única instância de relay fazendo todo o trabalho. O seu comportamento de ordenação é mais fácil de raciocinar, porque ele lê em ordem de `id` e publica nessa ordem, com a ressalva sobre `id` e ordem de commit na seção sobre preservar a ordem, mais adiante. Sua fraqueza é a disponibilidade: se essa instância morre, nada é publicado até que outra suba. Para muitos sistemas, um reinício em poucos segundos é aceitável, e um único relay com um supervisor que o reinicia é a resposta certa.

Quando um relay não dá conta, ou quando uma lacuna na publicação é inaceitável, você executa vários. Surge então um problema novo: dois relays não podem publicar a mesma linha ao mesmo tempo. A ferramenta padrão é um bloqueio de linha com semântica de pular. Muitos bancos de dados relacionais oferecem uma forma de selecionar linhas para atualização pulando as linhas que outra transação já bloqueou, comumente descrita como bloqueio no estilo `SKIP LOCKED`. O relay reserva um lote dentro de uma transação e os outros relays passam por cima dessas linhas.

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

Isso funciona, e traz custos que muitas vezes passam despercebidos. Primeiro, a transação permanece aberta enquanto o relay conversa com o broker, então um broker lento significa bloqueios mantidos por muito tempo e uma pilha crescente de transações abertas. Nos bancos de dados que mantêm versões antigas das linhas para leitores concorrentes, uma transação aberta por muito tempo também segura a limpeza das linhas mortas, o que prejudica justamente a tabela que mais rotaciona. Segundo, se a transação é abortada depois da publicação, por uma conexão perdida ou uma marcação que falhou, toda linha que ela publicou é publicada de novo. Uma alternativa mais curta é uma breve transação de reserva que carimba as linhas com um horário de reserva, uma publicação fora de qualquer

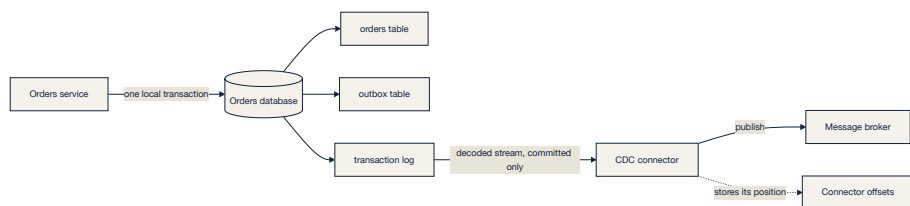
transação e uma transação de marcação separada, com um timeout de reserva para que as linhas de um relay morto voltem a poder ser reservadas. Isso custa mais um estado para raciocinar. Terceiro, relays paralelos perdem a ordenação global. O relay A pode reservar as linhas de 1 a 100 enquanto o relay B reserva as de 101 a 200, e B pode terminar primeiro. Para eventos sobre pedidos diferentes, isso não importa. Para dois eventos sobre o mesmo pedido, importa.

O remédio é particionar o trabalho por `partition_key` em vez de por linha. Atribua a cada relay um conjunto de chaves, por exemplo aplicando um hash de `partition_key` em um número fixo de buckets e dando cada bucket a exatamente um relay por vez. Dentro de um bucket, as linhas são tratadas em ordem por um worker, e entre buckets rodam em paralelo. É a mesma ideia das partições em um broker baseado em log, aplicada um passo antes. A palavra “exatamente” precisa do lease da seção de failover adiante, e a passagem de um bucket durante um rebalanceamento é o momento em que a ordem por chave corre mais risco, porque o proprietário antigo e o novo podem publicar juntos para a mesma chave. O Capítulo 9 explica por que a ordem por chave é a ordem que se consegue manter de forma realista.

Change data capture

O segundo mecanismo não consulta a tabela de outbox. O **change data capture (CDC)** lê o próprio log de transações de um banco de dados e transforma as mudanças de linha com commit em um stream de registros de mudança. Um conector de CDC, do qual o Debezium é o exemplo conhecido, se conecta a esse log, percebe que uma linha foi inserida em `outbox` e publica uma mensagem no broker.

A tabela mantém o seu papel, porque a garantia transacional continua vindo de gravá-la na mesma transação da mudança de estado. O que muda é quem a lê. Nenhum processo da aplicação faz polling. O banco de dados já registra toda mudança com commit em um log ordenado, para a sua própria recuperação e replicação, e o conector lê esse log do modo como uma réplica leria.



Observe o caminho do log de transações até o broker. O stream de mudanças que o banco de dados expõe aos conectores (o PostgreSQL o chama de decodificação lógica, o MySQL expõe o seu log binário) contém apenas transações com commit, e elas aparecem na ordem do commit. Isso remove por inteiro o problema do commit tardio da seção sobre polling: o log já é uma ordem total de commits, então uma linha que faz o commit tarde aparece tarde, no lugar certo.

Como ele se comporta

Algumas propriedades decorrem do design.

A ordenação segue a ordem do commit. Os eventos aparecem na ordem em que as transações fizeram o commit. Um conector que publica com a `partition_key` da linha como chave da mensagem mantém a ordem por chave em um broker baseado em log, desde que o produtor dentro dele esteja configurado de modo que um retry não possa reordenar requisições em andamento ao mesmo tempo. Verifique as configurações do seu cliente; a mesma ressalva vale para um relay com polling que publica de forma assíncrona.

A latência é baixa e estável. Não há intervalo de polling. O conector vê uma mudança logo depois do commit, então o atraso costuma ser pequeno e não depende da frequência com que algo acorda.

O conector acompanha a sua própria posição. Ele registra até onde leu no log e, depois de um reinício, retoma dali. Os conectores costumam salvar a posição periodicamente, e não após cada evento, então um crash pode republicar todo evento desde a última posição salva. O CDC é at-least-once, como o polling. A janela de duplicados mudou de lugar, não desapareceu.

A flag de publicado é opcional. Como o conector não precisa encontrar linhas não publicadas, a tabela não precisa de uma coluna `published_at` por causa do relay. Muitas equipes que usam CDC apagam cada linha do outbox logo depois de inseri-la. O insert já foi registrado no log, então o conector ainda o vê. A tabela fica minúscula, e o problema da limpeza em grande parte desaparece.

O envelope precisa ser moldado. Um registro de mudança bruto descreve uma linha, não um evento. O conector, ou um passo de transformação, precisa mapear as colunas do outbox para o envelope padrão: `event_id`, `event_type`, `partition_key` como chave da mensagem, `payload` como corpo. A maioria dos conectores oferece suporte a roteamento e mapeamento de campos exatamente para esse fim. Mantenha as colunas do outbox alinhadas com o envelope, para que o mapeamento seja mecânico.

O que ele custa

O CDC não é de graça, e os custos são operacionais.

Agora você opera um conector, em geral dentro de um framework de conectores, mais a infraestrutura de broker de que ele depende. Alguém precisa aprender como ele falha.

O banco de dados precisa expor o seu log ao conector, o que normalmente significa habilitar um nível de logging específico e criar uma posição de replicação que o banco de dados retém em nome do conector. Se o conector para e ninguém percebe, o banco de dados mantém segmentos de log que não pode descartar, e o disco enche. É uma forma de falha conhecida na família do CDC, e os detalhes variam conforme o banco de dados. Trataria “o conector está fora do ar” como um page, porque o dano não se limita ao broker.

As mudanças de esquema na tabela de outbox, e em qualquer tabela que o conector observe, precisam ser planejadas. Uma renomeação de coluna ou uma mudança de tipo pode quebrar o mapeamento.

A configuração inicial, os snapshots e os reinícios do conector depois de quedas longas também merecem ensaio. O comportamento depende do produto e do banco de dados, e não o suporia a partir de um diagrama.

Nota: O CDC é uma família de técnicas, e os produtos diferem na forma de ler o log, na forma de armazenar posições e no que acontece depois de uma queda longa. Trate esta seção como a forma geral e leia a documentação do seu conector e do seu banco de dados para saber o comportamento exato das versões que você executa.

Quando o log some

“O disco enche” é metade de uma troca, e a outra metade é pior. O banco de dados ou mantém log para a posição do conector sem limite, o que enche o disco, ou limita o que mantém, ou expira os logs em um cronograma, o que protege o disco. Muitos bancos de dados deixam você escolher. Se você impõe um limite e o conector fica fora do ar por mais tempo que ele, a posição salva do conector aponta para um log que não existe mais. Os eventos na lacuna se perdem, ou o conector precisa recomeçar de um snapshot.

O que um novo snapshot faz com um outbox depende de uma decisão anterior. Se a tabela ainda guarda linhas, um snapshot reemite todas elas, de modo que cada linha retida vira um duplicado para todos os consumidores. Se, como é comum com CDC, as linhas foram apagadas logo após o insert, o snapshot não encontra nada, e os eventos na lacuna não podem ser recuperados da tabela de forma alguma. Isso é uma perda silenciosa: nenhum erro, nenhum duplicado, um pedido que nunca chegou a Inventory. A exclusão que manteve a tabela minúscula é a mesma exclusão que remove a evidência.

Um failover do primary do banco de dados acrescenta um terceiro caso. Se a posição do conector é transferida para o novo primary varia conforme o banco de dados e a versão, e descobriria isso ensaiando um failover, e não lendo um diagrama.

Três hábitos decorrem disso. Escolha o limite de retenção de propósito, como uma troca de disco contra o tempo de queda que você consegue suportar, e anote o número. Crie um alerta sobre a folga, isto é, quanto da janela de retenção o atraso do conector já consumiu, antes que ela acabe. E decida antes da queda como você recuperaria os eventos da lacuna, por

exemplo a partir de um job de reconciliação (Capítulo 13) que compare o estado dos pedidos com o que foi publicado. O Capítulo 14 trata da recuperação depois que o dano já foi feito.

Escolhendo entre polling e CDC

Nenhum dos mecanismos é melhor em abstrato. A tabela mostra onde eles diferem.

Aspecto	Relay com polling	Conector de CDC
Peças em movimento	Um processo pequeno que você mantém	Conector, seu framework, configurações de log do banco de dados
Latência	Limitada por baixo pelo intervalo de polling	Em geral baixa e estável
Fonte da ordenação	Ordem da consulta mais o seu próprio cuidado	Ordem do commit vinda do log
Transações que fazem commit tarde	Seguro com uma consulta baseada em estado	Tratado pelo log
Carga no banco de dados	Consultas repetidas, bloqueios de linha	Leituras de log, segmentos de log retidos
Duplicados após um crash	Possíveis, a janela é o lote não marcado	Possíveis, a janela é a posição não salva
Crescimento da tabela	Exige trabalho de retenção	As linhas podem ser apagadas de imediato
Falha que mais machuca	Atraso silencioso, linhas travadas	Conector parado, log retido
Habilidades necessárias	Habilidades comuns de aplicação	Internos do log do banco de dados, operação de conectores

Meu padrão para uma equipe que está começando é um relay com polling. São algumas dezenas de linhas de código, a equipe já sabe operar e depurar um serviço assim, e seus modos de falha são visíveis na tabela que ele consulta. Passaria para o CDC quando uma de três coisas se tornasse verdadeira: a latência do polling for alta demais e não puder ser reduzida sem martelar o banco de dados, o volume do outbox fizer das consultas de polling um custo mensurável, ou você já operar uma plataforma de CDC e alguém for responsável por ela.

Não adotaria o CDC só porque ele soa mais sério. Uma equipe que nunca operou um log de replicação e coloca nele o seu fluxo de pedidos trocou um problema que entendia por um que não entende. O inverso também vale: uma equipe que já executa CDC para outros fins ganha pouco escrevendo um relay com polling ao lado.

Operando o relay

Escolher o mecanismo é a primeira metade do trabalho. A segunda metade é o trabalho que o mantém saudável.

Retenção e limpeza

Um relay com polling deixa linhas publicadas para trás. Se nada for feito, a tabela cresce sem limite, a consulta de linhas não publicadas fica mais lenta e o banco de dados gasta esforço com linhas que ninguém lê. A limpeza precisa ser projetada, não acrescentada depois.

Duas escolhas importam: por quanto tempo manter as linhas publicadas e como apagá-las. Mantê-las por um tempo tem valor. Quando um consumidor relata um evento ausente, o outbox é o primeiro lugar em que você olha para saber se ele chegou a ser publicado. Digamos que alguns dias sejam um ponto de partida razoável. Quanto tempo é o certo depende da rapidez com que a sua equipe percebe eventos ausentes, o que é uma questão sobre pessoas e alertas, não sobre armazenamento.

```
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)
```

Apague em lotes pequenos para que o job não mantenha bloqueios por muito tempo nem inunde as réplicas. Se a tabela é grande, particionar por tempo e descartar partições inteiras é mais barato do que apagar linha a linha, onde o seu banco de dados permitir. Seja qual for a escolha, a condi-

ção de limpeza deve incluir `published_at is not null`. Uma limpeza que apaga só pela idade vai, no dia em que o relay ficar para trás, apagar eventos que nunca foram publicados. Essa é a falha do cenário de abertura, e é uma perda de dados, não um duplicado.

Aviso: Nunca limpe linhas do outbox só pela idade. Um relay travado por mais tempo que o seu período de retenção terá as linhas não publicadas apagadas, e esses eventos se perdem para sempre. Limpe apenas as linhas marcadas como publicadas e crie um alerta para as linhas não publicadas que envelhecem.

Com o CDC, a limpeza é diferente, mas não ausente. As linhas podem ser apagadas logo após o insert, de modo que a tabela fica pequena. O que cresce agora é o log retido, e o que se deve observar é o atraso do conector em relação a ele. Apagar de imediato também significa que um novo snapshot não consegue recuperar nada, como explica a seção anterior sobre um log desaparecido, então pese a exclusão imediata contra uma retenção curta das linhas publicadas.

Monitorando o relay

Um relay que para é invisível por fora. Orders continua tendo sucesso, o outbox continua aceitando linhas, e todos os outros serviços ficam em silêncio sem eventos. Ninguém recebe um erro. É por isso que o relay precisa de sinais próprios, e colocaria três deles no lugar antes do lançamento.

1. **Idade da linha não publicada mais antiga.** Este é o melhor número isolado, porque mede o que os usuários experimentam: há quanto tempo o evento mais lento está esperando. Crie um alerta para quando ele exceder um limite ligado ao negócio, não à tecnologia. Se o e-mail de confirmação de um comprador deve chegar em um minuto, uma idade de cinco minutos para a linha não publicada mais antiga é um incidente.
2. **Contagem de linhas não publicadas.** Um backlog crescente diz que o relay é mais lento que o produtor, mesmo quando a linha mais antiga ainda é jovem.
3. **Taxa de erros de publicação e duração do lote.** Estes explicam os dois primeiros. Um relay que faz retry da mesma linha a cada poll

mostrará uma contagem de erros estável e uma idade crescente.

Para um conector de CDC, as mesmas perguntas se aplicam em outras unidades: a que distância da ponta do log está a posição do conector, se ele está rodando e quanto log o banco de dados está retendo para ele. O Capítulo 13 volta a esses sinais e à reconciliação, que pega o que as métricas de atraso deixam passar.

Failover e publicação dupla

Um relay precisa sobreviver aos próprios reinícios, e isso inclui o caso em que duas instâncias rodam por um instante ao mesmo tempo. Deploys, partições de rede que fazem a instância antiga parecer morta e orquestradores que reiniciam um processo antes de a instância antiga ter saído produzem essa sobreposição.

Para um relay de um único ativo, use um lease: uma linha ou um bloqueio no banco de dados que o relay ativo renova em um intervalo curto e que expira se quem o detém parar. Um standby assume depois da expiração. O detalhe que pega as pessoas é que o antigo detentor pode não saber que o seu lease expirou. Ele poderia ser pausado por uma coleta de lixo longa ou por uma rede travada, acordar e publicar um lote que não lhe pertence mais.

```

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

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

        if result is failure:
            break
        ackedIds.append(row.id)
    return ackedIds

```

O token crescente é um fencing token, e a ideia só funciona onde o recurso protegido o verifica. O banco de dados pode verificá-lo: um commit que carrega um token obsoleto pode ser feito falhar, e o Capítulo 7 usa exatamente isso para consumidores e APIs. O broker em geral não consegue, então o token não impede que um relay obsoleto publique. A marcação é outra questão. O detentor obsoleto marca apenas as linhas que o broker confirmou, então as suas marcações são verdadeiras, e desfazê-las só criaria mais duplicados. O que um lease deve limitar é a publicação concorrente, porque dois detentores publicando juntos geram duplicados e podem colocar uma linha mais antiga de uma chave depois de uma mais nova.

Quatro coisas a limitam. Mantenha o TTL do lease bem acima da duração do pior caso de um lote e mantenha os lotes pequenos, porque o lease é renovado uma vez por loop e um broker lento sozinho pode superá-lo. Verifique o lease restante antes de cada publicação, como o esboço faz. Onde um broker baseado em log oferecer suporte a fencing do produtor por

época, use-o, para que o próprio broker rejeite o antigo detentor. E, quando o antigo detentor acordar, deixe que ele descubra isso perguntando ao banco de dados, em uma transação que bloqueie a linha do lease, antes de publicar qualquer outra coisa. Uma verificação que não mantém o bloqueio é um check-then-act, e falha do mesmo modo.

Esse é o limite honesto. O failover pode produzir uma publicação dupla, e nenhum esquema de lease a remove. O que um lease lhe compra é que a janela seja curta e rara. O que remove o dano é um consumidor que trate a segunda cópia como o mesmo evento, usando o `event_id` que é estável nas duas publicações. O relay deve, portanto, tomar o `event_id` da linha do outbox e nunca gerar um novo quando publica. Um relay que cunha um identificador novo a cada envio transforma um duplicado inofensivo em dois eventos aparentemente distintos. Alguns brokers aceitam um id de deduplicação fornecido pelo produtor e descartam repetições dentro de uma janela de tempo; onde o seu aceitar, defina-o como `event_id`. Isso estreita a janela de graça e não substitui o inbox do consumidor. O Capítulo 7 trata do lado do consumo.

Preservando a ordem

O outbox preserva a ordem dos eventos apenas até onde o relay a preserva, e o relay só pode preservar a ordem que a tabela guarda. O Capítulo 4 mostrou que a ordem do `id` não é a ordem do commit: uma transação pode pegar um `id`, esperar por um bloqueio e fazer o commit depois de uma transação que pegou um `id` posterior. O `id` é a ordem da inserção, não a do commit.

Duas coisas decorrem disso para um relay com polling. Se um `id` menor de uma chave faz o commit depois que um maior da mesma chave já foi publicado, os dois saem invertidos, e a consulta baseada em estado que salvou a linha tardia é o que a publica tarde. Se os dois se tornam visíveis em um mesmo poll com os ids invertidos, saem invertidos em um mesmo lote. Um relay que publica “em ordem de `id`” portanto não entrega os eventos por agregado na ordem em que o agregado mudou, a menos que uma de duas condições valha.

A primeira é que os produtores bloqueiem a linha do agregado antes de inserir a linha do outbox, de modo que as transações da mesma chave se serializem e seus ids sigam a ordem de commit (Capítulo 4). A segunda é

que a ordem seja carregada nos dados: a coluna `aggregate_version`, que os consumidores comparam e que o Capítulo 9 usa. Faria as duas. O bloqueio é um seguro barato para o relay, e a versão é o que sobrevive a tudo o que vem depois do relay. Um conector de CDC evita o primeiro problema porque o log está na ordem do commit, e ainda assim não ajuda entre partições, então a versão continua útil.

Dito isso, as escolhas do próprio relay ainda importam. Cada uma destas pode quebrar a ordem que ele tem:

- Pular uma linha que falhou e publicar linhas posteriores da mesma chave.
- Executar relays paralelos sem particionar por chave.
- Fazer o retry de uma publicação que falhou fora de banda, de modo que uma linha posterior chegue primeiro.
- Usar chaves de mensagem diferentes para eventos de um mesmo agregado.

O padrão seguro é o head-of-line blocking por chave: se uma linha de `order-81724` não pode ser publicada, pare de publicar as linhas posteriores de `order-81724` e deixe as linhas de outras chaves seguirem. Isso custa latência para o único agregado bloqueado e protege a sua sequência. Se o problema é uma linha que nunca pode ser publicada, como um payload que o serializador rejeita, alguém precisa repará-la, e o monitoramento acima é como essa pessoa descobre.

Seja claro quanto ao que o esboço anterior faz. Com um `break` sobre o lote inteiro e uma consulta que traz as mais antigas primeiro, uma linha impuplicável é devolvida primeiro em todo poll e paralisa todas as chaves para sempre. O bloqueio por chave precisa de dois acréscimos, mostrados como um refinamento de `publishAcked` que substitui o seu `break`: o relay lembra quais chaves falharam neste lote, e uma linha que falha repetidamente é estacionada, de modo que a consulta deixa de devolvê-la e deixa de devolver também as linhas posteriores da sua chave.

```

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

```

O select então exclui qualquer chave que tenha uma linha estacionada, de modo que as linhas posteriores dessa chave esperam e as linhas de outras chaves fluem. O estacionamento é o momento em que a coluna `failed_at` do Capítulo 4 se faz necessária, junto com um contador `attempts`, para que uma falha transitória receba retry algumas vezes antes de a linha ser estacionada. Uma linha estacionada é aquela cujo `failed_at` está definido. Crie um alerta para cada linha estacionada, porque uma linha estacionada é um agregado bloqueado e uma pessoa precisa repará-la ou descartá-la. O custo são duas colunas, uma consulta mais difícil de indexar e um procedimento de reparo que alguém precisa escrever antes do primeiro incidente.

A ordem também se perde depois do relay, no consumidor e no broker. Um relay que faz a sua parte lhe dá uma sequência que vale a pena preservar, não a garantia de que ela sobrevive ao resto do caminho. O Capítulo 9 acompanha a ordem do broker até o handler.

Trade-offs e custos

Os dois mecanismos trocam simplicidade por alcance operacional, e os dois deixam uma janela de duplicados que você não consegue fechar.

Um relay com polling custa carga no banco de dados e latência em troca de permanecer dentro de um código que a sua equipe já entende. Seus pontos fracos são a consulta de linhas não publicadas em uma tabela que

crece, o comportamento dos bloqueios quando vários relays rodam e o atraso silencioso quando ninguém observa a linha mais antiga. Se o seu sistema publica em um ritmo modesto e um segundo de atraso é aceitável, o relay com polling é difícil de superar em custo de propriedade.

O CDC custa um componente novo e habilidades novas em troca de menor latência, leitura na ordem do commit e uma tabela que continua pequena. Seus pontos fracos são um conector parado que faz o banco de dados reter log, ou um log limitado que expira sob um conector parado e perde eventos, mudanças de esquema que quebram mapeamentos e procedimentos de recuperação que raramente são ensaiados. Se você não tem ninguém que vá ser responsável pelo conector, não o adotaria, por mais atraente que o diagrama de arquitetura pareça.

Quando não usar um relay de outbox: se um evento pode ser perdido sem consequência, como uma notificação informativa em que uma mensagem perdida não custa nada, uma publicação direta depois do commit pode ser aceitável e mais barata. Se os seus eventos precisam refletir o estado em um banco de dados que não oferece uma transação local entre o estado e o outbox, o relay não consegue reparar isso, e você precisa de um design diferente.

Dois custos são comuns aos dois. Nenhum dos dois lhe dá publicação exactly-once: o duplicado após um crash entre a publicação e a marcação, ou entre a publicação e o salvamento da posição, faz parte do design. E nenhum dos dois ajuda se o relay e o produtor discordam sobre o envelope, então mantenha as colunas do outbox e os campos do envelope em sincronia.

Erros comuns

- **Usar um cursor sobre o id em vez de uma consulta baseada em estado.** É tentador porque evita varrer em busca de linhas não publicadas. As transações que fazem commit tarde ficam então atrás do cursor e nunca são publicadas. Consulte por `published_at is null`, ou use um mecanismo baseado em log que já ordena pelo commit.
- **Marcar as linhas antes de o broker confirmar.** Marcar primeiro parece remover duplicados. Converte um crash em um evento perdido. Publique, aguarde o ack, depois marque.

- **Gerar um novo identificador de evento no relay.** Parece inofensivo, já que o relay é quem envia. O inbox do consumidor deixa então de reconhecer um evento republicado como o mesmo. Copie o `event_id` da linha.
- **Limpar só pela idade.** Mantém a tabela pequena e passa em todo teste em que o relay está saudável. Apaga eventos não publicados quando o relay não está. Limpe apenas as linhas publicadas e crie um alerta para as não publicadas antigas.
- **Pular uma linha que falhou para manter o throughput.** As linhas posteriores do mesmo agregado passam então à frente dela. Bloqueie por chave em vez disso, estacione a linha que continua falhando e corrija a causa.
- **Acreditar que a ordem do id é a ordem do commit.** Vale em todo teste com um único escritor. Com escritores concorrentes, um `id` menor pode fazer o commit depois, e o relay publica o par invertido. Bloqueie a linha do agregado antes do insert no outbox e carregue `aggregate_version`.
- **Confiar que o lease faz o fencing do broker.** Um relay obsoleto que acorda depois de o lease ter expirado ainda publica, porque o broker não verifica o token. Mantenha os lotes dentro do TTL, verifique o lease antes de cada publicação e deixe os consumidores absorverem o resto.
- **Executar CDC com um limite de retenção que ninguém escolheu.** O padrão parece um detalhe até que o conector fique fora do ar durante um fim de semana e a posição se perca. Escolha o limite, crie um alerta sobre a folga e saiba o que um novo snapshot faz com o seu outbox.
- **Executar vários relays sem particionar por chave.** Os bloqueios de linha evitam trabalho em dobro e não dizem nada sobre a ordem. Atribua buckets de chaves a workers únicos.
- **Não monitorar o relay.** O outbox torna o produtor confiável, o que faz da falha do relay a única que sobra, e um relay que parou tem exatamente a aparência de um sistema quieto. Crie um alerta sobre a idade da linha não publicada mais antiga.
- **Tratar a queda de um conector como um problema do broker.** No CDC, um conector parado faz o banco de dados reter o seu log. Envie

um page para essa queda e observe o disco, não só o atraso.

Pontos-chave

- Um relay faz três coisas separadas, encontrar, publicar e marcar, e o broker e o banco de dados não compartilham uma transação, de modo que um crash entre a publicação e a marcação sempre deixa um duplicado possível.
- Publique primeiro e marque depois: a outra ordem transforma um crash em um evento perdido, que é a falha que o outbox existe para remover.
- Um relay com polling deve selecionar as linhas não publicadas pelo estado, não por um cursor de id crescente, porque as transações fazem o commit em uma ordem diferente daquela em que alocam ids, e mesmo assim a ordem por agregado só se mantém se os produtores bloquearem o agregado primeiro e carregarem uma `aggregate_version`.
- Para executar vários relays com polling, reserve as linhas com um bloqueio no estilo skip-locked e particione por `partition_key`, de modo que as linhas de um agregado sejam sempre tratadas em ordem por um único worker.
- O change data capture lê as mudanças com commit do log do banco de dados na ordem do commit e elimina o polling e o problema do commit tardio, ao preço de um conector, de uma troca de retenção de log entre disco cheio e eventos perdidos e de habilidades que a sua equipe precisa ter.
- Comece com um relay com polling, a menos que a latência, a carga ou uma plataforma de CDC existente digam o contrário, e trate a escolha como uma troca de simplicidade por alcance operacional.
- Monitore a idade da linha não publicada mais antiga, limpe apenas as linhas publicadas, estacione as linhas que nunca são publicadas, copie o `event_id` da linha do outbox e espere duplicados após um failover, porque um lease limita a publicação concorrente e só um consumidor idempotente torna inofensivo o resto.

O relay agora entrega cada evento pelo menos uma vez, em uma ordem que vale a pena manter e, às vezes, duas vezes. O próximo problema está do outro lado do broker: o que um consumidor deve fazer quando o seu

trabalho falha, tema de que o Capítulo 6 trata com os retries.

PARTE III

Consumir com confiabilidade

Capítulo 6 — Retries bem-feitos

Imagine uma manhã de terça-feira na Parcelly, no início de uma promoção de vendedores. Orders envia um comando `CapturePayment`, e Payments o repassa ao provedor de pagamento externo. O provedor está com dificuldades e responde devagar. As chamadas de Payments começam a estourar o timeout, então Payments faz a mensagem falhar e o broker a entrega de novo. Todas as instâncias de Payments fazem a mesma coisa, no mesmo instante, com a mesma espera fixa de um segundo.

Em um minuto, o provedor está recebendo não a carga normal da promoção, mas a carga normal mais todas as chamadas que falharam voltando, de novo e de novo, em ondas sincronizadas. A lentidão do provedor, que poderia ter durado alguns segundos, agora dura enquanto os retries continuarem chegando. Enquanto isso, os consumidores de Payments estão ocupados refazendo mensagens que não vão ter sucesso, e as mensagens saudáveis ficam na fila atrás delas. Em algum ponto dessa pilha, algumas capturas que na verdade tinham dado certo no provedor estão sendo enviadas outra vez.

Ninguém escreveu uma linha de código ruim aqui. Cada instância fez a coisa sensata: falhou, esperou, tentou de novo. O dano veio de mil decisões sensatas que ninguém coordenou.

Este capítulo trata de fazer do retry uma decisão, e não um reflexo. A pergunta que ele responde é esta: quando uma chamada falha, como você decide se deve tentar de novo, quando, quantas vezes e em qual camada, de modo que o retry ajude o sistema a se recuperar em vez de impedi-lo?

O problema e por que ele é difícil

Um retry é a promessa de tentar de novo, não a promessa de ter sucesso. Neste capítulo, uma **tentativa** é qualquer tentativa feita, de modo que o total de tentativas é igual a um mais o número de retries. A abordagem

óbvia, “envolver a chamada em um loop e fazer retry algumas vezes”, falha por quatro motivos distintos, e eles se acumulam.

Primeiro, **nem toda falha vale um retry**. Uma requisição rejeitada porque o cartão foi recusado será rejeitada também na décima tentativa. Fazer retry dela desperdiça capacidade e atrasa o momento em que alguém percebe o problema real. O loop de retry precisa saber que tipo de falha tem diante de si, e a maioria dos loops não sabe.

Segundo, **os retries somam carga no pior momento**. Uma dependência falha quando está sobrecarregada, degradada ou reiniciando. Os retries são requisições extras apontadas exatamente para essa dependência. Se cada chamador fizer três retries, o que dá quatro tentativas no total, um serviço em dificuldade recebe até quatro vezes o tráfego normal justamente quando menos pode suportá-lo.

Terceiro, **os retries se sincronizam**. Chamadores que falham juntos fazem retry juntos. Um atraso fixo transforma uma falha isolada em um pico periódico, porque todos os chamadores acordam no mesmo instante. Esse é o thundering herd, e é o motivo pelo qual o atraso precisa de aleatoriedade e não apenas de duração.

Quarto, **um retry repete uma ação cujo primeiro resultado pode ser desconhecido**. O Capítulo 3 mostrou que um timeout não diz se a requisição foi processada. Fazer retry depois de um timeout, portanto, corre o risco de fazer o trabalho duas vezes. Sem idempotência (Capítulo 7), um retry é um duplicado esperando para acontecer.

Todo o resto deste capítulo é uma forma de responder a um desses quatro problemas. A classificação responde ao primeiro. O backoff com jitter e o retry budget respondem ao segundo e ao terceiro. A idempotência, que o Capítulo 7 constrói, responde ao quarto, e a seção sobre a sua ausência, mais adiante, explica por que ela não pode ser dispensada.

Classificação de falhas

Antes de decidir como fazer retry, decida se deve fazer. O Capítulo 1 deu três perguntas para qualquer junção: o que pode falhar, como você observaria a falha e qual garantia sobrevive a ela. Juntas, as respostas formam a semântica de uma falha, e uma decisão de retry é essa semântica posta

para trabalhar. Considero útil classificar toda falha em uma de três classes, conforme o que ela diz sobre a possibilidade de uma nova tentativa ajudar, e depois acrescentar um quarto sinal que não diz nada sobre a mensagem.

Falhas transitórias

Uma **falha transitória** é aquela cuja causa deve desaparecer sozinha. Uma conexão reiniciada, um timeout, um `503 Service Unavailable`, uma vítima de deadlock no banco de dados, uma resposta de throttling, uma eleição de líder no broker. A requisição não estava errada. O mundo esteve brevemente no caminho.

Fazer retry é a resposta certa, com um atraso, porque é o atraso que deixa a causa se dissipar.

Falhas permanentes

Uma **falha permanente** é aquela que vai se repetir em toda tentativa, porque a causa está na requisição ou nos dados. Um payload que não passa na validação, uma referência a um registro que continua ausente muito depois de quando deveria ter chegado (um registro ausente pode ser transitório em um sistema orientado a eventos, porque o evento que o cria pode chegar atrasado; o Capítulo 9 trata desse caso), um `400 Bad Request`, um cartão recusado por saldo insuficiente, um evento cuja versão de esquema o consumidor não entende.

Fazer retry é esforço desperdiçado e às vezes prejudicial. A resposta correta é parar, registrar o motivo e enviar a mensagem para onde uma pessoa ou um job de reparo possa tratá-la. O Capítulo 8 trata de para onde vão as falhas permanentes. Observe em especial a recusa do cartão: para Payments, ela não é um erro. É um resultado de negócio esperado, que se torna um evento `PaymentFailed`, e nenhum retry está envolvido.

Falhas desconhecidas

Uma **falha desconhecida** é aquela cuja classe você não consegue determinar. Um timeout é o caso clássico, e também o é uma exceção inesperada com uma mensagem que ninguém previu. O timeout pode ter sido transitório. A exceção pode ser um bug no consumidor que falhará em toda tentativa.

Desconhecida é uma classe de verdade, não uma lacuna na taxonomia, e o tratamento que você dá a ela é uma decisão de política. Meu padrão é tratar falhas desconhecidas como transitórias, mas com um limite mais estrito que o das transitórias conhecidas, porque o custo de fazer, por engano, um retry de uma falha desconhecida são algumas tentativas desperdiçadas, enquanto o custo de descartá-la por engano é trabalho perdido. Quando as tentativas de todas as camadas se esgotam, uma falha desconhecida é tratada como permanente: vai para as dead letters com todo o seu histórico.

Quando o resultado de uma tentativa é desconhecido, pergunte antes de agir de novo. Antes de um retry, e com mais razão antes de uma compensação, consulte o provedor sobre o status da operação pela chave de idempotência. Se a captura aconteceu, o passo seguinte correto é um reembolso, não uma anulação. Se não aconteceu, um retry ou uma anulação é seguro. Adivinhar em qualquer direção é como um comprador acaba cobrado por um pedido que a Parcelly acredita ter cancelado.

Adiada: a dependência está sabidamente fora do ar

O quarto sinal não é um veredicto sobre a mensagem. Quando um circuit breaker (descrito adiante) está aberto, a falha de uma chamada não diz nada sobre a requisição: a chamada nunca chegou à dependência. Chamo essa classe de **adiada**. Uma mensagem adiada não consome uma tentativa, não conta para nenhum limite e nunca vai para as dead letters por causa disso. O consumidor para de puxar mensagens, ou pausa a partição, e a mensagem espera onde está até que valha a pena chamar a dependência de novo.

A distinção importa porque a alternativa é aritmética. Um breaker que permanece aberto por 30 segundos, diante de um handler que permite três tentativas internas com um segundo de intervalo, deixa uma mensagem saudável queimar todo o seu orçamento nos primeiros três segundos, mais ou menos, e parar na DLQ por um problema que nunca teve.

Tornando a classificação explícita

A classificação precisa morar em algum lugar onde possa ser lida e revisada. Uma tabela que mapeia um sinal de falha para uma classe e um tratamento é um dos pequenos artefatos mais valiosos de um consumidor. Para

as chamadas de Payments ao provedor, uma versão ilustrativa é esta:

Sinal	Classe	Tratamento
Conexão reiniciada, falha de DNS	Transitória	Retry com backoff
Timeout em uma chamada de captura	Desconhecida	Retry com a mesma chave de idempotência, limite estrito
429, resposta de throttling	Transitória	Retry, respeitando a dica de espera do provedor, se houver
5xx do provedor	Transitória	Retry com backoff
400, falha de validação	Permanente	Não fazer retry, enviar para as dead letters
Cartão recusado	Permanente, resultado de negócio	Emitir <code>PaymentFailed</code> , não fazer retry
Erro não reconhecido	Desconhecida	Retry com limite estrito e, depois, dead letters
Circuit breaker aberto	Adiada	Nenhuma tentativa consumida; pausar o consumo e retomar quando o breaker permitir

Duas ressalvas. Primeiro, os códigos de status são dicas, não garantias; um provedor que devolve `500` para uma requisição malformada a classificou mal por você, e você terá de aprender isso com a experiência com esse provedor. Segundo, a tabela deve assumir como padrão a classe desconhecida, nunca a permanente ou a transitória, para qualquer sinal que ainda não tenha visto. Um erro novo que ninguém classificou não é evidência de nada.

O próprio classificador é pequeno:

```
// 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 com jitter

Decidido o retry, a pergunta seguinte é quando. Fazer retry imediatamente não dá tempo para uma causa transitória se dissipar. Fazer retry depois de um atraso fixo dá tempo, mas mantém todos os chamadores em sincronia.

O **exponential backoff** faz o atraso crescer a cada tentativa: digamos, um segundo, depois dois, depois quatro, até um teto. Os primeiros retries são rápidos, o que combina com falhas que passam num instante. Os retries posteriores são lentos, o que alivia uma dependência que está realmente fora do ar. O teto impede que o atraso cresça além do que o negócio consegue tolerar.

O **jitter** acrescenta aleatoriedade a cada atraso. Sem ele, chamadores que falharam no mesmo momento calculam os mesmos atrasos e fazem retry em sincronia, e o herd sobrevive intacto ao backoff. Com ele, os retries se espalham por um intervalo, e a dependência vê uma dispersão de requisições em vez de uma parede.

Há vários esquemas de jitter. Dois merecem ser conhecidos pelo nome. No **full jitter**, o atraso é um valor aleatório entre zero e o teto exponencial daquela tentativa. No **equal jitter**, metade do atraso é fixa e metade é aleatória, de modo que a espera mínima nunca fica perto de zero. O full jitter distribui melhor a carga, enquanto o equal jitter garante um piso para a espera. Começaria com o full jitter e só passaria ao equal jitter se a dependência sofresse com retries muito rápidos.


```

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

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

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

```

Esse loop está correto diante das falhas que trata, mas é incompleto de duas formas, que as próximas seções abordam. Ele não sabe quanto tempo o chamador ainda tem e não sabe quanto retry o sistema inteiro já está fazendo.

Se o provedor enviar uma dica sobre quando tentar de novo, como um valor de espera em uma resposta de throttling, respeite-a. O provedor conhece a própria recuperação melhor do que a sua fórmula. Trate a dica como um piso sob o atraso calculado, não como substituta dele, e mantenha o jitter para que chamadores com a mesma dica não voltem juntos. Um piso ainda não é licença para dormir além do deadline: se a dica for mais longa que o tempo restante, pare de fazer retry e deixe a camada seguinte decidir.

Deadlines e retry budgets

O backoff com jitter decide o espaçamento dos retries. Dois outros limites decidem quantos pode haver.

Deadlines

Um **deadline** é o momento depois do qual o resultado deixa de ser útil.

Um fluxo orientado a requisições tem um deadline natural: o comprador está olhando para a página de checkout e, daqui a 10 segundos, vai embora. Um consumidor em segundo plano tem um mais brando: uma etiqueta de remessa que chega depois do horário de corte da transportadora está tão atrasada quanto uma que nunca chega.

Uma lógica de retry que conta tentativas mas ignora o tempo pode ir muito além da utilidade. Três tentativas com timeout de 30 segundos cada e atrasos crescentes entre elas são minutos de espera. Prefiro dar a toda operação com retry um orçamento total de tempo e parar de fazer retry quando o próximo atraso o ultrapassar, não importa o que a contagem de tentativas diga.

```
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)
```

O deadline também deve viajar. Se Orders dá 10 segundos a um fluxo e chama Payments, Payments deve repassar o que sobrou, e não começar 10 segundos novos por conta própria. Caso contrário, cada camada faz retry dentro da paciência da camada anterior, e a soma silenciosamente excede o que a pessoa usuária está disposta a esperar.

Retry budgets

Um limite de tentativas por chamada não protege uma dependência quando muitas chamadas estão falhando ao mesmo tempo. Cada chamada permanece dentro do seu limite de três retries, e a dependência ainda recebe quatro vezes o tráfego normal.

Um **retry budget** limita os retries como uma parcela do tráfego total, e não por chamada. Uma forma comum é um token bucket: cada chamada bem-sucedida adiciona uma fração de token, cada retry gasta um, e, quando o bucket está vazio, as chamadas falham rápido, sem fazer retry. Em regime estável, os retries são uma pequena fração das requisições. Em uma queda generalizada, o budget se esgota em instantes e o sistema para de amplificar a própria falha.

Os números são seus para escolher e dependem de quanta carga extra a dependência consegue absorver. Não vou oferecer um valor, porque qualquer valor que desse seria um palpite sobre a sua dependência. O que importa é que o budget exista, que seja compartilhado entre os chamadores de um processo e que o seu esgotamento seja visível em um dashboard.

Um budget baseado em proporção, mantido por processo, costuma bastar, porque cada processo limitando a própria proporção já limita o total. Um budget compartilhado entre instâncias é uma opção, não uma melhoria estrita: ele exige um contador distribuído, e esse contador é uma nova dependência no caminho da falha, com um modo de falha próprio.

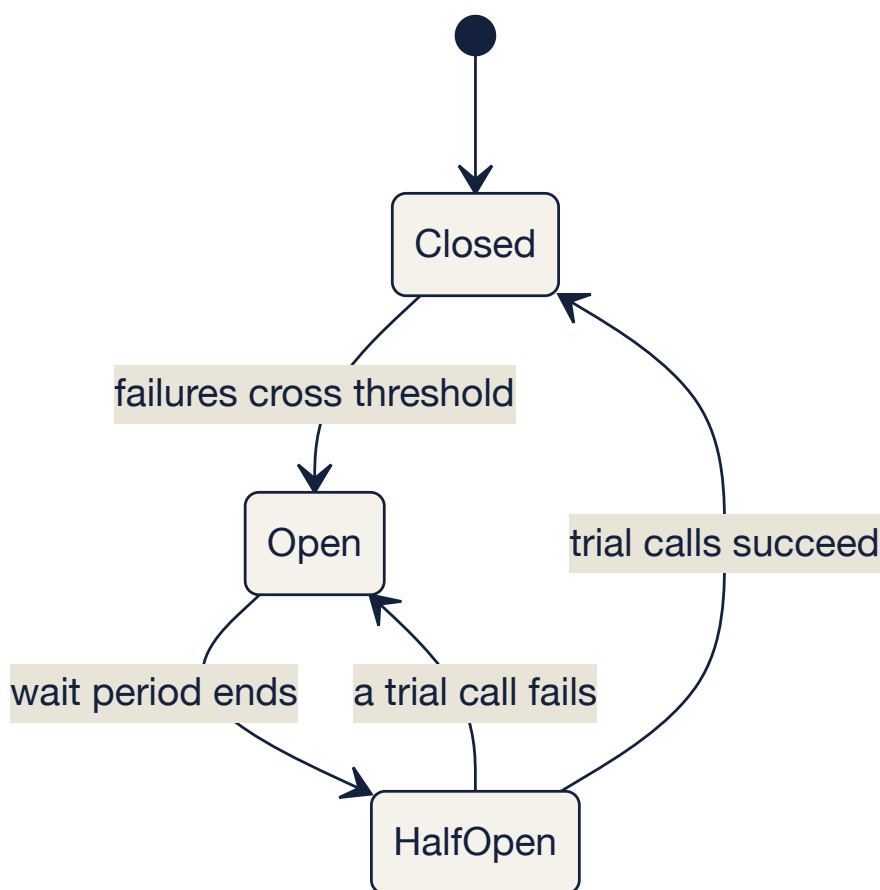
O preço é que algumas chamadas que teriam dado certo com um retry vão falhar. Esse é o objetivo de um budget: ele troca o sucesso individual pela sobrevivência do sistema, e deve fazê-lo de propósito.

Retry storms e o circuit breaker

Um **retry storm** é o que acontece quando o retry passa a ser a carga dominante sobre uma dependência que está falhando. O cenário de abertura da Parcelly era um deles. Ele precisa de três ingredientes: uma dependência lenta ou com falhas, chamadores que fazem retry e nenhum mecanismo que mande os chamadores pararem. As camadas pioram a situação. Se Orders faz três tentativas contra Payments, e Payments faz três tentativas contra o provedor, uma única captura que falhou pode virar nove chamadas ao provedor. Com mais uma camada de três tentativas, vira 27.

Duas defesas tratam a tempestade diretamente. O retry budget acima limita quanto retry o sistema fará. O circuit breaker limita se ele deve chamar, afinal.

Um **circuit breaker** observa os resultados das chamadas a uma dependência e tem três estados. Ele fica fechado na operação normal, com as chamadas passando. Quando as falhas cruzam um limiar, ele abre, e as chamadas falham imediatamente, sem tocar a dependência, dando a ela espaço para se recuperar. Depois de uma espera, ele passa a half-open e deixa passar algumas chamadas de teste. Se elas tiverem sucesso, ele fecha. Se não, abre de novo.



O breaker e a política de retry são complementares e respondem a perguntas diferentes. O backoff decide quanto esperar antes da próxima tentativa de uma chamada. O breaker decide, a partir do padrão entre as chamadas, se vale a pena fazer tentativas. Um breaker aberto transforma a próxima tentativa do loop de retry em uma falha instantânea. Essa falha não é transitória nem desconhecida: é o sinal **adiada** da classificação acima. O loop de retry não deve contá-la, nem aplicar backoff sobre ela, nem girar em cima dela. Ele devolve o controle ao consumidor, que decide o que deixar de fazer.

Um consumidor tem uma escolha extra que um chamador orientado a requisições não tem. Quando o breaker está aberto, um consumidor deve parar de puxar mensagens do broker, ou pausar a partição, em vez de consumir mensagens e fazer cada uma falhar. As mensagens que ficam no broker estão seguras pelo tempo que a retenção dele permitir, custam armaze-

namento e aparecem como atraso do consumidor. As mensagens que são consumidas, recebem retry e falham queimam a sua contagem de tentativas e podem ir para as dead letters por um problema que nunca foi delas. Pausar é a forma do breaker no lado do consumidor, e a construiria antes da maior parte do que há neste capítulo.

Pausar levanta duas perguntas de acompanhamento. Primeiro, sem tráfego fluindo, ninguém testa o estado half-open. Retome com concorrência de um, ou envie uma chamada de sondagem sintética, para que as chamadas de teste sejam feitas por algo que saiba que são testes. Segundo, em um broker baseado em log, um consumidor pausado precisa manter vivo o seu loop de polling, pausando as partições mas não o polling, ou o grupo decidirá que o consumidor morreu e reatribuirá as suas partições (a janela de vivacidade descrita adiante).

O custo de um breaker é que ele pode falhar do jeito errado. Limiares sensíveis demais abrem diante de uma oscilação breve e transformam uma falha transitória em uma indisponibilidade. Limiares tolerantes demais abrem tarde demais. Um breaker também acrescenta um estado que alguém precisa entender quando um fluxo para de funcionar às três da manhã: a primeira pergunta é se a dependência está fora do ar ou se apenas o breaker está aberto.

Retries sob sobrecarga: backpressure

Um breaker reage a uma dependência que falhou. **Backpressure** é a disciplina para a etapa anterior, quando uma dependência está lenta, saturada ou em recuperação, e o sistema precisa decidir quanto trabalho entregar a ela. O Capítulo 3 a situa dentro do modelo de falhas. Esta seção é o mecanismo.

Volte ao cenário de abertura. A lentidão do provedor não veio de um bug. Veio de mais trabalho do que ele conseguia absorver, e todas as instâncias de Payments continuaram alimentando-o. Um consumidor sem limite de trabalho em andamento se comporta exatamente assim: continua puxando, continua iniciando chamadas e continua acrescentando à pilha que já está pesada demais. Os limites abaixo são os que colocaria em prática, mais ou menos nesta ordem.

- **Limite o trabalho em andamento.** Restrinja o número de mensagens que um consumidor mantém ao mesmo tempo. Em um broker baseado em fila, isso é o prefetch ou a contagem de mensagens em andamento. Em um broker baseado em log, é o número de registros por poll mais o número de handlers em execução. Sem esse limite, a memória do consumidor e o relógio de reentrega do broker passam a ser o limite, e ambos são limites piores do que um que você escolheu.
- **Reduza a concorrência quando a dependência desacelerar.** A concorrência que é adequada a 50 milissegundos por chamada é errada a cinco segundos por chamada, porque o mesmo número de slots agora comporta muito mais trabalho em espera. Diminua o número de chamadas concorrentes quando a latência subir e aumente-o de novo, devagar, quando a latência se recuperar.
- **Pause em vez de falhar.** Quando a dependência está saturada, pare de puxar, como com um breaker aberto. Um consumidor pausado aparece como atraso, que é visível e seguro. Um consumidor que puxa e falha queima tentativas.
- **Atenda o tráfego ativo antes dos retries.** Os retries e o redrive competem com as primeiras tentativas pela mesma dependência. Quando a capacidade é curta, as primeiras tentativas devem vencer, porque um retry é trabalho que o sistema já sabe que está com dificuldades. Execute os consumidores de retry topic e de redrive com concorrência menor, ou deixe-os esperar enquanto o fluxo principal tiver backlog.
- **Descarte na entrada.** Em algum ponto a montante, tem de existir um limite para o trabalho aceito, ou a fila cresce sem limite enquanto a dependência se recupera. Rejeite ou adie o novo trabalho, com um sinal claro para o chamador, em vez de enfileirá-lo indefinidamente atrás de uma dependência que não consegue atendê-lo.

Esses limites e o retry budget são duas visões de uma só ideia. O budget limita quanto trabalho extra o retry acrescenta, e o backpressure limita quanto trabalho, extra ou não, está em andamento. Nenhum dos dois basta sozinho: um budget sem limite de trabalho em andamento deixa uma dependência lenta encher todos os slots de primeiras tentativas, e um limite de trabalho em andamento sem budget deixa os slots se encherem de retries.

O preço é atraso e rejeição. Um sistema que descarta carga diz não a alguns chamadores, de propósito, e um sistema que pausa entrega algumas mensagens com atraso. Ambos trocam throughput por sobrevivência, e quem é responsável pela resposta a “qual trabalho espera” é a pessoa responsável pelo fluxo no negócio, não a engenharia que ajusta os números. O Capítulo 8 aplica a mesma ideia de rate limit ao redrive, e o Capítulo 9 aplica o limite de trabalho em andamento às lanes por chave.

Onde vivem os retries

Uma mensagem pode receber retry em várias camadas, e as camadas se comportam de formas diferentes. Misturá-las sem saber é como uma mensagem acaba recebendo retry muito mais vezes do que alguém pretendia, ou nenhuma.

Retry no nível da aplicação

O consumidor faz retry dentro do handler: o código acima, envolvendo uma chamada ao provedor. A mensagem permanece sem ack enquanto os retries acontecem. Essa camada é barata, rápida e boa para falhas que passam em milissegundos ou segundos. Os limites dela são: ela retém a capacidade do consumidor enquanto espera; uma falha do consumidor perde o estado do retry (o broker vai reentregar e a contagem de tentativas recomeça, e é por isso que uma mensagem que derruba o processo precisa da detecção que o Capítulo 8 descreve); e esperas longas correm o risco de exceder os próprios timeouts do broker.

Em um broker baseado em fila, a mensagem sem ack fica oculta durante o **visibility timeout**, depois do qual é reentregue. Se os retries dentro do handler durarem mais que esse timeout, o broker reentrega enquanto a primeira cópia ainda está recebendo retry, e agora duas cópias estão em execução. Em um broker baseado em log, um consumidor que passa tempo demais entre polls pode ser considerado morto pelo seu grupo de consumidores e ter as partições reatribuídas, com o mesmo resultado. A regra que seguiria: o tempo total máximo dos retries dentro do handler deve ficar confortavelmente dentro do timeout do broker, e qualquer coisa mais longa pertence a outra camada.

Reentrega pelo broker

O consumidor faz a mensagem falhar, e o broker a entrega de novo. As duas famílias de brokers diferem aqui, e a diferença importa.

Em um broker baseado em fila, a reentrega é embutida. O consumidor dá nack na mensagem, ou deixa o visibility timeout expirar, e a mensagem volta para a fila, imediatamente ou após um atraso configurado, onde qualquer consumidor concorrente pode recebê-la.

Um broker baseado em log não reentrega por conta própria. A posição de leitura do consumidor avança conforme ele faz poll, e o offset do commit apenas registra onde um consumidor reiniciado retomaria (o Capítulo 2 trata disso). Falhar ao processar um registro e não fazer commit do offset dele, portanto, não faz nada por si só: o consumidor lê o registro seguinte e, se depois fizer commit, esse commit cobre também o registro que falhou, que é ignorado em silêncio. O registro que falhou só é lido de novo após uma reinicialização, um rebalanceamento ou um passo explícito do consumidor.

Em um log, então, o consumidor precisa escolher uma de três ações. Pode voltar (seek) ao offset que falhou, ou pausar a partição, e fazer retry no mesmo lugar, o que preserva a ordem e trava tudo o que vem atrás. Pode publicar o registro em um retry topic e seguir em frente. Ou pode escrevê-lo em um dead-letter topic e seguir em frente. Cada uma é uma decisão deliberada no código do consumidor, não algo que o broker faz por você.

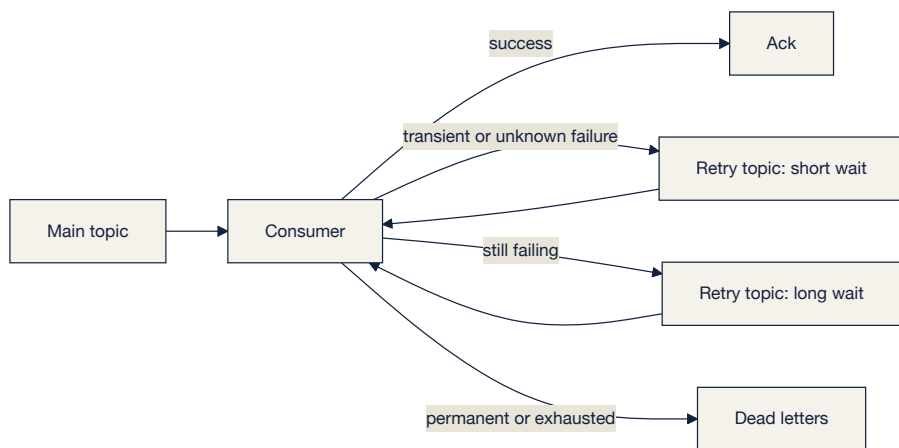
A reentrega pelo broker é durável: sobrevive a uma falha do consumidor. Também é grosseira. Muitos brokers reentregam de imediato ou em um cronograma fixo, sem backoff, e a contagem de tentativas pode estar disponível apenas como um contador de entregas que o consumidor precisa ler e tratar. Encare-a como a rede de segurança sob as outras camadas, não como a política principal de retry.

Retry topics e filas de atraso

Quando uma falha exige uma espera maior do que o retry dentro do handler consegue sustentar, uma estrutura comum é tirar a mensagem do fluxo principal e deixá-la voltar mais tarde. Um consumidor que faz uma mensagem falhar a publica, com a contagem de tentativas incrementada e um horário de não antes de, em um **retry topic** (ou fila de retry) e dá ack

na original. Um consumidor separado, ou o mesmo em outro topic, a recolhe quando o horário chega. Brokers que oferecem entrega atrasada, ou atrasos por mensagem, dão isso diretamente. Onde não oferecem, um arranjo comum é uma escada de retry topics, cada um com uma espera fixa, um para cada passo do backoff.

As esperas são fixas por topic por um motivo. Em um log, um consumidor não pode pular um registro que ainda não venceu: precisa esperar na cabeça até que o registro esteja pronto. Se todo registro de um topic tem a mesma espera, o topic já está em ordem de vencimento, e esperar na cabeça não custa nada a mais. Um topic que misturasse atrasos curtos e longos seguraria os curtos atrás dos longos. Horários de não antes de por mensagem só funcionam em brokers com entrega atrasada nativa.



Essa camada libera o fluxo principal, de modo que uma mensagem com falha deixa de segurar as mensagens que vêm atrás. Ela sobrevive a reinicializações, porque o estado vive no broker. E também dá um lugar para olhar: a profundidade do retry topic é uma medida direta de quanto trabalho está esperando por uma dependência em recuperação.

Ela custa mais maquinaria. Há mais topics para criar, nomear, monitorar e proteger. A republicação é, ela mesma, uma operação de produção que pode falhar, o que traz de volta o dual write do Capítulo 1, então a ordem importa: publique primeiro no retry topic e só depois dê ack na original. Se o processo morrer entre as duas, a mensagem existe duas vezes. Isso é um duplicado, que o consumidor precisa tolerar de qualquer forma.

Três regras impedem que a escada prejudique aquilo que deveria proteger:

- **Dedique o topic a um consumidor.** Um retry topic pertence a um grupo de consumidores ou a uma assinatura. Se outro grupo puder lê-lo, esse grupo reprocessa uma mensagem que nunca falhou para ele.
- **Preserve a identidade.** A mensagem republicada mantém a chave original, o `event_id` e os headers de negócio. Uma chave nova quebra a ordenação, e um `event_id` novo quebra o inbox.
- **Dimensione a retenção pela espera.** A retenção de um retry topic precisa exceder a maior espera mais o atraso que você espera nele, ou a mensagem expira enquanto aguarda.

O custo mais importante é a **ordenação**, que a próxima seção trata.

Escolhendo as camadas

Na prática, um consumidor usa as três. Um retry curto no nível da aplicação absorve as oscilações, como uma conexão reiniciada ou um deadlock, que passam em um ou dois segundos. Um retry topic ou fila de atraso cobre as falhas que precisam de minutos. A reentrega do broker cobre as quedas. Uma dead-letter queue (DLQ) encerra a cadeia, recebendo o que esgotou as tentativas ou nunca poderá ter sucesso; o Capítulo 8 trata do design dela.

As camadas devem ser projetadas em conjunto, com um único orçamento total de tentativas, e não três independentes. Se cada camada permite três tentativas, uma mensagem pode ser tentada 27 vezes, e ninguém planejou isso. A política ao final deste capítulo torna a multiplicação explícita e a limita.

Retries e ordenação

Fazer retry de uma mensagem muda a ordem em que as mensagens são processadas. Isso decorre das camadas descritas acima e passa despercebido até causar dano.

Considere o consumidor de Inventory da Parcelly, que recebe `OrderPlaced` e depois `OrderCancelled` para o mesmo pedido, nessa ordem. Suponha que o processamento de `OrderPlaced` falhe de forma transitória e que o consumidor a mova para um retry topic, passando a processar `OrderCancelled`. O cancelamento não encontra reserva para liberar,

porque, até onde Inventory sabe, ainda não existe nenhuma. Então o `OrderPlaced` do retry chega e cria uma reserva que nada vai liberar. O estado final está errado, e cada passo individual parecia razoável.

Há duas opções honestas, e cada uma tem um preço.

- **Bloquear e fazer retry no mesmo lugar.** Em um broker baseado em log, o consumidor volta (seek) ao offset que falha, ou pausa a partição, e faz retry dele, com backoff, antes de tocar em qualquer coisa que venha atrás. A ordem é preservada para aquela partição. O preço é que a partição inteira espera atrás de uma única mensagem, de modo que uma mensagem com falha pode travar todos os agregados dessa partição. Essa é a escolha certa quando a ordem importa mais que o throughput, e é por isso que classificar rapidamente uma falha permanente importa tanto: uma poison message na cabeça de uma partição bloqueia tudo o que vem atrás.
- **Desviar a mensagem e aceitar a reordenação.** A mensagem com falha vai para um retry topic e o consumidor continua. O throughput é preservado. O preço é que o consumidor precisa tolerar a chegada fora de ordem, com técnicas como verificações de versão, ou estacionando também toda mensagem posterior do mesmo agregado até que a anterior tenha sucesso. O Capítulo 9 trata dessas técnicas.

Escolheria por stream. Para um stream de eventos em que as mensagens de um agregado precisam ser aplicadas em ordem, faria retry no mesmo lugar e investiria em classificação rápida e em uma saída para dead letters. Para um stream de fatos independentes, em que a ordem entre mensagens não importa, desviar é barato e mantém o fluxo andando.

Em brokers baseados em fila sem recurso de agrupamento, a ordenação entre consumidores concorrentes já é fraca de saída, de modo que os retries pioram um pouco uma propriedade existente em vez de quebrar uma garantia. Mesmo assim, um consumidor que presume ordem não deve contar com uma fila para fornecê-la. Se você usa um recurso de agrupamento (filas FIFO, message groups, sessões), o trade-off é o mesmo de um log: uma mensagem com falha bloqueia o seu grupo até ter sucesso ou ir para as dead letters, do mesmo modo que um retry no mesmo lugar bloqueia uma partição.

Retries sem idempotência

Todo retry é uma tentativa repetida de um trabalho cujo resultado anterior pode ser desconhecido. Quando a tentativa anterior falhou antes de fazer qualquer coisa, o retry é inofensivo. Quando falhou depois de fazer o trabalho, o retry faz o trabalho duas vezes.

Isso não é um caso extremo raro. Considere o cenário de abertura. Payments chama o provedor para capturar 4.990 unidades menores de EUR. O provedor captura o dinheiro e envia a resposta, e a resposta se perde. Payments vê um timeout. O timeout é classificado como desconhecido, e a política manda fazer retry. A segunda requisição de captura sai. Se o comprador será cobrado uma vez ou duas passa a depender inteiramente de algo tornar a segunda requisição reconhecível como uma repetição.

Esse algo é uma **chave de idempotência**: um identificador estável, enviado com a requisição, que permite ao receptor reconhecer que já viu esse trabalho antes. Na chamada ao provedor, é a mesma chave em toda tentativa para a mesma captura, que um provedor que honra essas chaves tratará como uma única captura. Para consumir eventos, normalmente é o `event_id`, verificado no inbox na mesma transação do trabalho. O Capítulo 7 constrói ambos.

Aviso: Uma política de retry só é segura sobre uma operação idempotente. Antes de habilitar retries em qualquer chamada que tenha efeito, como cobrar um cartão, enviar um e-mail ou reservar estoque, estabeleça o que torna a segunda tentativa inofensiva. Se nada a torna, o número correto de retries automáticos é zero, e a falha deve ir para uma pessoa. Como um broker at-least-once vai reentregar depois de uma queda ou de um timeout, não importa o que o handler queira, o “zero” precisa ser imposto pelo handler: ele deve detectar que já viu essa mensagem, ou dar ack antes do efeito e aceitar at-most-once, e então encaminhar a falha para uma pessoa.

A dependência vale nos dois sentidos. A idempotência é o que torna os retries seguros, e os retries são o que tornam a idempotência necessária. Um sistema que faz retry de forma agressiva sem consumidores idempotentes

produzirá efeitos duplicados a uma taxa proporcional à sua taxa de falhas, ou seja, nos momentos em que o sistema já está sob estresse e ninguém tem tempo de limpar a bagunça.

Há ainda um caso mais sutil. Uma operação pode ser idempotente para a mesma chave e ainda assim ser insegura para fazer retry, porque a chave muda. Se um handler constrói um identificador novo a cada tentativa, como um UUID novo a cada chamada, o provedor vê requisições distintas todas as vezes. A chave precisa ser derivada uma única vez, a partir de entradas estáveis como o `event_id` ou o pedido, e reutilizada em toda tentativa.

Uma política de retry para a Parcelly

Para reunir as peças, eis o que registraria por escrito para o consumidor de `CapturePayment` em Payments. Os números são ilustrações, não recomendações, e a sua dependência precisará dos próprios.

```
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
```

Leia-a como um conjunto de decisões, cada uma com um motivo. A camada dentro do handler absorve as oscilações curtas. Os retry topics cobrem um provedor fora do ar por minutos. O deadline diz quando o trabalho deixa de valer a pena, e o responsável por essa decisão é o negócio, não o engenheiro. Uma captura obsoleta pode precisar se tornar um `OrderCancelled`, mas a compensação depende do que de fato aconteceu no provedor. Se a última tentativa estourou o timeout, a captura pode ter dado certo, e então o passo seguinte correto é um `PaymentRefunded`, não um `PaymentVoided`. É por isso que a política pede o status primeiro. A chave, derivada uma única vez, é o que impede toda a estrutura de cobrar duas vezes o comprador.

Duas linhas da política merecem uma segunda olhada. O `maxTotalAttempts` existe porque as camadas se multiplicam: três tentativas dentro do handler, vezes uma primeira passada mais três passos de retry topic, dá até 12 chamadas ao provedor, antes de contar qualquer reentrega depois de uma queda. Um total único de 8 (ilustrativo) é o número que alguém de fato controla. E todo o horizonte de retry, duas horas aqui, precisa ser menor que o tempo pelo qual o provedor se lembra das chaves de idempotência. Os provedores guardam as chaves por uma janela limitada, e um retry que chega depois da janela é uma requisição nova carregando uma chave antiga.

Quando o redrive do Capítulo 8 devolve uma captura morta, o contador de tentativas recomeça, mas um contador de redrive separado permanece, de modo que o histórico não se perde e uma mensagem que volta sempre fica visível como tal.

Observe o que falta: uma configuração única de “contagem de retries”. Em um sistema com camadas, esse número isolado esconde pelo menos quatro decisões.

Trade-offs e custos

Os retries são uma das formas mais baratas de melhorar a confiabilidade aparente de um sistema e uma das mais fáceis de usar em excesso. Os custos deles são específicos.

Latência. Todo retry acrescenta tempo de espera. O backoff e o jitter distribuem a carga, mas também esticam o tempo até o sucesso. Um fluxo que faz retry por minutos pode terminar corretamente e ainda assim ser inútil para o comprador que foi embora depois de 10 segundos. Onde há uma pessoa esperando, favoreceria uma falha rápida e uma mensagem clara em vez de um retry longo e silencioso.

Amplificação. Os retries multiplicam a carga, e retries em camadas a multiplicam de novo. Uma política razoável em cada serviço pode ser perigosa na cadeia. Alguém precisa ser responsável pelo orçamento de tentativas de ponta a ponta, porque nenhum serviço sozinho consegue enxergá-lo.

Capacidade. Um consumidor que faz retry no mesmo lugar ocupa o seu slot enquanto espera. Numa queda longa, todos os consumidores podem acabar esperando, e o sistema para de progredir até mesmo no trabalho que teria sucesso. Pausar o consumo por meio de um breaker, limitar o trabalho em andamento com backpressure e mover as esperas longas para retry topics são as defesas usuais.

Falhas ocultas. Um retry que acaba tendo sucesso parece um sucesso visto de fora. Se você não conta retries e tentativas, uma dependência pode se degradar por semanas atrás de uma política que a absorve em silêncio. O Capítulo 13 trata do que medir: contagens de tentativas, taxa de retry e a idade da mensagem com retry mais antiga pertencem a um dashboard.

Complexidade. Retry topics, filas de atraso e budgets são infraestrutura que alguém precisa construir, nomear, monitorar e explicar. Para um sistema pequeno, com um consumidor e uma dependência, um retry dentro do handler com backoff e uma saída para dead letters pode ser tudo de que você precisa, e uma escada de retry topics pode ser mais maquinaria do que o problema merece.

Quando não fazer retry. Não faça retry de falhas permanentes. Não faça retry de operações com efeito que não sejam idempotentes. Não faça retry além do deadline. Não faça retry quando um circuit breaker diz que a dependência está fora do ar, e não cobre dessa mensagem uma tentativa por isso. Não faça retry dentro de uma camada que já está dentro do retry de outra camada, a menos que você tenha calculado a multiplicação. E não faça retry como forma de evitar uma decisão: uma falha que insiste em voltar é informação, e a resposta certa muitas vezes é parar e olhar.

Erros comuns

Fazer retry de tudo do mesmo jeito

É tentador porque um único loop é o código mais curto. Ele trata uma recusa de cartão como uma conexão reiniciada e desperdiça capacidade em falhas que não podem ter sucesso. Classifique primeiro e dê a cada classe o seu tratamento.

Atrasos fixos sem jitter

Uma espera constante parece previsível e simples. Ela sincroniza todos os chamadores que falharam juntos em um pico que se repete. Use exponencial backoff com jitter e, na minha opinião, o full jitter como ponto de partida.

Contar tentativas, mas não o tempo

Um limite de tentativas parece um limite de segurança. Com timeouts longos e atrasos crescentes, ele ainda pode se estender por minutos além da utilidade. Defina um deadline para a operação inteira e pare quando o próximo atraso o cruzar.

Fazer retry em todas as camadas

O responsável por cada camada acrescenta um retry sensato, e a multiplicação é invisível para todos eles. Decida um orçamento total de tentativas para a cadeia e ponha os retries onde custam menos.

Fazer retry enquanto o breaker está aberto

É tentador manter o loop girando, porque o loop estava funcionando. Um consumidor que puxa mensagens e as faz falhar contra uma dependência morta consome as tentativas delas e pode mandá-las indevidamente para as dead letters. Trate o breaker aberto como um sinal de adiamento que não custa tentativa, pause o consumo e deixe as mensagens esperarem no broker.

Deixar um consumidor puxar sem limite

Um consumidor sem limite é o padrão, e dá a sensação de throughput máximo. Contra uma dependência lenta, ele enche todos os slots com trabalho em espera e alimenta o retry storm. Limite as mensagens em andamento, reduza a concorrência quando a latência subir e deixe as primeiras tentativas vencerem os retries.

Ignorar os timeouts do broker

Os retries dentro do handler parecem locais ao handler. Se durarem mais que o visibility timeout ou que a janela de vivacidade do grupo, o broker reentrega ou reatribui, e duas cópias do trabalho rodam ao mesmo tempo. Mantenha o tempo total dentro do handler bem dentro dos limites do broker.

Fazer retry passando por cima de uma mensagem que precisa manter a ordem

Mover uma mensagem com falha para um retry topic é um conserto rápido para uma partição travada. Ele reordena os eventos daquele agregado. Decida primeiro o requisito de ordenação e só então escolha entre bloquear e desviar.

Fazer retry de efeitos não idempotentes

O loop de retry leva uma tarde para ser adicionado, e a idempotência fica para depois. O primeiro timeout depois de uma chamada bem-sucedida produz um efeito duplicado. Estabeleça o que torna a segunda tentativa inofensiva antes de habilitar o primeiro retry.

Gerar uma chave nova a cada tentativa

Parece zelo, já que cada requisição fica identificada de forma única. Anula o propósito, porque o receptor não consegue reconhecer uma repetição. Derive a chave uma única vez a partir de entradas estáveis e reutilize-a.

Não contar os retries

Se apenas as falhas finais são registradas em log, uma dependência em degradação fica invisível até que os retries deixem de bastar. Registre cada tentativa, com a sua classe e o seu atraso.

Pontos-chave

- Um retry é a promessa de tentar de novo, não a promessa de ter sucesso, de modo que a primeira pergunta diante de qualquer falha é se tentar de novo pode ajudar.
- Classifique as falhas como transitórias, permanentes ou desconhecidas, registre a classificação como uma tabela e trate as falhas não reconhecidas como desconhecidas, com um limite mais estrito.
- Use exponential backoff com jitter, limite a operação inteira com um deadline e compartilhe um retry budget para que os retries não se tornem a carga dominante sobre uma dependência que está falhando.
- Um circuit breaker decide se vale a pena chamar; um breaker aberto é um sinal de adiamento que não custa tentativa, e para um consumidor a forma mais forte é pausar o consumo. O backpressure faz o mesmo trabalho antes, limitando o trabalho em andamento e deixando o tráfego ativo vencer os retries e o redrive.
- Os retries podem viver no handler, na reentrega do broker e em retry topics, e devem ser projetados em conjunto sob um único orçamento de tentativas, com o tempo dentro do handler mantido dentro dos timeouts do broker.
- Desviar uma mensagem com falha reordena os eventos do seu agregado; escolha entre bloquear no mesmo lugar e tolerar a reordenação conforme o requisito de ordenação do stream.
- Uma política de retry só é segura sobre operações idempotentes, com a chave de idempotência derivada uma única vez e reutilizada em toda tentativa.

Os retries decidem com que frequência você tenta de novo, mas não conseguem decidir se tentar duas vezes é inofensivo. O Capítulo 7 constrói a outra metade: o consumidor idempotente, que transforma um duplicado de desastre em um não evento.

Capítulo 7 — Idempotência

Imagine o serviço Ledger da Parcelly no meio de um rolling deploy, enquanto os seus consumidores são reiniciados um de cada vez. Um consumidor leu um evento `PaymentCaptured`, escreveu um lançamento contábil de 4.990 unidades menores de EUR e está prestes a dar ack na mensagem quando o processo recebe o sinal de desligamento. O ack nunca sai da máquina. O broker, fazendo exatamente o que a entrega at-least-once promete, entrega a mesma mensagem ao próximo consumidor que a pedir.

O segundo consumidor lê o evento, não encontra nada de incomum e escreve um segundo lançamento. O ledger é append-only e correto por construção, de modo que agora registra, fielmente e para sempre, que o comprador pagou duas vezes. Nada travou. Nenhum alerta disparou. Todos os componentes se comportaram como projetado, e os livros-razão estão errados.

Notifications tem a sua própria versão da mesma tarde. O comprador recebe três vezes o e-mail “sua encomenda está agendada”, porque a mensagem foi reentregue depois que um handler lento ultrapassou o seu visibility timeout. Esse caso é constrangedor. O do ledger é caro.

Os capítulos anteriores estabeleceram que os duplicados são normais: o broker reentrega, o relay publica duas vezes depois de um failover e os retries do Capítulo 6 são duplicados por construção. Este capítulo responde à pergunta que aqueles deixam em aberto. Quando a mesma mensagem chega duas vezes, o que exatamente um consumidor deve fazer para que a segunda chegada não mude nada?

O problema e por que ele é difícil

O objetivo parece modesto: processar cada mensagem uma vez, até onde o mundo consegue perceber. A dificuldade é que o consumidor não tem como saber se já viu uma mensagem sem se lembrar dela, e lembrar é, ele mesmo, uma escrita que pode falhar no pior momento.

Comece por onde os duplicados vêm, porque a lista é mais longa que o broker.

Origem	Como o duplicado aparece
Reentrega pelo broker	O ack se perdeu, o visibility timeout expirou ou um rebalanceamento moveu a partição antes de o offset sofrer commit
Relay	Um relay publicou uma linha, caiu antes de marcá-la como publicada e a publicou de novo depois de reiniciar (Capítulo 5)
Retry da aplicação	Um consumidor fez retry de uma chamada, ou um produtor fez retry de uma publicação cuja primeira tentativa na verdade tinha dado certo (Capítulo 6)
Redrive e replay	Alguém devolveu dead letters, ou releu de propósito um log de eventos a partir de um offset anterior
Pessoas e clientes	O comprador deu um clique duplo em “Pagar”, um aplicativo móvel fez retry em uma rede instável, um emissor de webhook entregou duas vezes

A maioria delas produz a *mesma* mensagem duas vezes, com o mesmo `event_id`. Algumas produzem duas mensagens diferentes que expressam a mesma intenção, e essa distinção vai importar quando escolhermos uma chave.

A solução tentadora é olhar antes de agir. Perguntar se esta mensagem já foi processada e, se não foi, processá-la. Quero dar um nome a isso, o anti-padrão **check-then-act**, porque ele parece correto e falha em silêncio.

```
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)
```

Leia-o sob falha. Se o processo morrer depois de `postLedgerEntry` e antes de `inboxInsert`, o lançamento existe e o inbox não sabe, de modo que a reentrega o lança de novo. Inverta as duas escritas e a falha também se inverte: uma queda depois de `inboxInsert` e antes da escrita no ledger

marca a mensagem como tratada quando não foi, e o efeito se perde para sempre. Nenhuma das ordens é segura, porque as duas escritas não são um único ato atômico.

Há uma segunda falha que nada tem a ver com quedas. Dois consumidores podem receber a mesma mensagem quase no mesmo momento, por exemplo quando uma primeira tentativa lenta ultrapassa o seu visibility timeout. Os dois executam a verificação, os dois não encontram nada e os dois prosseguem. A janela entre “verificar” e “agir” é pequena e, em volume suficiente, é atingida com regularidade.

Então o problema real tem duas metades. A memória do que foi processado precisa ser escrita atomicamente com o efeito, e a memória precisa ser capaz de arbitrar entre tentativas concorrentes. O restante do capítulo constrói as duas, começando pelo caso em que você não precisa de nenhuma.

Idempotência natural e forçada

Uma operação é **idempotente** quando aplicá-la duas vezes deixa o sistema no mesmo estado que aplicá-la uma vez. Algumas operações são idempotentes pela sua própria forma, e procuraria por elas antes de construir qualquer maquinaria.

Definir um valor é idempotente: `set order.status = "PAID"` pode rodar 10 vezes e o pedido continua PAID. Apagar uma linha por id é idempotente. Inserir uma linha com uma chave primária que você deriva do evento, ignorando o conflito, é idempotente. Escrever o novo estado completo de um agregado (“a remessa agora está DISPATCHED, com este código de rastreamento”) é idempotente. Chamo isso de **idempotência natural**: o duplicado não tem efeito extra por causa do que a operação é, não por causa de algo que você acrescentou.

Outras operações não são. `balance = balance + 4990` aplicada duas vezes vira dois depósitos. `send email` envia dois e-mails. `append a ledger entry` acrescenta dois. `increment the counter of units sold` conta a unidade duas vezes. Chamar o endpoint de captura de um provedor de pagamento sem uma chave pode cobrar duas vezes. Nesses casos, a idempoten-

tência precisa ser adicionada, o que chamo de **idempotência forçada**: você registra que o trabalho foi feito e consulta o registro antes de fazê-lo de novo.

A idempotência natural tem dois limites honestos. O primeiro é que ela depende, em silêncio, da ordem. Definir `status = "PAID"` é idempotente, mas um duplicado atrasado de um evento mais antigo pode sobrescrever um status mais novo: um `OrderPlaced` reentregue que chega depois de `OrderConfirmed` e volta o status para `PLACED`. A operação é idempotente e o sistema continua errado. O Capítulo 9 trata de ordenação e verificações de versão; por ora, perceba que “definir para um valor” só é seguro contra duplicados quando o valor não pode ficar obsoleto.

O segundo limite é que é tentador superestimá-la. Um handler que “só atualiza uma linha” muitas vezes também escreve um registro de auditoria, emite um evento de acompanhamento ou incrementa um contador, e cada um desses é um incremento disfarçado. Quando reviso um consumidor, listo todos os efeitos dele e pergunto, de cada um separadamente, se sobrevive a uma segunda execução. Um único efeito não idempotente basta para precisar do inbox.

Minha regra prática é preferir a idempotência natural onde o efeito permitir e escrever no handler por que ela vale. Quando o efeito é um contador, um append, um e-mail ou uma chamada externa, use a forçada. E, na dúvida, use o inbox de qualquer forma: o custo dele é pequeno e previsível, e o custo de errar não é.

Escolhendo a chave de idempotência

Uma **chave de idempotência** é o que o consumidor usa para reconhecer trabalho repetido. Antes de construir a tabela, você precisa decidir o que vai nela, e a escolha define quais duplicados você consegue pegar.

O `event_id`

Para o consumo de eventos, o padrão é o `event_id` do envelope. Ele é único por evento e, o que é crucial, **estável entre reentregas**: toda cópia da mensagem que o broker entrega a você carrega o mesmo valor. O produtor o atribui uma única vez, quando o evento é escrito no outbox (Capí-

tulo 4), de modo que um relay que publica duas vezes publica o mesmo `event_id` duas vezes. Essa é exatamente a propriedade de que uma verificação de deduplicação precisa.

O `event_id` identifica uma *mensagem*. Ele responde a “já tratei este fato específico antes?” e nada mais amplo que isso.

A chave de negócio

Às vezes duas mensagens diferentes descrevem a mesma intenção. Suponha que Orders repita um comando `CapturePayment` depois de um `time-out` e que, por um bug ou por uma escolha de design, cada retry seja escrito como um comando novo, com um `command_id` novo. Payments agora recebe dois comandos com ids diferentes que significam ambos “capturar o pagamento do pedido 81724”. Deduplicar pelo id da mensagem não pega nenhum dos dois.

O remédio é uma **chave de negócio**: um identificador derivado do significado do trabalho, e não do envelope. Para a captura, pode ser o `payment_id` mais a operação, digamos `capture:payment-5521`. Para uma entrada do ledger, pode ser o id do pagamento mais o tipo de entrada. Uma chave de negócio diz “esta unidade de trabalho pode acontecer uma vez”, não importa quantas mensagens a peçam.

Aspecto	<code>event_id</code> / <code>command_id</code>	Chave de negócio
Pega a reentrega da mesma mensagem	Sim	Sim
Pega duas mensagens com a mesma intenção	Não	Sim
Quem a atribui	O produtor, automaticamente	Uma decisão humana por operação
Risco	Deixa passar duplicados lógicos	Uma chave grosseira demais rejeita trabalho legítimo
Onde é armazenada	Tabela genérica de inbox	Em geral uma restrição de unicidade na tabela de domínio

O risco da penúltima linha merece um caso concreto. Se a chave de negócio dos reembolsos for apenas `payment_id`, um segundo reembolso parcial, legítimo, do mesmo pagamento é rejeitado como duplicado. A chave

precisa incluir tudo o que torna duas requisições *genuinamente diferentes*: aqui, talvez uma chave de idempotência do reembolso fornecida pelo chamador. Escolher a chave significa decidir o que “o mesmo trabalho” quer dizer no seu domínio, o que é uma decisão de design, e não uma configuração de framework.

Na minha opinião, as camadas funcionam bem juntas. Use o `event_id` em um inbox genérico como linha de base para todo consumidor e acrescente uma restrição de unicidade de chave de negócio na tabela de domínio sempre que duas mensagens diferentes puderem plausivelmente significar a mesma coisa. O inbox protege contra o broker e o relay. A restrição protege contra o resto do sistema, inclusive contra os bugs dele.

O padrão inbox

O **inbox** é a tabela de ids de mensagens processadas no lado do consumidor. O Capítulo 4 deu ao produtor um transactional outbox para que uma mudança de estado e o seu evento sofressem commit juntos. O inbox é a imagem espelhada: o consumidor faz commit da sua mudança de estado e do registro de ter tratado a mensagem juntos, em uma única transação local no seu próprio banco de dados.

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

A coluna `consumer` permite que um serviço execute vários consumidores independentes contra uma só tabela, de modo que o consumidor de lançamentos do Ledger e o seu consumidor de relatórios lembrem, cada um, do próprio histórico. É por isso que a chave primária é o par `(consumer, event_id)`. Só com `event_id`, o primeiro consumidor a tratar uma mensagem faria o insert do segundo consumidor falhar, e o efeito do segundo seria ignorado como se fosse um duplicado. Um serviço com um único consumidor por tabela pode dispensar a coluna e usar `event_id` como chave, que é a forma mais simples usada em outros pontos do livro.

O handler escreve a linha do inbox e a mudança de negócio na mesma transação.

```
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)
```

Percorra as falhas. Se o processo morre antes do commit, nem o lançamento nem a linha do inbox existem, de modo que a reentrega processa a mensagem do zero. Se morre depois do commit e antes do ack, ambos existem, e a reentrega segue o caminho rápido: a verificação encontra a linha, a mensagem recebe ack e nada é escrito. O efeito e a sua memória nunca podem divergir, porque sofrem commit como um só.

A verificação no topo é uma otimização. Ela poupa uma transação no caso comum de duplicado e pode ser removida sem perder correção. A rede de segurança é a restrição de unicidade dentro da transação, que é onde a arbitragem de verdade acontece.

Dentro de uma transação, a ordem das escritas não altera a atomicidade, de modo que o alerta sobre inverter as duas escritas vale apenas para a versão insegura acima. Ela altera o custo. Nesta versão, um duplicado concorrente faz todo o trabalho antes de falhar no insert, e em alguns bancos de dados uma violação aborta a transação inteira, de modo que o handler não consegue se recuperar dentro dela. A minha escolha seria reivindicar primeiro: inseriria a linha do inbox com semântica de “on conflict do nothing”, olharia se uma linha foi inserida e pularia o efeito se não foi. O duplicado então espera pela chave, não faz trabalho algum e nunca levanta erro. O exemplo da próxima seção usa essa forma.

Uma palavra sobre `ack`, porque o termo esconde uma diferença entre as duas famílias. Em um broker baseado em fila, ele dá ack em uma mensagem. Em um broker baseado em log, costuma ser um commit de offset, que significa “tudo até aqui”, e um consumidor que processa uma partição

em paralelo pode fazer commit além de uma mensagem que ainda não terminou e perdê-la. O inbox protege contra duplicados, não contra isso, e o Capítulo 9 olha para o problema onde a ordenação e o paralelismo se encontram.

Eventos de acompanhamento emitidos por um consumidor

Muitos consumidores fazem mais do que mudar o próprio estado: anunciam o que fizeram. O Ledger lança um registro e emite `LedgerEntryPosted`. Esse evento de acompanhamento é um efeito como qualquer outro, e precisa sobreviver a uma queda exatamente uma vez, nem zero, nem duas.

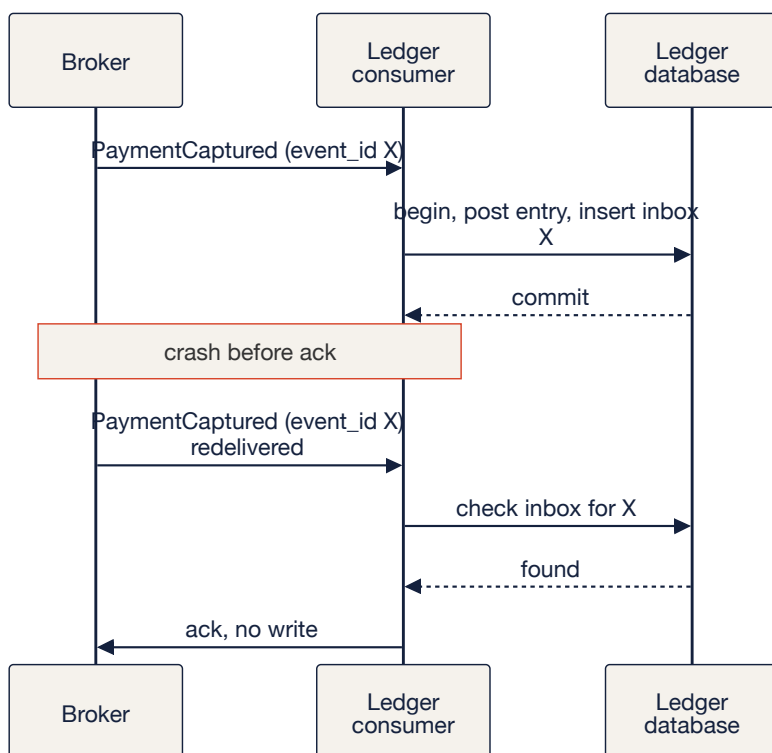
Um handler que publica o acompanhamento diretamente depois do seu commit o perde para sempre numa queda. A reentrega segue o caminho rápido do inbox, encontra a linha, dá ack e pula a publicação. O estado está certo e o evento nunca existiu. A correção é o outbox do Capítulo 4 aplicado ao lado do consumo: a linha do inbox, a mudança de estado e o insert no outbox sofrem commit em uma única transação local.

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

Se o processo morre antes do commit, nada existe e a reentrega faz tudo. Se morre depois do commit, o acompanhamento está no outbox e o relay o publica, enquanto a reentrega encontra a linha do inbox e não faz nada. O outbox do consumidor tem a mesma publicação at-least-once que o do

produtor, de modo que quem consome `LedgerEntryPosted` precisa do seu próprio inbox. O Capítulo 10 se apoia nessa regra quando um passo de saga reage a um evento e emite o seguinte.

É também por isso que o inbox vive no *mesmo banco de dados* do estado que protege. Se o registro de mensagens processadas ficasse em um cache ou em outro banco de dados, voltaríamos a ter duas escritas sem transação compartilhada, que é o dual write do Capítulo 4 no lado do consumidor. O inbox não é uma técnica que se possa encaixar por fora do armazenamento de dados do serviço.



Olhe a nota no meio: a janela da queda fica entre o commit e o ack, e a linha do inbox é o que transforma a reentrega em uma operação nula.

Duplicados concorrentes

A sequência acima supõe que o duplicado chega depois de a primeira tentativa ter terminado. A concorrência é mais difícil, e a restrição de unicidade é o que salva você.

Imagine dois consumidores processando a mesma mensagem ao mesmo tempo. Os dois iniciam uma transação. Os dois fazem o trabalho. Os dois tentam inserir a linha do inbox. O banco de dados deixa exatamente um deles fazer commit; o outro recebe uma violação da restrição de unicidade, a sua transação sofre rollback e o seu lançamento contábil desaparece com ela. O perdedor então deixa a mensagem sem ack, ou dá ack depois de verificar de novo, e de qualquer jeito o ledger fica com um lançamento.

Esse é o mecanismo inteiro, e vale dizer o que ele não exige: nenhum bloqueio distribuído, nenhuma coordenação entre os dois consumidores, nenhuma suposição sobre temporização. A restrição do banco de dados é um ponto único de arbitragem pelo qual as duas tentativas precisam passar.

```
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
```

Decidir se uma violação de unicidade deve receber ack ou retry é uma pequena decisão que vale tomar de forma deliberada. Se a violação é na chave do inbox, a mensagem foi tratada por uma transação que sofreu commit, e dar ack nela é correto. Se é em alguma outra restrição dentro de `applyChange`, o erro significa algo totalmente diferente, e dar ack nele esconderia um bug de verdade. Na minha opinião, o handler deve distingui-las, o que ele pode fazer verificando o nome da restrição violada, como o código acima faz.

Handlers mais longos e trabalho em andamento

O padrão acima mantém o handler inteiro dentro de uma transação. Isso é aceitável quando o trabalho são algumas escritas. É desconfortável quando o handler chama algo lento, porque a transação permanece aberta, os blo-

queios permanecem retidos e o visibility timeout pode expirar enquanto você espera. A mensagem expirada é reentregue a um segundo consumidor, que inicia o mesmo trabalho lento em paralelo.

Há duas respostas sólidas. A primeira é manter a transação curta: faça o trabalho externo lento *fora* dela, usando a abordagem de efeitos externos da próxima seção, e ponha dentro da transação apenas a mudança de estado local e a linha do inbox. A segunda, quando o trabalho realmente precisa ser reivindicado, é tornar a reivindicação explícita com um status.

```
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)
```

Um consumidor insere a linha como `processing` com um lease, e essa reivindicação sofre commit em sua própria transação curta antes de o trabalho lento começar. Um duplicado que encontra um lease ativo não deve dar ack na mensagem, porque a primeira tentativa ainda pode falhar e o ack apagaria o único retry. Ele recoloca a mensagem na fila com um atraso, maior que o lease restante, e olha de novo na próxima entrega. Um que encontra um lease expirado pode assumir com um lease token mais novo, que o detentor original deve verificar antes de fazer commit das suas mudanças locais; um que encontra `done` dá ack.

O token protege apenas o que pode verificá-lo. A transação local pode: ela bloqueia a linha do inbox, compara o token e sofre rollback se existir um mais novo. A chamada externa lenta não pode sofrer rollback, de modo que o efeito recebe a chave determinística da próxima seção, e um detentor obsoleto que acorda e faz retry da chamada é reconhecido pelo sistema a jusante. Essa é a ideia de fencing do lease do relay do Capítulo 5, com o mesmo limite: um fence só funciona onde o recurso protegido verifica o token.

O esquema funciona e acrescenta uma máquina de estados sobre a qual alguém precisa raciocinar, inclusive sobre o que acontece quando um worker morre segurando um lease. Só recorreria a ele quando a primeira abordagem não estivesse disponível, e manteria o lease curto e renovável.

Aviso: Não conte com um `SELECT` prévio ou um bloqueio de linha para manter os duplicados de fora. Se um segundo consumidor enxerga o trabalho ainda sem `commit` do primeiro depende do nível de isolamento do banco de dados, e uma verificação correta em um nível falha em outro. Elevar o nível para serializable também não elimina a necessidade da restrição: ele troca um duplicado silencioso por uma falha de serialização, que o handler deve tratar como passível de `retry` e executar de novo desde o início. Os bloqueios reduzem a contenção; a restrição de unicidade dentro da transação é a garantia.

Janelas de deduplicação e retenção

Um inbox que guarda todos os ids para sempre cresce para sempre. Em algum momento você vai querer apagar linhas antigas, e a exclusão muda a garantia de uma forma que merece um nome: a **janela de deduplicação**. Uma mensagem cuja linha no inbox foi apagada pode ser processada de novo se algum dia reaparecer.

Então, qual deve ser o tamanho da janela? Ela precisa cobrir o maior tempo que um duplicado poderia plausivelmente levar para chegar depois do original. Não sei esse número para o seu sistema, e ele depende de uma lista curta de fatores.

- **Retenção do broker e horizonte de reentrega.** Uma fila que reentrega por quatro dias, ou um log que retém por uma semana, significa que podem chegar duplicados tão tardios quanto isso.
- **Prática de dead letters e redrive.** Se uma equipe puder fazer o `redrive` de dead letters uma semana depois de terem falhado, a janela precisa durar mais que isso (Capítulo 8).
- **Replay.** Rerun um log de eventos a partir de um offset antigo cria duplicados de eventos tão antigos quanto o que você repete. Ou a janela é longa o bastante, ou o `replay` é executado contra um consumidor deliberadamente reiniciado, o que é um procedimento que você projeta com antecedência.
- **Failover do relay.** As publicações duplas depois de um failover costumam estar próximas no tempo do original, de modo que ficam bem

dentro de qualquer janela que você escolheria.

Um formato comum é um período de retenção confortavelmente maior que a retenção do próprio broker, digamos algumas semanas a título de ilustração, com um job agendado que apaga em lotes. Particionar a tabela do inbox por tempo torna a exclusão barata, porque descartar uma partição inteira custa bem menos que apagar milhões de linhas.

Lembre-se de que o inbox não é a única linha de defesa. Uma restrição de chave de negócio na tabela de domínio vive tanto quanto os dados, de modo que protege contra duplicados que chegam depois que a linha do inbox já se foi. Este é mais um motivo para usar as duas camadas: o inbox dá proteção barata, com memória limitada, contra o caso comum, e a restrição cobre a cauda.

Se o sistema puder aceitar, há uma alternativa à janela de tempo: derivar a chave de algo monotônico. A regra é mais restrita do que parece à primeira vista. Ela se aplica a eventos que carregam estado, o tipo que o Capítulo 9 distingue dos deltas, em que cada evento contém o novo estado completo de um agregado. Para esses, o consumidor armazena a maior versão de agregado aplicada e rejeita tudo o que não for mais novo. O offset é uma posição dentro de uma partição e a versão é uma propriedade de um agregado, de modo que os dois não são intercambiáveis: um offset não diz nada sobre agregados que vivem em outras partições.

Para eventos delta, “rejeitar se não for mais novo” está errado, porque uma versão pulada significa uma mudança perdida. Um consumidor assim aplica a versão `last + 1`, guarda em buffer uma versão posterior até que a lacuna seja preenchida e trata qualquer coisa igual ou inferior a `last` como duplicado; o Capítulo 9 descreve o buffer e os seus limites. Em ambos os casos a memória é limitada, sem TTL, e depende de garantias de ordenação que o Capítulo 9 examina. Ela serve para agregados com um contador de versão e não serve para consumidores alimentados por várias partições.

Efeitos colaterais externos

O inbox protege efeitos que vivem no seu próprio banco de dados. Muitos efeitos não vivem. Um e-mail sai do seu sistema no instante em que é enviado, e uma chamada ao provedor de pagamento cobra um cartão quer a

sua transação sofra commit depois, quer não. Nenhuma transação local pode desfazer isso, de modo que o truque da atomicidade não se aplica.

Considere Payments tratando um comando `CapturePayment`. Ele precisa chamar o provedor externo e depois registrar o resultado.

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

Numa queda entre a chamada ao provedor e a transação, o dinheiro se moveu e Payments não tem registro. Na reentrega, o handler chama o provedor de novo. Se o comprador será cobrado duas vezes passa a depender inteiramente do provedor.

Passando a chave adiante

A correção é levar a chave de idempotência até o sistema que executa o efeito, para que ele possa reconhecer a repetição. Os provedores de pagamento costumam oferecer suporte a isso: você envia uma chave com a requisição, e uma segunda requisição com a mesma chave devolve o primeiro resultado em vez de agir de novo. Os detalhes do header ou do campo variam conforme o provedor, e leria a documentação do seu para saber o comportamento exato e por quanto tempo ele se lembra das chaves.

O detalhe que mais importa é *como você escolhe a chave*. Ela precisa ser **determinística a partir do trabalho, não gerada por tentativa**. Uma chave aleatória criada dentro do handler parece idêntica a uma requisição nova em cada retry e não protege nada.

```

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

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

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

```

Agora percorra as falhas de novo. Uma queda depois da chamada ao provedor e antes do commit deixa o dinheiro movido e Payments sem saber. O comando reentregue executa o mesmo código com a mesma chave, o provedor devolve o resultado original e a transação o registra. Uma queda durante a chamada ao provedor, em que a resposta se perde, é a mesma situação: o retry leva a mesma chave, e o provedor ou executa a captura uma vez ou devolve o resultado anterior. Dois consumidores disputando o mesmo comando enviam a mesma chave ao mesmo tempo, e um provedor que rejeita o segundo como “em andamento” não fez o trabalho falhar: o handler trata essa resposta como passível de retry e olha de novo na próxima entrega.

A memória do provedor e o inbox não são a mesma proteção. Um provedor se lembra das chaves por um tempo limitado, e leria a documentação dele para saber quanto. Uma reentrega que chegue depois desse período faria uma segunda captura. A linha local do inbox, mantida por pelo me-

nos o tempo da janela de deduplicação descrita antes, é o que impede o handler de chamar o provedor, de modo que a retenção de chaves do provedor nunca é a janela em que você se apoia.

Se o trabalho é identificado pelo negócio e não pela mensagem, use aqui também uma chave derivada do negócio, como o id do pagamento mais a operação. Assim, até dois comandos com ids diferentes chegam ao provedor como uma única requisição.

Quando o sistema a jusante não tem chave

Nem todo sistema a jusante aceita uma. O SMTP puro não tem chave de idempotência, e muitos provedores de e-mail e SMS não oferecem uma de forma confiável. Você então enfrenta uma escolha entre dois resultados imperfeitos, e a faria de forma explícita, por efeito.

Envie antes de registrar, e uma queda pode levar a um e-mail duplicado. Registre antes de enviar, e uma queda pode levar a um e-mail faltando. São os comportamentos *at-least-once* e *at-most-once* para o efeito. Para uma notificação de “sua encomenda está agendada”, um duplicado raro é barato e uma mensagem perdida é um ticket de suporte, de modo que enviaria primeiro e aceitaria a repetição ocasional. Para um código de voucher único que concede valor, pesaria para o outro lado e acrescentaria um jeito de detectar e reparar um envio ausente.

Alguns efeitos podem ser tornados mais seguros registrando a *intenção* na transação local e executando o efeito a partir de um worker separado, que é a ideia do outbox aplicada a comandos. Isso não elimina o problema na fronteira com o provedor, porque o worker ainda pode cair entre a chamada e a marcação. Mas move o problema para um único lugar onde você pode vê-lo, fazer retry dele deliberadamente e reconciliá-lo. O Capítulo 13 trata da reconciliação: comparar os seus registros com os do provedor e reparar as diferenças, que é a defesa final para exatamente os efeitos que não podem ser tornados idempotentes.

Idempotency-Key e APIs HTTP

O mesmo problema aparece na sua própria porta de entrada. O telefone do comprador envia “fazer pedido”, a rede derruba a resposta e o aplicativo faz retry. Orders não consegue distinguir um retry de um segundo pedi-

do, a menos que o cliente diga.

A convenção amplamente usada é um header de requisição `Idempotency-Key`: o **cliente** gera um valor único para cada operação lógica e envia o mesmo valor em todo retry dela. Essa é a outra metade da definição dada antes: no consumo de eventos, a chave normalmente é o `event_id`, que o produtor atribuiu, e, nas requisições voltadas ao cliente, é fornecida por quem chama. O trabalho do servidor é fazer uma chave repetida se comportar como a primeira requisição.

Um servidor precisa de mais do que a chave. Ele armazena a chave junto com um fingerprint da requisição e a resposta que produziu.

```

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

```

A verificação do fingerprint importa. Um cliente que reutiliza uma chave para um corpo diferente tem um bug, e devolver em silêncio a primeira resposta o esconderia. O estado em andamento importa para o caso da concorrência: duas requisições simultâneas com uma só chave não podem ser executadas ambas. O bloqueio de linha apenas ordena a retomada de uma linha existente; a primeira reivindicação é arbitrada pela restrição de unicidade, e é por isso que o insert perdedor é capturado e respondido com 409.

O lease importa para o caso da queda, e o seu token importa para o caso da lentidão. Um detentor pausado além do seu lease pode acordar e tentar terminar, mas `requireLeaseToken` enxerga o token mais novo e a transação inteira dele, pedido e linhas do outbox inclusos, sofre rollback. Aqui o token funciona como um fence porque o recurso protegido, o banco de dados local, o verifica dentro da transação, a mesma ideia do lease do relay do Capítulo 5. A duração do lease apenas encurta a sobreposição; o token é o que faz o commit do detentor obsoleto falhar. O chamador obsoleto recebe um erro, e o seu retry recebe a resposta armazenada. As duas verificações usam o relógio do banco de dados, `dbNow()`, e não o do servidor de aplicação, pelos motivos que o Capítulo 3 dá sobre relógios. Delimitar a chave pelo chamador impede que um cliente colida com outro.

O pseudocódigo mostra o caminho de sucesso, e o caminho de falha também precisa de uma regra. Se a requisição falha por um motivo determinístico, como um erro de validação que devolve um 4xx, o servidor armazena essa resposta como a concluída. Um retry com a mesma chave recebe então o mesmo 4xx e não executa a validação de novo contra dados que podem ter mudado. Se falha por um motivo transitório, como um timeout ou uma dependência indisponível, a reivindicação é liberada ou deixada expirar, para que o retry do cliente possa executar o trabalho. Guardar em cache uma falha transitória como resposta final transformaria uma oscilação em um erro permanente para aquela chave.

A restrição de unicidade em `(caller, key)` cumpre o mesmo papel de arbitragem que a chave primária do inbox. O padrão é o mesmo padrão: uma reivindicação atômica, um resultado armazenado e uma resposta determinística para uma repetição. Onde a API produz eventos, a linha do pedido, as linhas do outbox e a conclusão do registro de idempotência podem ser escritas em uma única transação local, de modo que uma requisição repetida não cria um segundo pedido nem emite um segundo `OrderPlaced`. Mantenha a chave armazenada por pelo menos o tempo em que os clientes podem fazer retry, o que é uma decisão sobre o comportamento dos clientes e não sobre os brokers.

Uma conexão que vale registrar: a chave fornecida pelo cliente pode fluir para dentro do evento. `Orders` pode pô-la nos headers do evento ou derivar dela o `event_id` de `OrderPlaced`, de modo que os consumidores a ju-sante vejam uma só identidade, do toque do comprador até o lançamento

no ledger. Se o `event_id` é derivado, ele deve vir de forma determinística do chamador e da chave em conjunto, por exemplo como um UUID baseado em nome sobre os dois, para que dois chamadores que por acaso escolham a mesma chave nunca compartilhem uma identidade. O Capítulo 11 discute o que pertence a um evento; se a chave pertence a ele depende de até onde você quer que a idempotência acompanhe a requisição.

Trade-offs e custos

O inbox acrescenta uma escrita a cada mensagem. Cada mensagem consumida custa um insert extra, dentro da mesma transação, no caminho crítico. Para um consumidor que faz um punhado de escritas, o overhead é pequeno. Para um consumidor de volume muito alto e muito barato, como um que atualiza o cache de um modelo de leitura, pode ser o custo dominante, e a idempotência natural costuma ser a melhor escolha ali.

O armazenamento e a limpeza do inbox são um trabalho contínuo. A tabela cresce com o tráfego, e a política de retenção é uma decisão que você assume e revisita. Uma janela curta demais readmite duplicados antigos; uma janela longa demais custa armazenamento e tamanho de índice. Alguém precisa vigiar o job de limpeza, porque, quando ele para, a tabela cresce em silêncio até virar um problema de desempenho.

Uma restrição de unicidade no caminho crítico pode virar um ponto de contenção. Com uma chave bem distribuída ela é barata. Com uma chave que se agrupa, ou com muitos consumidores atingindo um agregado quente, ela pode formar fila. Indexar a tabela por consumidor e id do evento e manter curta a transação do handler mantém isso sob controle.

As chaves de negócio são uma decisão de modelagem, e uma decisão errada é difícil de enxergar. Fina demais, e duplicados lógicos passam; grosseira demais, e trabalho legítimo é rejeitado. A falha é silenciosa nos dois sentidos. Testaria a chave com casos deliberadamente incômodos, como duas operações legítimas sobre um agregado, antes de depender dela.

A idempotência não cobre tudo. Ela torna uma repetição inofensiva; não conserta um handler que está errado já na primeira vez, não ordena mensagens e não desfaz um efeito para o qual o provedor não tem chave. Um

inbox também não protege contra um duplicado que *mudou*: se um produtor republica “o mesmo” fato com um `event_id` diferente e não existe chave de negócio, nada o reconhece.

Quando não usar o inbox. Se todos os efeitos do consumidor são naturalmente idempotentes e não podem ficar obsoletos, como sobrescrever um campo desnormalizado com o estado completo mais recente, o inbox é custo sem benefício, e o dispensaria e registraria o raciocínio no código. Ele também não é adequado a um consumidor que não pode escrever em um banco de dados transacional próprio, como um encaminhador sem estado. Esse consumidor precisa que o sistema a jusante faça a deduplicação, com a chave repassada, ou precisa ser redesenhado para ter um armazenamento próprio.

Erros comuns

- **Verificar e agir em passos separados.** É tentador porque o código se lê naturalmente: “já vi isto? não, então faça.” Falha diante de uma queda entre os dois passos e diante da concorrência. Ponha a escrita no inbox na mesma transação do efeito e deixe a restrição de unicidade arbitrar.
- **Escrever a linha do inbox depois do ack, ou antes de o efeito sofrer commit.** Dar ack primeiro perde a mensagem se o trabalho falhar depois. Registrar antes de o efeito sofrer commit marca como feito um trabalho que nunca aconteceu. O efeito, a linha do inbox e o commit são uma unidade; o ack vem depois do commit.
- **Manter o inbox em um armazenamento diferente do efeito.** Um cache ou um banco de dados separado parece mais barato e recria o dual write. Mantenha-o no mesmo banco de dados do estado que protege.
- **Gerar uma chave nova por tentativa.** Um valor aleatório criado dentro do handler ou do loop de retry do cliente faz cada retry parecer novo. Derive a chave do comando, do evento ou da operação de negócio, de modo que seja idêntica em toda tentativa.
- **Usar apenas o `event_id` e achar que ele cobre todos os duplicados.** Ele não pega duas mensagens diferentes com a mesma intenção.

Acrescente uma restrição de chave de negócio onde o domínio tiver esse risco.

- **Escolher uma janela de deduplicação por palpite.** Uma janela menor que a retenção do broker, ou que a sua prática de redrive, readmitte duplicados antigos em silêncio. Derive a janela desses números e registre a derivação por escrito.
- **Tratar uma violação de unicidade como um erro genérico.** Dar ack em toda violação esconde bugs reais; fazer retry de toda violação entra em loop sobre uma mensagem já tratada. Capture a violação do inbox em separado.
- **Esquecer o efeito externo.** O inbox não pode desfazer um e-mail nem a cobrança de um cartão. Passe uma chave determinística ao provedor, ou decida deliberadamente entre um possível duplicado e uma possível perda.
- **Publicar eventos de acompanhamento diretamente do handler.** Uma queda entre o commit e a publicação perde o evento, e a reentrega segue o caminho rápido do inbox e nunca o repete. Escreva o acompanhamento no outbox na mesma transação da linha do inbox e da mudança de estado.
- **Dar ack em uma mensagem cujo lease pertence a outro.** Um duplicado que vê um lease ativo e dá ack removeu o único retry se a primeira tentativa falhar em seguida. Recoloque-a na fila com um atraso.

Pontos-chave

- Os duplicados vêm do broker, do relay, dos retries, do redrive e do replay, e dos clientes, de modo que todo consumidor deve ser escrito como se qualquer mensagem pudesse chegar duas vezes.
- O check-then-act é inseguro porque a verificação e o efeito são passos separados, e uma queda ou uma tentativa concorrente pode se infiltrar entre eles.
- A linha do inbox e a mudança de negócio devem sofrer commit em uma única transação local no banco de dados do próprio consumidor, e a restrição de unicidade na chave é o que de fato arbitra os duplicados concorrentes. Um consumidor que anuncia o que fez acrescenta uma linha de outbox a essa mesma transação.

- Use o `event_id` como chave de idempotência de referência para o consumo de eventos e acrescente uma chave de negócio onde duas mensagens diferentes puderem expressar a mesma intenção.
- A janela de deduplicação deve durar mais que a retenção do broker, a sua prática de redrive e qualquer replay que você planeje, e uma restrição de chave de negócio na tabela de domínio protege você depois que as linhas do inbox são apagadas.
- Para efeitos colaterais externos, envie uma chave de idempotência determinística derivada do trabalho e não da tentativa, mantenha a linha local do inbox como a proteção de longo prazo porque o provedor esquece as chaves e, onde o sistema a jusante não tem chave, escolha conscientemente entre um possível duplicado e uma possível perda.
- Na fronteira da API, quem chama fornece a chave de idempotência, e o servidor armazena a chave, um fingerprint da requisição e a resposta, de modo que uma repetição devolve o primeiro resultado.

A idempotência torna inofensiva uma mensagem repetida, mas pressupõe que o processamento acaba tendo sucesso. Algumas mensagens nunca terão: falham em toda tentativa, e os retries e as reentregas se acumulam atrás delas. O Capítulo 8 trata dessas poison messages e das dead letters que elas deixam.

Capítulo 8 — Poison messages e dead letters

Imagine a Parcelly na manhã seguinte ao lançamento de um novo campo, `gift_message`, no `OrderPlaced`. O `Notifications` lê esse campo para montar o e-mail de confirmação, e o pedido de teste de um vendedor traz uma mensagem de presente com um caractere que o template do e-mail não consegue codificar. O consumidor lança uma exceção nessa única mensagem. A mensagem volta. O consumidor lança a exceção de novo.

Na hora do almoço, a mensagem já foi entregue vários milhares de vezes. Cada tentativa falha no mesmo ponto, do mesmo jeito, em milissegundos. O consumidor está ocupado, saudável por todas as métricas de infraestrutura, e há horas não dá um ack em nada útil. Em um broker baseado em filas, as demais mensagens continuam sendo atendidas pelos outros workers, então o estrago parece pequeno. Em um topic baseado em log, uma partição parou de andar porque o consumidor volta com um seek ao registro que falha em vez de passar por cima dele, e todo pedido naquela partição espera atrás dele.

Alguém acaba percebendo, e alguém sugere mover a mensagem para outro lugar para que o trabalho continue. Essa é a metade curta da resposta. A metade difícil é o que acontece depois: quem olha para essa mensagem, como decide o que ela significa e como ela volta ao sistema sem causar um segundo incidente. Este capítulo trata das duas metades. O que um sistema deve fazer com uma mensagem que não consegue processar, e como uma equipe evita que essa resposta vire um lixão?

O problema e por que ele é difícil

Uma **poison message** é uma mensagem que falha toda vez que é processada. A expressão “toda vez” carrega a definição. Uma mensagem que falha porque um banco de dados ficou brevemente inacessível não é poison; ela terá sucesso quando o banco voltar. Uma poison message falha porque

algo na mensagem, ou no código que a lê, torna o sucesso impossível até que uma pessoa ou um deploy mude alguma coisa. No vocabulário do Capítulo 3, a semântica da falha diz que tentar de novo não pode ajudar, e é exatamente por isso que fazer retry é esforço desperdiçado.

As causas usuais se dividem em alguns grupos:

- **Conteúdo malformado ou inesperado.** Falta um campo obrigatório, um valor está fora do intervalo ou um payload não é interpretado. Mudanças de esquema que não eram compatíveis com um consumidor antigo produzem isso (o Capítulo 12 mostra como evitar).
- **Um bug no consumidor.** A mensagem é válida, mas o código trata mal um caso que ninguém testou, como um nulo onde nunca se esperava um.
- **Uma suposição de negócio violada.** Um evento `PaymentCaptured` chega a um consumidor que não tem registro da autorização correspondente, talvez porque os dados foram apagados, ou porque os eventos se cruzaram em trânsito e os retries já ultrapassaram qualquer janela razoável de reordenação. Dentro dessa janela, o mesmo sintoma é transitório, como observa o Capítulo 6 e como o Capítulo 9 aprofunda, e pertence aos retries, não à DLQ. Passada a janela, o consumidor não consegue prosseguir e não pode inventar o registro que falta.
- **Uma rejeição permanente a jusante.** Uma transportadora externa recusa um endereço por considerá-lo inválido. Tentar de novo produz a mesma recusa.

O Capítulo 6 apresentou a distinção entre falhas transitórias e permanentes, e este capítulo continua de onde aquele termina. Uma política de retry que funciona corretamente vai, no fim, desistir de uma mensagem. Os retries são limitados por tentativas, por tempo ou por budget. O que acontece nesse ponto é uma decisão de design e, se ninguém a toma, o broker a toma por você.

O loop infinito de reentrega

Se o consumidor não faz nada de especial, o comportamento padrão do broker assume. Um broker baseado em filas reentrega uma mensagem sem ack, seja imediatamente depois de um nack, seja quando o visibility

timeout expira, e em geral mantém uma contagem de reentregas que cresce a cada tentativa.

Um broker baseado em log funciona de outro jeito, e a diferença importa. Não há reentrega por mensagem: o consumidor mantém um offset e lê para a frente. Deixar de fazer o commit de um offset não faz o broker devolver o registro; a posição do consumidor em memória continua avançando, e o registro que falhou é silenciosamente pulado até que a próxima reinicialização ou rebalanceamento faça o consumidor voltar ao último offset com commit. Um loop em um log só existe porque o consumidor o constrói: ele volta com um seek ao offset que falhou e o lê de novo, ou deixa de avançar a partição enquanto faz retry no mesmo lugar. Um consumidor que não faz nenhuma das duas coisas e, mesmo assim, faz o commit além da falha perde o registro sem emitir qualquer sinal.

Nas duas famílias o resultado pode ser o mesmo: a mensagem volta, falha e volta de novo ou, em um log com tratamento descuidado de offsets, desaparece. As ferramentas corretas da família de logs são um seek ou uma pausa explícitos, um retry topic que recebe o registro com falha para que a partição principal possa avançar, e um dead-letter topic para os registros dos quais se desistiu.

O loop tem três custos. O primeiro é a capacidade desperdiçada: os workers gastam seu tempo em trabalho que não pode ter sucesso. O segundo é o ruído: os logs se enchem de stack traces idênticos, e os problemas reais se escondem entre eles. O terceiro é o custo que mais importa, o **head-of-line blocking**. Em um log, um consumidor processa uma partição em ordem, então um registro que ele não pula bloqueia todos os registros que vêm depois. Em uma fila simples, uma poison message não bloqueia as outras por si só, mas um punhado delas pode ocupar todos os workers. Se a fila agrupa mensagens para fins de ordenação (filas FIFO, message groups, sessions), uma poison message bloqueia o seu grupo, o que tem o mesmo raio de impacto de uma partição de log, só que mais estreito.

As duas fugas ruins

Equipes sob pressão costumam recorrer a uma de duas fugas, e as duas estão erradas pelo mesmo motivo: escondem a falha em vez de tratá-la.

A primeira é **descartar a mensagem**: capturar a exceção, registrar em log, dar o ack e seguir em frente. O pipeline volta a fluir, mas o evento de negócio se foi. Para o `OrderPlaced`, isso significa um pedido que existe no Orders e em nenhum outro serviço. Para o `PaymentCaptured`, significa um ledger que não corresponde à realidade. Descartar só é a escolha correta quando a mensagem não carrega valor algum, e um consumidor não tem como saber isso no momento da falha.

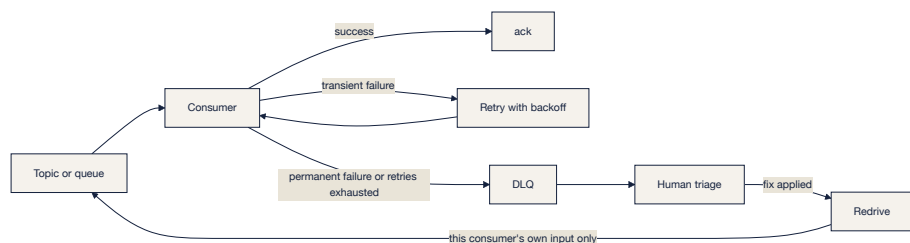
A segunda é **fazer retry para sempre** e confiar que alguém vai corrigir a causa. É o loop infinito descrito acima, e ele converte uma mensagem ruim em uma indisponibilidade para tudo o que vem depois dela.

A terceira opção fica entre as duas. Colocar a mensagem de lado, com contexto suficiente para que uma pessoa a entenda mais tarde, e deixar o resto do stream continuar. Essa opção é a dead-letter queue.

O padrão da dead-letter queue

Uma **dead-letter queue (DLQ)** é um destino separado onde um consumidor ou um broker coloca as mensagens das quais desistiu. Em um broker baseado em log, a mesma ideia costuma ser um **dead-letter topic**: um topic comum que recebe os registros com falha. Quando uma afirmação vale para as duas famílias, este capítulo diz DLQ.

O padrão tem um núcleo simples. Depois que uma mensagem esgota os retries, ou falha de um modo que o consumidor classifica como permanente, o consumidor a move para a DLQ e dá o ack na original. O stream continua. A mensagem que falhou não se perde; ela fica estacionada onde pode ser examinada.



Olhe para as duas últimas setas do diagrama. A DLQ não é uma saída. É uma sala de espera com uma porta de volta para a entrada do próprio consumidor que falhou, e toda a dificuldade deste capítulo está na distân-

cia entre essas duas portas. A porta não leva a um topic compartilhado: os outros consumidores já trataram o evento e não devem vê-lo de novo.

Onde o movimento acontece

Há duas maneiras de implementar o movimento, e a escolha afeta o que você consegue armazenar.

Dead-lettering gerenciado pelo broker. Muitos brokers baseados em filas podem ser configurados para que uma mensagem entregue mais de um certo número de vezes seja movida automaticamente para uma fila designada. Custa pouco configurar e funciona mesmo quando o consumidor sofre um crash antes de poder fazer qualquer coisa. Ele fornece uma contagem de entregas, mas em geral não o motivo da falha do consumidor, porque o broker não o conhece.

Dead-lettering gerenciado pelo consumidor. O consumidor captura a falha, monta um registro que inclui o motivo, o publica na DLQ e só então dá o ack na original. Isso permite classificar as falhas e enviar um erro permanente para a DLQ já na primeira tentativa, em vez de na quinta. Custa código em cada consumidor, e precisa ser escrito com cuidado: publique na DLQ primeiro e dê o ack depois, de modo que um crash entre os dois produza um duplicado na DLQ, nunca uma perda.

Usaria os dois onde o broker oferece o primeiro. A regra gerenciada pelo broker existe apenas em alguns brokers baseados em filas, e nem todo tipo de fila limita as entregas. Onde ela existe, é a rede de segurança para consumidores que sofrem crash ou travam de formas que o código não consegue capturar, e, junto com o caminho gerenciado pelo consumidor, faz com que uma poison message não possa ficar em loop para sempre, mesmo que o seu próprio tratamento tenha um bug.

Os brokers baseados em log em geral não têm nem um contador de entregas por mensagem nem uma DLQ no lado do broker, e isso deixa uma lacuna. Uma mensagem que derruba o processo, como uma falha do desserializador antes de o handler rodar, um erro de falta de memória ou um crash nativo, nunca chega ao seu bloco `try`. O consumidor reinicia no offset com commit, lê o mesmo registro e sofre crash de novo. O Capítulo 6 observou que a contagem de tentativas recomeça depois de um crash. Para um registro poison, esse é todo o problema.

O remédio é contar antes de processar, para que a contagem sobreviva ao crash:

```
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)
```

O contador é indexado por partição e offset (ou por `event_id`, se você consegue lê-lo sem desserializar). O preço é uma escrita durável por registro no caminho crítico, e um limite que precisa ser alto o bastante para que uma reinicialização comum de deploy não empurre registros saudáveis para além dele. A alternativa é aceitar que um registro poison em crash loop paralise a sua partição até alguém perceber, e é por isso que o alarme de atraso do consumidor descrito mais adiante importa.

```
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)
```

A etapa `deadLetter` é onde se captura o contexto necessário para a triagem, e a ordem das suas duas últimas linhas é o ponto central.


```

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)

```

Se o próprio `publish` falha, `deadLetter` lança uma exceção e a mensagem não recebe ack. Em um broker da família de filas, isso basta: a mensagem volta pela reentrega normal. Em um broker da família de logs, não basta, porque um offset sem commit não traz o registro de volta; o consumidor precisa voltar com um seek à posição do registro que falhou, ou pausar a partição, e tentar de novo a publicação na dead letter antes de ler mais adiante. De um jeito ou de outro, o objetivo é o mesmo modo de falha: a mensagem permanece no sistema até ter uma nova casa segura.

O que armazenar junto com uma dead letter

Uma dead letter sem contexto é um quebra-cabeça. Uma equipe que encontra uma mensagem em uma DLQ seis semanas depois precisa responder a quatro perguntas: o que era, de onde veio, por que falhou e ainda é seguro processá-la? O registro deve respondê-las sem que seja preciso abrir um repositório de código-fonte.

Campo	Por que está aí
Corpo e headers originais, byte a byte	O redrive precisa reproduzir a mensagem exatamente, inclusive o <code>event_id</code>
Nome e posição de origem (topic, partição e offset, ou nome da fila)	Localiza a mensagem no histórico, apoia decisões de ordenação
Chave original (chave do registro ou routing key) e timestamp original	O redrive precisa publicar com a mesma chave, ou a mensagem cai em outra partição e perde a sua ordem por chave
Entrada do consumidor (a fila própria ou o redrive topic do consumidor que falhou)	O único lugar onde um redrive pode publicar
<code>redrive_count</code>	Separado de <code>attempt_count</code> : quantas vezes esta mensagem já voltou
<code>error_class</code> e <code>error_message</code>	Agrupa as falhas por causa; trunque as mensagens longas
Stack trace ou um ponteiro para a linha de log	Permite que um engenheiro veja onde falhou sem reproduzir o problema
<code>attempt_count</code>	Distingue “falhou uma vez, permanente” de “falhou depois de muitos retries”
<code>first_failed_at</code> e <code>failed_at</code>	Mostra há quanto tempo a mensagem espera e quando o problema começou
Nome e versão do consumidor	Liga a falha a um deploy; muitas vezes é o caminho mais rápido até a causa
<code>correlation_id</code>	Conecta a mensagem ao resto do seu fluxo de negócio

Duas ressalvas se aplicam. Primeiro, mantenha o payload original sem modificação. Se você o envolve em um envelope próprio, a ferramenta de redrive precisa desembulhá-lo exatamente, ou o `event_id` e o `causation_id` vão divergir e as garantias de idempotência do Capítulo 7 se perdem. Segundo, lembre que uma DLQ é uma cópia de dados de produção, muitas vezes mantida por muito tempo. Ela herda os controles de acesso e as obrigações de retenção do original, e payloads com dados pessoais exigem na DLQ o mesmo cuidado que em qualquer outro lugar.

Parking lot versus DLQ

A palavra DLQ é usada para duas coisas diferentes, e separá-las torna a operação mais fácil de raciocinar.

Um **parking lot** é um local de espera para dead letters que precisam de uma decisão humana. A DLQ é onde o sistema coloca uma mensagem automaticamente; o parking lot é onde uma pessoa coloca uma mensagem de propósito, depois de olhá-la e decidir que um redrive não vai resolvê-la. É um lugar distinto, com um propósito distinto.

Considere o que cada um contém. A DLQ guarda mensagens que falharam e ainda não foram examinadas. Ela deve se esvaziar: seu estado esperado é vazia ou quase vazia, e qualquer coisa antiga nela é um sintoma. O parking lot guarda mensagens que alguém examinou e deixou de lado: um pedido que foi cancelado antes de o seu evento chegar, um payload que precisa de correção manual, um registro que tem de ser reconciliado à mão com outra equipe. Essas mensagens não estão esperando que uma correção seja implantada. Estão esperando uma decisão.

Misturá-los produz uma falha previsível. Se as mensagens já decididas e estacionadas permanecem na DLQ, a profundidade da DLQ nunca volta a zero, o limite do alerta é elevado para acabar com o ruído, e a próxima poison message de verdade chega sem ser notada. Mantê-los separados permite que o alarme da DLQ continue afiado.

Um parking lot pode ser tão modesto quanto uma tabela de banco de dados com um status, um responsável, um motivo e uma data de revisão. Ele nem precisa ser um destino do broker, porque nada vai consumi-lo automaticamente. A data de revisão é o que mais importa: uma mensagem estacionada sem data para a próxima olhada já foi esquecida.

Responsabilidade e alarmes

Uma DLQ sem responsável é um ralo que se enche em silêncio. A primeira decisão, antes de qualquer ferramenta, é quem responde por cada uma.

Meu padrão é que a equipe responsável pelo consumidor é responsável pela DLQ. Ela escreveu o código que falhou, sabe o que a mensagem deveria ter significado e é quem pode corrigir e implantar. Uma DLQ por consumidor, ou por grupo de consumidores e origem, torna essa responsabili-

dade evidente. Uma única DLQ compartilhada para a empresa inteira a torna ambígua, e ambiguidade significa que ninguém olha. Onde uma equipe de plataforma fornece as ferramentas, a equipe de plataforma é responsável pelo mecanismo e a equipe consumidora é responsável pelo conteúdo.

Sobre o que alarmar

Alarme na **chegada**, não só na profundidade. Uma única mensagem aparecendo em uma DLQ que normalmente está vazia merece uma notificação, porque a primeira dead letter é o momento mais barato para encontrar um bug. Um limite de profundidade sozinho não diz nada até que a fila já esteja funda.

Três sinais juntos cobrem a maioria dos casos:

- **Qualquer nova chegada** em uma DLQ que deveria estar vazia. Encaminhe isso a um canal que uma pessoa leia durante o horário comercial.
- **Taxa de chegada.** Uma rajada repentina costuma significar um deploy ruim ou uma mudança de esquema incompatível, e justifica enviar um page para alguém.
- **Idade da mensagem mais antiga.** Isso pega a falha lenta: a DLQ que recebe uma mensagem por dia, sobre a qual ninguém age, até que a mais antiga tenha um mês e o contexto de negócio tenha se perdido.

Não sei qual é o limite certo de page para o seu sistema, e ninguém pode dizê-lo sem conhecer o valor de negócio das mensagens em questão. Uma DLQ ligada aos consumidores de `PaymentCaptured` merece um limite mais baixo que uma ligada ao `Notifications`, porque um e-mail perdido é um incômodo e uma captura perdida é uma discrepância no ledger. Os limites devem refletir essa diferença, e a diferença deve estar escrita ao lado do alarme.

Nota: Alarme também no atraso do consumidor. Um consumidor preso em um registro poison em uma partição de log não necessariamente produz dead letters, porque nunca desiste. Atraso do consumidor crescente sem throughput é o sinal de que o loop descrito no início deste capítulo está em execução.

Redrive: trazendo as mensagens de volta

Redrive significa devolver as dead letters ao consumidor que falhou nelas, para reprocessamento. O alvo é a entrada do próprio consumidor: seu `retry topic` ou `redrive topic`, ou a sua própria fila. Nunca é o `topic` compartilhado de onde a mensagem veio originalmente, porque todos os outros consumidores desse `topic` já processaram o evento, e republicar ali o entregaria a todos eles uma segunda vez. A chave original viaja com a mensagem, de modo que ela fica na mesma partição e mantém a sua posição por chave. O contador de tentativas recomeça em zero, e o `redrive_count`, separado, sobe uma unidade, de modo que uma mensagem que não para de voltar fique visível em vez de parecer nova a cada vez. O `redrive` é a operação que torna uma DLQ valiosa e também a que mais provavelmente causa um segundo incidente. O Capítulo 3 descreveu o engenheiro que aperta um botão e faz o `redrive` de 40.000 mensagens de uma vez. O remédio é tratar o `redrive` como um procedimento projetado, não como um botão de conveniência.

Corrija a causa antes do redrive

Uma mensagem que falhou por causa de um bug no consumidor vai falhar de novo até que o bug seja corrigido e implantado. Fazer o `redrive` antes da correção move as mensagens em círculo e reabastece a DLQ com as mesmas entradas, agora com contagens de tentativas maiores e um histórico confuso. A ordem do trabalho é: entender a causa, implantar a correção, fazer o `redrive` de uma pequena amostra, ver que ela tem sucesso e só então fazer o `redrive` do resto.

O redrive precisa ser seguro para repetir

Uma mensagem que passou por `redrive` é um duplicado por definição: o sistema já viu esse `event_id` antes, e o consumidor pode tê-lo visto também. Se o processamento original teve sucesso parcial antes de falhar, alguns efeitos podem já existir. O Capítulo 7 é o motivo pelo qual isso é sobrevivível. Um consumidor idempotente com um `inbox` trata a mensagem reenviada como qualquer outra reentrega e aplica os seus efeitos uma única vez.

Vale registrar dois pontos. O redrive precisa preservar o `event_id` original, já que ele é a chave de idempotência. Uma ferramenta de redrive que gera um identificador novo para cada mensagem derrota o inbox, e o efeito pode ser aplicado duas vezes. E um consumidor que não é idempotente não deveria ser alvo de um redrive em massa: corrija isso primeiro, ou aceite que cada mensagem reenviada é uma avaliação manual de risco.

Redrive em lotes controlados

Uma ferramenta de redrive segura tem algumas propriedades, nenhuma delas exótica:

1. **Um filtro de seleção.** Faça o redrive por classe de erro, janela de tempo ou `correlation_id`, e não apenas “tudo”.
2. **Um rate limit.** As mensagens voltam à entrada do consumidor em um ritmo que ele e as suas dependências conseguem absorver. Uma enxurrada de mensagens reenviadas é um retry storm autoinfligido, e os limites de requisições em andamento e o load shedding de “Retries sob sobrecarga: backpressure” (Capítulo 6) se aplicam ao tráfego do redrive exatamente como ao tráfego ao vivo.
3. **Uma chave de pausa.** Um operador pode interromper o redrive no meio, se as falhas reaparecerem.
4. **Um dry-run.** A ferramenta informa o que moveria antes de mover qualquer coisa.
5. **Um registro de auditoria.** Quem fez o redrive de quê, quando e por quê, guardado em algum lugar durável.

```

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"

```

Observe a ordem dentro do loop: um registro é marcado como reenviado depois que a publicação tem sucesso, não antes. Se a ferramenta sofre um crash no meio, algumas mensagens passam por redrive e continuam sem marca na DLQ, o que leva a duplicados, não a perdas. Um consumidor com inbox absorve esses duplicados; um sem ele transforma um crash da ferramenta em um problema de dados. O registro é marcado, não apagado, para que a trilha de auditoria e o `redrive_count` sobrevivam; a limpeza segue a regra de retenção em “Evitando o lixo”, mais adiante.

Redrive e ordenação

O redrive quebra a ordenação a menos que você projete para ele. Uma mensagem enviada para a DLQ foi pulada; as mensagens posteriores do mesmo agregado foram processadas nesse meio-tempo. Quando a mensagem reenviada chega, chega depois delas. É o exemplo do Inventory do Capítulo 6 com uma lacuna maior: se o `OrderPlaced` é enviado para a DLQ em vez de passar por `retry`, o `OrderCancelled` continua sem encon-

trar reserva para liberar, e a colocação reenviada, horas depois, continua criando uma que nada vai liberar. Preservar a chave não conserta isso: a chave põe a mensagem na partição certa, mas a partição seguiu em frente.

Se isso importa depende do consumidor. Um consumidor **comutativo**, ou seja, cujo estado final é o mesmo seja qual for a ordem de chegada dos eventos, não é afetado. Um consumidor que depende de transições de estado é. O Capítulo 9 mostra como detectar e tratar a chegada fora de ordem com números de versão ou verificações de sequência, e um consumidor baseado em estado deveria usá-los antes que alguém faça um redrive para dentro dele. Se não os usa, o procedimento de redrive precisa incluir uma verificação de que o mundo não seguiu em frente desde que a mensagem foi enviada para a DLQ.

O que uma dead letter faz com um stream ordenado

Streams ordenados forçam a decisão mais difícil deste capítulo, porque a escolha que protege a ordenação é a que para o negócio.

Considere um topic baseado em log que usa `order_id` como chave, no qual o Ledger registra lançamentos. Os consumidores processam cada partição em sequência. A mensagem 41 de `order-81724` falha permanentemente. O consumidor tem duas opções:

- **Parar e esperar.** Segurar a partição na mensagem 41 até que alguém corrija a causa. A ordenação é preservada exatamente. Todo outro pedido que cai nessa partição por hash também para, embora as suas mensagens estejam boas. O raio de impacto é a partição inteira.
- **Enviar para a DLQ e continuar.** Estacionar a mensagem 41, processar a 42 e as seguintes. A partição flui. Mas a 42 pode ter dependido da 41, e o consumidor a terá processado contra um estado que não inclui a 41.

Nenhuma das escolhas é gratuita. Parar troca disponibilidade por correção; continuar troca correção por disponibilidade. Não existe configuração que dê as duas, e é melhor escolher de propósito para cada consumidor do que herdar o que quer que o padrão do framework seja.

Minha regra prática tem três partes:

1. **Se o consumidor é insensível à ordem ou comutativo**, envie para a DLQ e continue. O Notifications é o caso típico: um e-mail que falhou em um pedido não tem relação com o próximo.
2. **Se mensagens posteriores da mesma chave dependem da que falhou**, não continue para essa chave. Envie a mensagem que falhou para a DLQ e desvie também as mensagens seguintes da mesma `partition_key` para uma área de retenção até que a primeira seja resolvida. As outras chaves continuam. Isso mantém o raio de impacto em um agregado em vez de em uma partição, ao preço de um mecanismo adicional: o rastreamento por chave das chaves “bloqueadas”.
3. **Se o consumidor é um ledger financeiro, pare**. O custo de processar fora de ordem é um registro que está errado de um modo difícil de detectar e caro de reparar, e uma partição pausada é ruidosa e óbvia. Correção acima de velocidade foi a regra declarada para o Ledger, e aqui ela se aplica na sua forma mais simples.

A variante por chave da segunda regra merece um esboço, já que é a mais útil e a menos comum. Ela pressupõe uma pequena tabela `key_block` com uma linha por chave bloqueada (`partition_key`, `blocked_event_id`, `state`), e que um consumidor por vez trate o tráfego ao vivo de uma dada chave, que é o que uma partição ou um message group oferece.

```

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)

```

A liberação é um procedimento separado, e um sweep periódico a torna reiniciável.

```

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)

```

Três detalhes desse esboço decidem se ele funciona. O primeiro é o próprio redrive. A mensagem reenviada é justamente aquela em que a chave está bloqueada, então ela precisa contornar o bloqueio: a verificação com-para o seu `event_id` com o do bloqueio. Sem essa exceção, o redrive fica estacionado atrás das mensagens que deveria preceder, e a chave nunca é desbloqueada.

O segundo é a corrida com o tráfego ao vivo. Enquanto `resolveBlockedKey` esvazia as mensagens retidas, novas mensagens da mesma chave continuam chegando. Elas ainda precisam ser retidas, não processadas, até que o conjunto retido esteja vazio. Isso só vale se a decisão de estacionar e a decisão de liberar forem serializadas: ambas tomam o bloqueio de linha da chave, e o estacionamento é gravado na mesma transação que a verificação. Se o estacionamento fosse uma etapa separa-

da, uma mensagem ao vivo poderia ver uma chave bloqueada, depois encontrar o conjunto retido já esvaziado e a chave liberada, e acabar estacionada atrás de um bloqueio que não existe mais. Ninguém a esvaziaria, e mensagens posteriores seriam processadas à frente dela.

O terceiro é um crash no meio da liberação. Depois que a mensagem reenviada é processada, o estado `resolving` é gravado antes do ack, de modo que o trabalho a fazer fica registrado no banco de dados e não apenas em uma mensagem que nunca mais virá. O sweep recolhe qualquer chave deixada nesse estado e continua; uma mensagem processada pouco antes do crash é absorvida pelo inbox. Se uma mensagem retida falha permanentemente durante a liberação, a chave volta a ser bloqueada, ancorada nessa mensagem, com a sua própria dead letter, e o resto do conjunto retido espera pelo seu redrive. O bloqueio por chave é um loop, não um evento único.

O esboço não é completo. Um sweep pode se sobrepor a uma liberação ao vivo, de modo que dois resolvedores trabalhem na mesma chave. A verificação de estado sob o bloqueio impede que qualquer um deles ultrapasse um novo bloqueio: quando um resolvedor ancora o bloqueio em uma mensagem que falhou, o outro encontra a chave já fora de `resolving` e para. O que resta é trabalho duplicado sobre a mesma mensagem retida, que o inbox absorve; um lease sobre o estado `resolving` pouparia esse trabalho, não a correção. Ele também não impõe limite ao conjunto retido, e deixa de fora como um operador descarta uma mensagem retida que já não faz sentido. Trate-o como a forma do mecanismo, não como um mecanismo pronto.

Isso é maquinário real, com modos de falha próprios, como o armazenamento de mensagens retidas crescer sem limite, então o construiria apenas para streams em que a ordem por chave importa e a taxa de dead letters é baixa o bastante para que o conjunto retido continue pequeno. É também um mecanismo que não sabe se as mensagens retidas ainda fazem sentido depois que a que falhou foi resolvida. Quando as próprias transições de estado precisam ser validadas, e não apenas sequenciadas, o version buffer do Capítulo 9 é a ferramenta mais forte: ele retém um evento até que o seu predecessor tenha sido de fato aplicado, por número de versão, e funciona sem um registro de dead letter para ancorá-lo.

Evitando o lixo

Uma DLQ vira um lixo quando as mensagens entram mais rápido do que alguém decide o que elas significam. O sintoma é familiar: milhares de entradas, muitos erros diferentes, ninguém sabendo quais importam e uma regra não escrita de que não se mexe nela. Nesse ponto a DLQ deixou de ser um mecanismo de segurança; é um lugar onde dados perdidos são guardados com um nome educado.

Lixões têm causas, e cada uma tem o seu contra-ataque.

- **Sem responsável.** O contra-ataque é a regra de responsabilidade acima, escrita.
- **Sem cadência de triagem.** As dead letters devem ser revisadas segundo um calendário, e não apenas quando um alarme dispara. Para uma DLQ de baixo volume, uma revisão semanal é um ponto de partida razoável.
- **Retenção que sobrevive à utilidade, ou que expira antes da triagem.** Uma dead letter de uma reserva de estoque que expirou no mês passado não pode ser reenviada de forma significativa. Defina um período de retenção com base em quanto tempo a mensagem permanece válida para o negócio, e decida de antemão o que acontece na expiração: arquivar, descartar com uma entrada de auditoria ou escalonar. O erro oposto é tão comum quanto: uma retenção menor que a cadência de triagem apaga mensagens antes que alguém as tenha visto. Se a DLQ é uma fila do broker cuja retenção é menor que o seu intervalo de revisão, o broker remove a evidência primeiro. Defina a retenção da DLQ acima do maior intervalo entre revisões e leia a documentação da sua plataforma sobre a partir de quando o relógio da retenção começa a contar. Se ele começa no momento original do enfileiramento, e não no momento do dead-lettering, uma mensagem que esperou muito tempo antes de falhar chega já perto da expiração. É mais um motivo para o registro guardar o timestamp original.
- **Tudo vai para a DLQ.** Se resultados rotineiros de negócio, como uma rejeição de validação que é parte normal do fluxo, são enviados para a DLQ, as falhas reais se afogam. Um pedido rejeitado é um evento de negócio com tratamento próprio, não uma dead letter.

- **Falhas transitórias enviadas cedo demais para a DLQ.** Se a política de retry desiste depois de segundos durante uma indisponibilidade breve, toda mensagem em trânsito durante essa indisponibilidade acaba na DLQ. O remédio está no Capítulo 6: uma política de retry cujas tentativas e deadline sejam dimensionados para a indisponibilidade que você espera sobreviver.

Também daria a toda dead letter um desfecho. A triagem termina com um de uma lista curta de resultados: reenviada depois de uma correção, corrigida e reenviada, estacionada para uma decisão manual ou descartada com um motivo registrado. “Ainda está lá parada” não é um desfecho.

Trade-offs e custos

Uma DLQ move uma falha de um lugar onde ela é ruidosa, o consumidor bloqueado, para um lugar onde ela é silenciosa. Isso é ao mesmo tempo o benefício e o perigo. O stream flui, e a mensagem que falhou pode ser esquecida. Tudo neste capítulo que não é a DLQ em si, os alarmes, o responsável, a cadência de revisão e a retenção, existe para contrabalançar esse silêncio.

Custos específicos a pesar:

- **Carga operacional.** Alguém tem de fazer a triagem, decidir e fazer o redrive. Uma DLQ para a qual ninguém tem tempo é pior que nenhuma, porque esconde o problema atrás da aparência de um mecanismo.
- **Governança de dados.** As dead letters guardam payloads de produção, às vezes com dados pessoais, fora do ciclo de vida usual. Elas precisam de controles de acesso, regras de retenção e caminhos de exclusão.
- **Risco de ordenação.** Enviar para a DLQ uma mensagem de um stream ordenado a pula. A menos que você construa o bloqueio por chave ou pare a partição, os eventos posteriores são aplicados sem ela. O próprio bloqueio por chave acrescenta um armazenamento de mensagens retidas, um desvio para o redrive, um loop de liberação que precisa disputar com segurança com o tráfego ao vivo e um sweep que reinicia uma liberação interrompida.
- **Identidade armazenada.** Um redrive seguro precisa da chave, da partição e da entrada do consumidor originais em cada registro, e de um

`redrive_count` ao lado da contagem de tentativas. É mais coisa para armazenar e manter consistente, e sem isso o `redrive` só pode ser um palpite.

- **Investimento em ferramentas.** Uma ferramenta de `redrive` segura, com filtros, rate limits, desfecho por registro e trilha de auditoria, é um pequeno projeto. Comprá-la ou construí-la faz parte do custo de ter uma DLQ.

Quando não usar uma: se uma mensagem pode ser regenerada ou a sua perda é inofensiva, uma DLQ acrescenta mais processo do que valor. Um evento de aquecimento de cache ou uma atualização puramente cosmética pode ser descartado, com uma métrica contando os descartes. E onde a resposta certa à falha é parar, como no exemplo do ledger, uma DLQ que deixa o stream continuar trabalha contra você. Nesse caso, pare o consumidor, alerte em voz alta e corrija a causa.

Há também um limite para o que o padrão pode fazer. Uma DLQ guarda mensagens que falharam de forma visível. Ela não faz nada pelas mensagens que foram processadas incorretamente sem lançar um erro, que é o território do Capítulo 13 e do seu trabalho de reconciliação.

Aviso: Nunca dê o `ack` na mensagem original antes de a dead letter estar armazenada de forma durável. Se a publicação na DLQ falha depois do `ack`, a mensagem não existe em lugar nenhum, e nenhum alarme vai avisar.

Erros comuns

- **Enviar para a DLQ sem contexto.** É tentador porque mover o corpo bruto é o código mais curto. A correção é o registro de metadados descrito acima, incluindo os headers originais exatos, para que o `redrive` e o diagnóstico sejam ambos possíveis.
- **Fazer retry de falhas permanentes.** Um erro de parsing não melhora com o tempo, mas uma política de retry genérica o trata como um timeout. Classifique as falhas e envie as permanentes direto para a DLQ em vez de gastar o retry budget.
- **Fazer o `redrive` antes de corrigir a causa.** A pressão para “fazer as mensagens andarem” é real, e o resultado são as mesmas mensagens

de volta na DLQ. Implante a correção, faça o redrive de uma amostra e depois do resto.

- **Fazer o redrive de tudo de uma vez.** Uma enxurrada de mensagens antigas compete com o tráfego ao vivo e pode iniciar um novo incidente. Use rate limit, lotes e mantenha uma chave de pausa.
- **Alarmar só pela profundidade.** Parece a métrica óbvia, mas dispara tarde. Alarme na primeira chegada, na taxa e na idade da mensagem mais antiga.
- **Uma DLQ compartilhada para tudo.** É barata de criar e impossível de manter sob responsabilidade. Dê a cada consumidor a sua própria e atribua o responsável por nome.
- **Misturar mensagens estacionadas e mensagens não examinadas.** A DLQ nunca se esvazia, o alerta é afrouxado e a poison de verdade passa despercebida. Mantenha um parking lot separado, com responsáveis e datas de revisão.
- **Ignorar a ordem no redrive.** Uma mensagem reenviada chega depois dos eventos posteriores do mesmo agregado. Use as verificações de versão do Capítulo 9 ou confirme que o estado não seguiu em frente.
- **Fazer o redrive para o topic compartilhado.** Parece “pôr a mensagem de volta de onde ela veio” e entrega o evento uma segunda vez a todos os outros consumidores desse topic. Faça o redrive para a entrada do próprio consumidor que falhou, com a chave original.
- **Esperar para sempre por um registro em crash loop.** Em um broker de log, um consumidor que nunca desiste não produz dead letters nem alarme algum além do atraso do consumidor. Conte as tentativas de forma durável por partição e offset, para que uma reinicialização não zere a contagem.

Pontos-chave

- Uma poison message falha toda vez que é processada, então fazer retry dela para sempre transforma uma mensagem ruim em uma indisponibilidade para tudo o que está enfileirado atrás dela.
- Descartar uma mensagem que falhou e fazer retry dela para sempre são duas formas de esconder uma falha; uma dead-letter queue coloca

a mensagem de lado, com contexto, para que o resto do stream possa continuar.

- Uma dead letter deve carregar o corpo e os headers originais inalterados, a sua posição de origem, o erro, a contagem de tentativas e a versão do consumidor, porque a pessoa que a lê semanas depois só tem esse registro.
- Publique na DLQ antes de dar o ack na original, para que um crash produza um duplicado recuperável e nunca uma mensagem perdida.
- Toda DLQ precisa de uma equipe responsável nomeada, de um alarme na primeira chegada, na taxa e na idade, e de uma cadência regular de triagem, ou vira um lixão.
- Um redrive só é seguro quando a causa está corrigida, o `event_id` e a chave originais são preservados, a mensagem vai para a entrada do próprio consumidor que falhou e nunca para um topic compartilhado, o consumidor é idempotente e as mensagens voltam em lotes com rate limit e uma chave de pausa.
- Em um stream ordenado, enviar uma mensagem para a DLQ força uma escolha entre parar a partição e continuar além de uma lacuna; escolha por consumidor, de propósito, pare para ledgers e bloqueie por chave apenas onde o maquinário vale o seu custo.

O dead-lettering e o redrive dependem ambos de uma pergunta que este capítulo foi adiando: o que significa as mensagens chegarem na ordem certa, e o que se pode construir quando isso não acontece? O Capítulo 9 enfrenta essa pergunta diretamente.

Capítulo 9 — Ordenação

Imagine uma noite de quinta-feira na Parcelly. Um comprador faz um pedido, muda de ideia em questão de segundos e o cancela. O Orders emite `OrderPlaced`, depois `OrderConfirmed`, depois `OrderCancelled`, todos para `order-81724`, e todos na ordem certa. O relay também os publica nessa ordem.

O topic que alimenta o Shipping foi criado com uma configuração padrão: as mensagens se espalham por seis partições sem levar em conta o seu conteúdo. O `OrderConfirmed` cai na partição 2 e o `OrderCancelled` na partição 5. O Shipping roda um consumidor por partição. O consumidor da partição 5 está ocioso e rápido, então lê o cancelamento primeiro e encontra um envio que ainda não existe. Dá de ombros e dá o ack. Um instante depois, o consumidor da partição 2, que estava ocupado com um lote lento, lê a confirmação e agenda um envio para um pedido que já não existe.

Nada sofreu crash, nenhuma mensagem se perdeu e nenhuma foi duplicada. Cada componente fez o que estava configurado para fazer. Uma transportadora vai chegar na segunda-feira para recolher um pacote que ninguém pediu. Quanta ordem um sistema como esse precisa, onde ela se perde e o que se pode construir quando não se pode tê-la?

O problema e por que ele é difícil

A ordem não é uma propriedade única que um sistema tem ou deixa de ter. É uma cadeia, e cada elo pode rompê-la de forma independente. O relay pode publicar linhas em uma sequência diferente da ordem em que o banco de dados fez o commit (o Capítulo 4 e o Capítulo 5 tratam desse elo). O broker pode espalhar mensagens relacionadas por partições ou filas. Os consumidores podem processar em paralelo. Um retry pode colocar uma mensagem atrás das suas sucessoras (Capítulo 6), e uma dead letter pode pulá-la por completo (Capítulo 8).

Elo	Como a ordem se perde	Onde olhar
Produtor e relay	Relays paralelos, commits concorrentes, ids que não correspondem à ordem de commit	Capítulos 4 e 5
Broker	Mensagens relacionadas em partições ou filas diferentes	Este capítulo
Consumidor	Vários workers tratando mensagens de uma mesma partição ou fila	Este capítulo
Tratamento de falhas	Retry topics, nacks, dead letters, redrive	Capítulos 6 e 8
Mudança de topologia	Mudança na contagem de partições, rebalanceamentos de consumidores	Este capítulo

A resposta óbvia é exigir uma ordem global: todo evento do sistema, em uma única sequência, visto por todos os consumidores. Soa como o padrão seguro. É caro e, para a maioria dos sistemas, desnecessário.

Por que a ordem global é cara

Uma única ordem total significa um único ponto de sequência. Em um broker baseado em log, isso é uma partição, que é lida por no máximo um consumidor de um grupo de consumidores por vez. O seu teto de throughput passa a ser a velocidade de um escritor e de um leitor. Acrescentar capacidade não adianta nada, porque a capacidade vem de executar coisas em paralelo, e o paralelismo é justamente o que uma ordem total proíbe.

Um sistema que abrange vários serviços piora isso. Orders, Payments e Inventory têm cada um o seu banco de dados e o seu outbox. Uma ordem global entre todos eles exigiria um coordenador que carimbasse todo evento, e esse coordenador ficaria no caminho de escrita de todos os serviços. Você estaria comprando de volta o acoplamento que a arquitetura foi projetada para eliminar.

Por que a ordem global é, em geral, desnecessária

Pergunte o que o negócio realmente precisa. À Parcelly não importa se o `PaymentCaptured` de `order-81724` veio antes do `OrderPlaced` de `order-90311`. Esses pedidos não influenciam um ao outro. O que importa é que, para um pedido, o `OrderPlaced` seja visto antes do `OrderConfirmed`, e o `OrderConfirmed` antes do `OrderCancelled`.

Isso é a **ordenação por agregado**. Um agregado é a fronteira de consistência cujo identificador serve como chave de partição. Se todo evento sobre um agregado é visto na ordem em que foi produzido, e os eventos sobre agregados diferentes ficam livres para se intercalar, a maioria das máquinas de estados funciona corretamente. Essa é a ordem que você pode pagar e, na minha opinião, a ordem em torno da qual você deve projetar.

Há uma fronteira que nem a ordem por agregado atravessa. Eventos do mesmo pedido, produzidos por serviços diferentes e transportados em topics diferentes, não têm ordem entre si. `PaymentAuthorized` e `StockReserved` podem chegar ao `Orders` em qualquer sequência, e é exatamente por isso que o fluxo de pedidos usado ao longo deste livro só solicita a captura depois que os dois chegaram. Não trate uma chave de partição compartilhada como uma promessa entre topics.

Ordenação por agregado

O mecanismo é simples de enunciar e exigente de manter: escolha uma chave, envie tudo o que tem a mesma chave por um único caminho e processe esse caminho uma mensagem por vez.

A chave de partição

Em um broker baseado em log, o produtor aplica um hash à `partition_key` para escolher uma partição. Toda mensagem com a mesma chave cai na mesma partição, e uma partição é uma sequência ordenada. O envelope deste livro carrega `partition_key` por esse motivo, e o outbox a armazena para que o relay possa defini-la sem perguntar ao serviço produtor (Capítulo 4).

A garantia tem uma condição: **mesma chave, mesmo particionador, mesma contagem de partições**. O particionador é a função de hash mais a forma como a chave é serializada em bytes antes do hash. Se dois produtores usam particionadores diferentes, um mesmo `order_id` vai para duas partições e a ordem por chave se perde, sem erro algum. Esta é a falha poliglota mais concreta em ordenação. As bibliotecas de cliente em linguagens diferentes não compartilham todas a mesma função de hash padrão, então um serviço escrito em uma linguagem e um relay escrito em outra podem enviar o mesmo `order_id` para partições diferentes. Duas defesas

funcionam. Fixe o particionador explicitamente em todo produtor, usando o mesmo algoritmo e a mesma codificação de chave. Ou tire a decisão dos produtores: faça o relay calcular a partição ele mesmo, a partir da `partition_key` armazenada, de modo que um único trecho de código seja responsável pelo mapeamento. A segunda é natural quando todo evento já passa pelo relay do outbox do Capítulo 5.

Os brokers baseados em filas são diferentes. Consumidores concorrentes em uma mesma fila não oferecem garantia alguma de ordenação quando há mais de um, porque o broker entrega a próxima mensagem a quem estiver livre. Alguns brokers baseados em filas oferecem um recurso de agrupamento que serializa as mensagens que compartilham um identificador de grupo, por exemplo os message groups nas filas FIFO do Amazon SQS, e outros oferecem um modo em que um único consumidor fica ativo em uma fila por vez. Esses recursos funcionam, e cada um custa paralelismo de um jeito diferente. Verifique o que o seu broker promete sobre reentrega e visibility timeouts sob esse recurso, porque a garantia costuma ter letras miúdas. Diga em qual família você se apoia e o que essa família realmente promete.

Escolhendo a chave

A chave é o identificador do agregado, e a escolha é uma declaração sobre onde a consistência vive. Para o ciclo de vida de pedidos da Parcelly, a chave é `order_id`. Todo evento que move um pedido pelos seus estados a carga, o que dá ao Shipping, ao Orders e ao Ledger uma sequência consistente por pedido.

Algumas perguntas ajudam quando a resposta não é óbvia:

- O estado de qual entidade o consumidor altera? Use essa entidade como chave.
- Qual é a menor unidade dentro da qual dois eventos podem entrar em conflito? Esse é o agregado.
- O mesmo stream é consumido por vários serviços com agregados diferentes? Então ou o produtor escolhe o mais fino em comum, ou cada consumidor precisa da sua própria visão.

O Inventory mostra a tensão. Os níveis de estoque pertencem a um SKU, então os comandos do próprio Inventory usam naturalmente `sku` como chave. Os eventos que fluem do Orders para o Inventory usam `order_id` como chave. Um evento de reserva carrega os dois identificadores, e o Inventory troca a chave internamente se precisar de ordem por SKU. Escolher uma única chave para todos os topics do sistema é tentador, e desperdiça o sentido de ter agregados.

Partições quentes

Uma chave de partição distribui a carga de forma uniforme apenas se as chaves forem igualmente ocupadas. Elas não são. Imagine uma promoção relâmpago na Parcelly em que o produto único de um vendedor recebe a maior parte dos pedidos de um dia em uma hora. Se o Inventory usa `sku` como chave do seu próprio topic, essa chave produz a maior parte do tráfego, tudo cai em uma partição, e um consumidor precisa fazer o trabalho do grupo inteiro enquanto os outros ficam ociosos.

Acrescentar partições não ajuda. Uma única chave sempre mapeia para uma única partição, e acrescentar partições move as outras chaves ao redor dela. Há quatro respostas honestas:

- **Tornar o agregado menor.** Se a regra de negócio permite, use como chave algo mais fino, como um id de reserva em vez do SKU. Abre-se mão da ordem dentro do SKU em troca de distribuição, e isso só é aceitável se nenhuma regra exigir ordem por SKU.
- **Gastar no consumidor, não nas partições.** Torne cada mensagem barata de tratar, para que um consumidor consiga acompanhar a chave quente.
- **Isolar.** Encaminhe as chaves quentes conhecidas para um topic ou uma partição dedicados, para que elas não sufoquem o tráfego comum que divide a partição.
- **Aceitar ordenação relaxada para esse stream.** Acrescente um sufixo aleatório à chave para espalhar a carga e conte com verificações de versão (mais adiante neste capítulo) para consertar a ordem no consumidor. É a mais flexível e a que dá mais trabalho. Note o que o salting dá e o que não dá: ele espalha a carga no broker, mas a verificação de versão serializa a chave de novo no bloqueio de linha do agregado no banco de dados do consumidor. O throughput só melhora se o traba-

lho caro acontece antes da etapa serializada, ou se as operações são comutativas.

Mediria a assimetria das chaves antes de escolher. Uma distribuição de chaves que parece uniforme em uma semana tranquila pode ser muito desigual em um dia movimentado, e o primeiro sinal costuma ser o atraso do consumidor em exatamente uma partição (Capítulo 13).

Paralelismo e ordem puxam em direções opostas

Um grupo de consumidores escala dando cada partição a um consumidor, então o número de partições é o teto do paralelismo. Se o Shipping tem seis partições, seis consumidores podem trabalhar em paralelo. Um sétimo ficaria ocioso.

As equipes costumam tentar aumentar o throughput dentro de um consumidor entregando as mensagens a um thread pool. Isso parece inofensivo e quebra a ordem. Duas mensagens do mesmo pedido, lidas uma depois da outra, são entregues a dois workers, e o worker mais lento termina em segundo. A partição entregou as mensagens em ordem e o processamento não a preservou.

O jeito de ter as duas coisas é paralelizar entre chaves e serializar dentro de uma chave. Dentro do consumidor, aplique um hash à chave para obter uma lane fixa e dê a cada lane exatamente um 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)

```

Três detalhes sustentam o design. O primeiro é o offset. Os commits são por partição, e `contiguousDoneOffset` é o maior offset abaixo do qual toda mensagem daquela partição está concluída. Quando as lanes terminam em velocidades diferentes, fazer o commit do offset da lane mais rápida faria, depois de um crash, pular mensagens que uma lane mais lenta não tinha terminado. O commit acontece no poll loop porque muitos clientes de log não são seguros para serem chamados de várias threads. O preço do paralelismo dentro do consumidor é essa contabilidade, e os duplicados depois de um crash, que um inbox do Capítulo 7 trata.

O segundo é o limite. Sem um, uma lane lenta mantém o ponto de commit parado enquanto o poll loop continua enchendo a memória, e a distância entre o offset com commit e o trabalho em andamento é exatamente o número de mensagens reentregues depois de um crash. Pausar as partições quando uma lane passa da sua marca d'água alta transforma a sobrecarga em atraso do consumidor, que é visível e seguro, em vez de em crescimen-

to de memória, que não é nem uma coisa nem outra. O mecanismo, e a escolha entre pausar e descartar, é o descrito em “Retries sob sobrecarga: backpressure” no Capítulo 6.

O terceiro é a falha. Se `handleMessage` falha e a lane faz retry no mesmo lugar, toda chave que divide a lane espera atrás dela e o offset com commit deixa de avançar. Enviar a mensagem para o lado deixa a lane seguir adiante e reordena essa chave. É a mesma decisão que o Capítulo 6 tomou para partições inteiras, agora tomada por lane, e a resposta depende do consumidor da mesma maneira.

Em um broker baseado em filas, a mesma ideia aparece um nível acima. Se você precisa de ordem por chave entre consumidores concorrentes, ou usa o recurso de agrupamento do broker, ou aceita que a fila vai entregar fora de ordem e trata isso no consumidor. Não suponha que uma fila com vários consumidores preserve a ordem só porque as mensagens entraram em ordem. Ela não preserva.

Detectando a chegada fora de ordem com versões

Às vezes o broker não consegue lhe dar a ordem de que você precisa, por causa de um retry topic, de um redrive, de consumidores paralelos ou de uma mudança na contagem de partições. Nesses casos, o consumidor tem de perceber sozinho. Para isso, os eventos precisam carregar algo que um consumidor possa comparar.

Timestamps são a coisa errada para comparar. O Capítulo 3 explicou que `occurred_at` reflete o relógio de uma máquina, e que usá-lo para decidir qual atualização vence deixa um serviço com relógio adiantado vencer todo conflito que deveria perder. Use `occurred_at` para pessoas e para análises.

O que funciona é um **número de versão** (também chamado de número de sequência): um contador que o agregado incrementa a cada mudança, gravado na mesma transação da mudança de estado e transportado com o evento. Como apenas o serviço proprietário altera o agregado, e o faz dentro de uma transação, as versões formam uma sequência sem lacunas por agregado: 1, 2, 3. Sem lacunas exige também uma regra no sentido inverso: **cada incremento de versão produz exatamente um evento publica-**

do. Uma atualização interna que incrementa a versão sem publicar nada deixa uma lacuna permanente, e todo consumidor esperará por um evento que não existe.

O envelope padrão não tem um campo para isso, e não vale a pena mudar o envelope por causa de um capítulo. Minha escolha é levar esse número no payload, como `order_version` nos eventos do Orders. É a versão do agregado (Capítulo 4), o mesmo número que a linha do outbox armazena. Deixo a discussão mais profunda sobre o design do envelope para o Capítulo 11. O contador pertence a um produtor e a um agregado. Um consumidor compara valores de `order_version` apenas entre eventos do Orders para o mesmo pedido, e nunca com um evento do Payments ou do Inventory, que precisariam de um contador próprio. Uma dependência entre serviços não se resolve com uma sequência compartilhada. Resolve-se com uma verificação de estado ou de pré-condição, como faz a etapa de captura ao esperar tanto por `PaymentAuthorized` quanto por `StockReserved`. A linha do outbox o recebe de graça, já que o serviço grava a nova versão, a mudança de estado e a linha do outbox em uma única transação.

A decisão do consumidor

Um consumidor que mantém uma visão de um agregado mantém também a última versão que aplicou. Para cada evento que chega, ele compara as versões e segue um de três caminhos.

Versão que chega, comparada à última aplicada	Significado	Ação
Igual ou menor	Duplicado ou obsoleto	Ignorar, contar, dar o ack
Exatamente uma a mais	O próximo evento	Aplicar, registrar a nova versão
Mais de uma a mais	Uma lacuna: algo está faltando ou atrasado	Reter o evento em buffer, não aplicar

```

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)

```

Duas propriedades tornam isso seguro. A verificação de versão e a atualização de estado fazem commit de forma atômica, então dois workers tratando a mesma chave não podem ambos aplicar a versão 4. E a verificação é idempotente: um duplicado é uma versão obsoleta como outra qualquer e é ignorado, de modo que ela complementa o inbox do Capítulo 7 em vez de substituí-lo. `holdEvent` também precisa ser idempotente. Uma entrega duplicada de uma versão já retida bateria, de outro modo, na restrição de unicidade do buffer, faria a transação falhar, deixaria a mensagem sem ack e viraria um loop de poison.

Duas pré-condições costumam passar despercebidas. A primeira é o início da sequência. Um consumidor que começa no meio do stream, como um serviço novo ou um replay a partir de um offset posterior, vê a versão 5 primeiro e esperaria para sempre pelas versões 1 a 4. Alimente a sua visão a partir de um snapshot ou da API do serviço proprietário, junto com a versão que esse snapshot reflete, e comece a comparar a partir dali.

A segunda é que a detecção de lacunas só funciona se o consumidor vê todas as versões. Um consumidor que assina um subconjunto dos eventos de um agregado, digamos apenas `OrderCancelled`, verá as versões 3 e 7 e concluirá, erradamente, que de 4 a 6 estão faltando. Para esses consumidores a regra se enfraquece para “nunca aplicar nada que não seja mais novo do que o que tenho”, o que ainda protege contra sobrescritas obsoletas, mas não consegue detectar lacunas.

Armadilhas do last-writer-wins

A política de conflito mais simples é que a última escrita vence. É simples e esconde três armadilhas diferentes, dependendo do que “última” significa.

A última a chegar. A visão é sobrescrita pela mensagem que o consumidor tratou mais recentemente. Um `OrderPlaced` reentregue que chega depois do `OrderConfirmed` volta o status para `PLACED`, que é a falha para a qual o Capítulo 7 apontou. A ordem de chegada é a única coisa que a topologia do broker pode não prometer.

A última por timestamp. Melhor na aparência, pior na prática. O skew de relógio entre serviços significa que o evento vencedor pode ser o do relógio mais adiantado. Dois eventos gravados com milissegundos de diferença em máquinas diferentes têm uma ordem de timestamp sem significado.

A última por versão. Isso funciona, e tem o seu próprio limite. Descartar um evento obsoleto só é correto se o evento mais novo já contém tudo o que o obsoleto dizia. Isso vale para eventos que carregam estado (“o pedido agora está `CANCELLED`”). Não vale para eventos que carregam mudanças, como “somar 4.990 ao saldo”. Ignorar um evento de mudança obsoleto perde a mudança.

Um esclarecimento evita que isso contradiga a tabela acima. Com versões sem lacunas aplicadas estritamente em sequência, uma versão menor é sempre um duplicado, porque já foi aplicada; ignorá-la é correto até para deltas. A armadilha do delta aparece quando um consumidor salta para a versão mais nova e pula uma lacuna, já que as mudanças puladas então se perdem.

Essa distinção decide o que fazer diante de uma versão obsoleta ou fora de ordem:

- **Eventos que carregam estado** (o estado completo, ou um estado que sobrescreve o anterior): ignorar o evento obsoleto é correto, porque o posterior o substitui.
- **Eventos de delta ou de transição** (incrementos, lançamentos de ledger, qualquer coisa que não substitui): nunca os descarte. Retenha-os em buffer e reordene-os, ou processe-os em qualquer ordem se a operação for comutativa. O Ledger, em particular, não deve descartar um lançamento por ser mais antigo.

Uma armadilha relacionada é o agregado excluído. Se um consumidor apaga a sua linha ao ver `OrderCancelled`, um `OrderPlaced` atrasado não encontra linha, passa na verificação de versão sem nada com que comparar e cria um novo pedido. A ressurreição é o bug característico de ignorar a ordem em torno de exclusões. Mantenha um tombstone: a chave, a última versão e uma marca de excluído, por pelo menos o tempo em que eventos obsoletos ainda possam chegar. O Capítulo 8 mostrou a mesma ressurreição quando um `OrderPlaced` reenviado chega depois de um cancelamento, e um tombstone com uma verificação de versão é o que a impede.

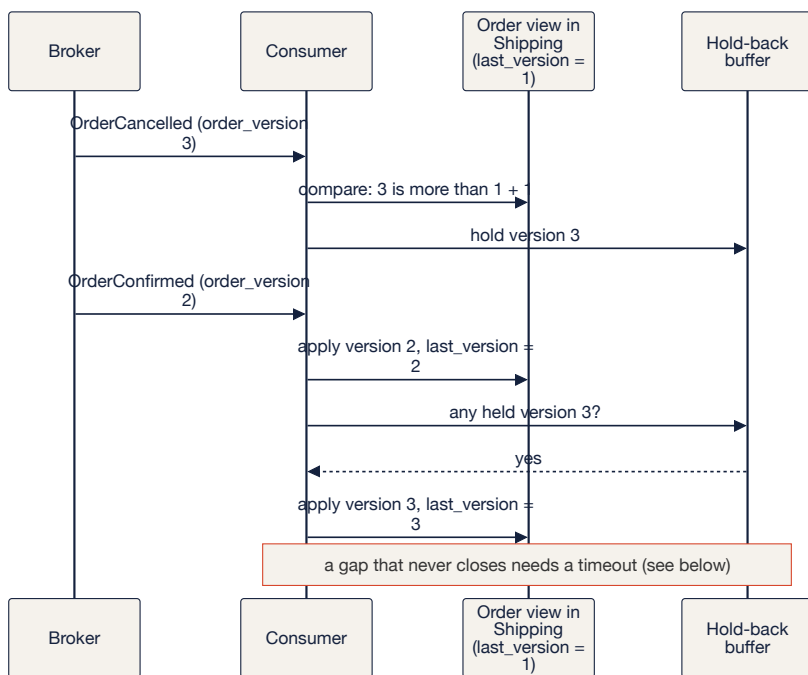
Por fim, ignorar é uma decisão que merece um rastro. Um evento obsoleto descartado em silêncio é um bug que ninguém consegue ver. Conte os eventos ignorados por consumidor e guarde a chave e as versões em uma linha de log, para que um pico na contagem apareça antes que uma pessoa cliente perceba (Capítulo 13).

Buffering e reordenação

Quando o consumidor vê uma lacuna, tem dois caminhos. Pode aplicar o evento mesmo assim e arriscar um estado errado, ou pode reter o evento até que o que falta chegue. A segunda opção é um **hold-back buffer**.

O diagrama de sequência abaixo mostra o Shipping recebendo do Orders dois eventos de `order-81724`, a versão 3 antes da versão 2, como no cenário de abertura. Ambos vêm de um único produtor e de um único agregado, o que torna significativa a comparação das suas versões. O Shipping

age apenas sobre `OrderConfirmed`, mas, para que isso funcione, ele lê o stream inteiro do `Orders` para o pedido e registra a versão de cada evento, inclusive dos tipos que de outro modo ignora. É isso que torna real o `last_version = 1` da visão e que torna possível a detecção de lacunas. Observe onde a versão 3 espera e o que a libera.



A etapa de liberação é um loop. Depois de aplicar qualquer evento, o consumidor verifica se o buffer guarda a próxima versão, a aplica e repete até que o buffer não tenha mais sucessora.

```

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)
  
```

O próprio buffer é uma tabela indexada pelo agregado e pela versão, de modo que sobrevive a uma reinicialização. Um buffer em memória desaparece com o processo, e o broker já considerou que essas mensagens receberam ack.

A limpeza precisa de mais uma regra. Sempre que `last_version` dá um salto, por exemplo depois de substituir a visão por um estado buscado, as linhas retidas com `version <= last_version` ficam obsoletas, e o consumidor as apaga na mesma transação. Do contrário, elas ficam na tabela para sempre, e o loop de drenagem nunca chega a elas.

A lacuna que nunca se fecha

O buffering é otimista: supõe que o evento que falta está apenas atrasado. Às vezes ele se perdeu, foi enviado para a DLQ ou nunca foi enviado a este consumidor. Sem um limite, o agregado fica bloqueado para sempre, e o conjunto retido cresce.

Todo buffer precisa de um timeout e de uma decisão sobre o que acontece quando ele expira. Há três respostas razoáveis, e escolheria pelo custo de errar:

- **Buscar o estado que falta.** Pedir ao serviço proprietário o estado atual por meio de uma API e substituir a visão. Funciona quando o produtor expõe uma API assim, e faz do buffer um caminho rápido com uma rede de segurança.
- **Alertar e esperar.** Uma pessoa olha. Certo para o Ledger, onde um lançamento errado custa mais que um lento.
- **Marcar a visão como suspeita e deixar a reconciliação repará-la.** Certo para modelos de leitura que toleram defasagem (Capítulo 13).

O bloqueio por chave do Capítulo 8 é a forma sem versões desse mesmo mecanismo. Lá, uma marca de dead letter bloqueia a chave, os eventos posteriores esperam em um armazenamento de retenção, e um desvio especial deixa o evento reenviado passar para liberá-los. Aqui, um evento enviado para a DLQ cria uma lacuna, os eventos posteriores esperam no buffer, e o evento reenviado os libera por si só, porque chega como `last_version + 1` e o loop de drenagem faz o resto. As versões substituem a marca e o desvio, e por isso este é o design mais forte onde o produtor consegue fornecê-las.

Os custos são reais: uma tabela de buffer para dimensionar e limpar, uma política de timeout para ajustar e uma nova classe de agregado travado sobre a qual alguém precisa receber um page. Se os seus eventos carregam o estado completo e os seus consumidores podem aplicar com segurança “o mais novo vence”, o buffer é desnecessário e o deixaria de fora.

Retries, dead letters e ordem

O Capítulo 6 e o Capítulo 8 terminaram, cada um, com uma pergunta sobre ordenação que deixaram para cá. A forma curta é que todo mecanismo de tratamento de falhas move uma mensagem no tempo, e isso é uma reordenação.

- **Retry no mesmo lugar** em uma partição de log bloqueia as sucessoras, então a ordem é preservada e a partição inteira paga em latência.
- **Um retry topic ou uma fila de atraso** deixa as sucessoras seguirem adiante, então a mensagem que falhou volta depois de eventos que ela precedia.
- **Um visibility timeout** em um broker baseado em filas devolve uma mensagem à fila depois do timeout, onde ela compete com mensagens enviadas depois dela.
- **Uma dead letter** pula a mensagem, e um **redrive** a traz de volta depois de tudo o que veio a seguir.

A verificação de versão é o que torna sobrevivíveis a segunda a quarta opções. Tome o exemplo que o Capítulo 6 deu: o Inventory recebe `OrderPlaced` e depois `OrderCancelled` de um pedido, vindos do Orders, e o `OrderPlaced` é movido para um retry topic depois de uma falha transitória. Com a versão do agregado vinda do Orders, `OrderPlaced` é a versão 1 e `OrderCancelled` é a versão 2. O cancelamento chega primeiro, e o Inventory ainda não tem visão do pedido, então `last_version` assume 0 como padrão e 2 é maior que 0 + 1. O cancelamento é retido, não aplicado contra uma reserva que não existe. Quando o `OrderPlaced` reenviado por retry chega como versão 1, ele é aplicado, e o loop de drenagem libera o cancelamento. A reserva órfã que o Capítulo 6 descreveu não aparece.

Isso só funciona para consumidores que mantêm versões e um buffer. Um consumidor que não as mantém precisa escolher, como o Capítulo 6 argumentou, entre bloquear a partição e tolerar a reordenação. E o conselho

do Capítulo 8 para ledgers continua valendo: para um consumidor em que um lançamento atrasado ou ausente significa um registro financeiro errado, pare e espere. Um consumidor pausado é ruidoso, e um Ledger que aplica lançamentos fora de ordem em silêncio não é.

Aviso: Um retry topic combinado com um consumidor sem verificação de versão reordena eventos em silêncio. Nada falha, nenhum alerta dispara e o estado fica errado. Se você encaminha as falhas para o lado, inclua a verificação de versão na mesma mudança.

Mudando o número de partições

A partição de uma chave é uma função da chave e do número de partições, em geral um hash módulo a contagem. Mude a contagem, e a função muda.

Suponha, como ilustração, que uma chave tenha hash 17. Com quatro partições ela mapeia para a partição 1 e, com seis partições, para a partição 5. Um hash 20 passa da partição 0 para a partição 2, enquanto 14 e 15 ficam onde estavam. A maioria das chaves se move. Sob hash por módulo, uma chave mantém a sua partição ao passar de quatro para seis apenas quando o seu hash módulo 12 é 0, 1, 2 ou 3, de modo que cerca de dois terços das chaves vão parar em outro lugar. A falha não é rara: atinge toda chave que tenha mensagens em trânsito durante a mudança.

O perigo é a sobreposição. As mensagens de uma chave que se move, gravadas antes da mudança, ficam na partição antiga, ainda não consumidas. As mensagens gravadas depois da mudança vão para a nova partição. Dois consumidores passam a processar a mesma chave ao mesmo tempo, sem ordem entre eles. Se um consumidor também mantém estado local por chave, como um cache ou uma janela de agregação, esse estado está na instância errada.

Formas de lidar com isso, em ordem aproximada de preferência:

1. **Provisionar partições suficientes desde o início.** Superdimensione em relação ao pico esperado de paralelismo, porque o número é barato de manter e caro de mudar. Muitos brokers baseados em log permi-

tem acrescentar partições, mas não removê-las, de modo que um erro em uma das direções é permanente.

2. **Esvaziar antes da mudança.** Pare os produtores ou pause o stream, deixe os consumidores chegarem ao fim das partições antigas e só então mude a contagem. A ordem é preservada ao custo de uma janela de indisponibilidade.
3. **Migrar para um novo topic.** Crie um topic com a contagem certa, execute uma virada em que os consumidores terminam o topic antigo antes de começarem o novo, e aposente o antigo. É a mais segura e a que dá mais trabalho.
4. **Contar com verificações de versão.** Se todo consumidor detecta e retém em buffer os eventos fora de ordem, a transição é sobrevivível sem uma pausa, e a própria transição é também um bom teste para saber se a verificação funciona.

Rebalanceamentos e revogação

Os rebalanceamentos de grupos de consumidores têm uma forma parecida em escala menor, e acontecem a cada deploy. Quando uma partição passa para outro consumidor, o trabalho em andamento no antigo pode se sobrepor às primeiras mensagens no novo. O que o consumidor faz na revogação decide o tamanho dessa sobreposição.

Na revogação, o consumidor para de despachar novas mensagens das partições revogadas, deixa as lanes terminarem o que têm ou o abandona, e faz o commit do offset contíguo antes de entregar a partição. Se um rebalanceamento revoga todas as partições ou só as que se movem depende da estratégia de atribuição do seu cliente, e uma estratégia de tudo de uma vez paralisa o grupo inteiro durante o processo; verifique qual você usa. Um consumidor lento para perceber que perdeu uma partição continua processando como um zumbi enquanto o novo proprietário faz o mesmo trabalho, então espere alguma sobreposição, façam o que fizerem.

O inbox e a verificação de versão mantêm essa sobreposição inofensiva apenas para efeitos dentro da transação do próprio banco de dados do consumidor. Eles não fazem nada por um e-mail ou por uma chamada à API da transportadora feita pelos dois consumidores. Esses efeitos colaterais externos precisam da sua própria chave de idempotência, como descreve o Capítulo 7, porque o inbox protege o estado local e mais nada.

O que cada consumidor da Parcelly precisa

Nem todo consumidor precisa da mesma ordenação, e tratá-los todos igual é fonte tanto de bugs quanto de desperdício. Uma tabela, desenhada para a Parcelly, mostra como decidiria.

Consumidor	Necessidade de ordenação	Mecanismo	Se uma mensagem estiver faltando
Notificações	Nenhuma; cada mensagem é um fato independente	Qualquer ordem, sem necessidade de chave	Enviar para a DLQ e continuar
Visão de status do Orders	Por pedido, com estado	Usar <code>order_id</code> como chave, ignorar versões obsoletas	Buscar o estado atual
Reservas do Inventory	Por pedido, transições	Usar <code>order_id</code> como chave, versões mais hold-back buffer	Reter a chave, alertar
Shipping	Por pedido, <code>OrderConfirmed</code> antes de <code>OrderCancelled</code>	Usar <code>order_id</code> como chave, versões mais hold-back buffer	Reter a chave, alertar
Ledger	Por pedido, sem lacunas, sem descartes	Usar <code>order_id</code> como chave, parar diante de uma lacuna na sequência por pedido pertencente ao Payments	Parar e esperar por uma pessoa

A linha do Ledger nomeia o seu contador de propósito. O Ledger consome eventos do Payments, então compara uma sequência que o Payments incrementa por pedido, não `order_version`, que pertence ao Orders. Um contador só é comparável entre eventos do único serviço que o possui.

A primeira linha é a mais barata e merece mais crédito do que recebe. A melhor maneira de lidar com a ordenação é projetar o consumidor de modo que ela não importe. Uma operação comutativa, como acrescentar um item a um conjunto ou tomar o máximo de dois valores, dá o mesmo resultado em qualquer ordem. Um consumidor construído com essas operações não precisa de chave, de buffer nem de versão. Quando você consegue remodelar um handler desse jeito, o faria antes de recorrer a maquinário.

Trade-offs e custos

Você está trocando paralelismo por ordem, e a taxa de câmbio é o agregado. Uma chave mais grossa dá ordenação mais forte e menos distribuição. Uma chave mais fina dá distribuição e ordenação mais fraca. Não existe chave que dê as duas, e a decisão pertence a quem conhece as regras de negócio.

As chaves quentes são um custo de design, não um acidente de operação. Uma chave de partição escolhida a partir de um identificador de negócio herda a assimetria do negócio. Uma promoção relâmpago a revela. Você pode mitigá-la, mas não removê-la sem abrir mão da ordem para essa chave.

As versões exigem disciplina dos produtores. O agregado precisa incrementar a versão na mesma transação da mudança de estado, a cada mudança, em todo caminho de código. Um único caminho que esquece deixa uma lacuna pela qual os consumidores esperarão para sempre. O inverso também vale: todo incremento precisa publicar exatamente um evento, ou a sequência tem uma lacuna que nenhum consumidor consegue fechar. Backfills, ferramentas administrativas e migrações de dados são onde isso falha, e um consumidor que começa no meio do stream precisa de uma visão alimentada antes de poder comparar qualquer coisa.

Os buffers acrescentam estado e uma nova falha. Uma tabela de hold-back precisa de dimensionamento, limpeza, um timeout, um responsável e um alerta. É também mais um lugar onde um agregado travado pode se esconder. Para consumidores que podem aplicar “o mais novo vence”, dispense-o.

As verificações de versão não ordenam entre agregados nem entre topics. Elas protegem a sequência de um agregado. Se uma regra precisa saber que um evento no topic A veio antes de um evento no topic B, um número de versão não o fornece. Isso é um problema de fluxo de trabalho, e o Capítulo 10 o trata como tal.

Quando não se incomodar. Se o consumidor é comutativo, se os eventos carregam o estado completo e “o mais novo vence” é correto, ou se um evento atrasado é inofensivo e uma reconciliação periódica vai repará-lo, dispense o maquinário. Fazer menos é uma decisão com custo, e deve ser uma decisão registrada.

Erros comuns

- **Pedir ordenação global.** Parece o padrão seguro e limita o throughput a um único leitor. Declare o requisito real, a ordem por agregado, e projete para ele.
- **Usar como chave algo que não é o agregado.** Usar como chave o tipo de evento, o serviço ou um valor aleatório acontece porque o padrão faz isso por você. A correção é usar como chave o identificador da entidade cujo estado o consumidor altera.
- **Usar timestamps para ordenar eventos.** Parecem números de versão naturais. O skew de relógio torna a ordem sem sentido; carregue, em vez disso, uma versão do agregado.
- **Paralelizar dentro de um consumidor sem lanes por chave.** Um thread pool aumenta o throughput e quebra a ordem da partição. Aplique um hash à chave para obter uma lane e dê a cada lane um worker.
- **Descartar eventos delta obsoletos.** Uma verificação de versão serve para eventos que carregam estado. Aplicada a incrementos e a lançamentos de ledger, descarta mudanças reais. Classifique cada evento como de estado ou de delta antes de escolher uma política.
- **Esquecer os tombstones.** Apagar uma linha no cancelamento deixa um evento atrasado recriar o pedido. Mantenha a chave e a última versão depois da exclusão por todo o tempo em que eventos obsoletos puderem chegar.
- **Reter eventos em memória.** Um buffer que morre com o processo perde eventos que o broker já considera entregues. Persista-o, na mesma transação da visão.
- **Mudar a contagem de partições em produção sem um plano.** O mapeamento muda para cerca de dois terços das chaves ao passar de quatro para seis partições, e a ordem muda em silêncio junto com ele. Esvazie, migre ou proteja-se com versões.
- **Misturar particionadores entre produtores.** Serviços em linguagens diferentes podem aplicar hashes diferentes à mesma chave. Fixe um único particionador em todo lugar, ou calcule a partição no relay.
- **Lanes sem limite.** Uma lane lenta então enche a memória enquanto o ponto de commit fica parado. Limite cada lane, pause as partições e faça o commit por partição a partir do poll loop.

- **Confiar no inbox para efeitos externos em um rebalanceamento.** O inbox cobre apenas o banco de dados do próprio consumidor. Um e-mail ou uma chamada à transportadora precisa da sua própria chave de idempotência.
- **Supor que chaves compartilhadas ordenam eventos entre topics.** O mesmo `order_id` em dois topics não dá sequência alguma entre eles. Faça o consumidor esperar pelos dois, ou modele a dependência em um fluxo de trabalho.

Pontos-chave

- A ordenação global custa todo o paralelismo e raramente é necessária; a ordenação por agregado é a ordem que vale a pena projetar.
- Uma chave de partição que identifica o agregado dá eventos ordenados por entidade em um broker baseado em log, mas só com o mesmo particionador e a mesma contagem de partições em todo lugar; os brokers baseados em filas precisam de um recurso de agrupamento ou de um consumidor que tolere a desordem.
- O paralelismo dentro de um consumidor precisa ser por chave, com um worker por lane, limitado e com commit por partição, ou desfaz em silêncio a ordem que a partição entregou.
- Os números de versão incrementados pelo agregado dentro da sua transação permitem a um consumidor detectar eventos obsoletos, duplicados e ausentes, o que os timestamps não conseguem.
- Ignorar um evento obsoleto é correto para eventos que carregam estado e errado para deltas e lançamentos de ledger, que devem ser retidos em buffer ou aplicados em qualquer ordem.
- Retry topics, redrive e mudanças de partições reordenam todos as mensagens, então todo consumidor que depende de ordem precisa de uma verificação de versão antes de usá-los.
- Todo hold-back buffer precisa de um timeout, de um responsável e de um alerta, porque uma lacuna que nunca se fecha é um agregado travado; o bloqueio por chave do Capítulo 8 é a forma sem versões da mesma ideia.

A ordenação dá a cada evento um lugar dentro de um agregado, mas um processo de negócio abrange vários agregados e serviços, e alguns de seus passos precisam ser desfeitos quando um posterior falha. O Capítulo 10 se volta para as sagas, onde a ordem dos passos é o design e a compensação é a forma como um fluxo de trabalho se recupera.

PARTE IV

Fluxos de trabalho e contratos

Capítulo 10 — Sagas e fluxos de trabalho de longa duração

Imagine uma tarde de terça-feira na Parcelly. Um comprador faz o pedido de uma luminária de cerâmica. O Orders emite `OrderPlaced`, o Payments autoriza o cartão e o Inventory reserva a última luminária do estoque do vendedor. Então o cartão é recusado e o Payments emite `PaymentFailed`. No mesmo instante, um segundo comprador olha para uma página de produto que diz “sem estoque”, porque a reserva do primeiro pedido ainda segura a luminária.

Ninguém fez nada de errado. Cada serviço cumpriu o seu papel, cada transação local fez commit e todas as mensagens foram entregues. O problema é que a operação de negócio, “vender esta luminária a este comprador”, se espalha por quatro serviços e vários minutos, e nenhuma transação a cobre. Metade aconteceu. A outra metade não vai acontecer, e alguém precisa devolver a luminária.

O momento importa. Nada foi capturado, então desfazer custa pouco: libera-se a reserva e deixa-se o pedido terminar. Se a mesma luminária já tivesse sido paga e a falha viesse depois, devolvê-la seria um ato diferente e mais caro, e o capítulo volta a essa linha.

Agora suponha que a mesma falha chegue enquanto o vendedor cancela o anúncio, enquanto o comprador edita o endereço de entrega ou enquanto a resposta da captura se perde no caminho de volta. Este capítulo trata da engrenagem que permite a um fluxo como esse terminar de forma limpa ou se desfazer de forma limpa. Como fazer um fluxo de trabalho com vários serviços acabar em um estado que o negócio aceita, quando não é possível manter um bloqueio durante todo ele e qualquer passo pode falhar?

O problema e por que ele é difícil

Dentro de um único banco de dados, uma transação oferece atomicidade: ou todas as mudanças acontecem, ou nenhuma. Entre serviços, isso não existe. Cada serviço é proprietário do seu banco de dados, e o message broker fica entre eles. O Capítulo 4 mostrou como tornar atômicos a mudança de estado de um único serviço e o evento que ele publica. Isso não tornou a reação do serviço seguinte atômica com a mudança do primeiro, e não tem como tornar.

A resposta antiga eram as transações distribuídas, em geral o commit em duas fases. Elas mantêm bloqueios em todos os participantes até que um coordenador decida e, se o coordenador falha depois de os participantes votarem, eles esperam segurando esses bloqueios até que ele volte. Exigem participantes que ofereçam suporte ao protocolo e fazem do participante mais lento ou menos disponível o limite de todos. Um provedor de pagamentos externo ou a API de uma transportadora não participa da sua transação de forma alguma. Para um fluxo que inclui uma chamada a terceiros, essa opção está fechada antes mesmo de você começar a avaliá-la.

Uma **saga** é a alternativa. Ela divide uma operação de negócio em uma sequência de transações locais, uma por serviço. Quando um passo falha, a saga não faz rollback, porque as transações anteriores já fizeram commit e outras pessoas podem ter lido seus resultados. Ela executa **compensações**: novas transações locais que desfazem semanticamente os passos anteriores. Uma compensação é uma ação para a frente que leva o negócio de volta a um estado aceitável, não uma máquina do tempo.

Três coisas tornam isso difícil na prática.

Primeiro, os estados intermediários são visíveis. Entre “estoque reservado” e “pagamento capturado”, o pedido não está nem confirmado nem cancelado, e tudo o que o lê vê esse estado intermediário. A atomicidade se foi; o isolamento também.

Segundo, a compensação é uma decisão de design, não um mecanismo. Alguns passos têm um inverso limpo, como liberar uma reserva. Outros têm apenas um inverso aproximado, como reembolsar um pagamento capturado, o que custa taxas e leva dias. Alguns não têm nenhum: um e-mail que saiu continua enviado.

Terceiro, a própria saga pode falhar. O processo que conduz o fluxo de trabalho pode cair entre passos, uma mensagem pode chegar duas vezes, uma resposta pode nunca chegar, e uma compensação pode falhar assim como uma ação. Tudo o que os capítulos 4 a 9 construíram para mensagens individuais agora precisa valer para um fluxo inteiro.

Nota: Um teste útil antes de adotar uma saga: liste cada passo anterior ao pivot (o passo depois do qual a saga só avança, definido mais adiante neste capítulo) com a sua compensação e diga quanto ela custa. Os passos posteriores ao pivot não devem ter compensação. Se vários passos anteriores a ele dizem “nenhuma” ou “um ticket de suporte”, a fronteira entre esses serviços pode estar no lugar errado, e mover os passos para a transação local de um único serviço pode sair mais barato do que coordená-los.

O pedido da Parcelly como uma saga

Pegue o fluxo do happy path e dê nome aos seus passos como uma saga. Cada passo é uma transação local em um serviço, seguida de um evento que é gravado no transactional outbox desse serviço na mesma transação.

Pas- so	Servi- ço	Transação local	Compensação
1	Orders	Criar o pedido em estado pendente, emitir <code>OrderPlaced</code>	Marcar como cancelado, emitir <code>OrderCancelled</code>
2	Pay- ments	Autorizar o cartão, emitir <code>PaymentAuthorized</code>	Anular a autorização, emitir <code>PaymentVoided</code>
3	Inven- tory	Reservar o estoque, emitir <code>StockReserved</code>	Liberar a reserva, emitir <code>StockReleased</code>
4	Pay- ments	Capturar o pagamento, emitir <code>PaymentCaptured</code>	Nenhuma dentro da saga: este é o pivot (um reembolso pertence a um fluxo de cancelamento separado)
5	Orders	Confirmar o pedido, emitir <code>OrderConfirmed</code>	Nenhuma: há retry até ter sucesso

Os dois passos do Payments não são transações de banco de dados no sentido estrito. Cada um é uma chamada a um provedor externo com uma chave de idempotência estável, seguida de um registro local do resultado,

que é o padrão do Capítulo 7. As garantias da saga se apoiam nesse pareamento, e um timeout na chamada ao provedor carrega a incerteza que o Capítulo 3 descreveu.

Os passos 2 e 3 rodam em paralelo, e a captura só é solicitada depois que `PaymentAuthorized` e `StockReserved` chegaram. O `Shipping` reage a `OrderConfirmed` e o `Notifications` reage ao que o `Shipping` emite. Por ora vamos tratar esses dois últimos como fora da saga e voltaremos ao motivo.

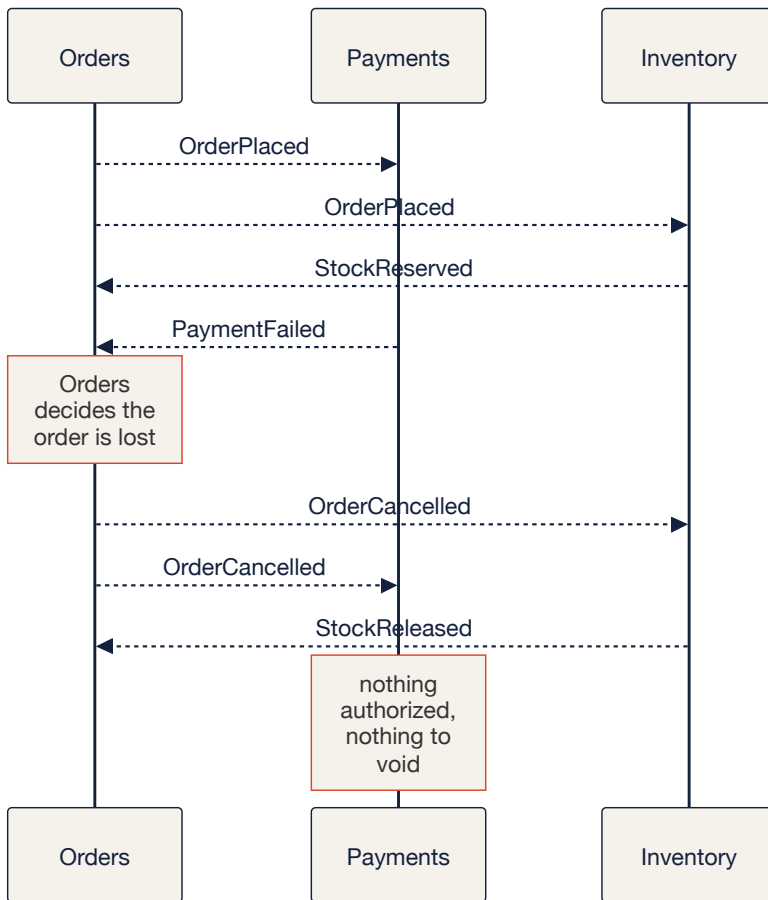
Os ramos de falha seguem a tabela do fluxo de eventos. Se `PaymentFailed` chega, não existe autorização, então a única compensação é o `Inventory` liberar uma reserva que pode ou não ter sido feita, emitindo `StockReleased`. Se `StockReservationFailed` chega, há uma autorização retida e nada foi capturado, então o `Payments` a anula e emite `PaymentVoided`. Um reembolso só aparece depois da captura. Antes da captura, o desfazer é uma liberação ou uma anulação, e chamá-lo de reembolso estaria errado no livro-razão e errado no extrato bancário do comprador.

Nos dois ramos de falha, o `Orders` reage e emite `OrderCancelled`, e os outros serviços compensam em resposta a ele. Essa frase é o design de coreografia inteiro. Se esse é o design inteiro que você quer é a próxima pergunta.

Coreografia

Em uma saga **coreografada** não há um proprietário central. Cada serviço escuta eventos, faz a sua transação local e publica o evento seguinte. O fluxo de trabalho existe apenas como a soma dessas assinaturas.

O diagrama abaixo mostra o ramo de falha do pagamento como uma coreografia. Observe quem decide o quê: ninguém dá instruções, e cada reação é disparada por um evento que o serviço já assina.



Os pontos fortes são reais. Os serviços continuam fracamente acoplados: o Inventory sabe o que `OrderCancelled` significa e nada sobre o Payments. Não existe um único componente cuja falha pare todos os fluxos de trabalho. Adicionar um consumidor, como um serviço de analytics que reage a `OrderConfirmed`, não exige mudança nos demais.

Os pontos fracos aparecem à medida que o fluxo cresce. A definição do fluxo de trabalho fica espalhada por assinaturas em vários códigos-fonte, e “o que acontece depois de `StockReserved`?” não se responde lendo um único arquivo. Dependências cíclicas se infiltram: o Orders escuta o Payments, que escuta o Orders. Cada novo ramo adiciona eventos, e todo serviço precisa ser atualizado em uma ordem cuidadosa. O mais importante: ninguém é responsável pela pergunta “onde está o pedido 81724 agora?”.

A resposta é uma junção entre os logs de vários serviços, que é o problema que o Capítulo 1 nomeou e ao qual o Capítulo 13 volta com tracing e reconciliação.

Considero que a coreografia funciona bem quando o fluxo é curto, tem poucos ramos e nenhum passo depende de uma conjunção de passos anteriores. Dois ou três serviços reagindo a um único evento é um encaixe natural. O fluxo de pedidos da Parcelly já passou dessa linha, porque a captura precisa que dois resultados anteriores tenham chegado.

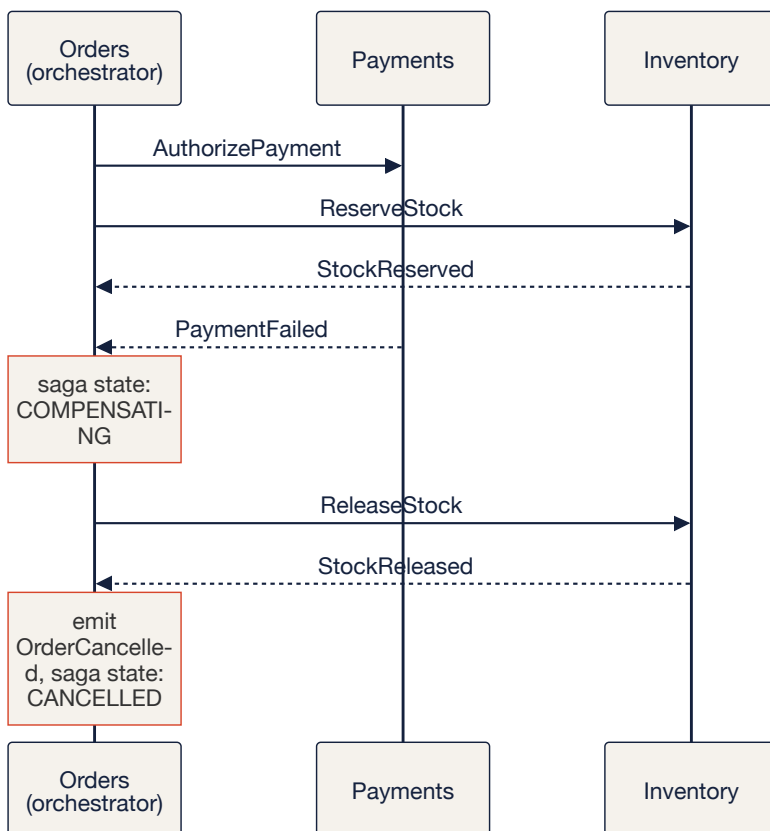
Orquestração

Em uma saga **orquestrada**, um componente é proprietário do fluxo de trabalho. Ele guarda o estado, decide o próximo passo, envia comandos aos participantes e reage às respostas deles. Os participantes sabem fazer o seu trabalho local e relatar o resultado; não sabem onde estão dentro do fluxo maior.

Esta seção introduz dois comandos além dos canônicos:

`AuthorizePayment` (do Orders para o Payments) e `ReserveStock` (do Orders para o Inventory). A confirmação não precisa de comando, porque é a própria transação local do orquestrador. Como todos os comandos neste livro, eles carregam um `command_id` no lugar de `event_id`, e as respostas são os eventos comuns. Na versão coreografada, o Payments e o Inventory começavam com `OrderPlaced`; aqui o orquestrador diz a eles o que fazer.

O diagrama mostra a mesma falha de antes, mas agora toda seta que sai do Orders é um comando. Observe como o estado da saga vive em um só lugar e como os comandos de compensação nomeiam exatamente o trabalho a ser desfeito.



O orquestrador é o lugar onde se responde à pergunta “o que acontece em seguida?”. Ele é uma máquina de estados, e a máquina de estados pode ser lida, testada e desenhada. Um novo ramo é uma mudança em um único lugar. O orquestrador sabe quais passos foram concluídos, então uma compensação mira o que de fato aconteceu, e não tudo o que poderia ter acontecido.

Os custos são o espelho disso. O orquestrador conhece todos os participantes e se torna um ponto de acoplamento. Se for descuidado, a lógica de domínio vaza dos serviços para dentro dele, e os participantes viram chamadas de procedimento remoto com passos extras. Ele também é um componente que precisa estar correto, sempre ativo e recuperável, já que um orquestrador que caiu e perdeu o estado é uma saga que ninguém jamais vai terminar.

Na minha opinião, um orquestrador que vive dentro do serviço proprietário do processo de negócio é a forma mais defensável. Na Parcelly, esse serviço é o Orders, que já é proprietário do ciclo de vida do pedido. Um serviço de orquestração independente, proprietário dos fluxos de trabalho de todo mundo, tende a virar aquilo que toda equipe precisa alterar para entregar qualquer coisa.

Escolhendo entre as duas

Nenhum dos estilos é melhor em geral. A escolha depende do formato do fluxo e de quem precisa responder a perguntas sobre ele.

Pergunta	Coreografia	Orquestração
Onde o fluxo é definido?	Espalhado por assinaturas	Em uma única máquina de estados
Acoplamento entre serviços	Apenas por meio dos contratos de eventos	O orquestrador conhece cada participante
Adicionar um passo ou ramo	Mexer em vários serviços	Mexer no orquestrador e no participante
Passos paralelos e junções	Complicado; cada serviço rastreia o que já viu	Natural; o orquestrador guarda a junção
“Onde está o pedido 81724?”	Junção entre logs	Ler uma linha
Ponto único de falha	Nenhum, mas também nenhum responsável	O orquestrador, que precisa ser recuperável
Risco quando cresce	Dependências cíclicas e ocultas	Um god service cheio de lógica de domínio
Bom encaixe	Fluxos curtos, poucos ramos	Fluxos longos, condições, timeouts, junções

Misturar os dois é normal e sensato. Uma saga de pedido orquestrada ainda pode emitir `OrderConfirmed` como um evento simples ao qual o Shipping e o Notifications reagem de forma coreografada. A saga deve cobrir a parte do fluxo que precisa de desfazer coordenado; notificações e analytics não precisam, e puxá-los para dentro só adiciona estados.

Compensações e passos que não podem ser desfeitos

Projete a compensação de um passo quando projetar o passo, não quando o primeiro incidente a exigir. Para cada passo, escreva o que o negócio quer caso a saga falhe depois dele.

Algumas regras mantêm as compensações honestas.

Uma compensação é uma nova ação de negócio, não um desfazer no banco de dados. `StockReleased` não apaga o histórico da linha da reserva. Ele registra que a reserva terminou. `PaymentRefunded` não apaga a captura; o Ledger guarda as duas entradas. É por isso que o Ledger é append-only e as compensações aparecem nele como eventos reais.

Ordene os passos pela dificuldade de desfazê-los. Coloque primeiro os passos fáceis de reverter e por último os difíceis ou impossíveis de reverter. Autorização e reserva são baratas de desfazer, e por isso vêm antes da captura. A captura é onde um reembolso passa a custar dinheiro e tempo. Isso leva à ideia de um **passo pivot**: o ponto depois do qual a saga não compensa mais e só avança. Na Parcelly, a captura é o pivot. Antes dela, falhar significa liberar e anular. Depois dela, uma falha na confirmação não é recuo, mas retry, porque o negócio tem o dinheiro do comprador e deve um pedido.

Algumas ações não têm inverso. Um e-mail que foi enviado, uma transportadora que recebeu o envio, um webhook sobre o qual o sistema de um vendedor agiu: você pode enviar um acompanhamento, mas não pode desenviar. Esses passos pertencem ao que vem depois do pivot, e suas falhas devem ser tratadas com retry e correção, não com compensação. É também por isso que o Notifications é puramente reativo e nunca faz parte da saga: seus passos não podem ser compensados e não deveriam poder bloquear nada.

Compensações precisam sofrer retry até terem sucesso. A falha de uma ação leva a uma compensação. A falha de uma compensação não leva a nada, exceto a uma saga travada e a um sistema inconsistente. Trate a compensação como algo que precisa ser concluído no fim, com a disciplina de retry do Capítulo 6 e o tratamento de poison messages do Capítulo 8.

Uma compensação que falha de forma permanente deve chegar a uma pessoa por meio de um parking lot, com o estado da saga registrado, e não ser descartada.

Compensações precisam ser idempotentes e tolerar a chegada antecipada. A primeira exigência é território do Capítulo 7: um `ReleaseStock` duplicado não pode liberar duas vezes. A segunda é mais sutil. Suponha que o `Orders` envie `ReserveStock`, sofra um timeout e comece a compensar enviando `ReleaseStock`. A reserva estava lenta, não perdida, e a liberação pode chegar ao `Inventory` primeiro. Se o `Inventory` trata uma liberação de uma reserva desconhecida como um no-op, o `ReserveStock` tardio então tem sucesso e a luminária fica retida para sempre. O formato seguro é o `Inventory` registrar a liberação contra o pedido, de modo que a reserva posterior a veja e se recuse. Alguns textos chamam isso de tombstone. Qualquer que seja o nome, a compensação precisa deixar uma marca que a ação consiga ler.

Estado da saga e timeouts

Uma saga orquestrada precisa de estado durável, e ele pertence ao mesmo banco de dados do serviço que a executa, para que as mudanças de estado entrem na mesma transação local das gravações no outbox e no inbox.

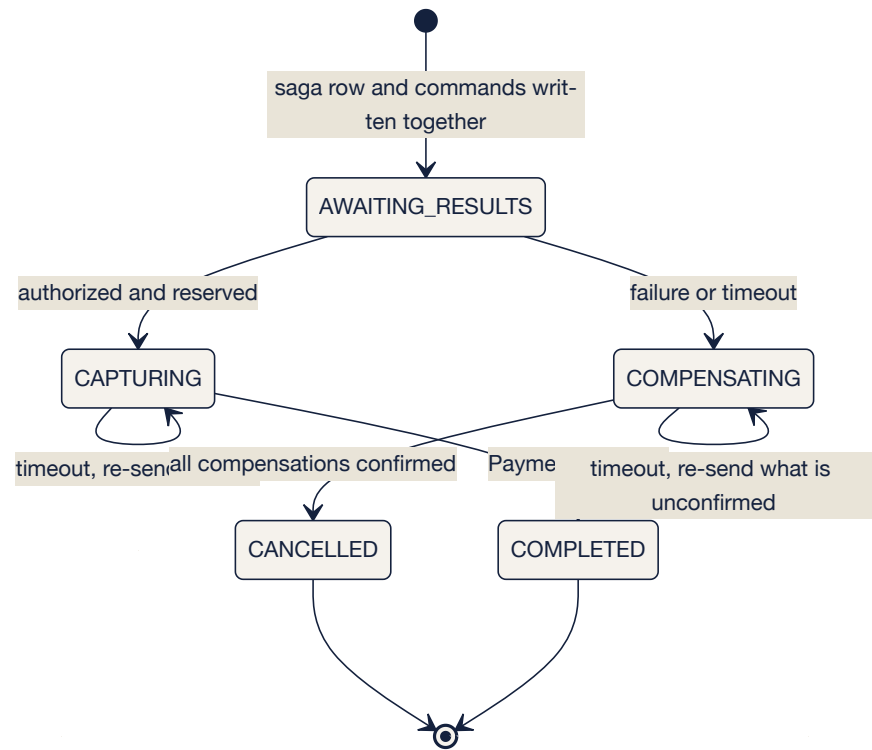
```
table saga_instance:
  saga_id      text primary key
  order_id     text unique not null
  state        text not null      // see below
  authorized   boolean not null
  reserved     boolean not null
  stock_released boolean not null // compensation confirmed
  auth_voided  boolean not null  // compensation confirmed
  rounds       int not null       // deadlines spent in this state
  deadline_at  timestamp not null
  version      int not null
  updated_at   timestamp not null
```

A coluna `state` é um pequeno conjunto de posições nomeadas: `AWAITING_RESULTS`, `CAPTURING`, `COMPLETED`, `COMPENSATING`, `CANCELLED`. A linha da saga e os primeiros comandos são gravados em uma única transação, então não existe uma posição “iniciada” separada que alguém pudesse observar. Os dois booleanos `authorized` e `reserved` registram quais resultados já chegaram, e é assim que a junção funciona: a captura é

solicitada quando ambos são verdadeiros e o estado é `AWAITING_RESULTS`. As duas flags `stock_released` e `auth_voided` registram quais compensações foram confirmadas, de modo que “todas as compensações confirmadas” é algo que a linha consegue responder. Uma compensação desnecessária já começa confirmada: depois de `PaymentFailed` não há autorização a anular, então `auth_voided` é verdadeiro na entrada em `COMPENSATING`.

Uma restrição `unique` em `order_id` impede que duas sagas rodem para o mesmo pedido. A coluna `version` protege contra dois handlers atualizando a mesma linha ao mesmo tempo, e os handlers abaixo dependem dela.

Uma máquina de estados desenhada uma vez vale mais do que qualquer descrição em prosa. O próximo diagrama mostra as transições legais; observe as saídas que não levam a `COMPLETED` e os dois estados que fazem loop no próprio timeout.



Note que `CAPTURING` não tem saída para `COMPENSATING`. Isso é o pivot expresso na máquina de estados. Uma captura cuja resposta nunca veio é re-enviada e, se o provedor informa uma recusa permanente, ela é tratada

como um caminho de negócio separado e explícito, com uma pessoa envolvida.

Os loops importam tanto quanto as saídas. Uma saga em `CAPTURING` ou `COMPENSATING` está esperando uma resposta, e uma resposta pode se perder como qualquer outra mensagem. Se só `AWAITING_RESULTS` tivesse um deadline, uma saga presa em qualquer um dos outros dois ficaria parada sem nada para despertá-la. Por isso todo estado não terminal tem um deadline, e o deadline é reiniciado a cada transição.

Timeouts

Toda espera em uma saga precisa de um deadline, porque uma resposta que nunca chega também é uma falha, e a mais provável de ser esquecida. O Capítulo 3 descreveu como um timeout esconde três possibilidades: a ação falhou, a ação teve sucesso e a resposta se perdeu, ou a ação ainda não terminou. A saga não consegue distinguir qual, então precisa de uma política para cada espera, escrita.

Um timeout também é uma mensagem. Este capítulo introduz um evento interno para ele, `SagaTimedOut`, emitido pelo temporizador do próprio orquestrador. A implementação usual é um processo temporizador que seleciona as sagas cujo `deadline_at` passou e entrega a cada uma um evento `SagaTimedOut` pelo mesmo handler de todas as outras entradas. Assim o timeout segue o mesmo caminho de todo o resto, com o mesmo inbox, as mesmas verificações de estado e o mesmo 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)

```

O ramo `AWAITING_RESULTS` envia comandos de compensação sem perguntar qual passo foi concluído. Isso é deliberado. No momento do timeout, o orquestrador não sabe se um `StockReserved` tardio está a caminho, então pede os desfazimentos que ainda estão pendentes e conta com os participantes para tratar “nada a desfazer” como sucesso. O Capítulo 7 é o motivo de isso ser seguro.

`armNextDeadline` faz duas coisas dentro da mesma transação. Ele define `deadline_at` como `now() + timeoutFor(saga.state)`, de modo que nenhum estado não terminal fique sem relógio. E quando `saga.rounds` passa de `MAX_ROUNDS`, digamos cinco, a saga deixa de ser um problema de

tempo e passa a ser um problema humano: ela dispara um alerta e registra o estado da saga no parking lot do Capítulo 8, e as rodadas continuam. Sagas terminais não recebem novo deadline. Como o deadline, a mudança de estado e os comandos fazem commit juntos, uma queda não pode deixar uma saga que seguiu em frente mas não está mais sendo vigiada.

```
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))
```

Uma saga em **CAPTURING** nunca é revertida. O pivot já passou, então os únicos movimentos são reenviar a captura com a mesma chave e acionar uma pessoa.

Escolher a duração de um timeout não é algo que possa fazer por você. Depende de quanto tempo leva o caminho saudável mais lento, de quanto um comprador espera e de quanto custa a um vendedor manter o estoque retido. Um deadline definido com um valor muito curto transforma lento em falho e inunda o sistema de compensações. Um muito longo mantém reservas enquanto os compradores olham uma página sem estoque.

Chegadas tardias também precisam de uma regra. Se **PaymentAuthorized** chega depois que a saga está **CANCELLED**, o orquestrador precisa emitir **VoidAuthorization** de novo, porque alguém está retendo fundos em um cartão por um pedido que não existe. O evento tardio prova que existe uma autorização. A anulação enviada no timeout ainda pode estar em trânsito, pode ter sido processada logo depois que a autorização foi concluída, ou a chamada ao provedor pode ter rodado antes de o Payments verificar a marca que a anulação deixa. Enviar a anulação de novo resolve os três casos.

Esse reenvio só funciona se não for confundido com uma repetição. Aqui duas regras sobre identidade se encontram. Um comando cuja repetição não deve causar trabalho novo, como **CapturePayment**, mantém um único **command_id** para a saga e o passo, de modo que um retry é reconhecido e

respondido com o primeiro resultado. Uma compensação que pode legitimamente ser emitida de novo toma o seu `command_id` da mensagem que a disparou: saga, passo e o `event_id` causador. A anulação reemitida é então um comando novo. Ela também precisa ser idempotente sobre o estado-alvo, e não apenas sobre o comando; por isso o Payments lê `VoidAuthorization` como “nenhuma autorização ativa pode permanecer para o pedido 81724” e verifica isso, em vez de repetir a resposta em cache “nada a anular” da primeira tentativa. Uma máquina de estados que ignora eventos que chegam no estado errado perde esses casos. Trate-os de forma explícita: decida, por estado e por evento, se deve aplicar, ignorar ou compensar.

Bloqueios semânticos

Como as sagas abrem mão do isolamento, outros fluxos podem ler e alterar um pedido no meio da saga. Um comprador pode editar o endereço de entrega enquanto a captura está pendente, um vendedor pode cancelar, uma pessoa do suporte pode reembolsar. Os números de versão do Capítulo 9 ajudam com eventos obsoletos, mas não impedem que dois fluxos façam mudanças de negócio incompatíveis.

Um **bloqueio semântico** é uma flag nos dados que diz “uma saga está trabalhando nisto”. Não é um bloqueio de banco de dados. É um valor de status que o restante do código é escrito para respeitar. Na Parcelly, o status de pedido `PENDING` é um deles: enquanto ele vale, o pedido não pode ser editado, enviado nem reembolsado, e a página do pedido mostra “processando”. Do mesmo modo, a reserva no Inventory é um bloqueio sobre a luminária, e é por isso que o segundo comprador viu “sem estoque”.

O bloqueio semântico precisa ser liberado em todos os caminhos: na conclusão, na compensação e no timeout. Uma saga que trava sem liberar o seu bloqueio deixa o pedido congelado, uma falha silenciosa que aparece como tickets de suporte. O timeout acima é a proteção, e a reconciliação do Capítulo 13 é a segunda linha.

Vale conhecer duas contramedidas adicionais. Uma é tornar as atualizações comutativas sempre que possível, para que a ordem não importe. A outra é projetar uma visão conservadora: por exemplo, mostrar ao vendedor uma contagem de estoque que já subtrai as reservas pendentes, para

que ninguém prometa o que está prestes a ser retido. Nenhuma das duas elimina a necessidade de decidir, para cada status, o que os outros fluxos podem fazer enquanto uma saga está em andamento. Considero essa tabela, status por operação permitida, parte do design de qualquer saga.

Outbox e idempotência dentro de uma saga

O handler da saga é onde os capítulos anteriores se encontram. Ele consome uma mensagem, muda o estado e emite o próximo comando ou evento. Feito sem cuidado, é um dual write em dobro.

O formato que funciona combina os capítulos 4 e 7. Cada handler executa uma transação local que faz três coisas: registra a mensagem no inbox, atualiza a linha da saga e grava os comandos ou eventos de saída no transactional outbox. O relay publica a partir daí.


```

function handleStockReserved(message):
  if inboxContains(message.event_id):
    ack(message)
    return
  try:
    transaction:
      // unique event_id: a concurrent duplicate fails here
      inboxInsert(message.event_id)
      saga = loadSagaByOrder(message.payload.order_id)
      if saga.state in ["CANCELLED", "COMPENSATING"]:
        // late success after we gave up: undo it
        outboxInsert(ReleaseStock, saga.order_id,
                     command_id: commandIdFor(saga.saga_id,
"release", message.event_id))
      else if saga.state == "AWAITING_RESULTS":
        saga.reserved = true
        if saga.authorized:
          saga.state = "CAPTURING"
          outboxInsert(CapturePayment, saga.order_id,
                       command_id:
commandIdFor(saga.saga_id, "capture"))
        // a conflict rolls back the whole transaction, inbox
row included
        saveSaga(saga, expected_version: saga.version)
        // any other state: irrelevant, recorded and dropped
        // commit point: inbox, saga state and outbox rows are one
unit
        ack(message)
      on UniqueViolation:
        ack(message)
      on VersionConflict:
        // the redelivery will see the other handler's flag
        nack(message)

```

Esta é a regra que o Capítulo 7 introduziu para consumidores que publicam: a linha do inbox, a mudança de estado e a inserção no outbox fazem commit em uma única transação. Se o processo cai antes do commit, nada aconteceu e o broker faz a reentrega. Se cai depois do commit e antes do ack, a reentrega encontra a linha do inbox, recebe ack e o trabalho é pulado; um duplicado concorrente perde na restrição de unicidade e recebe ack do mesmo modo. Uma mensagem obsoleta ou irrelevante também é registrada e recebe ack, para que nunca volte. A saga avança uma vez por mensagem, independentemente de quantas vezes a mensagem seja entregue.

A verificação de `expected_version` é o que torna a junção correta. `PaymentAuthorized` e `StockReserved` podem ser tratados no mesmo instante por dois workers, e cada um carrega uma linha com as duas flags

falsas. Sem a verificação, a segunda gravação sobrescreve a primeira, e a saga espera em `AWAITING_RESULTS` até que o seu timeout compense um pedido saudável. Com ela, uma transação perde, sofre rollback junto com a sua linha do inbox, e a sua reentrega carrega uma linha que já contém a outra flag. Carregar a linha com um bloqueio de linha alcança o mesmo resultado.

Os comandos que a saga envia são, eles próprios, mensagens que serão entregues at-least-once, então o participante precisa da mesma proteção, e é o `command_id` que a fornece. Para uma ação que não deve se repetir, derive-o da saga e do nome do passo. Isso permite que o Payments reconheça um `CapturePayment` repetido e responda com o primeiro resultado. Se o orquestrador gera um `command_id` aleatório novo a cada tentativa, todo retry parece trabalho novo, e o comprador pode ser cobrado duas vezes. As compensações que podem ser reemitidas seguem a outra regra da seção sobre chegadas tardias: o id vem da mensagem que as disparou, e o participante é idempotente sobre o estado-alvo. O `causation_id` e o `correlation_id` carregados no envelope também tornam cada saga rastreável: o `correlation_id` é o fluxo, e cada `causation_id` é um passo dentro dele.

Aviso: Uma compensação enviada sem chave de idempotência é tão perigosa quanto um pagamento enviado sem ela. Um `RefundPayment` repetido sem uma chave estável pode reembolsar o comprador duas vezes, e o segundo reembolso é dinheiro que não volta.

Onde entram as engines de fluxos de trabalho

Tudo o que foi descrito acima pode ser construído com uma tabela, um temporizador e alguns handlers, e para um fluxo do tamanho do pedido da Parcelly começaria por aí. Em algum ponto, a engrenagem passa a ser a maior parte do código: temporizadores duráveis, retries por passo, versionamento de fluxos de trabalho em execução, um histórico visual de cada instância. É esse o problema que as engines de fluxos de trabalho resolvem.

A classe de ferramentas é ampla. Algumas são bibliotecas ou serviços que persistem o progresso do fluxo de trabalho e o retomam depois de uma queda, com um código que parece um programa sequencial. Outras interpretam uma definição declarativa de estados e transições. Provedores de nuvem oferecem versões gerenciadas, e existem vários projetos open source. Não vou comparar produtos, porque o cenário muda mais rápido do que um livro e a escolha depende da sua linguagem, das suas habilidades operacionais e da sua tolerância a operar mais um componente crítico.

O que perguntaria a qualquer engine é uma lista curta. Onde vive o estado do fluxo de trabalho, e é possível inspecioná-lo? Como ela torna atômicos o efeito colateral de um passo e o seu próprio progresso, ou ela deixa o dual write para você? O que acontece com um fluxo de trabalho já em execução quando você faz o deploy de uma definição alterada? Como compensações e timeouts são expressos? Uma engine que guarda o progresso mas executa o seu passo em uma transação separada ainda exige passos idempotentes; a engine desloca o problema, não o remove.

O resumo honesto é que uma engine economiza o trabalho de construir e operar a máquina de estados, e custa uma dependência, um modelo a aprender e mais uma coisa que pode ficar fora do ar. Para uma ou duas sagas pequenas, uma tabela é mais barata. Para muitos fluxos de trabalho de longa duração com passos humanos e longas esperas, uma engine se paga.

Trade-offs e custos

Uma saga troca atomicidade por disponibilidade e independência, e o preço é pago em várias moedas.

Complexidade. Toda saga multiplica estados. Um fluxo com três passos, cada um com uma ação e uma compensação, mais timeouts e chegadas tardias, tem muito mais estados alcançáveis do que o happy path sugere, e cada um precisa de uma decisão e de um teste. O Capítulo 15 volta a como testá-los.

Estados intermediários visíveis. Não há isolamento. Compradores, vendedores, ferramentas de suporte e outros serviços veem um pedido feito pela metade. Você projeta para isso com bloqueios semânticos e status claros, e aceita que alguns usuários o verão mesmo assim.

Consistência eventual, de propósito. Entre uma falha e a última compensação, o sistema fica inconsistente, por segundos ou, com um timeout ruim, por horas. Isso só é aceitável se o negócio concordou com isso.

Responsabilidade operacional. Alguém precisa vigiar as sagas presas em um estado além do deadline, as compensações que não conseguem ser concluídas e a idade da saga em execução mais antiga. Sem um responsável e um alarme, as sagas falham em silêncio.

Compensar não é de graça. Um reembolso tem taxas e atraso, um envio cancelado pode ter uma tarifa, e uma mensagem ao comprador dizendo “seu pedido foi cancelado” é, ela mesma, um evento visível. Projete o fluxo de modo que a falha mais comum seja barata de desfazer.

Quando não usar uma saga: quando os passos podem compartilhar uma transação local; quando a falha é rara e uma correção manual é mais barata do que construir compensações, o que é uma escolha legítima para um processo interno de baixo volume; quando o “fluxo de trabalho” são, na verdade, reações independentes que não precisam de desfazer coordenado, como enviar um e-mail ou atualizar um índice de busca. E quando você não consegue definir a compensação. Se um passo não pode ser desfeito e não pode ser adiado, a pergunta de design honesta é se a fronteira entre os serviços está no lugar errado.

Erros comuns

- **Construir uma saga onde uma transação bastaria.** É tentador porque os serviços já existem e o diagrama parece moderno. Se dois passos têm o mesmo proprietário e o mesmo banco de dados, junte-os em uma única transação local.
- **Projetar o happy path e deixar as compensações para depois.** O depois chega como um incidente. Escreva a compensação ao lado do passo e dê nome ao evento que ela emite.
- **Chamar todo desfazer de rollback ou de reembolso.** Antes da captura a compensação é uma liberação ou uma anulação; um reembolso só existe depois da captura. Usar o termo errado esconde o estado real no livro-razão e no extrato do comprador.
- **Deixar uma saga esperar para sempre.** Um deadline ausente é invisível até que um cliente ligue. Dê a cada espera, em cada estado não

terminal, um timeout e um resultado nomeado, incluindo as esperas pela resposta da captura e pelas confirmações de compensação.

- **Ignorar eventos que chegam no estado errado.** Um `PaymentAuthorized` tardio para um pedido cancelado é um cartão retido. Decida por estado e evento se deve aplicar, ignorar ou compensar.
- **Compensações não idempotentes.** Uma liberação ou um reembolso repetido que age duas vezes causa um segundo incidente ao consertar o primeiro. Dê às ações que não devem se repetir um `command_id` estável, derive o id de uma compensação reemitível da sua causa e torne os participantes idempotentes sobre o estado-alvo. Um id fixo para uma anulação reenviada repete “nada a anular” e deixa a autorização retida.
- **Pôr o negócio inteiro no orquestrador.** O orquestrador deve decidir a ordem e o resultado; os participantes devem ser proprietários das regras. Se o orquestrador valida níveis de estoque, ele está fazendo o trabalho do Inventory.
- **Pôr passos sem compensação no início.** Enviar o repasse à transportadora antes da captura significa que uma falha deixa você com um envio e sem dinheiro. Ordene os passos de modo que os difíceis de desfazer venham por último.
- **Pular a liberação do bloqueio.** Um bloqueio semântico definido em `PENDING` e nunca removido no timeout ou na compensação congela o pedido. Libere-o em todos os caminhos de saída.

Pontos-chave

- Uma saga substitui uma transação distribuída por uma sequência de transações locais e compensações, e abre mão do isolamento para isso.
- Uma compensação é uma nova ação de negócio, como `StockReleased` ou `PaymentVoided`, não um rollback, e antes da captura nunca é um reembolso.
- A coreografia serve a fluxos curtos com poucos ramos, a orquestração serve a fluxos com junções, timeouts e muitos ramos, e os dois costumam ser misturados, com o orquestrador dentro do serviço proprietário do processo.

- Ordene os passos pela dificuldade de desfazê-los e trate o primeiro passo que não pode ser compensado, como a captura, como um pivot depois do qual a saga só avança.
- Toda espera em todo estado não terminal precisa de um deadline, todo estado precisa de uma regra para eventos tardios e duplicados, e toda compensação precisa ser idempotente e tolerar a chegada antes da ação que desfaz.
- Dentro de um handler, uma transação local deve combinar o inbox, o estado da saga e o outbox, protegida por uma verificação de versão para que handlers concorrentes não se sobrescrevam, e com um `command_id` fixo para uma ação que não deve se repetir, como a captura, mas derivado da mensagem que a disparou para uma compensação que pode ser reemitida.
- Uma engine de fluxos de trabalho pode ser proprietária da máquina de estados, dos temporizadores e do histórico, mas não elimina a necessidade de passos idempotentes nem de uma responsabilidade clara sobre os fluxos de trabalho travados.

Uma saga pressupõe que os eventos descrevem o que aconteceu em uma forma que todo participante consegue ler e em que consegue confiar. Se isso é verdade depende do que cada evento carrega, de como ele se chama e de quem é o seu proprietário, que é o assunto do Capítulo 11.

Capítulo 11 — Design de eventos

Imagine que a equipe de Shipping da Parcelly quer agendar os envios mais depressa. Hoje ela recebe `OrderConfirmed`, que carrega um `order_id` e pouco mais, e depois chama o Orders por HTTP para buscar o endereço de entrega e as linhas do pedido. O Orders fica lento nas manhãs de segunda-feira, e o Shipping desacelera junto com ele. Uma tech lead propõe a solução óbvia: colocar o pedido inteiro no evento, para que o Shipping nunca precise perguntar.

Em uma semana o evento cresceu. O Orders acrescenta o e-mail do comprador, o endereço completo, a conta de repasse do vendedor e um campo chamado `internal_status`, que a sua própria tela de administração usa. O Notifications passa a ler o e-mail do evento, em vez de da sua própria tabela. A equipe do Ledger percebe que a conta de repasse agora trafega pelo broker e fica em um topic com um longo período de retenção. Então o Orders renomeia `internal_status` durante uma refatoração, e três consumidores falham na mesma tarde.

Ninguém nessa história cometeu um erro de confiabilidade no sentido dos capítulos anteriores. O outbox funcionou, os consumidores eram idempotentes, os retries eram sensatos. O dano veio do design da própria mensagem: o que ela diz, o que ela carrega e quem pode alterá-la. Este capítulo pergunta como projetar um evento para que continue útil, seguro e alterável por anos depois de o primeiro consumidor lê-lo.

O problema e por que ele é difícil

O Capítulo 1 apontou que os serviços de um sistema orientado a eventos são acoplados pelo contrato do evento: o seu nome, os seus campos e o seu significado. Esse acoplamento é invisível no momento em que você escreve o evento, porque nesse momento há um produtor e um consumidor, ambos escritos por pessoas que se sentam perto uma da outra. O contrato parece um detalhe interno.

Ele deixa de ser interno no instante em que um segundo consumidor se inscreve, e o produtor costuma ficar sabendo tarde ou nunca. Em um broker baseado em log, um novo grupo de consumidores pode começar a ler o topic sem avisar ninguém. Em um broker baseado em filas, alguém vincula uma nova fila ao mesmo exchange ou à mesma assinatura de topic. A partir desse ponto, cada campo do payload é uma promessa a pessoas que você talvez nunca encontre.

Três coisas tornam isso difícil na prática.

- **É fácil adicionar coisas a um evento e difícil tirá-las.** Adicionar um campo custa uma linha de código. Removê-lo exige encontrar todos os consumidores, inclusive os que estão em repositórios que você não lê e os que fazem replay do histórico antigo.
- **A pressão do design aponta na direção errada.** A equipe produtora quer entregar, então emite o que tem à mão. As equipes consumidoras querem evitar chamadas extras, então pedem mais campos. Ninguém é pago para dizer não.
- **Os erros são duráveis.** Um evento escrito hoje pode ficar em um log de eventos por meses e passar por replay para um consumidor que ainda não existe. Um evento mal nomeado ou um campo pessoal vazado permanece no armazenamento muito depois de o código que o produziu ter desaparecido.

Portanto, o design de eventos não é decoração. É a parte do sistema em que poucas horas de reflexão decidem o quanto cada mudança futura será cara.

Eventos, comandos e documentos

A primeira decisão de design é que tipo de mensagem você está enviando. Três tipos são comuns, e eles têm regras diferentes.

Um **evento** declara um fato que já aconteceu: `PaymentCaptured`. O produtor não se importa com quem escuta e não espera nada em troca. Eventos são nomeados no pretérito, porque relatam algo que já aconteceu; um evento posterior pode contrabalançá-lo, mas não pode desfazê-lo. Um consumidor pode ignorar um evento, e essa é uma resposta legítima.

Um **comando** pede a um serviço específico que faça algo: `CapturePayment`. Ele tem um único destinatário pretendido e a expectativa de que o destinatário aja. Um comando pode ser rejeitado, e quem o envia normalmente se importa com o resultado. Comandos são imperativos, carregam um `command_id` no lugar de `event_id` e nunca são chamados de eventos. Uma saga orquestrada, de que trata o Capítulo 10, conduz os seus participantes com comandos desse tipo.

Uma **mensagem de documento** carrega um bloco de dados sem dizer que algo aconteceu nem que alguém deva agir, como um snapshot noturno do catálogo de um vendedor. O receptor decide o que isso significa. Mensagens de documento são legítimas, mas são o tipo mais frequentemente disfarçado de evento, e o disfarce causa confusão.

Tipo	Diz	Destinatário	Nome	O receptor pode recusar?
Evento	Um fato aconteceu	Quem estiver inscrito	Pretérito: <code>OrderPlaced</code>	Sim, ignorando-o
Comando	Faça isto	Um serviço nomeado	Imperativo: <code>CapturePayment</code>	Sim, respondendo com uma falha
Documento	Aqui estão alguns dados	Quem o ler	Substantivo: <code>CatalogSnapshot</code>	Não se aplica

A falha comum é o comando com roupa de evento. Suponha que o `Orders` publique `SendConfirmationEmail`, um nome imperativo, e o chame de evento porque ele trafega pelo `topic`. O `Orders` passa a depender de o `Notifications` se comportar de determinada maneira, mas não tem como ouvir uma recusa. Se o `Notifications` está fora do ar, o `Orders` não sabe. Se um segundo serviço se inscreve, ele se depara com uma instrução que não é endereçada a ele.

Aplicaria um teste rápido. Pergunte se o produtor ainda estaria correto caso ninguém consumisse a mensagem. Para `OrderPlaced`, a resposta é sim: o pedido foi feito de qualquer modo. Para `SendConfirmationEmail`, a resposta é não, porque a mensagem só tem sentido se alguém agir. Esse segundo tipo pertence a um canal em que o destinatário é nomeado e a falha volta para quem enviou.

Parte disso também é uma questão de direção. Eventos apontam do proprietário de um fato para qualquer interessado. Comandos apontam de um coordenador para um executor. Quando o nome de um evento começa a soar como uma instrução a um serviço específico, a direção foi invertida.

Como é um comando

Os dois tipos também falham de formas diferentes, e o envelope precisa refletir isso. Um evento perdido é silencioso por definição: o produtor nunca esperou que alguém em particular agisse, então ninguém nota a ausência. Um comando perdido ou tardio é diferente. Quem o enviou está esperando um resultado, e a única coisa que revela a perda é o seu próprio deadline, e é por isso que a saga do Capítulo 10 coloca um deadline em todo estado não terminal. Um comando também pode chegar depois que quem o enviou desistiu e compensou, então o receptor precisa de um jeito de recusá-lo.

Um comando, portanto, carrega um envelope menor e diferente do de um evento:

```
{
  command_id:      "cmd-capture-saga-7731",          // stable across
retries of the same intent
  command_type:    "CapturePayment",
  target:          "payments",                      // one named
recipient
  reply_to:        "orders.saga-replies",           // where the
outcome goes
  expects:         ["PaymentCaptured", "PaymentCaptureFailed"],
  correlation_id:   "order-81724",
  causation_id:    "e3",
  expires_at:      "2026-10-06T14:05:00Z",          // optional; judged
on the receiver's clock
  payload:         { order_id: "81724", amount_minor: 4990, currency:
"EUR" }
}
```

`command_id` substitui `event_id`, e o Capítulo 10 explica como ele é derivado: da saga e do passo, para uma ação que não deve se repetir, e com o `event_id` causador, para uma compensação que pode ser emitida de novo. `reply_to` e `expects` dão ao resultado um caminho de volta, de modo que uma recusa seja algo que quem enviou consiga ouvir. `expires_at` permite que um receptor descarte um comando que chega depois de quem o enviou ter parado de esperar. Ele é opcional porque tem

um custo: é comparado com o relógio do receptor, e o Capítulo 3 deu os motivos pelos quais não se pode confiar nesse relógio até o segundo. Defina-o com uma margem e trate-o como proteção contra comandos de horas atrás, não de segundos.

Eventos de domínio e eventos de integração

Dentro de um serviço, o código costuma levantar eventos sobre os seus próprios agregados: `OrderLineAdded`, `DiscountApplied`, `AddressCorrected`. Esses **eventos de domínio** descrevem o modelo interno e mudam sempre que o modelo muda. Eles são úteis dentro do serviço, por exemplo para atualizar um modelo de leitura ou disparar uma regra na mesma transação.

Um **evento de integração** é a versão que cruza a fronteira do serviço. Ele é escolhido de propósito, nomeado pelo que os outros serviços precisam saber e mantido estável. `OrderPlaced` é um evento de integração. Não é um despejo de tudo o que o agregado do pedido era naquele momento.

A tentação é publicar eventos de domínio direto no broker, porque a engrenagem já existe. Na minha opinião, essa é a forma mais comum de variar estrutura interna. Considere o que acontece quando a equipe do Orders divide um agregado em dois ou funde dois tipos de linha. Se os eventos de domínio fossem o contrato público, todo consumidor quebraria, embora nada no negócio tivesse mudado.

A correção é uma etapa de tradução no serviço produtor. Os eventos de domínio continuam livres para mudar. Uma pequena função de mapeamento decide quais deles se tornam eventos de integração, com quais nomes e quais campos. Essa função é onde você decide qual é o contrato público, e o transactional outbox do Capítulo 4 é um lugar natural para chamá-la: o evento de integração é o que é gravado na linha do outbox.

```
function toIntegrationEvent(domainEvent, context):
    // the mapping is the public contract; internals stay behind it
    if domainEvent.kind == "OrderSubmitted":
        return {
            event_id:      newEventId(),
            event_type:     "OrderPlaced",
            schema_version: 1,
            occurred_at:    domainEvent.occurred_at,
            producer:       "orders",
            correlation_id:  context.correlation_id,
            causation_id:   context.causation_id,
            partition_key:  domainEvent.order_id,
            payload: {
                order_id:    domainEvent.order_id,
                buyer_id:   domainEvent.buyer_id,
                line_count:  domainEvent.lines.length,
                total_minor: domainEvent.total_minor,
                currency:    domainEvent.currency
            }
        }
    // most domain events never leave the service
    return nothing
```

Note que o nome interno `OrderSubmitted` e o nome público `OrderPlaced` são diferentes. Isso é permitido, e saudável. O nome público é o que as outras equipes aprendem. O nome interno pode acompanhar o código.

Thin events e fat events

Quanto o payload deve carregar? Vale dar nome aos dois extremos.

Um **thin event**, às vezes chamado de notificação, carrega um identificador e pouco mais: “o pedido 81724 foi feito”. Um consumidor que precisa de detalhes chama o produtor de volta. Um **fat event** carrega estado suficiente para que os consumidores ajam sem chamar de volta. Esse estilo costuma ser chamado de event-carried state transfer, porque o evento move um pedaço de estado de um serviço para outro.

Cada lado tem um modo de falha que os capítulos anteriores ajudam a explicar.

Os thin events trazem de volta o acoplamento síncrono que os eventos deveriam remover. O Shipping só consegue prosseguir quando o Orders está acessível naquele momento, que é exatamente o problema da manhã de segunda-feira da abertura. Eles também criam uma corrida: quando o con-

sumidor chama de volta, os dados podem ter mudado, e o consumidor lê um estado mais novo do que aquele que disparou o evento. Se o callback falha, o consumidor cai no mundo dos retries do Capítulo 6, contra uma dependência que pode ficar lenta por muito tempo.

Os fat events custam em outros lugares. O payload cresce, e o número de campos que você prometeu manter estáveis também. Os consumidores começam a tratar o evento como uma réplica dos dados do produtor, e a réplica envelhece. Um consumidor que guardou em cache o endereço de `OrderConfirmed` na segunda-feira pode usá-lo na sexta, depois de o comprador tê-lo corrigido, a menos que ele leia também o evento posterior. E tudo o que está no payload é copiado para todos os consumidores e para todos os segmentos de log retidos, o que eleva o risco dos campos sensíveis.

	Thin	Fat
Payload	Identificador e tipo do evento	Identificador mais o estado de que os consumidores precisam
O consumidor precisa que o produtor esteja no ar	Sim, para buscar os detalhes	Não
Risco de obsolescência	Baixo na leitura, mas disputa com mudanças posteriores	O consumidor guarda um snapshot que envelhece
Tamanho do contrato	Pequeno	Grande
Exposição de privacidade	Baixa	Maior
Replay	Precisa dos dados atuais do produtor, que podem ser diferentes dos de então	Autocontido, reproduz o que se sabia

Costumo preferir um meio-termo: **carregue o que o consumidor precisa para decidir e agir, e nada do que ele apenas poderia querer**. Para `OrderConfirmed`, o Shipping precisa do id do pedido, das linhas que deve embalar e de um endereço de entrega, e por isso esses itens pertencem ao evento. Ele não precisa da forma de pagamento do comprador, dos dados de repasse do vendedor nem das regras de desconto que produziram o to-

tal. Uma boa maneira de achar a fronteira é perguntar a cada equipe consumidora o que ela faria com cada campo. Um campo sem consumidor nomeado e sem uso declarado é candidato à remoção.

Um segundo detalhe importa aqui. Se o evento carrega estado, diga no payload a que momento esse estado pertence. Com a versão do agregado (Capítulo 4), que o Capítulo 9 chama de `order_version` nos pedidos da Parcelly, um consumidor consegue dizer se o snapshot que guarda é mais novo ou mais antigo do que um segundo evento; `occurred_at` é apenas uma dica aproximada, pelos motivos dados mais adiante. O Capítulo 9 trata de como detectar a chegada fora de ordem com esses números, então este capítulo só observa que um fat event sem versão convida à armadilha do last-writer-wins descrita ali. Não confunda esse número com `schema_version` no envelope: a versão do agregado conta as mudanças de um pedido e avança a cada mudança, enquanto `schema_version` nomeia a versão incompatível do formato do payload e avança raramente (Capítulo 12).

A escolha não precisa ser global. Uma equipe pode começar com thin para eventos pouco usados e adicionar campos quando um consumidor real pedir. Começar thin e crescer é mais barato do que começar fat e encolher, porque encolher significa uma mudança incompatível, que o Capítulo 12 trata em profundidade.

Nomeando eventos

Os nomes de eventos da Parcelly usados ao longo do livro seguem um pequeno conjunto de regras, e esta seção as enuncia com seus motivos.

- **Pretérito**, porque o evento é um fato: `PaymentCaptured`, e não `CapturePayment`, que é o comando.
- **Sujeito primeiro, em PascalCase**: o agregado vem primeiro, depois o que aconteceu. `ShipmentDispatched`, e não `DispatchedShipment`. Em ordem alfabética, os eventos sobre um mesmo agregado ficam agrupados em catálogos e dashboards.
- **Linguagem de negócio, não linguagem técnica**. `OrderConfirmed` diz algo que um product owner entende. `OrderRowUpdated` diz algo sobre uma tabela.

- **Nenhuma tecnologia e nenhum consumidor no nome.**

`OrderPlacedKafkaEvent` vaza o broker. `OrderPlacedForShipping` vaza um consumidor, e o próximo consumidor terá de se perguntar se o evento é para ele.

- **Nenhum prefixo de serviço, a menos que dois serviços emitam fatos sobre o mesmo substantivo.** Suponha que o `Orders` e o `Shipping` possam ambos informar um `DeliveryDelayed`: um significa que o vendedor não entregou o pacote, o outro que a transportadora está atrasada. Os consumidores não conseguem distingui-los pelo nome, então qualificar se torna necessário (`OrderDeliveryDelayed`, `ShipmentDeliveryDelayed`). O exemplo é ilustrativo; os serviços da `Parcelly` neste livro não emitem os dois. Fora isso, o produtor já aparece no campo `producer`.

A regra de nomeação mais importante é a do significado. `OrderUpdated` é um nome que reprova no teste do fato: diz que algo mudou e nada mais. O consumidor precisa comparar o novo estado com o antigo para saber o que aconteceu, e não consegue distinguir um endereço alterado de um total alterado. Prefira nomes que declarem o fato de negócio:

`OrderCancelled`, `ShipmentAddressCorrected`. Quando você não encontra um nome de negócio, muitas vezes é sinal de que o evento descreve uma mudança de banco de dados, e não uma mudança de negócio.

Outra regra é a estabilidade. Quando os consumidores dependem de um nome de evento, renomeá-lo é uma mudança incompatível, com o mesmo custo de remover um campo. Gaste os 10 minutos extras no nome antes do primeiro release.

O envelope

Todo evento neste livro carrega o mesmo envelope: `event_id`, `event_type`, `schema_version`, `occurred_at`, `producer`, `correlation_id`, `causation_id`, `partition_key` e `payload`. O envelope é a parte da mensagem em que as ferramentas podem confiar sem entender o negócio. Um relay, um tratador de dead letters, um sistema de tracing e uma ferramenta de replay conseguem trabalhar com ele sem analisar o `payload`.

Cada campo responde a uma pergunta que algum componente faz:

Campo	A pergunta que ele responde	Quem pergunta
<code>event_id</code>	Já vi exatamente este evento antes?	Inbox, deduplicação (Capítulo 7)
<code>event_type</code>	Como interpreto o payload?	Roteamento, desserialização
<code>schema_version</code>	Qual versão incompatível do formato o payload tem?	Consumidores, schema registry (Capítulo 12)
<code>occurred_at</code>	Quando o fato aconteceu, do ponto de vista do produtor?	Reconciliação, analytics, depuração
<code>producer</code>	Qual serviço é proprietário deste fato?	Responsabilidade, suporte, encaminhamento de perguntas
<code>correlation_id</code>	A qual fluxo de negócio isto pertence?	Suporte, reconciliação, busca em logs
<code>causation_id</code>	O que causou isto diretamente?	Reconstrução da cadeia
<code>partition_key</code>	De qual agregado trata este evento?	Broker, ordenação (Capítulo 9)
<code>payload</code>	Quais são os detalhes?	Lógica de negócio

Dois pontos pedem cuidado. `occurred_at` é o relógio do produtor, e o Capítulo 3 explicou por que você não deve ordenar eventos por ele. Ele diz aproximadamente quando o fato aconteceu e não serve como sequência. A `partition_key` é o identificador do agregado, e a sua escolha foi o assunto do Capítulo 9.

`correlation_id` não é um trace id. O trace context (W3C Trace Context) trafega em headers ao lado do envelope, e o Capítulo 13 mostra como ele é capturado no outbox. Os traces sofrem amostragem e expiram;

`correlation_id` é a chave de negócio que sobrevive a ambos, e é a pela qual uma pessoa do suporte busca.

O envelope entre linguagens

Os serviços da Parcelly não são todos escritos na mesma linguagem. O Orders e o Payments rodam na JVM, o Notifications é Node, e os jobs de reconciliação da equipe do Ledger são em Python. A lista de campos acima diz o que o envelope contém. Ela não diz como cada campo é escrito na transmissão, e é aí que uma segunda linguagem quebra um contrato que parecia bom na primeira.

Campo	Tipo na transmissão	Regra
event_id, correlation_id, causation_id, partition_key	string	Identificadores são strings, mesmo quando parecem numéricos. Um id de 64 bits como número JSON é arredondado em silêncio por um consumidor JavaScript
event_type	string	Um nome estável e neutro quanto à linguagem, como <code>PaymentCaptured</code> , nunca o nome de uma classe ou de um pacote, e nunca um header de tipo que um framework preenche por padrão
schema_version	integer	Pequeno, e um inteiro em todas as linguagens
occurred_at	string	RFC 3339 em UTC com o sufixo <code>Z</code> , com precisão fixa de milissegundos, sem offsets. Os parsers divergem em todo o resto
Dinheiro em payload	integer ou string	Unidades menores inteiras (<code>total_minor</code>) dentro de um intervalo documentado, ou uma string decimal. Nunca um float. Os números do JavaScript são exatos apenas até $2^{53} - 1$, então um campo que pode exceder isso trafega como string
Campos do tipo enum	string	Trate o conjunto como aberto: um consumidor deve lidar com um valor que nunca viu (Capítulo 12)

Duas regras de posicionamento completam o contrato. O envelope vive no corpo da mensagem, e os headers do broker apenas espelham o que o corpo já diz, porque os headers são a parte que se perde quando uma mensagem é copiada, enviada para dead-letter ou transposta para outro broker. E o tipo de conteúdo é declarado uma vez, no esquema, e não adivinhado por mensagem. O que acontece com os campos que um consumidor não conhece, e como os formatos diferem ao decodificá-los, é uma questão de compatibilidade e pertence ao Capítulo 12.

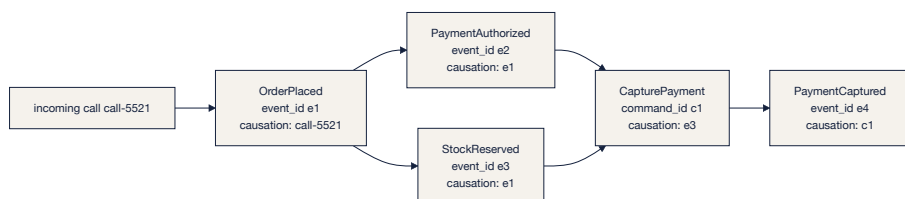
Correlação e causalidade

Os dois identificadores que mais confundem os leitores são `correlation_id` e `causation_id`. Eles respondem a perguntas diferentes.

O `correlation_id` é compartilhado por todos os eventos de um fluxo de negócio. Quando um comprador faz o pedido 81724, tudo o que vem depois carrega `correlation_id: "order-81724"`: a autorização, a reserva,

a captura, o envio. Você busca por ele quando um comprador escreve perguntando o que aconteceu.

O `causation_id` aponta para a única mensagem que causou diretamente esta. É o `event_id` de um evento ou o `command_id` de um comando. Ele não se repete ao longo do fluxo. Cada salto o define como o identificador do seu gatilho imediato, e seguir a cadeia de trás para a frente reconstrói o caminho.



Observe os dois últimos passos. `CapturePayment` é disparado por dois eventos, e o `Orders` precisa escolher um para o `causation_id`, aqui aquele que completou a condição. A ambiguidade é real, e escreveria a regra para que todos os produtores escolham do mesmo jeito. O evento `PaymentCaptured` nomeia o comando, não o evento anterior, e é por isso que o envelope admite os dois tipos de identificadores nesse campo.

Todo evento do fluxo compartilha `correlation_id: "order-81724"`. O primeiro evento de um fluxo define o identificador de correlação, seja derivando-o do agregado, seja carregando um da requisição de entrada. Os eventos seguintes o copiam sem alteração.

Este livro mantém o `order-81724` com escopo de agregado do começo ao fim, e isso é uma escolha com consequência. Um id com escopo de agregado é simples, mas todo fluxo sobre aquele pedido se funde sob ele: um reembolso três semanas depois compartilha o id com o checkout original, e os dois não podem ser distinguidos. A alternativa é um id com **escopo de fluxo**, gerado novo para cada fluxo (`checkout-81724-1`, `refund-81724-1`), com o `order_id` mantido pesquisável no payload. Isso custa mais uma coisa a gerar e a propagar, e vale o custo quando um agregado pode iniciar um segundo fluxo que você precisa distinguir do primeiro. O restante deste capítulo vale de qualquer forma. Copiar é todo o trabalho, e um consumidor que esquece de copiar quebra o rastro de tudo o que vem depois.

```
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
    }
```

A `partition_key` do gatilho só pode ser copiada quando as duas mensagens tratam do mesmo agregado. Um evento de acompanhamento sobre um pagamento ou um envio que herdasse a chave do pedido seria ordenado contra o stream errado (Capítulo 9). A Parcelly usa o pedido como chave de quase tudo, e é por isso que o erro é difícil de notar.

O custo de errar nisso é pago na pior hora. Durante um incidente, a pergunta é “o que aconteceu com este pedido ao longo de cinco serviços?”. Sem um `correlation_id` consistente, uma pessoa de engenharia busca em cinco logs à mão. O Capítulo 13 mostra como o tracing e a reconciliação usam esses campos, então o trabalho deste capítulo é garantir que eles estejam presentes e corretos desde o primeiro evento.

O que não pertence a um evento

Um evento sai do banco de dados do seu produtor e entra em um sistema com regras de acesso diferentes, retenção diferente e muitos leitores. Assim, a pergunta de design é também uma pergunta sobre o que você está disposto a copiar para esse sistema.

Dados pessoais. Um evento com um endereço de e-mail, um telefone ou um endereço de entrega completo é armazenado onde quer que o broker armazene, possivelmente durante toda a janela de retenção, e lido por todos os consumidores. Em um log de eventos, apagar depois os dados de uma pessoa é complicado, porque o log é append-only e os consumidores já copiaram os campos. Algumas equipes respondem com esquemas de

criptografia ou topics separados. Começaria antes: coloque um identificador no evento, como `buyer_id`, e deixe que um consumidor com necessidade legítima busque os campos pessoais no serviço proprietário naquele momento, sujeito às regras de acesso desse serviço. Isso se inclina para `thin events` nos campos sensíveis, e acho que a troca vale a pena. Se um `fat event` realmente precisa de um endereço, como o `Shipping`, então o evento deve carregá-lo deliberadamente, para esse público, com o período de retenção escolhido sabendo que ele está ali.

Quais campos contam como dados pessoais e que obrigações decorrem disso depende da sua jurisdição e dos seus assessores jurídicos. Não conheço as suas restrições. O hábito de design é o mesmo em toda parte: todo campo sensível em um evento deve ter um motivo nomeado.

Secrets e credenciais. Tokens, chaves e números de conta de repasse não pertencem a um payload. Uma mensagem é registrada em log, copiada para destinos de `dead-letter`, exibida em ferramentas de administração e anexada a tickets. Presuma que todo payload será lido por uma pessoa.

Detalhes internos. Nomes de tabelas, códigos de status internos, feature flags, ids de banco de dados que só fazem sentido dentro de um serviço e valores cujo significado depende do código do produtor.

`internal_status` no cenário de abertura é o exemplo. Um campo assim é uma promessa pública no instante em que alguém o lê.

Instruções a consumidores específicos. Um flag como `skip_email_if_gift` diz ao `Notifications` como se comportar. O evento deve declarar o fato, e o consumidor decide o que fazer com ele.

Aviso: Tudo o que entra em um log de eventos deve ser tratado como permanente e amplamente legível. Um `secret` ou um campo pessoal gravado uma vez é copiado para destinos de `dead-letter`, backups e ferramentas de replay, e removê-lo de todos eles é muito mais difícil do que deixá-lo de fora.

Um checklist curto ajuda ao revisar um novo evento: cada campo tem um consumidor que precisa dele, cada campo é algo que o produtor aceita manter estável, e o campo poderia prejudicar alguém se fosse lido por uma pessoa que não deveria vê-lo?

Granularidade

Qual o tamanho de um evento? Há dois erros opostos.

Fino demais. Publicar um evento `FieldChanged` para cada atualização de coluna produz um stream de ruído. Os consumidores precisam remontar o significado a partir de muitas mensagens, e a ordem dessas mensagens passa a importar de maneiras sobre as quais o Capítulo 9 alertou. Uma mudança que era um único fato de negócio, o comprador corrigindo um endereço, agora chega como quatro eventos que podem ser vistos em um estado aplicado pela metade.

Grosso demais. Um evento `OrderChanged` que dispara a cada modificação e carrega o pedido inteiro obriga cada consumidor a comparar o novo estado com o antigo e a adivinhar qual era o significado de negócio. Isso também significa que todo consumidor acorda a cada mudança, inclusive nas que não lhe interessam.

Usaria o fato de negócio como unidade: um evento por decisão ou ocorrência que alguém no negócio nomearia. Um comprador fazendo um pedido é um fato. Um pagamento sendo capturado é outro. Quando uma transação produz vários fatos, publique vários eventos na mesma gravação do transactional outbox, para que façam commit ou rollback juntos.

Um teste prático é perguntar se um consumidor conseguiria reagir de forma significativa a este evento sozinho. Se a resposta for “só depois que os próximos três eventos chegarem”, a granularidade é fina demais. Se a maioria dos consumidores ignora a maior parte do payload, ou se o nome do evento precisa de “e” para descrevê-lo, ele pode ser grosso demais e esconder dois fatos.

Responsabilidade e o contrato público

Todo evento tem exatamente um proprietário: o serviço proprietário do agregado que o evento descreve. O `Payments` é proprietário de `PaymentCaptured`. O `Orders` é proprietário de `OrderPlaced`. Os consumidores podem ler o evento, mas só o proprietário o define. Um consumidor que quer outro formato pede ao proprietário, e não publica uma variante própria com o mesmo nome.

Tratar um evento como um **contrato público** tem consequências práticas, e as escreveria nos acordos de trabalho da equipe.

- **O proprietário documenta o evento:** o seu nome, o significado, quando é emitido, quando não é, cada campo com o seu tipo e significado, e as garantias sobre ordenação e chave de partição.
- **O evento tem uma versão e um processo de mudança.** O Capítulo 12 descreve como evoluir esquemas sem quebrar os consumidores. A regra básica pertence a este ponto: o proprietário não altera um evento publicado sem conhecer as consequências para os seus leitores.
- **Os consumidores se declaram.** Um registro curto de quem consome o quê transforma uma assinatura silenciosa em uma dependência conhecida. Ele não precisa ser sofisticado. Uma lista no repositório que um pull request precise atualizar basta para começar.
- **Os consumidores leem apenas o que precisam.** A regra do tolerant reader do Capítulo 12 começa aqui: ignore os campos que você não usa, para que mudanças aditivas não quebrem o seu código.
- **A emissão faz parte do comportamento.** Um serviço que deixa de emitir `OrderCancelled` em algum caminho de código mudou a sua interface pública tanto quanto um que remove um endpoint HTTP.

As palavras “contrato” e “proprietário” carregam peso. Um contrato pode ser testado. Um proprietário pode receber um page. Se ninguém sabe dizer quem é o proprietário de um evento, ninguém sabe dizer quem pode alterá-lo, e o cenário de abertura se repete.

Trade-offs e custos

A disciplina custa velocidade. Mapear eventos de domínio para eventos de integração, nomear com cuidado, escrever documentação e manter uma lista de consumidores retardam o primeiro release. O retorno chega depois, como mudanças mais baratas. Para um protótipo com um produtor e um consumidor escritos pela mesma pessoa, a cerimônia pode não compensar, e a manteria leve até que uma segunda equipe apareça.

Os fat events trocam o acoplamento no tempo pelo acoplamento no formato. Os consumidores não precisam mais que o produtor esteja acessível, mas dependem dos campos que ele copia. Um payload maior signifi-

ca um contrato maior, mais a manter estável e mais a proteger.

Os thin events são baratos de publicar e caros de consumir. Todo consumidor paga um callback, e o produtor precisa atender essa carga de leitura, que costuma atingir o pico ao mesmo tempo que o evento. Se a API do produtor é lenta, o thin event move o problema do produtor para cada consumidor.

Envelopes cuidadosos custam bytes e atenção. Nove campos em cada mensagem somam em alto volume, e todo produtor precisa preenchê-los corretamente. A alternativa é reconstruir a informação que falta durante um incidente, o que custa mais. O quanto isso importa depende do seu volume de mensagens. A abordagem usual é manter o envelope pequeno e o payload enxuto.

Regras no nível da transmissão custam flexibilidade. Strings para ids e dinheiro, um formato de timestamp fixo e um envelope no corpo tornam um sistema poliglota entediante no bom sentido. Elas também significam converter nas bordas, e uma equipe que quer tipos nativos na sua própria linguagem paga essa conversão a cada mensagem.

A expiração de um comando custa uma dependência de relógio.

`expires_at` protege contra comandos obsoletos, mas é julgado pelo relógio do receptor. A margem que você adiciona é tempo em que um comando genuinamente tardio ainda é aceito.

Eventos anonimizados ou apenas com identificador limitam o que os consumidores conseguem fazer sozinhos. Um consumidor que precisa contatar o comprador precisa de uma chamada ao proprietário ou de uma cópia local dos dados que teve motivo para guardar. Isso é atrito, e é um atrito escolhido.

Quando não pôr tanta estrutura em torno de um evento. Um evento puramente interno dentro de um serviço, consumido pelo mesmo código-fonte e nunca retido, não precisa de envelope, de versão nem de lista de proprietários. Reserve o tratamento completo para os eventos que cruzam uma fronteira e, em particular, para os que sobrevivem a um deploy.

Erros comuns

- **Publicar eventos de domínio diretamente.** É tentador porque a engrenagem existe, e o resultado é que refatorações internas viram mudanças incompatíveis. Adicione um mapeamento explícito para eventos de integração e trate esse mapeamento como o contrato.
- **Chamar um comando de evento.** O pretérito é abandonado ou acrescentado de forma cosmética, e o produtor acaba dependendo de um destinatário específico sem como ouvir uma recusa. Envie comandos a um destinatário nomeado e espere um resultado de volta.
- **Eventos Updated.** `OrderUpdated` não diz nada sobre o que mudou, então cada consumidor compara estados e adivinha. Nomeie o fato de negócio e, se não existe nome, questione se o evento é necessário.
- **Despejar o agregado inteiro no payload.** Economiza uma conversa e cria um contrato público do tamanho da tabela do banco de dados. Pergunte a cada consumidor o que ele faz com cada campo e corte o que ninguém usa.
- **Incluir dados pessoais e campos internos porque estavam à mão.** Corrija isso na hora da revisão, com um checklist, porque depois da publicação os dados já estão em um armazenamento que você não controla.
- **Codificar pelo padrão da linguagem.** O nome de uma classe como `event_type`, ids como números, dinheiro como floats, timestamps com o offset que o runtime escolheu. Funciona até a segunda linguagem ler. Escreva as regras de transmissão uma vez, como faz a tabela acima.
- **Copiar a `partition_key` do gatilho.** Ela está certa quando as duas mensagens tratam de um mesmo agregado e errada para um acompanhamento sobre outro. Derive a chave do agregado do próprio evento.
- **Regenerar o `correlation_id` a cada salto.** Alguém escreve um helper que cria um novo identificador por padrão. Copie-o do gatilho e teste se todo consumidor faz isso.
- **Usar `occurred_at` como ordem.** Um timestamp é uma leitura de relógio, e os relógios divergem. Use a chave de partição e um número de versão, como descreve o Capítulo 9.

- **Nenhum proprietário.** Um evento emitido pela “plataforma” ou por dois serviços com o mesmo nome não tem ninguém que possa aprovar uma mudança. Atribua um serviço proprietário por tipo de evento.

Pontos-chave

- Um evento declara um fato, um comando solicita uma ação a um destinatário nomeado e um documento carrega dados; misturar os três leva a acoplamento oculto e recusas perdidas, e um comando precisa de um caminho de resposta e de uma expiração opcional, porque um comando perdido só é notado pelo deadline de quem o enviou.
- Eventos de domínio descrevem o modelo interno de um serviço, enquanto eventos de integração são o contrato público escolhido deliberadamente, e um mapeamento explícito entre eles impede que a refatoração se torne uma mudança incompatível.
- Os thin events mantêm os contratos pequenos, mas reintroduzem o acoplamento em tempo de execução e as corridas de callback, enquanto os fat events eliminam o callback, mas ampliam o contrato; portanto, carregue o que os consumidores precisam para agir e nada do que apenas poderiam querer.
- Nomeie os eventos no pretérito, com o sujeito primeiro, use linguagem de negócio e evite nomes do tipo `Updated`, que escondem o fato.
- O envelope existe para que as ferramentas consigam deduplicar, rotear, rastrear e ordenar sem ler o payload; a sua codificação de transmissão (ids em string, timestamps RFC 3339 em UTC, dinheiro seguro, nomes de tipo neutros quanto à linguagem) é o que o mantém legível em todas as linguagens, e o `correlation_id` é copiado ao longo do fluxo enquanto o `causation_id` aponta para o único gatilho direto.
- Mantenha dados pessoais, secrets, estado interno e instruções específicas de consumidores fora dos payloads, porque tudo o que está em um log de eventos é copiado amplamente e removido com dificuldade.
- Todo evento tem um serviço proprietário, um significado documentado e uma lista de consumidores conhecidos, e é isso que transforma uma mensagem em um contrato que pode ser alterado de propósito.

Um evento bem projetado ainda precisa mudar, porque o negócio vai mudar, e os consumidores vão se atualizar em velocidades diferentes enquanto os eventos antigos permanecem no log. O Capítulo 12 trata da evolução de esquemas: como alterar um contrato do qual outras equipes já dependem sem quebrá-las.

Capítulo 12 — Evolução de esquemas

Imagine que a equipe de Payments da Parcelly quer oferecer suporte a capturas parciais. Um vendedor envia metade de um pedido de dois itens, e o comprador deve ser cobrado agora por essa metade e pelo restante depois. O evento `PaymentCaptured` carrega um único campo, `amount_minor`, que até agora sempre significou “o pedido inteiro”. A equipe o renomeia para `captured_minor` e adiciona `remaining_minor`, o que é mais limpo e diz o que o campo significa. Os testes do repositório de Payments passam, e o deploy sai no fim da tarde de uma quinta-feira.

O Ledger não está nesse repositório. Ele lê `amount_minor` para lançar um registro de partida dobrada, não encontra nada e rejeita todos os novos eventos como malformados. O Notifications lê o mesmo campo para montar um recibo e imprime um valor vazio. Um job de relatórios que faz o replay dos eventos do mês passado agora encontra duas populações no mesmo topic, nomes antigos e nomes novos, e seu parser trata apenas uma delas.

Nada disso é um bug em Payments. O código fez o que a equipe pediu. A falha é que um evento é lido por programas que a pessoa autora nunca conhece, e alguns deles leem eventos escritos meses atrás. Como mudar um contrato cujos leitores você não controla, enquanto as escritas antigas continuam no log?

O problema e por que ele é difícil

Um esquema de evento é um contrato entre um produtor e todos os consumidores, presentes e futuros. A assinatura de uma função no seu próprio código também é um contrato, mas um compilador verifica os dois lados e você pode mudar ambos em um único commit. Um esquema de evento

não tem compilador entre as duas pontas. Produtor e consumidores fazem o deploy em dias diferentes, pertencem a equipes diferentes, e o broker aceita quaisquer bytes que cheguem.

Três propriedades tornam isso mais difícil do que o versionamento comum de APIs.

Primeiro, **os eventos sobrevivem ao código que os escreveu**. Em um broker baseado em log, os eventos permanecem pelo período de retenção configurado, e alguns sistemas os mantêm indefinidamente. Em um broker baseado em fila a janela é menor, mas um backlog ou uma dead-letter queue (DLQ) pode guardar mensagens por dias. Um consumidor implantado hoje pode ter de ler algo escrito sob um esquema de um ano atrás.

Segundo, **o replay faz do passado uma entrada viva**. O replay, definido no Capítulo 2, é como você reconstrói uma projeção, faz o backfill de um novo serviço ou se recupera de um bug. Ele entrega ao código de hoje todos os esquemas históricos de uma só vez. Um consumidor que trata apenas o formato mais recente funciona até o dia em que alguém faz um replay, e então falha primeiro nos eventos mais antigos. O formato é só metade da questão: o replay também aplica as regras de negócio e os efeitos colaterais de hoje a fatos antigos, como enviar um recibo de novo ou chamar o provedor de pagamentos de novo. Um consumidor reconstruído por replay precisa saber que está em replay, por meio de um modo explícito ou de uma faixa de offsets, suprimir seus efeitos externos e manter regras que sejam válidas para fatos históricos. O Capítulo 7 trata do lado da deduplicação.

Terceiro, **você não consegue atualizar todo mundo de uma vez**. Com rolling deploys, versões antigas e novas do mesmo serviço rodam lado a lado por minutos ou horas. Uma mudança que está correta depois que tudo foi atualizado continua sendo um incidente durante a atualização.

A abordagem óbvia é tratar o esquema como um detalhe interno e corrigir os consumidores depois. Isso funciona enquanto há um consumidor e uma equipe. Deixa de funcionar quando o produtor não sabe quantos consumidores existem, que é exatamente a situação que o design orientado a eventos pretende criar. O Capítulo 11 defendeu que um evento publicado é um contrato público. A evolução de esquemas é como você mantém essa promessa ao longo do tempo.

Compatibilidade, definida

As mesmas palavras são usadas de forma frouxa em muitos lugares, então aqui estão as definições que o livro adota.

- **Compatibilidade retroativa:** um consumidor que usa o novo esquema consegue ler eventos escritos com o esquema antigo. Leitores novos, dados antigos.
- **Compatibilidade futura:** um consumidor que usa o esquema antigo consegue ler eventos escritos com o novo esquema. Leitores antigos, dados novos.
- **Compatibilidade total:** as duas ao mesmo tempo.
- As variantes **transitivas** de cada uma verificam o novo esquema contra todas as versões anteriores, não apenas contra a mais recente.

É fácil inverter a direção na cabeça, então faço uma pergunta concreta: “se fizer o deploy desta mudança, quem quebra?”

Compati- bilidade	Quem lê o quê	Ordem segura de deploy	Protege contra
Retroativa	Leitor novo, da- dos antigos	Primeiro os consumido- res, depois os produtores	Replay do histórico, backlogs atrasados
Futura	Leitor antigo, da- dos novos	Primeiro os produtores, depois os consumidores	Consumidores que não po- dem ser atualizados sob demanda
Total	Ambas as direções	Qualquer ordem	Frotas mistas, consumidores desconhecidos
Nenhuma	Nenhuma das duas é garantida	Coordenada, com um plano de migração	Nada; veja a seção sobre migração

Uma ressalva sobre os nomes. Registries e bibliotecas de serialização usam essas palavras com pequenas diferenças. Leia as definições da ferramenta que você usa antes de confiar em uma configuração e verifique qual vari-
ante o seu registry aplica por padrão: uma regra não transitiva não basta
para um log com replay. Teste com eventos de exemplo reais de cada
versão.

Para um log com replay, a compatibilidade retroativa contra todo o históri-
co é a propriedade de que você precisa, porque um consumidor em replay
pode encontrar qualquer versão já publicada. Para uma fila com retenção

curta, verificar contra a versão anterior pode bastar. Isso é tanto uma decisão de retenção quanto uma decisão de esquema.

Mudanças seguras e mudanças incompatíveis

Uma mudança é **aditiva** quando os consumidores antigos podem ignorá-la e os consumidores novos conseguem lidar com a ausência dela. Uma mudança é **incompatível** quando algum leitor, antigo ou novo, deixa de conseguir interpretar algum evento. Os veredictos abaixo supõem que os leitores se comportam como descrito na seção sobre o tolerant reader.

Mudança em <code>PaymentCaptured</code>	Veredicto usual	Por quê
Adicionar um campo opcional (<code>payment_method</code>)	Segura	Leitores antigos o ignoram; leitores novos tratam a ausência
Adicionar um campo obrigatório sem default	Incompatível	Leitores novos não conseguem interpretar eventos antigos
Remover um campo opcional	Segura só se ninguém o lê	Leitores que dependem dele não veem nada
Remover um campo obrigatório	Incompatível	Leitores antigos o esperam
Renomear um campo	Incompatível	Lê-se como uma remoção mais uma adição; alguns formatos conseguem resolver um renome com aliases
Mudar um tipo (<code>int</code> para <code>string</code>)	Incompatível	Parsers falham ou fazem coerção silenciosa
Ampliar um tipo numérico	Depende do formato	Alguns formatos promovem o tipo, outros não
Mudar o significado de um campo	Incompatível, e do pior tipo	Nada falha; os resultados ficam errados
Adicionar um valor a uma enumeração	Arriscada	Leitores antigos podem rejeitá-lo ou tratá-lo mal

Duas linhas merecem um olhar mais demorado.

Mudar o significado é a mudança perigosa

Renomear um campo quebra de forma barulhenta. O Ledger falha no primeiro evento, um alerta dispara e alguém lê o stack trace. Mudar o que um campo significa, mantendo nome e tipo, quebra em silêncio. Imagine que Payments decide que `amount_minor` passará a significar “valor capturado neste evento” em vez de “valor total do pedido”. Todos os parsers continuam funcionando. O Ledger lança valores parciais como se fossem totais, e o erro aparece semanas depois em uma reconciliação (Capítulo 13) como uma lacuna que ninguém sabe explicar.

Nenhuma ferramenta de esquema detecta isso, porque o esquema não mudou. Só uma pessoa revisando a mudança detecta. Minha regra: se o significado muda, é um novo campo ou um novo tipo de evento, nunca uma edição do antigo.

Enumerações são um contrato oculto

Adicionar um valor a uma enumeração é aditivo no esquema e muitas vezes não é nos consumidores. Um consumidor que mapeia `payment_method` para um ramo de tratamento, com um ramo `else` que supõe “card”, vai tratar mal um novo “`bank_transfer`” sem nenhum erro. Os produtores podem documentar que as enumerações são abertas. Os consumidores podem tratar valores desconhecidos como desconhecidos e encaminhá-los para um erro visível ou para um default seguro, em vez de dobrá-los para dentro de um caso conhecido.

Schema registries e formatos

Um **schema registry** é um serviço que armazena os esquemas usados por um sistema, atribui identificadores a eles e consegue verificar um esquema proposto contra os anteriores. Ele cumpre dois papéis que vale separar.

1. **Consulta.** Uma mensagem carrega, ou aponta para, o identificador do esquema com o qual foi escrita. Um consumidor busca esse esquema e decodifica os bytes. É isso que torna viáveis os formatos binários compactos: os nomes dos campos não viajam em cada mensagem.
2. **Controle de entrada.** Antes de um produtor registrar um novo esquema, o registry o compara com as versões anteriores sob uma regra de compatibilidade escolhida e o rejeita se ele violar a regra.

O segundo papel é o que defenderia com mais firmeza. Regras que vivem em um checklist de revisão são puladas no dia em que a equipe está ocupada. Uma verificação no pipeline de deploy não fica ocupada.

Subjects, topics e tipos de evento

Um registry verifica a compatibilidade por **subject**, e o fato de um subject ser um topic ou um tipo de evento decide se vários tipos de evento podem compartilhar um topic. O compartilhamento é o que a ordenação por agregado entre tipos de evento exige (Capítulo 9): o `PaymentAuthorized` e o `PaymentCaptured` de um mesmo pagamento só permanecem em ordem se viajarem na mesma partição do mesmo topic. Os exemplos da Parcelly supõem um topic por tipo de agregado, como `payments.payment-events`, carregando vários tipos de evento com chave pelo id do agregado, com a compatibilidade configurada por tipo de evento e não por topic. O custo é que os consumidores filtram por `event_type`, e que um topic versionado (Estratégia 3) move de uma só vez todos os tipos de evento daquele agregado. Um topic por tipo de evento é mais simples de configurar e abre mão da ordenação entre tipos.

Três famílias de formato

O livro é agnóstico quanto ao formato, então isto é orientação, não uma comparação de produtos. Avro, Protobuf e JSON Schema diferem no aspecto que os dados têm na transmissão, o que determina o que é seguro mudar.

Família	Na transmissão	Campos identificados por	Consequência prática	Armadilha
Estilo Avro	Binário compacto, sem nomes de campos	Nome, resolvido entre o esquema do escritor e o do leitor	O leitor precisa do esquema do escritor; os <code>de-faults</code> carregam a compatibilidade	Aliases permitem que um leitor resolva um campo renomeado, então um <code>rename</code> nem sempre é incompatível. Um campo ausente é preenchido com o <code>default</code> do leitor, de modo que a ausência é invisível

Família	Na transmissão	Campos identificados por	Consequência prática	Armadilha
Estilo Proto-buf	Binário compacto	Número do campo	Renomear é barato; reutilizar um número é perigoso	Barato apenas na transmissão binária: ProtoJSON e o código gerado usam nomes, então um renome quebra o código-fonte e o JSON. Um escalar proto3 sem presença explícita decodifica um campo ausente como 0
JSON Schema	JSON em texto	Nome; esquema opcional em tempo de execução	Fácil de depurar; a compatibilidade é uma convenção, a menos que seja imposta	Com um modelo fechado (<code>additionalProperties: false</code>), adicionar um campo quebra os leitores antigos, e os registries julgam modelos abertos e fechados de forma diferente

Três ideias valem para todos eles.

- **Defaults tornam as adições seguras, se o default for honesto.** Um novo campo com um default definido permite que um leitor novo processe um evento antigo. Sem ele, uma adição vira uma quebra. Para um campo que o leitor precisa conseguir detectar como ausente, como um valor monetário, o default tem de ser “sem valor”: uma união Avro de `null` e do tipo, com default `null`, ou `optional` do proto3, ou um tipo wrapper. Um default 0 decodifica um evento antigo como uma captura de zero, e o leitor não consegue perceber a diferença. O leitor então verifica a presença, não o valor.
- **Identidade não deve ser reutilizada.** Quer um campo seja identificado por nome ou por número, dar a uma identidade antiga um novo significado é uma quebra. Aposente os identificadores; não os recicle.
- **Campos desconhecidos devem ser tolerados.** Se um decoder mantém, ignora ou rejeita campos que não conhece, isso depende do formato e de suas configurações. Descubra isso antes de projetar em torno dele.

Se você usa JSON sem uma linguagem de esquema, ainda obtém a maior parte do benefício mantendo os arquivos de esquema no controle de versão, validando nos testes do produtor e guardando eventos de exemplo reais por versão. O registry acrescenta consulta central e imposição de regras. Não acrescenta o raciocínio.

Onde mora a versão

O envelope padrão carrega `schema_version` como um inteiro ao lado de `event_type`, e um consumidor lê o par para saber qual formato esperar. Mantê-lo no envelope significa que um roteador, um contador de métricas ou uma ferramenta de DLQ conseguem vê-lo sem decodificar o payload. “Quantos eventos `schema_version: 1` ainda estão chegando?” vira uma consulta barata em vez de um projeto.

Dois números são fáceis de confundir aqui, então este capítulo dá nomes distintos a eles. A **versão incompatível** é o `schema_version` do envelope. Ela só avança para uma mudança que um leitor antigo não consegue tratar: um renome, uma remoção, uma mudança de tipo ou uma mudança de significado. A **revisão do registry** é o identificador que o registry atribui a cada esquema registrado, e ela avança a cada mudança registrada, inclusive as aditivas. Um consumidor rejeita um evento cuja versão incompatível ele não conhece. Ele lê um evento com uma revisão do registry mais nova, mas com uma versão incompatível conhecida, como o formato mais recente que conhece, e ignora o que não reconhece. Nenhum dos dois é a versão do agregado do Capítulo 4, que conta as mudanças de um pedido e não tem nada a ver com o formato do payload.

Se toda mudança aditiva incrementasse `schema_version`, cada consumidor ainda não atualizado rejeitaria todos os novos eventos, e a compatibilidade futura estaria perdida. O preço da divisão é ter dois números para manter distintos. Um exemplo da mesma linha é o BabelQueue (babelqueue.com), um padrão de envelope de código aberto e independente de linguagem que congela seu envelope em `schema_version 1`, permite novos campos apenas como opcionais e ignoráveis, exige um incremento de versão para um renome, uma remoção ou uma mudança de tipo, faz os consumidores rejeitarem uma versão que não conhecem e impede que campos desconhecidos quebrem o processamento, preservando-os quando a mensagem é publicada de novo.

Um evento como Payments o publica durante o passo de dual-publish da migração abaixo:

```
{
  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"
  }
}
```

Compatibilidade entre linguagens

Um esquema pode ser perfeitamente compatível pelas regras do registry e ainda assim quebrar entre duas linguagens, porque as regras descrevem a estrutura e a quebra está em como um runtime a lê. O Capítulo 11 fixou a codificação do envelope. Estas são as armadilhas no payload e no que um consumidor faz com campos que não conhece.

Campos desconhecidos se perdem na reemissão. Suponha que um consumidor Node no Notifications desserialize cada `PaymentCaptured` em uma classe com `order_id`, `captured_minor` e `currency` e copie o evento para um topic voltado a parceiros. Quando Payments adicionar `payment_method` mais tarde, a cópia o descarta em silêncio. Nada falha. O campo simplesmente nunca está lá. Os decoders binários de Protobuf geralmente mantêm campos desconhecidos; muitos mapeadores JSON tipados não. Regra: um salto que encaminha uma mensagem preserva os campos desconhecidos, carregando o payload bruto ou um mapa genérico, ou então publica um novo evento próprio, sob o seu próprio contrato.

Números. `captured_minor` cabe com folga no inteiro de qualquer linguagem. Um identificador ou contador int64 não cabe em um número JSON do JavaScript além de $2^{53} - 1$, e o consumidor o arredonda sem erro. Regra: documente a faixa de cada campo numérico e envie como string tudo o que pode ultrapassá-la, assim como qualquer decimal.

Timestamps. O `timestamp-millis` e o `timestamp-micros` do Avro guardam o mesmo tipo de valor em escalas diferentes. Um escritor em microssegundos lido como milissegundos resulta em uma data a dezenas de milhares de anos de distância, e ainda assim passa pelo parser. Regra: fixe a unidade no esquema e nunca a mude; uma mudança de unidade é uma versão incompatível.

Enumerações. Um valor desconhecido é tratado de modo diferente pelo código gerado: alguns o mantêm, alguns o mapeiam para um default, alguns lançam uma exceção. Regra: trate toda enumeração como aberta e decida no consumidor o que significa “desconhecido”.

Nulo e ausente. Muitos decoders não os distinguem, e alguns preenchem a lacuna com um default. Regra: para campos cuja ausência importa, use as formas que registram presença, do item sobre defaults acima.

Nenhuma dessas armadilhas exige uma versão específica de uma biblioteca específica. Todas exigem que uma pessoa teste com um consumidor em cada linguagem que a Parcelly executa, um custo que os testes de contrato do Capítulo 15 podem carregar.

Estratégias de versionamento

Há três maneiras de introduzir uma mudança, da mais barata à mais cara. Tentaria essas maneiras em ordem e só subiria quando o degrau de baixo não conseguisse sustentar a mudança.

Estratégia 1: adicionar um campo opcional

Se a mudança é algo novo que os leitores antigos podem ignorar, adicione um campo opcional. Adicionar `payment_method` a `PaymentCaptured` é o caso modelo. Isso registra uma nova revisão do registry, e a regra de compatibilidade decide se o registry a aceita. Não muda `schema_version`, porque nenhum leitor antigo perde nada. Incrementar a versão incompatível para uma mudança aditiva faria cada consumidor ainda não atualizado rejeitar os novos eventos, o que anula o objetivo de adicionar o campo de forma compatível.

Essa estratégia não consegue expressar um renome, uma mudança de tipo, a remoção de algo obrigatório nem uma mudança de significado. Para esses casos, suba de degrau.

Estratégia 2: introduzir um novo tipo de evento

Quando o significado muda, diga isso com um novo nome. Suponha que a Parcelly trate a captura parcial como um fato de negócio diferente de uma captura total. O modelo honesto é um novo evento, nomeado pelas regras do Capítulo 11. Este capítulo introduz `PaymentPartiallyCaptured` como um evento adicional.

O evento antigo mantém seu significado e seus consumidores continuam funcionando. O custo é que agora os dois eventos existem, e um consumidor que quer “todas as capturas” precisa assinar ambos. Essa é a troca certa quando o novo fato é conceitualmente diferente, e a errada quando é o mesmo fato com um formato melhor. Esse segundo caso é um problema de representação, e um upcaster se encaixa melhor nele.

Estratégia 3: um topic versionado

Quando uma mudança incompatível atinge o stream inteiro, coloque o novo formato em um novo topic, por exemplo `payments.payment-events.v2`. Os consumidores migram no próprio ritmo, e o topic antigo é aposentado quando seu último leitor sai. Com o layout descrito acima, o novo topic carrega todos os tipos de evento do agregado.

Essa é a opção mais clara e a mais cara. Chaves de partição, ordenação e offsets dos dois topics não têm relação entre si, então um consumidor que passa de um para o outro precisa decidir onde começar a ler e como evitar processar um fato duas vezes. As regras de ordenação do Capítulo 9 se aplicam a cada topic separadamente, e nada ordena eventos entre eles. Em um broker baseado em fila, o equivalente é uma segunda fila com seus próprios bindings, tratamento de dead letters e alertas. Só recorreria a um topic versionado para uma quebra real que um upcaster não consiga contornar, e só quando conseguisse dizer quem vai apagar o antigo.

Upcasting e o tolerant reader

Duas técnicas permitem que os consumidores lidem com várias versões sem obrigar o produtor a congelar.

Upcasting

Um **upcaster** converte um evento escrito sob um esquema antigo para o esquema atual antes que a lógica de negócio o veja. O código de negócio é escrito contra um único formato, e a conversão de cada versão histórica vive em um só lugar.

```
function upcastPaymentCaptured(event):
    p = event.payload
    if event.schema_version == 1:
        // during the migration a v1 event carries either field, or
        both

        // check presence, not value
        if hasField(p, "captured_minor"):
            captured = p.captured_minor
        else:
            // v1 only ever described full captures, so the
            // order amount is the captured amount; may be absent
            captured = p.amount_minor
        return { order_id: p.order_id, captured_minor: captured,
            currency: p.currency }
    else if event.schema_version == 2:
        // v2 carries captured_minor only
        return { order_id: p.order_id, captured_minor:
            p.captured_minor, currency: p.currency }
    else:
        // an unknown newer breaking version must not be guessed at
        raise UnsupportedSchemaVersion
```

O comentário no ramo da versão 1 carrega o trabalho de verdade. Um upcaster codifica uma afirmação sobre o passado, e essa afirmação precisa ser verdadeira. Se alguns eventos antigos eram parciais na prática, o upcaster passa a informar em silêncio um valor parcial como se fosse a captura inteira. Teste os upcasters com eventos arquivados reais, não inventados.

`UnsupportedSchemaVersion` precisa de uma classificação própria. Para o código atual ela é permanente, já que tentar de novo não muda nada, mas um deploy do consumidor a resolve. Por isso o evento fica retido para re-

drive depois que o consumidor for atualizado (Capítulo 8), e não vai para o parking lot, onde esperaria por uma pessoa que não tem nada a decidir.

O lado do consumidor, dentro do handler antes da lógica de negócio, é o lugar mais simples para começar. Reescrever o histórico em um novo topic é útil quando muitos consumidores precisariam do mesmo upcaster. Reescrever eventos armazenados no lugar é uma decisão diferente: sistemas com event sourcing muitas vezes se recusam a fazê-lo, porque o log é evidência. Respeitaria isso, a menos que haja uma razão legal ou de segurança, e então executaria a reescrita como uma migração de dados, com um plano.

O tolerant reader

Um **tolerant reader** pega de um evento apenas o que precisa e ignora o resto. Ele lê `order_id` e `captured_minor` e não se importa que também exista um `payment_method`. É liberal no que aceita e rigoroso quanto ao que usa como dependência.

```
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
    }
```

A tolerância tem dois lados. Ignorar campos desconhecidos é seguro quando os produtores acrescentam coisas. Substituir em silêncio por zero um valor ausente é perigoso. O leitor acima se recusa a continuar quando um campo de que depende está ausente, e pergunta sobre a presença, não sobre o valor. Isso só é honesto se o decoder conseguir distinguir um campo ausente de um zero, e é por isso que os defaults vistos acima importam.

Essa recusa é uma classificação: o evento nunca poderá ter sucesso, então segue o caminho de falha permanente do Capítulo 6 e termina como um dead letter, como descreve o Capítulo 8.

Nota: A tolerância é um acordo entre os dois lados. O produtor promete acrescentar e não mudar o significado. O consumidor promete ignorar o que não entende e recusar o que não consegue fazer com segurança. Se qualquer um dos lados quebrar a sua metade, o outro não consegue salvar o sistema.

O registry verifica o produtor antes de ele publicar. O tolerant reader protege o consumidor depois que algo passa. Não confiaria em um no lugar do outro.

Migrando uma mudança incompatível

Às vezes uma quebra não pode ser evitada: o formato antigo está errado, uma regulamentação obriga a retirar um campo, ou dois serviços precisam ser remodelados juntos. A resposta é transformar um deploy incompatível em vários deploys compatíveis. Esse é o padrão **expand and contract** aplicado a eventos: expandir o contrato para oferecer suporte aos dois formatos, migrar todos os leitores e então contrair, removendo o formato antigo.

Um plano de migração para `PaymentCaptured`

Digamos que a Parcelly realmente precise renomear `amount_minor` para `captured_minor`, e que nada mais no evento mude. O valor mantém o significado: o valor capturado para o pedido inteiro. Trata-se de um renome puro, e o plano abaixo só é seguro por causa disso. As capturas parciais são uma mudança de significado e são tratadas separadamente, depois dos passos.

1. **Registrar a adição.** A nova revisão do registry mantém `amount_minor` e adiciona `captured_minor` como um campo opcional cujo default é “sem valor”, não 0. Sob uma regra de compatibilidade retroativa, o registry a aceita. `schema_version` continua 1, porque nenhum leitor antigo perde nada.
2. **Atualizar os consumidores primeiro.** Cada consumidor aprende a ler `captured_minor` e a recorrer a `amount_minor` quando ele está ausen-

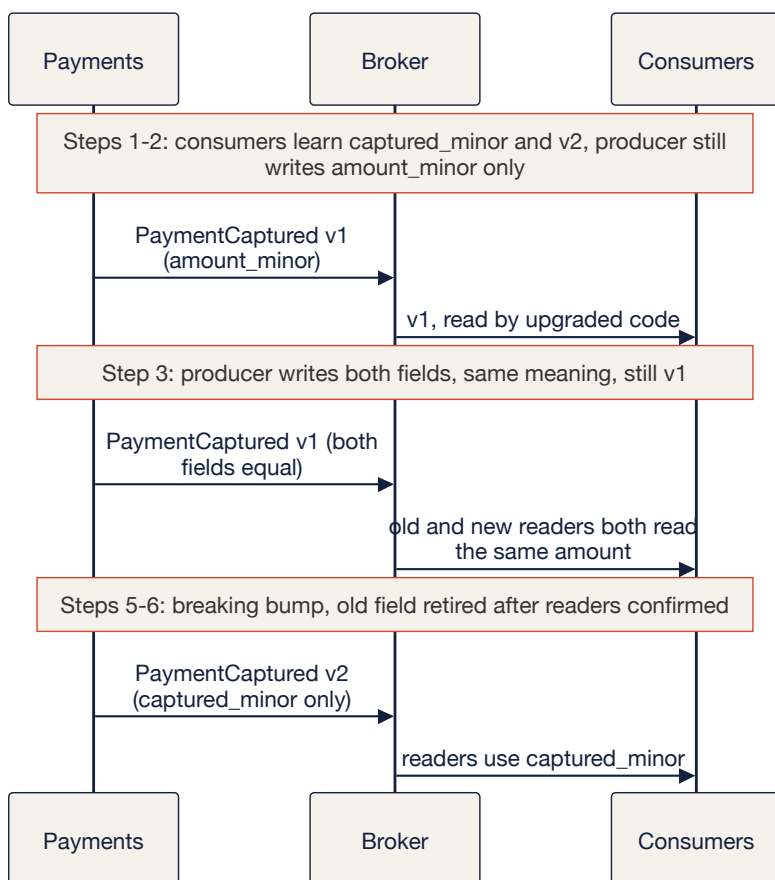
te. Também aprende a ler a versão incompatível 2, que ainda não existe, para que o incremento do passo 5 não encontre ninguém que o rejeite. Leitores novos existem antes dos dados novos.

3. **Dual-publish.** Payments grava os dois campos em cada evento, ainda em `schema_version` 1, ou, se a mudança exige um novo tipo de evento ou topic, publica a forma antiga e a nova do mesmo fato. Isso não é o dual write do Capítulo 4: as duas mensagens saem da mesma transação do transactional outbox, então vale a garantia habitual.
4. **Observar.** Meça quem ainda lê o campo antigo ou o topic antigo. As métricas do produtor sobre `schema_version` mostram o que ainda é escrito; a atividade dos grupos de consumidores no topic antigo, a lista de consumidores do Capítulo 11 e os testes de contrato do Capítulo 15 mostram quem ainda o lê. Um consumidor que nunca se move é uma conversa a ter, não um número a ignorar.
5. **Parar de escrever o formato antigo.** Payments publica apenas `captured_minor` e incrementa `schema_version` para 2, porque este é o único passo que um leitor antigo não consegue suportar. Um consumidor que perdeu o passo 2 rejeita o evento, e ele vai para a DLQ para um redrive depois que esse consumidor for atualizado (Capítulo 8). O passo 4 existe para manter essa lista vazia. `amount_minor` permanece no esquema como obsoleto até que o evento mais antigo que o carrega tenha saído da retenção.
6. **Contrair.** Remova o campo antigo e aposente o topic antigo, se houve um. Remova o fallback dos consumidores só depois que o evento mais antigo sem `captured_minor` tiver saído da retenção, ou transforme-o em um upcaster permanente.

As capturas parciais ficam fora deste plano. Quando uma captura pode ser parcial, `amount_minor` não tem um valor honesto. Se ele carrega o valor capturado, seu significado mudou no lugar, e um leitor que pulou o passo 2, seja por desconhecimento ou por rollback, registra o valor parcial como se fosse o total do pedido, sem nenhum erro. Se ele carrega o total do pedido, o mesmo leitor registra valor demais. O registry aprova as duas versões, porque a estrutura não mudou. Essa é a lição: uma verificação estrutural de compatibilidade não enxerga uma mudança de significado. Coloque o novo significado em um novo campo, em um novo tipo de evento (Estratégia 2, `PaymentPartiallyCaptured`) ou em uma nova versão mai-

or em um novo topic. Depois, não publique capturas parciais enquanto todos os leitores não tiverem passado para a nova forma, e publique-as apenas na nova forma a partir daí.

A ordem é o coração do plano. Para uma mudança retroativamente compatível, os consumidores vão primeiro; para uma mudança com compatibilidade futura, os produtores podem ir primeiro. Quando você não sabe qual dos casos se aplica, execute o plano como está escrito e mantenha os dois campos por um tempo.



Dual-publish entre topics

Quando o plano usa um topic versionado, o passo 3 publica o fato duas vezes, uma em cada topic. Um consumidor que lê os dois não pode processar o mesmo fato de negócio duas vezes. Daria às cópias valores distintos de `event_id`, porque são mensagens distintas, e levaria o id do original em um campo documentado. Seja qual for a escolha, a chave de idempo-

tência do consumidor para esse fato precisa ser definida de propósito. O Capítulo 7 trata das chaves; o ponto aqui é que uma migração é exatamente o momento em que as premissas de idempotência são postas à prova.

Aviso: Durante o dual-publish, nunca deixe um consumidor ler o stream antigo e o novo e aplicar os dois. Um consumidor com duas fontes para um mesmo fato precisa de uma regra de deduplicação, escrita, ou lançará duas vezes uma entrada no ledger.

Rollbacks e histórico

O rollback faz parte do plano. Se eventos da versão 2 já foram publicados, fazer o rollback do produtor não os remove do log. Um consumidor que sofreu rollback para um código que não consegue ler a versão 2 agora falha em eventos que já existem. Portanto, os consumidores precisam conseguir ler a versão 2 antes que qualquer produtor a escreva, e o alvo de rollback de um consumidor precisa conseguir ler todas as versões que estão na retenção, a versão 2 incluída. Um código que lê apenas a versão 1 é um alvo de rollback a evitar, mesmo que tenha sido a última release antes da mudança.

A compatibilidade retroativa lida com eventos antigos sem tocar neles. Se você quer apagar o código de fallback, tem três escolhas: manter o upcaster até os eventos antigos expirarem, reescrever os eventos armazenados em um novo topic, ou tirar um snapshot do estado em outro lugar e deixar o histórico caducar. Cada uma custa algo: código permanente, uma migração em que você precisa confiar, ou a capacidade de fazer replay. Não sei qual delas o seu sistema deve escolher sem saber com que frequência você faz replay e por quanto tempo seus reguladores esperam que você guarde os registros.

Descontinuando eventos

Eventos são mais fáceis de adicionar do que de remover, porque um consumidor que você não conhece pode depender de qualquer um deles. Um ciclo de vida razoável tem quatro estágios.

1. **Anunciar.** Marque o evento como obsoleto no registry ou no catálogo de eventos, indique o substituto, informe uma data de aposentadoria e um contato.
2. **Medir.** Descubra quem ainda o lê: a participação em grupos de consumidores em um log, os bindings em uma fila, as consultas ao registry. Trate a lista como evidência, não como prova, porque um job em lote mensal pode estar ausente hoje e presente na semana que vem.
3. **Parar de produzir.** Depois da data, pare de publicar.
4. **Aposentar.** Remova o esquema e o topic ou a fila quando a retenção tiver eliminado o último evento e os leitores tiverem ido embora.

Duas ressalvas. Uma descontinuação sem prazo nunca termina, então ponha a data por escrito e a mantenha. E um evento obsoleto que passa por replay depois de aposentado vai falhar, então decida o que o replay deve fazer para o período afetado antes de aposentar.

A responsabilidade importa. A equipe do produtor é responsável pelo evento e por sua descontinuação, como o Capítulo 11 descreveu. Os consumidores são responsáveis por suas migrações. Uma política que deixa o produtor decidir sozinho, ou que deixa os consumidores recusarem indefinidamente, vai falhar. A política útil é um aviso do produtor e uma data cumprida pelos consumidores.

Trade-offs e custos

- **Um registry é mais um serviço para operar.** Ele fica no caminho do deploy de todo produtor e, dependendo da configuração, da decodificação dos consumidores. Precisa de disponibilidade, controle de acesso e um responsável, e os clientes precisam de um plano para quando ele estiver inacessível.
- **As regras de compatibilidade desaceleram os produtores, de propósito.** Uma equipe que poderia ter renomeado um campo em uma tarde agora executa uma migração de seis passos. Esse é o preço dos deploys independentes.
- **Os upcasters se acumulam.** Cada versão histórica acrescenta um ramo e um teste, e o código não fica mais simples até que os eventos antigos saiam da retenção. Retenção indefinida significa upcasters indefinidos.

- **Os tolerant readers escondem erros.** Um consumidor que ignora campos desconhecidos também deixa de perceber quando um produtor começa a enviar algo que deveria importar a ele. Tolerância sem monitoramento transforma falhas barulhentas em deriva silenciosa.
- **O dual-publish multiplica o tráfego e o trabalho de deduplicação** enquanto a migração durar.
- **Dois números de versão em vez de um.** A versão incompatível e a revisão do registry impedem que mudanças aditivas façam os consumidores falharem, e custam à equipe o esforço de distingui-las em dashboards e em conversas.
- **Números de versão não são documentação.** `schema_version: 4` diz que algo mudou, não o quê nem quem é afetado. Um changelog mantido junto ao esquema vale mais do que o número.
- **Regras entre linguagens custam conveniência.** Strings para números grandes, preservação do payload bruto e unidades fixas de timestamp significam código de conversão extra em cada linguagem, e um teste por linguagem.

Quando não usar esses padrões

Não monte a maquinaria completa para um evento com um produtor, um consumidor e uma equipe que faz o deploy de ambos juntos. Um deploy coordenado é mais barato, e o risco contra o qual você contrataria o seguro não existe. Reveja quando aparecer um segundo consumidor, especialmente um que pertença a outra equipe.

Não trate o modo estrito de um registry como substituto do design. Uma sequência de adições seguras pode deixar um evento com 40 campos opcionais e sem propósito claro. Se um evento não para de crescer, provavelmente são dois eventos, e valem as regras de granularidade do Capítulo 11.

Não crie um topic versionado para cada mudança. Um sistema que faz isso acaba com muitos streams meio aposentados e ninguém que saiba qual deles é o atual.

Erros comuns

- **Tratar um renome como uma mudança pequena.** A tentação vem de o novo nome ser mais limpo e de os testes passarem. Para cada leitor, é uma remoção mais uma adição. Adicione o novo nome ao lado do antigo, migre os leitores e só então remova o antigo.
- **Mudar o significado de um campo no lugar.** Nome e tipo permanecem, então nenhuma ferramenta reclama. Dê o novo significado a um novo campo ou tipo de evento.
- **Adicionar um campo obrigatório sem default.** Parece uma adição, mas os eventos antigos não o têm. Torne-o opcional com um default, ou introduza-o com uma nova versão e um upcaster.
- **Testar a compatibilidade apenas contra a versão mais recente.** O replay lê tudo o que está na retenção. Verifique contra todas as versões que ainda estão lá e mantenha eventos de exemplo de cada uma nos testes dos consumidores (Capítulo 15).
- **Atualizar os produtores antes dos consumidores.** A equipe do produtor é responsável pela mudança e fica pronta primeiro. Para uma mudança retroativamente compatível, os consumidores vão primeiro. Confirme a ordem antes do deploy.
- **Falhar de forma barulhenta diante de campos desconhecidos.** Um parser estrito transforma toda mudança aditiva em um incidente. Ignore o que você não usa; rejeite apenas aquilo de que depende.
- **Aplicar um default em silêncio a um campo ausente.** Substituir por zero um valor ausente esconde um defeito, e um decoder que preenche 0 o esconde antes que o seu código rode. Use um default “sem valor” para esses campos, verifique a presença e lance um erro, para que o caminho de falha retenha o evento para uma pessoa.
- **Incrementar `schema_version` a cada mudança.** Cada mudança aditiva vira então uma falha grave para todo consumidor ainda não atualizado. Incremente a versão incompatível apenas para mudanças que um leitor antigo não consegue tratar, e deixe a revisão do registry contar o resto.
- **Encaminhar por meio de uma classe tipada.** A cópia descarta todo campo que a classe não declara, e nada reclama. Preserve os campos desconhecidos na reemissão ou publique um novo evento sob o seu próprio contrato.

- **Reutilizar um número de campo ou um nome de topic.** Um número reciclado lê dados antigos como o novo campo, e um topic reutilizado mistura dois formatos. Aposente os identificadores.
- **Descontinuar sem data nem responsável.** O evento se arrasta e a próxima equipe tem medo de removê-lo. Dê a toda descontinuação uma data, um substituto e uma pessoa que responda às perguntas.
- **Fazer rollback de um consumidor sem verificar o esquema.** Confirme que o alvo do rollback consegue ler todas as versões que estão na retenção.

Pontos-chave

- Um esquema de evento é um contrato com leitores que você não controla e com dados escritos há muito tempo; compatibilidade retroativa significa que leitores novos conseguem ler dados antigos, compatibilidade futura significa que leitores antigos conseguem ler dados novos, e um log com replay precisa de compatibilidade retroativa contra todas as versões que ainda estão na retenção.
- Adicionar um campo opcional é seguro, enquanto renomes, mudanças de tipo, novos campos obrigatórios e mudanças de significado são incompatíveis, e a mudança de significado é a pior porque nada falha de forma barulhenta; só uma mudança incompatível move `schema_version`, e a revisão do registry conta o resto.
- Um schema registry conquista seu lugar ao impor a compatibilidade no pipeline de deploy, mas ele verifica a estrutura e não o significado, então uma pessoa ainda precisa revisar o que um campo deve dizer.
- Prefira a estratégia de versionamento mais barata que sustente a mudança: primeiro um campo opcional, um novo tipo de evento quando o significado é um fato diferente, e um topic versionado apenas quando o stream inteiro precisa quebrar.
- Upcasters e tolerant readers permitem que os consumidores lidem com várias versões, desde que o upcaster codifique uma afirmação verdadeira sobre o passado e que o leitor verifique a presença, não o valor, e recuse o que não consegue interpretar com segurança.
- Entre linguagens, as quebras silenciosas são campos desconhecidos perdidos, inteiros grandes, unidades de timestamp, enums fechados e

nulo versus ausente; por isso, preserve o que você não conhece e deixe a representação escrita.

- Migre uma mudança incompatível como uma sequência de mudanças compatíveis, com o incremento no único passo que um leitor antigo não consegue suportar, e planeje os rollbacks no mesmo documento.

Um contrato que evolui com segurança só tranquiliza se você consegue ver o que está fluindo pelo sistema, e é aí que o Capítulo 13 começa.

PARTE V

Operar e comprovar

Capítulo 13 — Observabilidade e reconciliação

Imagine uma manhã de quinta-feira na Parcelly. O suporte encaminha um ticket: um comprador pagou um pedido há dois dias, o extrato do cartão mostra a cobrança, e a página do pedido ainda diz “Aguardando confirmação”. O suporte abre a tela de administração de Orders e vê `PAYMENT_PENDING`. Eles perguntam à equipe de Payments, que encontra uma linha `PaymentCaptured` no banco de dados e um outbox limpo. Perguntam à equipe de Orders, cujos logs do consumidor não mostram nenhum erro para esse pedido. O dashboard de todo mundo está verde.

Três engenheiros agora procuram nos logs de três serviços pelo número do pedido. Uma linha de log usa `order-81724`, outra usa `81724`, e o relay de Payments registra apenas o id interno da linha. Depois de uma hora, eles encontram o evento no outbox de Payments, marcado como publicado. Depois de mais uma hora, descobrem que o topic do broker que alimenta Orders tinha um grupo de consumidores que ficou para trás durante um deploy. A retenção do broker apagou um trecho de mensagens que o grupo não tinha lido, e a política de reset de offset do grupo estava configurada para saltar para a posição mais recente, então ele retomou depois da lacuna sem levantar nenhum erro. O evento existia, foi publicado e nunca foi lido.

Nada disso foi um bug de código no sentido que um revisor apontaria. O sistema fez o que foi construído para fazer, e ninguém conseguia vê-lo fazendo isso. Pior, o ticket existe porque um comprador percebeu antes de qualquer alarme, e ninguém sabe quantos outros pedidos estão na mesma situação. Este capítulo faz duas perguntas. O que uma equipe precisa enxergar para responder “onde está este pedido?” em minutos, e como uma equipe encontra os pedidos sobre os quais ninguém reclamou ainda?

O problema e por que ele é difícil

Em um sistema síncrono, uma falha costuma chegar com um stack trace anexado a uma requisição. Em um sistema orientado a eventos, o trabalho de um fluxo de negócio se espalha por vários serviços, vários bancos de dados, um broker e uma série de momentos no tempo. Nenhum processo isolado guarda a história inteira, e nenhum log isolado a contém.

Três propriedades tornam isso difícil. Primeiro, **os saltos são desacoplados no tempo**. O produtor termina e esquece, e o consumidor pode rodar segundos ou horas depois, em outro processo, sem nada da call stack do produtor. As ferramentas que seguem uma requisição por chamadas aninhadas a perdem na primeira mensagem.

Segundo, **as piores falhas são silenciosas**. O Capítulo 3 descreveu mensagens não lidas que um broker apaga quando um consumidor se atrasa além da retenção. O Capítulo 8 descreveu mensagens que foram processadas de forma incorreta sem levantar um erro. Nos dois casos nada falha, então nada alerta. Um sistema só consegue perceber um evento que nunca chegou se souber que ele deveria ter chegado.

Terceiro, **componentes com aparência saudável podem somar um fluxo quebrado**. O relay está rodando, o broker está de pé, o consumidor não relata erros, e mesmo assim um pedido está parado. A saúde de um componente responde “este processo está vivo?”, e o negócio pergunta “este pedido foi concluído?”. São perguntas diferentes, e a lacuna entre elas é onde este capítulo trabalha.

O Capítulo 3 descreveu a semântica de uma falha como três perguntas: o que pode falhar, como você a observaria e qual garantia sobrevive a ela. Este capítulo trata da segunda pergunta, e do que fazer quando a resposta acaba sendo “você não a observaria”.

A abordagem tem duas metades. A observabilidade dá os meios de seguir um fluxo e de ver a pressão se acumulando. A **reconciliação**, que compara as visões da verdade de dois sistemas e repara as diferenças, dá um jeito de encontrar o que nenhum sinal relatou. Uma equipe que tem só a primeira metade vê as falhas barulhentas. Uma equipe que tem só a segunda encontra os problemas com dias de atraso. Construiria as duas.

Carregando contexto através de saltos assíncronos

Todo evento já carrega dois identificadores do Capítulo 11:

`correlation_id`, compartilhado por todos os eventos de um fluxo de negócio, e `causation_id`, que aponta para o gatilho direto. Eles são a espinha dorsal da história. O distributed tracing acrescenta um segundo fio, que registra o tempo e a relação de pai e filho entre unidades de trabalho.

Um **trace** é uma árvore de spans cronometrados, cada um sendo uma unidade de trabalho em um processo. Dentro de um processo, a biblioteca de tracing mantém o span atual em estado ambiente. Através de uma mensagem, esse estado desaparece, a menos que você o coloque na mensagem. A abordagem padrão é serializar o trace context, um trace id, um span id e uma flag de amostragem, em headers da mensagem. O W3C Trace Context define um formato comum para isso, com dois headers, `traceparent` e `tracestate`, e a maioria das bibliotecas de tracing o segue. A capacidade dos headers e a propagação variam conforme o broker e a ponte: alguns limitam o número ou o tamanho dos headers, e algumas pontes os descartam, então verifique o caminho que as suas mensagens realmente percorrem.

O ponto onde o contexto é capturado importa mais do que parece à primeira vista. O Capítulo 4 colocou uma coluna `headers` no outbox exatamente para isso. O relay roda depois, em outro processo, sem span ambiente, então um relay que injetasse o contexto no momento da publicação anexaria o próprio trace do relay, em vez do trace de quem fez o pedido. O contexto precisa ser capturado quando a linha do outbox é escrita, dentro da transação do produtor.

```
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)
```

No lado do consumidor, há duas maneiras de relacionar o novo span ao antigo. Um span de consumidor pode ser filho do span do produtor, o que desenha uma única árvore. Ou ele pode iniciar um novo trace e manter um **link** para o contexto do produtor. A escolha certa depende do caso.

Caso	Relação	Por quê	O que custa
Mensagem única, consumida logo	Filho do span do produtor	Uma única cascata da requisição ao efeito	Se a entrega atrasa, o trace fica aberto durante todo o atraso e as durações enganam
Lote ou fan-in	Novo trace, um link por produtor	Um span tem apenas um pai, então um consumidor de lote não pode ser filho de doze produtores	O backend precisa oferecer suporte a links, e alguém precisa navegar por eles
Fan-out	Um span de produtor, um span de consumidor filho por assinante	A árvore continua sendo uma árvore	O trace cresce a cada assinante adicionado
Replay, re-drive ou entrega tardia	Novo trace com um link	Um pai de horas atrás esticaria o trace original	A decisão de amostragem é tomada de novo, então um trace de produtor amostrado pode ter um link para um trace de consumidor não amostrado

Ainda adotaria links como padrão, porque o atraso e a reentrega são o caso normal para o qual este livro planeja. O preço é a cascata única de ponta a ponta que a maioria das interfaces de tracing desenha para pai e filho. Para um consumidor que sempre roda poucos instantes depois do seu produtor, pai e filho é a melhor escolha, e a adotaria.

```
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
```

Repare que o span registra `event_id` e a tentativa de entrega. Brokers baseados em log não mantêm contagem de entregas, e apenas alguns brokers baseados em fila expõem uma, então decida de onde o número vem antes de confiar nele. Quando uma mensagem é reentregue, você obtém um segundo span com o mesmo `event_id`. O par diz que a mensagem foi entregue duas vezes, e o campo `outcome` descrito abaixo diz se ela foi aplicada duas vezes.

O tracing tem limites, e teria cuidado para não me apoiar nele como garantia. Os backends fazem amostragem, então o trace do único pedido que lhe interessa pode não existir. Os headers de contexto podem ser descartados por uma ponte, por uma cópia de dead letter ou por um produtor escrito à mão. Uma forma de não depender de headers é carregar o identificador do trace no corpo: o envelope do BabelQueue (babelqueue.com), um padrão de código aberto e neutro quanto à linguagem, carrega no corpo da mensagem um `trace_id` que o primeiro produtor gera, que os produtores seguintes copiam sem alterar e que sobrevive ao envio para a dead-letter queue, de modo que o rastro se mantém mesmo quando um salto descarta os headers. Um trace que termina em uma linha do outbox nunca chega a um consumidor que ainda não leu a linha. Por isso os identificadores no envelope importam mais do que a biblioteca de tracing: `correlation_id` precisa aparecer em toda linha de log e em toda linha de banco de dados que trata o evento, e então ele funciona mesmo com o tracing desligado.



Leia o diagrama atrás da coisa que não muda: o `correlation_id` permanece o mesmo em todo salto, enquanto cada serviço armazena um trace context novo no próprio outbox.

Reconstruindo a história de um pedido

Com os identificadores no lugar, a pergunta “onde está o pedido 81724?” vira uma consulta. A implementação mais barata é uma convenção, não um produto: todo serviço escreve uma linha de log estruturada por evento tratado e por evento publicado, com `correlation_id`, `event_id`,

`event_type`, `causation_id`, resultado e timestamp como campos. Pesquisar esses logs por `correlation_id = "order-81724"` dá todos os saltos em todos os serviços.

```
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
}
```

O campo `outcome` merece atenção. Use um conjunto pequeno e fechado: `applied`, `duplicate_skipped`, `stale_ignored`, `failed_retry` e `dead_lettered`. O Capítulo 9 defendeu que um evento obsoleto ignorado merece um registro, e o Capítulo 7 trata da verificação do inbox que produz `duplicate_skipped`. Quando esses resultados são campos, “quantos eventos este consumidor descartou como obsoletos na última hora?” é uma contagem, e não uma investigação.

Ordenar por `occurred_at` dá uma linha do tempo, e seguir `causation_id` dá uma árvore: `OrderPlaced` causou `PaymentAuthorized`, que com `StockReserved` causou `CapturePayment`, e assim por diante. No cenário da abertura, a árvore teria mostrado `PaymentCaptured` publicado por `Payments` e nenhuma linha `consumed` em `Orders`. O primeiro salto ausente é o lugar onde olhar. Isso é mais útil do que qualquer linha de log isolada: uma lacuna na cadeia é evidência de que algo se perdeu, enquanto uma cadeia longa mostra apenas que o fluxo progrediu.

Se você opera uma saga do Capítulo 10, a linha `saga_instance` dela é uma segunda fonte. Ela responde “o que o orquestrador acredita ser o estado?”, e os logs respondem “o que realmente aconteceu?”. Quando os dois discordam, você encontrou algo que merece um ticket.

Também manteria um **journal de eventos** para os fluxos mais importantes: um registro append-only de envelopes de eventos, cada um com um hash do payload em vez do próprio payload, escrito por cada serviço ao lado do seu inbox. Ele custa armazenamento e alguma carga de escrita, e em troca você consegue responder a perguntas depois que o broker esque-

ceu os eventos, inclusive se um evento reconstruído ainda corresponde ao original (veja a seção de reparo abaixo). Mantenha a retenção longa o bastante para que a sua reconciliação mais lenta consiga rodar.

Métricas que importam

As métricas informam sobre a pressão antes que qualquer mensagem isolada dê errado. A lista de métricas úteis em um sistema orientado a eventos é curta, e a maioria delas já foi sugerida em capítulos anteriores. Começaria com estas.

Sinal	O que mede	Por que importa
Atraso do consumidor (mensagens)	Quão atrás do head o consumidor está	Tamanho do backlog; útil para capacidade
Idade da mensagem, lado do consumidor	Tempo desde que a mensagem não processada mais antiga foi produzida	O que o negócio experimenta
Idade do outbox	Idade da linha não publicada mais antiga	Saúde do relay (Capítulo 5)
Latência de ponta a ponta	Tempo do primeiro <code>occurred_at</code> do fluxo até o efeito no consumidor final	O deadline real do fluxo
Taxa de retries e contagem de tentativas	Parcela das mensagens que precisam de mais de uma tentativa	Degradação escondida pelos retries (Capítulo 6)
Chegadas à DLQ e idade	Novos dead letters e o mais antigo ainda sem revisão	Falhas que precisam de uma pessoa (Capítulo 8)
Contagens de ignorados e duplicados	Eventos obsoletos descartados, duplicados ignorados	Pressão de ordenação e de reentrega
Taxa de escoamento e tempo para se atualizar	Taxa de processamento em relação à taxa de chegada; atraso dividido pela taxa líquida de escoamento	Se um backlog se esvazia antes do seu deadline ou do limite de retenção (Capítulo 6)

Atraso, e por que a idade da mensagem é melhor

O **atraso do consumidor** em mensagens é o número que a maioria dos dashboards mostra primeiro, e é o mais fácil de interpretar mal. Um atraso de 10.000 mensagens significa pouco por si só: se o consumidor escoar

5.000 por segundo, ele está dois segundos atrasado, e se escoa 10 por segundo, está um quarto de hora atrasado. O atraso em mensagens responde a uma pergunta de capacidade. O atraso em tempo responde à pergunta do negócio.

A **idade da mensagem** é o tempo desde que a mensagem mais antiga que ainda está esperando foi produzida. Alguns brokers baseados em fila expõem diretamente a idade da mensagem mais antiga; nos outros, derive-a de um timestamp da mensagem mais antiga em espera. Em um broker baseado em log, derive-a do timestamp do registro no offset cujo commit o consumidor fez por último. Nos dois casos o número está em segundos, de modo que pode ser comparado com um deadline escolhido pelo negócio.

O timestamp do qual você subtrai importa, porque os candidatos medem coisas diferentes. O horário de append do broker mede quanto tempo uma mensagem esperou no broker, que é a saúde do consumidor. `occurred_at` mede o tempo desde que o fato aconteceu, incluindo o tempo que uma linha ficou no outbox esperando o relay, que é a visão do negócio sobre o atraso. O horário de criação do produtor depende da configuração: dependendo dela, um broker de log armazena esse horário ou o horário de append, e um relay que republica uma linha antiga recebe um novo horário de criação, a menos que defina um explicitamente. A minha escolha seria usar o horário de append para o gauge de idade do consumidor e `occurred_at` para a latência de ponta a ponta. Usar `occurred_at` no gauge faz um consumidor saudável parecer muito atrasado no instante em que o relay republica uma linha antiga. A idade de um grupo é o máximo entre as suas partições, e os alertas devem usar esse máximo.

Uma armadilha merece ser nomeada. Uma partição sem tráfego novo tem atraso zero e, com um cálculo ingênuo, idade zero ou uma idade que cresce para sempre. Decida como uma partição ociosa se reporta e teste isso. A mesma função precisa sobreviver à falha de retenção descrita abaixo: um offset com commit feito que ficou para trás do início do log aponta para dados apagados, então essa verificação vem primeiro e é relatada à parte. Um consumidor travado em um registro poison, como no Capítulo 8, mostra throughput zero e uma idade crescente, e essa combinação deve ser o que envia um page para alguém.

```

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)

```

A retenção é o deadline que você não vê

A idade da mensagem tem uma segunda função, menos óbvia. Em um broker baseado em log, um consumidor que se atrasa mais do que o período de retenção perde mensagens em silêncio: o broker apaga dados que nunca foram lidos, e o consumidor continua de onde a sua política de reset de offset o colocar. Em um broker baseado em fila, o mesmo acontece quando as mensagens expiram. O Capítulo 3 deu nome a essa falha, e nenhum erro a acompanha.

Essa política de reset é uma decisão, e cada opção tem um preço. Reiniciar o offset para o mais antigo disponível faz o replay do que sobrou, então a perda é parcial e o consumidor precisa absorver uma enxurrada de mensagens antigas. Reiniciar o offset para o mais recente salta adiante e torna a perda silenciosa, que foi o que aconteceu no cenário da abertura. Falhar, de modo que o consumidor pare e espere por uma pessoa, transforma uma perda silenciosa em um incidente. Escolheria falhar para qualquer fluxo em que uma lacuna custa dinheiro, e aceitaria que o consumidor fica parado até alguém decidir. O Capítulo 14 trata de como se recuperar da lacuna depois que você a percebeu: reiniciar o offset para um timestamp e fazer o replay, com os efeitos colaterais suprimidos.

A defesa é aritmética. Compare a idade da mensagem com o período de retenção ou de expiração, e alerte quando a idade atingir uma fração dele, digamos metade. Nesse ponto o deadline ainda está à frente, e uma pessoa tem tempo de adicionar consumidores, pausar um produtor ou estender a retenção. Depois que o deadline passa, a única evidência é a ausência de eventos, e é isso que a reconciliação existe para detectar. Poria essa razão no mesmo dashboard da própria idade, porque um atraso que é aceitável para o negócio e fatal para a retenção é uma situação real.

Para um conector de CDC, o sinal equivalente é quanto do log do banco de dados o conector ainda não consumiu, comparado com quanto o banco de dados se dispõe a guardar para ele. Um conector que fica fora do ar por mais tempo do que o banco de dados retém o seu log pode precisar de um novo snapshot para se recuperar, e isso é uma indisponibilidade por si só.

Latência, retries e o resto

A **latência de ponta a ponta** precisa de uma definição antes de precisar de um gráfico. Mediria do `occurred_at` do primeiro evento de um fluxo até o momento em que o efeito final tem o commit feito, por exemplo de `OrderPlaced` até `OrderConfirmed`. Reporte percentis, não uma média: a cauda lenta são os compradores que escrevem para o suporte. O consumidor final vê apenas o envelope que consumiu, e esse envelope carrega a hora do seu próprio fato, não o início do fluxo. Ou você propaga a hora de início do fluxo em um header, ou a busca por `correlation_id` no journal de eventos ou no serviço proprietário. Depois, rotule a medição com o `event_type` final. Os dois timestamps vêm de relógios de serviços diferentes, então espere algum skew no resultado; ela é pequena perto de um deadline de negócio, e é uma razão para não alertar sobre diferenças de menos de um segundo.

A **taxa de escoamento** é a forma observável do backpressure. Divida o atraso pela diferença entre a taxa com que o consumidor processa mensagens e a taxa com que elas chegam, e você obtém o tempo para se atualizar. Se as chegadas superam o processamento, o atraso nunca se esvazia, e esse é o número sobre o qual alertar, muito antes de a idade chegar ao limite de retenção. O Capítulo 3 definiu o backpressure e o Capítulo 6 trata de como limitar o trabalho que um consumidor assume.

A **taxa de retries e a contagem de tentativas** revelam o problema que o Capítulo 6 chamou de falhas escondidas: uma dependência degradada há semanas enquanto os retries a absorvem em silêncio. O histograma da contagem de tentativas é mais informativo do que a contagem de erros, porque uma mensagem que sempre tem sucesso na terceira tentativa não produz nenhum erro.

As **chegadas à DLQ** e a idade do dead letter mais antigo ainda sem revisão vêm do Capítulo 8. Some a isso as **contagens de duplicados e de obsoletos** dos capítulos 7 e 9. Uma contagem de duplicados em alta logo após um deploy é normal, já que a posse do trabalho se move, e uma contagem em alta em um horário estável aponta para handlers lentos. Em um broker baseado em fila, a causa habitual é um visibility timeout menor do que o tempo de execução do handler. Em um broker baseado em log, é um rebalanceamento, ou um consumidor que excede o seu poll timeout e perde as partições.

Alertas que significam algo

A falha da maioria das configurações de alertas não é a falta de dados. É que os dados enviam um page para pessoas por causas em vez de por efeitos, e que, depois de algumas semanas de ruído, ninguém mais confia no pager. Submeteria os alertas a um único critério: se este disparar às três da manhã, uma pessoa nomeada sabe dizer o que fazer a respeito.

Alerte sobre **sintomas ligados a um deadline de negócio**. “A linha não publicada mais antiga do outbox tem mais de dois minutos” é um sintoma de que algo entre uma mudança de estado e o seu anúncio está travado. “A CPU do consumidor está acima de 80 por cento” é uma causa, e pode ou não importar. Use as métricas de causa em dashboards onde alguém investiga, e reserve o pager para os sintomas.

Dê a cada alerta um **responsável e um limite com uma razão**. O Capítulo 8 fez a mesma observação para as DLQs: um e-mail perdido e uma captura perdida merecem limites diferentes, e a diferença deve estar escrita ao lado do alarme. A razão importa porque os limites derivam, e um número sem razão declarada é elevado na primeira vez que incomoda.

Alerte sobre a **ausência, e não só sobre a presença**. Um relay que cai não emite erro; não emite nada. Um grupo de consumidores que perdeu todos os seus membros mostra atraso, mas um topic que parou de receber eventos por completo não mostra atraso nem erros. Para fluxos com um ritmo estável, como pedidos em horário comercial, um alerta como “nenhum evento `OrderPlaced` observado há 10 minutos” pega um silêncio do lado do produtor que todos os outros sinais deixam passar. O limite precisa acompanhar o padrão de tráfego, e para fluxos sem ritmo essa é a ferramenta errada.

Por fim, dois níveis bastam para a maioria das equipes: um page para tudo o que ameaça um deadline ou perde dados, e um ticket para pressões lentas, como uma DLQ crescendo com mensagens de baixo valor. Um terceiro nível tende a ser ignorado.

Aviso: Elevar o limite de um alerta para parar o ruído é como uma equipe troca um incômodo por um ponto cego. Se um alerta dispara com frequência demais, corrija a causa ou a definição dele. Não aumente o número até ele parar de disparar.

Reconciliação

Métricas e traces observam o que o sistema faz. Eles não conseguem informar sobre um evento que nunca entrou no sistema, sobre um handler que fez commit do estado errado sem um erro, nem sobre um reparo que alguém aplicou à mão. Para esses casos, você compara o sistema com um registro independente do que deveria ser verdade. Isso é a reconciliação.

Um job de reconciliação pega duas visões dos mesmos fatos e relata as diferenças. Os pares costumam ser óbvios quando você os procura:

- **Orders contra Payments:** todo pedido no estado `PAID` tem um pagamento capturado, e todo pagamento capturado pertence a um pedido que sabe dele.
- **Payments contra Ledger:** todo `PaymentCaptured` tem um `LedgerEntryPosted` correspondente, com o mesmo valor na mesma moeda.

- **Seus registros contra os de um provedor:** o relatório de liquidação do provedor de pagamentos contra os seus próprios registros de captura. O Capítulo 7 apontou para cá no caso de efeitos que não podem ser tornados idempotentes, em que a reconciliação é a última linha de defesa.
- **Um modelo de leitura contra a sua fonte:** um índice de busca ou uma view que o Capítulo 9 disse que podia ser marcado como suspeito e reparado depois.

Três tipos de diferença cobrem a maioria dos casos: **ausente** (a fonte tem, o lado derivado não), **obsoleta** (os dois têm, mas o lado derivado discorda, em geral porque está atrasado) e **extra** (o lado derivado tem algo que a fonte desconhece). Cada uma aponta para uma causa diferente. Ausente sugere eventos perdidos ou nunca consumidos, obsoleta sugere ordenação ou atraso, ou de vez em quando um handler que gravou o valor errado, e extra sugere um bug ou uma mudança manual.

A parte que merece cuidado é que eventos em trânsito se parecem exatamente com discrepâncias. Um pagamento capturado há quatro segundos e que ainda não está no Ledger é normal. Um job que o relata produz ruído e ensina as pessoas a ignorá-lo. Por isso o job compara apenas registros mais antigos que um **período de carência** escolhido a partir da sua latência normal de ponta a ponta, e confirma uma divergência uma segunda vez antes de relatá-la.

```

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)

```

Vários detalhes deste bloco carregam a correção. A janela é limitada e o corte é aplicado, então o job lê dados que tiveram tempo de se assentar. Cada lote é uma busca por chave em vez de um join entre bancos de dados, porque os serviços da Parcelly não leem as tabelas uns dos outros, então o job busca o outro lado pela interface do próprio serviço ou por um extrato publicado. O job apenas registra discrepâncias e não altera nada. E como ele faz upsert em uma tabela de discrepâncias com chave por item e tipo, rodá-lo duas vezes, ou depois de uma falha no meio de uma janela, produz a mesma tabela, sem uma pilha crescente de linhas duplicadas. A mesma função fecha as discrepâncias quando os lados concordam, que é o passo em que o código de reparo abaixo se apoia. Um pagamento lançado

duas vezes no Ledger é pego pela passagem direta, que conta as correspondências. A passagem reversa é o que encontra uma entrada “extra” sem nenhuma captura por trás, como o lançamento de um pagamento que nunca foi capturado, e uma verificação em uma só direção nunca encontraria.

Escolha a cadência pelo dano que uma discrepância causa. Payments contra Ledger é financeiro e merece uma execução frequente, com uma comparação completa diária como retaguarda. Um índice de busca pode precisar de uma execução noturna. A saúde do próprio job também importa: alerte se ele não tiver terminado na janela esperada, porque uma reconciliação que parou em silêncio é pior do que nenhuma, já que as pessoas acreditam que ela está vigiando.

Nota: Um job de reconciliação é um consumidor dos dados que verifica, e pode estar errado. Trate uma discrepância primeiro como uma pista para investigação. As primeiras semanas de um job novo são gastas, em sua maior parte, encontrando falsos positivos causados por timing, arredondamento ou uma regra de comparação que não corresponde ao negócio.

Reparo e backfill

Encontrar uma discrepância é metade do trabalho. A outra metade é mudar o estado para que ele corresponda, sem criar uma segunda discrepância.

A regra a que me prenderia é que **os reparos passam pelo mesmo caminho das mudanças normais**. Se o Ledger não tem uma entrada para um pagamento capturado, não insira a linha com um console de banco de dados. O Capítulo 3 tratou as ações do operador como parte do modelo de falhas, e uma linha editada à mão não emite evento, então todos os outros consumidores desse fato continuam errados, e a próxima reconciliação relata uma inconsistência de sua própria autoria. Em vez disso, republique o fato, ou envie o comando que o causa, pelo outbox do produtor, para que os consumidores o tratem como tratam todo o resto.

Um reparo pode passar por retry, ser interrompido e ser executado duas vezes por dois engenheiros, e precisa continuar seguro em cada caso. A idempotência do Capítulo 7 só responde a isso se o evento de reparo carregar o `event_id` **original**. Um evento de reparo com um id novo é um evento diferente para todos os consumidores. Se o primeiro evento apenas atrasou, ele chega depois do reparo, e o Ledger lança o pagamento duas vezes. Por isso o reparo republica o fato que já existe: o mesmo envelope, com o mesmo `event_id` e o mesmo `causation_id`, e o inbox trata como duplicado a cópia que chegar em segundo lugar, seja qual for.

Isso exige que o produtor guarde o id original. A forma mais simples é armazenar `event_id` no registro de negócio (aqui, a captura) quando a linha do outbox é escrita, junto com um hash do payload. Se a linha do outbox ainda existe, ela é marcada como não publicada e o relay a envia de novo. Se foi removida, o envelope é reconstruído a partir do registro com o id armazenado.

A reconstrução tem um limite. O registro é o estado atual, e o evento é um fato do passado. Se o registro mudou desde então, digamos que um reembolso parcial ajustou o valor, um payload reconstruído a partir dele difere do original embora carregue o mesmo `event_id`. É por isso que o hash é armazenado: o reparo reconstrói o payload, compara o hash dele com o armazenado e para, à espera de uma pessoa, se forem diferentes. Um registro que não tem id nem hash armazenados, em geral um escrito antes de o hábito existir, precisa recorrer a um id derivado da sua chave de negócio e a campos que não podem mudar, e então o consumidor também precisa estar protegido por uma restrição de unicidade nessa chave, como descreve o Capítulo 7. Adicionaria essa restrição ao Ledger de qualquer forma, como segunda linha de defesa.

Uma republicação também **se espalha em fan-out**. O topic entrega o envelope a todos os assinantes, não só ao Ledger. Um assinante cujo inbox ainda guarda o `event_id` o ignora. Um cuja linha do inbox foi removida, porque o evento é mais antigo que a sua janela de deduplicação (Capítulo 7), o aplica uma segunda vez, e o Notifications poderia enviar o mesmo e-mail de novo. Portanto há dois caminhos seguros. Se o evento é mais novo que a menor janela de deduplicação entre todos os assinantes, republique pelo outbox do produtor, como acima. Se é mais antigo, entregue o reparo apenas ao consumidor que o perdeu, pela fila ou topic de reparo desse

próprio consumidor, ou reiniciando os offsets desse único grupo, e confie em uma restrição de chave de negócio como proteção. O Capítulo 8 aplica o mesmo princípio ao redrive.

```
function repairMissingLedgerEntry(discrepancy):
    capture = loadCapture(discrepancy.item_key)
    if ageOf(capture.event_id) > shortestDedupWindow():
        // a broadcast could be applied twice where the inbox row is
    gone
        return sendRepairToConsumer("ledger", capture, discrepancy)
    try:
        transaction:
            original = outboxFindByEventId(capture.event_id)
            if original exists:
                // same envelope and event_id; the relay publishes it
    again
                outboxMarkUnpublished(original.id)
            else:
                // purged: rebuild the identical envelope, never a new
    id
                envelope = rebuildOriginalEnvelope(capture)
                if hashOf(envelope.payload) != capture.payload_hash:
                    // the record changed since the event was
    published
                    raise RepairNeedsReview(discrepancy.id)
                    outboxInsert(envelope)
                    recordRepairAttempt(discrepancy.id, actor, now())
                // commit point: the repair is durable and the relay delivers
    it
                on UniqueViolation as e:
                    if e.constraint != "outbox_event_id_key":
                        // a different constraint: a real error
                        raise e
                    // a concurrent repair inserted the same event_id first
                    return
                // resolution is recorded by the next reconciliation, not here
```

Dois detalhes carregam a segurança. Marcar de novo como não publicada uma linha não publicada é inofensivo quando repetido, e a violação de unicidade de uma segunda inserção concorrente é tratada como “já tratado” em vez de um erro. O envelope republicado também mantém o seu `causation_id` original, de modo que a cadeia de causação continua verdadeira, e o vínculo com a discrepância fica no registro do reparo. O último comentário é o importante. A função de reparo não marca a discrepância como resolvida. Quem faz isso é a próxima execução da reconciliação, quando os dois lados concordam. Isso impede que um reparo que foi publicado e depois se perdeu seja relatado como corrigido.

O reparo em massa, em geral chamado de **backfill**, acrescenta escala. Um backfill de 30.000 eventos em um consumidor ativo compete com o tráfego normal e pode empurrar a latência do fluxo além do seu deadline. Execute-o em pequenos lotes, com um rate limit e um interruptor de pausa, como o Capítulo 8 recomendou para o redrive, e acompanhe a idade da mensagem e o atraso do consumidor enquanto ele roda. Em um broker baseado em log, você pode preferir um replay deliberado para um grupo de consumidores separado que reconstrói uma view derivada e depois faz a troca. Isso é mais limpo quando o consumidor é uma projeção, e errado quando o consumidor tem efeitos colaterais, como enviar e-mail.

O reparo pelo produtor serve para um punhado de registros. Quando o dano é mais amplo, como um grupo de consumidores que pulou uma lacuna de retenção inteira, um banco de dados restaurado ou um cluster de broker perdido, republicar registro por registro deixa de ser a ferramenta certa. O Capítulo 14 trata da recuperação por replay, dos desastres e da migração de brokers.

Todo reparo precisa de um registro: quem o executou, que discrepâncias ele cobriu, o que publicou e quando. A trilha de auditoria é como o próximo engenheiro descobre que o pico de duplicados às 14:10 foi um backfill, e é como um órgão regulador ou a equipe de finanças descobre depois que uma entrada do Ledger era uma correção.

Invariantes de negócio

A reconciliação compara dois sistemas. Uma **verificação de invariante** compara um sistema com uma regra que o negócio considera verdadeira. Os invariantes costumam ser mais baratos de avaliar e detectam problemas que nenhum segundo sistema conseguiria, como um bug que faz dois sistemas concordarem com a resposta errada.

Para a Parcelly, alguns exemplos mostram o formato:

- Nenhum pedido está **CONFIRMED** sem um pagamento capturado.
- Nenhuma reserva fica retida além do seu deadline.
- Nenhuma instância de saga fica em **AWAITING_RESULTS** depois do seu `deadline_at`.

- Todo pagamento capturado tem exatamente uma entrada no Ledger, nem zero nem duas.
- A soma das entradas do Ledger de um pedido fecha em zero.

Cada um é uma consulta que não deveria retornar nenhuma linha. Execute-a de forma agendada e trate cada linha retornada como uma discrepância com um responsável. O exemplo da saga é o que o Capítulo 10 deixou em aberto: o handler de timeout é a primeira proteção contra um pedido congelado, e uma consulta de invariante sobre instâncias travadas é a segunda, porque o handler que deveria ter disparado pode ele mesmo ter falhado.

```
table invariant_violation:
  invariant_name text not null
  item_key       text not null
  first_seen_at  timestamp not null
  last_seen_at   timestamp not null
  resolved_at    timestamp null
  primary key (invariant_name, item_key)
```

Usar como chave o invariante e o item significa que uma violação que persiste é atualizada, não duplicada, e `first_seen_at` diz há quanto tempo um problema está aberto. Essa idade é, por si só, uma métrica sobre a qual vale alertar.

Escreveria os invariantes com as pessoas de produto e de finanças, e não para elas. Os engenheiros sabem o que o sistema consegue fazer, e nem sempre sabem o que o negócio considera um defeito. Uma regra como “um reembolso nunca excede a captura” é óbvia para as finanças e pode não estar escrita em lugar nenhum do código.

Trade-offs e custos

Observabilidade e reconciliação não são gratuitas, e não rodam sozinhas.

A telemetria tem um preço. Logs estruturados para cada evento tratado multiplicam o volume de logs, e labels de alta cardinalidade, como `order_id` nas métricas, podem sobrecarregar um backend de métricas. Mantenha os identificadores nos logs e nos traces, e limite os labels das

métricas a conjuntos pequenos, como serviço, topic e tipo de evento. Faça amostragem dos traces e aceite que o trace de que você precisa pode não existir.

Um journal de eventos duplica dados. Vale a pena para fluxos em que “o que aconteceu?” tem peso legal ou financeiro. Para um fluxo de baixo valor, como uma notificação cosmética, os logs bastam, e o diário é um custo sem leitor.

A reconciliação lê dados de produção. Um job de comparação pode sobrecarregar o banco de dados de origem, e um que seja descuidado com o processamento em lotes pode deixar mais lento o serviço que ele protege. Execute-o contra uma réplica quando houver uma e limite o tamanho e a taxa dos seus lotes. Ele também acrescenta uma dependência: se a interface do outro lado estiver fora do ar, o job não consegue rodar.

Alertas custam atenção. Cada page gasta a credibilidade do próximo. Um pequeno conjunto de alertas bem atribuídos faz mais do que um conjunto grande que ninguém lê.

O reparo tem um risco próprio. Um mecanismo de reparo é um caminho para alterar o estado de produção fora do fluxo normal de requisições. Sem auditoria, sem aprovações para os tipos perigosos e sem um design idempotente, ele troca uma classe de problema por uma pior.

Quando não construir tudo isso: um sistema com um serviço e um consumidor, de baixo risco e fluxos curtos, não precisa de um journal de eventos, de tabelas de invariantes nem de uma reconciliação noturna. Logs estruturados com `correlation_id` e um alerta sobre a idade da mensagem são o mínimo que manteria em qualquer escala. O resto cresce com o custo de uma falha silenciosa. Se um evento perdido custa uma discrepância no ledger, construa a reconciliação primeiro. Se custa um resultado de busca desatualizado, construa-a por último.

Erros comuns

- **Alertar sobre a saúde, e não sobre o fluxo.** Gráficos verdes de CPU e memória dizem que os processos estão vivos e não dizem nada sobre os pedidos serem concluídos. A tentação vem de as métricas de infra-

estrutura já existirem. Adicione os alertas de sintoma: idade da mensagem, idade do outbox e violações de invariantes.

- **Observar o atraso só em mensagens.** Uma contagem de mensagens não tem um deadline associado. Converta para idade e compare a idade com o deadline de negócio e com o período de retenção.
- **Injetar o contexto do trace no relay.** O relay não tem span ambiente, então anexa o errado, ou nenhum. Capture o contexto quando a linha do outbox é escrita e encaminhe-o sem alterar.
- **Identificadores inconsistentes.** Um serviço registra `order-81724`, outro `81724`. A busca por correlação falha exatamente quando é necessária. Defina o formato uma única vez no design de eventos e imponha-o com um helper de logging compartilhado. Mantenha esse mesmo formato também para o tráfego sintético: o Capítulo 15 marca as sondas com um header separado e nunca com um `correlation_id` especial.
- **Usar um único timestamp para todas as idades.** O horário de criação do produtor, o horário de append do broker e `occurred_at` medem atrasos diferentes. Use o horário de append para o gauge de idade do consumidor e `occurred_at` para a latência de ponta a ponta.
- **Republicar um reparo para todos os assinantes.** Um consumidor cuja linha do inbox foi removida aplica o evento de novo. Confira o evento contra a menor janela de deduplicação, ou entregue o reparo ao único consumidor que o perdeu.
- **Reconciliar sem um período de carência.** O job relata como discrepância todo evento em trânsito, a equipe aprende a ignorá-lo, e o único achado real se perde no ruído. Compare apenas registros assentados e confira de novo antes de relatar.
- **Reparar editando linhas à mão.** Isso corrige uma tabela, não emite evento e deixa errados todos os consumidores a jusante. Repare pelo produtor e pelo outbox, para que a mudança viaje como qualquer outra.
- **Um reparo que não é idempotente.** Uma segunda execução duplica a correção. Republique com o `event_id` original, para que o inbox reconheça tanto uma segunda execução quanto um original tardio.
- **Deixar a reconciliação se deteriorar.** O job quebra depois de uma mudança de esquema, ninguém percebe, e a equipe acredita estar co-

berta. Alerta quando uma execução atrasa ou falha e revise o backlog de discrepâncias em uma agenda.

Pontos-chave

- Um fluxo espalhado por serviços precisa de identificadores que sobrevivam a cada salto: `correlation_id` em toda linha de log e de banco de dados, e o contexto do trace armazenado na linha do outbox para que o relay encaminhe o contexto certo.
- O tracing ajuda, mas é amostrado e pode ser descartado, e links versus pai e filho é um trade-off que depende do caso, então os identificadores do envelope precisam bastar por si sós para reconstruir a história de um pedido.
- A idade da mensagem, medida contra um deadline de negócio e contra a retenção do broker, é mais útil do que o atraso em mensagens, e é o sinal que avisa antes de um consumidor atrasado perder eventos em silêncio; a política de reset de offset decide se essa perda é silenciosa ou barulhenta.
- Alerta sobre sintomas ligados a deadlines, dê a cada alerta um responsável e uma razão declarada para o seu limite, e inclua alertas para o silêncio, porque as falhas que perdem dados muitas vezes não produzem nenhum sinal.
- A reconciliação encontra o que nenhuma métrica relata: ela compara registros assentados entre duas visões da verdade, classifica as diferenças como ausentes, obsoletas ou extras e as registra sem alterar nada.
- Os reparos precisam percorrer o caminho normal, republicando o `event_id` original para que um original tardio e um reparo não possam ter efeito ambos, com uma trilha de auditoria, e uma discrepância é fechada pela próxima comparação bem-sucedida, não pelo reparo em si. Uma republicação alcança todos os assinantes, então eventos antigos vão apenas ao consumidor que os perdeu.
- As verificações de invariantes de negócio, como nenhum pedido confirmado sem um pagamento capturado, detectam defeitos que dois sistemas concordantes esconderiam.

A observação e a reconciliação dizem que o sistema está se comportando mal, e o reparo conserta um punhado de registros. O Capítulo 14 trata do que fazer quando o dano é mais amplo do que um reparo alcança: recuperar-se por replay, sobreviver a um desastre e migrar para um broker diferente.

Capítulo 14 — Recuperação, desastres e migração de brokers

Imagine uma terça-feira na Parcelly. O cluster de broker gerenciado que transporta todos os eventos do marketplace fica inacessível. A página de status do provedor diz que a zona que o hospeda falhou e que um cluster substituto estará disponível em até uma hora. Quando ele chega, os topics existem e estão vazios, ou guardam dados até algum ponto dos últimos minutos. Ninguém sabe dizer exatamente qual é esse ponto.

Os serviços estão bem. Orders, Payments, Inventory, Shipping e Ledger têm cada um o seu banco de dados, e todos os bancos estão intactos. Cada serviço tem um relay que continua tentando publicar de novo, e os outboxes vão enchendo. Os consumidores estão ociosos, esperando por um broker que não está lá. Enquanto isso, a equipe precisa decidir o que fazer quando ele voltar, e todas as opções parecem plausíveis. Apontar os consumidores para o novo cluster e deixá-los rodar. Republicar tudo a partir dos outboxes. Restaurar o último backup de alguma coisa. Cada opção é certa para um armazenamento e errada para outro.

Este capítulo trata do dia seguinte àquele em que um armazenamento dá errado, desaparece ou é substituído de propósito. Ele faz uma única pergunta. Quando o broker, o banco de dados ou a região deixa de ser o que era, quais das garantias que você construiu continuam de pé, e como usá-las para voltar a um estado em que se possa confiar?

O problema e por que ele é difícil

Os capítulos anteriores defenderam o sistema contra falhas que deixam os armazenamentos intactos: um crash entre dois passos, um consumidor lento, um duplicado, uma poison message. Uma recuperação é diferente, porque o que deu errado é a própria memória. Um armazenamento voltou no tempo, ou perdeu parte do seu histórico, ou está sendo trocado por ou-

tro. Toda defesa deste livro depende de algum armazenamento lembrar de alguma coisa, e por isso a pergunta passa a ser qual memória sobreviveu e qual não.

Isso é difícil por dois motivos. Primeiro, os armazenamentos falham de forma independente e em direções diferentes. Um banco de dados restaurado para seis horas atrás e um broker que perdeu seus últimos dois minutos discordam entre si de um jeito que nenhum dos dois consegue detectar. Ambos são consistentes consigo mesmos. O Capítulo 3 chamou isso de perder um armazenamento inteiro e observou que, no início, nada gera um erro.

Segundo, o trabalho de recuperação é raro, e trabalho raro é feito mal sob pressão. Os comandos que movem um grupo de consumidores para trás, as configurações que impedem um replay de mandar uma segunda rodada de e-mails, a ordem em que os serviços são reiniciados: tudo isso são decisões baratas de tomar com antecedência e caras de improvisar.

A tese deste capítulo é a tese do livro em sua forma mais crua. O broker não é o que sobrevive. O que sobrevive a um cluster perdido, a um banco restaurado ou a uma troca de produto é a semântica de falhas que você projetou em torno dele: o transactional outbox (Capítulo 4) que lembra o que nunca foi confirmado, o inbox que reconhece o que já foi feito, o identificador estável, o contrato e o hábito de comparar duas visões da verdade. Se tudo isso está no lugar, o broker é substituível. Se não está, o broker carregava a correção do sistema, e ninguém sabia.

Recuperação por replay

O **replay**, definido no Capítulo 2, é a releitura deliberada do histórico a partir de um log de eventos. Ele se aplica apenas à família dos logs, porque uma fila apaga o que entrega. O Capítulo 15 usa o replay como técnica de teste. Aqui ele é uma ferramenta de recuperação, e tem um procedimento.

A situação em que ele se encaixa é a que abriu o Capítulo 13: um grupo de consumidores ficou para trás, a retenção apagou mensagens que ele não tinha lido, e o grupo seguiu adiante, deixando uma lacuna. Se os dados

ainda estão retidos, a primeira recuperação a considerar não é reproduzir os eventos um a um a partir do produtor. É mover a posição do grupo para antes da lacuna e ler de novo.

Fazendo o reset de um grupo para um ponto no tempo

A posição de um grupo de consumidores é armazenada como um offset por partição. Movê-la para trás é uma ação administrativa no broker, e a maioria dos brokers baseados em log aceita como alvo um offset explícito ou um timestamp, que o broker resolve para o primeiro offset naquele horário ou depois dele. Prefiro o timestamp. Um offset é um número que só significa algo para uma partição de um topic em um cluster, e o timestamp é o que você sabe: o deploy começou às 09:40, então volte para 09:30. Confira na documentação do seu broker como ele resolve um timestamp, já que o horário do produtor e o horário de append do broker nem sempre são o que ele usa.

O procedimento é curto, e a ordem importa.

1. Pare o grupo. Fazer o reset dos offsets de um grupo em execução falha ou é sobrescrito pelo commit seguinte, dependendo do broker.
2. Registre os offsets atuais antes de mudar qualquer coisa. Esse é o único caminho de volta se o reset estiver errado.
3. Escolha o alvo com uma margem. Volte mais do que você acha necessário, porque duplicados são baratos e uma lacuna não é.
4. Faça o reset para o alvo, reinicie o grupo e acompanhe a idade da mensagem não processada mais antiga (Capítulo 13) enquanto ele drena.
5. Continue acompanhando até o grupo voltar ao ponto mais recente e então reconcilie, porque o reset repara apenas o que o log ainda guardava.

Escolher uma margem só é seguro por causa do inbox (Capítulo 7), que transforma a releitura das mensagens que o grupo já processou em no-ops. Um grupo cujos consumidores não são idempotentes não deve sofrer reset, diga o que disser o procedimento.

Limitando o replay a um grupo de consumidores

Um log permite que 10 grupos leiam o mesmo topic, e cada um tem seus próprios offsets. Essa é a propriedade que torna seguro o reset direcionado. Faça o reset do grupo que perdeu os eventos, e nenhum outro grupo percebe. Faça o reset do menor conjunto de grupos que perderam alguma coisa e nomeie esses grupos no plano antes de executá-lo. O consumidor de `OrderPlaced` em Payments mantém sua posição enquanto o grupo de Inventory relê o último dia.

Uma segunda forma da mesma disciplina é usar um grupo separado para o replay. Se uma projeção precisa ser reconstruída, por exemplo a visão de busca de Orders, inicie um novo grupo com outro nome que leia desde o começo para dentro de uma tabela nova, e troque o leitor quando ele tiver alcançado o ponto atual. O grupo antigo nunca se move, então não há nada a desfazer se a reconstrução der errado. Isso custa armazenamento e um segundo consumidor por um tempo, um preço pequeno por um roll-back que não custa nada.

Suprimindo efeitos colaterais

Um replay aplica o código de hoje aos fatos de ontem. Se um handler envia e-mails, chama o provedor de pagamento ou notifica uma transportadora, o replay faz com que ele repita tudo isso. O Capítulo 12 levantou esse ponto, e a regra que ele deu é o ponto de partida: um consumidor reconstruído por replay precisa saber que está em replay.

A primeira ideia que as pessoas têm é uma flag na mensagem, um header que diz “isto é um replay”. Isso não funciona, porque uma mensagem reproduzida é a mensagem original. O broker guarda os bytes que o produtor escreveu, e nada indica que estão sendo lidos pela segunda vez. O modo pertence ao consumidor, não à mensagem. O grupo em replay roda como um deployment separado, com um valor de configuração como `mode: replay`, ou a camada de efeitos verifica o nome do grupo.

Com o modo conhecido, há quatro opções para os efeitos em si, e cada uma tem um preço.

- **Colocar um gate nos efeitos, na borda.** Toda chamada externa passa por uma única função que verifica o modo e, em replay, não faz nada. É o desenho mais claro e o mais barato de auditar, mas só funciona se

nenhum handler contornar o gate. Um novo handler que chame um provedor diretamente o derrota, então a regra precisa de um teste (Capítulo 15).

- **Colocar chaves de idempotência nas chamadas externas.** Se a chamada de captura carrega uma chave derivada de `event_id`, o provedor reconhece a segunda tentativa. O Capítulo 7 tratou disso. A proteção vale apenas enquanto o provedor se lembrar da chave, então é uma segunda linha de defesa, e não aquela em que se deve confiar para um replay do mês passado.
- **Pausar o outbox dos eventos derivados.** Um handler em replay escreve novas linhas no seu próprio outbox na mesma transação, e essas linhas seguem para o próximo serviço. Se os novos eventos carregam identificadores novos, os consumidores a jusante os tratam como fatos novos. Ou se deriva o identificador de cada evento de saída de forma determinística a partir do `event_id` de entrada e do nome do handler, de modo que a segunda execução produza os mesmos identificadores e os inboxes a jusante os absorvam, ou se mantém retido o relay do serviço em replay até verificar o que ele produziu.
- **Fazer o replay em outro ambiente.** Para uma projeção sem efeitos, uma tabela nova é a resposta mais limpa, como acima. Para um handler com efeitos, um deployment diferente, com a camada de efeitos ligada a um stub, é uma forma de testar o replay antes de executá-lo de verdade.

Aviso: Fazer o reset de um grupo que envia e-mails, cobra cartões ou chama transportadoras, sem gate e sem chaves, transforma cada evento histórico em uma nova ação. O inbox não ajuda se você também fizer o reset dele, e para efeitos externos nunca poderia ajudar, porque o provedor não faz rollback junto com o seu banco de dados. Decida o método de supressão antes de mover os offsets, não depois da primeira reclamação.

A janela de deduplicação e o inbox

O inbox muda o que um replay faz, e isso pode trabalhar a seu favor ou contra você, dependendo do que você quer.

Se o banco de dados do consumidor está intacto e o intervalo que você está relendo cabe dentro da sua janela de deduplicação (Capítulo 7), o inbox reconhece cada `event_id` que já processou e o ignora. É isso que você quer depois de uma lacuna: o grupo processa apenas o que falta, e o resto custa pouco. É também por isso que um replay consegue reparar o consumidor que nunca viu um evento, sem perturbar os que viram.

Se o intervalo é mais antigo que a janela, as linhas do inbox já se foram. O consumidor aplica eventos antigos pela segunda vez. Para um consumidor protegido por uma restrição de chave de negócio, como uma entrada de ledger por captura, a segunda aplicação falha sem causar dano. Para um consumidor sem essa proteção, o efeito acontece duas vezes. Por isso a janela de deduplicação faz parte do plano de recuperação: antes de um replay, declare o evento mais antigo do intervalo, a menor janela entre os grupos que o verão e o que protege o consumidor onde os dois não se sobrepõem. O Capítulo 7 deu a regra de que a janela precisa durar mais que qualquer replay que você planeje. Este é o plano.

O inbox também pode ser o obstáculo. Um consumidor com um bug escreveu estado errado. Você corrige o bug e faz o replay, e toda mensagem é ignorada, porque todo `event_id` está no inbox. Esse caso aparece em breve.

Replay, reparo e redrive

Três ferramentas movem mensagens de novo, e elas diferem no que atuam.

Ferramenta	Sobre o que atua	Capítulo	Quando é a escolha certa
Reparo	Um punhado de registros, republicados pelo outbox do produtor com o <code>event_id</code> original	13	Poucos itens estão faltando ou errados, e o produtor sabe dizer quais
Redrive	Dead letters, devolvidas ao consumidor que as rejeitou	8	As mensagens falharam de forma visível e a causa foi corrigida
Replay	Um intervalo do log, relido por um grupo	14	Muitas chaves foram afetadas, os dados ainda estão na retenção e o consumidor é idempotente

A pergunta que distingue as três é quem sabe o que está errado. No reparo, o produtor ou um job de reconciliação nomeia os itens específicos. No redrive, o próprio sistema separou as mensagens. No replay, ninguém tem uma lista: o dano é um intervalo de tempo, e a forma mais barata de cobrir um intervalo é ler tudo de novo e deixar o inbox separar o que já foi feito. O replay troca precisão por cobertura.

A retenção é um limite de recuperação

O replay só alcança até onde o log guarda dados. A retenção, uma configuração escolhida há muito tempo pensando no custo de disco, é portanto também o limite da recuperação mais barata que você tem. Uma indisponibilidade que dura mais que a retenção, ou um bug descoberto depois dela, tira você do caminho barato e o leva a republicar a partir dos produtores e reconciliar.

Definiria a retenção em função do tempo de que a equipe precisa para perceber e corrigir um problema, e não apenas em função do disco. Se um job de reconciliação roda diariamente e uma correção leva um dia para ir ao ar, uma retenção de um dia não deixa nada. A retenção também interage com o lado dos custos: uma retenção maior aumenta o número de esquemas antigos que um replay pode encontrar (Capítulo 12) e prolonga o tempo que os dados pessoais vivem no log. Não sei qual é o número certo para o seu sistema. Ele depende da rapidez com que você percebe o problema, de quanto tempo o reparo leva e do que o log pode guardar.

Um consumidor que errou sem falhar

O Capítulo 3 tem uma linha para um bug de consumidor sem erro: o handler teve sucesso e escreveu o resultado errado. Esta é a recuperação que mais precisa do replay, e aquela em que o inbox atrapalha.

Imagine que o consumidor de `PaymentCaptured` em Ledger converteu valores com o expoente errado para uma moeda, durante uma semana. Toda mensagem recebeu ack. Nada falhou, e o inbox guarda todos os `event_id`. A reconciliação (Capítulo 13) acaba relatando que os totais de Ledger para essa moeda divergem dos de Payments. A correção vai ao ar, e agora as linhas erradas existem e precisam ser corrigidas.

Fazer o replay do topic para o mesmo consumidor não adianta nada, porque o inbox diz que cada evento já foi feito. Há dois caminhos honestos, e a escolha depende do que Ledger é.

Se o estado do consumidor é derivado e pode ser descartado, como uma projeção, reconstrua-o. Inicie um novo grupo apontando para um novo armazenamento, com o inbox vazio, faça o replay a partir do primeiro evento afetado, verifique o resultado contra a origem com a reconciliação e troque. O estado errado é descartado, não corrigido.

Se o estado do consumidor é um registro que não pode ser reescrito, como em um ledger append-only, o replay ainda ajuda, mas não para sobrescrever. Leia o intervalo afetado com o código corrigido em um modo de correção. Para cada evento, calcule o que a entrada deveria ter sido, compare com o que está lançado, já líquido das correções anteriores, e lance uma entrada de correção para a diferença, com um identificador derivado do `event_id` original e do nome da própria correção, de modo que executar a correção duas vezes seja um no-op. A restrição de unicidade sobre o identificador da própria entrada do ledger faz esse trabalho, então a correção não precisa de uma linha no inbox, e o loop trata uma chave duplicada como “já corrigido” e passa ao evento seguinte.

As entradas erradas permanecem, como história. As correções são fatos novos que dizem por quê. Isso é mais lento e mais explícito que uma reconstrução, e é a única forma que um auditor aceita.


```

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

```

O fio condutor é que o replay encontra o intervalo afetado a baixo custo, e a decisão sobre o que fazer com cada evento pertence à semântica do próprio consumidor.

Cenários de desastre

Um desastre é uma falha em que a perda não é de uma mensagem, mas de um armazenamento. A forma útil de organizar as possibilidades é por qual armazenamento voltou no tempo. Escreva cada cenário no padrão que o Capítulo 3 usou: o que quebrou, como isso aparece e qual garantia ainda se sustenta. A última das três é a útil, porque nomeia aquilo em que você está se apoiando, e um fluxo sem nada para colocar ali está exposto nesse cenário. Antes dos cenários, uma palavra sobre o que você está buscando.

Objetivos para cada armazenamento

Dois números descrevem uma recuperação. O **objetivo de ponto de recuperação (RPO)** é a quantidade de dados que você pode se dar ao luxo de perder, medida em tempo: um backup a cada seis horas significa um RPO de até seis horas. O **objetivo de tempo de recuperação (RTO)** é por quanto tempo você pode ficar sem o serviço. Ambos são decisões, e o erro habitual é tomar uma só para o sistema inteiro.

Eles diferem porque os armazenamentos diferem. Para os armazenamentos deste capítulo, um esboço fica assim.

Armazenamento	O que uma perda custa	Caminho típico de volta	O que determina
Um banco de dados de serviço	Estado que não existe em nenhum outro lugar, a menos que um evento o carregue	Backup mais log shipping	RPO por banco de dados, definido pelo valor de negócio dos dados
O outbox	Eventos ainda não publicados	Mesmo banco de dados, portanto mesmo RPO	Qualquer linha perdida é um evento que nunca existiu
O inbox	A memória do que foi processado	Mesmo banco de dados dos efeitos	Volta no tempo junto com eles (Capítulo 3)
O broker	Eventos em trânsito e histórico retido	Replicação, ou republicação a partir dos produtores	O RPO é definido pela replicação, o RTO pela rapidez com que os consumidores drenam
Uma projeção derivada	Nada, se puder ser reconstruída	Replay	O RTO é o tempo do replay, o RPO é zero se o log guarda tudo
Um provedor externo	Efeitos fora do seu controle	Os registros dele, e a reconciliação	Fora dos dois objetivos

O ponto da tabela é que um RTO para a projeção reproduzida é um cálculo de drenagem. Se o log guarda um mês e o grupo consegue processar um dia de dados por hora, a reconstrução leva cerca de 30 horas, e esse é o seu RTO (suponha esses números; os seus vêm de medição, como no Capítulo 13). Uma equipe que nunca mediu a sua taxa de drenagem não conhece o seu RTO.

Também associaria os objetivos ao modelo de falhas do Capítulo 3, fluxo a fluxo. O fluxo de captura de pagamento tem uma tolerância diferente da do fluxo de notificação, e a tolerância do Ledger a uma lacuna está mais perto de zero. Um objetivo uniforme acaba rígido demais para as notificações e frouxo demais para o ledger.

Restaurando um banco de dados a partir de um backup

Suponha que o banco de dados de Orders seja restaurado para um ponto seis horas atrás. O que isso faz com a maquinaria deste livro?

O **outbox** volta com o estado. As linhas escritas nessas seis horas se foram, junto com os pedidos que anunciavam, embora Payments e Inventory possam já ter agido sobre os eventos. As linhas que estavam não publicadas no ponto da restauração, mas foram publicadas depois na linha do tempo antiga, voltam a ficar não publicadas, então o relay as publica uma segunda vez. É um duplicado inofensivo, porque o `event_id` é o mesmo.

Há um perigo mais agudo. Se os identificadores de pedido vêm de uma sequence, a sequence restaurada foi rebobinada, e o próximo pedido receberá um identificador que já foi dado, a um pedido diferente, na linha do tempo perdida. Payments agora guarda uma captura para `order-81724` que pertence a um comprador de que Orders se esqueceu, e o novo `order-81724` pertence a outra pessoa. Por esse motivo preferiria identificadores aleatórios e opacos para os agregados. Se você precisa manter uma sequence, avance-a além do maior valor já emitido antes de aceitar tráfego, um número que se pode ler nos serviços a jusante.

O **inbox** de um consumidor cujo banco de dados é restaurado volta junto com os efeitos. O Capítulo 3 apontou que isso é correto: as mensagens processadas antes do ponto da restauração continuam no inbox e seus efeitos sobreviveram, e as mensagens posteriores são esquecidas nos dois lugares. Essa propriedade só vale quando o inbox e os efeitos vivem no mesmo banco de dados.

O que não volta é o **offset** do consumidor, que vive no broker. O grupo está posicionado no ponto mais recente, além de seis horas de eventos de que o banco restaurado se esqueceu. O consumidor nunca os verá de novo a menos que alguém mova o grupo para trás, então o plano de restauração inclui um reset desse grupo.

O alvo exige cuidado, porque o ponto da restauração é um horário no relógio do banco de dados, enquanto um reset por timestamp trabalha com os horários dos eventos no log. No momento da restauração, o grupo não estava no ponto mais recente. Estava em alguma posição, e o evento nessa posição havia sido produzido algum tempo antes, em até a idade da mensagem do grupo (Capítulo 13). Então o alvo é o horário do evento na posi-

ção do grupo naquele momento, menos uma margem. Se o alvo for definido apenas a partir do ponto da restauração, um grupo que estava uma hora atrasado pula a hora que ainda não havia processado quando o banco voltou no tempo.

Leia a posição a partir do histórico da idade das mensagens ou dos offsets registrados, e mantenha esse histórico fora do banco de dados que você está prestes a restaurar. Algumas equipes armazenam o offset no mesmo banco de dados dos efeitos, na mesma transação, para que os dois sejam rebobinados juntos. Isso elimina o passo manual e assume do broker o controle contábil dos offsets, um custo a ser avaliado.

Os efeitos externos ficam onde estavam. A captura que Payments enviou ao provedor na linha do tempo perdida existe nos registros do provedor e não no banco de dados de Payments. É aqui que a reconciliação (Capítulo 13) é necessária. Compare os seus registros com os do provedor e repare as diferenças.

A ordem em que o sistema é reiniciado importa, e vale a pena escrevê-la como um procedimento.

```
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")
```

Os passos são o ponto. Pausar os dois lados primeiro impede que o serviço restaurado piore o seu estado inconsistente, e reconciliar antes de retomar segue o Capítulo 3. O reset transforma a restauração em um problema de replay, com um acréscimo.

Um Orders restaurado que relê as horas perdidas produz novas linhas de outbox, incluindo um comando `CapturePayment`, e um `command_id` novo pareceria novo para o inbox de Payments e poderia capturar duas vezes. Por isso os ids dos eventos derivados vêm do `event_id` de entrada, como na seção sobre efeitos colaterais. Um comando disparado por dois eventos, como é o caso de `CapturePayment`, toma o seu id da chave de negócio, como `order_id` mais o tipo do comando (Capítulo 7), porque qual dos dois é lido por último pode mudar no replay. O relay fica retido até que alguém verifique o que foi produzido.

Perdendo dados do broker

Um broker pode perder dados sem perder o cluster. Em uma família de log, uma réplica que estava atrasada pode ser promovida a líder, e as mensagens que ela não tinha se perdem. O nível de acknowledgment e o número mínimo de réplicas em sincronia decidem se o produtor foi informado de que tal mensagem estava segura (Capítulo 2). Alguns brokers permitem ou proíbem, por configuração, a promoção de uma réplica desatualizada, e descobriria o padrão do seu e o registraria por escrito. A retenção pode fazer o mesmo em silêncio, como mostrou o Capítulo 13.

O que você vê: posições de consumidores que saltam ou são reportadas como fora do intervalo, um topic cujo offset final é menor do que era e, depois do fato, achados de reconciliação do tipo “ausente”. O acknowledgment do broker é uma evidência com um limite, e o limite é o número de cópias que existiam naquele momento.

O que sobrevive: o outbox, se você manteve as linhas. Um relay que apaga as linhas logo depois de publicar não tem memória do que enviou. Um que mantém as linhas publicadas por um período, digamos alguns dias a título de ilustração, pode marcar como não publicadas as linhas da janela afetada e enviá-las de novo. A republicação mantém o `event_id` original, então os consumidores que receberam o evento tratam a segunda cópia como um duplicado. É o mesmo passo do reparo do Capítulo 13, aplicado

a um intervalo, e traz a mesma cautela com o fan-out: eventos mais antigos que a janela de deduplicação de um assinante precisam ser entregues ao único consumidor que os perdeu, não a todos.

Se as linhas do outbox se foram, os registros do próprio produtor são a fonte seguinte: reconstrua o envelope a partir do registro de negócio, com o `event_id` e o hash do payload armazenados, e pare para um humano onde o hash diferir. E se nenhum dos dois existe, o evento não pode ser recuperado e a reconciliação continuará relatando a falta. Nesse ponto a pergunta certa é de negócio: o fato ausente é reconstruído a partir de outro sistema, ou aceito como perdido e explicado? Projetar para que essa pergunta raramente seja feita é o que se ganha mantendo as linhas do outbox por algum tempo.

Perdendo uma região

A perda de uma região combina as duas anteriores e acrescenta um problema próprio. Suponha que a Parcellly mantenha um cluster standby em uma segunda região, alimentado por uma ferramenta de replicação que copia os topics continuamente, e que os bancos de dados replicam de forma assíncrona.

A primeira propriedade a declarar é que a **replicação assíncrona tem um RPO**, e ele é o atraso de replicação no momento da falha. Tudo o que foi escrito na região primária e ainda não foi copiado está perdido do ponto de vista do standby. Essa é a medida do que você perde, e é um número que varia minuto a minuto. Vale também criar um alerta sobre ele, contra um limite declarado, do modo como o Capítulo 13 trata a idade das mensagens: você está vigiando o seu RPO.

A segunda propriedade é que **os offsets não viajam**. O standby guarda uma cópia das mensagens, mas a cópia tem os seus próprios offsets, atribuídos quando a réplica a escreveu. A posição do último commit de um consumidor no primário não diz nada sobre o standby. Algumas ferramentas de replicação oferecem tradução de offsets; verificaria o que a sua garante, e não desenharia uma recuperação que dependa de ela ser exata.

O plano portátil é iniciar cada grupo no standby a partir do horário do evento na posição que ele havia alcançado quando o primário falhou, menos uma margem, e deixar o inbox absorver a sobreposição. É a regra da

restauração de banco de dados. Um grupo que estava 40 minutos atrasado começa 40 minutos antes. Os eventos que nunca foram replicados não estão no standby de forma alguma, e a janela do outbox, mais adiante, cuida deles.

Como as informações de commit do próprio primário se perdem com a região, o histórico da idade das mensagens precisa ser mantido fora dela. Se ele se foi, recorra ao ponto mais antigo dentro das janelas de deduplicação dos consumidores e aceite os duplicados extras. Um limite de alerta não basta, porque o alerta pode ter estado disparado na hora.

A terceira propriedade é que **duplicados no failover são certos, não possíveis**. Um consumidor que reinicia antes da sua posição verdadeira processará eventos duas vezes. Um relay que estava no meio de uma publicação quando a região falhou pode publicar uma segunda vez a partir do banco de dados que sobreviveu.

As lacunas são o outro lado: eventos aos quais o primário deu ack e que o standby nunca recebeu. Se o banco de dados do produtor sobreviveu e está mais à frente que o broker do standby, ele guarda linhas marcadas como publicadas que o standby nunca viu. Para elas, amplie a janela de republicação do outbox como no caso da perda do broker: marque como não publicadas as linhas do último intervalo de atraso de replicação, mais uma margem, e envie-as de novo. O atraso de replicação dimensiona essa janela, e somente ela. Duplicados não custam nada no consumidor. Lacunas custam exatamente aquilo que você está tentando proteger.

Se o banco de dados do produtor se perde com a região e é restaurado a partir de uma réplica mais antiga, o caso se reduz à restauração a partir de backup, com as sequences rebobinadas e os pedidos esquecidos que vêm junto.



Leia o diagrama da esquerda para a direita. Os dois passos que lidam com o desconhecido, a janela de republicação e a margem do timestamp, erram ambos na direção da duplicação. Cada passo seguinte existe para tornar a duplicação inofensiva.

Migração de brokers

A migração é um desastre que você agenda. Uma equipe passa de um broker para outro por custo, por fim de suporte, por uma oferta gerenciada ou por uma mudança no padrão de acesso, e a mudança tem os mesmos ingredientes dos desastres acima: dados de um lado, um lugar novo e vazio do outro, posições que não se transferem e um período em que dois sistemas estão vivos ao mesmo tempo. A diferença é que você tem tempo para ser cuidadoso, e nenhuma emergência o obriga a improvisar.

As ferramentas do livro fazem da migração uma mudança controlada. O motivo é que várias delas foram colocadas de propósito entre a aplicação e o broker.

O relay como único ponto de troca

Com um transactional outbox (Capítulo 4), um serviço produtor nunca fala com o broker. Ele escreve no seu próprio banco de dados, e o relay (Capítulo 5) leva as linhas ao broker. Assim, a escolha do broker fica codificada em um único processo. Mudar para onde os eventos vão significa mudar para onde o relay publica, e o código do produtor não muda. É uma mudança menor que uma espalhada por todos os serviços, um benefício colateral de um padrão que o Capítulo 4 recomenda por outro motivo.

O relay também é o proprietário do envelope. Ele pega o `event_id` da linha do outbox e o coloca na mensagem que publica. No novo broker o mesmo evento tem o mesmo `event_id`, e é essa igualdade que torna segura o resto da migração.

Dual-publish

O primeiro passo mais seguro é o relay publicar cada linha nos dois brokers. Por um tempo, todo evento existe no broker antigo, onde os consumidores de hoje o leem, e no novo, onde ninguém ainda lê.

O dual-publish é, ele próprio, um dual write, e o Capítulo 4 explica por que eles são perigosos. Aqui o risco é menor, porque a linha do outbox é a fonte da verdade e um crash deixa a linha para ser enviada de novo.

A questão de design é pelo que o relay espera. Se ele marca uma linha como publicada apenas depois que as duas publicações têm sucesso, um broker sombra inacessível deixa linhas não publicadas e interrompe o fluxo ativo de que os consumidores dependem, por causa de um broker que ninguém lê. Então acompanhe os dois destinos separadamente. A linha está publicada, no sentido do Capítulo 5, quando o broker antigo a tem, e uma segunda marca registra o novo broker. Uma passada, em ordem de `id`, alimenta o novo broker com as linhas que não têm a segunda marca, o que preserva a ordem por chave, e a limpeza espera pelos dois. O custo é mais uma coluna e mais uma passada, e em troca uma indisponibilidade do novo broker aparece como uma lacuna crescente na comparação e não como uma indisponibilidade da Parcelly.

Durante essa fase, o broker antigo continua sendo aquele em que os consumidores confiam. O novo broker é uma sombra, e a sua única função é ser comparado com o original.

Espelhamento e bridges

Uma alternativa ao dual-publish no relay é copiar mensagens entre brokers com uma ferramenta: uma ferramenta de espelhamento para log-para-log, ou uma bridge ou um conector para log-para-fila e fila-para-log. Isso não exige mudança no relay e também carrega o histórico que existia antes de a migração começar, o que o dual-publish não faz.

Duas cautelas. Um espelho ou uma bridge é um consumidor e um produtor ao mesmo tempo, então é at-least-once e produz duplicados, como todo par consumidor-produtor. E ele precisa repassar o `event_id`, no corpo da mensagem ou em um header, sem alterá-lo. Uma bridge que gera o seu próprio identificador transforma cada evento copiado em um evento novo, e o inbox não consegue reconhecer nenhum deles. Teste isso com uma amostra antes de confiar. Se um espelho preserva o particionamento, as chaves e os headers, e em que ordem entre as partições, varia conforme a ferramenta, e a resposta está na documentação dela, não neste livro.

Offsets não se traduzem

O offset de um grupo de consumidores no broker antigo não significa nada no novo. Mesmo entre dois brokers do mesmo tipo, a cópia tem a sua própria numeração. Entre uma fila e um log, não existe offset algum

de um dos lados.

A resposta é a mesma a que a seção sobre perda de região chegou: escolher um ponto de partida seguro e confiar na idempotência. Para cada grupo, escolha um timestamp tal que todo evento que o grupo ainda não havia terminado de processar no broker antigo tenha sido publicado depois dele. Tome o atraso do grupo no momento da troca, converta-o em tempo e some uma margem. Inicie o novo grupo nesse timestamp no novo broker. Ele relê eventos que o grupo antigo já tratou, e o inbox os ignora.

A janela de deduplicação se aplica aqui como se aplica a um replay: o evento mais antigo que o novo grupo vai reler precisa estar dentro da janela de todos os consumidores.

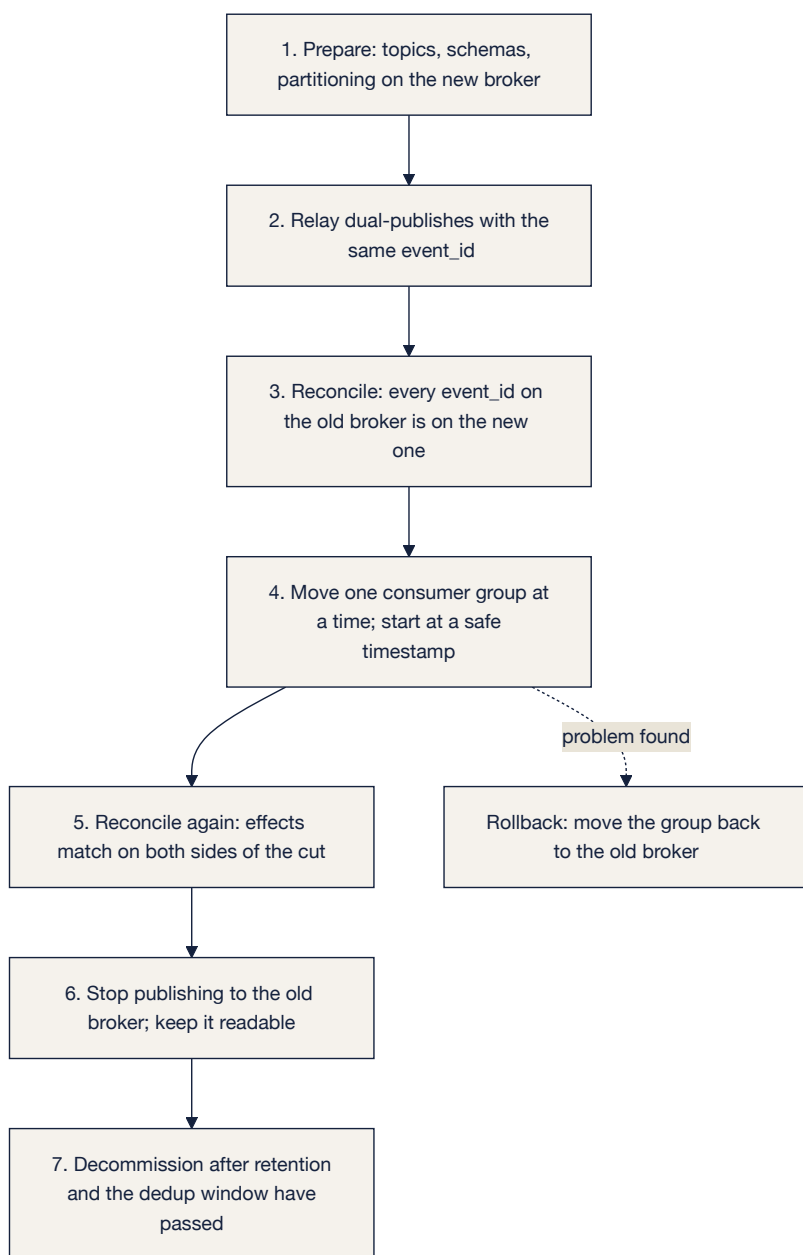
Ordenação e partições

O Capítulo 9 deu a condição para a ordem por chave: mesma chave, mesmo particionador, mesma contagem de partições. Uma migração muda pelo menos uma das três. O novo broker pode usar um particionador diferente, de modo que um mesmo `order_id` caia em uma partição onde nunca caiu, e o novo topic pode ser criado com um número diferente de partições, porque esse é o momento em que as pessoas reconsideram o número.

Duas coisas decorrem disso. Calcule a partição no relay a partir da `partition_key` armazenada, como sugeriu o Capítulo 9, de modo que um único trecho de código seja responsável pelo mapeamento, e teste que os eventos de uma chave caem em uma única partição no novo broker. E planeje o cutover de modo que nenhuma chave tenha eventos em trânsito nos dois lugares. Se um consumidor começa a ler o novo topic enquanto o topic antigo ainda guarda eventos não processados para o mesmo `order_id`, os eventos novos podem ser tratados antes dos antigos: `OrderCancelled` antes de `OrderConfirmed` de novo, a falha do Capítulo 9. Há duas formas de evitar isso. Drenar o backlog antigo até zero antes de iniciar o novo grupo, o que pausa o processamento pelo tempo da drenagem. Ou tornar os consumidores tolerantes por projeto, com eventos que carregam estado e uma versão do agregado que rejeita qualquer coisa mais antiga, como o Capítulo 9 descreveu. Drenaria primeiro em qualquer fluxo em que a ordem importe e a pausa seja suportável.

O cutover

A ordem das operações é a parte sobre a qual as pessoas perguntam: primeiro os consumidores, ou primeiro os produtores? Com o dual-publish, a pergunta em grande parte se dissolve, porque os produtores já estão escrevendo nos dois antes de um único consumidor se mover. A ordem passa a ser esta.



A reconciliação envolve a mudança dos dois lados. Antes dela, compare o conjunto de valores de `event_id` nos dois brokers ao longo de uma janela: tudo o que está no broker antigo e falta no novo é um defeito do dual-publish, encontrado enquanto ninguém depende do novo broker. Depois dela, compare os efeitos: um grupo movido às 14:00 deveria ter produzido o mesmo estado que teria produzido no broker antigo, e a reconciliação entre serviços do Capítulo 13 é o que verifica isso. Uma migração declarada concluída porque os consumidores estão saudáveis verificou apenas que nada caiu.

Mova os grupos de consumidores um de cada vez, começando pelo menos crítico, digamos Notifications, e terminando com o Ledger. Cada grupo é um passo pequeno e reversível por si só.

Rollback

Uma migração sem plano de rollback é uma porta de mão única. Com este desenho, o plano vem quase de graça, enquanto uma condição se mantém: o broker antigo continua recebendo todos os eventos. Enquanto o relay faz dual-publish, o broker antigo está completo, então o rollback significa apontar o grupo de volta para o broker antigo em um timestamp com margem. O inbox absorve os eventos que o grupo tratou no lugar novo.

A condição é o motivo de o passo seis vir tarde. Quando o relay para de escrever no broker antigo, o rollback exige que o fluxo seja invertido, com o relay escrevendo de novo no antigo e os consumidores do broker antigo começando do ponto em que ele parou. Isso ainda é possível porque o outbox guarda as linhas, e é mais lento. O rollback também é limitado pela retenção do broker antigo e pela janela de deduplicação, então só o desativa quando as duas tivessem passado desde o último evento que ele recebeu. Aposentar o broker antigo no dia em que o último consumidor se move elimina o rollback no momento em que os problemas do novo broker são mais prováveis de aparecer.

Entre fila e log

Quando a migração cruza famílias, a semântica muda, e um plano que a trata como uma mudança de endereço está errado.

Aspecto	Fila para log	Log para fila
Histórico	Ganho: o replay se torna possível (e a retenção se torna uma decisão)	Perdido: sem replay, então a recuperação depende do outbox e dos produtores
Acknowledgment	O ack por mensagem vira um commit de offset; uma mensagem lenta agora pode bloquear a sua partição (Capítulo 9)	Um commit de offset vira ack por mensagem, com visibility timeouts e reentrega
Ordenação	A ordem por grupo, onde a fila a oferece, vira ordem por partição; escolha a chave	A ordem por partição precisa ser recriada, por exemplo com grupos de mensagens onde a fila os suporta
Dead letters	O redrive é algo que você constrói ou configura; o log não tem dead-letter queue (DLQ) embutida, apenas um dead-letter topic que você cria (Capítulo 8)	Os recursos de DLQ da fila aparecem, e seus limites e sua retenção são diferentes
Atrasos e expiração	Entrega atrasada e expiração por mensagem podem não existir e exigir um substituto	Atraso e expiração podem ser nativos; a recuperação baseada em retenção deixa de existir

Cada linha é um lugar em que um comportamento de que seus consumidores dependiam pode mudar em silêncio. Escreveria a lista de comportamentos de que cada consumidor depende antes de escolher o destino e testaria os que importam, como uma poison message que não pode bloquear a sua partição, em um ensaio (Capítulo 15).

Ensaando a recuperação

Nada disso vale muito se a primeira vez que você o executa é sob pressão. Uma restauração que nunca foi feita é uma crença. Um procedimento de replay que nunca rodou com um volume realista não tem duração.

Dois ensaios compensam o seu custo. Um **teste de restauração** pega um backup, restaura-o em um ambiente descartável, verifica o que a restauração faz com o outbox, o inbox e os offsets, e então executa o procedimento de retomada e o cronometra. Um **game day** remove algo de propósito em um lugar seguro, um broker, uma zona, um banco de dados de consumidor, e faz a equipe se recuperar usando apenas o runbook. O Capítulo 15 trata de como projetar essas simulações. O motivo pertence a este capítulo: só executar a recuperação mostra quais passos você esqueceu.

Um runbook para os cenários deste capítulo precisa responder a uma lista curta de perguntas, escritas e não guardadas na cabeça de alguém. O Capítulo 16 as incorpora à sua orientação sobre runbooks.

- Qual armazenamento foi afetado, e qual voltou no tempo?
- Qual é o ponto de recuperação, e como ele foi determinado?
- Quais grupos de consumidores precisam de reset, para qual timestamp e com que margem?
- Quais efeitos precisam ser suprimidos durante o replay, e por qual mecanismo?
- Todo intervalo a ser relido está dentro da janela de deduplicação de todos os consumidores afetados?
- O que é pausado, em que ordem, e quem retoma?
- Quais reconciliações confirmam que a recuperação funcionou, e quem as lê?
- Como tudo isso é desfeito se estiver dando errado?

Trade-offs e custos

A retenção é um orçamento de recuperação. Uma retenção maior mantém disponível por mais tempo a recuperação barata, com os custos listados antes. Uma retenção menor empurra você para republicar a partir dos produtores.

Manter linhas do outbox custa armazenamento e um job de limpeza. As mesmas linhas que tornam possível uma janela de republicação são as que um desenho mais enxuto apaga imediatamente. Com change data capture (CDC) e exclusão imediata, a janela não existe (Capítulo 5). Se você quer a segurança, mantenha as linhas publicadas por um período declarado e monitore a tabela.

O replay é um instrumento rombudo. Ele relê tudo no intervalo, inclusive o que estava bem, e depende de todo consumidor ser idempotente e de todo efeito estar sob um gate. Para alguns registros conhecidos, o reparo é mais barato e mais seguro. O replay ganha o seu lugar quando o dano é amplo e ninguém tem uma lista.

A supressão de efeitos colaterais é código que precisa ser testado. Um gate que existe mas é contornado por um novo handler é pior que nenhum, porque cria confiança. O custo é um teste permanente e uma regra na revisão de código.

Uma migração com dual-publish custa o dobro. Dois brokers rodam ao mesmo tempo, o relay faz mais trabalho e acompanha dois destinos, e os duplicados no lado antigo aumentam. Em troca você obtém verificação antes de alguém depender do novo broker, e um rollback. Se o sistema tolera uma janela de manutenção, um cutover direto é mais barato: pare o relay, drene os consumidores antigos, aponte o relay para o novo broker e inicie os consumidores lá. Com um outbox, isso exige uma indisponibilidade e nenhuma perda, porque as linhas esperam. Abre-se mão da verificação e do rollback, e não o escolheria para o ledger.

Quando não construir tudo isso. Um fluxo em que uma lacuna é barata, como um feed interno de métricas, não precisa de uma janela de republicação, de um runbook nem de um game day. Ajuste a recuperação ao custo da falha e registre por escrito quais fluxos você decidiu deixar sem cobertura. A ausência deles no plano deve ser uma decisão.

Erros comuns

- **Fazer o reset de um grupo sem verificar seus efeitos colaterais.** É tentador porque o reset é um único comando e os dados estão lá. O resultado é uma segunda rodada de e-mails e uma segunda tentativa de cobrança. Decida como os efeitos são suprimidos antes de mover qualquer offset.
- **Fazer o reset de grupos demais.** Recomeçar com todos parece rigoroso. Cada grupo que sofre reset relê o histórico e carrega o risco dos seus próprios efeitos. Faça o reset apenas dos grupos que perderam algo.
- **Confiar em um header de replay.** É tentador porque uma flag na mensagem parece o sinal mais barato. Uma mensagem reproduzida é a mensagem original e não carrega sinal algum de estar sendo lida de novo. O modo pertence ao deployment do consumidor ou à camada de efeitos.

- **Fazer o replay para um consumidor com o inbox do bug.** Depois de um bug de consumidor, o inbox ignora toda mensagem como já feita. Reconstrua um estado derivado em um armazenamento novo, ou lance correções explícitas para um estado append-only.
- **Um único objetivo de recuperação para o sistema inteiro.** É tentador porque é um número só para combinar. Ele é rígido demais para as notificações e frouxo demais para o Ledger. Defina objetivos por fluxo e por armazenamento.
- **Restaurar um banco de dados e retomar de imediato.** O serviço está saudável e os dashboards estão verdes, então o tráfego é retomado. O serviço restaurado então publica estado rebobinado, reutiliza identificadores e perde eventos que os seus offsets pularam. Pause, reconcilie, faça o reset dos offsets e só então retome.
- **Reutilizar identificadores baseados em sequence depois de uma restauração.** As sequences são o padrão e nada parece errado até dois pedidos compartilharem um identificador. Uma sequence rebobinada entrega um identificador que já pertence a outro pedido. Use identificadores opacos, ou avance a sequence além do maior valor já emitido.
- **Confiar na tradução de offsets.** É tentador porque a ferramenta a anuncia e ela pouparia a aritmética dos timestamps. Uma ferramenta de replicação que copia mensagens pode ou não mapear os offsets com exatidão. Planeje uma recuperação que funcione sem ela, usando timestamps e o inbox.
- **Uma bridge que gera identificadores novos.** Uma bridge costuma ser configurada para encapsular cada mensagem em seu próprio envelope, e isso passa em um teste com um consumidor. Os eventos copiados parecem novos, então o inbox não reconhece nada e cada duplicado vira um efeito. Repasse o `event_id` sem alteração e teste.
- **Desativar o broker antigo junto com a mudança do último consumidor.** Parece que a migração terminou. Isso elimina o rollback exatamente quando ele tem mais chance de ser necessário. Espere a retenção e a janela de deduplicação passarem.
- **Nunca restaurar um backup.** O job de backup está verde. A primeira restauração então é a de verdade. Restaure em um ambiente descartável de forma periódica e cronometre.

Pontos-chave

- Quando um armazenamento é perdido, restaurado ou substituído, o que sobrevive é a semântica de falhas ao redor dele, isto é, o outbox, o inbox, o `event_id` estável, o contrato e a reconciliação, e o próprio broker é substituível.
- O replay é uma ferramenta de recuperação para a família dos logs: pare um grupo de consumidores, registre seus offsets, faça o reset para um timestamp medido a partir de onde o grupo realmente estava, com uma margem, e deixe o inbox absorver os duplicados, desde que os dados ainda estejam dentro da retenção.
- Uma mensagem reproduzida se parece com a original, então o consumidor em replay precisa conhecer o seu modo e colocar um gate nos seus efeitos, derivar os identificadores a jusante de forma determinística ou reter o seu outbox, e nunca confiar em um header.
- A janela de deduplicação faz parte de todo plano de recuperação, porque um intervalo mais antigo que a janela readmite duplicados, e um bug de consumidor transforma o inbox de proteção em obstáculo que exige uma reconstrução ou correções explícitas.
- Os objetivos de ponto e de tempo de recuperação pertencem a cada armazenamento e a cada fluxo, e um banco de dados restaurado rebobina o seu outbox, o seu inbox e as suas sequences enquanto os offsets ficam onde o broker os deixou, de modo que o procedimento de retomada pausa, reconcilia, faz o reset dos offsets e só então retoma.
- A replicação assíncrona entre regiões tem um RPO igual ao seu atraso e carrega dados, mas não os offsets dos consumidores, de modo que um failover inicia cada grupo a partir do horário do evento da sua própria posição no momento da falha, dimensiona uma janela de republicação do outbox pelo atraso de replicação e trata os duplicados como certos e as lacunas como o risco real.
- Uma migração de broker usa o relay como único ponto de troca, faz dual-publish com o mesmo `event_id`, move um grupo de consumidores de cada vez a partir de um timestamp seguro, reconcilia antes e depois e mantém o broker antigo legível até que o rollback não seja mais necessário.

Um procedimento de recuperação que nunca foi executado é uma hipótese, e o mesmo vale para o código que deveria fazer um sistema sobreviver a falhas. O Capítulo 15 trata de como testar sistemas orientados a eventos, incluindo como ensaiar as restaurações, os failovers e os replays que este capítulo descreveu.

Capítulo 15 — Como testar sistemas orientados a eventos

Imagine a equipe de Inventory na Parcelly na sexta-feira anterior a um feriado prolongado. A suíte está verde: 400 testes, alguns minutos de execução, uma cobertura sobre a qual ninguém discute. O handler de `OrderPlaced` tem um teste que monta um evento, chama o handler e verifica que o estoque diminuiu em dois. Ele passa sempre. Os testes usam um substituto em memória para o broker, que entrega cada mensagem uma vez, em ordem, imediatamente.

No sábado, um deploy reinicia os consumidores. Os handlers rodam mais devagar que de costume enquanto aquecem, e um deles ultrapassa o seu `visibility timeout`, de modo que o broker entrega a mesma mensagem `OrderPlaced` a um segundo consumidor enquanto o primeiro ainda a processa. Os dois leem o inbox, os dois não encontram nada, e os dois seguem adiante para reservar o estoque. A restrição de unicidade que deveria ter feito um deles falhar não existia no esquema real, e nenhum teste percebeu, porque os testes rodavam sobre um banco de dados fake escrito à mão, que nunca impunha restrições e nunca executava dois handlers ao mesmo tempo. A suíte nunca tinha errado em nada do que verificava. Ela nunca verificou a única coisa que importava.

Enquanto isso, um teste na suíte de Orders falha uma vez a cada 20 execuções. A equipe parou de olhar para ele. Ele espera dois segundos para um evento chegar, e às vezes o evento leva três.

As duas histórias têm a mesma raiz: os testes descrevem um sistema que não existe. Este capítulo responde a uma pergunta prática. Como testar um código cujos piores bugs só aparecem sob duplicação, atraso e reordenação, sem construir uma suíte lenta, instável ou ignorada?

O problema e por que ele é difícil

O Capítulo 1 argumentou que o happy path é aquele que se consegue testar sem esforço e aquele em que se tende a acreditar. Os capítulos 2 a 14 construíram então defesas contra falhas que o happy path nunca mostra. Defesas são código, e código que nunca é exercitado sob a falha que ele trata é uma esperança. O Capítulo 3 apresentou um modelo de falhas com um campo “verificado por” em cada entrada. Este capítulo é onde esse campo é preenchido.

Três propriedades tornam difícil esse tipo de teste.

As falhas são interleavings, não entradas. Um teste de função fornece uma entrada e verifica uma saída. O bug da reserva duplicada exige uma temporização particular: duas entregas sobrepostas no tempo, ou um commit seguido de um crash e de uma reentrega. Nenhum valor de entrada o dispara. O teste precisa controlar quando as coisas acontecem, o que significa que o harness de teste precisa ser proprietário do comportamento do broker, do relógio e dos pontos de crash.

A assincronia esconde o momento da verdade. Depois que um handler roda, o efeito pode chegar a outro serviço segundos depois. Um teste não pode fazer a asserção imediatamente, e não sabe quanto esperar. Dormir um tempo fixo está errado nas duas direções: curto demais e o teste fica instável, longo demais e a suíte fica lenta.

Os fakes mentem exatamente nos lugares que importam. A idempotência se apoia em uma restrição de unicidade. A atomicidade se apoia em uma transação. A ordenação se apoia na atribuição de partições. Um fake que pula essas coisas dá a você uma suíte verde e nenhuma informação. O custo do realismo é a velocidade, e por isso o ofício está em decidir onde pagá-lo.

Adaptando a pirâmide de testes

A pirâmide de testes clássica continua valendo: muitos testes rápidos na base, poucos lentos no topo. O que muda é para que serve cada camada e que falha cada camada é responsável por pegar.

Camada	O que roda	Real ou fake	O que detecta
Testes de handler	Um handler contra um banco de dados real	Banco de dados real, sem broker	Efeito errado, inbox ausente, atômica quebrada, tratamento ruim de duplicados e de ordem
Testes de contrato	Formatos de eventos entre produtor e consumidor	Nenhum serviço	Mudanças de esquema que quebram um consumidor
Testes de componente	Um serviço com um broker, um outbox e um relay reais	Broker real, outros serviços substituídos por stubs	Ligações: do outbox ao relay ao topic, acks, retries, dead-lettering
Testes de fluxo	Alguns serviços juntos, com uma falha injetada	Tudo real, em pequena escala	Comportamento entre serviços, como compensações e timeouts
Probes de produção	Eventos sintéticos no sistema ativo	Real	Problemas de deploy e de configuração que nenhum ambiente de pré-produção mostra

Duas escolhas da tabela merecem uma frase cada. A camada de handler usa um motor de banco de dados real, porque o inbox, a escrita no outbox e a verificação de versão dependem de restrições e transações que um fake teria de reimplementar, e reimplementá-las foi como a suíte de Inventory se deixou enganar. O broker, por outro lado, fica de fora nessa camada. Um handler é uma função de uma mensagem e de um estado para um novo estado e algumas mensagens de saída. Se você pode chamá-lo diretamente, pode testá-lo rapidamente.

O formato da pirâmide também decorre de onde as falhas vivem. A maior parte das defesas deste livro é lógica do lado do consumidor ou do lado do produtor, que a camada de handler alcança a baixo custo. Só a ligação entre as peças precisa de um broker, e ali o número de testes é pequeno, porque cada um prova uma propriedade da configuração, e não da lógica de negócio.

Testando handlers

Um teste de handler deve enunciar o contrato na linguagem do livro: dada esta mensagem e este estado, o efeito acontece uma vez e o evento de saída é escrito no outbox. Depois, o mesmo teste ganha um segundo ato, em que a mensagem chega de novo.

```
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
```

A última asserção é a que as equipes esquecem. Um duplicado que não reserva duas vezes, mas emite `StockReserved` duas vezes, ainda causa dano a jusante, e verificar o outbox o detecta.

Esse teste cobre duplicados sequenciais, que a verificação de caminho rápido no inbox trata. O caso perigoso é o do duplicado concorrente, em que dois workers passam pela verificação. Para testá-lo, o harness executa dois handlers ao mesmo tempo e usa uma barreira para segurar ambos logo antes da transação, de modo que o interleaving seja forçado e não esperado por sorte.

```
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
```

Esse teste falha se alguém remove a restrição de unicidade e passa a confiar na verificação de ler-e-depois-inserir, que é exatamente a regressão mais difícil de enxergar em uma revisão. O Capítulo 7 explicou por que a

restrição é a rede de segurança. Um teste como esse torna a explicação executável.

Os pontos de crash funcionam do mesmo modo. Se o handler chama `crashPoint("before-ack")` onde o ack iria, um teste pode armar esse ponto e provocar ali um crash simulado.

```
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
```

Leia a asserção dentro do bloco `on`. Ela verifica que o commit sobreviveu ao crash, que é metade da história. A outra metade é a reentrega que vem a seguir. Um teste de crash que verificasse apenas a segunda metade também passaria para um handler que nunca fez commit de nada.

Injetando duplicados, reordenação e atraso

Testes individuais codificam os casos em que você pensou. O modelo de falhas também contém os casos em que você não pensou, então acrescentaria uma camada de testes que perturbam um stream inteiro de eventos e verificam que o resultado não muda.

A ideia é uma propriedade de convergência. Tome um stream gravado, corretamente ordenado, de um agregado. Calcule o estado final esperado dobrando os eventos diretamente, sem broker algum. Depois entregue os

mesmos eventos pelo handler em uma sequência perturbada, com alguns embaralhados dentro de uma janela e alguns duplicados, e afirme que o estado final é igual ao esperado.

```
function testOrderViewConvergesUnderAnyDelivery(seed):
  // versions 1 to 6, true order
  events = recordedOrderStream("order-1")
  // reference, no broker, no handler
  expected = foldEvents(events)
  random = randomSource(seed)
  // a failure must be reproducible
  log("seed", seed)

  schedule = shuffleWithin(events, random, window: 3)
  schedule = duplicateSome(schedule, random, rate: 0.3)

  db = freshDatabase()
  for each message in schedule:
    handleOrderEvent(db, message)

  // assumes held events are released when a gap closes (Chapter 9)
  assert heldEventCount(db, "order-1") == 0
  assert loadOrderView(db, "order-1") == expected
```

Execute-o para algumas centenas de seeds no CI. Dois detalhes o mantêm honesto. A seed é registrada em log, de modo que uma falha na seed 4172 é uma falha que você pode reexecutar em um laptop, em vez de uma história. E o resultado de referência vem de um caminho de código diferente do handler, de modo que um bug no handler não pode aparecer também na expectativa.

O atraso é a terceira perturbação. Uma mensagem atrasada é aquela que chega depois das suas sucessoras, ou depois de um timeout ter disparado. O tempo é o que se deve controlar aqui: handlers e sagas que leem o relógio do sistema diretamente não podem ter seus timeouts testados sem esperar. Passaria o relógio como dependência e deixaria o teste avançá-lo. O Capítulo 10 listou os estados em que uma saga pode estar. Um teste de saga então vira uma tabela: para cada estado, entregue cada evento que poderia chegar, inclusive os tardios, e verifique o estado resultante e a compensação emitida.

Estado da saga	Evento que chega	Resultado esperado
AWAITING_RESULTS	PaymentAuthorized	Permanece em AWAITING_RESULTS, aguardando o estoque

Estado da saga	Evento que chega	Resultado esperado
AWAITING_RESULTS, pagamento retido	O timeout dispara	COMPENSATING; os comandos ReleaseStock e VoidAuthorization são escritos no outbox
COMPENSATING ou CANCELLED	StockReserved tardio	Nenhuma transição da saga. Um StockReserved tardio significa que Inventory reservou antes de ReleaseStock chegar até ele, então esse comando libera o estoque. Se a liberação tivesse chegado primeiro, o seu tombstone teria recusado a reserva (Capítulo 10)
COMPLETED	PaymentCaptured duplicado	Ignorado

A terceira linha é a que importa. A chegada tardia depois de um timeout é o caso que as sagas erram, porque o happy path e o caminho do timeout foram testados cada um sozinho e a colisão nunca foi. A tabela precisa ser escrita por alguém que pergunta “o que mais poderia chegar neste estado?”, e não por alguém que acompanha o diagrama do fluxo.

Nota: Se os seus testes dependem de uma sequência aleatória, guarde algumas seeds que falharam como casos de regressão fixos assim que forem encontradas. Um bug que foi reproduzido uma vez deve continuar reproduzido.

Testes de contrato e verificações de compatibilidade

Os testes de handler montam seus próprios eventos, o que significa que testam o consumidor contra a ideia que o consumidor tem do produtor. Quando o produtor muda um campo, nada na suíte do consumidor percebe. O Capítulo 12 descreveu regras de compatibilidade e um schema registry que as impõe. Isso cobre a estrutura. Os testes de contrato cobrem a parte que o registry não enxerga: aquilo de que cada consumidor de fato depende.

Em um **contrato dirigido pelo consumidor**, cada consumidor registra por escrito os campos e tipos que lê de um evento, e o pipeline do produtor verifica se o seu esquema ainda os fornece. A direção importa. O produtor

não tem como saber quais campos sustentam carga, e o consumidor tem. Um contrato é, portanto, um pequeno arquivo de propriedade do consumidor e guardado onde o CI do produtor possa lê-lo.

```
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)
```

São necessárias duas metades. O pipeline do produtor executa `verifyContracts` contra todos os contratos registrados. O pipeline do consumidor decodifica cada evento de amostra dos seus contratos com o seu desserializador real e executa o handler sobre ele. Juntas, as duas detectam um produtor que deixa de fornecer um campo e um consumidor que passa a ler um campo que nunca declarou. A segunda detecção só vale se a amostra for construída a partir do contrato e contiver apenas os campos declarados; uma amostra copiada do evento completo do produtor permite que o consumidor leia um campo não declarado e mesmo assim passe.

As amostras devem cobrir todas as versões ainda na retenção, como recomendou o Capítulo 12. O replay lê o que estiver lá, e um consumidor que trata apenas o formato mais novo falha no primeiro evento antigo. É também aqui que a verificação de compatibilidade do registry pertence ao CI: uma mudança de esquema do produtor deve quebrar o build antes de poder ser registrada, e não depois de chegar ao topic.

Seja honesto quanto ao limite. Um contrato prova que `amount_minor` existe e é um inteiro. Não prova que ele ainda significa “o valor em unidades menores”. Mudar o significado sem mudar a estrutura passa em todos os testes de contrato, e só um revisor humano, ou um teste comportamental no consumidor, percebe. Não trataria contratos verdes como uma afirmação sobre significado.

Asserções sobre consistência eventual

Um teste de ponta a ponta publica algo e depois faz uma asserção sobre um efeito em outro lugar. O efeito chega com o tempo, então a asserção precisa de uma forma que tolere isso. A forma padrão é um polling com deadline.

```
function eventually(check, timeout, interval):
    deadline = now() + timeout
    while true:
        observation = check()
        if observation.satisfied:
            return
        if now() >= deadline:
            // report what was last seen, not only that time ran out
            raise AssertionError(observation.description)
        sleep(interval)
```

A estrutura importa mais do que parece. A verificação roda mais uma vez depois que o deadline é alcançado, de modo que uma condição satisfeita durante o último sleep ainda passa, o intervalo é curto em relação ao timeout, e a mensagem de falha carrega a última observação. Uma mensagem que diz “timed out” não diz nada. Uma que diz “order-1 stuck at AWAITING_RESULTS, 1 held event” diz onde olhar.

Os testes assíncronos instáveis têm uma lista curta de causas, e a maioria é corrigível na causa.

- **Sleeps fixos.** Substitua por `eventually`. O timeout vira um teto e o teste termina assim que a condição se cumpre.
- **Timeouts ajustados para um laptop rápido.** Defina o teto com folga, já que ele não custa nada quando a condição é atendida depressa, e mantenha-o separado das expectativas de desempenho. Um orçamento de latência é outro teste, com outro ambiente.

- **Estado compartilhado entre testes.** Uma mensagem esquecida em um topic compartilhado faz o teste seguinte ver um evento que ele não publicou. Dê a cada teste o seu próprio correlation id e filtre por ele, ou o seu próprio topic ou fila.
- **Ordem presumida onde nenhuma é prometida.** Se um teste faz uma asserção sobre a ordem de dois eventos de partições diferentes, ele afirma algo que o broker nunca prometeu (Capítulo 9).
- **Afirmar que nada aconteceu.** “Nenhum segundo e-mail foi enviado” não pode ser transformado em sucesso por polling, já que esperar mais poderia ainda mostrar um. Prefira levar o sistema à quiescência e só então fazer a asserção. A quiescência é uma propriedade do caminho inteiro, não de um consumidor: o outbox está vazio, o relay está ocioso, o atraso de todo consumidor envolvido é zero, nenhum retry ou timer está agendado, e as filas de retry e de atraso estão vazias. Só então não resta nada que possa produzir o efeito, inclusive um salto a jusante cuja mensagem ainda está em trânsito.

Para o último caso, o harness precisa de um modo de dizer “execute até que nada esteja pendente”. Em um teste de handler ou de componente, o harness pode bombear o consumidor até a fila ser drenada e o relógio injetado não ter mais timers a disparar. Isso é mais rápido e mais determinístico que esperar pelo tempo de relógio, e é o principal motivo de manter a maioria dos testes comportamentais na camada de handler e usar tempo real apenas onde a ligação entre as peças é o objeto do teste.

Injeção de falhas e chaos

Tudo até aqui testa o código isoladamente. A pergunta seguinte é se o sistema montado se comporta como o modelo de falhas afirma. A injeção de falhas coloca uma falha em um ambiente em execução de propósito e verifica a resposta do sistema.

Comece pelo modelo de falhas do Capítulo 3 e escolha entradas, não ferramentas. Para cada uma, escreva a hipótese nos termos da própria entrada: “se um consumidor é morto entre o commit e o ack, a reentrega não muda nada”. Depois injete: mate o processo naquele momento, pause o relay, atrase as respostas do broker, corte a conexão entre um serviço e o seu banco de dados, reinicie um nó do broker no meio de uma publicação.

Em seguida, verifique os invariantes, não os logs: o estoque bate com as reservas, o ledger tem uma entrada por captura, nenhuma linha do outbox está travada.

Acrescente simulações de recuperação a essa lista, porque uma defesa que nunca foi exercitada é uma esperança. Um backup que nunca foi restaurado é o exemplo clássico. Restaure um em um ambiente descartável e verifique o que uma restauração faz com o outbox, o inbox e os offsets dos consumidores, o que o Capítulo 14 trata: linhas do outbox ressuscitadas, linhas do inbox rebobinadas de modo que os duplicados deixam de ser reconhecidos, offsets que já não conferem. Faça o failover para um cluster standby, ou ensaie-o, e compare o tempo de recuperação e os dados perdidos com os objetivos de recuperação escolhidos para esse fluxo no Capítulo 14. E teste o alcançar o ponto atual: pause um consumidor por tempo suficiente para formar um backlog, retome-o enquanto o tráfego continua chegando e verifique que o backlog se esvazia antes do deadline do fluxo e antes de a retenção expirar. Isso verifica a taxa de drenagem do Capítulo 13 e os limites do trabalho em andamento do Capítulo 6.

Executaria isso em camadas de raio de impacto crescente. Testes de fluxo em um ambiente dedicado injetam falhas de forma agendada no CI, pequenos e repetíveis. Um game day periódico em um ambiente maior as injeta como exercício de equipe, em parte para encontrar bugs e em parte para descobrir se alguém percebe. Os alertas do Capítulo 13 estão tanto sob teste quanto o código. Se você matou um consumidor e nenhum alerta disparou, encontrou um bug no monitoramento.

O chaos em produção é um próximo passo possível, não um ponto de partida. Só faz sentido quando o sistema já mostrou que sobrevive à mesma falha em pré-produção, quando o raio de impacto pode ser limitado e quando alguém pode interromper o experimento instantaneamente. Uma equipe sem essas condições não deve começar por aí, e não me envergonharia de ficar nos game days por muito tempo. O valor de uma prática de injeção de falhas está no ciclo de hipótese, injeção e verificação, não no quão dramática é a injeção.

Aviso: Uma execução de injeção de falhas que compartilha um broker, um banco de dados ou uma conta de provedor de pagamento com o tráfego real é uma indisponibilidade com plano. Isole o ambiente, ou restrinja a injeção a tráfego sintético, antes de desligar qualquer coisa.

Replay como teste de regressão

Um log de eventos é uma gravação do que realmente aconteceu, e o replay, a releitura deliberada do histórico, o transforma em uma suíte de regressão. Tome uma fatia de eventos reais, passe-a pela nova versão de um consumidor em um ambiente descartável e compare o estado derivado com o que a versão antiga produziu. As diferenças são intencionais, como em uma correção de bug, ou regressões. O replay também tem um uso operacional, o de recuperar-se depois de uma lacuna ou de uma restauração relendo o histórico, e o Capítulo 14 trata disso. Esta seção trata apenas de usá-lo para testar.

Isso pega coisas que fixtures escritas à mão nunca pegarão: formatos de evento de três versões de esquema atrás, sequências que ninguém imaginou, um campo que estava sempre presente nos testes e é nulo em uma pequena fração do histórico. É também o que mais se aproxima de um teste da afirmação do Capítulo 12 de que os eventos antigos ainda são decodificados.

Um procedimento viável tem cinco passos.

1. **Selecione uma fatia.** Uma janela de tempo, ou o histórico completo de uma amostra de agregados. Uma amostra que inclua todas as versões de esquema, mais incidentes conhecidos, vale mais que uma aleatória.
2. **Aplique mascaramento.** Os eventos reproduzidos contêm dados de compradores. Anonimize antes que saiam da sua fronteira original, ou mantenha o ambiente descartável sob os mesmos controles da produção.
3. **Desative os efeitos e os eventos emitidos.** O consumidor em replay não pode cobrar cartões, enviar e-mails nem chamar transportadoras. Os efeitos vão para stubs que registram o que lhes foi pedido, e os re-

gistros dos stubs fazem parte da saída. O seu outbox e o seu relay devem publicar em um broker descartável, ou ser capturados, e nunca apontar para topics de produção, ou o replay republica o histórico para consumidores ativos.

4. **Execute as versões antiga e nova sobre a mesma fatia.** Compare o estado derivado e os eventos emitidos, agregado por agregado.
5. **Classifique cada diferença.** Intencional, inofensiva ou uma regressão. Uma diferença que ninguém sabe explicar é uma regressão até que se prove o contrário.

O replay se aplica à família dos logs, onde o histórico ainda está lá. Em uma fila clássica, as mensagens consumidas se foram. O equivalente é uma amostra capturada: um tap que copia uma fração das mensagens, anonimizadas, para um armazenamento sob o seu controle. É uma ferramenta mais fraca, porque a captura é tão boa quanto a amostragem, mas serve ao mesmo propósito.

O replay também tem um limite de custo. Históricos grandes levam tempo e armazenamento reais, e as ferramentas de comparação são algo que a sua equipe precisa construir e manter funcionando. Reservaria o replay do histórico completo para mudanças que afetam a forma como os eventos são interpretados, e usaria uma fatia menor nos releases de rotina.

Verificando em produção

Nenhum ambiente de pré-produção reproduz o formato do tráfego, os dados, a configuração e as esquisitices da produção. Algumas falhas, como uma permissão ausente em um topic depois de um deploy, um secret desatualizado ou uma fila mal roteada, só aparecem no sistema real. A resposta não é desistir dos testes de pré-produção, e sim acrescentar uma última camada que roda onde as falhas estão.

Um **evento sintético** é um tráfego deliberadamente injetado e explicitamente rotulado, que percorre o pipeline real e é verificado no final. Para a Parcelly, é um pedido feito a cada poucos minutos por um probe, fluindo de `OrderPlaced` até `ShipmentScheduled`.

```

function runOrderProbe():
    probeId = newId()
    // recorded first, so a crash of the probe itself leaves a trail
    recordProbe(probeId, startedAt: now())
    try:
        // goes through the real API, outbox, relay and broker; the
flag
        // travels in a header and is copied like correlation_id
        orderId = placeSyntheticOrder(probeId, headers: { synthetic:
"true" })
        eventually(
            check: flowReached(orderId, "ShipmentScheduled"),
            timeout: probeBudget,
            interval: probeInterval)
    finally:
        // runs on success, on timeout and on error alike
        cancelSyntheticOrder(probeId)

```

O rótulo é o mecanismo de segurança, e é um header `synthetic` dedicado, não um prefixo em `correlation_id`. Um identificador que também controla comportamento é uma armadilha, e o Capítulo 13 pede um formato uniforme de id, `order-81724`, para que a busca por correlação funcione em todo lugar. O preço é que um salto que esquece de copiar o header transforma um probe em tráfego real sem nenhum erro. Todo serviço o copia com o mesmo helper que copia `correlation_id`, o probe também usa uma conta de comprador sintético dedicada, que Payments e Shipping reconhecem de forma independente, e um teste remove o header de propósito e verifica que nenhuma cobrança ou envio real resulta.

Cada serviço reconhece a flag e muda as suas bordas, não a sua lógica central: Payments fala com o sandbox do provedor, Shipping pula a entrega à transportadora, Notifications suprime o envio, e o Ledger lança em uma conta de teste que as finanças excluem dos relatórios. Esses ramos são código que roda apenas para probes, então um erro neles é um bug que só a produção expõe. Inventory precisa do mesmo tratamento, ou cada probe reserva estoque real: reserve contra um SKU sintético dedicado, ou pule a reserva para pedidos rotulados. Testaria os próprios ramos na camada de handler, e testaria o probe quebrando algo de propósito e verificando que ele falha.

A limpeza tem mais uma armadilha. Um probe que estoura o timeout não necessariamente deixou de terminar; seus eventos ainda podem estar em trânsito, e um `StockReserved` ou `PaymentAuthorized` tardio pode chegar

depois do cancelamento. É por isso que o probe é registrado logo no início e todo serviço se orienta pela flag `synthetic`: um sweep periódico pode encontrar probes mais antigos que o orçamento e cancelar o que quer que tenham deixado, e o cancelamento é ele próprio idempotente, de modo que executá-lo duas vezes é inofensivo.

Um probe também precisa de um orçamento, e o orçamento precisa de um responsável. Um probe que falha envia um page para alguém, então o orçamento precisa ser generoso o bastante para evitar falsos alarmes e apertado o bastante para ter significado. O Capítulo 13 trata da latência de ponta a ponta e de como criar alertas sobre ela. O probe é a fonte mais barata dessa medição, já que exercita o caminho inteiro, inclusive as partes em que nenhum pedido real passou na última hora.

A verificação em produção inclui os jobs de reconciliação do Capítulo 13. Uma execução de reconciliação é um teste de invariantes sobre dados reais, executado continuamente, e detecta as falhas que ninguém pensou em simular. Conto essa execução como a última camada da pirâmide.

Trade-offs e custos

O realismo custa tempo. Bancos de dados reais, brokers reais e falhas injetadas deixam os testes mais lentos. A suíte que era rápida era rápida porque pulava o que importava. Orçaria as camadas lentas de forma explícita: testes de handler em segundos, testes de componente em alguns minutos, testes de fluxo e injeção de falhas de forma agendada e não a cada commit.

Os testes de contrato acrescentam coordenação. Cada consumidor é proprietário de arquivos que vivem no pipeline do produtor, e alguém precisa mantê-los atualizados. Para um sistema com dois serviços e uma equipe, um teste compartilhado do esquema pode bastar. O teste de contrato começa a compensar quando as equipes de produtor e consumidor fazem o deploy de forma independente, e rende pouco antes disso.

As sequências aleatórias acrescentam trabalho de investigação. Uma seed que falha é um quebra-cabeça e um defeito real ao mesmo tempo. As equipes que não gostam disso acabam reduzindo o número de seeds até o teste não encontrar mais nada. A cura é registrar as seeds em log e manter regressões fixas, não abandonar a técnica.

Chaos e probes têm um raio de impacto. Ambos tocam sistemas ativos ou compartilhados. Quando não usá-los: quando o sistema ainda não sobrevive à falha em isolamento, quando ninguém consegue interromper a execução imediatamente, ou quando o tráfego rotulado não pode ser mantido separado de dinheiro real e de compradores reais.

O replay é caro de montar. A infraestrutura de mascaramento, de stubs de efeitos e de comparação é um projeto à parte. Um sistema pequeno com um histórico curto ganha mais com testes de handler e pontos de crash. Não sei onde fica o seu ponto de equilíbrio, já que ele depende da frequência com que os consumidores mudam e do custo de uma interpretação errada do histórico, mas uma equipe que já teve um incidente causado por um evento antigo inesperado costuma conseguir adivinhá-lo.

Erros comuns

- **Testar contra um fake de broker que nunca se comporta mal.** É tentador porque é rápido e os primeiros testes passam. Faça o fake duplicar, atrasar e reordenar, ou coloque a falha nos testes de handler com chamadas explícitas, como neste capítulo.
- **Simular as restrições do banco de dados com um fake.** Um fake escrito à mão não impõe unicidade, isolamento nem atomicidade, então os testes passam existindo ou não a defesa. Use o motor real, em um contêiner ou em uma instância descartável, em todo teste que dependa de uma restrição.
- **Dormir por um tempo fixo.** Parece a forma mais simples de esperar. Faça polling da condição com um teto e informe a última observação na falha.
- **Nunca testar o crash entre o commit e o ack.** Parece exótico, mas acontece em todo deploy. Acrescente um ponto de crash em cada handler e um teste que o arme.
- **Testar apenas o estado final, e não o outbox.** Um duplicado que não muda o estado ainda pode emitir um segundo evento. Faça asserções sobre o que foi escrito no outbox.
- **Testes de contrato que verificam a presença, mas não a obrigatoriedade.** Um campo que é opcional no esquema do produtor pode estar ausente. Faça o contrato dizer “required” e verifique isso.

- **Testes aleatórios sem seeds.** Uma falha que não pode ser reproduzida acaba apagada. Registre a seed em log e guarde as seeds que falharam como casos fixos.
- **Confiar em um backup que nunca foi restaurado.** O job de backup informa sucesso, então a defesa parece testada. Restaure-o em um ambiente descartável de forma periódica e verifique depois o outbox, o inbox e os offsets.
- **Executar probes contra dinheiro real.** Tráfego sintético que chega ao provedor de pagamento ativo ou a uma transportadora real não é um probe. Roteie o tráfego rotulado para sandboxes na borda e teste esse roteamento.

Pontos-chave

- Testes que supõem um broker entregando cada mensagem uma vez, em ordem, testam um sistema que não existe, então toda defesa deste livro precisa de um teste que aplique a falha contra a qual ela defende.
- Os testes de handler pertencem a um motor de banco de dados real, porque o inbox, o outbox e as verificações de versão dependem de restrições e transações que os fakes não reproduzem.
- Duplicados, reordenação e atraso são testados melhor como uma propriedade de convergência sobre um stream perturbado, com a seed registrada em log para que qualquer falha possa ser reproduzida.
- Os contratos dirigidos pelo consumidor detectam quebras estruturais antes do deploy, mas não enxergam uma mudança de significado, de modo que a revisão humana do que um campo deve dizer continua fazendo parte do processo.
- Substitua os sleeps fixos por polling contra uma condição e um teto, faça asserções sobre a quiescência e não sobre o silêncio, e injete o relógio para que os timeouts possam ser testados sem esperar.
- A injeção de falhas deve partir das entradas do modelo de falhas e verificar invariantes depois, aumentando o raio de impacto apenas à medida que o sistema prova que sobrevive a cada etapa, e deve incluir simulações de recuperação para restaurações, failovers e o alcançar o ponto atual.

- O replay do histórico real e os probes sintéticos em produção detectam o que fixtures escritas à mão e ambientes de pré-produção não conseguem, e precisam de mascaramento, stubs de efeitos e tráfego rotulado para serem seguros.

Um sistema que foi testado dessa maneira merece um pouco mais de confiança, embora nunca completa, e isso deixa a questão do que uma equipe deve verificar antes de confiar em um design. O Capítulo 16 reúne o livro em um checklist de confiabilidade e termina com aquilo que o checklist não consegue fazer.

Capítulo 16 — Um checklist de confiabilidade e reflexões finais

Imagine uma revisão de design na Parcelly, numa quinta-feira tranquila. A equipe está prestes a adicionar um novo fluxo: quem vende poderá oferecer reembolsos parciais, e o Payments vai emitir um novo evento para isso. O diagrama está limpo. Alguém desenhou as setas, nomeou o evento e listou os consumidores. A reunião está marcada para 30 minutos, e tudo indica que vai terminar em 20.

Então a pessoa que atua como tech lead faz uma pergunta simples: “O que acontece se o reembolso for emitido e o evento nunca for publicado?” A sala fica em silêncio, não porque ninguém conheça o padrão, mas porque ninguém havia perguntado. O outbox existe em outra parte do código. O consumidor idempotente também existe em outra parte. Se esse novo fluxo os usa é outra questão, e o diagrama não diz.

É assim que a confiabilidade costuma se erodir. A equipe conhece os padrões. O que se perde é o hábito de verificar, a cada novo fluxo, se cada padrão está de fato no lugar. Por isso a pergunta deste último capítulo é prática: o que você perguntaria, fluxo a fluxo, para descobrir se o sistema consegue cumprir suas promessas num dia ruim, e quem precisa ser responsável pelas respostas?

O problema e por que ele é difícil

A confiabilidade é uma propriedade das junções, e as junções não são visíveis em nenhum arquivo isolado. Um produtor pode estar correto, um consumidor pode estar correto, e o fluxo entre eles ainda pode perder eventos, porque a passagem de um para o outro nunca foi examinada. O Capítulo 1 apresentou esse argumento no início do livro, e os capítulos seguintes trabalharam as junções uma a uma.

A dificuldade agora é que o conhecimento está espalhado por 15 capítulos. Ninguém guarda tanto assim na cabeça durante uma revisão. Sob pressão de prazo, as perguntas que são puladas são aquelas cuja resposta é “funciona até o dia em que deixa de funcionar”: a janela de deduplicação, o responsável pela dead-letter queue, a verificação de compatibilidade, o alerta que dispara pela idade e não pela contagem.

Um checklist é a resposta de sempre, e tem limites conhecidos. Ele não pensa. Quem não entende um item ainda pode marcá-lo, e uma lista preenchida passa então uma falsa confiança. Usaria a lista abaixo como roteiro de conversa, não como um gate que um formulário consiga atravessar. Cada item aponta para o capítulo que explica por que ele está ali, de modo que quem discordar possa encontrar o raciocínio e discutir a premissa.

Trate cada item como uma pergunta sobre um fluxo específico, como o fluxo de pedidos da Parcelly, e não sobre o sistema em geral. “Usamos um outbox” não é uma resposta. “O Orders grava o `OrderPlaced` no seu outbox na mesma transação da linha do pedido” é.

O método por trás da lista

A lista não é uma ideia nova. É a tese do livro aplicada a uma revisão: sistemas confiáveis orientados a eventos são construídos em torno da **semântica de falhas**, não dos message brokers: para uma determinada junção, o que pode falhar, como você observaria a falha e qual garantia sobrevive a ela.

Isso dá três perguntas, e cada item abaixo é uma delas feita sobre uma junção. Um item que nomeia uma falha pergunta o que pode falhar. Um item que nomeia uma métrica, um alerta ou um teste pergunta como você observaria a falha. Um item que nomeia uma configuração, uma restrição ou um padrão pergunta qual garantia sobrevive a ela. Uma junção cujos itens respondem apenas a uma das três está pela metade, e uma junção sem resposta para a segunda pergunta vai falhar sem avisar ninguém.

O checklist

A lista está agrupada pelo lugar onde a falha vive. A coluna da direita indica o capítulo a reler quando a resposta for “não” ou “não tenho certeza”.

Produtor

Verificação	Capítulo
A mudança de estado e o evento são gravados em uma única transação local, sem publicação direta no caminho de negócio.	4
A linha do outbox carrega o envelope completo, e o <code>event_id</code> é criado no momento da gravação e nunca regenerado.	4
A equipe registrou por escrito que a publicação é at-least-once e que os duplicados são o preço.	4
O relay publica primeiro e marca depois, e seleciona por estado, não por um cursor de id crescente.	5
Vários relays, se houver, reivindicam linhas com segurança e mantêm as linhas de um mesmo agregado em um único worker.	5
A idade da linha não publicada mais antiga é medida, e apenas linhas publicadas são removidas.	5

Se uma equipe puder responder a só uma pergunta numa revisão, gostaria que fosse a primeira, porque toda garantia posterior pressupõe que o evento exista. Quando um fluxo não precisa de outbox, porque seus passos podem compartilhar uma transação local, a equipe deve dizer isso e dizer por quê (Capítulo 1).

Broker

Verificação	Capítulo
Para cada evento importante, a equipe registrou o que “publicado” significa: as configurações de replicação e de flush por trás de um ack do produtor.	2, 3
O ack ou o commit de offset acontece depois do efeito, de modo que a semântica é at-least-once e não um at-most-once acidental.	2
Os visibility timeouts, os timeouts de sessão e o tempo de retry dentro do handler são definidos em conjunto, para que um trabalho lento não seja reentregue enquanto ainda está em execução.	2, 6
A chave de partição identifica o agregado, e a equipe sabe qual ordenação a sua família de brokers oferece.	9
Todo produtor, em qualquer linguagem, usa o mesmo partitioner e a mesma codificação de chave, ou o relay calcula a partição por conta própria.	9
A retenção supera a mais longa interrupção realista somada ao tempo para percebê-la.	3, 13

Verificação	Capítulo
Os consumidores são encerrados de forma graciosa no deploy, para que o rebalanceamento não seja uma fonte rotineira de duplicados.	3

A maioria desses itens são configurações, não código. Eles ficam nos valores padrão até que um incidente mostre o que os padrões eram, e a essa altura é difícil rastreá-los.

Consumidor

Verificação	Capítulo
A linha do inbox e a mudança de negócio fazem commit em uma única transação local, com uma restrição de unicidade na chave.	7
A janela de deduplicação supera a retenção, o redrive e o replay planejado, e uma restrição de chave de negócio cobre a tabela de domínio para além dela.	7
Os efeitos colaterais externos carregam uma chave de idempotência determinística, derivada do trabalho e não da tentativa.	7
As falhas são classificadas como transitórias, permanentes ou desconhecidas, e os retries usam exponential backoff com jitter, um deadline e um budget compartilhado.	6
O trabalho em andamento tem um teto, a concorrência diminui quando a dependência fica lenta, e o consumidor pausa em vez de falhar, de modo que a sobrecarga apareça como atraso; um limite na entrada rejeita ou adia o trabalho novo com um sinal claro para quem chamou.	3, 6
Uma poison message termina em uma DLQ com corpo, headers, posição de origem, erro, número de tentativas e versão do consumidor, publicada antes de o original receber o ack.	8
Um consumidor preso em um único registro é detectado, pelo atraso sem throughput e por um contador de falhas por posição, e não fica em loop até que alguém perceba.	8, 13
O redrive preserva o <code>event_id</code> original, roda em lotes com limite de taxa e tem uma chave de pausa.	8
A ordem por agregado é protegida por lanes com chave, verificações de versão e hold-back buffers que têm timeouts.	9
Todo passo do fluxo anterior ao pivot tem uma compensação idempotente, e toda espera tem um deadline.	10

Este é o grupo mais longo porque a maioria dos padrões do livro termina aqui. Se ele parece pesado, a impressão é correta: a entrega at-least-once coloca o fardo no consumidor.

Contrato

Verificação	Ca- pítu- lo
Cada evento enuncia um fato, é nomeado no passado e tem um único serviço responsável.	11
O evento de integração é um contrato público deliberado, mapeado a partir do evento de domínio, sem dados pessoais nem secrets no payload.	11
Um comando é nomeado no imperativo, tem um único destinatário, carrega um <code>command_id</code> , tem um caminho de resposta para que uma recusa chegue a quem enviou, e nunca é chamado de evento; quem envia espera com um deadline próprio.	10, 11
O <code>correlation_id</code> é copiado ao longo do fluxo e o <code>causation_id</code> aponta para o gatilho direto.	11
Os tipos de transmissão do envelope, como timestamps, identificadores e valores monetários, são fixados para toda linguagem que o lê ou escreve.	11
Toda mudança é classificada como compatível ou incompatível, e a compatibilidade é verificada no pipeline de deploy.	12
Uma mudança de significado é revisada por uma pessoa, porque nenhuma ferramenta a detecta.	12
Os consumidores ignoram campos que não usam e toleram valores que nunca viram, um salto que repassa uma mensagem preserva os campos desconhecidos, e um evento cujo <code>schema_version</code> eles não conhecem é rejeitado ou estacionado.	12
As mudanças incompatíveis migram como uma sequência de passos compatíveis, com os rollbacks planejados no mesmo documento, e os consumidores que ainda leem a versão antiga são conhecidos.	11, 12

Operações

Verificação	Ca- pítu- lo
O trace context e o <code>correlation_id</code> sobrevivem a todo salto assíncrono, de modo que a história de um pedido possa ser reconstruída a partir do seu identificador.	13
O atraso do consumidor, a idade da mensagem, a idade do outbox, a taxa de retri- es e as chegadas à DLQ são medidos, e os alertas disparam pela idade e pela taxa.	13

Verificação	Ca- pítu- lo
Jobs de reconciliação comparam a fonte da verdade com o estado derivado em intervalos regulares, e os reparos são revisados antes de rodar.	13
Os invariantes de negócio, como “todo pagamento capturado tem um lançamento no ledger”, são verificados diretamente.	13
Os handlers são testados com duplicados, reordenação e atraso, e os testes de contrato rodam no CI.	15
Existem injeção de falhas e testes de regressão baseados em replay para os fluxos que mais importam.	15
Cada armazenamento tem um objetivo de ponto de recuperação (RPO) e um objetivo de tempo de recuperação (RTO) declarados, e a equipe sabe qual armazenamento volta no tempo em uma restauração.	14
Um procedimento de replay nomeia os grupos de consumidores, o timestamp e a sua margem, os efeitos colaterais que ele suprime e a janela de deduplicação dentro da qual todo intervalo reprocessado precisa caber.	7, 14
Um plano de migração de broker usa o relay como único ponto de comutação, faz dual-publish com deduplicação no consumidor e tem passos de cutover e de rollback.	14
As restaurações e os game days foram ensaiados, e os runbooks abaixo foram escritos a partir do que o ensaio mostrou.	14, 15
Um modelo de falhas escrito nomeia responsáveis, testes e datas de revisão.	3

Perguntas para uma revisão de design

O checklist verifica um design que existe. Estas perguntas ajudam a dar forma a um que ainda não existe. Faria essas perguntas nesta ordem, porque cada resposta restringe a seguinte.

- 1. Qual é o fato, e quem é o responsável por ele?** Se você não consegue nomear o evento no passado e nomear um único serviço responsável, o design ainda não está pronto para um broker (Capítulo 11).
- 2. Isto precisa mesmo de um evento?** Se os passos podem compartilhar uma transação local, ou se quem chama precisa de uma resposta imediata, uma chamada síncrona pode ser a ferramenta melhor (Capítulo 1).
- 3. O que acontece entre o commit e a publicação?** Peça a janela de falha, não o padrão. “Publicamos logo depois do commit” é uma perda

esperando por um deploy (Capítulo 4).

4. **O que acontece quando esta mensagem chega duas vezes, e o que acontece quando chega tarde?** São falhas diferentes, com correções diferentes (capítulos 7 e 9).
5. **Qual é o limite do “no fim”?** Nomeie uma duração, um alerta e um estado visível ao usuário para a espera (Capítulo 1).
6. **O que acontece quando esta mensagem nunca puder ter sucesso?** Pergunte onde ela vai parar, quem a vê e como ela volta (Capítulo 8).
7. **O que acontece quando o consumidor, ou a dependência por trás dele, é mais lento que o produtor?** Pergunte o que tem teto, o que pausa e qual trabalho espera, e quem é responsável por essa escolha (capítulos 3 e 6).
8. **Se um passo posterior falhar, o que desfaz os anteriores?** Nomeie a compensação e o pivot depois do qual nada mais é desfeito (Capítulo 10).
9. **Como vamos mudar este contrato daqui a um ano?** Se ninguém pensou nisso, a resposta será improvisada durante um incidente (Capítulo 12).
10. **Como vamos saber que está quebrado, e como vamos saber que foi consertado?** Detecção e verificação são perguntas separadas (capítulos 13 e 15).
11. **Se um armazenamento, o broker ou uma região for perdido, quanto perdemos e quanto tempo levamos para voltar?** Peça um RPO e um RTO por armazenamento, e o replay ou a restauração que vem depois (Capítulo 14).
12. **Quem recebe o page, e o que essa pessoa faz primeiro?** Se a resposta é um nome, você tem um responsável. Se é “a equipe”, talvez não tenha.

Algumas dessas perguntas não têm uma resposta limpa num dado dia, e tudo bem. “Ainda não sabemos, e eis como vamos descobrir” é um resultado melhor do que uma resposta confiante que ninguém verificou.

O lado organizacional

Todo padrão deste livro tem uma metade humana. Um outbox precisa de alguém que observe a idade da sua linha mais antiga. Uma DLQ precisa de alguém que a leia. Um schema registry precisa de alguém que decida o que “compatível” significa. Quando falta a metade humana, o padrão continua existindo no diagrama e deixa de cumprir o seu papel.

Responsabilidade

Registraria por escrito três tipos de responsabilidade.

- **Responsabilidade sobre o evento.** Todo evento tem um único serviço e uma única equipe responsáveis, que documentam o que o evento significa e quem o consome (Capítulo 11).
- **Responsabilidade operacional.** Toda DLQ, parking lot, relatório de reconciliação e alerta tem um responsável nomeado. Um alerta que não pertence a ninguém é uma notificação, e as notificações são ignoradas.
- **Fluxos entre serviços.** O fluxo de pedidos da Parcelly atravessa seis serviços. Alguma equipe precisa ser responsável pelo fluxo como um todo, incluindo seus timeouts e seus workflows travados, mesmo quando nenhum serviço isolado o é (Capítulo 10).

Runbooks

Um runbook é o que você escreve supondo que quem o lê está cansado e não vê o sistema há seis meses. Para cada alerta, diga o que ele significa, o que olhar primeiro, quais ações são seguras, quais não são e para quem ligar se os primeiros passos não ajudarem.

Alguns runbooks estão ligados a um alerta e outros a um evento que acontece tão raramente que ninguém da equipe o fez recentemente. O Capítulo 14 acrescenta o segundo tipo. Sua seção sobre ensaiar a recuperação lista oito perguntas que um runbook de recuperação precisa responder, por escrito e não guardadas na cabeça de alguém:

1. Qual armazenamento foi afetado, e qual voltou no tempo?
2. Qual é o ponto de recuperação, e como ele foi determinado?

3. Quais grupos de consumidores precisam ser reiniciados, para qual timestamp e com que margem?
4. Quais efeitos precisam ser suprimidos durante o replay, e por qual mecanismo?
5. Todo intervalo a ser relido está dentro da janela de deduplicação de cada consumidor afetado?
6. O que é pausado, em que ordem, e quem retoma?
7. Quais reconciliações confirmam que a recuperação funcionou, e quem as lê?
8. Como tudo isso é desfeito se estiver dando errado?

A tabela lista os eventos para os quais escreveria um runbook. A última coluna indica quais das oito perguntas esperaria que cada um respondesse. É um ponto de partida, e o sistema de cada equipe vai acrescentar perguntas próprias.

Evento	A ação perigosa	Perguntas
Um consumidor preso em uma poison message (Capítulo 8)	Pular a posição sem registro disso, ou aumentar o limite de retries até a mensagem passar	6, 7, 8
Um consumidor cujo atraso está se aproximando da retenção (capítulos 13 e 14)	Deixar as mensagens expirarem sem serem lidas, ou avançar o reset sem que ninguém tenha decidido isso	2, 3, 5, 7
Reset de offset e replay (Capítulo 14)	Fazer o replay com os efeitos colaterais ativos, ou além da janela de deduplicação	3, 4, 5, 6, 8
Redrive (Capítulo 8)	Devolver uma DLQ inteira de uma só vez a uma dependência que ainda está em dificuldade	5, 6, 8
Reparo e backfill (Capítulo 13)	Rodar um script sem revisão contra o estado de produção	4, 7, 8
Restauração de um banco de dados a partir de um backup (Capítulo 14)	Retomar os consumidores sem reiniciar seus offsets, ou reutilizar identificadores que a sequência revertida volta a fornecer	1, 2, 3, 4, 5, 6, 7, 8
Perda de dados do broker ou failover (Capítulo 14)	Retomar grupos em posições que já não significam o que significavam	1, 2, 3, 5, 6, 7
Evacuação de região (Capítulo 14)	Promover um standby sem saber quanto ele estava atrasado	1, 2, 3, 6, 7, 8

Evento	A ação perigosa	Perguntas
Cutover da migração de broker (Capítulo 14)	Mover os consumidores antes de os pares dual-published serem deduplicados, ou sem caminho de volta	3, 5, 6, 8

Dois destes merecem cuidado particular. O redrive vem primeiro, porque um redrive descuidado transforma uma falha contida em uma enxurrada (Capítulo 8). O reparo e o backfill vêm em segundo, porque um script que corrige estado derivado é uma mudança em produção e merece uma revisão, um dry run e um registro do que fez (Capítulo 13).

On-call

O on-call de um sistema orientado a eventos difere do on-call de um serviço de requisição e resposta em um ponto que importa: a falha costuma ser silenciosa. Um serviço síncrono devolve erros e um gráfico fica vermelho. Um sistema orientado a eventos pode não devolver nada, porque a mensagem está parada numa fila, esperando. Os alertas sobre idade, atraso e trabalho não processado são o sinal principal, e a pessoa on-call precisa de acesso suficiente para olhar dentro da fila, da DLQ e do outbox sem esperar por uma permissão.

Nota: Pratique o reparo antes de precisar dele. Uma equipe que fez o redrive de uma DLQ de teste uma vez, durante o dia, com um colega observando, vai lidar com a DLQ real de forma diferente de uma equipe que apenas leu o runbook. O Capítulo 14 faz o mesmo argumento para as restaurações e os game days, e o Capítulo 15 trata de como projetar as simulações.

Trade-offs e custos

Um checklist tem custos, e o primeiro é a atenção. Uma lista de 43 itens aplicada a todo fluxo é longa demais, e uma equipe que aplica todos eles a um fluxo de notificação de baixo risco vai concluir que o problema é o checklist. Ponderaria a lista pelo custo da falha. Um fluxo que movimenta dinheiro ou altera estoque merece a lista inteira. Um e-mail de lembrete merece idempotência e um responsável pela DLQ, e talvez pouco mais.

O segundo custo é a falsa garantia. Uma lista preenchida diz que alguém considerou cada item, não que o sistema está correto. Os capítulos 13, 14 e 15 existem porque designs bem pensados ainda falham de maneiras que seus autores não imaginaram. Trate o checklist como a camada de defesa mais barata, não como a última.

O terceiro custo é a deriva. Uma lista escrita uma vez se torna o registro do que a equipe acreditava num determinado momento. Revisite-a depois de cada incidente e apague os itens que deixaram de merecer o seu lugar. Uma lista curta que as pessoas usam vale mais do que uma longa que as pessoas assinam.

Quando não usar: não use o checklist para substituir o julgamento, e não o exija para protótipos descartáveis. Se um fluxo será apagado no mês que vem, o custo da lista completa supera o custo da falha. Diga isso explicitamente e anote a data até a qual o protótipo ou morre ou recebe uma revisão adequada.

Erros comuns

- **Tratar o checklist como um gate.** Um gate pode ser auditado contando marcações, e essa é a tentação. O resultado é um formulário preenchido e um design sem discussão. Use os itens como perguntas e registre as respostas que surpreenderam alguém.
- **Verificar padrões em vez de fluxos.** “Temos um outbox” parece progresso. Pergunte se este fluxo o usa, e nomeie a tabela e a transação.
- **Aplicar tudo em todo lugar.** Parece rigoroso, depois leva ao cansaço e ao abandono. Pondere a lista pelo custo da falha e diga quais itens você pula e por quê.
- **Deixar os itens humanos para depois.** Responsáveis, runbooks e datas de revisão parecem menos urgentes do que o código, e são a primeira coisa que falta quando um alerta dispara de madrugada. Atribua-os na própria revisão de design.
- **Nunca revisitar as respostas.** Um fluxo que passou na revisão há um ano ganhou consumidores, versões e pessoas desde então. Coloque uma data de revisão nas respostas, como o Capítulo 3 sugere para o modelo de falhas.

- **Acreditar que a lista está completa.** O seu sistema tem modos de falha que este livro não nomeia. Acrescente-os à sua própria lista quando os encontrar.

Pontos-chave

- A confiabilidade se erode mais por perguntas puladas do que por padrões desconhecidos, de modo que um conjunto fixo de perguntas feitas a cada novo fluxo protege um design melhor do que a experiência presente na sala.
- Todo item do checklist é uma das três perguntas do livro, o que pode falhar, como você observaria a falha e qual garantia sobrevive a ela, feita sobre uma junção, de modo que cada item deve ser respondido para um fluxo específico e não para o sistema em geral.
- Pondere a lista pelo custo da falha: fluxos que movimentam dinheiro ou estoque merecem todos os itens, e fluxos de baixo risco merecem os poucos que evitam efeitos duplicados e perdas silenciosas.
- Todo padrão deste livro tem uma metade humana, de modo que responsáveis, runbooks, destinatários de alertas e datas de revisão pertencem à mesma revisão de design que o código.
- As falhas em sistemas orientados a eventos são silenciosas, de modo que o on-call depende de alertas sobre idade, atraso e trabalho não processado, e os eventos raros, uma restauração, um replay, a perda de uma região e uma migração de broker, exigem runbooks que tenham sido de fato executados.
- Um checklist preenchido mostra que alguém pensou em cada item, não que o sistema está correto, e é por isso que a reconciliação e os testes existem como camadas adicionais.
- A confiabilidade não é um recurso que se entrega uma vez, mas uma disciplina que se continua praticando, em perguntas pequenas, respostas escritas e reparos ensaiados antes de serem necessários.

A Parcelly nunca se torna confiável do modo como um recurso fica pronto. Ela se torna mais confiável cada vez que alguém pergunta o que acontece quando a mensagem chega duas vezes, anota a resposta e dá a ela um responsável. Nenhum broker oferece isso, e nenhuma biblioteca o entrega pronto. É um hábito de atenção, mantido por pessoas que sabem que todo

sistema que constroem será um dia maltratado por uma rede, um deploy ou uma pessoa cansada, e que decidiram projetar para esse dia sem drama. É isso que pediria que você levasse deste livro: não um conjunto de padrões, mas o hábito de fazer a próxima pergunta.

Apêndice: Glossário e referência rápida de padrões

Este apêndice tem duas partes. A Parte A é um glossário dos termos que o livro usa, em ordem alfabética, com o capítulo em que cada termo é definido ou usado de forma mais completa. A Parte B é uma referência rápida dos padrões, escrita como uma tabela que você pode percorrer com os olhos durante uma revisão de design.

Nenhuma das partes substitui os capítulos. Uma definição aqui é um lembrete, e uma linha da tabela é uma indicação. O raciocínio, os custos e os casos em que um padrão não deve ser usado estão no capítulo citado na última coluna.

A. Glossário

ack (acknowledgment). O sinal do consumidor ao broker de que uma mensagem foi tratada e pode ser removida da fila, ou de que a posição do consumidor pode avançar. A forma negativa é o **nack**. Capítulo 2.

adiado. A classe de retry para uma falha que não diz nada sobre a mensagem, porque se sabe que a dependência está fora do ar, como quando um circuit breaker está aberto. Uma mensagem adiada não consome nenhuma tentativa e nunca vai para a dead letter por isso. O consumidor pausa e a mensagem espera. Capítulo 6.

agregado. A fronteira de consistência que é proprietária de uma chave de partição. Os eventos de um agregado são ordenados entre si; os eventos de agregados diferentes não são. Capítulo 9.

at-least-once. Uma semântica de entrega em que uma mensagem é entregue uma ou mais vezes, de modo que duplicados são possíveis. Ela vale enquanto a mensagem ainda está retida e o consumidor dá o ack apenas

depois do efeito. Um consumidor que se atrasa além da retenção ainda perde mensagens em silêncio. É o padrão do livro. O Capítulo 2 a define; o Capítulo 3 trata da perda.

at-most-once. Uma semântica de entrega em que uma mensagem é entregue zero ou uma vez, de modo que uma mensagem pode ser perdida, mas nunca é entregue duas vezes. Capítulo 2.

atraso do consumidor. Quanto um consumidor está atrás da cabeça de uma partição ou fila, medido em mensagens ou em tempo. O Capítulo 2 o define; os capítulos 3 e 13 o usam.

backfill. Reparo ou reconstrução em massa que alimenta um consumidor ativo com um grande conjunto de eventos, executado em pequenos lotes com limite de taxa. Capítulo 13.

backpressure. Limitar o trabalho em andamento em cada etapa para que a carga permaneça finita. Em um sistema mediado por broker, ele aparece como limites na entrada de um consumidor, como o `prefetch` ou os registros por `poll`, e como atraso que o broker absorve. A pressão só chega a um produtor por meio do controle de fluxo do broker ou de um `throttling` explícito. O Capítulo 3 o situa no modelo de falhas; o Capítulo 6 apresenta os mecanismos.

bloqueio semântico. Um flag de status nos dados que indica que uma saga está trabalhando em um registro, e que o restante do código é escrito para respeitar. Não é um bloqueio de banco de dados. Capítulo 10.

broker (message broker). A infraestrutura que transporta mensagens entre serviços. Duas famílias aparecem no livro: baseada em log e baseada em fila. Capítulo 2.

causation_id. O campo do envelope que aponta para o evento ou comando que causou diretamente este, enquanto o `correlation_id` nomeia o fluxo inteiro. Capítulo 11.

change data capture (CDC). Ler o log de transações de um banco de dados para transformar mudanças com `commit` em mensagens. No contexto do outbox, um relay construído sobre CDC lê as linhas do outbox a partir do log em vez de fazer `polling` na tabela. Capítulo 5.

chave de idempotência. A chave que um consumidor ou uma API usa para reconhecer trabalho repetido. Para o consumo de eventos, normalmente é o `event_id`. Capítulo 7.

chave de negócio. Um identificador derivado do significado do trabalho, como o id de um pagamento mais uma operação, e não do envelope. Protege contra o mesmo trabalho chegando em duas mensagens diferentes. Capítulo 7.

chave de partição. O identificador do agregado usado para rotear eventos a uma partição e, assim, preservar a ordem por agregado. Capítulo 9.

circuit breaker. Uma proteção em torno de uma dependência que interrompe as chamadas depois que as falhas ultrapassam um limiar e, após uma espera, deixa passar chamadas de teste. Capítulo 6.

comando. Um pedido para fazer algo, nomeado no imperativo, como `CapturePayment`. Um comando é endereçado a um único serviço e pode ser recusado. Não é um evento. Capítulo 11.

commit de offset. O registro do consumidor, em um broker baseado em log, de que ele processou tudo o que vem antes de uma posição. Substitui o ack por mensagem. Capítulo 2.

compatibilidade futura. Um consumidor que usa o esquema antigo consegue ler eventos escritos com o esquema novo: leitores antigos, dados novos. Capítulo 12.

compatibilidade retroativa. Um consumidor que usa o esquema novo consegue ler eventos escritos com o esquema antigo: leitores novos, dados antigos. Capítulo 12.

compensação. O passo que desfaz semanticamente um passo anterior de uma saga, como liberar estoque ou anular uma autorização. O livro não a chama de rollback. Capítulo 10.

consistência eventual. A propriedade de que réplicas ou serviços convergem para o mesmo estado se as atualizações cessarem, sem um limite para o tempo que isso leva. Capítulo 1.

consumidor. Um processo que lê mensagens de um topic ou de uma fila e age sobre elas. Capítulo 2.

consumidor idempotente. Um consumidor cujo processamento repetido da mesma mensagem tem o mesmo efeito que processá-la uma vez. Capítulo 7.

consumidores concorrentes. Vários consumidores que compartilham uma fila, de modo que cada entrega vai para um deles. Uma mensagem reentregue pode chegar a outro consumidor. É a contrapartida, na família de filas, de um grupo de consumidores. Capítulo 2.

contrato orientado ao consumidor. Um arquivo de propriedade de um consumidor que lista os campos e os tipos que ele lê de um evento, verificado no pipeline do produtor. Capítulo 15.

correlation_id. O campo do envelope compartilhado por todos os eventos de um mesmo fluxo de negócio. Seu complemento, o `causation_id`, aponta para o evento ou comando que causou diretamente este. Capítulo 11.

cutover. O passo de uma migração de broker em que um grupo de consumidores passa a ler do novo broker, dado um grupo de cada vez depois que os dois brokers foram reconciliados e com um rollback planejado. Capítulo 14.

dead letter. Uma mensagem que foi posta de lado porque seu processamento falhou. Como modificador, o termo leva hífen: `dead-letter queue`. Capítulo 8.

dead-letter queue (DLQ). Um destino separado onde um consumidor ou broker coloca as mensagens das quais desistiu. Em um broker baseado em log, a mesma ideia é um **dead-letter topic**. Capítulo 8.

deadline. O momento depois do qual um resultado deixa de ser útil. Os retries precisam parar antes dele. Capítulo 6.

dual write. Atualizar dois sistemas, como um banco de dados e um broker, sem uma transação compartilhada. Capítulo 4.

dual-publish. Publicar o mesmo fato em duas formas ou para dois destinos durante uma migração: os dois campos ou os dois topics durante uma mudança de esquema (Capítulo 12), os dois brokers durante uma migra-

ção de broker (Capítulo 14). Em uma migração de broker, o relay mantém um segundo marcador para o novo destino, de modo que o broker antigo permaneça completo e o novo possa ser reconciliado com ele.

duplicado. Uma mensagem processada ou entregue mais de uma vez. Capítulo 2.

envelope. O conjunto padrão de campos que viaja com todo evento, como `event_id`, `event_type`, `schema_version`, `occurred_at`, `correlation_id` e `causation_id`, separado do payload. Capítulo 11.

espelhamento. Copiar mensagens entre brokers com uma ferramenta: uma ferramenta de espelhamento entre dois logs, ou uma bridge ou um connector entre um log e uma fila. Também carrega o histórico anterior a uma migração e pode manter um cluster standby alimentado, e os offsets na cópia são próprios dela. Capítulo 14.

event_id. O campo do envelope que identifica um evento. É criado quando a linha do outbox é gravada e nunca é regenerado, e é a chave de idempotência usual para o consumo. Capítulos 4 e 7.

evento. Uma mensagem cujo conteúdo é um fato que já aconteceu, nomeada no passado. Capítulo 11.

evento de integração. Um evento publicado como um contrato público deliberado, mapeado a partir de um evento de domínio e livre de dados pessoais e de secrets. Capítulo 11.

evento sintético. Tráfego rotulado injetado no pipeline ativo por uma sonda e verificado no fim do fluxo. Capítulo 15.

evolução de esquemas. Mudar a forma dos eventos ao longo do tempo sem quebrar os leitores e escritores que existem. Capítulo 12.

exactly-once. Usado no livro apenas como um efeito qualificado: entrega at-least-once mais um consumidor idempotente, que o livro também chama de effectively-once. Não é uma garantia de entrega. O processamento exactly-once de um broker é real dentro da fronteira desse broker e termina nela. Capítulo 2.

expand and contract. Um padrão de migração que transforma uma mudança incompatível em vários passos compatíveis: suportar as duas formas, migrar todos os leitores e então remover a forma antiga. Capítulo 12.

failover. Uma troca planejada para outra cópia de uma partição de broker, de um banco de dados ou de um cluster quando o ativo falha. Pode perder o que ainda não havia sido replicado. Capítulos 3 e 14.

fat event. Um evento que carrega estado suficiente para que os consumidores ajam sem chamar o produtor de volta. O oposto é o **thin event**. Capítulo 11.

fila. Uma estrutura de broker em que cada mensagem é consumida e recebe o ack individualmente. Capítulo 2.

grupo de consumidores. Um conjunto de consumidores que compartilham as partições de um topic, de modo que cada partição é lida por um membro de cada vez. Capítulo 2.

head-of-line blocking. Um registro que um consumidor não quer pular bloqueia todos os registros atrás dele na mesma partição ou no mesmo stream. Capítulo 8.

hold-back buffer. Uma tabela durável em que um consumidor guarda um evento cujo antecessor não chegou, e o libera quando a lacuna se fecha ou um timeout expira. Capítulo 9.

idade da mensagem. O tempo desde que foi produzida a mensagem mais antiga que ainda está esperando, comparado com um deadline de negócio e com o período de retenção. Capítulo 13.

inbox. A tabela de ids de mensagens processadas no lado do consumidor, gravada na mesma transação dos efeitos do handler. Capítulo 7.

janela de deduplicação. O período durante o qual um consumidor se lembra dos ids processados. Uma mensagem cuja linha do inbox foi apagada pode ser processada de novo. Capítulo 7.

jitter. Aleatoriedade adicionada ao atraso de um retry para que muitos clientes não façam retry em uníssono. “Exponential backoff com jitter” é a expressão padrão. Capítulo 6.

journal de eventos. Um registro append-only de envelopes de eventos, sem payloads, mantido por um serviço ao lado do seu inbox para que seja possível responder a perguntas depois que o broker já esqueceu os eventos. Capítulo 13.

link (trace). Uma referência de um span de consumidor ao trace context do produtor, usada no lugar de uma relação pai-filho quando a entrega é tardia, repetida ou em lote. Capítulo 13.

log de eventos. A sequência durável e append-only de eventos. Capítulo 2.

mensagem. Uma unidade de dados enviada por um broker: um evento, um comando ou um documento. Capítulo 11.

modelo de falhas. A lista escrita das junções de um sistema e, para cada uma, o que pode falhar, como a falha seria observada e qual garantia sobrevive a ela, com responsáveis, testes e datas de revisão. Capítulo 3.

objetivo de ponto de recuperação (RPO). Quantos dados você pode se permitir perder, medido em tempo. Um backup a cada seis horas significa um RPO de até seis horas, e cada armazenamento tem o seu. Capítulo 14.

objetivo de tempo de recuperação (RTO). Por quanto tempo você pode ficar sem o serviço. Para uma projeção reprocessada por replay, é um cálculo de escoamento. Capítulo 14.

offset. A posição de um consumidor em uma partição de um log de eventos. Capítulo 2.

outbox. A tabela gravada na mesma transação local da mudança de estado, a partir da qual um relay publica eventos. O termo completo é **transactional outbox**. Capítulo 4.

outbox do consumidor. Um outbox gravado por um consumidor que publica uma mensagem de continuação, na mesma transação da sua linha do inbox e da sua mudança de estado. Uma falha antes do commit não deixa nada para trás, e uma falha depois dele é absorvida pelo inbox na reentrega. O Capítulo 7 introduz a regra; o Capítulo 10 a aplica aos passos de uma saga.

parking lot. Um lugar de espera para dead letters que precisam de uma decisão humana, distinto da própria DLQ. Capítulo 8.

partição. Uma unidade de um topic baseado em log que é ordenada e lida por um consumidor de um grupo de cada vez. Capítulos 2 e 9.

passo de pivot. O ponto de uma saga depois do qual o fluxo de trabalho deixa de compensar e só avança. Capítulo 10.

período de carência. A idade que um registro precisa atingir antes que um job de reconciliação o compare, para que eventos ainda em trânsito não sejam relatados como discrepâncias. Capítulo 13.

poison message. Uma mensagem que falha toda vez que é processada. Em um broker baseado em log, uma que derruba o processo deixa o consumidor preso em uma única posição, e por isso é detectada pelo atraso sem throughput e por um contador por posição. Capítulo 8.

produtor. Um processo ou serviço que publica mensagens, e o responsável pelos eventos que emite. Capítulo 11.

rebalanceamento. A reatribuição de partições quando membros entram ou saem de um grupo de consumidores, ou quando um membro é considerado sem resposta. Dois consumidores podem trabalhar brevemente na mesma partição. Capítulo 3.

reconciliação. Comparar as visões de verdade de dois sistemas e reparar as diferenças. Capítulo 13.

redrive. Devolver dead letters ao consumidor que falhou com elas, à sua própria entrada e nunca ao topic compartilhado de onde vieram. A chave e o `event_id` são mantidos, a contagem de tentativas recomeça e o `redrive_count` registra a rodada. Capítulo 8.

reentrega. O broker entregando uma mensagem de novo, por exemplo depois de um ack perdido ou de um visibility timeout expirado. O replay é outra coisa. Capítulo 2.

relay. O processo que move as linhas do outbox para o broker. Capítulo 5.

replay. Reler deliberadamente o histórico de um log de eventos. O Capítulo 2 o define; os capítulos 7, 12, 13, 14 e 15 o usam.

reset de offset. Mover a posição de commit de um grupo de consumidores, em geral para trás, até um timestamp com uma margem, de modo que o histórico seja lido de novo. Os offsets pertencem a um único cluster e não se traduzem para outro. Capítulo 14.

retenção. Por quanto tempo um broker guarda uma mensagem, tenha ela sido lida ou não: um limite de tempo ou de tamanho em um log, uma expiração em uma fila. Ela limita o replay, o redrive e por quanto tempo um consumidor atrasado pode ficar fora do ar. O Capítulo 2 a define; os capítulos 3, 13 e 14 a usam.

retry budget. Um limite para os retries como fração do tráfego total, e não por chamada. Capítulo 6.

retry storm. Uma onda de retries que acrescenta carga a uma dependência em dificuldade e a mantém fora do ar. Capítulo 6.

revisão do registry. O identificador que um schema registry atribui a cada esquema registrado. Ele avança a cada mudança registrada, inclusive as aditivas, e é um número diferente da **versão do esquema** e da **versão do agregado**. Capítulo 12.

saga. Um fluxo de trabalho de longa duração composto de transações locais com compensações, orquestrado por um coordenador ou coreografado por meio de eventos. Capítulo 10.

schema registry. Um serviço que armazena esquemas, atribui identificadores e verifica um esquema proposto contra os anteriores. Capítulo 12.

semântica de falhas. Para uma determinada junção, o que pode falhar, como você observaria a falha e qual garantia sobrevive a ela. É a tese do livro. O Prefácio a enuncia, o Capítulo 3 a aplica como o modelo de falhas e o Capítulo 16 a transforma no checklist de revisão.

tolerant reader. Um consumidor que pega de um evento apenas os campos de que precisa e ignora o resto. Capítulo 12.

tombstone. A chave, a última versão e um marcador de exclusão que um consumidor mantém depois de apagar a linha de um agregado, para que um evento tardio não possa recriá-la. Capítulo 9.

topic. Um stream nomeado de mensagens em qualquer uma das famílias de broker. Capítulo 2.

trace context. O trace id e o span id que situam uma unidade de trabalho em um trace distribuído, capturados na linha do outbox e enviados nos headers da mensagem para que o span do consumidor possa se ligar ao do produtor. Capítulo 13.

upcaster. Código que converte um evento escrito sob um esquema antigo para o atual antes que a lógica de negócio o veja. Capítulo 12.

verificação de invariante. Uma consulta agendada para uma regra que o negócio considera verdadeira, como todo pagamento capturado ter exatamente um lançamento no Ledger. Cada linha devolvida é uma discrepância. Capítulo 13.

verificação de versão. Um consumidor comparando a versão de um evento com a última versão que aplicou para aquele agregado, para ignorar eventos obsoletos e detectar lacunas. Capítulo 9.

versão do agregado. Um contador das mudanças de um agregado, armazenado na linha do outbox e carregado no payload, como o `order_version` do Orders. Ele ordena os eventos de um agregado e não diz nada sobre a forma do payload. É um número diferente da **versão do esquema** e da **revisão do registry**. O Capítulo 4 o define; o Capítulo 9 o usa.

versão do esquema. O inteiro `schema_version` no envelope, que o Capítulo 12 também chama de versão incompatível. Ele avança apenas para uma mudança que um leitor antigo não consegue tratar, e é um número diferente da **revisão do registry** e da **versão do agregado**. Capítulo 12.

visibility timeout. Em um broker baseado em fila, o tempo que uma mensagem entregue permanece oculta antes de ser reentregue se não recebeu o ack. Capítulo 2.

B. Referência rápida de padrões

Cada linha nomeia a falha que o padrão resolve, nos termos do livro, e o que ele custa a você. Um padrão move uma falha para um lugar onde você consegue tratá-la; ele não remove a falha. O Capítulo 16 transforma estas linhas em um checklist de revisão.

Padrão	Falha tratada	Custos	Capítulo
Transactional outbox	Dual write entre um banco de dados e um broker	Uma tabela extra, um relay para operar, duplicados empurrados para os consumidores	4

Padrão	Falha tratada	Custos	Ca- pí- tulo
Relay com polling	Mover as linhas do outbox para o broker com pouca infraestrutura	Atraso do polling, carga no banco de dados, projeto custoso do cursor	5
Change data capture	Publicar as linhas do outbox a partir do log do banco de dados com baixo atraso	Peso operacional, failover do connector, retenção do log a gerenciar	5
Exponential backoff com jitter	Retries que sobrecarregam uma dependência em recuperação	Latência adicional, ajuste dos atrasos e dos limites de tentativas	6
Retry budget	Retries que amplificam uma interrupção	Algumas chamadas falham rápido e talvez tivessem dado certo	6
Backpressure e controle de fluxo	Uma dependência ou um consumidor alimentado com mais trabalho do que consegue absorver	Atraso, consumidores pausados e trabalho rejeitado, uma decisão de negócio sobre o que espera	6
Circuit breaker	Chamadas repetidas a uma dependência que está falhando	Estado a ajustar; chamadas rejeitadas enquanto o breaker está aberto	6
Retry topic ou fila de atraso	Atrasos de retry longos demais para esperar dentro do handler	Mais topics, mais partes móveis, ordenação mais difícil	6
Consumidor idempotente com inbox	Duplicados da entrega at-least-once	Gravações de tabela a cada mensagem, retenção a gerenciar	7
Chave de negócio	O mesmo trabalho chegando em mensagens diferentes	Projeto da chave para cada operação, e restrições de unicidade	7
Chave de idempotência repassada adiante	Efeitos colaterais externos duplicados	Exige suporte do serviço seguinte	7
Dead-letter queue	Poison messages presas em um loop de reentrega	Um acúmulo que precisa de um responsável, alarmes e um caminho de reparo	8
Parking lot	Dead letters que precisam de uma decisão humana	Trabalho manual e uma regra clara de responsabilidade	8
Redrive	Recuperar dead letters depois que a causa foi corrigida	Duplicados, reordenação e carga sobre o fluxo principal	8

Padrão	Falha tratada	Custos	Ca- pí- tulo
Outbox do consumidor	Um consumidor que muda estado e publica uma mensagem de continuação, perdendo uma das duas coisas	Um outbox no lado do consumidor e um relay para operar	7, 10
Chave de partição por agregado	Eventos fora de ordem dentro de um mesmo agregado	Partições quentes, limites ao paralelismo	9
Verificação de versão no consumidor	Eventos obsoletos ou fora de ordem sobrescrevendo um estado mais novo	Um campo de versão e uma decisão para cada lacuna	9
Hold-back buffer	Lacunas em uma sequência que precisa ser aplicada em ordem	Memória, timeouts e a lacuna que nunca se fecha	9
Saga (orquestrada ou coreografada)	Um fluxo de negócio que abrange serviços sem uma única transação	Estados, timeouts e compensações a projetar e operar	10
Compensação	Desfazer passos concluídos de uma saga depois de uma falha posterior	Alguns passos não podem ser desfeitos; o passo de pivot marca o limite	10
Bloqueio semântico	Mudanças concorrentes em um registro em que uma saga está trabalhando	Operações bloqueadas enquanto a saga roda	10
Tipos de transmissão fixados por campo entre linguagens	O mesmo campo do envelope lido de maneira diferente por dois runtimes	Disciplina em cada linguagem e testes que atravessam linguagens	11
Envelope de evento padrão	Eventos que as ferramentas e os consumidores não conseguem rastrear nem rotear	Disciplina em todos os produtores	11
Thin event ou fat event	Escolher entre chamadas de volta e estado carregado	Thin: acoplamento no momento da leitura. Fat: contrato maior e dados copiados	11
Schema registry com verificações de compatibilidade	Produtores quebrando consumidores em silêncio	Um serviço para operar e um processo a seguir	12
Tolerant reader e upcaster	Leitores quebrando com campos novos ou versões antigas	Código de conversão mantido para cada versão histórica	12

Padrão	Falha tratada	Custos	Ca- pí- tulo
Expand and contract	Uma mudança incompatível que não pode ser evitada	Vários deploys, dual-publish, uma migração mais longa	12
Trace context e correlation id	Perder a história de um fluxo ao longo de saltos assíncronos	Propagação em todo produtor e consumidor	13
Reconciliação e reparo	Divergência silenciosa entre um sistema de origem e o estado derivado	Jobs de comparação, ferramentas de reparo e revisão dos resultados	13
Journal de eventos	Perguntas sobre eventos que o broker já esqueceu	Uma tabela gravada a cada mensagem e retenção a gerenciar	13
Backfill	Estado derivado que precisa ser reconstruído ou preenchido a partir de dados existentes	Lotes com limite de taxa e uma chave de pausa, e a carga sobre um consumidor ativo	13
Verificações de invariantes	Quebras no nível do negócio que nenhuma métrica técnica mostra	Definir os invariantes e ser responsável pelos alertas	13
Replay com reset de offset	Um consumidor que estava errado, ou um armazenamento que voltou no tempo, e precisa do histórico de novo	Uma janela de deduplicação que cubra o intervalo, efeitos colaterais suprimidos e a retenção como limite	14
Objetivos de ponto e de tempo de recuperação por armazenamento	Desastres tratados por improviso, sem perda ou indisponibilidade acordadas	Um número a medir e ensaiar para cada armazenamento	14
Replicação entre clusters para recuperação	Perda de um cluster de brokers ou de uma região	Um standby para operar, um atraso de replicação que é o RPO e offsets que não se traduzem	14
Dual-publish e cutover	Migrar para outro broker sem perder eventos nem duplicar efeitos	Um segundo marcador no outbox, reconciliação e uma migração mais lenta	14
Testes de contrato orientados ao consumidor	Deriva de contrato entre equipes	Manutenção dos testes e acordo sobre quem os executa	15
Testes de handlers e de perturbação	Handlers nunca exercitados sob duplicados, atraso ou reordenação	Infraestrutura de testes com banco de dados real e suítes mais lentas	15

Padrão	Falha tratada	Custos	Ca- pí- tulo
Injeção de falhas	Entradas do modelo de falhas nunca exercitadas em um sistema em execução	Um ambiente dedicado, um raio de impacto a delimitar	15
Teste de regressão baseado em replay	Mudanças que quebram o comportamento sobre o histórico real	Dados de amostra curados e anonimizados a manter	15
Eventos sintéticos em produção	Falhas silenciosas no caminho ativo	Dados de teste que precisam ser isolados dos dados reais do negócio	15

Como ler a tabela

Muitas linhas vêm em pares: um padrão que corrige uma falha e um segundo padrão que corrige a falha que o primeiro introduziu. O outbox cria duplicados, então o consumidor precisa de um inbox. Os retries criam carga, então precisam de um budget. Uma dead-letter queue cria um acúmulo, então precisa de um responsável e de um procedimento de redrive.

Essa cadeia é a forma do livro. Se você adota uma linha e ignora a sua parceira, a falha mudou de lugar, mas não desapareceu. Na minha opinião, a tabela é mais útil lida da direita para a esquerda: comece pela falha que o seu sistema tem hoje, encontre a linha que a nomeia e leia a coluna de custos antes de ler o capítulo.

SOBRE O LIVRO

Sistemas confiáveis orientados a eventos são construídos em torno da semântica de falhas, não dos message brokers.

Em cada junção, em que um componente passa trabalho a outro, o livro faz as mesmas três perguntas: o que pode falhar, como você observaria isso e qual garantia sobrevive a essa falha. A entrega at-least-once é tratada como um fato, e não como um defeito; por isso os handlers são idempotentes, e os duplicados, as mensagens perdidas, os retries e a chegada fora de ordem são previstos no projeto, em vez de esperar que desapareçam. Acompanhando a Parcelly, o livro percorre o transactional outbox, os relays e o change data capture, os retries, a idempotência, as dead letters, a ordenação, as sagas, a evolução de esquemas, a observabilidade e a recuperação, cada um com o custo que traz.

SOBRE O AUTOR

Muhammet Şafak desenvolve software desde 2008. Durante muitos anos atuou ativamente nas áreas de saúde, finanças e telecomunicações, e defende que o desenvolvimento deve se concentrar no problema, não na tecnologia. Desde 2024 projeta e desenvolve sistemas de alto tráfego escaláveis de ponta a ponta.

Faz mentoria de equipes de desenvolvimento de software de pequeno e médio porte, conduz projetos de código aberto e registra por escrito as decisões, com suas razões e seu custo.

Sistemas confiáveis não são construídos com peças confiáveis. São construídos em torno das peças que falham.

I
ACESSO

II
EVENTOS

III
AGENTES

