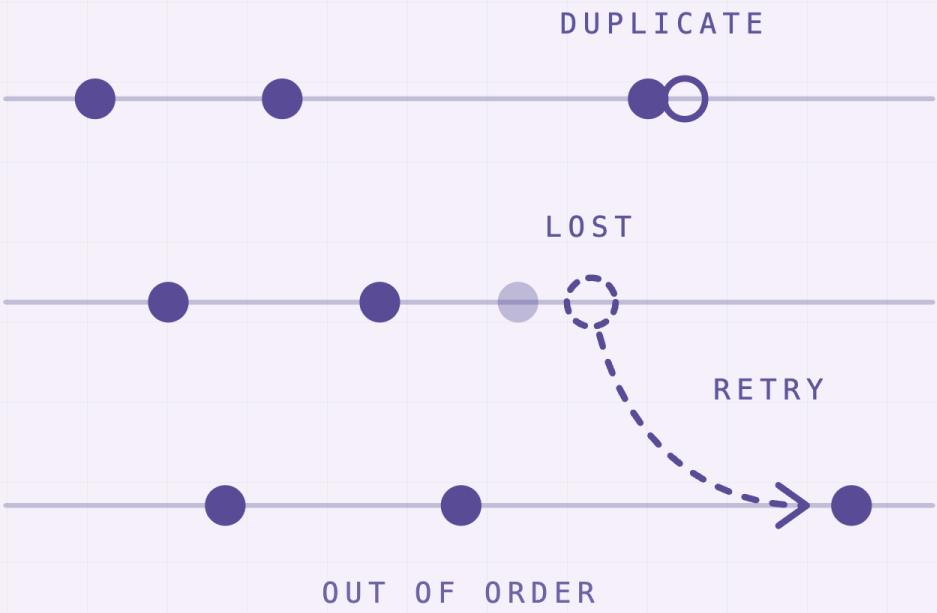
THE UNRELIABLE
SYSTEMS TRILOGY

II



Concevoir des systèmes orientés événements fiables

Muhammet Şafak

Concevoir des systèmes orientés événements fiables

Messagerie, retries, idempotence, outbox, ordre des messages et évolution des schémas

Muhammet Şafak

© 2026 Muhammet Şafak
Téléchargement et lecture gratuits.

Table des matières

Préface 5

PARTIE I — FONDEMENTS 13

Chapitre 1 — Le mensonge du happy path 14

Chapitre 2 — Sémantique de la messagerie 27

Chapitre 3 — Un modèle de défaillances pragmatique 45

PARTIE II — PUBLIER DE FAÇON FIABLE 64

Chapitre 4 — Le problème du dual write et le transactional outbox 65

Chapitre 5 — Relays, polling et change data capture 86

PARTIE III — CONSOMMER DE FAÇON FIABLE 108

Chapitre 6 — Des retries bien faits 109

Chapitre 7 — Idempotence 135

Chapitre 8 — Poison messages et dead letters 158

Chapitre 9 — L'ordre des messages 181

PARTIE IV — WORKFLOWS ET CONTRATS 203

Chapitre 10 — Sagas et workflows de longue durée 204

Chapitre 11 — Conception des événements 226

Chapitre 12 — Évolution des schémas 247

PARTIE V — EXPLOITER ET PROUVER 270

Chapitre 13 — Observabilité et réconciliation 271

Chapitre 14 — Reprise, sinistres et migration de brokers 295

Chapitre 15 — Tester les systèmes orientés événements 322

Chapitre 16 — Une checklist de fiabilité et réflexions finales 341

Annexe : Glossaire et référence rapide des patterns 354

Préface

Imaginez un mardi après-midi chez Parcelly, une marketplace qui n'existe pas. Le parcours de checkout fonctionne sans accroc depuis des mois. Un acheteur passe une commande, le service Orders l'écrit dans sa base de données, et un événement `OrderPlaced` part vers le message broker. Payments autorise la carte. Inventory réserve le stock. Shipping planifie l'expédition du colis. Tous les tableaux de bord sont au vert.

Puis un déploiement traverse Orders. Chaque instance traite une commande de la même façon : elle fait le commit de la commande dans sa base de données, puis publie `OrderPlaced` vers le message broker. Pendant quelques secondes, des instances s'arrêtent tandis que d'autres démarrent. Une instance fait le commit d'une commande et est arrêtée avant l'appel de publication. La commande est stockée en sécurité, et l'événement ne part jamais. Rien ne plante bruyamment et aucune alerte ne se déclenche. Le lendemain matin, un acheteur regarde une commande marquée comme passée, mais Payments n'a jamais autorisé de carte et Inventory n'a jamais réservé le moindre article. Orders croit que le travail est en cours, et le reste du système n'en a jamais entendu parler.

Personne n'a écrit de bug au sens habituel. Chaque ligne de code a fait ce qu'elle annonçait. Le système a défailli dans l'espace entre les lignes : entre un commit en base de données et un appel réseau, entre un message livré et un message compris, entre « nous l'avons envoyé » et « ils l'ont fait ».

C'est de cet espace que parle ce livre.

Pourquoi ce livre existe

L'architecture orientée événements a un problème de réputation, et elle le mérite dans les deux sens. Ses partisans décrivent des services faiblement couplés qui passent à l'échelle indépendamment et évoluent sans coordi-

nation. Ses détracteurs décrivent un spaghetti distribué, où personne ne sait dire ce qui se passe quand une commande est passée. Les deux décrivent le même système, à des jours différents.

Ce qui sépare ces deux jours, c'est rarement le choix du broker, du framework, ni même le diagramme d'architecture. C'est une courte liste de questions sans éclat. Que se passe-t-il si ce message est livré deux fois ? Que se passe-t-il s'il n'arrive jamais ? Que se passe-t-il s'il arrive après le message qui devait le suivre ? Que se passe-t-il si le consommateur échoue à chaque fois ? Que se passe-t-il quand le producteur doit modifier un champ alors qu'une centaine de consommateurs exécutent encore l'ancien code ?

Ces questions sont la thèse de ce livre. Les systèmes orientés événements fiables se construisent autour de la **sémantique des défaillances**, et non des message brokers : pour une jonction donnée, ce qui peut défaillir, comment on l'observerait, et quelle garantie y survit. Une jonction est tout endroit où un composant passe du travail à un autre, comme un commit en base de données suivi d'une publication, ou une livraison suivie d'un effet. Les patterns présentés ici existent à cause du réseau et de la base de données, et non à cause d'un produit particulier, et un broker n'est qu'un composant de plus, avec ses propres jonctions.

Les équipes rencontrent généralement ces questions une à une, en production, dans l'ordre où le système se trouve défaillir. Chaque réponse s'apprend séparément, et les leçons finissent rarement au même endroit. La documentation d'un broker vous explique comment publier et consommer. Elle est beaucoup plus discrète sur ce à quoi ressemble un service correct autour de ces deux verbes.

Ce livre essaie d'être cet endroit. C'est un guide des patterns qui rendent fiables les systèmes orientés événements : le transactional outbox, les consommateurs idempotents, les retries avec backoff et jitter, le backpressure, les dead-letter queues, les stratégies d'ordre des messages, les sagas, une conception soignée des événements, l'évolution des schémas, l'observabilité, la reprise après une perte de données et un changement de broker, et les tests. J'ai essayé de traiter chacun comme une décision qui a un prix plutôt que comme une règle à suivre. Un pattern adapté aux paiements de Parcelly peut être du gaspillage pour ses notifications, et le livre le dira quand cela compte.

Je veux aussi être clair sur ce que ce livre n'est pas. Ce n'est pas un tutoriel pour un broker en particulier. Ce n'est pas un catalogue de tous les enterprise integration patterns jamais nommés. Il ne promet pas que le suivre rendra votre système correct. La fiabilité n'est pas une propriété que l'on ajoute ; c'est un ensemble de défaillances dont vous avez décidé, délibérément, comment les traiter, et un ensemble que vous avez décidé, délibérément, d'accepter. Mon but est de rendre ces décisions visibles.

À qui s'adresse ce livre

Ce livre est écrit pour les **ingénieurs backend et plateforme** qui construisent ou exploitent des services communiquant par événements, et pour les **tech leads** qui doivent décider comment ces services doivent se comporter quand les choses tournent mal. Si vous avez déjà contemplé une dead-letter queue qui a grossi pendant la nuit, débattu de la nécessité de rendre un consommateur idempotent, ou hérité d'un topic dont personne ne sait pleinement expliquer les événements, c'est à vous que je pense.

Une expertise en messagerie n'est pas nécessaire. Vous devriez être à l'aise avec ce qui suit :

- Les bases de données relationnelles et les transactions : ce qu'est un commit, ce que fait une contrainte d'unicité, et ce que signifie l'isolation dans les grandes lignes.
- Les services qui communiquent sur un réseau, et le vocabulaire de base des timeouts et des requêtes et réponses de type HTTP.
- L'idée générale d'un message broker : quelque chose qui accepte des messages de producteurs et les remet à des consommateurs.

Aucune expérience d'un produit particulier n'est nécessaire. Quand un chapitre cite Kafka, RabbitMQ ou Amazon SQS, c'est comme exemple d'une famille de systèmes, et il décrit un comportement valable en général plutôt que dans une version donnée. Si vous n'en avez utilisé aucun, le livre restera compréhensible. Si vous en connaissez un en profondeur, vous découvrirez qu'une partie de ce que vous savez lui est propre, et le livre essaiera d'indiquer quelles parties.

Si vous êtes tech lead et n'écrivez pas de code tous les jours, vous pouvez lire le scénario d'ouverture, la section sur les compromis et les points clés de chaque chapitre, et en tirer l'essentiel de la valeur pour la prise de décision. Le pseudocode est là pour les personnes qui ont besoin de voir où se situe exactement la frontière de la transaction.

Comment le livre est organisé

Le livre compte cinq parties, conçues pour être lues dans l'ordre la première fois et consultées dans n'importe quel ordre ensuite.

Partie I, Fondements (chapitres 1 à 3). Nous commençons par démonter le happy path. Le Chapitre 1 montre comment un flux qui paraît simple cache en réalité une douzaine de décisions. Le Chapitre 2 précise ce que signifient réellement les garanties de messagerie, et pourquoi la livraison at-least-once est le point de départ réaliste. Le Chapitre 3 construit un modèle de défaillances pragmatique, un petit vocabulaire pour les façons dont les choses cassent, des timeouts et des crashs jusqu'aux messages qui échouent toujours et à la perte d'un store entier. Le reste du livre l'utilise comme langage commun.

Partie II, Publier de façon fiable (chapitres 4 et 5). Le Chapitre 4 traite le problème du dual write, la faille exacte de la scène d'ouverture, et présente le transactional outbox. Le Chapitre 5 couvre la mécanique qui transporte les lignes de l'outbox jusqu'au broker, en comparant les relays de polling et le change data capture, et ce que chacun vous coûte.

Partie III, Consommer de façon fiable (chapitres 6 à 9). C'est la partie la plus longue, parce que c'est chez les consommateurs que se produisent la plupart des effets en doublon, des files d'attente bloquées et des mises à jour perdues. Le Chapitre 6 couvre les retries : quand faire un retry, quand s'en abstenir, pourquoi le backoff a besoin de jitter, et comment éviter qu'un consommateur soit surchargé par ses propres retries. Le Chapitre 7 porte sur l'idempotence et l'inbox. Le Chapitre 8 traite des poison messages et des dead letters. Le Chapitre 9 aborde l'ordre des messages, et le peu que l'on peut généralement se permettre d'en exiger.

Partie IV, Workflows et contrats (chapitres 10 à 12). Le Chapitre 10 présente les sagas pour les workflows qui traversent plusieurs services et ne peuvent pas partager une transaction. Le Chapitre 11 porte sur la concep-

tion des événements eux-mêmes : ce qu'on y met, ce qu'on en exclut, comment on les nomme, et comment une enveloppe survit au fait d'être écrite et lue dans plusieurs langages. Le Chapitre 12 couvre l'évolution des schémas, la discipline discrète qui permet aux producteurs et aux consommateurs de changer à des rythmes différents.

Partie V, Exploiter et prouver (chapitres 13 à 16). Le Chapitre 13 porte sur la capacité à voir ce que fait votre système, et sur la réconciliation, l'habitude de confronter la vue d'un système à celle d'un autre. Le Chapitre 14 traite de la reprise après une perte de données, la perte d'une région et un changement de broker. Le Chapitre 15 explique comment tester du code dont les pires bugs n'apparaissent qu'en présence de doublons, de retards et de messages livrés dans le désordre. Le Chapitre 16 rassemble le tout dans une checklist de fiabilité et clôt le livre.

Une annexe contient un glossaire et une référence rapide des patterns, pour que vous puissiez retrouver une notion sans relire un chapitre.

Chaque chapitre, du 1 au 15, suit la même structure. Il s'ouvre sur un court scénario, explique pourquoi le problème est difficile, présente le pattern, puis passe à deux sections que je considère comme les plus précieuses : **Compromis et coûts**, qui dit quand ne pas utiliser le pattern, et **Erreurs courantes**. Chaque chapitre se termine par cinq à sept points clés. Si vous manquez de temps, ces trois parties vous mèneront plus loin que vous ne le pensez.

Vous pouvez lire les chapitres de façon sélective. Si votre problème du moment est une dead-letter queue qui grossit, allez directement au Chapitre 8, et suivez les renvois vers les chapitres 6 et 7 selon les besoins. Si vous concevez un nouveau service, je lirais la Partie II avant d'écrire la première ligne de code de publication, parce que l'outbox est bien plus facile à adopter dès le départ qu'à ajouter après coup.

Faites la connaissance de Parcelly

Tout au long du livre, un système fictif porte les exemples : **Parcelly**, une marketplace en ligne où des vendeurs indépendants proposent des produits et où des acheteurs les commandent. Elle compte six services.

- **Orders** est responsable du cycle de vie des commandes, et c'est là que commencent la plupart des workflows.
- **Payments** autorise et capture les paiements, et dialogue avec un fournisseur de paiement externe.
- **Inventory** suit le stock et gère des réservations temporaires.
- **Shipping** planifie les expéditions et remet les colis aux transporteurs.
- **Notifications** envoie des e-mails, des notifications push et des SMS, et ne doit jamais bloquer quoi que ce soit d'autre.
- **Ledger** tient un registre append-only en partie double des mouvements d'argent.

Chaque service possède sa propre base de données relationnelle et ne lit celle d'aucun autre. Ils communiquent via un message broker en échangeant des événements aux noms tels que `OrderPlaced`, `PaymentAuthorized`, `StockReserved`, `PaymentFailed` et `ShipmentScheduled`. Les événements suivent une forme cohérente, une enveloppe qui porte un identifiant, un type, une version de schéma, des champs de corrélation et un payload, pour que les chapitres puissent en parler d'une seule voix.

J'ai choisi une marketplace à dessein. Encaisser de l'argent, réserver des marchandises et déplacer des colis, c'est un domaine où le coût d'un message perdu ou d'un effet en doublon se fait sentir vite. Un e-mail en double est un désagrément. Une capture en double, c'est un remboursement, un ticket de support et des excuses. Une réservation perdue, c'est un article survendu. Ce gradient, de l'inoffensif au coûteux, est exactement ce qu'il vous faut pour décider combien de mécanique de fiabilité chaque flux mérite.

Parcelly est une fiction, et ses incidents aussi. Les scénarios qui ouvrent chaque chapitre sont inventés pour illustrer une forme de défaillance, et ils sont toujours présentés comme de la fiction. J'utiliserai la première personne pour les jugements, comme « j'ajouterais une contrainte d'unicité ici » ou « à mon avis, cela ne vaut pas la complexité », mais je ne déguiserai pas un raisonnement en histoire vécue. Vous devez pouvoir compter sur ceci : quand le livre dit que quelque chose se produit, c'est en raison du fonctionnement de ces systèmes, et non d'une anecdote.

Le livre est aussi délibérément **indépendant du broker**. Lorsqu'un point dépend de la nature de votre broker, basé sur un log, avec des consommateurs qui suivent un offset dans une partition ordonnée, ou basé sur des files d'attente, avec des messages qui reçoivent un ack et sont livrés à nouveau un par un, je le préciserai. Lorsque cela n'a pas d'importance, je ne prétendrai pas le contraire.

Comment lire le code

Le code de ce livre est du **pseudocode**. Il n'appartient à aucun langage, il ne compilera pas, et rien n'y appelle une vraie bibliothèque. C'est voulu. L'intérêt de chaque extrait est de montrer où une transaction commence et se termine, où un message reçoit son ack, et ce qui se passe quand une étape échoue à mi-chemin. Ces faits sont les mêmes dans tous les langages, et la syntaxe ne ferait que les masquer.

Les blocs sont courts, 30 lignes au plus. Lorsqu'un schéma de table aide, il est écrit dans une notation proche de SQL. Les diagrammes utilisent Mermaid, et chacun est introduit par une phrase qui vous dit quoi y chercher. Si vous transposez un pattern dans votre propre stack, traitez le pseudocode comme une spécification d'ordre et d'atomicité, et non comme une implémentation.

Une remarque sur la certitude

Je ferai des recommandations dans ce livre, et je tiendrai fermement à certaines d'entre elles. Je ne publierais pas, par exemple, des événements avec un simple dual write lorsque la perte d'un événement pèse sur l'activité. D'autres recommandations relèvent du goût ou du contexte, et j'essaierai de dire lesquelles.

Il y aura aussi des endroits où la réponse honnête est « cela dépend, et voici de quoi ». Les retry budgets, la rétention des dead letters, le nombre de partitions et le bon niveau d'ordre dépendent tous de faits concernant votre activité et votre équipe que je ne peux pas voir. À ces endroits, je nommerai les variables au lieu d'inventer un chiffre. Si vous terminez un chapitre en sachant quelles questions poser dans votre propre système, je considérerai que c'est réussi.

Le travail de fiabilité a une propriété malheureuse : quand il réussit, rien de visible ne se passe. Les commandes sont honorées, les comptes s'équilibrent, et personne n'écrit de message pour vous remercier. J'espère que ce livre vous aidera à construire ce genre de calme.

Commençons par le happy path, et par les raisons de ne pas lui faire confiance.

PARTIE I

Fondements

Chapitre 1 — Le mensonge du happy path

Imaginez Parcelly lors de sa première année. Orders appelle Payments en HTTP, attend la réponse, puis appelle Inventory, attend de nouveau, puis appelle Shipping. Le code se lit de haut en bas, une défaillance est une exception, et un acheteur qui voit une page d'erreur peut faire un retry. C'est lent quand le fournisseur de paiement est lent, et tout tombe quand Inventory est indisponible, mais personne ne se demande ce qui s'est passé.

Puis l'équipe décide de passer aux événements. Orders publiera `OrderPlaced` vers un message broker, et les autres services réagiront à leur propre rythme. La démo est superbe. Un acheteur clique sur « Passer la commande », et en moins d'une seconde le stock est réservé, la carte est débitée, la commande est confirmée et une expédition est planifiée. Tout le monde applaudit.

Imaginez maintenant le premier mois en production. Un acheteur est débité, mais la commande reste « en attente » pour toujours. Une autre commande est expédiée deux fois. Une troisième a une écriture dans le ledger sans paiement correspondant, et personne ne sait dire quel service a raison. Le support demande « qu'est-il arrivé à la commande 81724 ? », et la réponse honnête est que quatre services en connaissent chacun un morceau et qu'aucun ne connaît le tout.

Rien dans la démo n'était faux. Elle était incomplète, parce qu'une démo est une histoire où chaque étape réussit. La question à laquelle répond ce chapitre est ce que l'équipe a abandonné en quittant le modèle requête-réponse, et ce que « fiable » doit signifier avant que quiconque commence à le reconstruire.

Le problème, et pourquoi il est difficile

Le happy path est la séquence d'événements qui se produit quand chaque composant fonctionne, chaque appel réseau arrive, chaque message est livré une seule fois et dans l'ordre, et chaque consommateur termine ce qu'il commence. C'est le chemin que l'on dessine au tableau blanc et celui qu'une démo exerce. C'est aussi le chemin le plus rare dans un système qui tourne pendant des mois.

Les défaillances n'ont rien d'exotique. Les déploiements redémarrent des processus. Des connexions tombent au milieu d'une écriture. Un broker bascule vers une réplique. Un consommateur met plus longtemps que prévu et son message est confié à quelqu'un d'autre. Chacune de ces choses arrive quelque part dans un système chargé, chaque jour. La forme habituelle d'un mauvais mois en production, c'est une poignée d'événements ordinaires qui rencontrent du code écrit pour le happy path.

Ce qui rend cela difficile, c'est que les systèmes orientés événements suppriment ce qui rendait les défaillances visibles. Dans un appel synchrone, l'appelant attend, et l'issue revient sur la même connexion : un résultat ou une erreur, à côté du code qui l'a provoqué. Dans une conception orientée événements, le producteur remet un message au broker et passe à autre chose. Si quelque chose tourne mal trois services et dix secondes plus loin, le producteur est parti depuis longtemps, et les preuves sont dispersées dans des logs que personne n'a encore corrélés.

La réponse évidente consiste à rendre chaque pièce fiable : un bon broker, des services bien testés, des retries sur chaque appel. Chacune aide, et aucune ne suffit, parce que les défaillances qui font mal vivent dans les interstices entre les composants. Un message écrit dans la base de données mais jamais publié, ou traité mais sans jamais recevoir d'ack, n'est un bug d'aucun des deux composants. C'est une propriété de la jonction. L'essentiel de ce livre porte sur les jonctions.

Pour chaque jonction, le livre pose les trois mêmes questions : qu'est-ce qui peut défaillir, comment l'observerait-on, et quelle garantie y survit. J'appellerai les réponses la **sémantique des défaillances** de la jonction. La documentation d'un broker vous donne la sémantique des jonctions

propres au broker. Les jonctions qui l'entourent, entre votre base de données et votre appel de publication ou entre une livraison et son effet, c'est à vous de les élucider, et le Chapitre 3 montre comment.

Couplage synchrone et asynchrone

Le passage aux événements est généralement décrit comme un « découplage », un mot qui promet plus qu'il ne tient. Il est plus exact de dire que le couplage change de forme.

Un appel synchrone couple deux services dans le **temps**, dans la **disponibilité** et dans la **connaissance**. L'appelé doit fonctionner maintenant, la latence de l'appelant inclut celle de l'appelé, et l'appelant sait qui est l'appelé et ce qu'il répondra. L'avantage est un modèle de défaillances simple : l'appelant sait que l'appel a échoué et peut décider immédiatement de faire un retry, de basculer sur un fallback ou de signaler une erreur. Le coût est que la chaîne n'est disponible qu'à hauteur de son maillon le plus faible. Une API de transporteur lente donne un checkout lent.

Un message transmis par un broker supprime le couplage dans le temps et dans la disponibilité. Orders peut accepter une commande pendant qu'Inventory redémarre, parce que le broker conserve le message jusqu'au retour d'Inventory. Un nouveau consommateur, comme un service d'analytics, peut commencer à écouter sans qu'Orders change. Ce gain est réel, et c'est la raison d'adopter ce style.

Mais le couplage s'est déplacé, il n'a pas disparu. Les services sont désormais couplés par le **contrat de l'événement** : son nom, ses champs et sa signification. Ils sont couplés par le **comportement du broker** : ce qu'il promet en matière de durabilité, d'ordre et de livraison. Et ils sont couplés par des hypothèses sur le timing des uns et des autres que personne n'a mises par écrit. Le Chapitre 11 et le Chapitre 12 traitent du contrat, et le Chapitre 2 traite du broker.

En bref, les systèmes synchrones défaillent bruyamment et en chaîne. Les systèmes asynchrones défaillent en silence et par morceaux.

Le prix du découplage temporel

Le découplage temporel signifie que le producteur et le consommateur n'ont pas besoin d'être actifs en même temps. C'est la principale chose que l'on achète avec les événements, il vaut donc la peine d'être précis sur le prix.

Le producteur n'apprend plus l'issue

Quand Orders appelait Payments de façon synchrone, il savait si le paiement avait fonctionné avant de répondre à l'acheteur. Avec les événements, Orders publie `OrderPlaced` et répond « reçu ». La réussite du paiement est une question pour plus tard, et la réponse arrive, si elle arrive, sous la forme d'un autre événement. La commande a désormais un état intermédiaire, « passée mais pas encore confirmée », et le système doit décider de ce que voit l'acheteur tant qu'il dure et de ce qui se passe s'il dure plus longtemps que prévu.

Personne n'est responsable du flux entier

Dans la version synchrone, une seule fonction contenait la séquence, et on pouvait lire le processus métier à un seul endroit. Dans la version orientée événements, le processus est la somme de réactions éparpillées entre les services. Payments réagit à `OrderPlaced`, Orders réagit à `PaymentAuthorized` et `StockReserved`, Shipping réagit à `OrderConfirmed`. Le flux n'existe que comme propriété émergente, et personne ne peut montrer où il est défini.

Cela ne pose pas de problème quand tout fonctionne et coûte cher quand quelque chose ne fonctionne pas, parce que la question « où cette commande s'est-elle bloquée ? » n'a pas d'endroit unique où chercher. Le Chapitre 10 y revient avec les sagas, et le Chapitre 13 couvre ce qu'il faut observer pour pouvoir seulement répondre à la question.

Les défaillances cessent d'être des exceptions

Dans une chaîne d'appels, une erreur remonte la pile jusqu'à ce que quelque chose la traite. Dans un flux de messages, il n'y a pas de pile. Un consommateur en échec peut faire un ack du message et le perdre, le refuser et le voir livré à nouveau, ou le mettre de côté pour un humain. Chacune de ces options est une décision de politique avec des conséquences

pour des services qui ne voient jamais l'erreur. À mon avis, la gestion des erreurs dans un système synchrone relève du flux de contrôle, et dans un système asynchrone elle relève de la conception, parce que le traitement est réparti dans le temps, entre des processus et parfois entre des équipes.

Le temps devient une variable

Entre « publié » et « traité », il y a un intervalle sans borne supérieure, à moins que vous ne lui en donniez une. Pendant cet intervalle, un produit peut être épuisé, un acheteur peut annuler, un prix peut changer. Un consommateur doit décider si un événement qui était vrai au moment de sa publication a encore un sens au moment de son traitement.

> **Remarque :** C'est pour cela que les événements sont nommés au passé. `OrderPlaced` est un fait concernant un moment révolu. Un consommateur qui le traite comme une instruction d'agir tout de suite a supposé sans le dire qu'aucun temps ne s'était écoulé. Un command, comme `CapturePayment`, demande à un service de faire quelque chose et peut être refusé ; un événement rapporte quelque chose qui s'est déjà produit. Le Chapitre 11 développe la distinction.

Ce que signifie vraiment «à terme»

La plupart des explications de la **cohérence à terme** tiennent en une phrase : si aucune nouvelle mise à jour n'est faite, toutes les répliques convergent vers la même valeur. C'est exact et presque inutile pour qui construit un parcours de checkout, parce que cela omet combien de temps dure « à terme » et quoi faire pendant l'attente.

«À terme» n'est pas une durée

Un système qui converge en 200 millisecondes et un système qui converge après un traitement par lots nocturne sont tous deux cohérents à terme. Pour un acheteur qui fixe une page de confirmation, ce sont des produits différents. Je demanderais un chiffre pour chaque saut qui compte, même approximatif. Disons qu'Orders doit afficher une commande confirmée dans les quelques secondes qui suivent la capture, et que Notifications peut prendre une minute. Ces chiffres sont des illustrations et les vôtres

seront différents, mais ils transforment « à terme » en une exigence que l'on peut tester et sur laquelle on peut alerter. Un délai sans borne n'est pas un choix de conception. C'est l'absence de choix.

«À terme» n'est pas une garantie

La convergence n'a lieu que si chaque message qui compte finit par être livré, que chaque consommateur le traite et qu'aucune étape ne le perd en silence. Si un seul message est perdu entre un commit en base de données et une publication vers le broker, les vues ne convergent pas. Elles restent différentes jusqu'à ce qu'une personne ou un job de réparation s'en aperçoive. « À terme » est une promesse conditionnelle, et la condition est que le système dispose de mécanismes qui rendent la perte détectable et réparable. Choisir un broker ne les fournit pas. Les parties II et III de ce livre empêchent l'essentiel des pertes, et le Chapitre 13 détecte et répare le reste.

Que faire pendant l'attente

Pendant la fenêtre d'incohérence, d'autres lecteurs voient un état intermédiaire. Prenons cet instantané d'une commande, un instant après que l'acheteur a été débité :

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

Chaque service a raison sur ses propres données. La combinaison paraît incohérente et elle est normale pendant quelques secondes. Elle ne devient un problème que si elle perdure. La question opérationnelle utile n'est pas « les services sont-ils cohérents en ce moment ? » mais « depuis combien de temps cette commande est-elle dans cet état, et est-ce plus long que ce que nous tolérons ? »

Une définition opérationnelle de la fiabilité

« Fiable » est employé de façon si lâche que deux ingénieurs peuvent convenir qu'un système doit être fiable et vouloir dire des choses opposées. Je veux une définition que l'on peut vérifier. Pour les systèmes orientés événements, j'utilise quatre propriétés. Elles disent ce qui est protégé ; les trois questions ci-dessus sont la façon de vérifier une jonction à leur aune.

Aucune perte. Chaque événement qui représente un fait métier atteint chaque consommateur qui en a besoin. La perte est la défaillance la plus dommageable parce qu'elle est silencieuse : le système continue de tourner, et le seul symptôme est une absence que personne ne remarque jusqu'à ce qu'un acheteur se plaigne. L'absence de perte est une propriété de tout le chemin, pas du broker. Un broker durable n'aide en rien si le producteur n'a jamais réussi à stocker le message, ou si le consommateur a fait un ack avant de finir le travail. Le Chapitre 2 examine ce que promettent les brokers, et le Chapitre 4 et le Chapitre 5 traitent du côté producteur.

Aucune duplication nuisible. Certains messages seront livrés plus d'une fois. Ce n'est pas un défaut d'un broker particulier mais le prix à payer pour ne pas perdre de messages : quand un émetteur ne peut pas savoir si un message est arrivé, le geste sûr est de le renvoyer. Un système fiable n'essaie pas d'empêcher les doublons au niveau du transport. Il les rend inoffensifs. Le mot « nuisible » compte. Deux e-mails identiques sont un désagrément, tandis qu'une seconde capture du même paiement est un incident financier. Le Chapitre 7 couvre l'idempotence, l'outil principal.

Délai borné. Chaque événement est traité dans un délai que l'activité tolère, ou le système signale haut et fort qu'il ne l'a pas été. Un message livré correctement avec trois jours de retard est, à bien des égards, un message perdu. Borné ne veut pas dire rapide. Cela veut dire qu'une limite existe, que quelqu'un l'a choisie et que son dépassement produit une alerte. Une confirmation de paiement peut exiger quelques secondes ; un relevé hebdomadaire peut tolérer des heures. Le Chapitre 6 traite des retries et du backoff, et le Chapitre 13 du retard du consommateur et des alertes.

Explicabilité. Pour toute commande, à tout moment, quelqu'un peut reconstituer ce qui s'est passé : quels événements ont été produits et consommés, ce que chaque service a décidé et pourquoi. C'est la propriété la moins prestigieuse, et c'est elle qui décide si les trois autres peuvent être tenues. Un système qui perd un message mais peut montrer exactement où est réparable. Un système qui se comporte mal d'une manière que personne ne peut retracer ne l'est pas. Le Chapitre 13 couvre les identifiants, les enregistrements et les comparaisons nécessaires.

Les quatre tirent dans des directions opposées, c'est pourquoi une définition qui n'en retient qu'une est dangereuse. Viser l'absence de perte avec des retries illimités menace le délai borné. Abandonner les messages lents pour borner le délai menace l'absence de perte. Refuser les retries pour éviter les doublons ramène la perte. La fiabilité est un équilibre délibéré, choisi flux par flux et mis par écrit.

Pourquoi le happy path échoue en production

Avec cette définition en main, revenons au premier mois de Parcelly et classons les défaillances par propriété. Quelques mécanismes expliquent l'essentiel des dégâts.

Ce que Parcelly a vu	Propriété menacée	Cause sous-jacente	Chapitre
Commande acceptée, mais rien ne se passe jamais en aval	Aucune perte	Événement enregistré par un commit en base de données mais jamais publié	4
Commande expédiée deux fois	Aucune duplication nuisible	Message livré à nouveau et traité une seconde fois	7
Un seul message défectueux bloque une partition	Délai borné	Un message échoue à chaque fois et fait l'objet de retries sans fin	8
Commande « confirmée » avant d'être « passée »	Explicabilité	Événements liés traités dans le désordre	9
Ledger ne concorde pas avec les paiements	Explicabilité	Aucun identifiant partagé, aucun moyen de comparer les vues	13

La quatrième ligne est la plus délicate. Le désordre ne duplique rien. Il endommage l'état, par exemple quand une mise à jour périmée écrase une plus récente, et aucune des quatre propriétés ne nomme cela directement. Je la range sous l'explicabilité parce que le symptôme est un historique que personne ne peut expliquer, et le Chapitre 9 traite des dégâts sur l'état eux-mêmes.

La première ligne mérite qu'on la parcoure pas à pas, parce que c'est la défaillance qui met le plus souvent fin à la lune de miel.

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

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

Ce bloc n'est pas sûr, et il est présenté comme le point de départ défaillant. Le commit en base de données et la publication vers le broker concernent deux systèmes sans transaction partagée, ce que le livre appelle un **dual write**. Si le processus s'arrête entre les deux lignes, la commande existe et personne n'en entend parler. Si les lignes sont inversées, un événement peut être publié pour une commande qui n'a jamais fait l'objet d'un commit. Aucun des deux ordres n'est correct, et tous deux passent tous les tests qui ne tuent pas le processus exactement au mauvais moment.

Ce n'est pas un bug rare. C'est le comportement par défaut d'une conception qui n'a jamais eu de modèle de défaillances. Le Chapitre 3 donne le modèle, et le Chapitre 4 donne la correction.

Une carte du livre

La suite du livre répond aux questions que soulève la définition, dans l'ordre où une équipe les rencontre habituellement.

Question que vous vous poserez	Chapitre
Que promet réellement le broker, et que ne promet-il pas ?	2
Qu'est-ce qui peut casser, et que verrai-je quand cela arrivera ?	3

Question que vous vous poserez	Chapitre
Comment modifier ma base de données et publier un événement sans en perdre un seul ?	4
Comment les événements publiés vont-ils de la base de données au broker ?	5
Comment faire des retries sans aggraver les choses ?	6
Comment rendre un traitement répété inoffensif ?	7
Que faire d'un message qui ne pourra jamais réussir ?	8
Quand l'ordre compte-t-il, et comment le préserver ?	9
Comment coordonner le travail entre services, et comment l'annuler ?	10
Que doit contenir un événement, et comment le nommer ?	11
Comment modifier un événement sans casser les consommateurs ?	12
Comment voir, mesurer et réparer ce qui se passe ?	13
Comment se remettre d'une perte de données, de la perte d'une région ou d'un changement de broker ?	14
Comment tester les chemins de défaillance ?	15
Que vérifier avant la mise en production ?	16

La Partie I, que ce chapitre ouvre, construit le vocabulaire et le modèle de défaillances. La Partie II couvre la publication fiable, la Partie III la consommation fiable, la Partie IV les workflows et les contrats, et la Partie V l'exploitation et la preuve du système. Lisez les parties dans l'ordre pour une première lecture. Si vous combattez un problème précis aujourd'hui, le tableau sert d'index.

Compromis et coûts

Passer aux événements n'est pas une erreur, et ce chapitre n'est pas un plaidoyer contre. C'est un plaidoyer pour faire ce choix en connaissance de cause, parce que le coût est réel et qu'il se paie à des endroits précis.

Vous payez en complexité d'état. Le flux synchrone avait deux issues : terminé ou échoué. Le flux orienté événements a de nombreux états intermédiaires, et chacun doit être nommé, montré aux utilisateurs, surveillé et

finalement résolu. Quelqu'un doit être responsable de cette machine à états.

Vous payez en compétence opérationnelle. Déboguer une chaîne d'appels demande une stack trace. Déboguer un flux de messages demande des correlation ids, des tableaux de bord de retard des consommateurs et la capacité de raisonner sur le timing. Une équipe qui n'a pas ces moyens constate que le temps nécessaire pour comprendre un incident augmente, même si les incidents deviennent moins fréquents. Le coût retombe sur la personne d'astreinte.

Vous payez en effort de test. Le happy path demande peu d'effort à tester. Les chemins de défaillance demandent des messages en double, des messages dans le désordre et des redémarrages à des moments délicats. Le Chapitre 15 montre comment le faire sans une suite lente et flaky, mais l'effort n'est pas nul.

Vous payez en latence pour des choses simples. Une opération qui tenait en un appel de fonction devient plusieurs sauts à travers un broker. Quand l'appelant a besoin de la réponse tout de suite, par exemple pour valider un code de réduction avant d'afficher un prix, un appel synchrone est le meilleur outil.

Je n'utiliserais pas d'événements quand deux étapes doivent réussir ou échouer ensemble et vivent dans le même service et la même base de données. Une transaction locale est plus simple, plus forte et moins chère que n'importe quel pattern de ce livre. J'hésiterais aussi lorsque l'équipe n'a pas la capacité d'exploiter un broker et la supervision qui l'entoure, ou lorsque le système est assez petit pour qu'une seule unité déployable avec une seule base de données fasse l'affaire. Beaucoup de bons systèmes mélangent les deux styles : un appel synchrone pour une question qui attend une réponse, un événement pour un fait qui peut intéresser d'autres. Je choisirais interaction par interaction, et non système par système.

La définition en quatre propriétés a aussi une limite. C'est une grille de lecture, pas une spécification. Elle ne vous dit pas la bonne borne de délai pour votre activité ni quels doublons sont nuisibles pour un consommateur donné. Je ne connais pas vos chiffres, et ces réponses viennent rarement de l'ingénierie seule. Ce que je peux offrir, c'est l'habitude de les demander.

Erreurs courantes

- **Ne tester que le happy path.** C'est tentant parce que la demande de fonctionnalité le décrivait et que les tests passent vite. Traitez chaque flèche du diagramme de flux comme une question sur la défaillance, et écrivez au moins un test par réponse (Chapitre 15).
- **Traiter le broker comme la couche de fiabilité.** Les brokers mettent en avant leur durabilité, ce qui rend cette erreur tentante. Un broker couvre le segment entre l'acceptation et la livraison, et les défaillances qui font le plus mal se situent avant et après. Listez ce que promet le broker, puis listez les lacunes autour de cette promesse (Chapitre 2 et Chapitre 3).
- **Utiliser « à terme » comme conception.** Cela ressemble à une décision et esquive la question plus difficile du « combien de temps ». Associez une borne et une alerte à chaque saut asynchrone qui compte, et dites ce que voit l'utilisateur pendant l'attente.
- **Supposer que les doublons sont trop rares pour être traités.** Dans un environnement de test calme, ils n'apparaissent jamais. Concevez les consommateurs en partant du principe que chaque message peut arriver plus d'une fois (Chapitre 7).
- **Faire le commit en base de données et la publication vers le broker en deux étapes simples.** Cela se lit naturellement et fonctionne dans chaque démo. La correction est le transactional outbox du Chapitre 4.
- **Définir la fiabilité comme l'uptime.** L'uptime est peu coûteux à mesurer, mais un service peut rester disponible tout le mois et quand même perdre des messages ou les traiter en retard. Mesurez directement les quatre propriétés : ce qui a été perdu, ce qui a été dupliqué, quel était le retard, et si vous pouviez l'expliquer.

Points clés

- Le happy path est une histoire dans laquelle chaque étape réussit, et un système testé uniquement face à lui n'a été démontré fonctionnel que dans le seul cas qui tient rarement longtemps.
- Passer des appels synchrones aux événements ne supprime pas le couplage ; cela le déplace du temps et de la disponibilité vers les contrats,

le comportement du broker et des hypothèses de timing non écrites.

- Les défaillances des systèmes orientés événements sont silencieuses et réparties entre les services, si bien qu'une conception sans modèle de défaillances tend à découvrir ses jonctions en production.
- « À terme » est une promesse conditionnelle sans durée associée, donc chaque saut asynchrone qui compte a besoin d'une borne, d'une alerte et d'un état défini, visible par l'utilisateur, pour la période d'attente.
- Un système fiable n'a aucune perte, aucune duplication nuisible, un délai borné et de l'explicabilité, et comme ces propriétés se font concurrence, chaque flux a besoin d'un équilibre délibéré et écrit.
- Les jonctions entre composants, comme un commit en base de données suivi d'une publication vers le broker, causent plus de dégâts que les composants eux-mêmes, et l'essentiel de ce livre porte sur ces jonctions, examinées à travers trois questions : qu'est-ce qui peut défaillir, comment l'observerait-on, et quelle garantie y survit.
- Les événements ne sont pas toujours le bon outil : quand les étapes peuvent partager une transaction locale, ou que l'appelant a besoin d'une réponse immédiate, un appel synchrone est plus simple et plus fort.

Le chapitre suivant examine le broker lui-même. Avant de pouvoir concevoir autour de ses lacunes, nous devons savoir exactement ce qu'une file d'attente ou un log promet quand il dit qu'un message a été livré, et ce qu'il vous laisse discrètement.

Chapitre 2 — Sémantique de la messagerie

Imaginez un vendredi soir chez Parcelly, en pleine période de soldes. Le service Notifications envoie depuis des mois des e-mails « votre commande est confirmée » sans la moindre réclamation. À mesure que le backlog grossit, l'autoscaling ajoute des consommateurs, puis, quelques minutes plus tard, en retire quelques-uns. À la fin de la soirée, le support a deux tickets. Un acheteur a reçu trois fois le même e-mail de confirmation. Un autre n'en a reçu aucun, alors que la commande est confirmée.

L'équipe ouvre le tableau de bord du broker et constate que rien n'est cassé. Chaque offset a fait l'objet d'un commit. Chaque consommateur est en bonne santé. Les courbes sont plates. Pourtant, le même système a produit un doublon pour un acheteur et un silence pour un autre, et les deux résultats sont conformes à la documentation.

Quelqu'un dit que le broker promettait une « livraison at-least-once », et quelqu'un d'autre dit avoir supposé que le broker s'en chargeait. Les deux affirmations sont vraies, et aucune n'explique ce qui est arrivé aux deux acheteurs. Supposons que Notifications lise un topic basé sur un log en tant que groupe de consommateurs unique, avec un client qui fait automatiquement le commit des offsets, et qu'un rééquilibrage se soit produit à chaque changement de taille du groupe. Ces deux faits peuvent produire les deux tickets, et le chapitre montre comment.

Pour l'expliquer, l'équipe a besoin d'une image précise de ce que promet un broker, de ce qu'il ne promet pas, et des parties de la vie du message qui relèvent plutôt de l'application. Par quoi un message passe-t-il en réalité entre un producteur et un effet de bord achevé, et où chaque garantie commence-t-elle et s'arrête-t-elle ?

Le problème, et pourquoi il est difficile

La plupart des ingénieurs découvrent un broker par sa description la plus simple : vous publiez un message, et un consommateur le reçoit. Cette description masque trois passages de relais distincts, et chaque question de fiabilité de ce livre se situe dans l'un d'eux.

Le premier passage va du producteur au broker. Le deuxième va du broker au consommateur. Le troisième va du consommateur à l'effet : l'écriture en base de données, l'e-mail, l'appel à un transporteur. Un broker a beaucoup à dire sur les deux premiers et presque rien sur le troisième, parce que le troisième se joue dans du code et dans des systèmes que le broker ne voit pas.

La difficulté tient à ce que chaque passage se termine par un ack, et un ack est une déclaration sur le passé, faite par une partie à une autre à travers un réseau. Il dit « j'ai reçu ceci » ou « j'ai terminé ceci ». Il ne dit pas « et rien d'autre ne peut mal tourner ». Quand un ack est perdu, l'émetteur ne peut pas savoir si le travail a eu lieu. Cette incertitude est la racine des trois sémantiques de livraison, et nous y reviendrons une fois le vocabulaire posé.

Une deuxième difficulté tient à ce que le mot « broker » recouvre deux familles qui se comportent différemment, et ces différences comptent. Une **file d'attente** est une structure de broker dans laquelle chaque message est livré à un seul consommateur et reçoit un ack individuel. Un **log d'événements** est une séquence d'événements durable et append-only, que les consommateurs lisent à leur propre rythme en enregistrant une position. Ce qui est vrai pour une famille est souvent faux pour l'autre. Le livre nomme la famille chaque fois que la distinction compte, et c'est dans ce chapitre que la distinction se construit.

Une troisième difficulté tient à ce que les garanties de la documentation d'un produit ont un périmètre. « Livré au moins une fois » est une affirmation qui vaut à la frontière du broker lui-même. Elle ne couvre pas ce que fait le consommateur après avoir reçu le message, et elle dépend de réglages que le producteur a choisis au moment de publier. Lire la garantie sans son périmètre, c'est ainsi qu'une équipe se retrouve avec le vendredi soir décrit plus haut.

Files d'attente et logs

Le modèle de la file d'attente

Dans une file d'attente, le broker suit l'état de chaque message. Un message est en attente, livré à un consommateur, ou a reçu un ack et a été supprimé. Plusieurs consommateurs peuvent lire la même file d'attente ; cette organisation s'appelle les **consommateurs concurrents**, parce que chaque message est livré à un seul d'entre eux à la fois et qu'ils se disputent le suivant.

Quand un consommateur fait un ack sur un message, le broker le supprime. La rétention est donc courte par construction : un message vit jusqu'à ce qu'il soit traité, ou jusqu'à ce qu'il expire. Il n'existe aucun moyen intégré de redemander à la file d'attente un message qui a déjà reçu un ack, parce que la file d'attente ne le conserve pas.

La conséquence est qu'une file d'attente convient bien à un travail qui doit être fait une fois puis oublié. Un job de Parcelly comme « générer le PDF de cette facture » est un candidat naturel : on fait le travail, puis le message n'a plus d'utilité. Le coût est que l'historique n'est pas disponible. Si un bug envoie le mauvais texte pendant une heure, les messages qui portaient les faits ont disparu, et la réparation doit partir d'ailleurs.

Le modèle du log

Dans un log d'événements, le broker ne suit pas d'état par message. Il stocke les messages dans l'ordre, dans des partitions, et les conserve selon une politique de rétention indépendante de qui les a lus. Une **partition** est une sous-séquence ordonnée d'un topic, et un **topic** est un stream de messages nommé. Chaque message a un **offset**, sa position dans la partition.

Les consommateurs suivent eux-mêmes jusqu'où ils ont lu. Cette position s'appelle aussi un offset, et le consommateur, ou le broker pour son compte, l'enregistre. Un **groupe de consommateurs** est un ensemble de consommateurs qui se partagent le travail d'un topic : chaque partition est attribuée à un seul consommateur du groupe à la fois. Des groupes différents lisent le même topic indépendamment, chacun avec ses propres offsets.

La propriété qui définit le log est que la lecture ne consomme pas. Dix groupes peuvent lire le même topic, et un nouveau groupe peut partir du début. Le **replay**, la relecture délibérée de l'historique, est le terme que ce livre réserve à cet acte, et il n'est possible que parce que le log conserve les données. Le Ledger de Parcelly est un bon exemple de consommateur qui en tire profit : si l'on trouve un bug dans la façon dont les écritures sont construites, on peut corriger le service Ledger et le faire repartir sur d'anciens événements.

Le coût est que le log reporte la comptabilité du côté du consommateur. Dans le modèle de log classique, le broker ne peut pas livrer à nouveau un message en échec tout en passant au suivant, parce qu'il ne suit pas les messages individuellement. Un consommateur qui ne parvient pas à traiter le message à l'offset 500 doit décider quoi faire avant de pouvoir passer sans risque à 501.

Choisir entre les deux

Préoccupation	Famille des files d'attente	Famille des logs
Unité de suivi	Chaque message	Une position par partition
Rétention	Jusqu'à l'ack ou l'expiration	Par durée ou par taille, indépendamment des lecteurs
Plusieurs lecteurs indépendants	Nécessite une copie du message par lecteur	Naturel, une position par groupe
Replay de l'historique	Indisponible	Disponible pendant la rétention
Un message en échec	Livré à nouveau seul, les autres continuent	Bloque la partition sauf traitement explicite
Ordre	FIFO dans le stockage au mieux ; perdu avec les consommateurs concurrents et les nouvelles livraisons	Fort au sein d'une partition

Je ne choisirais pas une famille selon la popularité de l'outil. Je partirais de trois questions : quelqu'un a-t-il besoin de relire cet historique, plusieurs consommateurs ont-ils besoin des mêmes faits, et l'ordre compte-t-il

pour un même agrégat ? Un oui à l'une d'elles oriente vers un log. Une pure file de travail, sans besoin d'historique et avec un besoin de retries indépendants pour chaque message, oriente vers une file d'attente.

Beaucoup d'équipes finissent par exploiter les deux. Les événements à partir desquels d'autres services construisent leur état vont sur un log, et les tâches qui doivent avoir lieu une seule fois vont sur une file d'attente. C'est une conception légitime, et elle s'accompagne du coût d'exploiter et de comprendre deux jeux de sémantiques. Le Chapitre 9 revient sur ce que signifie l'ordre dans chaque famille.

Push, pull et le flux de livraison

Les brokers diffèrent aussi par l'initiateur de la livraison. Dans un modèle **push**, le broker envoie les messages aux consommateurs à mesure qu'ils deviennent disponibles. Dans un modèle **pull**, le consommateur demande des messages quand il est prêt.

Les brokers à files d'attente font souvent du push (RabbitMQ livre aux consommateurs, alors que les consommateurs d'Amazon SQS font du polling, si bien que même cela varie au sein de la famille), avec une limite sur le nombre de messages sans ack qu'un consommateur peut détenir à la fois. Cette limite s'appelle souvent un prefetch count ou une fenêtre. Les brokers basés sur un log font souvent du pull : le consommateur demande le lot suivant à partir de son offset courant. On trouve les deux modèles dans l'une et l'autre famille, je traiterais donc cela comme une tendance et non comme une loi.

La différence compte surtout pour ce qui se passe quand un consommateur est lent. Dans un modèle push sans limite, le broker continue d'envoyer, la mémoire du consommateur se remplit de messages qu'il n'a pas commencé à traiter, et ces messages restent dans un état où ils ne sont ni disponibles pour un autre consommateur ni terminés. Dans un modèle pull, le consommateur contrôle son propre rythme, et un consommateur lent se manifeste par un retard du consommateur croissant, et non par un processus surchargé. Le **retard du consommateur** est la distance qui sépare un consommateur du head d'une partition ou d'une file d'attente, en messages ou en temps.

La règle pratique est qu'un consommateur ne devrait jamais détenir plus de travail qu'il ne peut en terminer avant que sa réservation de ce travail n'expire. Le Chapitre 3 traite les consommateurs lents et le backpressure comme des défaillances à part entière, et le Chapitre 6 explique comment borner la charge. Pour l'instant, retenez que « livré » et « en cours de traitement » sont des états différents, et que c'est dans l'écart entre les deux que le temps passe.

Ack, nouvelle livraison et offsets

L'ack dans une file d'attente

Un **acknowledgment**, généralement abrégé en **ack**, est la déclaration du consommateur selon laquelle il en a fini avec un message et que le broker peut l'oublier. Un **nack**, un acknowledgment négatif, dit que le consommateur n'a pas pu traiter le message. Selon le broker et les réglages, un nack remet le message dans la file d'attente, le supprime, ou l'envoie vers une dead-letter queue (DLQ).

La question de conception clé est de savoir quand faire l'ack. Il n'existe que deux positions raisonnables.

- **Ack avant le traitement.** Le broker oublie immédiatement le message. Si le consommateur plante en plein travail, le message est perdu. Cela produit un comportement at-most-once.
- **Ack après le traitement.** Le broker conserve le message jusqu'à ce que le travail soit fait. Si le consommateur plante après avoir terminé mais avant de faire l'ack, le message revient et le travail est fait deux fois. Cela produit un comportement at-least-once.

Aucune position ne donne les deux propriétés, parce que l'ack et le travail sont deux actions distinctes sur deux systèmes distincts, et qu'un plantage peut tomber entre les deux.

Visibility timeout et nouvelle livraison

Comment une file d'attente décide-t-elle qu'un consommateur a planté ? Elle ne peut pas l'observer directement. Certains brokers surveillent la connexion et traitent une connexion fermée comme une défaillance. D'autres utilisent un **visibility timeout** : quand un message est livré, il de-

vient invisible pour les autres consommateurs pendant un temps donné. Si le consommateur fait l'ack avant l'échéance, le message est supprimé. Sinon, le message redevient visible et il est livré à quelqu'un d'autre. Cette deuxième livraison est une **nouvelle livraison**.

Le timeout est une estimation de la durée du traitement, et les deux façons de se tromper coûtent quelque chose. S'il est trop court, un consommateur lent mais vivant verra son message confié à un second consommateur alors qu'il travaille encore dessus. Deux consommateurs traitent alors le même message, et l'ack que le premier finira par envoyer peut échouer, être ignoré ou, selon le broker, réussir et supprimer le message pendant que le second consommateur y travaille encore. Dans ce dernier cas, un échec de la seconde tentative fait perdre le message. S'il est trop long, un vrai plantage laisse le message caché pendant toute cette période, et le délai se manifeste sous forme de latence pour l'acheteur.

Supposons que le consommateur Inventory de Parcelly termine normalement une réservation en bien moins d'une seconde, et que quelqu'un règle le timeout à 30 secondes. Ce chiffre est illustratif. Un hoquet de la base de données qui fait durer une réservation 40 secondes produira désormais une deuxième livraison du même `OrderPlaced`, et Inventory essaiera de réserver le stock deux fois. Rien n'est cassé dans le broker. Le timeout encodait une hypothèse, et l'hypothèse a cessé de tenir.

Certains systèmes permettent à un consommateur de prolonger le timeout pendant qu'il travaille. Cela aide pour les jobs légitimement longs, et cela ajoute une nouvelle chose à rater : un consommateur qui prolonge indéfiniment sur un message qu'il ne terminera jamais.

Les offsets dans un log

Dans un log, il n'y a pas d'ack par message. Le consommateur effectue à la place un **commit d'offset** : il enregistre « j'ai traité tout ce qui précède cette position ». Après un redémarrage ou une réaffectation, le consommateur suivant lit à partir de l'offset du dernier commit.

Les deux mêmes positions existent ici. Faites le commit de l'offset avant le traitement, et un plantage fait perdre les messages situés après le dernier message terminé, jusqu'à la position du commit. Faites le commit après le traitement, et un plantage répète les messages compris entre le dernier

commit et le plantage. Les vrais consommateurs font leurs commits par lots, si bien que le nombre de messages répétés après un plantage est un lot, et non un seul message.

Beaucoup de bibliothèques clientes savent faire le commit automatiquement. C'est pratique, et c'est aussi la façon la moins explicite de fixer votre sémantique de livraison, parce que, selon le client, le commit peut avoir lieu à un moment sans rapport avec la fin de votre traitement. Si le traitement est confié à un autre thread, le client peut faire le commit d'un travail qui ne s'est jamais terminé. À mon avis, un consommateur qui compte devrait faire ses commits explicitement, à un endroit que l'équipe peut montrer dans le code et sur lequel elle peut raisonner.

Groupes de consommateurs et rééquilibrage

Quand des consommateurs rejoignent ou quittent un groupe de consommateurs, le broker redistribue les partitions entre eux. C'est ce qu'on appelle le **rééquilibrage**. Pendant un rééquilibrage, une partition que lisait un consommateur est réaffectée à un autre, et le nouveau propriétaire démarre à l'offset du dernier commit.

Les groupes basés sur un log font, sous une autre forme, la même estimation que le visibility timeout dans une file d'attente. Un consommateur qui ne revient pas vers le broker dans un intervalle configuré est présumé mort, ses partitions sont réaffectées, et son travail sans commit est relu par le nouveau propriétaire. Un consommateur lent mais encore vivant est la cause ordinaire des doublons de rééquilibrage.

C'est de là que viennent les doublons du scénario d'ouverture. Prenons un consommateur qui a traité les messages 100 à 150 mais n'a fait le commit que jusqu'à 120. Un déploiement le redémarre, et la partition passe à un second consommateur, qui lit à partir de 120. Les messages 120 à 150 sont traités deux fois, et aucune partie du système n'a dysfonctionné. Le rééquilibrage est un événement normal qui peut se produire à chaque déploiement, à chaque montée en charge et à chaque plantage ; la nouvelle livraison est donc un fait quotidien, et non un accident rare.

Le silence a la cause inverse, en miroir. Si le client a fait le commit des offsets de messages que le consommateur avait seulement confiés à un émetteur en arrière-plan, et que le processus a été arrêté avant que l'émetteur

ait terminé, le propriétaire suivant démarre après ces messages. Ils ne sont jamais relus, et aucune ligne de log ne le signale. Le même rééquilibrage peut donc répéter le message d'un acheteur et faire disparaître celui d'un autre.

Il existe un danger voisin : pendant un court instant, l'ancien propriétaire peut ne pas encore savoir qu'il a perdu la partition et continuer à travailler dessus. Deux consommateurs peuvent alors traiter la même partition en même temps. Les clients bien conçus essaient d'empêcher cela, mais je ne concevrais pas un consommateur qui dépend d'une prévention parfaite.

Sémantiques de livraison

Une fois la mécanique en place, les trois sémantiques de livraison découlent de l'endroit où vous placez l'ack.

At-most-once

At-most-once signifie que chaque message est livré zéro ou une fois. On l'obtient en faisant l'ack avant le traitement, ou en ne faisant pas de retry sur une publication échouée. Un message peut être perdu, et il ne peut jamais être dupliqué.

C'est un choix raisonnable quand perdre un message coûte moins cher que gérer des doublons. Un ping périodique de « position actuelle » envoyé par un transporteur de Parcelly en est un exemple : le ping suivant rend le ping perdu sans importance. C'est le mauvais choix pour `PaymentCaptured`, où un message perdu signifie que de l'argent a circulé sans que personne l'ait enregistré.

At-least-once

At-least-once signifie que chaque message est livré une ou plusieurs fois. On l'obtient en faisant l'ack après le traitement et en faisant des retries jusqu'à ce qu'un ack arrive. Un message n'est jamais perdu tant que le broker le détient, et il peut être livré de nouveau.

C'est la sémantique sur laquelle le livre s'appuie. C'est la seule des trois où la défaillance est visible et récupérable : un doublon est quelque chose que vous pouvez détecter et ignorer, alors qu'un message perdu est quelque

chose que vous risquez de ne découvrir qu'au bout de plusieurs jours. Le prix est que chaque consommateur doit tolérer un **doublon**, c'est-à-dire un message traité ou livré plus d'une fois. Le Chapitre 7 explique comment.

«Exactly-once»

La livraison est un acte physique à travers un réseau peu fiable, et l'ack du destinataire peut se perdre comme n'importe quel autre message. Un émetteur qui n'a pas reçu d'ack a deux options : envoyer de nouveau, au risque d'un doublon, ou ne pas le faire, au risque d'une perte. Aucun protocole ne supprime ce choix pour les deux parties prises isolément.

Exactly-once est néanmoins un résultat réel et utile, et c'est un effet, non une garantie de livraison. Quand une livraison at-least-once rencontre un consommateur dont le traitement répété a le même effet qu'un traitement unique, le système combiné se comporte comme si chaque message avait été traité exactement une fois. La recette est une livraison at-least-once plus un **consommateur idempotent**.

Certains brokers annoncent un traitement exactly-once, et cette affirmation est légitime à l'intérieur d'une frontière. Un broker à log peut rendre atomique un cycle lecture-traitement-écriture quand l'entrée, la sortie et le commit d'offset vivent tous à l'intérieur de ce même broker. À l'intérieur de cette frontière, un plantage annule tout ensemble. La garantie exige aussi que chaque lecteur en aval ne lise que des données ayant fait l'objet d'un commit, et elle s'appuie sur une identité de producteur stable pour qu'une ancienne instance soit écartée après un redémarrage. La frontière s'arrête là où commencent vos effets de bord. Dès que le consommateur écrit dans une base de données relationnelle, appelle un fournisseur de paiement ou envoie un e-mail, la transaction du broker ne couvre plus l'effet, et le problème des doublons revient.

Pour cette raison, le choix par défaut du livre reste at-least-once plus des consommateurs idempotents. Cela fonctionne avec n'importe quel broker et n'importe quel effet de bord, cela ne dépend pas de l'activation d'une fonctionnalité, et cela survit à une migration vers un autre produit, opération que couvre le Chapitre 14. Si vous pouvez garder un pipeline entier à l'intérieur de la frontière transactionnelle d'un seul broker, la fonctionnalité peut réduire le travail, et je concevrais tout de même les consommateurs pour qu'ils n'en dépendent pas.

> **Remarque :** Les producteurs peuvent eux aussi créer des doublons, avant même qu'un consommateur n'intervienne. Un producteur qui atteint son timeout en attendant l'ack d'une publication publiera généralement de nouveau, et le broker peut alors détenir deux copies d'un même événement. Les deux copies portent le même `event_id`, à condition que le producteur l'ait attribué une seule fois à la création de l'événement et n'en ait pas fabriqué un nouveau pour le retry. Certains brokers dédupliquent les retries d'un même producteur, par numéro de séquence au sein d'une session de producteur ou par identifiant de déduplication dans une fenêtre de temps. Cela couvre un retry à l'intérieur d'une connexion. Cela ne couvre pas un processus qui redémarre et publie de nouveau, comme un relay d'outbox, et c'est pourquoi le consommateur vérifie quand même `event_id`. Le Chapitre 4 explique comment publier sans perdre d'événements, et le Chapitre 7 comment reconnaître les doublons.

Ce que le broker garantit réellement

Tout ce qui précède décrit le côté broker-consommateur. Le côté producteur-broker a ses propres limites, et elles dépendent de la configuration plus que du nom du produit.

Les acks du producteur

Quand un producteur publie, le broker peut faire l'ack à différents moments, et la manière utile de distinguer les réglages est de compter combien de nœuds détiennent le message au moment où le producteur entend « terminé ». Sans ack, le producteur n'apprend rien sur le stockage. Avec un ack du nœud récepteur, un nœud a accepté le message, qu'il ait ou non fait le flush sur disque. Avec un ack après réplication, le message existe sur un nombre requis de nœuds. Le flush sur disque est une question distincte, traitée plus bas.

Le choix du producteur compte pour ce qu'un plantage peut provoquer. Avec les deux premiers réglages, un nœud de broker qui meurt juste après l'ack peut emporter le message avec lui. Avec l'ack répliqué, la réponse ar-

rive plus tard et la fenêtre de perte est plus petite. Le prix est une latence plus élevée pour chaque publication et une dépendance à la disponibilité d'un nombre suffisant de nœuds pour mener l'écriture à bien.

Les brokers qui proposent des publisher confirms, dans la famille des files d'attente, et des niveaux d'ack, dans la famille des logs, décrivent la même idée. Vérifiez ce que signifie réellement, dans votre configuration, l'ack que vous attendez. À mon avis, c'est l'une des pages de documentation les plus précieuses qu'une équipe puisse lire une fois et consigner par écrit.

Réplication et fsync

La réplication protège contre la perte d'un nœud : le message existe sur plusieurs. Elle ne protège pas contre une défaillance simultanée de tout, et sa garantie dépend du nombre de copies qui devaient exister avant qu'on dise « terminé » au producteur. Vérifiez à la fois le niveau d'ack et le nombre minimal de répliques qu'il exige, parce que « toutes les répliques » peut signifier une seule quand les autres ont pris du retard. Vérifiez aussi si un failover peut promouvoir une réplique qui n'était pas à jour, car cela peut faire perdre des données dont on avait dit au producteur qu'elles étaient en sécurité.

Il existe un point voisin, et souvent mal compris, concernant le disque. Écrire dans un fichier place normalement les données dans le cache du système d'exploitation, et les données n'atteignent le disque physique que plus tard, à moins que le programme ne le force par un flush synchrone, couramment appelé fsync. Forcer le flush à chaque message est lent, si bien que beaucoup de brokers ne le font pas à chaque écriture. Ils comptent sur la réplication pour couvrir l'écart, et ils font le flush selon un calendrier. Que l'ack d'un broker donné implique ou non que les données soient sur disque dépend de sa configuration, et je ne le supposerais pas.

Je ne sais pas lequel de ces réglages convient à votre système, et cela dépend de ce que coûte un événement perdu. Un `PaymentCaptured` perdu et une « page vue » perdue sont des problèmes différents, et il serait raisonnable de les configurer différemment.

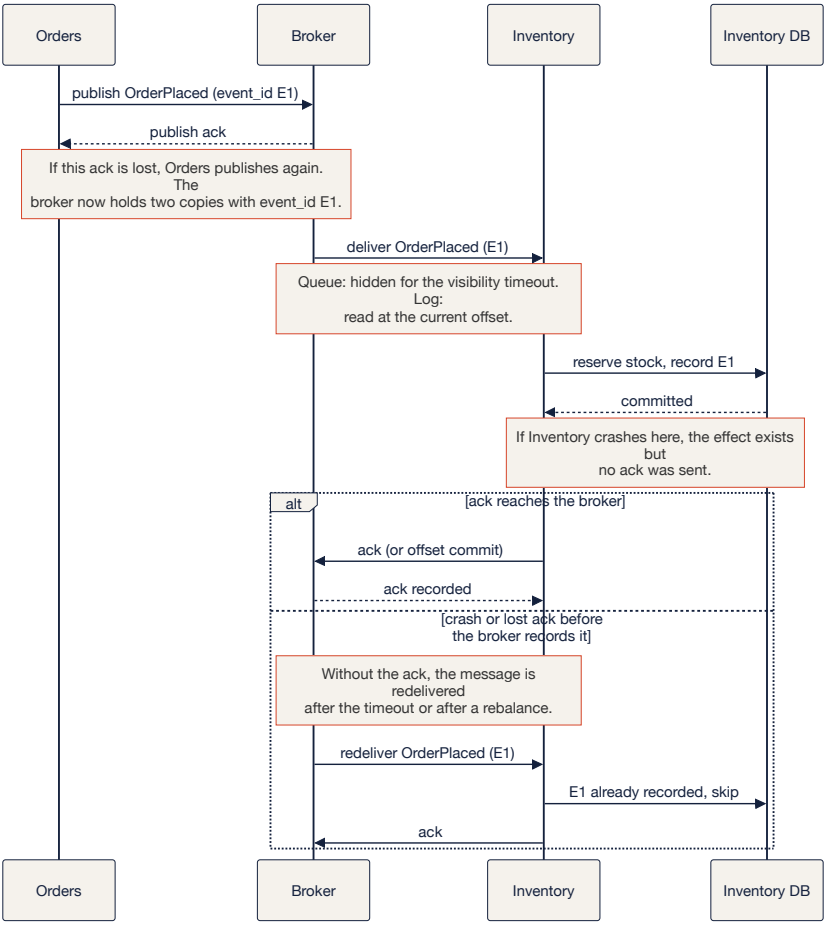
La limite de la garantie

Mettez tout cela ensemble et la frontière devient claire. Un broker peut promettre qu'un message qu'il a accepté sera conservé et proposé aux consommateurs, dans les limites de sa rétention ou de son expiration, et seulement avec la durabilité que permettent les réglages d'ack et de réplication ci-dessus. Il ne peut pas promettre qu'un message atteigne le broker en premier lieu, parce que cela dépend de ce qu'a fait le producteur quand sa publication a atteint son timeout. Il ne peut pas non plus promettre que l'effet d'un consommateur ait eu lieu exactement une fois.

Deux écarts restent entre les mains de l'application. Le premier se situe côté producteur : une commande enregistrée dans la base de données d'Orders et un événement `OrderPlaced` publié sur le broker sont deux écritures, et un plantage peut tomber entre les deux. Le Chapitre 1 a nommé cela le dual write, et le Chapitre 4 le supprime. Le second se situe côté consommateur, entre l'effet et l'ack. Le Chapitre 7 comble cet écart. Vus à travers les trois passages de relais qui ont ouvert ce chapitre, le broker énonce une sémantique des défaillances pour les deux premiers, et les deux écarts sont ce qui reste entre vos mains : le dual write en amont du premier passage, et l'écart entre l'effet et l'ack dans le troisième.

La vie d'un message

Le diagramme ci-dessous suit un message `OrderPlaced` depuis le service Orders jusqu'à un effet achevé dans Inventory. Repérez les deux endroits où une défaillance laisse l'émetteur sans réponse, signalés dans les notes : l'ack de publication perdu et l'ack du consommateur perdu. Tout ce que le chapitre a décrit se produit à l'intérieur de cette seule séquence.



La branche `else` du diagramme est l'enjeu du chapitre. La nouvelle livraison n'est pas une erreur et aucun composant ne s'est mal comporté. La seule chose qui empêche le stock d'être réservé deux fois est qu'Inventory a enregistré `E1` dans la même transaction que la réservation et l'a vérifié de nouveau lors de la deuxième livraison.

Voici à quoi ressemble, en pseudocode, un consommateur doté de cette propriété. Il utilise une **inbox**, une table des valeurs `event_id` traitées dans la propre base de données du consommateur, et place l'ack après le 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)
```

C'est la conception qui rend at-least-once sûr, et elle n'est introduite ici que pour montrer où se place l'ack. Le Chapitre 7 la développe, y compris les cas où la vérification de l'inbox seule ne suffit pas.

Compromis et coûts

Choisir une sémantique, c'est choisir quelle défaillance vous préférez gérer. Aucune des options ne supprime la défaillance, et chacune la déplace ailleurs.

At-least-once vous coûte un travail d'idempotence dans chaque consommateur. Chaque consommateur a besoin d'un moyen de reconnaître un travail répété, en général une table d'inbox dans sa propre base de données. Cela représente du stockage, une contrainte d'unicité sur le chemin critique et une politique de nettoyage. Pour une équipe qui a une douzaine de consommateurs, c'est un pattern à appliquer une douzaine de fois. Quand l'effet d'un consommateur est naturellement idempotent, comme fixer un statut à une valeur donnée, l'inbox peut être inutile, et je consignerais tout de même pourquoi.

At-most-once vous coûte la complétude. C'est simple et rapide, et ce n'est acceptable que si quelqu'un a décidé à l'avance qu'un message perdu coûte moins cher qu'un doublon. Cette décision doit être explicite et consi-

gnée. La forme habituelle de l'erreur est une équipe qui l'a choisi par défaut, à travers des réglages d'ack automatique, sans s'en rendre compte.

Le traitement transactionnel à l'intérieur d'un broker vous coûte la portabilité. Il peut réduire la gestion des doublons à l'intérieur de sa frontière, et il lie la conception aux fonctionnalités d'un seul produit, à l'obligation pour chaque participant de rester à l'intérieur de cette frontière, et à l'effort opérationnel nécessaire pour l'exploiter. Dès que vous passez à une base de données relationnelle ou à une API externe, vous avez de toute façon besoin d'idempotence. Je ne rejetterais pas la fonctionnalité, et je ne bâtirais pas la justesse sur elle seule.

Des acks de producteur forts coûtent de la latence et de la disponibilité. Attendre plusieurs répliques ralentit chaque publication et fait échouer l'écriture quand trop peu de nœuds sont joignables. Pour les événements qui ne doivent pas être perdus, ce compromis est généralement le bon. Pour une télémétrie à fort volume et à faible valeur, il peut ne pas l'être.

Les logs coûtent de la comptabilité côté consommateur, et les files d'attente coûtent l'historique. Si vous choisissez un log, la politique de commit d'offset et la gestion d'un message qui bloque une partition vous reviennent. Si vous choisissez une file d'attente, vous renoncez au replay. Quand vous avez besoin de retries indépendants pour chaque message et d'aucun historique, un log peut être plus de machinerie que le travail ne le mérite.

Quand ne pas appliquer les conseils de ce chapitre : un système avec un seul producteur, un seul consommateur et une tolérance aux pertes occasionnelles, comme un flux de métriques interne, n'a besoin ni d'inbox ni de réplication forte. Le coût des garanties doit être à la mesure du coût de la défaillance. L'erreur n'est pas de choisir une garantie plus faible ; l'erreur est de ne pas savoir que vous l'avez fait.

Erreurs courantes

- **Traiter « livré » comme « traité ».** C'est tentant parce que le tableau de bord montre les livraisons et les acks, et que ce sont ces chiffres qui ont l'air sains. Livré signifie que le broker a remis le message. Traité signifie que l'effet a fait l'objet d'un commit. Suivez les deux, et faites en sorte que votre ack suive l'effet.

- **Faire l'ack avant que le travail soit fait.** C'est tentant parce que cela évite les nouvelles livraisons et donne à la file d'attente un aspect propre. Cela transforme chaque plantage de consommateur en message perdu. Faites l'ack après le commit de l'effet, et gérez les doublons qui en résultent.
- **Laisser l'auto-commit des offsets activé par défaut.** C'est tentant parce que cela fonctionne en démo et ne demande aucun code. Selon le client, il peut faire le commit à un moment sans rapport avec votre traitement, si bien qu'un plantage peut faire perdre ou répéter une tranche arbitraire. Faites le commit explicitement, après l'effet.
- **Régler le visibility timeout au jugé.** C'est tentant parce que la valeur par défaut semble généralement convenir. Un timeout plus court qu'une exécution lente mais saine provoque un traitement concurrent de doublons. Fondez-le sur le temps de traitement légitime le plus lent que vous puissiez observer, plus une marge, et prolongez-le délibérément pour les jobs longs.
- **Croire que la fonctionnalité exactly-once d'un broker couvre votre base de données.** C'est tentant parce que la documentation emploie l'expression. La garantie est limitée à la frontière du broker lui-même. Tout ce qui est en dehors, comme une ligne dans votre base de données ou un e-mail, a besoin d'idempotence.
- **Supposer qu'un message publié est un message stocké.** C'est tentant parce que l'appel de publication a rendu la main. Selon le niveau d'ack, le message peut n'exister que dans la mémoire d'un seul nœud. Sachez ce qu'attend votre producteur, et consignez-le par écrit.
- **Supposer que le rééquilibrage est rare.** C'est tentant parce qu'il se produit sans bruit. Les déploiements, les changements d'échelle et les plantages peuvent tous en déclencher un, selon le protocole du groupe et les réglages d'appartenance, et chaque rééquilibrage peut répéter un lot de messages. Concevez comme si la nouvelle livraison avait lieu tous les jours, parce que le coût d'une erreur est un effet en double.

Points clés

- Un message passe par trois passages de relais, du producteur au broker, du broker au consommateur et du consommateur à l'effet, et les garanties d'un broker ne couvrent que les deux premiers.

- Les files d'attente suivent chaque message et le suppriment à l'ack, tandis que les logs conservent les messages indépendamment des lecteurs et laissent chaque groupe de consommateurs garder son propre offset, ce qui rend le replay possible.
- L'endroit où vous placez l'ack ou le commit d'offset décide de la sémantique : avant le traitement, on obtient at-most-once, après le traitement, at-least-once, et aucun placement n'évite à la fois la perte et les doublons.
- La nouvelle livraison est routinière, et non exceptionnelle, parce que les visibility timeouts expirent, que les consommateurs plantent et que les rééquilibrages (que les déploiements provoquent couramment) répètent des lots.
- Exactly-once est un effet construit à partir d'une livraison at-least-once et d'un consommateur idempotent, et le traitement exactly-once propre à un broker s'arrête à la frontière de ce broker.
- Un ack de producteur ne vaut que ce que valent les réglages de réplication et de flush qui le sous-tendent ; l'équipe doit donc savoir et consigner ce que signifie « publié » pour chaque événement important.
- Chaque option sémantique déplace la défaillance au lieu de la supprimer, et le bon choix est celui dont vous avez décidé que vous pouvez vous permettre de gérer la défaillance restante.

La sémantique de la messagerie vous dit ce que le broker fera et ne fera pas un jour ordinaire. Le Chapitre 3 se tourne vers les jours anormaux, et construit un modèle pragmatique de ce qui casse réellement : partitions réseau, plantages, consommateurs lents, horloges et humains.

Chapitre 3 — Un modèle de défaillances pragmatique

Imaginez Payments chez Parcelly en pleine capture. Orders a envoyé une commande `CapturePayment` à Payments, et Payments a appelé le fournisseur de paiement externe pour capturer l'argent. La connexion avec le fournisseur se tait. Trente secondes plus tard, le client HTTP de Payments abandonne et lève un timeout.

Payments doit maintenant répondre à une question à laquelle il ne peut pas répondre. Le fournisseur a-t-il capturé le paiement avant que la connexion ne tombe, ou n'a-t-il jamais reçu la requête ? Les deux histoires produisent exactement la même observation : un silence, puis un timeout. Si Payments fait un retry alors que la première capture avait réussi, l'acheteur est débité deux fois. Si Payments signale un échec alors que la première capture avait réussi, Orders annule une commande déjà payée.

Rien dans cette histoire n'est exotique. Il n'y a ni bug, ni rapport d'incident, ni ingénieur qui aurait commis une erreur. Un réseau a fait ce que font les réseaux, et le système n'avait aucun moyen de savoir dans lequel des deux mondes il vivait.

La plupart de ce qui tourne mal dans un système orienté événements a cette forme : une étape a eu lieu, ou non, et la preuve manque. Ce chapitre nomme les façons dont cette preuve vient à manquer, afin que vous puissiez concevoir délibérément pour chacune d'elles. La question à laquelle il répond est la suivante : qu'est-ce qui peut casser exactement dans un système comme celui de Parcelly, que verrez-vous quand cela arrivera, et à quel endroit de ce livre chaque défaillance est-elle traitée ?

Le problème, et pourquoi il est difficile

Un modèle de défaillances est une liste écrite des choses dont vous supposez qu'elles peuvent mal tourner, et de celles dont vous supposez qu'elles ne le peuvent pas. Chaque système en a un, que quelqu'un l'ait écrit ou non. La version non écrite contient généralement deux entrées : « le service plante » et « le réseau est lent ». Ces défaillances sont réelles, mais ce sont celles qui s'annoncent d'elles-mêmes.

Le cas simple est une défaillance que vous pouvez détecter, comme une connexion refusée ou une erreur de base de données. Chacune vous dit que l'étape ne s'est pas achevée.

Le cas difficile est la défaillance qui vous laisse dans l'incertitude. Un timeout ne dit pas « l'opération a échoué ». Il dit « j'ai cessé d'attendre ». L'opération a pu échouer, a pu réussir, ou peut être encore en cours et sur le point de réussir après que vous avez abandonné. Il suffit d'un timeout et d'un effet de bord à l'autre bout pour le ressentir.

C'est pourquoi l'approche évidente, écrire le code du happy path et ajouter un `try` et un retry autour de chaque appel, ne tient pas. Faire un retry est une décision sur l'avenir, prise sans connaître le passé. Si la tentative précédente a réussi, le retry est un doublon. Si vous renoncez au retry, un message peut être perdu. Aucun des deux choix n'est faux en soi ; chacun est faux dans un monde que vous ne pouviez pas distinguer de celui dans lequel vous étiez.

Il y a une seconde raison. Les défaillances se combinent : un déploiement redémarre un consommateur pendant qu'un broker fait un failover et qu'un producteur fait des retries. Les énumérer une à une reste le bon point de départ, mais la production les livre ensemble.

Le travail de ce chapitre est donc une classification. Pour chaque défaillance, je poserai les trois questions du Chapitre 1 : qu'est-ce qui peut défaillir à cette jonction, comment l'observeriez-vous, et quelle garantie y survit. Dans les entrées ci-dessous, « ce qui est réellement vrai » répond à la première, « ce que vous voyez » répond à la deuxième, et le pattern nommé à la fin de chaque entrée répond à la troisième. Prises ensemble, ces trois réponses forment la sémantique d'une défaillance, et le reste du livre ne cesse de la réclamer.

Dix défaillances qui comptent

La liste ci-dessous est celle par laquelle je commencerais pour un système construit à partir de services, de bases de données relationnelles et d'un broker de messages. Elle n'est pas complète, et aucune liste ne l'est.

Partitions réseau et incertitude des timeouts

Le scénario d'ouverture est le cas d'école. Deux parties sont reliées par un réseau, et le réseau perd, retarde ou réordonne le trafic. Une **partition réseau** en est la forme extrême, dans laquelle deux groupes de machines ne peuvent plus du tout se joindre. Un timeout en est la forme quotidienne, et il est bien plus courant.

Ce que chaque côté peut observer est asymétrique. L'appelant voit seulement qu'aucune réponse n'est arrivée. L'appelé a pu faire le travail et perdre la réponse, ou n'avoir jamais vu la requête. Il en va de même pour chaque saut chez Parcelly : un service qui appelle un transporteur externe, un producteur qui envoie au broker, un consommateur qui envoie son ack au broker.

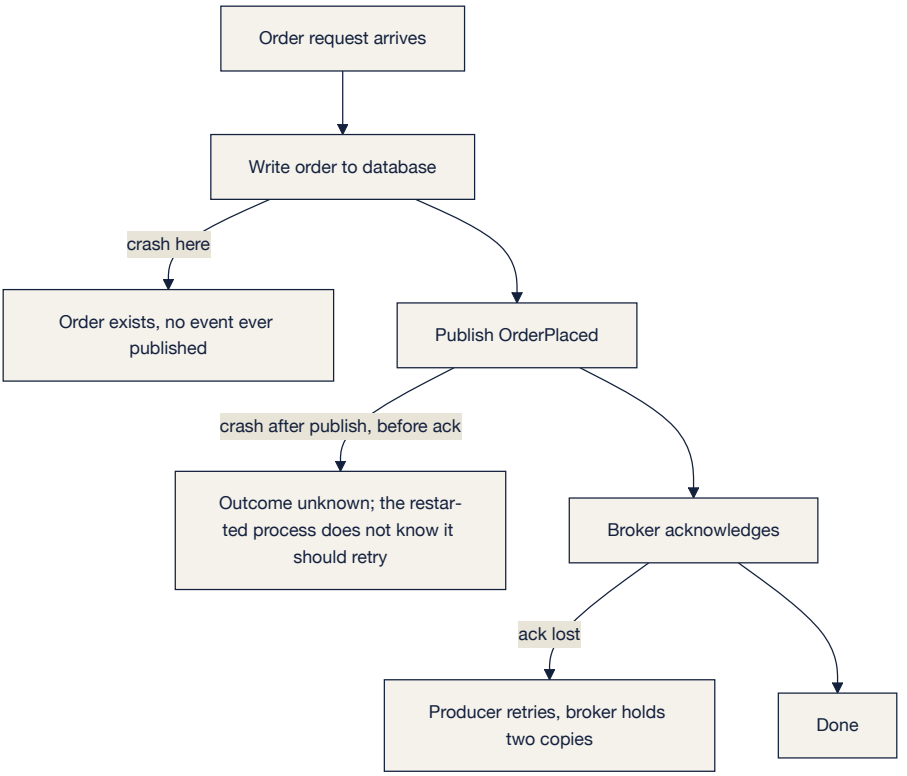
La seule réponse honnête est de rendre l'opération sûre à répéter. Si Payments envoie au fournisseur une clé d'idempotence stable avec chaque capture, et que le fournisseur traite une clé répétée comme la même requête, alors faire un retry après un timeout devient inoffensif. La conception cesse de dépendre de la réponse à « est-ce que cela a eu lieu ? », parce que les deux réponses mènent au même état final. Le Chapitre 6 explique quand et comment faire un retry, et le Chapitre 7 traite l'idempotence qui rend le retry sûr.

Certains systèmes externes n'offrent pas de telle clé. Vous demandez alors au système ce qui s'est passé avant d'agir de nouveau, ce qui est une étape de réconciliation, et le Chapitre 13 la traite.

Fenêtres de plantage du producteur

Prenons Orders qui traite une nouvelle commande. Il doit enregistrer la commande dans sa base de données et publier `OrderPlaced` sur le broker. Ce sont deux systèmes, et aucune transaction ne les englobe. C'est le dual write du Chapitre 1, et la partie intéressante est ce que laisse derrière elle chaque fenêtre de plantage.

Le processus peut planter entre les deux écritures, et l'ordre des écritures décide du dommage que vous subissez.



Le diagramme représente l'ordre « base de données d'abord » et montre trois fenêtres : un plantage avant la publication, un plantage après celle-ci avec un résultat inconnu, et un ack perdu. Si le commit en base de données vient en premier et que le processus meurt avant de publier, la commande existe et personne en aval n'en entend jamais parler. Payments n'autorise jamais, Inventory ne réserve jamais, et l'acheteur attend une confirmation qui ne vient jamais. Rien ne lève d'erreur. La commande reste là.

L'ordre inverse, que le diagramme ne représente pas, a sa propre fenêtre. Si vous publiez d'abord, un plantage avant le commit laisse un état pire : un événement qui annonce un fait qui n'est jamais devenu vrai. Les services en aval agissent sur une commande qui n'existe pas. Et dans l'un ou l'autre ordre, si l'ack du broker se perd au retour, le producteur fera un retry et le broker pourra détenir deux copies du même événement.

Ce que vous voyez de l'extérieur est discret : des commandes sans événements, des événements sans commandes, et des doublons occasionnels. Rien dans les logs ne dit « perdu ». Le Chapitre 4 présente le transactional outbox, qui supprime la fenêtre en faisant du changement d'état et de l'intention de publier une seule transaction locale. Le Chapitre 5 explique comment les lignes de l'outbox voyagent ensuite jusqu'au broker.

Failover du broker

Les brokers sont répliqués pour qu'une machine puisse mourir sans emporter les données avec elle. Quand la machine qui détient la copie leader d'une partition ou d'une file d'attente meurt, une autre prend le relais. C'est un **failover**, et c'est un comportement prévu, non un dysfonctionnement.

Ce qui compte, c'est ce qu'on a dit au producteur avant le failover. Qu'un message ayant reçu un ack quelques instants avant la mort d'un leader survive ou non dépend de la façon dont le broker a été configuré pour faire l'ack : pas du tout, après que le nœud récepteur l'a accepté (qu'il ait ou non fait le flush sur disque), ou après qu'un nombre requis de répliques le détient. Les réglages les plus forts coûtent de la latence et du débit. Les réglages les plus faibles peuvent perdre des messages ayant reçu un ack lors d'un failover. Cela diffère selon les familles de brokers et selon les configurations, je n'affirmerai donc pas ici un comportement précis pour un produit précis ; l'essentiel est que la garantie a une frontière, et que cette frontière est un réglage que quelqu'un a choisi.

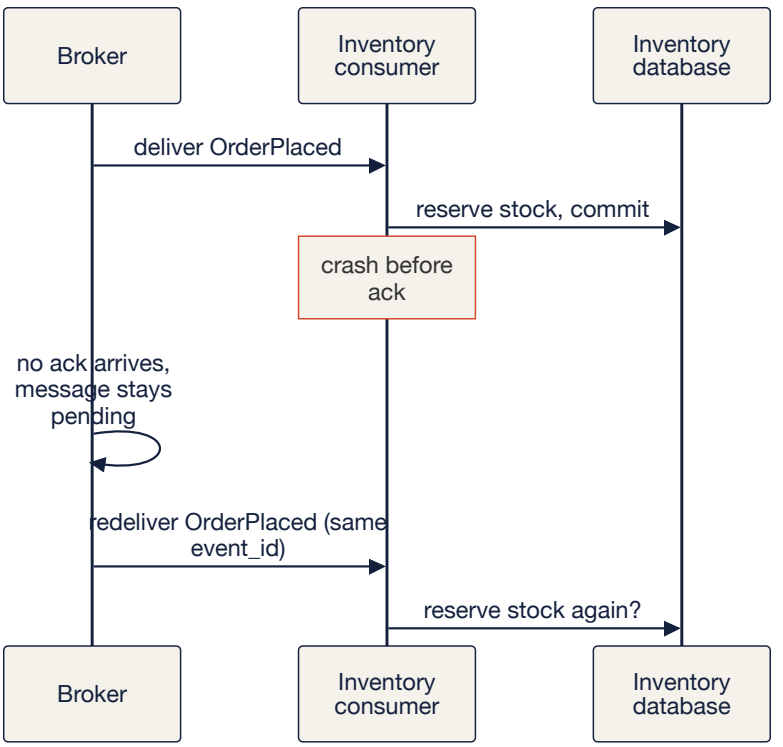
Pendant un failover, les producteurs voient aussi des erreurs et des timeouts, font des retries, et peuvent publier des doublons de messages que l'ancien leader avait déjà acceptés. Le Chapitre 2 décrit où s'arrêtent les garanties du broker. La règle ici est de traiter « ack reçu du broker » comme un indice fort et non comme une preuve.

Ce que vous voyez : une rafale d'erreurs de publication, une baisse du débit des consommateurs, puis une vague de nouvelles livraisons. Si un message a été perdu, vous ne voyez rien du tout, et c'est pourquoi la réconciliation du Chapitre 13 compte.

Plantage du consommateur: traité mais sans ack

Passons maintenant au côté consommateur. Inventory reçoit `OrderPlaced`, réserve le stock dans sa base de données et fait le commit. Puis le processus meurt avant d'avoir pu faire l'ack du message.

Du point de vue du broker, le message n'a jamais été terminé. Sur un broker à files d'attente, le broker remarque la connexion fermée ou le visibility timeout expire, et le message fait l'objet d'une nouvelle livraison. Sur un broker à log, l'offset du consommateur n'a jamais fait l'objet d'un commit, si bien que le consommateur suivant qui devient propriétaire de la partition repart de l'ancienne position et relit le message. Dans les deux familles, la deuxième livraison arrive pour un travail déjà fait.



La dernière ligne du diagramme est tout le problème. Un consommateur qui réserve le stock sans condition à chaque livraison le réserve deux fois. C'est la source ordinaire des doublons, et c'est la raison pour laquelle le choix par défaut du livre est une livraison at-least-once combinée à des consommateurs idempotents. Un consommateur peut planter à différents points, et chaque point a une conséquence différente :

```
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.
```

Voici une version compacte du handler du Chapitre 2, réduite pour montrer les points de plantage. L'effet du point de plantage A est nul. L'effet du point de plantage C est nul. Le cas intéressant est B, et seul un consommateur qui enregistre ce qu'il a traité, dans la même transaction que le travail, y survit. Le Chapitre 7 construit cela correctement.

Ce que vous voyez : des effets de bord en double, comme deux réservations, deux e-mails, deux écritures au Ledger, et aucune erreur nulle part, parce que chaque tentative de traitement a réussi.

Consommateurs lents et backpressure

Toutes les défaillances ne sont pas des plantages. Un consommateur simplement lent peut faire autant de dégâts, et c'est plus difficile à remarquer parce que rien n'est cassé.

Supposons que Notifications appelle un fournisseur d'e-mails qui se met à prendre 10 secondes par requête au lieu de 100 millisecondes. Notifications fonctionne toujours, mais chaque message occupe désormais un worker 100 fois plus longtemps. Les messages arrivent plus vite qu'ils ne sont terminés, si bien que le retard du consommateur, que le Chapitre 2 a défini, augmente régulièrement.

La lenteur déclenche ensuite des défaillances secondaires. Sur un broker à files d'attente, un message qui dépasse son visibility timeout réapparaît alors que le premier worker le traite encore, si bien qu'un second worker

commence le même message. Les doublons ajoutent de la charge à un système déjà lent. Sur un broker à log, un consommateur qui ne revient pas vers le broker à temps peut être déclaré mort et voir ses partitions réaffectées, ce qui déclenche la défaillance suivante de cette liste.

Le **backpressure** est la discipline qui consiste à borner le travail en cours à chaque étape pour que la charge reste finie. Dans un système qui passe par un broker, le broker absorbe la différence, et c'est l'avantage du découplage temporel, mais le tampon est fini, et il s'épuise de deux façons différentes. Quand le disque du broker se remplit, il peut brider ou bloquer les producteurs, ou cesser complètement de servir, et savoir lequel de ces comportements adopte votre broker est une chose à découvrir avant le jour où cela arrive. Quel qu'il soit, le producteur le voit, et c'est un vrai backpressure : bruyant et visible. Quand un consommateur prend plus de retard que la période de rétention d'un log, ou qu'un message en file d'attente survit à son expiration, le broker supprime des messages qui n'ont jamais été lus. Ce n'est pas un retard. C'est une perte silencieuse : aucune erreur n'est levée, aucun consommateur n'échoue, et les événements `PaymentCaptured` non lus ont disparu. Bien avant l'une ou l'autre limite, un backlog peut déjà violer une échéance métier, comme une autorisation de paiement qui expire avant d'être capturée.

Comme le tampon du broker ne fait que repousser le problème, la charge doit être bornée à plusieurs endroits à la fois. Un consommateur peut plafonner le nombre de messages en cours, ou le lot qu'il précharge. Il peut plafonner la concurrence de ses workers, pour qu'un fournisseur d'e-mails lent n'immobilise qu'un nombre fixe de workers et non tous. Il peut appliquer un rate limit à ses appels vers la dépendance lente, ou suspendre entièrement la consommation pendant que la dépendance se rétablit. Une file d'attente peut avoir une limite de longueur avec une politique de débordement explicite, pour que ce soit vous qui choisissiez ce qui est abandonné ou refusé au lieu de laisser le broker choisir.

Le côté producteur et la topologie fixent aussi des limites. Côté producteur, l'outbox absorbe un broker qui refuse les publications, parce que les lignes attendent dans la base de données jusqu'à ce que le relay puisse les livrer. Et sur un broker à log, le nombre de partitions plafonne le nombre de consommateurs d'un groupe qui peuvent travailler en parallèle, si bien qu'ajouter des instances cesse d'aider à partir d'un nombre fixé à l'avance.

Chacune de ces limites échange du débit contre de la stabilité, et chacune a besoin d'une métrique, faute de quoi vous ne saurez pas que c'est elle qui vous freine.

Ce que vous voyez : un retard et un âge des messages qui augmentent, un backlog qui grossit, des compteurs de nouvelles livraisons qui grimpent. Le Chapitre 6 traite le backoff, les limites de travail en cours et la suspension de la consommation, le Chapitre 8 traite ce qu'il faut faire des messages qui ne peuvent pas être traités à temps, et le Chapitre 13 traite les métriques de retard et d'âge qui vous préviennent tôt.

Les messages qui échouent à chaque fois, et les dépendances qui échouent pour chaque message

Jusqu'ici, chaque défaillance relevait du hasard : un plantage, une réponse perdue, un mauvais moment. Certaines défaillances sont déterministes. Un **poison message** est un message qui échoue à chaque tentative, quel que soit le moment. Supposons qu'un `OrderPlaced` arrive chez Inventory avec un SKU qui n'existe pas dans son catalogue. Inventory lève une erreur, le message fait l'objet d'une nouvelle livraison, l'erreur se répète, et rien dans la tentative suivante ne diffère de la précédente.

La cause peut aussi se trouver en dehors du message. Si le fournisseur de Payments est en panne, chaque capture échoue, et chaque message ressemble à un poison message tant que dure la panne. Les deux cas appellent des réponses opposées. Un message qui ne réussira jamais doit quitter rapidement le flux principal, vers une **dead-letter queue (DLQ)**, un endroit distinct où les messages qui ne peuvent pas être traités attendent un humain ou un job de réparation. Un message qui échoue parce qu'une dépendance est en panne ne doit pas du tout être envoyé vers la DLQ, parce qu'il réussirait après le rétablissement, et envoyer tout un backlog vers la DLQ pendant une panne transforme un problème temporaire en problème manuel.

Ce qui tourne mal entre les deux suit un schéma. Les retries sur un poison message consomment des tentatives et de la capacité sans aucune chance de succès. Sur un broker à log, ou sur une file d'attente qui préserve l'ordre, un seul message de ce type peut bloquer tout ce qui se trouve derrière lui. De nombreux consommateurs qui font des retries contre une même dépendance défaillante ajoutent de la charge à ce qui essaie de se

rétablir : c'est un retry storm. Et un bug de code peut échouer sans aucune erreur : le consommateur fait l'ack, rend la main et écrit la mauvaise valeur, si bien que rien dans le chemin de défaillance ne s'exécute jamais.

Ce que vous voyez : le même `event_id` qui apparaît encore et encore dans les logs, des compteurs de nouvelles livraisons qui ne baissent jamais, une partition bloquée, une DLQ qui grossit pendant une panne et, pour le bug silencieux, rien du tout jusqu'à ce qu'une vérification de réconciliation ne concorde pas. Le Chapitre 6 explique comment distinguer les défaillances qui valent la peine d'un retry, le Chapitre 8 traite les poison messages et le chemin vers la DLQ, et les chapitres 13 et 14 expliquent comment trouver et réparer les dégâts d'un bug qui n'a levé aucune erreur. Le prix de toute classification est qu'elle peut être fausse ; gardez donc la décision visible et réversible.

Horloges et horodatages auxquels on ne peut pas se fier

Chaque machine a une horloge, et deux horloges ne s'accordent jamais exactement. Elles dérivent, la synchronisation les corrige, et une correction peut faire reculer une horloge. Une machine virtuelle mise en pause se réveille avec une horloge fausse d'autant de temps qu'elle a dormi.

La conséquence pour les événements est précise. `occurred_at` est un fait sur le moment où quelque chose s'est produit selon une machine. Comparer le `occurred_at` d'un événement d'Orders avec celui d'un événement de Payments vous renseigne autant sur leurs horloges que sur l'ordre des événements. Si vous triez par horodatage pour décider quelle mise à jour l'emporte, un service dont l'horloge avance de quelques secondes gagnera tous les conflits qu'il devrait perdre.

Les timeouts ont un piège voisin. Une échéance exprimée en temps d'horloge, « cette autorisation est valable jusqu'à 14 h 00 », est comparée par une machine dont l'idée de 14 h 00 peut différer de celle de l'émetteur. Pour mesurer des durées sur une même machine, une horloge monotone, qui ne fait qu'avancer, est le bon outil. Pour ordonner entre machines, je n'utiliserais pas du tout le temps d'horloge.

Ce qui ordonne les événements de manière fiable, c'est la structure du système : sur un broker à log, une partition dont la clé est l'agrégat, ce qui donne un ordre total pour les événements de cette clé tant qu'il existe un

seul chemin de production par clé et que le nombre de partitions ne change pas, ou un numéro de version explicite sur l'agrégat, que les consommateurs peuvent comparer. Le Chapitre 9 traite l'ordre en profondeur. Considérez `occurred_at` comme une information pour les humains et pour l'analytique, et méfiez-vous-en comme entrée de la justesse.

Ce que vous voyez : des mises à jour appliquées dans le désordre, des séquences « impossibles » dans les logs où un effet semble précéder sa cause, et des bugs qui n'apparaissent que sur un seul hôte.

Déploiements et rééquilibrages

Un déploiement est une défaillance planifiée. Des instances sont arrêtées puis démarrées, et dans l'intervalle, le travail est interrompu en plein vol. Si votre processus de déploiement envoie un signal d'arrêt et que le consommateur se termine avec des messages en cours, vous avez créé délibérément le cas « traité mais sans ack », de façon répétée, à chaque release.

Le **rééquilibrage** en est la version propre à la famille des logs, et il peut se produire sans déploiement. Quand des membres rejoignent ou quittent un groupe de consommateurs, ou quand un membre est jugé non réactif, les partitions sont réaffectées. Pendant la passation, deux consommateurs peuvent brièvement se croire propriétaires de la même partition, et un consommateur qui était lent peut continuer à traiter des messages après que sa partition est partie ailleurs. L'équivalent dans la famille des files d'attente est celui de consommateurs concurrents dont les visibility timeouts expirent alors que le travail continue.

Le schéma des dégâts est une rafale de doublons au moment du déploiement, et parfois un court blocage pendant la réaffectation des partitions. Un déploiement qui parcourt toutes les instances en succession rapide peut déclencher plusieurs rééquilibrages d'affilée, et dans le pire des cas les consommateurs passent plus de temps à réaffecter le travail qu'à le faire. Deux mesures atténuent cela, au prix d'une configuration supplémentaire à comprendre : la réaffectation coopérative, où l'on ne retire que les partitions qui doivent bouger, et une identité de membre stable, pour qu'une instance redémarrée puisse récupérer ses anciennes partitions sans redistribution complète.

Faites en sorte que les consommateurs s'arrêtent proprement en terminant ou en libérant les messages en cours avant de se terminer, et supposez que le chemin d'arrêt propre échouera parfois, si bien que le consommateur idempotent reste la vraie défense. J'échelonnerais aussi les déploiements progressifs pour qu'ils ne déclenchent pas de réaffectations en boucle serrée. Ce que vous voyez : des doublons et des pics de retard qui coïncident avec les heures de release.

Erreurs humaines

La dernière catégorie est celle que la plupart des documents d'architecture omettent. Des personnes exploitent ces systèmes, et les personnes font des erreurs sous pression, généralement en réparant une défaillance antérieure.

Représentez-vous l'ingénieur d'astreinte de Parcelly qui trouve 40 000 dead letters et appuie sur le bouton pour faire un redrive de toutes. Si la cause a été corrigée, elles passent. Si elle ne l'a pas été, elles échouent de nouveau, immédiatement, et inondent le système que l'ingénieur essayait de soigner. Si certaines avaient déjà été traitées par un contournement manuel, ce sont des doublons d'un travail que quelqu'un a fait à la main.

D'autres formes sont tout aussi ordinaires. Quelqu'un modifie directement une ligne pour débloquer un client, en contournant l'outbox, si bien qu'aucun événement n'est publié et que les services en aval divergent. Quelqu'un déploie un producteur avec un payload modifié avant que les consommateurs puissent le lire. Quelqu'un purge une file d'attente dans le mauvais environnement. Voilà ce qui arrive quand les outils de réparation sont larges et que les garde-fous manquent.

Je traiterais les actions des opérateurs comme une partie du modèle de défaillances et je les concevrais comme n'importe quel autre composant : un redrive par petits lots avec un interrupteur de pause, des actions de réparation qui passent par le même chemin que les changements normaux pour que des événements soient émis, et chaque action manuelle consignée dans une piste d'audit. Le Chapitre 8 traite les dead letters et le redrive, le Chapitre 12 traite les changements de payload, et le Chapitre 13 traite la réconciliation, qui est la façon dont vous découvrez qu'une correction humaine a mal tourné.

Ce que vous voyez : un pic soudain juste après que quelqu'un a touché à quelque chose, et des données qui divergent entre services sans aucun message en échec pour l'expliquer.

Perdre tout un stockage

Chaque défaillance jusqu'ici supposait que les stockages survivent : la base de données conserve ce dont elle a fait le commit, et le broker conserve ce dont il a fait l'ack. La dernière entrée abandonne cette hypothèse. Un cluster ou une région est perdu. Un broker perd des données au-delà de ce que couvrait sa réplication. La base de données d'un consommateur est restaurée à partir d'une sauvegarde vieille de six heures. La base de données d'un producteur est restaurée à un point antérieur aux dernières centaines de commandes.

Ce n'est pas une seule défaillance, parce que les dégâts dépendent du stockage qui est revenu en arrière dans le temps. Si la base de données d'Inventory est restaurée, elle oublie les réservations faites après le point de restauration, et le broker peut ne plus détenir les événements qui permettraient de les recréer. Si la base de données d'Orders est restaurée, elle oublie des commandes sur lesquelles les services en aval ont déjà agi, et Payments détient des captures pour des commandes qu'Orders ne connaît plus. Aucun des deux côtés ne lève d'erreur. Chacun est cohérent avec lui-même et se trompe sur l'autre.

Une propriété aide. Comme l'inbox vit dans la même base de données que les effets, une restauration les ramène tous deux au même instant. Les messages traités avant le point de restauration sont toujours dans l'inbox, si bien qu'un replay les ignore, ce qui est correct puisque leurs effets ont survécu eux aussi. Les messages traités après le point de restauration sont ceux qui sont en danger, et faire un replay de ces messages est sûr : l'inbox ne s'en souvient plus, et leurs effets ont été perdus avec elle.

Les effets externes font exception. Une capture chez le fournisseur de paiement n'est pas annulée avec votre base de données ; ces appels ont donc besoin de leurs propres clés d'idempotence, faute de quoi le replay débite l'acheteur une seconde fois. La règle est de réconcilier avant de reprendre la consommation : comparer ce que croit chaque service, réparer les différences, et seulement ensuite laisser le flux reprendre.

Ce que vous voyez : rien, au début, et c'est là le problème. Les services sont en bonne santé et les tableaux de bord sont au vert, alors que les totaux divergent entre eux. Le Chapitre 13 traite la réconciliation qui détecte la divergence, et le Chapitre 14, Reprise, sinistres et migration de brokers, explique comment restaurer, faire un replay et reprendre. Concevoir pour ce cas coûte des sauvegardes réellement testées, une rétention qui dépasse votre temps de reprise, et la discipline de répéter une restauration avant d'en avoir besoin.

La carte des défaillances

Voici la même matière réunie en un seul endroit. Gardez-la à portée de main pendant vos revues de conception.

Défaillance	Ce que vous voyez	Ce qui est réellement vrai	Où c'est traité
Partition réseau ou timeout	L'appelant n'obtient aucune réponse	L'opération a pu s'exécuter ou non	Chapitre 6, Chapitre 7, Chapitre 13
Plantage du producteur entre l'écriture et la publication	Des commandes sans événements, ou des événements sans commandes	L'état et l'événement divergent	Chapitre 4, Chapitre 5
Ack du broker perdu	Des événements en double sur le broker	La première publication a réussi	Chapitre 7
Failover du broker	Des erreurs de publication, puis de nouvelles livraisons ; rarement, une perte silencieuse	La garantie dépend des réglages d'ack	Chapitre 2, Chapitre 13
Plantage du consommateur après le travail	Des effets de bord en double, aucune erreur	Travail fait, message toujours en attente	Chapitre 7
Consommateur lent	Retard et âge des messages croissants	Consommateur vivant, qui prend du retard	Chapitre 6, Chapitre 8, Chapitre 13
Rétention ou expiration pendant le retard	Rien ; les événements manquants n'arrivent jamais	Le broker a supprimé des messages non lus	Chapitre 8, Chapitre 13

Défaillance	Ce que vous voyez	Ce qui est réellement vrai	Où c'est traité
Poison message ou dépendance défaillante	Le même message livré de nouveau ; partition bloquée ; DLQ qui grossit pendant une panne	La défaillance est déterministe, ou la cause est en dehors du message	Chapitre 6, Chapitre 8
Bug du consommateur sans erreur	Rien ; les totaux divergent plus tard	Le handler a réussi et a écrit le mauvais résultat	Chapitre 13, Chapitre 14
Décalage d'horloge	Mises à jour dans le désordre, séquences impossibles	Les horodatages contredisent la causalité	Chapitre 9
Déploiement ou rééquilibrage	Doublons et retard au moment de la release	La propriété du travail a changé en plein vol	Chapitre 7, Chapitre 13
Erreur d'opérateur	Pic après une action manuelle ; divergence silencieuse	Une réparation a modifié l'état hors du chemin normal	Chapitre 8, Chapitre 13
Perdre tout un stockage	Rien au début ; les services divergent après une restauration	Un stockage est revenu en arrière dans le temps, ou ses données ont disparu	Chapitre 13, Chapitre 14

Deux choses ressortent. D'abord, les doublons apparaissent dans la colonne « Ce que vous voyez » pour trois causes différentes (ack perdu, plantage du consommateur, déploiement), et les nouvelles livraisons pour une quatrième. C'est pourquoi un seul pattern, le consommateur idempotent, mérite un chapitre entier. Ensuite, la dernière colonne revient sans cesse aux chapitres 13 et 14. Certaines défaillances ne peuvent être ni empêchées ni même détectées au moment où elles se produisent ; le système a donc aussi besoin d'un moyen de remarquer après coup que deux visions de la vérité divergent.

Mettre le modèle de défaillances par écrit

Un modèle de défaillances n'est utile que s'il est écrit, à jour et partagé. À mon avis, la pratique a plus de valeur que n'importe quelle entrée, parce que l'acte d'écrire impose la question « que supposons-nous ici ? », et que c'est dans les hypothèses que vivent les surprises.

Je le tiendrais sous la forme d'un court document dans le dépôt, à côté de la description de l'architecture, avec une entrée par défaillance et les mêmes champs à chaque fois. Les champs reprennent les trois questions : « where » et « we assume » disent ce qui peut défaillir, « observable as » dit comment vous le verriez, et « defense » dit quelle garantie y survit. Une entrée utilisable ressemble à ceci :

```
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
```

Les champs qui comptent sont les trois derniers. « Verified by » transforme l'entrée d'une croyance en une affirmation qui peut être testée, et le Chapitre 15 montre comment la tester. « Owner » signifie que quelqu'un est appelé. « Last reviewed » admet que le modèle se dégrade : des services sont ajoutés, des brokers sont remplacés, et une hypothèse vraie l'an dernier peut ne plus l'être aujourd'hui.

> **Remarque :** Le modèle doit aussi énoncer ce dont vous supposez que cela ne peut pas arriver. « Nous supposons que la base de données ne perd jamais une transaction dont elle a fait le commit » est une entrée légitime. Mieux vaut écrire l'hypothèse, et savoir ce que vous feriez si elle était fausse, que la laisser implicite.

Utilisez le modèle à trois endroits. En revue de conception, confrontez chaque nouveau flux à la liste et demandez-vous à quelles défaillances il est exposé. En revue d'incident, trouvez à quelle entrée appartient l'incident et, s'il n'y en a aucune, ajoutez-en une. Dans les tests, faites de chaque entrée un scénario. Une défaillance qui figure dans le modèle mais dans aucun test est un espoir.

Compromis et coûts

Écrire un modèle de défaillances coûte du temps, et l'inconfort de consigner ce que vous ne savez pas. Une petite équipe avec un service et une file d'attente n'a pas besoin de chaque entrée de cette liste. Commencez par les trois ou quatre défaillances qui feraient le plus mal.

Le modèle a aussi une frontière. Chaque défense de ce livre suppose que certaines défaillances sont hors périmètre : que la base de données est durable, que la réplication du broker fonctionne comme configurée. Ces hypothèses sont raisonnables, et chacune est un endroit où le modèle peut se tromper. Les énumérer vous permet de décider si elles sont acceptables, ce que vous ne pouvez pas faire tant qu'elles sont invisibles. Un bug dans votre propre code est différent. Il est dans le périmètre, comme défaillance à détecter et dont il faut se remettre, parce qu'il se produira, et le modèle doit dire comment vous le remarqueriez et comment vous répareriez ce qu'il a écrit.

Un modèle de 40 entrées ne sera pas lu, et un modèle de trois manquera des cas. Je viserais une liste qu'un nouveau membre de l'équipe peut lire d'une traite.

Concevoir pour chaque défaillance avec une force maximale n'est pas non plus l'objectif. Survivre à un failover du broker sans aucune perte coûte de la latence et de l'argent, et survivre à une double défaillance coûte davantage. Un e-mail de Notifications parfois envoyé deux fois ne justifie pas la même défense qu'une écriture du Ledger qui ne doit jamais être dupliquée. Le rôle du modèle est de vous laisser dépenser selon le risque. Si je ne peux pas nommer le coût métier d'une défaillance, j'hésiterais à payer un coût technique pour l'empêcher.

Le modèle ne rend pas le système fiable. Il rend vos hypothèses inspectables.

Erreurs courantes

- **Ne modéliser que les plantages.** Un plantage est tentant à modéliser parce qu'il est net et détectable. Les défaillances coûteuses sont les ambiguës, comme les timeouts et les consommateurs lents. Correc-

tion : pour chaque appel, écrivez ce que fait l'appelant quand aucune réponse n'arrive.

- **Traiter un timeout comme un échec.** Il semble naturel de lire « pas de réponse » comme « cela n'a pas eu lieu ». Correction : traitez-le comme inconnu, et concevez l'opération pour que la répéter soit sûr ou vérifiable.
- **Se fier à l'ack du broker comme à une preuve.** Le mot « acknowledged » sonne définitif. Correction : sachez ce que garantit réellement votre réglage d'ack, et réconciliez pour les cas où il ne le garantit pas.
- **Ordonner par horodatage.** `occurred_at` est là, dans l'enveloppe, et trier dessus tient en une ligne. Correction : ordonnez au sein d'une clé de partition par version d'agrégat, et gardez les horodatages pour les humains.
- **Ignorer les déploiements.** Les releases semblent relever de la routine, pas de la défaillance. Correction : comptez chaque déploiement comme un plantage de consommateur programmé, et testez l'arrêt pendant que des messages sont en cours.
- **Laisser les opérateurs de côté.** Les outils de réparation font l'objet de moins de revues que le code produit. Correction : mettez le redrive, le replay et les modifications manuelles dans le modèle, avec des limites de lot, des interrupteurs de pause et des pistes d'audit.
- **Traiter une base de données restaurée comme un nouveau départ.** Une restauration donne l'impression de clore l'incident. Correction : après toute restauration, réconciliez avec les autres services avant que la consommation ne reprenne, et vérifiez les effets externes qui n'ont pas été annulés.
- **Faire un retry sur une défaillance qui ne changera pas.** Les retries donnent une impression de diligence. Correction : séparez les défaillances qui dépendent du moment de celles qui dépendent du message, et envoyez tôt la seconde catégorie vers le chemin de la DLQ.
- **Écrire le modèle une seule fois.** Un document écrit pendant une phase de conception est un instantané. Correction : désignez un responsable, révisez-le après les incidents, et datez chaque entrée.

Points clés

- Un timeout signifie « j'ai cessé d'attendre », et non « cela a échoué » ; concevez donc des opérations qui restent correctes quelle que soit la façon dont l'inconnu se résout.
- Chaque jonction a ses propres fenêtres de plantage, et les dégâts sont silencieux. Un dual write laisse des commandes sans événements, ou des événements sur des commandes qui n'existent pas, et un consommateur qui plante après le commit mais avant l'ack reçoit un doublon à chaque nouvelle livraison, ce qui fait des consommateurs idempotents une exigence.
- L'ack d'un broker est un indice dont la frontière est fixée par les réglages de réplication et de flush ; connaissez cette frontière et prévoyez de réconcilier ce qui tombe en dehors.
- Les consommateurs lents et les déploiements provoquent des doublons et des blocages sans qu'aucun composant soit cassé ; bornez donc la charge, mesurez le retard et l'âge des messages, et arrêtez les consommateurs proprement.
- Un message qui échoue à chaque fois, ou une dépendance qui échoue pour chaque message, appelle une réponse différente de celle d'une défaillance due au hasard ; classez donc avant de faire un retry, et réservez l'envoi vers la DLQ à ce qui ne se rétablira pas.
- Les horodatages en temps d'horloge divergent d'une machine à l'autre ; ordonnez donc les événements au sein d'une clé de partition par version d'agrégat, et utilisez `occurred_at` pour les humains.
- Les opérateurs et les restaurations font partie du système : un stockage restauré est revenu en arrière dans le temps, réconciliez donc avant de reprendre la consommation, et tenez un modèle de défaillances écrit qui nomme les responsables, les tests et les dates de revue, pour que les réparations ne deviennent pas le prochain incident.

La première défaillance à supprimer est celle du producteur, où un changement d'état et son événement peuvent se dissocier sans que personne ne le remarque. Le Chapitre 4 ouvre la Partie II avec le transactional outbox, qui ferme cette fenêtre.

PARTIE II

Publier de façon fiable

Chapitre 4 — Le problème du dual write et le transactional outbox

Imaginez Parcelly quelques semaines après le lancement des promotions vendeurs. Une ingénieure du support ouvre un ticket qui dit, en entier : « L'acheteuse dit avoir terminé le paiement de la commande 81724, mais le vendeur n'a jamais reçu d'étiquette d'expédition. La commande apparaît comme passée. » L'ingénieure consulte la base de données d'Orders et trouve la ligne, statut `PLACED`, créée à 14:02:11. Puis elle consulte Payments, et il n'y a aucune autorisation pour cette commande. Inventory n'a pas de réservation non plus. Pour le reste de Parcelly, la commande n'existe pas.

Elle fouille les logs d'Orders autour de 14:02 et trouve un marqueur de déploiement. Le service a redémarré 200 millisecondes après avoir fait le commit de la commande et avant de publier `OrderPlaced`. Le code a fait exactement ce pour quoi il avait été écrit : enregistrer, puis publier. Aucune erreur n'a été levée, parce que le processus qui l'aurait levée avait déjà disparu. Supposons qu'une requête rapide révèle 11 autres commandes dans la même situation, toutes dans la même minute.

L'équipe corrige les 11 à la main et ajoute une note au runbook. Personne n'en est satisfait. Le code n'a rien de faux qu'un relecteur signalerait, et pourtant il peut perdre du travail chaque fois qu'un processus meurt au mauvais moment. Une machine plus grosse n'y changera rien, pas plus qu'un développeur plus soigneux. La question de ce chapitre est de savoir si un service peut modifier sa propre base de données et en informer le reste du système d'une manière qui ne puisse pas se désolidariser, sans transaction qui couvre à la fois la base de données et le broker.

Le problème, et pourquoi il est difficile

Le Chapitre 1 a défini le dual write : deux systèmes mis à jour sans transaction partagée. Orders doit écrire une ligne dans sa base de données relationnelle et publier un message sur le broker. La base de données garantit l'atomicité de la première écriture. Le broker garantit la durabilité, dans une certaine configuration, de la seconde. Rien ne garantit que les deux aient lieu, ou qu'aucune n'ait lieu.

Le Chapitre 3 a esquissé dans un diagramme les fenêtres de crash du producteur. Je veux ici les parcourir sous forme de tableau, parce que l'ordre des deux écritures est une décision de conception que la plupart des équipes prennent sans s'en apercevoir, et que chaque choix cache une défaillance qui attend son heure.

Chaque ordre casse

Prenez les deux séquences évidentes et demandez-vous, pour chaque point de défaillance possible, à quoi ressemble le monde ensuite. C'est le même exercice que d'écrire la sémantique des défaillances de la jointure entre la base de données et le broker : ce qui peut échouer, comment vous l'observeriez, et quelle garantie y survit. Le tableau ci-dessous est cet exercice pour le producteur.

Séquence	Point de défaillance	État de la base de données	État du broker	Dégâts
Commit, puis publication	Crash après le commit, avant la publication	Commande enregistrée	Aucun événement	Événement perdu. L'aval n'apprend jamais l'existence de la commande
Commit, puis publication	La publication échoue ou atteint un timeout, le processus survit	Commande enregistrée	Inconnu	L'appelant doit faire un retry ou abandonner. Abandonner perd l'événement
Commit, puis publication	Le broker accepte, l'ack est perdu	Commande enregistrée	Événement présent	Le retry publie un doublon

Séquence	Point de défaillance	État de la base de données	État du broker	Dégâts
Publication, puis commit	Crash après la publication, avant le commit	Aucune commande	Événement présent	Événement fantôme. L'aval agit sur un fait qui n'est jamais devenu vrai
Publication, puis commit	Le commit échoue (contrainte, deadlock, timeout)	Aucune commande	Événement présent	Même événement fantôme, avec cette fois une erreur visible côté producteur
Publication dans la transaction, commit en dernier	Le commit échoue après la publication	Aucune commande	Événement présent	Le bloc de transaction donne au code un faux sentiment de sécurité, et le broker n'en fait pas partie

La dernière ligne mérite un commentaire, parce que c'est celle qui paraît la plus sûre. De nombreux frameworks vous laissent publier depuis l'intérieur d'un bloc de transaction de base de données. On lit le code comme si la publication faisait partie de la transaction. Ce n'est pas le cas. L'appel au broker a lieu immédiatement et ne peut pas faire l'objet d'un rollback, et le commit a toujours lieu après. C'est l'ordre publication-puis-commit avec une meilleure indentation.

L'image miroir est le callback « after commit » que proposent de nombreux frameworks, un hook qui s'exécute une fois le commit de la transaction fait. Les équipes le croient souvent sûr, parce que la publication n'a lieu que pour des données dont le commit est fait. C'est la première ligne du tableau : commit, puis publication, avec l'intention conservée dans la mémoire du processus. Un crash entre le commit et le hook perd l'événement, et rien n'enregistre qu'il était dû.

Il existe aussi une quatrième forme que le tableau ne capture pas : un broker lent. Si la publication a lieu dans la transaction, la base de données garde des verrous et une connexion pendant qu'elle attend le broker. Un broker lent pendant une minute devient une base de données lente pendant une minute. Vous avez couplé la disponibilité de deux systèmes censés être indépendants.

Pourquoi le retry n'aide pas

La réparation tentante est une boucle de retry autour de la publication. Les retries aident pour la deuxième ligne du tableau, où le processus survit et peut faire un retry. Ils ne font rien pour la première ligne, où le processus a disparu, et ils créent le doublon de la troisième ligne. Le Chapitre 6 traite les retries en détail. Pour l'instant, remarquez qu'une boucle de retry vit dans la mémoire du processus, et que la défaillance qui nous intéresse est la disparition du processus.

La deuxième réparation tentante est un job périodique qui compare la base de données avec le broker et republie ce qui manque. Certaines équipes le construisent, et je pense que c'est un filet de sécurité raisonnable, auquel le Chapitre 13 revient sous le nom de réconciliation. Comme mécanisme principal, il pose deux problèmes. Le broker peut rarement être interrogé par clé métier, si bien que « ce qui manque » devient une supposition coûteuse. Et le job lui-même effectue le même dual write, si bien qu'il peut échouer de la même manière, seulement moins souvent et avec plus de confusion.

Pourquoi pas une transaction distribuée

Si le problème est celui de deux systèmes sans transaction partagée, la réponse classique est de leur en donner une. Le **commit en deux phases (2PC)**, standardisé pour les bases de données et les gestionnaires de transactions sous le nom de XA, coordonne plusieurs gestionnaires de ressources. Dans la première phase, chaque participant se prépare, c'est-à-dire qu'il promet de pouvoir faire le commit et conserve ses verrous. Dans la seconde phase, un coordinateur dit à chaque participant de faire le commit, ou le rollback si l'un d'eux n'a pas pu se préparer.

Sur le papier, cela ferme la fenêtre. En pratique, c'est rarement la réponse pour un broker et la base de données d'un service, pour des raisons surtout opérationnelles.

- **La prise en charge est inégale.** Beaucoup de brokers populaires n'implémentent aucune interface de gestionnaire de transactions. Là où l'un d'eux le fait, les garanties et la gestion des défaillances diffèrent de celles de la base de données, et vous héritez des deux jeux de règles.

- **Les participants gardent leurs verrous pendant que le coordinateur décide.** Si le coordinateur tombe entre les deux phases, les transactions préparées restent en doute. Les lignes restent verrouillées jusqu'à ce que quelqu'un, ou quelque chose, les résolve. La fenêtre de doute est exactement le genre de défaillance rare et longue que personne ne répète.
- **Le coordinateur devient un composant que vous devez exécuter, sécuriser et restaurer.** Il a besoin de ses propres logs durables, et d'une procédure de reprise qui doit être juste.
- **Les indisponibilités se multiplient.** Une transaction qui exige que la base de données, le broker et le coordinateur soient sains ensemble n'est disponible que lorsque les trois le sont. L'indépendance qui rendait le broker attrayant a disparu.
- **Le débit et la latence en pâtissent.** Deux allers-retours, des écritures forcées chez chaque participant et des verrous tenus pendant des appels réseau, tout cela s'additionne.
- **Cela ne s'étend pas au monde extérieur.** Un fournisseur d'e-mail ou un fournisseur de paiement ne rejoindra pas votre transaction. Même une équipe qui adopte le 2PC pour le broker a encore besoin d'une autre réponse pour chaque appel externe.

Je n'exclurais pas le 2PC dans tous les cas. Une équipe qui contrôle les deux ressources, exploite un gestionnaire de transactions mature et possède les compétences opérationnelles pour résoudre les transactions en doute peut le faire fonctionner. Pour la plupart des équipes qui construisent une marketplace à partir de services, chacun avec sa propre base de données et un broker qui n'expose aucune interface XA, le coût est élevé et le bénéfice étroit. Ce qu'elles veulent est plus petit qu'une transaction distribuée : elles veulent que le changement d'état et l'intention de publier fassent l'objet d'un commit ensemble. C'est un problème local, et les transactions locales sont bien comprises.

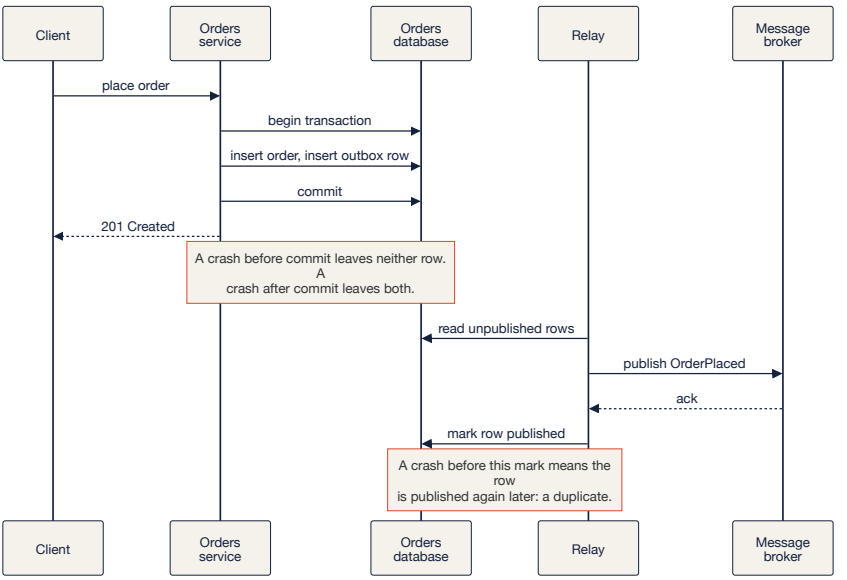
Le transactional outbox

L'idée est de transformer le dual write en une seule écriture suivie d'un déplacement ultérieur et répétable. Au lieu de publier sur le broker pendant la requête, le service écrit une ligne décrivant l'événement dans une table

de sa propre base de données, dans la même transaction locale que le changement d'état. Cette table est le **transactional outbox**. À partir d'ici, le livre l'appelle l'outbox.

Un processus séparé, le **relay**, lit les lignes de l'outbox et les publie sur le broker. Le Chapitre 5 est consacré au relay. Dans ce chapitre, nous le traitons comme une fonction qui finit par livrer chaque ligne au moins une fois, et nous regardons ce que cela nous apporte.

Le raisonnement est court. Soit la commande et sa ligne d'outbox font l'objet d'un commit ensemble, soit aucune des deux, parce qu'elles vivent dans une seule base de données et une seule transaction. Une fois que la ligne existe, l'intention de publier est durable. Un crash après le commit ne perd plus l'événement, parce que la ligne est toujours là quand le service, ou son remplaçant, redémarre. Un crash avant le commit ne laisse ni commande ni ligne, donc aucun événement fantôme n'existe.



Regardez les deux notes du diagramme. La première montre la fenêtre que nous avons fermée. La seconde montre la fenêtre que nous avons gardée : le relay peut publier et tomber avant d'avoir enregistré qu'il l'a fait. L'outbox ne supprime pas les doublons, il déplace la défaillance vers un endroit où elle est bornée et prise en charge. J'y reviens plus bas.

La table

L'outbox suit la forme qu'utilise le reste du livre. Il porte chaque champ de l'enveloppe, si bien que le relay peut construire l'événement sans rien demander au service producteur. Ce chapitre ajoute quatre colonnes au-delà de l'enveloppe, chacune pour une raison.

```
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
```

Quelques-uns de ces choix demandent à être défendus.

- **`id` et `event_id` sont deux choses différentes.** L'`event_id` est un UUID que les consommateurs utiliseront comme clé d'idempotence, et il ne doit pas changer quand l'événement est publié deux fois. L'`id` est un numéro de séquence attribué par la base de données, que le relay peut utiliser pour trouver les lignes. Ce n'est pas une affirmation sur l'ordre des commits, et la section suivante montre pourquoi.
- **`aggregate_version` est la source de vérité de l'ordre.** C'est la version de l'agrégat, incrémentée dans la même transaction que celle qui écrit la ligne. Pour les commandes de Parcelly, c'est l'`order_version`. Les consommateurs qui ont besoin de l'ordre par agrégat comparent ce numéro, pas l'`id` ni l'ordre d'arrivée. Le Chapitre 9 montre comment un consommateur l'utilise. L'enveloppe standard n'a pas de champ pour lui, donc quand le relay construit l'événement, il copie la colonne

dans le payload sous le nom `order_version`, et c'est là que les consommateurs la lisent.

- **`aggregate_type` et `partition_key` sont séparés.** La `partition_key` est ce que le broker utilise pour l'ordre, et c'est l'identifiant de l'agrégat, comme `order-81724`. L'`aggregate_type` dit au relay, et à un humain qui lit la table, à quel genre de chose la clé se rapporte, et c'est un moyen commode de router les lignes vers différents topics.
- **`created_at` est distinct d'`occurred_at`.** Le premier indique quand la ligne a été écrite, le second quand le fait s'est produit. En général, les deux sont à quelques millisecondes l'un de l'autre. Ils divergent quand un service enregistre un fait qu'il a appris tardivement, comme un scan de transporteur qui arrive après l'événement. Une métrique de retard du relay doit utiliser `created_at`, parce que le retard est une propriété de la table, pas du métier. Laissez la base de données le remplir via une valeur par défaut de colonne au lieu d'appeler `now()` dans le code applicatif, afin que l'horloge soit une seule horloge. Cela reste l'heure d'insertion, pas l'heure du commit : une transaction longue fait paraître la ligne plus ancienne que ce que le relay en voit, et cette part du retard n'est pas la faute du relay.
- **`headers` porte les préoccupations de transport.** Le contexte de trace et les indications de routage vont ici, afin que le payload reste le fait métier. Une équipe qui n'arrive pas à décider si un champ est une donnée ou une indication de transport devrait lire le Chapitre 11 avant de l'ajouter au payload.
- **`published_at` peut être nul, et nul a un sens.** Une ligne sans `published_at` est une ligne que le relay doit encore au broker. Certaines équipes utilisent plutôt une colonne `status`, avec des valeurs comme `PENDING`, `PUBLISHED` et `FAILED`. Les deux conceptions portent la même information. Je préfère l'horodatage parce qu'il vous dit aussi quand, ce qui est utile lorsque vous demandez quel retard avait le relay, et parce qu'il y a moins à mal faire. Si vous avez besoin d'un moyen de mettre en parking les lignes qui échouent de façon permanente, ajoutez une colonne `failed_at` à ce moment-là, pas avant.

Ce moment arrive plus tôt qu'on ne le croit. Une ligne que le broker n'acceptera jamais, comme un payload trop volumineux ou un payload que le sérialiseur rejette, bloque sa clé, et dans le relay le plus simple la table en-

tière, jusqu'à ce que quelqu'un la répare. La première fois que cela arrive, il vous faut une colonne `failed_at` (ou un statut `FAILED`, si vous avez choisi les colonnes de statut) et une alerte sur les parked rows. Le Chapitre 5 montre comment le relay s'en sert.

Un index sur les lignes non publiées, dans l'ordre d'insertion, est le seul index dont le relay a besoin. Sans lui, le relay parcourt une table qui grandit indéfiniment. Le Chapitre 5 traite du nettoyage.

Écrire dans la même transaction

Le code producteur est court, et cette brièveté est tout l'intérêt. La fonction suivante représente tout le changement apporté au chemin de requête d'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
```

Trois détails comptent plus qu'il n'y paraît.

Premièrement, rien dans la fonction ne parle au broker. Une panne du broker ne fait pas échouer la prise de commande, et un broker qui met une minute à répondre ne maintient pas une transaction de base de données

ouverte. C'est un vrai gain de disponibilité : Orders peut accepter des commandes pendant que le broker est en panne, et l'outbox grossit jusqu'à ce que le relay puisse le vider.

Deuxièmement, l'`event_id` est créé avant le commit et stocké dans la ligne. Chaque republication ultérieure de cet événement porte le même identifiant. C'est la propriété dont dépendent les consommateurs, et c'est pourquoi générer l'identifiant dans le relay, au moment de la publication, serait une erreur.

Troisièmement, le payload est construit dans la transaction, à partir de l'état que la transaction a écrit. Si le payload était construit plus tard en relisant la commande, une mise à jour concurrente pourrait le modifier, et l'événement décrirait un fait qui n'existait pas à ce moment-là. Un événement est un instantané de ce qui s'est passé, et l'instantané est pris au même endroit que le changement.

Le pseudocode suppose que l'insertion de la commande et l'insertion dans l'outbox utilisent la même connexion à la base de données et la même transaction. Cette hypothèse est cassée par accident plus souvent qu'on ne le croit. Une couche d'accès aux données qui ouvre une seconde connexion pour l'outbox, ou qui fait un auto-commit de l'insertion dans l'outbox, rétablit silencieusement le dual write, désormais entre deux tables. Le test correspondant a sa place dans votre suite : forcez une défaillance après l'insertion de la commande et vérifiez qu'aucune ligne d'outbox n'existe.

Ordre des commits, ordre des `id` et verrou de l'agrégat

Une nouvelle commande n'a pas de concurrent, donc `placeOrder` n'a besoin d'aucun verrou. Chaque modification ultérieure d'une commande existante en a besoin, et le verrou doit venir en premier. Voici `cancelOrder`, qui émet `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
```

Le verrou est ce qui rend la version digne de confiance, et c'est aussi la seule chose qui donne un sens à l'ordre des `id` pour un agrégat. Sans lui, prenez deux transactions sur la même commande. La transaction B insère sa ligne d'outbox et obtient l'`id` 101, puis se bloque sur la ligne de commande que tient la transaction A. A insère l'`id` 102 et fait son commit, et B fait le sien ensuite. L'ordre des commits est A puis B, et l'ordre des `id` est B puis A. Un relay qui voit les deux lignes dans un même polling les publie à l'envers. Avec le verrou pris en premier, B ne peut pas insérer avant que A ait fait son commit, donc les identifiants d'un agrégat suivent son ordre de commit.

Même alors, je ne ferais pas dépendre les consommateurs de l'`id`. Les lignes d'agrégats différents s'entrelacent, un relay parallèle réordonne d'un lot à l'autre, et personne en dehors de la base de données d'Orders ne peut voir l'`id`. L'`aggregate_version` voyage avec l'événement, sous le nom `order_version` dans le payload, et dit dans quel ordre les faits sont arrivés à cette commande. Le Chapitre 5 revient sur ce qu'un relay peut et ne peut pas préserver, et le Chapitre 9 montre le côté consommateur.

Ce que le relay doit

Le contrat du relay est délibérément réduit. Il sélectionne les lignes non publiées, publie chacune sur le broker, et la marque comme publiée une fois que le broker confirme. La version la plus simple figure ci-dessous, et le Chapitre 5 traite des préoccupations de production qu'elle ignore, comme le traitement par lots, le verrouillage, l'ordre et le 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())
```

Le commentaire à l'intérieur de la boucle est toute la garantie. La ligne n'est marquée qu'après la confirmation du broker, c'est pourquoi l'appel attend l'ack et pourquoi une défaillance arrête le lot. Un crash entre la publication et le marquage laisse la ligne non publiée. Au passage suivant, le relay la publie à nouveau. L'alternative, marquer d'abord et publier ensuite, ferait de la défaillance un événement perdu au lieu d'un doublon. L'outbox choisit délibérément la défaillance la plus sûre.

Ce que l'outbox garantit

L'outbox vous donne trois choses et vous en retire une.

- **L'atomicité entre l'état et l'intention.** Le changement d'état et la décision de publier font l'objet d'un commit ensemble. Il n'existe pas de commande dont le commit est fait sans ligne d'outbox, ni de ligne d'outbox sans commande dont le commit est fait.
- **La durabilité de l'intention.** Une fois le commit de la transaction fait, l'événement est aussi durable que la base de données. Le relay peut être en panne, le broker peut être en panne, le producteur peut être remplacé, et la ligne attend.
- **Une publication at-least-once.** Chaque ligne dont le commit est fait finit par être publiée, à condition que le relay et le broker finissent par se rétablir, que le relay trouve les lignes par leur état et non par un curseur sur l'id, que le nettoyage ne purge que les lignes publiées, et qu'une ligne que le broker n'acceptera jamais soit mise en parking et

fasse l'objet d'une alerte au lieu de rester bloquante indéfiniment (tout cela relève du Chapitre 5). Une ligne peut être publiée plus d'une fois.

- **Non garanti : une publication exactly-once.** Le relay ne peut pas publier atomiquement sur le broker et marquer une ligne dans la base de données, pour la même raison que l'application ne le pouvait pas. La fenêtre s'est déplacée du chemin de requête vers le relay, où elle est plus petite, observable et réparable.

Je veux être prudent avec ce résumé. L'outbox n'affirme pas que les événements ne sont jamais perdus. Il affirme que la fenêtre de perte côté producteur du dual write est fermée, à condition que l'application écrive via l'outbox pour chaque changement d'état qui doit produire un événement. Un développeur qui met à jour la table à la main, ou par un chemin qui contourne l'outbox, rouvre la fenêtre. Le Chapitre 3 l'a classé parmi les erreurs humaines, et c'est la façon la plus courante dont une équipe dotée d'un outbox perd encore des événements.

Le doublon devient le problème du consommateur

Parce que la publication est at-least-once, chaque consommateur d'un événement publié par un outbox doit tolérer les doublons. Ce n'est pas un défaut à éliminer par l'ingénierie, c'est le prix de la garantie, et il vaut la peine de dire clairement qui le paie.

L'équipe productrice fait la partie la plus légère. Elle ajoute une table, écrit une ligne et exploite un relay. Les équipes consommatrices font la partie la plus dure, parce qu'elles doivent reconnaître un `event_id` déjà vu et éviter d'en répéter l'effet. Payments ne doit pas autoriser deux fois. Notifications ne doit pas envoyer deux fois le même e-mail. Ledger ne doit pas passer deux fois la même écriture.

> **Avertissement :** Adopter un outbox sans consommateurs idempotents échange un problème d'événement perdu contre un problème d'effet en double. Pour certains événements, comme un double débit, le second problème est le plus coûteux. Livrez les deux ensemble, et voyez le Chapitre 7 pour savoir comment un consommateur reconnaît un travail répété.

Je dirais aussi ceci aux équipes productrices, en termes simples : l'outbox est un contrat avec vos consommateurs. L'`event_id` doit rester stable pour chaque republication du même fait, et le contrat doit être écrit dans la documentation des événements, pas laissé implicite.

Un bref parcours des défaillances

Pour confronter la conception aux fenêtres du Chapitre 3, faites repasser Orders par les crashs.

1. Le processus meurt avant que la transaction ne fasse son commit. Pas de commande, pas de ligne d'outbox, pas d'événement. Le client voit une erreur ou un timeout et peut faire un retry de la requête, ce qui est un problème distinct pour la couche API, traité par la discussion sur l'idempotence au Chapitre 7.
2. Le processus meurt juste après le commit. La commande et la ligne existent toutes les deux. Le relay publie la ligne à son passage suivant. Le client n'a peut-être pas reçu sa réponse, mais le système est cohérent.
3. Le relay meurt après la publication et avant le marquage. L'événement est dans le broker et la ligne n'est pas publiée. Le passage suivant la publie à nouveau. Les consommateurs voient un doublon avec le même `event_id`.
4. Le broker est indisponible. Le relay ne parvient pas à publier, laisse les lignes non publiées et fait un retry plus tard. Orders continue d'accepter les requêtes. Le retard augmente, et une alerte devrait suivre.
5. Le service perd sa connexion pendant le commit et ne connaît pas le résultat. Soit la commande et sa ligne existent, soit aucune des deux, et le service ne peut pas savoir laquelle. Il ne doit pas deviner. La réponse sûre est de laisser le retry du client porter une clé d'idempotence au niveau de l'API, afin que, quel que soit l'état de la base de données, le retry converge vers une seule commande et un seul événement (Chapitre 7).

Chaque cas se termine soit par la cohérence, soit par un doublon, et aucun ne se termine par un événement fantôme ou une perte silencieuse. C'est tout ce que l'outbox affirme, et cela suffit à le justifier.

Un doublon se situe en amont de l'outbox, et aucune défense en aval ne le voit. Quand un client fait un retry après une réponse perdue et que l'API n'a pas de clé à elle, Orders crée une seconde commande et un second `OrderPlaced` avec un `event_id` neuf. L'inbox de chaque consommateur voit alors deux événements différents, parce qu'ils le sont. L'outbox a fait son travail deux fois. La correction a sa place au niveau de l'API, avec l'approche `Idempotency-Key` du Chapitre 7.

Alternatives et voisins

L'outbox n'est pas le seul moyen d'arrêter le dual write. Deux autres conceptions méritent une comparaison honnête, car une équipe pourrait raisonnablement préférer l'une ou l'autre.

Listen to yourself

Dans le pattern **listen-to-yourself**, le service n'écrit pas d'abord dans sa base de données. Il publie l'événement sur le broker, puis consomme son propre événement et applique le changement d'état dans le handler. Le broker devient l'écriture unique, et la base de données devient une projection de ce que le service a entendu.

Cela supprime le dual write, puisqu'il n'y a qu'une écriture. Cela change aussi la forme du service. La requête du client se termine avant que l'état n'existe, donc l'appelant ne peut pas lire sa propre écriture à moins que le système n'offre un moyen d'attendre. La validation doit avoir lieu avant la publication, puisqu'un événement, une fois publié, est un fait, et qu'un événement invalide ne peut pas être rejeté après coup sans compensation. Et le handler qui applique l'événement doit lui-même être idempotent, ce qui ramène le même fardeau côté consommateur.

J'envisagerais cela pour des services qui pensent déjà en événements et peuvent tolérer la cohérence à terme dans leurs propres lectures. Pour Orders, où le navigateur de l'acheteur s'attend à voir la commande juste après le clic sur le bouton, je ne le ferais pas. L'écart de read-your-own-write est un problème produit que l'équipe finirait par résoudre avec des contournements.

Event sourcing

Avec l'**event sourcing**, les événements sont la source de vérité. Le service stocke la séquence des événements par agrégat, et l'état courant en est dérivé. L'écriture dans l'event store est le changement d'état, il n'y a donc rien à maintenir cohérent avec un événement séparé. La publication sur le broker lit ensuite depuis l'event store, ce qui est un relay sous un autre nom.

L'event sourcing résout le dual write en changeant ce qu'est la base de données. Il vous donne aussi un historique complet et la capacité de reconstruire les projections. Il coûte un modèle de programmation différent, une évolution des schémas des événements stockés que vous ne pouvez jamais réécrire à la légère, et une courbe d'apprentissage pour toute l'équipe. C'est un engagement architectural important et je ne l'adopterais pas uniquement pour éviter un outbox. Si vous l'utilisez déjà, vous avez quand même besoin d'un relay, et tout ce que dit ce chapitre sur la publication at-least-once s'applique.

Comparer les options

Option	Ce qu'elle supprime	Ce qu'elle coûte	Convient le mieux quand
Commit, puis publication	Rien	Perte silencieuse en cas de crash	Un événement perdu est acceptable, comme des métriques best-effort
2PC / XA	La fenêtre, en théorie	Verrous, coordinateur, prise en charge inégale	Vous contrôlez les deux ressources et assurez vous-même la reprise
Transactional outbox	La fenêtre côté producteur	Une table de plus, un relay, des doublons	L'état vit dans une base de données relationnelle et les événements ne doivent pas être perdus
Listen to yourself	La seconde écriture	Cohérence à terme dans ses propres lectures, validation anticipée	Le service est déjà orienté événements
Event sourcing	L'écriture d'état séparée	Un modèle d'état différent	L'historique est lui-même une exigence

Pour un service qui stocke déjà son état dans une base de données relationnelle et doit en informer les autres, l'outbox est le plus petit changement. C'est pourquoi la plus grande partie de ce livre le suppose.

Compromis et coûts

L'outbox est simple à décrire et n'est pas gratuit à exploiter. Voici les coûts que je mettrais sous les yeux d'une équipe avant qu'elle ne s'engage.

Plus de pièces mobiles. Il y a une table, un processus relay et un retard à surveiller. Quand quelque chose casse, il existe un nouvel endroit où regarder. Le relay est un composant qui a un responsable, et il a besoin de déploiements, d'alertes et d'un runbook. Le Chapitre 5 donne le détail.

Une charge supplémentaire sur la base de données. Chaque événement est une insertion dans la même transaction que l'écriture métier, et chaque publication est une lecture et une mise à jour. Les transactions s'allongent un peu, et le volume d'écriture augmente. Pour la plupart des services, l'augmentation est faible comparée au reste du travail. Un service qui écrit des volumes d'événements très élevés devrait la mesurer sur son propre matériel au lieu de se fier à mon « faible ».

Une latence ajoutée. L'événement atteint le broker après le commit de la transaction, et après que le relay a remarqué la ligne. Avec le polling, le délai dépend de l'intervalle de polling. Avec une approche fondée sur le log, il peut être plus court, et le Chapitre 5 compare les deux. Dans les deux cas, l'outbox n'est pas le bon choix pour les flux où l'événement doit atteindre le broker dans les quelques millisecondes qui suivent la requête, et vous devriez connaître votre budget de latence avant de choisir.

Des doublons par conception. Traité plus haut. Si un consommateur ne peut pas être rendu idempotent, alors l'outbox n'améliore rien pour lui, et l'équipe doit décider si ce consommateur doit exister sous cette forme.

L'ordre n'est que partiellement préservé. Le relay lit les lignes par `id`, et l'ordre des `id` n'est pas l'ordre des commits, à moins que chaque transaction ne verrouille la ligne de l'agrégat avant d'insérer la ligne d'outbox. Ce verrou sérialise les écrivains d'un même agrégat, ce qui est un coût en soi sur les agrégats très sollicités. Les relays parallèles dégradent l'ordre, et le verrou n'aide pas entre agrégats. La colonne `aggregate_version` donne

aux consommateurs quelque chose à vérifier, et elle coûte un incrément de version à chaque écriture. L'outbox vous donne un endroit où imposer l'ordre, pas l'ordre gratuitement. Le Chapitre 5 et le Chapitre 9 y reviennent.

Croissance et nettoyage. Les lignes publiées s'accumulent. Sans suppression ni archivage, la table grossit jusqu'à ce que la requête du relay ralentisse. La rétention est une décision, et elle interagit avec la réconciliation, puisque les anciennes lignes sont aussi votre preuve de ce qui a été publié.

Ne fonctionne que pour un état stocké dans une base de données qui prend en charge les transactions. Si l'état vit dans un stockage sans transactions multi-lignes, il vous faut une autre approche, comme un change stream propre au stockage ou une conception en event sourcing. Un stockage qui n'offre l'atomicité qu'au sein d'un seul document a sa propre variante : gardez les événements en attente dans le document de l'agrégat, et laissez le relay les vider à partir de là.

Quand ne pas l'utiliser

Je n'ajouterais pas d'outbox quand :

- l'événement est best-effort et sa perte est acceptable, comme une métrique ou une indication analytique de faible valeur
- le service ne détient aucun état durable qui lui soit propre, si bien qu'il n'y a rien dont faire le commit avec l'événement
- le système est assez petit pour qu'un seul processus possède à la fois l'état et ses consommateurs, et qu'un appel direct soit plus simple
- l'équipe ne sait pas encore rendre les consommateurs idempotents, auquel cas je réglerais d'abord ce point, puisque l'outbox augmente l'enjeu de ce manque

> **Remarque :** « Nous utilisons un outbox » est une affirmation sur le chemin producteur d'un seul service. Elle ne dit rien sur la sûreté des consommateurs, sur le maintien de l'ordre ni sur la bonne conception des événements. Traitez-le comme un pattern dans un ensemble, pas comme un sceau de fiabilité.

Erreurs courantes

- **Écrire la ligne d'outbox dans une transaction séparée.** C'est tentant parce qu'un hook de framework ou une abstraction de bus d'événements cache la connexion. Les deux écritures deviennent alors un dual write entre deux tables de la même base de données, et la fenêtre est de retour. Utilisez une connexion et une transaction, et testez le rollback.
- **Publier depuis un hook « after commit ».** C'est tentant parce qu'il ne se déclenche que pour des données dont le commit est fait et ne nécessite aucune table. C'est commit-puis-publication avec l'intention en mémoire, donc un crash entre les deux perd l'événement. Écrivez la ligne dans la transaction et laissez le relay publier.
- **Insérer la ligne d'outbox avant de verrouiller l'agrégat.** Le code fonctionne dans tous les tests avec un seul écrivain. En concurrence, l'ordre des `id` et l'ordre des commits divergent pour une même commande, et le relay peut publier ses événements à l'envers. Verrouillez d'abord la ligne de l'agrégat, incrémentez sa version, puis insérez.
- **Publier depuis le chemin de requête et garder l'outbox comme secours.** Cela donne l'impression de porter ceinture et bretelles. Cela produit deux publieurs, des doublons à des moments différents, et un ordre sur lequel vous ne pouvez pas raisonner. Laissez le relay être le seul publieur.
- **Générer l'`event_id` dans le relay.** Cela semble inoffensif, puisque c'est le relay qui construit le message. Une republication reçoit alors un nouvel identifiant, et les consommateurs ne peuvent pas la reconnaître comme un doublon. Générez l'identifiant quand la ligne est écrite.
- **Marquer la ligne comme publiée avant que le broker ne confirme.** Cela réduit les doublons et paraît plus propre. Cela convertit un doublon en événement perdu, ce qui est la pire défaillance. Marquez après la confirmation, et acceptez le doublon.
- **Construire le payload après le commit.** Relire l'agrégat pour remplir l'événement semble garder le code producteur simple. Une autre transaction peut modifier les données entre-temps, et l'événement ne correspond plus au fait. Construisez le payload à partir de ce que la transaction a écrit.

- **Laisser certains chemins de code contourner l'outbox.** Un script de migration, un outil d'administration ou un correctif SQL exécuté à la main modifie l'état et n'émet rien. Faites passer chaque changement d'état qui compte par la même fonction, et relisez les réparations manuelles pour les événements qu'elles auraient dû produire.
- **Oublier le nettoyage.** La table fonctionne pendant des mois, puis la requête du relay ralentit au pire moment. Décidez de la rétention quand vous créez la table.
- **Considérer l'outbox comme la fin de l'histoire de la fiabilité.** Il ferme une fenêtre. Les consommateurs ont encore besoin d'idempotence, de retries et d'un endroit pour les messages qui ne réussissent jamais.

Points clés

- Un dual write ne peut pas être rendu sûr en choisissant un meilleur ordre : commit-puis-publication perd des événements, publication-puis-commit en invente, et une publication à l'intérieur d'un bloc de transaction reste le second cas.
- Le commit en deux phases peut fermer la fenêtre en principe, mais pour la base de données d'un service et un broker, il coûte généralement plus en verrous, en exploitation du coordinateur et en disponibilité que le problème ne le vaut.
- Le transactional outbox transforme le dual write en une transaction locale suivie d'un déplacement répétable, en écrivant l'événement sous forme de ligne dans la même transaction que le changement d'état.
- La ligne d'outbox porte l'enveloppe complète de l'événement, un `event_id` stable créé au moment de l'écriture afin que chaque republication puisse être reconnue, et un `aggregate_version` incrémenté sous le verrou de l'agrégat, parce que l'ordre des `id` n'est pas l'ordre des commits.
- L'outbox garantit l'atomicité de l'état et de l'intention, la durabilité de l'intention et une publication at-least-once, et il ne garantit pas qu'un message soit publié exactement une fois.
- Les doublons sont le prix de la garantie, donc un outbox sans consommateurs idempotents échange une défaillance contre une autre.

- Listen-to-yourself et l'event sourcing suppriment eux aussi le dual write, mais ils changent la façon dont le service lit et stocke son état, donc je les choisirais pour ce qu'ils vous apportent et pas simplement pour éviter un relay.

La publication n'est que la moitié du déplacement : quelque chose doit lire ces lignes et les porter jusqu'au broker sans les perdre, sans les réordonner et sans l'inonder. Le Chapitre 5 traite du relay, du polling face au change data capture, et de la façon d'exploiter l'outbox en production.

Chapitre 5 — Relays, polling et change data capture

Imaginez un mercredi matin chez Parcelly. L'équipe Orders a livré le transactional outbox il y a deux semaines. Chaque événement `OrderPlaced` est désormais écrit dans la table `outbox` dans la même transaction locale que la commande elle-même, et la fenêtre de dual write du Chapitre 4 est fermée. L'équipe considère le problème comme résolu et passe au ticket suivant.

Le relay est un petit processus qui se réveille chaque seconde, lit les lignes dont l'`id` est supérieur au dernier qu'il a vu, les publie et retient le nouvel `id` le plus élevé. Il tourne en une seule instance, et personne n'y a pensé depuis le jour de son déploiement.

Puis un déploiement de routine remplace le relay au mauvais moment. L'ancienne instance a publié un lot de 200 lignes sur le broker mais n'a pas encore enregistré sa nouvelle position. La nouvelle instance repart de la dernière position enregistrée, lit les mêmes 200 lignes et les publie à nouveau. Inventory réserve du stock une seconde fois pour des commandes qu'il a déjà traitées. Une semaine plus tard, on ajoute un job de nettoyage qui supprime discrètement des lignes que le relay n'a pas encore lues. Le mois suivant, une transaction longue fait son commit en retard et ses lignes ne quittent jamais la table.

Aucune de ces défaillances ne se trouve dans le pattern outbox lui-même. Toutes se trouvent dans la partie qui déplace les lignes vers le broker. Comment faire tourner cette partie de sorte qu'elle soit lente à échouer, rapide à se rétablir, et honnête sur les doublons qu'elle peut encore produire ?

Le problème, et pourquoi il est difficile

Le transactional outbox vous donne une garantie : si la commande a fait l'objet d'un commit, l'intention de publier son événement a fait l'objet d'un commit avec elle. Il ne dit rien de la seconde moitié du trajet. Quelqu'un doit lire les lignes, les remettre au broker et se souvenir de l'avoir fait. Ce quelqu'un est le **relay**, le processus qui déplace les lignes de l'outbox vers le broker.

Un relay fait trois choses : il trouve les lignes à publier, il les publie, et il enregistre qu'il l'a fait. Ce sont trois étapes distinctes, et le broker et la base de données ne partagent pas de transaction. Le même problème de dual write que nous avons retiré du producteur réapparaît à l'intérieur du relay, avec une fenêtre plus étroite.

Les fenêtres de crash sont assez peu nombreuses pour être énumérées. Le tableau est la sémantique des défaillances du relay : ce qui peut échouer, comment vous l'observeriez, et quelle garantie y survit.

Le crash survient	État de la ligne	Résultat
Avant la publication	Non publiée	La ligne est reprise, l'événement est publié une fois. Sûr.
Après la publication, avant le marquage	Non publiée, mais l'événement est sur le broker	La ligne est reprise, l'événement est publié deux fois. Un doublon.
Après le marquage	Publiée	Rien à faire. Sûr.

Aucun ordre des trois étapes ne ferme la ligne du milieu. Si le relay marque d'abord et publie ensuite, un crash entre les deux perd l'événement, ce qui est précisément la défaillance que l'outbox existe pour empêcher. Le relay publie donc d'abord et marque ensuite, et accepte qu'un crash entre les deux produise un doublon. C'est le même compromis que celui du chapitre sur l'outbox : une livraison at-least-once, avec la gestion des doublons repoussée vers les consommateurs (Chapitre 7).

La première question de conception n'est donc pas de savoir comment éviter les doublons. Elle est de savoir comment garder la fenêtre petite, comment faire en sorte que les lignes sortent dans un ordre sensé, et comment voir le relay prendre du retard avant qu'un client ne le voie. La

seconde question de conception est un choix entre deux mécanismes de lecture des lignes : faire du polling sur la table, ou lire le propre journal de transactions de la base de données. Chacun déplace la difficulté ailleurs.

Le relay par polling

Un **relay par polling** interroge la table d'outbox à intervalle régulier, publie tout ce qu'il trouve et marque ces lignes comme publiées. C'est le mécanisme le plus simple qui fonctionne, et je commencerais par là, sauf si les chiffres disent le contraire.

La boucle de base

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

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

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

La requête derrière `selectUnpublished` est l'endroit où la plupart des relays par polling se trompent, elle mérite donc qu'on s'y attarde. La forme évidente est « donnez-moi les lignes dont l'`id` est supérieur au dernier que j'ai traité ». Elle paraît efficace, et elle n'est pas sûre.

Pourquoi un curseur sur l'id n'est pas sûr

Les identifiants de ligne sont généralement attribués au moment de l'insertion, pas au moment du commit de la transaction. Imaginez deux transactions d'Orders. La transaction A insère une ligne d'outbox avec l'id 101, puis passe un certain temps à appeler une fonction lente avant de faire son commit. La transaction B insère l'id 102 et fait son commit immédiate-

ment. Le relay fait son polling, voit 102, le publie et déplace son curseur sur 102. A fait alors son commit. La ligne 101 est désormais derrière le curseur et ne sera jamais lue.

La défaillance est silencieuse. Rien n'échoue, aucune ligne de log n'apparaît, et l'événement n'est jamais publié. C'est la même perte que l'outbox a été conçu pour supprimer, réintroduite par une optimisation.

La correction que je préfère consiste à cesser de retenir une position et à interroger par état : sélectionner les lignes dont `published_at` est nul, triées par `id`. Une ligne qui fait son commit en retard est toujours non publiée quand arrive le polling suivant, elle est donc trouvée. Le coût est un index sur les lignes non publiées, et une table qui doit rester assez petite pour que cette requête soit bon marché. Un index partiel qui ne couvre que les lignes non publiées garde la requête rapide quel que soit le volume d'historique que contient la table, si votre base de données le prend en charge. Sinon, le travail de rétention décrit plus loin dans ce chapitre compte davantage. Un index partiel ne supprime pas tout le coût : dans les bases de données qui conservent les anciennes versions des lignes, chaque ligne est insérée, mise à jour puis supprimée, et ce churn laisse des lignes mortes que le moteur doit nettoyer.

La lecture par état corrige la perte. Elle ne corrige pas l'ordre, qui est une question distincte à laquelle répond la section sur la préservation de l'ordre.

Lots et intervalle de polling

Deux réglages façonnent le comportement du relay : combien de lignes il prend par polling, et combien de temps il dort quand il n'y a rien à faire.

Un lot plus grand amortit le coût de la requête et de la mise à jour de marquage, et améliore le débit. Il élargit aussi la fenêtre de crash : un relay qui meurt au milieu d'un lot de 500 laisse jusqu'à 500 événements à republier. Un lot plus petit republie moins après un crash et, avec la réservation de lignes décrite plus bas, tient des verrous moins longtemps. Disons que 50 à 200 lignes est une première estimation raisonnable ; mesurez avant d'ajuster.

L'intervalle de polling est un plancher de latence. Avec un intervalle d'une seconde, un événement peut attendre près d'une seconde avant que le relay ne le regarde. Raccourcir l'intervalle réduit la latence et augmente le nombre de requêtes vides que la base de données doit servir. Pour Parcelly, une ou deux secondes de délai entre le commit d'`OrderPlaced` et sa publication sont invisibles pour les acheteurs. Pour un flux où une personne attend devant un écran, ce n'est peut-être pas le cas.

Marquer après la publication

L'étape de marquage mérite son propre soin. Publiez chaque ligne, recueillez l'ack du broker, et ne marquez que les lignes que le broker a confirmées. Ne marquez pas une ligne dont la publication n'a pas été confirmée, même si l'appel est revenu sans erreur, car de nombreuses bibliothèques clientes envoient de façon asynchrone et signalent l'échec plus tard.

```
function publishAcked(rows):
    ackedIds = []
    for each row in rows:
        result = publishAndAwaitAck(topicFor(row.aggregate_type),
buildEnvelope(row),
                                    key: row.partition_key)
        if result is failure:
            // stop here: later rows with the same partition_key must
not overtake this one
            break
        ackedIds.append(row.id)
    // only these rows may ever be marked
    return ackedIds
```

Cette fonction ne touche jamais la table. Elle renvoie les identifiants que le broker a confirmés, et chaque relay de ce chapitre marque exactement ces identifiants et aucun autre. Une ligne qui a échoué, et toutes les lignes qui la suivent dans le lot, restent non publiées et sont relues au polling suivant. La publication et le marquage sont séparés à dessein, afin que le marquage puisse être protégé par un verrou ou un lease à l'unique endroit où il a lieu. Chaque événement va dans le topic de son type d'agrégat, avec `partition_key` comme clé de message, le même routage que celui du Chapitre 4.

Le `break` est un choix délibéré, et il concerne l'ordre, qui revient plus bas. Sauter une ligne en échec et publier les lignes qui la suivent est plus rapide, et peut réordonner les événements d'un agrégat. S'arrêter est plus lent et sûr. Je m'arrêterais, et je laisserais la supervision me dire que le relay est bloqué. Cette esquisse arrête tout le lot par simplicité ; ne bloquer que la clé concernée est le raffinement décrit plus bas, sous la préservation de l'ordre.

Un seul relay actif ou plusieurs

La topologie la plus simple est une seule instance de relay qui fait tout le travail. Son comportement vis-à-vis de l'ordre est plus facile à raisonner, parce qu'elle lit dans l'ordre des `id` et publie dans cet ordre, avec la réserve sur l'`id` et l'ordre des commits exposée plus bas dans la section sur la préservation de l'ordre. Sa faiblesse est la disponibilité : si cette instance meurt, rien n'est publié tant qu'une autre n'a pas démarré. Pour de nombreux systèmes, un redémarrage en quelques secondes est acceptable, et un relay unique avec un superviseur qui le redémarre est la bonne réponse.

Quand un relay ne suit plus, ou quand une interruption de la publication est inacceptable, vous en faites tourner plusieurs. Vous avez alors un nouveau problème : deux relays ne doivent pas publier la même ligne en même temps. L'outil standard est un verrou de ligne avec une sémantique de saut. Beaucoup de bases de données relationnelles offrent un moyen de sélectionner des lignes pour mise à jour en sautant celles qu'une autre transaction a déjà verrouillées, ce qu'on décrit couramment comme un verrouillage de type `SKIP LOCKED`. Le relay réserve un lot dans une transaction et les autres relays passent sur ces lignes.

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

Cela fonctionne, et cela entraîne des coûts souvent oubliés. Premièrement, la transaction reste ouverte pendant que le relay parle au broker, donc un broker lent signifie des verrous tenus longtemps et une pile croissante de transactions ouvertes. Dans les bases de données qui conservent les anciennes versions des lignes pour les lecteurs concurrents, une transaction ouverte longtemps retarde aussi le nettoyage des lignes mortes, ce qui pénalise justement la table qui change le plus. Deuxièmement, si la transaction est annulée après la publication, à cause d'une connexion perdue ou d'un marquage échoué, chaque ligne qu'elle a publiée est publiée à nouveau. Une alternative plus courte est une brève transaction de réservation qui estampille les lignes avec une heure de réservation, une publication hors de toute transaction, et une transaction de marquage séparée, avec un timeout de réservation pour que les lignes d'un relay mort redeviennent réservables. Cela coûte un état de plus à raisonner. Troisièmement, les relays parallèles perdent l'ordre global. Le relay A peut réserver les lignes 1 à 100 pendant que le relay B réserve 101 à 200, et B peut finir en premier. Pour des événements concernant des commandes différentes, cela n'a pas d'importance. Pour deux événements concernant la même commande, si.

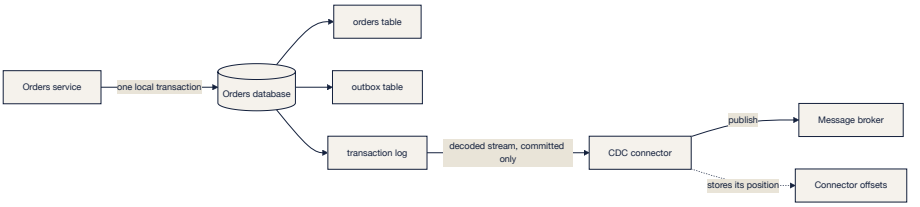
Le remède consiste à partitionner le travail par `partition_key` plutôt que par ligne. Attribuez à chaque relay un ensemble de clés, par exemple en hachant `partition_key` dans un nombre fixe de buckets et en confiant chaque bucket à exactement un relay à la fois. Au sein d'un bucket, les lignes sont traitées dans l'ordre par un seul worker, et d'un bucket à l'autre elles s'exécutent en parallèle. C'est la même idée que les partitions d'un broker fondé sur un log, appliquée une étape plus tôt. Le mot « exactement » a besoin du lease de la section sur le failover plus bas, et le transfert des buckets pendant un rééquilibrage est le moment où l'ordre par clé est le plus menacé, parce que l'ancien et le nouveau propriétaire peuvent publier pour la même clé en même temps. Le Chapitre 9 explique pourquoi l'ordre par clé est l'ordre que vous pouvez réalistement conserver.

Change data capture

Le second mécanisme n'interroge pas du tout la table d'outbox. Le **change data capture (CDC)** lit le propre journal de transactions d'une base de données et transforme les changements de lignes dont le commit est fait

en un stream d'enregistrements de changement. Un connecteur CDC, dont Debezium est l'exemple familier, se branche sur ce journal, remarque qu'une ligne a été insérée dans `outbox` et publie un message sur le broker.

La table garde son rôle, parce que la garantie transactionnelle vient toujours du fait de l'écrire dans la même transaction que le changement d'état. Ce qui change, c'est qui la lit. Aucun processus applicatif ne fait de polling. La base de données enregistre déjà chaque changement dont le commit est fait dans un journal ordonné pour sa propre reprise et sa réplication, et le connecteur lit ce journal comme le ferait une réplique.



Regardez le chemin qui va du journal de transactions au broker. Le stream de changements que la base de données expose aux connecteurs (PostgreSQL l'appelle logical decoding, MySQL expose son binary log) ne contient que des transactions dont le commit est fait, et elles apparaissent dans l'ordre des commits. Cela élimine entièrement le problème du commit tardif vu dans la section sur le polling : le journal est déjà un ordre total des commits, donc une ligne qui fait son commit en retard apparaît en retard, à la bonne place.

Comment il se comporte

Quelques propriétés découlent de cette conception.

L'ordre suit l'ordre des commits. Les événements apparaissent dans l'ordre où les transactions ont fait leur commit. Un connecteur qui publie avec la `partition_key` de la ligne comme clé de message conserve l'ordre par clé sur un broker fondé sur un log, à condition que le producteur qu'il contient soit configuré de sorte qu'un retry ne puisse pas réordonner des requêtes en vol simultanément. Vérifiez les réglages de votre client ; la même réserve s'applique à un relay par polling qui publie de façon asynchrone.

La latence est faible et stable. Il n'y a pas d'intervalle de polling. Le connecteur voit un changement peu après le commit, donc le délai est généralement faible et ne dépend pas de la fréquence à laquelle quoi que ce soit se réveille.

Le connecteur suit sa propre position. Il enregistre jusqu'où il a lu dans le journal, et après un redémarrage il reprend à partir de là. Les connecteurs enregistrent couramment la position périodiquement et non après chaque événement, si bien qu'un crash peut republier tous les événements depuis la dernière position enregistrée. Le CDC est at-least-once, comme le polling. La fenêtre de doublon s'est déplacée, elle n'a pas disparu.

L'indicateur de publication est facultatif. Comme le connecteur n'a pas besoin de trouver les lignes non publiées, la table n'a pas besoin de colonne `published_at` pour le relay. Beaucoup d'équipes qui utilisent le CDC suppriment chaque ligne d'outbox peu après son insertion. L'insertion a déjà été enregistrée dans le journal, donc le connecteur la voit quand même. La table reste minuscule, et le problème du nettoyage disparaît en grande partie.

L'enveloppe doit être mise en forme. Un enregistrement de changement brut décrit une ligne, pas un événement. Le connecteur ou une étape de transformation doit faire correspondre les colonnes de l'outbox à l'enveloppe standard : `event_id`, `event_type`, `partition_key` comme clé de message, `payload` comme corps. La plupart des connecteurs prennent en charge le routage et la correspondance de champs exactement dans ce but. Gardez les colonnes de l'outbox alignées sur l'enveloppe pour que la correspondance soit mécanique.

Ce qu'il coûte

Le CDC n'est pas gratuit, et ses coûts sont opérationnels.

Vous faites désormais tourner un connecteur, généralement au sein d'un framework de connecteurs, plus l'infrastructure de broker dont il dépend. Quelqu'un doit apprendre comment il tombe en panne.

La base de données doit exposer son journal au connecteur, ce qui implique généralement d'activer un niveau de journalisation particulier et de créer une position de réplication que la base de données conserve pour le compte du connecteur. Si le connecteur s'arrête et que personne ne le re-

marque, la base de données garde des segments de journal qu'elle ne peut pas supprimer, et le disque se remplit. C'est une forme de défaillance connue dans la famille CDC, et les détails varient selon la base de données. Je traiterais « le connecteur est arrêté » comme un appel d'astreinte, parce que les dégâts ne se limitent pas au broker.

Les changements de schéma de la table d'outbox, et de toute table que surveille le connecteur, doivent être planifiés. Un renommage de colonne ou un changement de type peut casser la correspondance.

La mise en place initiale, les snapshots et les redémarrages du connecteur après de longues pannes méritent aussi d'être répétés. Le comportement dépend du produit et de la base de données, et je ne le déduirais pas d'un diagramme.

> **Remarque :** Le CDC est une famille de techniques, et les produits diffèrent dans leur façon de lire le journal, de stocker les positions et dans ce qui se passe après une longue panne. Traitez cette section comme la forme générale, et lisez la documentation de votre connecteur et de votre base de données pour le comportement exact des versions que vous exploitez.

Quand le journal a disparu

« Le disque se remplit » est une moitié d'un compromis, et l'autre moitié est pire. Soit la base de données conserve sans limite le journal pour la position du connecteur, ce qui remplit le disque, soit elle plafonne ce qu'elle conserve, ou fait expirer les journaux selon un calendrier, ce qui protège le disque. Beaucoup de bases de données vous laissent choisir. Si vous plafonnez et que le connecteur est arrêté plus longtemps que le plafond, sa position enregistrée pointe vers un journal qui n'existe plus. Les événements de l'intervalle sont perdus, ou le connecteur doit repartir d'un snapshot.

Ce qu'un nouveau snapshot fait à un outbox dépend d'une décision antérieure. Si la table contient encore des lignes, un snapshot les réémet toutes, si bien que chaque ligne conservée devient un doublon pour chaque consommateur. Si, comme c'est courant avec le CDC, les lignes ont été supprimées juste après leur insertion, le snapshot ne trouve rien, et les

événements de l'intervalle ne peuvent pas du tout être récupérés depuis la table. C'est une perte silencieuse : pas d'erreur, pas de doublon, une commande qui n'a jamais atteint Inventory. La suppression qui gardait la table minuscule est la même suppression qui efface la preuve.

Un failover du primary de la base de données ajoute un troisième cas. Le fait que la position du connecteur soit reportée sur le nouveau primary varie selon la base de données et la version, et je le découvrirais en répétant un failover, pas en lisant un diagramme.

Trois habitudes en découlent. Choisissez le plafond de rétention délibérément, comme un compromis entre le disque et la durée de panne à laquelle vous pouvez survivre, et notez le chiffre. Déclenchez une alerte sur la marge, c'est-à-dire la part de la fenêtre de rétention que le retard du connecteur a déjà consommée, avant qu'elle ne s'épuise. Et décidez avant la panne comment vous récupéreriez les événements de l'intervalle, par exemple grâce à un job de réconciliation (Chapitre 13) qui compare l'état des commandes avec ce qui a été publié. Le Chapitre 14 traite de la reprise une fois les dégâts faits.

Choisir entre polling et CDC

Aucun des deux mécanismes n'est meilleur dans l'absolu. Le tableau montre où ils diffèrent.

Préoccupation	Relay par polling	Connecteur CDC
Pièces mobiles	Un petit processus qui vous appartient	Le connecteur, son framework, les réglages du journal de la base de données
Latence	Bornée inférieurement par l'intervalle de polling	Généralement faible et stable
Source de l'ordre	L'ordre de la requête plus votre propre vigilance	L'ordre des commits issu du journal
Transactions qui font leur commit en retard	Sûres avec une requête par état	Gérées par le journal
Charge sur la base de données	Requêtes répétées, verrous de ligne	Lectures du journal, segments de journal conservés

Préoccupation	Relay par polling	Connecteur CDC
Doublons après un crash	Possibles, la fenêtre est le lot non marqué	Possibles, la fenêtre est la position non enregistrée
Croissance de la table	Nécessite un travail de rétention	Les lignes peuvent être supprimées immédiatement
Défaillance la plus douloureuse	Retard silencieux, lignes bloquées	Connecteur bloqué, journal conservé
Compétences requises	Compétences applicatives ordinaires	Fonctionnement interne du journal de la base de données, exploitation du connecteur

Mon choix par défaut pour une équipe qui débute est un relay par polling. Il tient en quelques dizaines de lignes de code, l'équipe sait déjà exploiter et déboguer un tel service, et ses modes de défaillance sont visibles dans la table qu'il interroge. Je passerais au CDC quand l'une de trois choses devient vraie : la latence du polling est trop élevée et ne peut pas être réduite sans marteler la base de données, le volume de l'outbox fait des requêtes de polling un coût mesurable, ou vous exploitez déjà une plateforme CDC et quelqu'un en est responsable.

Je n'adopterais pas le CDC uniquement parce qu'il paraît plus sérieux. Une équipe qui n'a jamais exploité de journal de réplication et y place son flux de commandes a échangé un problème qu'elle comprenait contre un problème qu'elle ne comprend pas. L'inverse est vrai aussi : une équipe qui exploite déjà le CDC à d'autres fins gagne peu à écrire un relay par polling à côté.

Exploiter le relay

Choisir le mécanisme est la première moitié du travail. La seconde moitié est le travail qui le maintient en bonne santé.

Rétention et nettoyage

Un relay par polling laisse derrière lui les lignes publiées. Livrée à elle-même, la table grandit sans limite, la requête sur les lignes non publiées ralentit, et la base de données dépense des efforts sur des lignes que personne ne lit. Le nettoyage doit être conçu, pas ajouté plus tard.

Deux choix comptent : combien de temps garder les lignes publiées, et comment les supprimer. Les garder un moment a de la valeur. Quand un consommateur signale un événement manquant, l'outbox est le premier endroit où vous regardez pour savoir s'il a jamais été publié. Disons que quelques jours sont un point de départ raisonnable. La bonne durée dépend de la rapidité avec laquelle votre équipe remarque les événements manquants, ce qui est une question de personnes et d'alerting, pas de stockage.

```
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)
```

Supprimez par petits lots, afin que le job ne tienne pas de verrous longtemps et n'inonde pas les répliques. Si la table est grande, partitionner par temps et supprimer des partitions entières coûte moins cher que de supprimer ligne par ligne, là où votre base de données le permet. Quoi que vous choisissiez, la condition de purge doit inclure `published_at is not null`. Un nettoyage qui supprime selon l'âge seul supprimera, le jour où le relay prend du retard, des événements qui n'ont jamais été publiés. C'est la défaillance du scénario d'ouverture, et c'est une perte de données, pas un doublon.

> **Avertissement :** Ne purgez jamais les lignes d'outbox selon leur seul âge. Un relay bloqué plus longtemps que votre période de rétention verra ses lignes non publiées supprimées, et ces événements seront perdus définitivement. Ne purgez que les lignes marquées comme publiées, et déclenchez une alerte sur les lignes non publiées qui vieillissent.

Avec le CDC, le nettoyage est différent mais pas absent. Les lignes peuvent être supprimées juste après leur insertion, donc la table reste petite. Ce qui grandit désormais, c'est le journal conservé, et ce qu'il faut surveiller, c'est

le retard du connecteur par rapport à lui. Supprimer immédiatement signifie aussi qu'un nouveau snapshot ne peut rien récupérer, comme l'explique la section précédente sur un journal disparu, donc pesez la suppression immédiate face à une courte rétention des lignes publiées.

Surveiller le relay

Un relay qui s'arrête est invisible de l'extérieur. Les commandes réussissent toujours, l'outbox accepte les lignes, et tous les autres services se passent silencieusement d'événements. Personne ne reçoit d'erreur. C'est pourquoi le relay a besoin de ses propres signaux, et j'en mettrais trois en place avant le lancement.

1. **L'âge de la plus ancienne ligne non publiée.** C'est le meilleur chiffre unique, parce qu'il mesure ce que vivent les utilisateurs : depuis combien de temps l'événement le plus lent attend. Déclenchez une alerte quand il dépasse un seuil lié au métier, pas à la technologie. Si l'e-mail de confirmation d'un acheteur doit arriver en moins d'une minute, un âge de cinq minutes pour la plus ancienne ligne non publiée est un incident.
2. **Le nombre de lignes non publiées.** Un backlog qui grandit indique que le relay est plus lent que le producteur, même quand la plus ancienne ligne est encore jeune.
3. **Le taux d'erreurs de publication et la durée des lots.** Ils expliquent les deux premiers. Un relay qui fait des retries sur la même ligne à chaque polling affichera un nombre d'erreurs stable et un âge croissant.

Pour un connecteur CDC, les mêmes questions s'appliquent dans d'autres unités : à quelle distance de la tête du journal se trouve la position du connecteur, s'il tourne, et quelle quantité de journal la base de données conserve pour lui. Le Chapitre 13 revient sur ces signaux et sur la réconciliation, qui attrape ce que les métriques de retard manquent.

Failover et double publication

Un relay doit survivre à ses propres redémarrages, y compris dans le cas où deux instances tournent brièvement en même temps. Les déploiements, les partitions réseau qui font paraître l'ancienne instance morte, et les or-

chestrateurs qui redémarrent un processus avant que l'ancien ne soit sorti produisent tous ce chevauchement.

Pour un relay actif unique, utilisez un lease : une ligne ou un verrou dans la base de données que le relay actif renouvelle à intervalle court et qui expire si le détenteur s'arrête. Un relay de secours prend le relais après l'expiration. Le détail qui fait trébucher, c'est que l'ancien détenteur peut ne pas savoir que son lease a expiré. Il pourrait être mis en pause par un long garbage collection ou un réseau bloqué, se réveiller et publier un lot qui ne lui appartient plus.

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

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

Le token croissant est un fencing token, et l'idée ne fonctionne que là où la ressource protégée le vérifie. La base de données peut le vérifier : on peut faire échouer un commit qui porte un token périmé, et le Chapitre 7 utilise exactement cela pour les consommateurs et les API. Le broker, en général, ne le peut pas, donc le token n'empêche pas un relay périmé de publier. Le marquage est une autre affaire. Le détenteur périmé ne marque que les lignes que le broker a confirmées, donc ses marques sont vraies, et les an-

nuler ne ferait que créer davantage de doublons. Ce qu'un lease doit limiter, c'est la publication concurrente, parce que deux détenteurs qui publient ensemble produisent des doublons et peuvent placer une ligne plus ancienne d'une clé après une plus récente.

Quatre choses la limitent. Gardez le TTL du lease bien au-dessus de la durée d'un lot dans le pire cas, et gardez les lots petits, parce que le lease n'est renouvelé qu'une fois par boucle et qu'un broker lent peut à lui seul le dépasser. Vérifiez le lease restant avant chaque publication, comme le fait l'esquisse. Là où un broker fondé sur un log prend en charge le fencing des producteurs par époque, utilisez-le, afin que le broker lui-même rejette l'ancien détenteur. Et quand l'ancien détenteur se réveille, laissez-le le découvrir en interrogeant la base de données, dans une transaction qui verrouille la ligne du lease, avant qu'il ne publie quoi que ce soit d'autre. Une vérification qui ne tient pas le verrou est un check-then-act, et elle échoue de la même manière.

C'est la limite honnête. Le failover peut produire une double publication, et aucun schéma de lease ne la supprime. Ce qu'un lease vous apporte, c'est que la fenêtre est courte et rare. Ce qui supprime le préjudice, c'est un consommateur qui traite la seconde copie comme le même événement, à l'aide de l'`event_id` qui reste stable d'une publication à l'autre. Le relay doit donc prendre l'`event_id` dans la ligne d'outbox et ne jamais en générer un nouveau au moment de publier. Un relay qui fabrique un identifiant neuf à chaque envoi transforme un doublon inoffensif en deux événements apparemment distincts. Certains brokers acceptent un identifiant de déduplication fourni par le producteur et écartent les répétitions dans une fenêtre de temps ; là où le vôtre le fait, réglez-le sur `event_id`. Cela réduit la fenêtre gratuitement et ne remplace pas l'inbox du consommateur. Le Chapitre 7 traite du côté consommateur.

Préserver l'ordre

L'outbox ne préserve l'ordre des événements que dans la mesure où le relay le fait, et le relay ne peut préserver que l'ordre que contient la table. Le Chapitre 4 a montré que l'ordre des `id` n'est pas l'ordre des commits : une transaction peut prendre un `id`, attendre un verrou, et faire son commit après une transaction qui a pris un `id` ultérieur. L'`id` est l'ordre d'insertion, pas l'ordre de commit.

Deux conséquences en découlent pour un relay par polling. Si un `id` inférieur pour une clé fait son commit après qu'un identifiant supérieur pour la même clé a déjà été publié, les deux sortent à l'envers, et c'est la requête par état qui avait sauvé la ligne tardive qui la publie en retard. Si les deux deviennent visibles dans un même polling avec des identifiants inversés, ils sortent à l'envers dans un même lot. Un relay qui publie « dans l'ordre des `id` » ne livre donc pas les événements d'un agrégat dans l'ordre où l'agrégat a changé, sauf si l'une de deux conditions est remplie.

La première est que les producteurs verrouillent la ligne de l'agrégat avant d'insérer la ligne d'outbox, afin que les transactions sur la même clé se sérialisent et que leurs identifiants suivent leur ordre de commit (Chapitre 4). La seconde est que l'ordre soit porté par les données : la colonne `aggregate_version`, que les consommateurs comparent et qu'utilise le Chapitre 9. Je ferais les deux. Le verrou est une assurance bon marché pour le relay, et la version est ce qui survit à tout ce qui vient après le relay. Un connecteur CDC évite le premier problème parce que le journal est dans l'ordre des commits, et ne vous aide toujours pas entre partitions, donc la version reste utile.

Cela étant, les propres choix du relay comptent encore. Chacun de ceux-ci peut casser l'ordre dont il dispose :

- Sauter une ligne en échec et publier les lignes ultérieures de la même clé.
- Faire tourner des relays parallèles sans partitionner par clé.
- Faire le retry d'une publication échouée hors du flux normal, si bien qu'une ligne ultérieure arrive en premier.
- Utiliser des clés de message différentes pour les événements d'un même agrégat.

Le choix sûr par défaut est le head-of-line blocking par clé : si une ligne pour `order-81724` ne peut pas être publiée, arrêtez de publier les lignes ultérieures pour `order-81724`, et laissez passer les lignes des autres clés. Cela coûte de la latence pour l'agrégat bloqué et protège sa séquence. Si le problème est une ligne qui ne pourra jamais être publiée, comme un payload que le sérialiseur rejette, quelqu'un doit la réparer, et la supervision décrite plus haut est la façon dont il l'apprend.

Soyez clair sur ce que fait l'esquisse précédente. Avec un `break` sur tout le lot et une requête qui prend les plus anciennes d'abord, une ligne impubliable est renvoyée en premier à chaque polling, et elle bloque toutes les clés indéfiniment. Bloquer par clé demande deux ajouts, présentés comme un raffinement de `publishAcked` qui remplace son `break` : le relay retient quelles clés ont échoué dans ce lot, et une ligne qui échoue de manière répétée est mise en parking, afin que la requête cesse de la renvoyer et cesse aussi de renvoyer les lignes ultérieures de sa clé.

```
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
```

La sélection exclut alors toute clé qui a une parked row, si bien que les lignes ultérieures de cette clé attendent et que les lignes des autres clés passent. Le parking est le moment où la colonne `failed_at` du Chapitre 4 devient nécessaire, avec un compteur `attempts`, afin qu'une défaillance transitoire donne lieu à quelques retries avant que la ligne ne soit mise en parking. Une parked row est une ligne dont le `failed_at` est renseigné. Déclenchez une alerte sur chaque parked row, parce qu'une parked row est un agrégat bloqué et qu'une personne doit la réparer ou l'écarter. Le coût est deux colonnes, une requête plus difficile à indexer, et une procédure de réparation que quelqu'un doit écrire avant le premier incident.

L'ordre se perd aussi après le relay, dans le consommateur et le broker. Un relay qui fait sa part vous donne une séquence qui vaut la peine d'être préservée, pas une garantie qu'elle survive au reste du chemin. Le Chapitre 9 suit l'ordre du broker jusqu'au handler.

Compromis et coûts

Les deux mécanismes échangent de la simplicité contre de la portée opérationnelle, et tous deux laissent une fenêtre de doublon que vous ne pouvez pas fermer.

Un relay par polling coûte de la charge sur la base de données et de la latence, en échange du fait de rester dans un code que votre équipe comprend déjà. Ses points faibles sont la requête sur les lignes non publiées dans une table qui grandit, le comportement des verrous quand plusieurs relays tournent, et le retard silencieux quand personne ne surveille la plus ancienne ligne. Si votre système publie à un rythme modeste et qu'une seconde de délai est acceptable, le relay par polling est difficile à battre en coût de possession.

Le CDC coûte un nouveau composant et de nouvelles compétences, en échange d'une latence plus faible, d'une lecture dans l'ordre des commits et d'une table qui reste petite. Ses points faibles sont un connecteur bloqué qui oblige la base de données à conserver son journal, ou un journal plafonné qui expire sous un connecteur bloqué et perd des événements, des changements de schéma qui cassent les correspondances, et des procédures de reprise rarement répétées. Si vous n'avez personne qui sera responsable du connecteur, je ne l'adopterais pas, aussi séduisant que soit le diagramme d'architecture.

Quand ne pas utiliser de relay d'outbox du tout : si un événement peut être perdu sans conséquence, comme une notification informative dont la perte ne coûte rien, une publication directe après le commit peut être acceptable et moins chère. Si vos événements doivent refléter l'état d'une base de données qui n'offre pas de transaction locale couvrant l'état et l'outbox, le relay ne peut pas y remédier, et il vous faut une autre conception.

Deux coûts sont communs aux deux. Aucun ne vous donne une publication exactly-once : le doublon après un crash entre la publication et le marquage, ou entre la publication et l'enregistrement de la position, fait partie de la conception. Et aucun n'aide si le relay et le producteur ne s'accordent pas sur l'enveloppe, donc gardez les colonnes de l'outbox et les champs de l'enveloppe en phase.

Erreurs courantes

- **Utiliser un curseur sur l'id au lieu d'une requête par état.** C'est tentant parce que cela évite de rechercher les lignes non publiées. Les transactions qui font leur commit en retard passent alors derrière le curseur et ne sont jamais publiées. Interrogez par `published_at is null`, ou utilisez un mécanisme fondé sur le log qui ordonne déjà par commit.
- **Marquer les lignes avant que le broker ne confirme.** Marquer d'abord semble supprimer les doublons. Cela convertit un crash en événement perdu. Publiez, attendez l'ack, puis marquez.
- **Générer un nouvel identifiant d'événement dans le relay.** Cela semble inoffensif, puisque c'est le relay qui envoie. L'inbox du consommateur ne peut alors plus reconnaître un événement republié comme le même. Copiez l'`event_id` depuis la ligne.
- **Purger selon l'âge seul.** Cela garde la table petite et passe tous les tests où le relay est en bonne santé. Cela supprime des événements non publiés quand il ne l'est pas. Ne purgez que les lignes publiées et déclenchez une alerte sur les lignes non publiées anciennes.
- **Sauter une ligne en échec pour maintenir le débit.** Les lignes ultérieures du même agrégat la dépassent alors. Bloquez plutôt par clé, mettez en parking la ligne qui continue d'échouer, et corrigez la cause.
- **Croire que l'ordre des `id` est l'ordre des commits.** C'est vrai dans tous les tests à un seul écrivain. Avec des écrivains concurrents, un `id` inférieur peut faire son commit plus tard, et le relay publie la paire à l'envers. Verrouillez la ligne de l'agrégat avant l'insertion dans l'outbox et portez `aggregate_version`.
- **Compter sur le lease pour faire du fencing sur le broker.** Un relay périmé qui se réveille après l'expiration de son lease publie quand même, parce que le broker ne vérifie pas le token. Gardez les lots dans le TTL, vérifiez le lease avant chaque publication et laissez les consommateurs absorber le reste.
- **Exploiter le CDC avec un plafond de rétention que personne n'a choisi.** La valeur par défaut semble un détail jusqu'à ce que le connecteur soit arrêté pendant un week-end et que la position ait disparu. Choisissez le plafond, déclenchez une alerte sur la marge, et sachez ce qu'un nouveau snapshot fait à votre outbox.

- **Faire tourner plusieurs relays sans partitionner par clé.** Les verrous de ligne empêchent le double travail et ne disent rien de l'ordre. Attribuez les buckets de clés à des workers uniques.
- **Ne pas surveiller le relay.** L'outbox rend le producteur fiable, ce qui fait de la défaillance du relay la seule qui reste, et un relay arrêté ressemble exactement à un système calme. Déclenchez une alerte sur l'âge de la plus ancienne ligne non publiée.
- **Traiter une panne du connecteur comme un problème de broker.** Avec le CDC, un connecteur bloqué oblige la base de données à conserver son journal. Déclenchez l'astreinte pour cela, et surveillez le disque, pas seulement le retard.

Points clés

- Un relay fait trois choses distinctes, trouver, publier et marquer, et le broker et la base de données ne partagent aucune transaction, donc un crash entre la publication et le marquage laisse toujours un doublon possible.
- Publiez d'abord et marquez ensuite : l'autre ordre transforme un crash en événement perdu, ce qui est précisément la défaillance que l'outbox existe pour supprimer.
- Un relay par polling doit sélectionner les lignes non publiées par état, et non par un curseur d'id croissant, parce que les transactions font leur commit dans un autre ordre que celui où elles allouent les identifiants, et même alors l'ordre par agrégat ne tient que si les producteurs verrouillent d'abord l'agrégat et portent un `aggregate_version`.
- Pour faire tourner plusieurs relays par polling, réservez les lignes avec un verrouillage de type skip-locked et partitionnez par `partition_key`, afin que les lignes d'un agrégat soient toujours traitées dans l'ordre par un seul worker.
- Le change data capture lit les changements dont le commit est fait dans le journal de la base de données, dans l'ordre des commits, et élimine le polling et le problème du commit tardif, au prix d'un connecteur, d'un compromis de rétention du journal entre un disque plein et des événements perdus, et de compétences que votre équipe doit maîtriser.

- Commencez par un relay par polling, sauf si la latence, la charge ou une plateforme CDC existante dit le contraire, et traitez ce choix comme un compromis entre simplicité et portée opérationnelle.
- Surveillez l'âge de la plus ancienne ligne non publiée, ne purgez que les lignes publiées, mettez en parking les lignes qui ne se publient jamais, copiez l'`event_id` depuis la ligne d'outbox, et attendez-vous à des doublons après un failover, parce qu'un lease limite la publication concurrente et que seul un consommateur idempotent rend le reste inoffensif.

Le relay livre désormais chaque événement au moins une fois, dans un ordre qui vaut la peine d'être conservé, et parfois deux fois. Le problème suivant se situe de l'autre côté du broker : ce qu'un consommateur doit faire quand son travail échoue, que le Chapitre 6 aborde avec les retries.

PARTIE III

Consommer de façon fiable

Chapitre 6 — Des retries bien faits

Imaginez un mardi matin chez Parcelly, au début d'une promotion pour les vendeurs. Orders envoie un command `CapturePayment`, et Payments le transmet au fournisseur de paiement externe. Le fournisseur est en difficulté et répond lentement. Les appels de Payments commencent à dépasser leur timeout, donc Payments fait échouer le message et le broker le livre de nouveau. Chaque instance de Payments fait la même chose, au même moment, avec la même attente fixe d'une seconde.

En une minute, le fournisseur ne reçoit plus la charge normale de la promotion, mais la charge normale plus chaque appel échoué qui revient, encore et encore, par vagues synchronisées. La lenteur du fournisseur, qui aurait pu durer quelques secondes, dure désormais aussi longtemps que les retries continuent d'arriver. Pendant ce temps, les consommateurs de Payments sont occupés à faire des retries sur des messages qui ne réussiront pas, si bien que les messages sains s'accumulent derrière eux. Quelque part dans cette pile, quelques captures qui avaient en réalité réussi chez le fournisseur sont envoyées une nouvelle fois.

Personne n'a écrit de mauvaise ligne de code ici. Chaque instance a fait ce qui était raisonnable : elle a échoué, elle a attendu, elle a fait un retry. Les dégâts sont venus d'un millier de décisions raisonnables que personne n'a coordonnées.

Ce chapitre traite de la manière de faire d'un retry une décision plutôt qu'un réflexe. La question à laquelle il répond est la suivante : quand un appel échoue, comment décider s'il faut faire un retry, quand, combien de fois et à quelle couche, pour que le retry aide le système à se rétablir au lieu de l'en empêcher ?

Le problème, et pourquoi il est difficile

Un retry est une promesse de refaire un essai, pas une promesse de réussir. Dans ce chapitre, une **tentative** désigne tout essai, donc le nombre de tentatives est égal à un plus le nombre de retries. L'approche évidente, « envelopper l'appel dans une boucle et faire quelques retries », échoue pour quatre raisons distinctes, et ces raisons se cumulent.

Premièrement, **toutes les défaillances ne valent pas un retry**. Une requête rejetée parce que la carte a été refusée sera aussi rejetée à la dixième tentative. Faire un retry gaspille de la capacité et retarde le moment où quelqu'un remarque le vrai problème. La boucle de retry doit savoir à quel type de défaillance elle a affaire, et la plupart des boucles ne le savent pas.

Deuxièmement, **les retries ajoutent de la charge au pire moment**. Une dépendance échoue quand elle est surchargée, dégradée ou en train de redémarrer. Les retries sont des requêtes supplémentaires dirigées précisément vers cette dépendance. Si chaque appelant fait trois retries, soit quatre tentatives en tout, un service en difficulté voit jusqu'à quatre fois son trafic normal au moment où il peut le moins se le permettre.

Troisièmement, **les retries se synchronisent**. Les appelants qui échouent ensemble font leurs retries ensemble. Un délai fixe transforme une défaillance unique en pic périodique, parce que chaque appelant se réveille au même instant. C'est le thundering herd, et c'est la raison pour laquelle le délai a besoin d'aléa et pas seulement de durée.

Quatrièmement, **un retry répète une action dont le premier résultat peut être inconnu**. Le Chapitre 3 a montré qu'un timeout ne dit pas si la requête a été traitée. Faire un retry après un timeout risque donc d'exécuter le travail deux fois. Sans idempotence (Chapitre 7), un retry est un doublon en puissance.

Tout le reste de ce chapitre est une manière de répondre à l'un de ces quatre problèmes. La classification répond au premier. Le backoff avec jitter et un retry budget répondent au deuxième et au troisième. L'idempotence, que construit le Chapitre 7, répond au quatrième, et la section consacrée à son absence, plus bas, explique pourquoi on ne peut pas s'en passer.

Classer les défaillances

Avant de décider comment faire un retry, décidez s'il faut en faire un. Le Chapitre 1 a donné trois questions à poser à toute jonction : ce qui peut échouer, comment vous l'observeriez et quelle garantie y survit. Les réponses réunies forment la sémantique d'une défaillance, et une décision de retry consiste à mettre cette sémantique au travail. Je trouve utile de ranger chaque défaillance dans l'une de trois classes, selon ce qu'elle dit de la possibilité qu'un nouvel essai aide, puis d'ajouter un quatrième signal qui ne dit rien du tout sur le message.

Défaillances transitoires

Une **défaillance transitoire** est une défaillance dont on s'attend à ce que la cause disparaisse d'elle-même. Une connexion réinitialisée, un timeout, un `503 Service Unavailable`, une victime de deadlock dans la base de données, une réponse de throttling, une élection de leader dans le broker. La requête n'était pas fausse. Le monde était brièvement en travers du chemin.

Le retry est la bonne réponse, avec un délai, car c'est le délai qui laisse à la cause le temps de se dissiper.

Défaillances permanentes

Une **défaillance permanente** est une défaillance qui se reproduira à chaque tentative, parce que la cause réside dans la requête ou dans les données. Un payload qui échoue à la validation, une référence à un enregistrement qui reste absent bien après le moment où il aurait dû arriver (un enregistrement absent peut être transitoire dans un système orienté événements, car l'événement qui le crée peut arriver en retard ; le Chapitre 9 couvre ce cas), un `400 Bad Request`, une carte refusée pour provision insuffisante, un événement dont le consommateur ne comprend pas la version de schéma.

Le retry est un effort perdu, et parfois nuisible. La réponse correcte est de s'arrêter, d'enregistrer pourquoi, et d'envoyer le message à un endroit où une personne ou un job de réparation pourra s'en occuper. Le Chapitre 8 couvre la destination des défaillances permanentes. Notez en particulier le

refus de carte : pour Payments, ce n'est pas du tout une erreur. C'est un résultat métier attendu qui devient un événement `PaymentFailed`, et aucun retry n'intervient.

Défaillances inconnues

Une **défaillance inconnue** est une défaillance dont vous ne pouvez pas déterminer la classe. Le timeout en est le cas classique, tout comme une exception inattendue portant un message que personne n'avait prévu. Le timeout était peut-être transitoire. L'exception est peut-être un bug du consommateur qui échouera à chaque tentative.

L'inconnu est une vraie classe, pas une lacune de la taxonomie, et la manière de la traiter relève d'une décision de politique. Mon choix par défaut est de traiter les défaillances inconnues comme transitoires, mais avec une limite plus stricte que pour les défaillances transitoires connues, parce que le coût d'un retry injustifié sur une défaillance inconnue se résume à quelques tentatives perdues, alors que le coût d'un abandon injustifié est du travail perdu. Quand les tentatives de toutes les couches sont épuisées, une défaillance inconnue est traitée comme permanente : elle part vers les dead letters avec son historique complet.

Quand le résultat d'une tentative est inconnu, demandez avant d'agir de nouveau. Avant un retry, et à plus forte raison avant une compensation, interrogez le fournisseur sur le statut de l'opération à l'aide de sa clé d'idempotence. Si la capture a eu lieu, la bonne suite est un remboursement, pas une annulation (void). Si elle n'a pas eu lieu, un retry ou une annulation est sans danger. Deviner dans un sens ou dans l'autre, c'est ainsi qu'un acheteur se retrouve débité pour une commande que Parcelly croit avoir annulée.

Différée: la dépendance est connue pour être indisponible

Le quatrième signal n'est pas un verdict sur le message. Quand un circuit breaker (décrit plus bas) est ouvert, l'échec d'un appel ne dit rien de la requête : l'appel n'a jamais atteint la dépendance. J'appelle cette classe **différée**. Un message différé ne consomme pas de tentative, ne compte dans aucune limite et n'est jamais envoyé vers la DLQ pour cette raison. Le

consommateur cesse de tirer des messages, ou met la partition en pause, et le message attend là où il est jusqu'à ce que la dépendance vaille de nouveau la peine d'être appelée.

La distinction compte parce que l'alternative est arithmétique. Un breaker qui reste ouvert pendant 30 secondes, devant un handler qui autorise trois tentatives dans le handler espacées d'une seconde, laisse un message sain brûler tout son budget en à peu près trois secondes et finir dans la DLQ pour un problème qu'il n'a jamais eu.

Rendre la classification explicite

La classification doit résider quelque part où l'on peut la lire et la relire. Une table qui associe un signal de défaillance à une classe et à un traitement est l'un des petits artefacts les plus précieux d'un consommateur. Pour les appels de Payments au fournisseur, une version illustrative ressemble à ceci :

Signal	Classe	Traitement
Connexion réinitialisée, échec DNS	Transitoire	Retry avec backoff
Timeout sur un appel de capture	Inconnue	Retry avec la même clé d'idempotence, limite stricte
`429`, réponse de throttling	Transitoire	Retry, en respectant l'indication d'attente du fournisseur si elle est fournie
`5xx` du fournisseur	Transitoire	Retry avec backoff
`400`, échec de validation	Permanente	Pas de retry, envoyer vers la DLQ
Carte refusée	Permanente, résultat métier	Émettre `PaymentFailed`, pas de retry
Erreur non reconnue	Inconnue	Retry avec limite stricte, puis envoyer vers la DLQ
Circuit breaker ouvert	Différée	Aucune tentative consommée ; mettre la consommation en pause, la reprendre quand le breaker le permet

Deux mises en garde. D'abord, les codes de statut sont des indications, pas des garanties ; un fournisseur qui renvoie `500` pour une requête malformée l'a mal classée à votre place, et vous devrez l'apprendre par l'expé-

rience avec ce fournisseur. Ensuite, la table doit retomber par défaut sur inconnue, jamais sur permanente ou transitoire, pour tout signal qu'elle n'a pas encore vu. Une nouvelle erreur que personne n'a classée n'est la preuve de rien.

Le classificateur lui-même est petit :

```
// 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 avec jitter

Une fois décidé de faire un retry, la question suivante est : quand ? Faire un retry immédiatement ne laisse à une cause transitoire aucun temps pour se dissiper. Faire un retry après un délai fixe lui en laisse, mais garde tous les appelants au même pas.

L'**exponential backoff** fait croître le délai à chaque tentative : par exemple une seconde, puis deux, puis quatre, jusqu'à un plafond. Les premiers retries sont rapides, ce qui convient aux défaillances qui se dissipent en un instant. Les retries suivants sont lents, ce qui soulage une dépendance réellement indisponible. Le plafond empêche le délai de dépasser ce que le métier peut tolérer.

Le **jitter** ajoute de l'aléa à chaque délai. Sans lui, les appelants qui ont échoué au même moment calculent les mêmes délais et font leurs retries au pas, et le troupeau traverse le backoff intact. Avec lui, les retries s'étalent sur un intervalle, et la dépendance voit une traînée de requêtes au lieu d'un mur.

Il existe plusieurs schémas de jitter. Deux méritent d'être connus par leur nom. Avec le **full jitter**, le délai est une valeur aléatoire entre zéro et le plafond exponentiel de cette tentative. Avec l'**equal jitter**, la moitié du délai est fixe et l'autre moitié aléatoire, de sorte que l'attente minimale n'est jamais proche de zéro. Le full jitter répartit le mieux la charge, tandis que l'equal jitter garantit un plancher à l'attente. Je commencerais par le full jitter, et ne passerais à l'equal jitter que si la dépendance souffre de retries très rapides.

```
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
```

Cette boucle est correcte face aux défaillances qu'elle traite, mais elle est incomplète de deux manières que les sections suivantes abordent. Elle ne sait pas combien de temps il reste à l'appelant, et elle ne sait pas quelle quantité de retries le système entier est déjà en train de faire.

Si le fournisseur envoie une indication sur le moment de faire un retry, comme une valeur d'attente dans une réponse de throttling, respectez-la. Le fournisseur connaît son propre rétablissement mieux que votre formule. Traitez l'indication comme un plancher sous votre délai calculé, pas comme un substitut, et conservez le jitter pour que les appelants ayant reçu la même indication ne reviennent pas ensemble. Un plancher n'autorise pas pour autant à dormir au-delà de la deadline : si l'indication est plus longue que le temps restant, arrêtez les retries et laissez la couche suivante décider.

Deadlines et retry budgets

Le backoff avec jitter décide de l'espacement des retries. Deux limites supplémentaires décident de leur nombre.

Deadlines

Une **deadline** est le moment au-delà duquel le résultat n'est plus utile. Un flux piloté par une requête en a une naturelle : l'acheteur a les yeux fixés sur une page de paiement, et dans 10 secondes il partira. Un consommateur en arrière-plan en a une plus souple : une étiquette d'expédition qui arrive après l'heure limite du transporteur est aussi en retard qu'une étiquette qui n'arrive jamais.

Une logique de retry qui compte les tentatives mais ignore le temps peut largement dépasser le point d'utilité. Trois tentatives avec un timeout de 30 secondes chacune et des délais croissants entre elles représentent des minutes d'attente. Je préfère donner à chaque opération qui fait des retries un budget de temps total, et arrêter les retries quand le prochain délai le dépasserait, quoi qu'en dise le nombre de tentatives.

```
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)
```

La deadline doit aussi voyager. Si Orders donne 10 secondes à un flux et appelle Payments, Payments doit transmettre ce qui reste, et non démarrer ses propres 10 secondes à neuf. Sinon, chaque couche fait ses retries à l'intérieur de la patience de la couche précédente, et la somme dépasse sans bruit ce que l'utilisateur acceptera d'attendre.

Retry budgets

Une limite de tentatives par appel ne protège pas une dépendance quand de nombreux appels échouent en même temps. Chaque appel reste dans sa limite de trois retries, et la dépendance reçoit malgré tout quatre fois son trafic normal.

Un **retry budget** limite les retries en proportion du trafic global, et non par appel. Une forme courante est le token bucket : chaque appel réussi ajoute une fraction de token, chaque retry en dépense un, et quand le bucket est vide, les appels échouent rapidement sans retry. En régime stable, les retries représentent une petite fraction des requêtes. Lors d'une panne généralisée, le budget s'épuise en quelques instants et le système cesse d'amplifier sa propre défaillance.

Les chiffres sont à vous de choisir et dépendent de la charge supplémentaire que la dépendance peut absorber. Je ne proposerai pas de valeur, car toute valeur que je donnerais serait une supposition sur votre dépendance. Ce qui compte, c'est que le budget existe, qu'il soit partagé entre les appelants d'un même processus, et que son épuisement soit visible sur un tableau de bord. Un budget fondé sur un ratio et tenu par processus suffit généralement, car si chaque processus plafonne son propre ratio, le total est déjà plafonné. Un budget partagé entre instances est une option, pas une amélioration stricte : il exige un compteur distribué, et ce compteur est une nouvelle dépendance sur le chemin de défaillance, avec un mode de défaillance qui lui est propre.

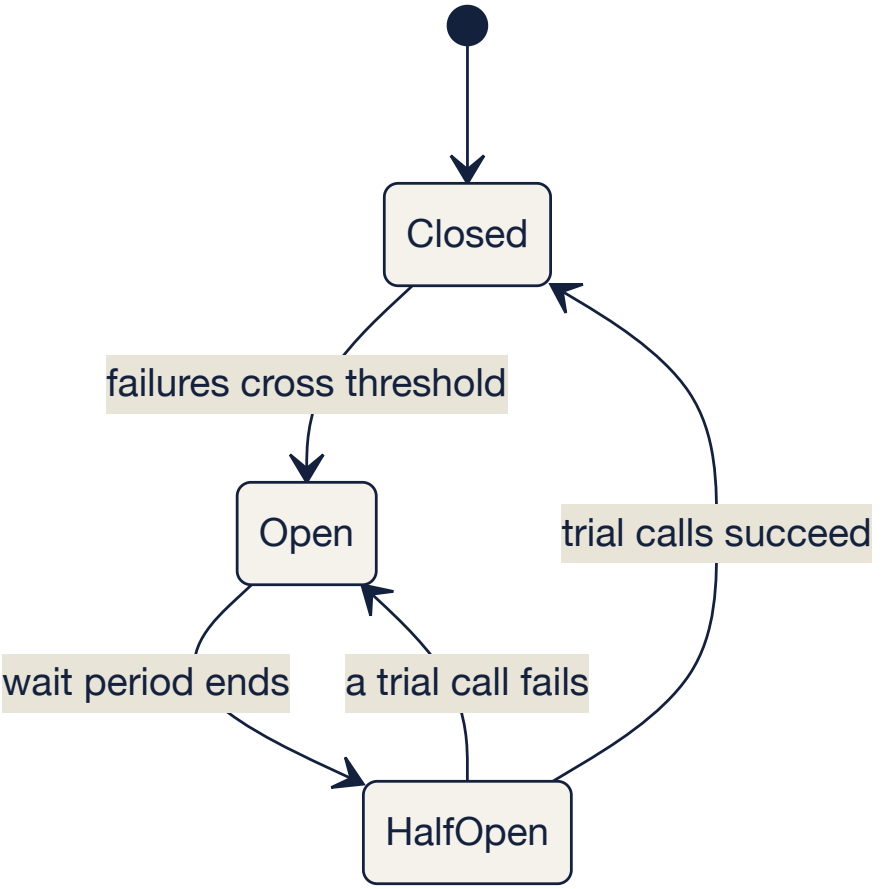
Le prix à payer est que certains appels qui auraient réussi après un retry échoueront à la place. C'est tout l'objet d'un budget : il échange la réussite individuelle contre la survie du système, et il doit le faire délibérément.

Retry storms et circuit breaker

Un **retry storm** est ce qui se produit quand les retries deviennent la charge dominante sur une dépendance défaillante. Le scénario d'ouverture chez Parcelly en était un. Il lui faut trois ingrédients : une dépendance lente ou défaillante, des appelants qui font des retries, et aucun mécanisme qui dise aux appelants de s'arrêter. Les couches l'aggravent. Si Orders fait trois tentatives auprès de Payments, et que Payments fait trois tentatives auprès du fournisseur, une capture échouée peut devenir neuf appels au fournisseur. Avec une couche supplémentaire de trois tentatives, cela devient 27.

Deux défenses s'attaquent directement au storm. Le retry budget décrit plus haut limite la quantité de retries que le système fera. Le circuit breaker décide s'il faut appeler tout court.

Un **circuit breaker** observe les résultats des appels vers une dépendance et possède trois états. Il est fermé en fonctionnement normal, et les appels passent. Quand les défaillances franchissent un seuil, il s'ouvre, et les appels échouent immédiatement sans toucher la dépendance, ce qui laisse à celle-ci de la marge pour se rétablir. Après une attente, il passe en semi-ouvert et laisse passer quelques appels d'essai. S'ils réussissent, il se ferme. Sinon, il s'ouvre de nouveau.



Le breaker et la politique de retry sont complémentaires et répondent à des questions différentes. Le backoff décide combien de temps attendre avant la tentative suivante sur un appel donné. Le breaker décide, d'après le schéma observé sur l'ensemble des appels, si les tentatives valent la peine d'être faites. Un breaker ouvert transforme la tentative suivante de la boucle de retry en échec instantané. Cet échec n'est ni transitoire ni inconnu : c'est le signal **différé** de la classification ci-dessus. La boucle de retry ne doit ni le compter, ni appliquer de backoff dessus, ni tourner en boucle dessus. Elle rend la main au consommateur, qui décide de ce qu'il cesse de faire.

Un consommateur dispose d'un choix supplémentaire qui manque à un appelant piloté par une requête. Quand le breaker est ouvert, un consommateur doit cesser de tirer des messages du broker, ou mettre la partition en pause, au lieu de consommer des messages et de les faire échouer un par

un. Les messages qui restent dans le broker sont en sécurité aussi longtemps que sa rétention le permet ; ils coûtent du stockage et apparaissent comme du retard du consommateur. Les messages consommés, ayant subi un retry ou échoués brûlent leurs compteurs de tentatives, et pourraient être envoyés vers la DLQ pour un problème qui n'a jamais été le leur. La pause est la forme côté consommateur du breaker, et je la construirais avant la plupart des autres éléments de ce chapitre.

La pause soulève deux questions de suivi. D'abord, sans trafic, personne ne teste l'état semi-ouvert. Reprenez avec une concurrence de un, ou envoyez un appel de sonde synthétique, pour que les appels d'essai soient faits par quelque chose qui sait que ce sont des essais. Ensuite, sur un broker à log, un consommateur en pause doit garder sa boucle de poll vivante, en mettant les partitions en pause mais pas le polling, sinon le groupe décidera que le consommateur est mort et réattribuera ses partitions (la fenêtre de liveness décrite plus bas).

Le coût d'un breaker est qu'il peut se tromper. Des seuils trop sensibles l'ouvrent sur un bref accroc et transforment une défaillance transitoire en panne. Des seuils trop tolérants l'ouvrent trop tard. Un breaker ajoute aussi un état que quelqu'un doit comprendre quand un flux cesse de fonctionner à trois heures du matin : la première question est de savoir si la dépendance est indisponible ou si c'est seulement le breaker qui est ouvert.

Les retries sous surcharge: backpressure

Un breaker réagit à une dépendance qui a déjà échoué. La **backpressure** est la discipline de l'étape précédente, quand une dépendance est lente, saturée ou en cours de rétablissement, et que le système doit décider quelle quantité de travail lui confier. Le Chapitre 3 la place dans le modèle de défaillances. Cette section en est le mécanisme.

Revenons au scénario d'ouverture. La lenteur du fournisseur ne venait pas d'un bug. Elle venait d'un volume de travail supérieur à ce qu'il pouvait absorber, et chaque instance de Payments continuait de l'alimenter. Un consommateur sans limite sur le travail en vol se comporte exactement ainsi : il continue de tirer, continue de lancer des appels, et continue d'alourdir une pile déjà trop lourde. Les limites ci-dessous sont celles que je mettrais en place, à peu près dans cet ordre.

- **Bornez le travail en vol.** Plafonnez le nombre de messages qu'un consommateur détient à la fois. Sur un broker à files d'attente, c'est le prefetch ou le nombre de messages en vol. Sur un broker à log, c'est le nombre d'enregistrements par poll plus le nombre de handlers en cours d'exécution. Sans plafond, ce sont la mémoire du consommateur et l'horloge de nouvelle livraison du broker qui deviennent la limite, et ce sont toutes deux de moins bonnes limites que celle que vous auriez choisie.
- **Réduisez la concurrence quand la dépendance ralentit.** Une concurrence adaptée à 50 millisecondes par appel est inadaptée à cinq secondes par appel, car le même nombre de slots contient désormais beaucoup plus de travail en attente. Abaissez le nombre d'appels concurrents quand la latence augmente, et relevez-le de nouveau, lentement, quand la latence se rétablit.
- **Mettez en pause au lieu d'échouer.** Quand la dépendance est saturée, cessez de tirer des messages, comme avec un breaker ouvert. Un consommateur en pause apparaît comme du retard, ce qui est visible et sans danger. Un consommateur qui tire et échoue brûle des tentatives.
- **Servez le trafic en direct avant les retries.** Les retries et le redrive entrent en concurrence avec les premières tentatives pour la même dépendance. Quand la capacité manque, les premières tentatives doivent l'emporter, car un retry est un travail dont le système sait déjà qu'il est en difficulté. Faites tourner les consommateurs des retry topics et du redrive avec une concurrence plus faible, ou faites-les attendre tant que le flux principal a du backlog.
- **Délestez à l'entrée.** Quelque part en amont, une limite sur le travail accepté doit exister, sinon la file d'attente grossit sans borne pendant que la dépendance se rétablit. Rejetez ou différez le travail nouveau, avec un signal clair pour l'appelant, plutôt que de le mettre indéfiniment en file derrière une dépendance incapable de le servir.

Ces limites et le retry budget sont deux visages d'une même idée. Le budget plafonne le travail supplémentaire qu'ajoutent les retries, et la backpressure plafonne le travail en vol, supplémentaire ou non. Aucun des

deux ne suffit seul : un budget sans plafond en vol laisse une dépendance lente remplir chaque slot de premières tentatives, et un plafond en vol sans budget laisse les slots se remplir de retries.

Le prix est le retard et le rejet. Un système qui déleste dit non à certains appelants, délibérément, et un système qui se met en pause livre certains messages en retard. Les deux échangent du débit contre la survie, et la personne responsable de la réponse à « quel travail attend » est le responsable métier du flux, pas l'ingénieur qui règle les chiffres. Le Chapitre 8 applique la même idée de rate limit au redrive, et le Chapitre 9 applique la borne sur le travail en vol aux lanes à clé.

Où vivent les retries

Un message peut faire l'objet de retries à plusieurs couches, et ces couches se comportent différemment. Les mélanger sans le savoir, c'est ainsi qu'un message finit par subir bien plus de retries que quiconque ne l'avait voulu, ou pas du tout.

Retry au niveau applicatif

Le consommateur fait ses retries à l'intérieur du handler : le code ci-dessus, enroulé autour d'un appel au fournisseur. Le message reste sans ack pendant les retries. Cette couche est bon marché, rapide et bien adaptée aux défaillances qui se dissipent en quelques millisecondes ou secondes. Ses limites : elle immobilise la capacité du consommateur pendant qu'elle attend ; un crash du consommateur perd l'état du retry (le broker livrera de nouveau, et le compteur de tentatives repart de zéro, ce qui explique pourquoi un message qui fait planter le processus a besoin de la détection décrite au Chapitre 8) ; et les longues attentes risquent de dépasser les propres timeouts du broker.

Sur un broker à files d'attente, le message sans ack est masqué pendant le **visibility timeout**, après quoi il est livré de nouveau. Si les retries dans le handler durent plus longtemps que ce timeout, le broker livre de nouveau le message alors que la première copie est encore en cours de retry, et deux copies s'exécutent désormais. Sur un broker à log, un consommateur qui passe trop de temps entre deux polls peut être considéré comme mort par son groupe de consommateurs et voir ses partitions réattribuées, avec

le même résultat. La règle que je suivrais : la durée totale la plus longue des retries dans le handler doit tenir confortablement dans le timeout du broker, et tout ce qui est plus long relève d'une autre couche.

Nouvelle livraison par le broker

Le consommateur fait échouer le message, et le broker le livre de nouveau. Les deux familles de brokers diffèrent ici, et la différence compte.

Sur un broker à files d'attente, la nouvelle livraison est intégrée. Le consommateur fait un nack sur le message, ou laisse expirer le visibility timeout, et le message retourne dans la file d'attente, immédiatement ou après un délai configuré, où n'importe quel consommateur concurrent peut le recevoir.

Un broker à log ne livre pas de nouveau de lui-même. La position de lecture du consommateur avance à mesure qu'il fait des polls, et l'offset dont le commit est fait ne fait qu'enregistrer l'endroit où un consommateur redémarré reprendrait (le Chapitre 2 couvre ce point). Échouer à traiter un enregistrement et ne pas faire le commit de son offset ne fait donc rien en soi : le consommateur lit l'enregistrement suivant, et s'il fait ensuite un commit, ce commit couvre aussi l'enregistrement échoué, qui est sauté sans bruit. L'enregistrement échoué n'est relu qu'après un redémarrage, un rééquilibrage ou une action explicite du consommateur.

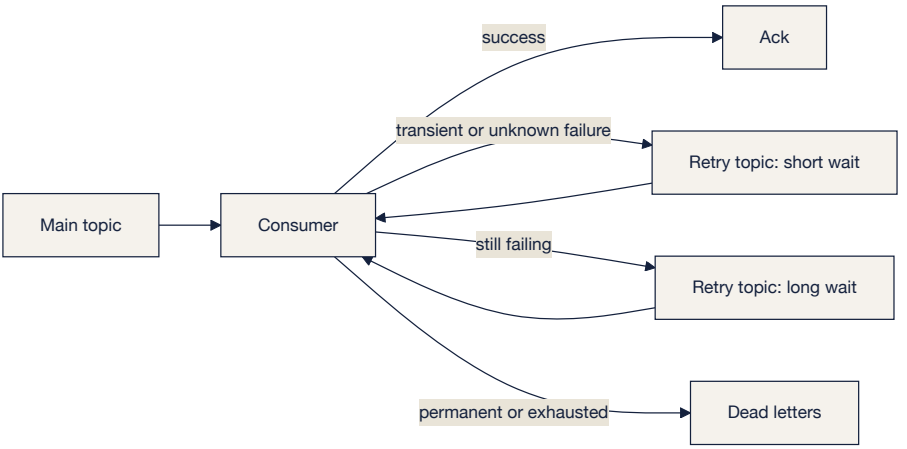
Sur un log, le consommateur doit donc choisir l'une de trois actions. Il peut revenir (seek) à l'offset échoué, ou mettre la partition en pause, et faire le retry sur place, ce qui préserve l'ordre et bloque tout ce qui se trouve derrière. Il peut publier l'enregistrement dans un retry topic et passer à la suite. Ou il peut l'écrire dans un dead-letter topic et passer à la suite. Chacune est une décision délibérée dans le code du consommateur, et non quelque chose que le broker fait pour vous.

La nouvelle livraison par le broker est durable : elle survit à un crash du consommateur. Elle est aussi grossière. Beaucoup de brokers livrent de nouveau immédiatement ou selon un calendrier fixe, sans backoff, et le nombre de tentatives n'est parfois disponible que sous la forme d'un compteur de livraisons que le consommateur doit lire et exploiter. Traitez-la comme le filet de sécurité sous les autres couches, pas comme la politique de retry principale.

Retry topics et files d'attente à délai

Quand une défaillance exige une attente plus longue que ce que le retry dans le handler peut tenir, une structure courante consiste à sortir le message du flux principal et à le laisser revenir plus tard. Un consommateur qui fait échouer un message le publie, avec un compteur de tentatives incrémenté et une heure « pas avant », dans un **retry topic** (ou une retry queue), puis fait un ack sur l'original. Un consommateur distinct, ou le même sur un autre topic, le reprend quand l'heure arrive. Les brokers qui prennent en charge la livraison différée, ou les délais par message, offrent cela directement. Là où ils ne le font pas, un agencement courant est une échelle de retry topics, chacun avec une attente fixe, un par étape du backoff.

Les attentes sont fixes par topic pour une raison. Sur un log, un consommateur ne peut pas sauter un enregistrement qui n'est pas encore dû : il doit attendre au head jusqu'à ce que l'enregistrement soit prêt. Si chaque enregistrement d'un topic a la même attente, le topic est déjà dans l'ordre des échéances, et attendre au head ne coûte rien de plus. Un topic qui mélangerait des délais courts et longs retiendrait les courts derrière les longs. Les heures « pas avant » par message ne fonctionnent que sur les brokers dotés d'une livraison différée native.



Cette couche libère le flux principal, si bien qu'un message défaillant ne retient plus les messages qui le suivent. Elle survit aux redémarrages, car l'état vit dans le broker. Elle vous donne aussi un endroit où regarder : la

profondeur du retry topic est une mesure directe de la quantité de travail qui attend une dépendance en cours de rétablissement.

Elle coûte davantage de machinerie. Il y a plus de topics à créer, nommer, surveiller et sécuriser. La republication est elle-même une opération de production qui peut échouer, ce qui ramène le dual write du Chapitre 1 ; l'ordre compte donc : publiez d'abord dans le retry topic, puis faites un ack sur l'original. Si le processus meurt entre les deux, le message existe deux fois. C'est un doublon, que le consommateur doit tolérer de toute façon.

Trois règles empêchent l'échelle d'endommager ce qu'elle est censée protéger :

- **Un topic par consommateur.** Un retry topic appartient à un seul groupe de consommateurs ou à un seul abonnement. Si un autre groupe peut le lire, ce groupe retraite un message qu'il n'a jamais fait échouer.
- **Préservez l'identité.** Le message republié conserve la clé d'origine, l'`event_id` et les en-têtes métier. Une nouvelle clé casse l'ordre, et un nouvel `event_id` casse l'inbox.
- **Dimensionnez la rétention selon l'attente.** La rétention d'un retry topic doit dépasser l'attente la plus longue plus le retard prévu sur ce topic, sinon le message expire pendant qu'il attend.

Le coût le plus important est l'**ordre des messages**, que couvre la section suivante.

Choisir les couches

En pratique, un consommateur utilise les trois. Un court retry au niveau applicatif absorbe les accrocs, comme une connexion réinitialisée ou un deadlock, qui se dissipent en une ou deux secondes. Un retry topic ou une file d'attente à délai couvre les défaillances qui demandent des minutes. La nouvelle livraison par le broker couvre les crashs. Une dead-letter queue (DLQ) termine la chaîne, en recevant ce qui a épuisé ses tentatives ou ne pourra jamais réussir ; le Chapitre 8 couvre sa conception.

Les couches doivent être conçues ensemble, avec un seul budget de tentatives total, et non trois budgets indépendants. Si chaque couche autorise trois tentatives, un message peut être essayé 27 fois, et personne ne l'a

prévu. La politique présentée à la fin de ce chapitre rend la multiplication explicite et la plafonne.

Retries et ordre des messages

Faire un retry d'un message change l'ordre dans lequel les messages sont traités. Cela découle de la superposition des couches décrite ci-dessus, et passe inaperçu jusqu'à ce que cela cause des dégâts.

Prenez le consommateur d'Inventory chez Parcelly, qui reçoit `OrderPlaced` puis `OrderCancelled` pour la même commande, dans cet ordre. Supposons que le traitement de `OrderPlaced` échoue de manière transitoire et que le consommateur le déplace vers un retry topic, puis passe au traitement de `OrderCancelled`. L'annulation ne trouve aucune réservation à libérer, car pour autant qu'Inventory le sache, aucune n'existe encore. Puis le `OrderPlaced` du retry arrive et crée une réservation que rien ne libérera jamais. L'état final est faux, et chaque étape prise isolément semblait raisonnable.

Il existe deux options honnêtes, et chacune a son prix.

- **Bloquer et faire le retry sur place.** Sur un broker à log, le consommateur revient à l'offset défaillant, ou met la partition en pause, et fait le retry, avec backoff, avant de toucher à quoi que ce soit derrière. L'ordre est préservé pour cette partition. Le prix est que toute la partition attend derrière un seul message, si bien qu'un seul message défaillant peut bloquer chaque agrégat de cette partition. C'est le bon choix quand l'ordre compte plus que le débit, et c'est pourquoi il est si important de classer rapidement une défaillance permanente : un poison message en tête d'une partition bloque tout ce qui se trouve derrière lui.
- **Mettre de côté et accepter le réordonnancement.** Le message défaillant part dans un retry topic et le consommateur continue. Le débit est préservé. Le prix est que le consommateur doit tolérer les arrivées dans le désordre, grâce à des techniques comme les vérifications de version, ou en mettant aussi de côté (parking) chaque message ultérieur du même agrégat jusqu'à ce que le précédent réussisse. Le Chapitre 9 couvre ces techniques.

Je choisirais stream par stream. Pour un stream d'événements où les messages d'un même agrégat doivent s'appliquer dans l'ordre, je ferais le retry sur place et j'investirais dans une classification rapide et une sortie vers la DLQ. Pour un stream de faits indépendants, où l'ordre entre messages n'a pas d'importance, mettre de côté est bon marché et garde le flux en mouvement.

Sur les brokers à file d'attente sans fonction de regroupement, l'ordre entre consommateurs concurrents est faible dès le départ, donc les retries dégradent légèrement une propriété existante plutôt que de casser une garantie. Même ainsi, un consommateur qui suppose un ordre ne doit pas compter sur une file d'attente pour le fournir. Si vous utilisez une fonction de regroupement (files FIFO, message groups, sessions), le compromis est le même que sur un log : un message défaillant bloque son groupe jusqu'à ce qu'il réussisse ou soit envoyé vers la DLQ, de la même manière qu'un retry sur place bloque une partition.

Retries sans idempotence

Chaque retry est une nouvelle tentative d'un travail dont le résultat antérieur peut être inconnu. Quand la tentative précédente a échoué avant de faire quoi que ce soit, le retry est inoffensif. Quand elle a échoué après avoir fait le travail, le retry fait le travail deux fois.

Ce n'est pas un cas limite rare. Reprenez le scénario d'ouverture. Payments appelle le fournisseur pour capturer 4 990 unités mineures d'EUR. Le fournisseur capture l'argent et envoie sa réponse, et la réponse se perd. Payments voit un timeout. Le timeout est classé comme inconnu, et la politique dit de faire un retry. La seconde requête de capture part. Que l'acheteur soit débité une ou deux fois dépend désormais entièrement de l'existence de quelque chose qui rende la seconde requête reconnaissable comme une répétition.

Ce quelque chose est une **clé d'idempotence** : un identifiant stable, envoyé avec la requête, qui permet au destinataire de reconnaître qu'il a déjà vu ce travail. Pour l'appel au fournisseur, c'est la même clé à chaque tentative pour la même capture, qu'un fournisseur respectant de telles clés trai-

tera comme une seule capture. Pour la consommation d'événements, c'est normalement l'`event_id`, vérifié contre l'inbox dans la même transaction que le travail. Le Chapitre 7 construit les deux.

> **Avertissement :** Une politique de retry n'est sûre que sur une opération idempotente. Avant d'activer les retries sur un appel qui a un effet, comme débiter une carte, envoyer un e-mail ou réserver du stock, établissez ce qui rend la seconde tentative inoffensive. Si rien ne le fait, le bon nombre de retries automatiques est zéro, et la défaillance doit aller à une personne. Comme un broker at-least-once livrera de nouveau après un crash ou un timeout quoi que veuille le handler, ce « zéro » doit être imposé par le handler : il doit détecter qu'il a déjà vu ce message, ou faire l'ack avant l'effet et accepter l'at-most-once, puis acheminer la défaillance vers une personne.

La dépendance joue dans les deux sens. L'idempotence est ce qui rend les retries sûrs, et les retries sont ce qui rend l'idempotence nécessaire. Un système qui fait des retries agressifs sans consommateurs idempotents produira des effets en double à un rythme proportionnel à son taux de défaillance, c'est-à-dire aux moments où le système est déjà sous tension et où personne n'a le temps de nettoyer.

Il existe aussi un cas plus subtil. Une opération peut être idempotente pour une même clé tout en restant dangereuse pour un retry, parce que la clé change. Si un handler construit un nouvel identifiant à chaque tentative, comme un nouvel UUID à chaque appel, le fournisseur voit chaque fois des requêtes distinctes. La clé doit être dérivée une seule fois, à partir d'entrées stables comme l'`event_id` ou la commande, et réutilisée à chaque tentative.

Une politique de retry pour Parcelly

Pour rassembler les morceaux, voici ce que j'écrirais pour le consommateur de `CapturePayment` dans Payments. Les chiffres sont des illustrations, pas des recommandations, et votre dépendance aura besoin des siens.

```
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
```

Lisez-la comme un ensemble de décisions, chacune avec sa raison. La couche dans le handler absorbe les accrocs courts. Les retry topics couvrent un fournisseur indisponible pendant des minutes. La deadline dit quand le travail cesse de valoir la peine, et le responsable de cette décision est le métier, pas l'ingénieur. Une capture périmée peut devoir devenir un `OrderCancelled`, mais la compensation dépend de ce qui s'est réellement passé chez le fournisseur. Si la dernière tentative a dépassé son timeout, la capture a peut-être réussi, et alors la bonne suite est un `PaymentRefunded`, pas un `PaymentVoided`. C'est pourquoi la politique demande d'abord le statut. La clé, dérivée une seule fois, est ce qui empêche toute la structure de débiter deux fois un acheteur.

Deux lignes de la politique méritent un second regard. `maxTotalAttempts` existe parce que les couches se multiplient : trois tentatives dans le handler, multipliées par un premier passage plus trois étapes de retry topic, font jusqu'à 12 appels au fournisseur, avant même de compter toute nouvelle livraison après un crash. Un total unique de 8 (illustratif) est le chiffre dont quelqu'un est réellement responsable. Et l'horizon complet des retries, deux heures ici, doit être plus court que la durée pendant laquelle

le fournisseur se souvient des clés d'idempotence. Les fournisseurs conservent les clés pendant une fenêtre limitée, et un retry qui arrive après cette fenêtre est une nouvelle requête portant une ancienne clé.

Quand le redrive du Chapitre 8 remet en circulation une capture morte, le compteur de tentatives repart de zéro, mais un compteur de redrive distinct demeure, de sorte que l'historique n'est pas perdu et qu'un message qui revient sans cesse est visible comme tel.

Remarquez ce qui manque : un paramètre unique de « nombre de retries ». Dans un système à plusieurs couches, ce chiffre unique cache au moins quatre décisions.

Compromis et coûts

Les retries sont l'un des moyens les moins chers d'améliorer la fiabilité apparente d'un système, et l'un des plus faciles à surutiliser. Leurs coûts sont précis.

Latence. Chaque retry ajoute du temps d'attente. Le backoff et le jitter répartissent la charge, mais ils allongent aussi le temps nécessaire pour réussir. Un flux qui fait des retries pendant des minutes peut aboutir correctement et rester inutile pour l'acheteur parti au bout de 10 secondes. Là où une personne attend, je privilégierais un échec rapide et un message clair plutôt qu'un long retry silencieux.

Amplification. Les retries multiplient la charge, et les retries en couches la multiplient encore. Une politique raisonnable dans chaque service peut être dangereuse dans la chaîne. Quelqu'un doit être responsable du budget de tentatives de bout en bout, car aucun service ne peut le voir à lui seul.

Capacité. Un consommateur qui fait des retries sur place occupe son slot pendant qu'il attend. Lors d'une longue panne, tous les consommateurs peuvent se retrouver en attente, et le système cesse de progresser même sur du travail qui réussirait. Mettre la consommation en pause via un breaker, borner le travail en vol avec la backpressure et déplacer les longues attentes vers des retry topics sont les défenses habituelles.

Défaillances cachées. Un retry qui finit par réussir ressemble, vu de l'extérieur, à une réussite. Si vous ne comptez pas les retries et les tentatives, une dépendance peut se dégrader pendant des semaines derrière une poli-

tique qui l'absorbe sans bruit. Le Chapitre 13 couvre ce qu'il faut mesurer : le nombre de tentatives, le taux de retry et l'âge du plus ancien message ayant subi un retry ont leur place sur un tableau de bord.

Complexité. Les retry topics, les files d'attente à délai et les budgets sont de l'infrastructure que quelqu'un doit construire, nommer, surveiller et expliquer. Pour un petit système avec un consommateur et une dépendance, un retry dans le handler avec backoff et une sortie vers la DLQ peut suffire, et une échelle de retry topics peut représenter plus de machinerie que le problème n'en mérite.

Quand ne pas faire de retry. Ne faites pas de retry sur les défaillances permanentes. Ne faites pas de retry sur les opérations à effet qui ne sont pas idempotentes. Ne faites pas de retry au-delà de la deadline. Ne faites pas de retry quand un circuit breaker dit que la dépendance est indisponible, et ne facturez pas de tentative à ce message pour cela. Ne faites pas de retry à l'intérieur d'une couche qui se trouve déjà dans le retry d'une autre couche, à moins d'avoir fait le calcul de la multiplication. Et ne faites pas de retry pour éviter une décision : une défaillance qui revient sans cesse est une information, et la bonne réponse est souvent de s'arrêter et de regarder.

Erreurs courantes

Faire un retry de tout, de la même manière

C'est tentant parce qu'une seule boucle est le code le plus court. Elle traite un refus de carte comme une connexion réinitialisée et gaspille de la capacité sur des défaillances qui ne peuvent pas réussir. Classez d'abord, et donnez à chaque classe son propre traitement.

Délais fixes sans jitter

Une attente constante paraît prévisible et simple. Elle synchronise chaque appelant ayant échoué en même temps en un pic qui se répète. Utilisez l'exponential backoff avec jitter, et, à mon avis, le full jitter comme point de départ.

Compter les tentatives mais pas le temps

Une limite de tentatives ressemble à une borne de sécurité. Avec de longs timeouts et des délais croissants, elle peut encore tourner pendant des minutes au-delà du point d'utilité. Fixez une deadline pour l'opération entière et arrêtez-vous quand le prochain délai la dépasserait.

Faire des retries à chaque couche

Le responsable de chaque couche ajoute un retry raisonnable, et la multiplication est invisible pour chacun d'eux. Décidez d'un seul budget de tentatives total pour la chaîne, et placez les retries là où ils coûtent le moins.

Faire des retries pendant que le breaker est ouvert

Il est tentant de laisser tourner la boucle, puisque la boucle fonctionnait. Un consommateur qui tire des messages et les fait échouer contre une dépendance morte consomme leurs tentatives et peut les envoyer à tort vers la DLQ. Traitez le breaker ouvert comme un signal différé qui ne coûte aucune tentative, mettez la consommation en pause et laissez les messages attendre dans le broker.

Laisser un consommateur tirer sans limite

Un consommateur sans borne est le comportement par défaut, et il donne l'impression d'un débit maximal. Face à une dépendance lente, il remplit chaque slot de travail en attente et alimente le retry storm. Bornez les messages en vol, abaissez la concurrence quand la latence augmente, et laissez les premières tentatives l'emporter sur les retries.

Ignorer les timeouts du broker

Les retries dans le handler semblent locaux au handler. S'ils durent plus longtemps que le visibility timeout ou que la fenêtre de liveness du groupe, le broker livre de nouveau ou réattribue, et deux copies du travail s'exécutent en même temps. Gardez la durée totale dans le handler bien en deçà des limites du broker.

Faire un retry en dépassant un message qui doit rester dans l'ordre

Déplacer un message défaillant vers un retry topic est une solution rapide pour une partition bloquée. Cela réordonne les événements de cet agrégat. Décidez d'abord de l'exigence d'ordre, puis choisissez entre bloquer et mettre de côté.

Faire un retry d'effets non idempotents

La boucle de retry s'ajoute en un après-midi, et l'idempotence est repoussée. Le premier timeout après un appel réussi produit un effet en double. Établissez ce qui rend la seconde tentative inoffensive avant d'activer le premier retry.

Générer une nouvelle clé à chaque tentative

Cela ressemble à de la rigueur, puisque chaque requête est identifiée de manière unique. Cela va à l'encontre du but, car le destinataire ne peut pas reconnaître une répétition. Dérivez la clé une seule fois à partir d'entrées stables et réutilisez-la.

Ne pas compter les retries

Si seules les défaillances finales sont journalisées, une dépendance qui se dégrade reste invisible jusqu'à ce que les retries ne suffisent plus. Enregistrez chaque tentative, avec sa classe et son délai.

Points clés

- Un retry est une promesse de refaire un essai, pas une promesse de réussir ; la première question face à toute défaillance est donc de savoir si un nouvel essai peut aider.
- Classez les défaillances en transitoires, permanentes ou inconnues, consignez la classification dans une table, et traitez les défaillances non reconnues comme inconnues, avec une limite plus stricte.
- Utilisez l'exponential backoff avec jitter, bornez l'opération entière par une deadline, et partagez un retry budget pour que les retries ne puissent pas devenir la charge dominante sur une dépendance défaillante.

- Un circuit breaker décide s'il faut appeler tout court ; un breaker ouvert est un signal différé qui ne coûte aucune tentative, et pour un consommateur, la forme la plus forte consiste à mettre la consommation en pause. La backpressure fait le même travail plus tôt, en bornant le travail en vol et en laissant le trafic en direct l'emporter sur les retries et le redrive.
- Les retries peuvent vivre dans le handler, dans la nouvelle livraison par le broker et dans des retry topics, et ils doivent être conçus ensemble sous un seul budget de tentatives, avec une durée dans le handler maintenue à l'intérieur des timeouts du broker.
- Mettre de côté un message défaillant réordonne les événements de son agrégat ; choisissez entre bloquer sur place et tolérer le réordonnancement selon l'exigence d'ordre du stream.
- Une politique de retry n'est sûre que sur des opérations idempotentes, avec une clé d'idempotence dérivée une seule fois et réutilisée à chaque tentative.

Les retries décident combien de retries vous faites, mais ils ne peuvent pas décider si essayer deux fois est inoffensif. Le Chapitre 7 construit l'autre moitié : le consommateur idempotent, qui transforme un doublon d'une catastrophe en non-événement.

Chapitre 7 — Idempotence

Imaginez le service Ledger de Parcelly au milieu d'un déploiement progressif, pendant que ses consommateurs sont redémarrés un par un. Un consommateur a lu un événement `PaymentCaptured`, écrit une écriture comptable de 4 990 unités mineures d'EUR, et s'apprête à faire l'ack du message quand le processus reçoit son signal d'arrêt. L'ack ne quitte jamais la machine. Le broker, qui fait exactement ce que promet la livraison at-least-once, remet le même message au prochain consommateur qui le demande.

Le second consommateur lit l'événement, ne trouve rien d'inhabituel, et écrit une seconde écriture comptable. Le grand livre est append-only et correct par construction ; il enregistre donc désormais, fidèlement et définitivement, que l'acheteur a payé deux fois. Rien n'a planté. Aucune alerte ne s'est déclenchée. Chaque composant s'est comporté comme prévu, et les comptes sont faux.

Notifications vit sa propre version du même après-midi. Un acheteur reçoit trois fois l'e-mail « votre colis est programmé », parce que le message a été livré de nouveau après qu'un handler lent a dépassé son visibility timeout. Celui-là est gênant. Celui du grand livre coûte cher.

Les chapitres précédents ont établi que les doublons sont normaux : le broker livre de nouveau, le relay publie deux fois après un failover, et les retries du Chapitre 6 sont des doublons par construction. Ce chapitre répond à la question que ces chapitres laissent ouverte. Quand le même message arrive deux fois, que doit faire exactement un consommateur pour que la seconde arrivée ne change rien ?

Le problème, et pourquoi il est difficile

L'objectif paraît modeste : traiter chaque message une seule fois, pour autant que le monde puisse en juger. La difficulté est que le consommateur ne peut pas savoir s'il a déjà vu un message sans s'en souvenir, et se souve-

nir est en soi une écriture qui peut échouer au mauvais moment.

Commençons par l'origine des doublons, car la liste est plus longue que le broker.

Source	Comment le doublon apparaît
Nouvelle livraison par le broker	L'ack a été perdu, le visibility timeout a expiré, ou un rééquilibrage a déplacé la partition avant le commit de l'offset
Relay	Un relay a publié une ligne, a planté avant de la marquer comme publiée, et l'a publiée de nouveau après le redémarrage (Chapitre 5)
Retry applicatif	Un consommateur a fait un retry d'un appel, ou un producteur a fait un retry d'une publication dont la première tentative avait en réalité réussi (Chapitre 6)
Redrive et replay	Quelqu'un a remis en circulation des dead letters, ou a délibérément relu un log d'événements à partir d'un offset antérieur
Personnes et clients	Un acheteur a double-cliqué sur « Payer », une application mobile a fait un retry sur un réseau instable, un expéditeur de webhook a livré deux fois

La plupart de ces sources produisent deux fois le *même* message, avec le même `event_id`. Quelques-unes produisent deux messages différents qui expriment la même intention, et cette distinction comptera quand nous choisirons une clé.

La solution tentante est de regarder avant d'agir. Demander si ce message a été traité, et sinon, le traiter. Je veux nommer cela l'anti-pattern **check-then-act**, parce qu'il paraît correct et échoue sans bruit.

```
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)
```

Lisez-le sous l'angle des défaillances. Si le processus meurt après `postLedgerEntry` et avant `inboxInsert`, l'écriture existe et l'inbox l'ignore, donc la nouvelle livraison l'enregistre de nouveau. Inversez les deux écritures et la défaillance s'inverse aussi : un crash après

`inboxInsert` et avant l'écriture comptable marque le message comme traité alors qu'il ne l'a pas été, et l'effet est perdu pour toujours. Aucun des deux ordres n'est sûr, parce que les deux écritures ne forment pas un seul acte atomique.

Il existe une seconde défaillance qui n'a rien à voir avec les crashs. Deux consommateurs peuvent recevoir le même message presque au même instant, par exemple quand une première tentative lente survit à son visibility timeout. Tous deux font la vérification, tous deux ne trouvent rien, et tous deux continuent. La fenêtre entre « vérifier » et « agir » est petite, et avec un volume suffisant, elle est atteinte couramment.

Le vrai problème a donc deux moitiés. La mémoire de ce qui a été traité doit être écrite de manière atomique avec l'effet, et cette mémoire doit pouvoir arbitrer entre des tentatives concurrentes. Le reste du chapitre construit les deux, en commençant par le cas où vous n'avez besoin ni de l'un ni de l'autre.

Idempotence naturelle et idempotence forcée

Une opération est **idempotente** quand l'appliquer deux fois laisse le système dans le même état que l'appliquer une fois. Certaines opérations sont idempotentes par leur forme, et je les chercherais avant de construire la moindre machinerie.

Affecter une valeur est idempotent : `set order.status = "PAID"` peut s'exécuter 10 fois et la commande est toujours `PAID`. Supprimer une ligne par son identifiant est idempotent. Insérer une ligne avec une clé primaire dérivée de l'événement, en ignorant le conflit, est idempotent. Écrire l'état complet d'un agrégat (« l'expédition est maintenant `DISPATCHED`, avec ce numéro de suivi ») est idempotent. J'appelle cela l'**idempotence naturelle** : le doublon n'a pas d'effet supplémentaire en raison de ce qu'est l'opération, et non de quoi que ce soit que vous ayez ajouté.

D'autres opérations ne le sont pas. `balance = balance + 4990` s'applique deux fois comme deux dépôts. `send email` envoie deux e-mails. `append a ledger entry` en ajoute deux. `increment the counter of units sold` compte l'unité deux fois. Appeler l'endpoint de capture d'un fournisseur de paiement sans clé peut débiter deux fois. Pour ces opérations, l'idempo-

tence doit être ajoutée, ce que j'appelle l'**idempotence forcée** : vous enregistrez que le travail a été fait et consultez cet enregistrement avant de le refaire.

L'idempotence naturelle a deux limites honnêtes. La première est qu'elle dépend discrètement de l'ordre. Affecter `status = "PAID"` est idempotent, mais un doublon retardé d'un événement plus ancien peut écraser un statut plus récent : un `OrderPlaced` livré de nouveau, qui arrive après `OrderConfirmed` et remet le statut à `PLACED`. L'opération est idempotente, et le système a pourtant tort. Le Chapitre 9 couvre l'ordre et les vérifications de version ; pour l'instant, retenez qu'« affecter une valeur » n'est sûr face aux doublons que lorsque la valeur ne peut pas devenir périmée.

La seconde limite est qu'il est tentant de la surestimer. Un handler qui « ne fait que mettre à jour une ligne » écrit souvent aussi un enregistrement d'audit, émet un événement de suivi ou incrémente un compteur, et chacun de ces effets est un incrément déguisé. Quand je relis un consommateur, je liste chacun de ses effets et je demande séparément, pour chacun, s'il survit à une seconde exécution. Un seul effet non idempotent suffit pour avoir besoin de l'inbox.

Ma règle empirique est de préférer l'idempotence naturelle là où l'effet le permet, et d'écrire dans le handler pourquoi c'est le cas. Quand l'effet est un compteur, un ajout, un e-mail ou un appel externe, utilisez la forme forcée. Et en cas de doute, utilisez quand même l'inbox : son coût est faible et prévisible, ce qui n'est pas le cas du coût d'une erreur.

Choisir la clé d'idempotence

Une **clé d'idempotence** est ce que le consommateur utilise pour reconnaître un travail répété. Avant de construire la table, vous devez décider de ce qu'elle contient, et ce choix détermine les doublons que vous pourrez intercepter.

L'event_id

Pour la consommation d'événements, le choix par défaut est l'`event_id` de l'enveloppe. Il est unique par événement et, surtout, **stable d'une nouvelle livraison à l'autre** : chaque copie du message que le broker vous remet porte la même valeur. Le producteur l'attribue une seule fois, quand

l'événement est écrit dans l'outbox (Chapitre 4), de sorte qu'un relay qui publie deux fois publie deux fois le même `event_id`. C'est exactement la propriété dont une vérification de déduplication a besoin.

L'`event_id` identifie un *message*. Il répond à la question « ai-je déjà traité ce fait précis ? » et à rien de plus large.

La clé métier

Il arrive que deux messages différents décrivent la même intention. Supposons qu'Orders fasse un retry d'un command `CapturePayment` après un timeout et que, à cause d'un bug ou d'un choix de conception, chaque retry soit écrit comme un nouveau command avec un nouveau `command_id`. Payments reçoit maintenant deux commands avec des identifiants différents qui signifient tous deux « capturer le paiement de la commande 81724 ». Dédupliquer sur l'identifiant du message n'intercepte ni l'un ni l'autre.

Le remède est une **clé métier** : un identifiant dérivé du sens du travail plutôt que de l'enveloppe. Pour une capture, ce peut être le `payment_id` plus l'opération, par exemple `capture:payment-5521`. Pour une écriture comptable, ce peut être l'identifiant du paiement plus le type d'écriture. Une clé métier dit « cette unité de travail peut avoir lieu une fois », quel que soit le nombre de messages qui la demandent.

Aspect	`event_id` / `command_id`	Clé métier
Intercepte la nouvelle livraison du même message	Oui	Oui
Intercepte deux messages pour la même intention	Non	Oui
Qui l'attribue	Le producteur, automatiquement	Une décision humaine par opération
Risque	Manque les doublons logiques	Une clé trop grossière rejette un travail légitime
Où elle est stockée	Table d'inbox générique	Souvent une contrainte d'unicité sur la table du domaine

Le risque de l'avant-dernière ligne mérite un cas concret. Si la clé métier des remboursements n'est que le `payment_id`, un second remboursement partiel, légitime, du même paiement est rejeté comme doublon. La clé doit inclure tout ce qui rend deux requêtes *réellement différentes* : ici, peut-être une clé d'idempotence de remboursement fournie par l'appelant. Choisir la clé revient à décider ce que « le même travail » signifie dans votre domaine, ce qui est une décision de conception et non un réglage de framework.

À mon avis, les couches fonctionnent bien ensemble. Utilisez l'`event_id` dans un inbox générique comme base pour chaque consommateur, et ajoutez une contrainte d'unicité sur une clé métier dans la table du domaine partout où deux messages différents pourraient vraisemblablement signifier la même chose. L'inbox protège contre le broker et le relay. La contrainte protège contre le reste du système, y compris ses bugs.

Le pattern inbox

L'**inbox** est la table des identifiants de messages traités, côté consommateur. Le Chapitre 4 a donné au producteur un transactional outbox pour que le commit d'un changement d'état et celui de son événement se fassent ensemble. L'inbox en est l'image miroir : le consommateur fait le commit de son changement d'état et de la trace du traitement du message ensemble, dans une seule transaction locale de sa propre base de données.

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

La colonne `consumer` permet à un service de faire tourner plusieurs consommateurs indépendants sur une même table, de sorte que le consommateur de comptabilisation de Ledger et son consommateur de reporting gardent chacun leur propre historique. C'est pourquoi la clé primaire est la paire `(consumer, event_id)`. Avec l'`event_id` seul, le premier consommateur à traiter un message ferait échouer l'insertion du second, et l'effet du second consommateur serait sauté comme s'il s'agissait

d'un doublon. Un service avec un seul consommateur par table peut supprimer la colonne et prendre l'`event_id` pour clé, ce qui est la forme plus simple utilisée ailleurs dans le livre.

Le handler écrit la ligne d'inbox et le changement métier dans la même transaction.

```
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)
```

Parcourez les défaillances. Si le processus meurt avant le commit, ni l'écriture comptable ni la ligne d'inbox n'existent, donc la nouvelle livraison traite le message depuis le début. S'il meurt après le commit et avant l'ack, les deux existent, et la nouvelle livraison prend le chemin rapide : la vérification trouve la ligne, l'ack est fait sur le message, et rien n'est écrit. L'effet et sa mémoire ne peuvent jamais diverger, parce que leur commit se fait d'un seul tenant.

La vérification en tête est une optimisation. Elle économise une transaction dans le cas courant d'un doublon, et peut être supprimée sans perdre en exactitude. Le filet de sécurité est la contrainte d'unicité à l'intérieur de la transaction, et c'est là que se joue le véritable arbitrage.

À l'intérieur d'une transaction, l'ordre des écritures ne change rien à l'atomicité ; la mise en garde sur l'inversion des deux écritures ne s'applique donc qu'à la version non sûre présentée plus haut. Il change en revanche le coût. Dans cette version, un doublon concurrent fait tout le travail avant d'échouer sur l'insertion, et dans certaines bases de données une violation annule toute la transaction, si bien que le handler ne peut pas s'en remettre à l'intérieur de celle-ci. Je réserverais d'abord : insérer la ligne d'inbox avec une sémantique « on conflict do nothing », regarder si une ligne a

été insérée, et sauter l'effet sinon. Le doublon attend alors sur la clé, ne fait aucun travail et ne lève jamais d'erreur. L'exemple de la section suivante utilise cette forme.

Un mot sur `ack`, car le terme cache une différence entre les deux familles. Sur un broker à files d'attente, il fait un ack sur un message. Sur un broker à log, c'est généralement le commit d'un offset, qui signifie « tout jusqu'ici », et un consommateur qui traite une partition en parallèle peut faire un commit au-delà d'un message qu'il n'a pas terminé, et le perdre. L'inbox protège contre les doublons, pas contre cela, et le Chapitre 9 examine ce cas là où l'ordre et le parallélisme se rencontrent.

Événements de suivi émis par un consommateur

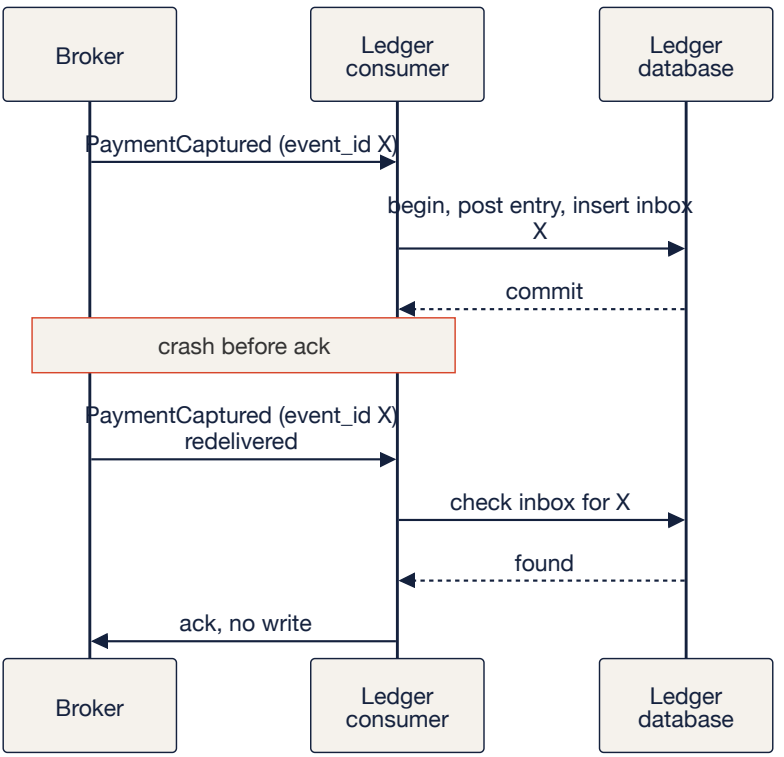
Beaucoup de consommateurs font plus que modifier leur propre état : ils annoncent ce qu'ils ont fait. Ledger enregistre une écriture et émet `LedgerEntryPosted`. Cet événement de suivi est un effet comme un autre, et il doit survivre à un crash exactement une fois, ni zéro fois ni deux fois.

Un handler qui publie l'événement de suivi directement après son commit le perd définitivement en cas de crash. La nouvelle livraison prend le chemin rapide de l'inbox, trouve la ligne, fait l'ack et saute la publication. L'état est juste, et l'événement n'a jamais existé. La correction est l'outbox du Chapitre 4 appliquée côté consommateur : la ligne d'inbox, le changement d'état et l'insertion dans l'outbox font l'objet d'un seul commit dans une transaction locale.

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

Si le processus meurt avant le commit, rien n'existe et la nouvelle livraison fait tout. S'il meurt après le commit, l'événement de suivi attend dans l'outbox et le relay le publie, tandis que la nouvelle livraison trouve la ligne d'inbox et ne fait rien. L'outbox du consommateur offre la même publication at-least-once que celle du producteur ; quiconque consomme `LedgerEntryPosted` a donc besoin de sa propre inbox. Le Chapitre 10 s'appuie sur cette règle quand une étape de saga réagit à un événement et émet le suivant.

C'est aussi pourquoi l'inbox vit dans la *même base de données* que l'état qu'elle protège. Si la trace des messages traités se trouvait dans un cache ou dans une autre base de données, nous reviendrions à deux écritures sans transaction commune, c'est-à-dire au dual write du Chapitre 4, côté consommateur. L'inbox n'est pas une technique qu'on peut greffer de l'extérieur sur le stockage de données du service.



Regardez la note au milieu : la fenêtre de crash se situe entre le commit et l'ack, et c'est la ligne d'inbox qui transforme la nouvelle livraison en opération sans effet.

Doublons concurrents

La séquence ci-dessus suppose que le doublon arrive une fois la première tentative terminée. La concurrence est plus difficile, et c'est la contrainte d'unicité qui vous sauve.

Imaginez deux consommateurs qui traitent le même message en même temps. Tous deux ouvrent une transaction. Tous deux font le travail. Tous deux tentent d'insérer la ligne d'inbox. La base de données laisse exactement l'un d'eux faire le commit ; l'autre reçoit une violation de contrainte d'unicité, sa transaction est annulée, et son écriture comptable disparaît avec elle. Le perdant laisse alors le message sans ack, ou fait l'ack après une nouvelle vérification, et dans les deux cas le grand livre contient une seule écriture.

C'est tout le mécanisme, et il vaut la peine de dire ce qu'il n'exige pas : aucun verrou distribué, aucune coordination entre les deux consommateurs, aucune hypothèse sur le timing. La contrainte de la base de données est un point d'arbitrage unique par lequel les deux tentatives doivent passer.

```
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
```

Savoir s'il faut faire l'ack d'une violation d'unicité ou faire un retry est une petite décision qui mérite d'être prise délibérément. Si la violation porte sur la clé de l'inbox, le message a été traité par une transaction dont le commit est fait, et faire l'ack est correct. Si elle porte sur une autre contrainte à l'intérieur de `applyChange`, l'erreur signifie tout autre chose,

et faire l'ack cacherait un vrai bug. À mon avis, le handler doit les distinguer, ce qu'il peut faire en vérifiant le nom de la contrainte violée, comme le fait le code ci-dessus.

Handlers plus longs et travail en cours

Le pattern ci-dessus place tout le handler dans une seule transaction. Cela convient quand le travail se résume à quelques écritures. C'est inconfortable quand le handler appelle quelque chose de lent, car la transaction reste ouverte, les verrous restent détenus, et le visibility timeout peut expirer pendant que vous attendez. Le message expiré est livré de nouveau à un second consommateur, qui démarre le même travail lent en parallèle.

Il existe deux réponses solides. La première est de garder la transaction courte : faire le travail externe lent *en dehors* d'elle, selon l'approche des effets externes de la section suivante, et ne mettre dans la transaction que le changement d'état local et la ligne d'inbox. La seconde, quand le travail doit réellement être réservé, est de rendre la réservation explicite à l'aide d'un statut.

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

Un consommateur insère la ligne avec le statut `processing` et un lease, et le commit de cette réservation est fait dans sa propre transaction courte avant que le travail lent ne commence. Un doublon qui trouve un lease actif ne doit pas faire l'ack du message, car la première tentative peut encore échouer et l'ack effacerait le seul retry. Il remet le message en file d'attente avec un délai, plus long que le lease restant, et regarde de nouveau à la livraison suivante. Celui qui trouve un lease expiré peut prendre la relève avec un lease token plus récent, que le détenteur initial doit vérifier avant de faire le commit de ses changements locaux ; celui qui trouve `done` fait l'ack.

Le token ne protège que ce qui peut le vérifier. La transaction locale le peut : elle verrouille la ligne d'inbox, compare le token, et annule si un token plus récent existe. L'appel externe lent ne peut pas être annulé ; l'effet reçoit donc la clé déterministe de la section suivante, et un détenteur périmé qui se réveille et répète l'appel est reconnu par le système en aval. C'est l'idée de fencing du lease du relay au Chapitre 5, avec la même limite : un fencing ne fonctionne que là où la ressource protégée vérifie le token.

Ce schéma fonctionne, et il ajoute une machine à états sur laquelle quelqu'un doit raisonner, y compris la question de ce qui se passe quand un worker meurt en détenant un lease. Je n'y aurais recours que lorsque la première approche n'est pas disponible, et je garderais le lease court et renouvelable.

> **Avertissement :** Ne comptez pas sur un `SELECT` préalable ou sur un verrou de ligne pour écarter les doublons. Qu'un second consommateur voie ou non le travail du premier dont le commit n'est pas fait dépend du niveau d'isolation de la base de données, et une vérification correcte sous un niveau échoue sous un autre. Passer au niveau serializable ne supprime pas non plus le besoin de la contrainte : cela remplace un doublon silencieux par un échec de sérialisation, que le handler doit traiter comme pouvant faire l'objet d'un retry et recommencer depuis le début. Les verrous réduisent la contention ; la contrainte d'unicité à l'intérieur de la transaction est la garantie.

Fenêtres de déduplication et rétention

Un inbox qui garde chaque identifiant pour toujours grandit pour toujours. À un moment, vous voudrez supprimer les anciennes lignes, et la suppression modifie la garantie d'une manière qui mérite un nom : la **fenêtre de déduplication**. Un message dont la ligne d'inbox a été supprimée peut être traité de nouveau s'il réapparaît un jour.

Quelle doit donc être la longueur de la fenêtre ? Elle doit couvrir le délai le plus long après lequel un doublon pourrait vraisemblablement arriver après l'original. Je ne connais pas ce chiffre pour votre système, et il dépend d'une courte liste d'éléments.

- **Rétention du broker et horizon de nouvelle livraison.** Une file d'attente qui livre de nouveau pendant quatre jours, ou un log qui conserve une semaine, signifie que des doublons peuvent arriver aussi tard.
- **Pratique des dead letters et du redrive.** Si une équipe peut faire un redrive de dead letters une semaine après leur échec, la fenêtre doit durer plus longtemps (Chapitre 8).
- **Replay.** Relire un log d'événements à partir d'un ancien offset crée des doublons d'événements aussi anciens que le replay. Soit la fenêtre est assez longue, soit le replay s'exécute contre un consommateur délibérément réinitialisé, ce qui est une procédure que vous concevez à l'avance.
- **Failover du relay.** Les doubles publications après un failover sont généralement proches dans le temps de l'original, si bien qu'elles se situent bien à l'intérieur de toute fenêtre que vous choisiriez.

Une forme courante est une période de rétention nettement plus longue que la propre rétention du broker, disons quelques semaines à titre d'illustration, avec un job planifié qui supprime par lots. Partitionner la table d'inbox par date rend la suppression bon marché, car supprimer une partition entière coûte bien moins cher que supprimer des millions de lignes.

Gardez à l'esprit que l'inbox n'est pas la seule ligne de défense. Une contrainte sur une clé métier dans la table du domaine vit aussi longtemps que les données, et protège donc contre les doublons qui arrivent après la disparition de la ligne d'inbox. C'est une raison de plus d'utiliser les deux couches : l'inbox vous donne une protection bon marché, à mémoire bornée, pour le cas courant, et la contrainte couvre la traîne.

Si le système peut l'accepter, il existe une alternative à une fenêtre temporelle : dériver la clé de quelque chose de monotone. La règle est plus étroite qu'il n'y paraît. Elle s'applique aux événements porteurs d'état, ceux que le Chapitre 9 distingue des deltas, où chaque événement contient l'état complet d'un agrégat. Pour ceux-là, le consommateur stocke la version d'agrégat la plus élevée appliquée et rejette tout ce qui n'est pas plus récent. L'offset est une position dans une partition et la version est une propriété d'un agrégat ; les deux ne sont donc pas interchangeables : un offset ne dit rien des agrégats qui vivent sur d'autres partitions.

Pour les événements delta, « rejeter si ce n'est pas plus récent » est faux, car une version sautée signifie un changement perdu. Un tel consommateur applique la version `last + 1`, met en tampon une version ultérieure jusqu'à ce que le trou soit comblé, et traite tout ce qui est inférieur ou égal à `last` comme un doublon ; le Chapitre 9 décrit cette mise en tampon et ses limites. Dans les deux cas, la mémoire est bornée sans TTL, et elle dépend de garanties d'ordre que le Chapitre 9 examine. Elle convient aux agrégats dotés d'un compteur de version et ne convient pas aux consommateurs alimentés par plusieurs partitions.

Effets de bord externes

L'inbox protège les effets qui vivent dans votre propre base de données. Beaucoup d'effets n'y vivent pas. Un e-mail quitte votre système au moment où vous l'envoyez, et un appel au fournisseur de paiement débite une carte, que le commit de votre transaction soit fait ensuite ou non. Aucune transaction locale ne peut les annuler, si bien que l'astuce de l'atomicité ne s'applique pas.

Prenez Payments qui traite un command `CapturePayment`. Il doit appeler le fournisseur externe, puis enregistrer le résultat.

```
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)
```

En cas de crash entre l'appel au fournisseur et la transaction, l'argent a bougé et Payments n'en a aucune trace. À la nouvelle livraison, le handler appelle de nouveau le fournisseur. Que l'acheteur soit débité deux fois dépend désormais entièrement du fournisseur.

Transmettre la clé en aval

La correction consiste à transmettre la clé d'idempotence au système qui produit l'effet, pour qu'il puisse reconnaître la répétition. Les fournisseurs de paiement le permettent couramment : vous envoyez une clé avec la re-

quête, et une seconde requête avec la même clé renvoie le premier résultat au lieu d'agir de nouveau. Les détails de l'en-tête ou du champ varient selon le fournisseur, et je lirais la documentation de votre fournisseur pour connaître son comportement exact et la durée pendant laquelle il se souvient des clés.

Le détail qui compte le plus est *la manière de choisir la clé*. Elle doit être **déterministe à partir du travail, et non générée à chaque tentative**. Une clé aléatoire créée à l'intérieur du handler ressemble à chaque retry à une requête nouvelle, et ne protège rien.

```
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
```

Retracez maintenant les défaillances. Un crash après l'appel au fournisseur et avant le commit laisse l'argent déplacé et Payments dans l'ignorance. Le command livré de nouveau exécute le même code avec la même clé, le fournisseur renvoie le résultat d'origine, et la transaction l'enregistre. Un crash pendant l'appel au fournisseur, où la réponse est perdue, est la même situation : le retry porte la même clé, et le fournisseur soit effectue la capture une seule fois, soit renvoie le résultat antérieur. Deux consom-

mateurs en concurrence sur le même command envoient la même clé au même moment, et un fournisseur qui rejette la seconde comme « en cours » n'a pas fait échouer le travail : le handler traite cette réponse comme pouvant faire l'objet d'un retry et regarde de nouveau à la livraison suivante.

La mémoire du fournisseur et l'inbox ne sont pas la même protection. Un fournisseur se souvient des clés pendant une durée limitée, et je lirais sa documentation pour savoir combien de temps. Une nouvelle livraison qui arrive après cette période provoquerait une seconde capture. La ligne d'inbox locale, conservée au moins aussi longtemps que la fenêtre de déduplication décrite plus haut, est ce qui empêche le handler d'appeler le fournisseur tout court ; la rétention des clés par le fournisseur n'est donc jamais la fenêtre sur laquelle vous comptez.

Si le travail est identifié par le métier et non par le message, utilisez ici aussi une clé dérivée du métier, comme l'identifiant du paiement plus l'opération. Ainsi, même deux commands avec des identifiants différents atteignent le fournisseur sous la forme d'une seule requête.

Quand le système en aval n'a pas de clé

Tous les systèmes en aval n'en acceptent pas. Le SMTP simple n'a pas de clé d'idempotence, et beaucoup de fournisseurs d'e-mail et de SMS n'en proposent pas de manière fiable. Vous faites alors face à un choix entre deux issues imparfaites, et je le ferais explicitement, effet par effet.

Envoyez avant d'enregistrer, et un crash peut provoquer un e-mail en double. Enregistrez avant d'envoyer, et un crash peut provoquer un e-mail manquant. Ce sont des comportements at-least-once et at-most-once pour l'effet. Pour une notification « votre colis est programmé », un doublon occasionnel coûte peu et un message perdu devient un ticket de support ; j'enverrais donc d'abord, en acceptant la répétition occasionnelle. Pour un code de bon d'achat à usage unique qui accorde une valeur, je pencherais dans l'autre sens et j'ajouterais un moyen de détecter et de réparer un envoi manquant.

Certains effets peuvent être rendus plus sûrs en enregistrant l'*intention* dans la transaction locale et en produisant l'effet depuis un worker distinct, ce qui revient à appliquer l'idée de l'outbox aux commands. Cela

n'élimine pas le problème à la frontière du fournisseur, car le worker peut encore planter entre l'appel et le marquage. Cela déplace en revanche le problème en un seul endroit où vous pouvez le voir, en faire un retry délibérément et le réconcilier. Le Chapitre 13 couvre la réconciliation : comparer vos enregistrements avec ceux du fournisseur et réparer les écarts, ce qui est la défense ultime précisément pour les effets qui ne peuvent pas être rendus idempotents.

Idempotency-Key et API HTTP

Le même problème apparaît à votre propre porte d'entrée. Le téléphone d'un acheteur envoie « passer la commande », le réseau perd la réponse, et l'application fait un retry. Orders ne peut pas distinguer un retry d'une seconde commande à moins que le client ne le dise.

La convention largement utilisée est un en-tête de requête `Idempotency-Key` : le **client** génère une valeur unique pour chaque opération logique et envoie la même valeur à chaque retry de celle-ci. C'est l'autre moitié de la définition donnée plus haut : pour la consommation d'événements, la clé est normalement l'`event_id`, attribué par le producteur, et pour les requêtes côté client, elle est fournie par l'appelant. Le rôle du serveur est de faire en sorte qu'une clé répétée se comporte comme la première requête.

Un serveur a besoin de plus que la clé. Il stocke la clé avec un fingerprint de la requête et la réponse qu'il a produite.

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

La vérification du fingerprint compte. Un client qui réutilise une clé pour un corps différent a un bug, et renvoyer sans bruit la première réponse le cacherait. L'état « en cours » compte pour le cas de la concurrence : deux requêtes simultanées portant une même clé ne doivent pas s'exécuter toutes les deux. Le verrou de ligne n'ordonne que la reprise d'une ligne existante ; la première réservation est arbitrée par la contrainte d'unicité, et c'est pourquoi l'insertion perdante est interceptée et reçoit une réponse 409.

Le lease compte pour le cas du crash, et son token pour le cas de la lenteur. Un détenteur suspendu au-delà de son lease peut se réveiller et tenter de terminer, mais `requireLeaseToken` voit le token plus récent, et toute sa transaction, commande et lignes d'outbox comprises, est annulée. Ici, le token sert de fencing parce que la ressource protégée, la base de données locale, le vérifie à l'intérieur de la transaction, la même idée que le lease du relay au Chapitre 5. La durée du lease ne fait que raccourcir le chevauchement ; c'est le token qui fait échouer le commit du détenteur périmé. L'appelant périmé reçoit une erreur, et son retry reçoit la réponse stockée. Les deux vérifications utilisent l'horloge de la base de données, `dbNow()`, et non celle du serveur d'application, pour les raisons que le Chapitre 3 donne au sujet des horloges. Limiter la portée de la clé à l'appelant empêche un client d'entrer en collision avec un autre.

Le pseudocode montre le chemin de réussite, et le chemin d'échec a lui aussi besoin d'une règle. Si la requête échoue pour une raison déterministe, comme une erreur de validation qui renvoie un 4xx, le serveur stocke cette réponse comme la réponse terminée. Un retry avec la même clé obtient alors le même 4xx et ne relance pas la validation sur des données qui ont pu changer. Si elle échoue pour une raison transitoire, comme un timeout ou une dépendance indisponible, la réservation est libérée ou laissée à expiration, pour que le retry du client puisse exécuter le travail. Mettre en cache une défaillance transitoire comme réponse définitive transformerait un accroc en erreur permanente pour cette clé.

La contrainte d'unicité sur `(caller, key)` joue le même rôle d'arbitrage que la clé primaire de l'inbox. Le pattern est le même : une réservation atomique, un résultat stocké et une réponse déterministe pour une répétition. Là où l'API produit des événements, la ligne de commande, les lignes d'outbox et la finalisation de l'enregistrement d'idempotence peuvent toutes être écrites dans une seule transaction locale, de sorte qu'une requête dont on fait un retry ne crée ni une seconde commande ni un second `OrderPlaced`. Conservez la clé stockée au moins aussi longtemps que les clients peuvent faire des retries, ce qui est une décision sur le comportement des clients et non sur les brokers.

Un lien mérite d'être énoncé : la clé fournie par le client peut se propager dans l'événement. Orders peut la placer dans les en-têtes de l'événement ou en dériver l'`event_id` de `OrderPlaced`, pour que les consommateurs

en aval voient une seule identité, du geste de l'acheteur jusqu'à l'écriture comptable. Si l'`event_id` est dérivé, il doit provenir de manière déterministe de l'appelant et de la clé ensemble, par exemple sous la forme d'un UUID basé sur un nom calculé sur les deux, pour que deux appelants qui choisiraient par hasard la même clé ne partagent jamais une identité. Le Chapitre 11 traite de ce qui a sa place dans un événement ; que la clé y ait sa place dépend de la distance sur laquelle vous voulez que l'idempotence suive la requête.

Compromis et coûts

L'inbox ajoute une écriture à chaque message. Chaque message consommé coûte une insertion supplémentaire, dans la même transaction, sur le chemin critique. Pour un consommateur qui fait une poignée d'écritures, le surcoût est faible. Pour un consommateur à très gros volume et très bon marché, comme celui qui met à jour un cache de modèle de lecture, ce peut être le coût dominant, et l'idempotence naturelle y est souvent le meilleur choix.

Le stockage et le nettoyage de l'inbox sont un travail continu. La table grandit avec le trafic, et la politique de rétention est une décision qui vous appartient et que vous revisitez. Une fenêtre trop courte réadmet d'anciens doublons ; une fenêtre trop longue coûte du stockage et de la taille d'index. Quelqu'un doit surveiller le job de nettoyage, car lorsqu'il s'arrête, la table grandit sans bruit jusqu'à devenir un problème de performance.

Une contrainte d'unicité sur le chemin critique peut devenir un point de contention. Avec une clé bien répartie, elle est bon marché. Avec une clé qui se concentre, ou avec de nombreux consommateurs qui sollicitent un même agrégat très actif, elle peut créer une file d'attente. Indexer la table par consommateur et identifiant d'événement, et garder courte la transaction du handler, rend cela gérable.

Les clés métier sont une décision de modélisation, et une mauvaise décision est difficile à voir. Trop fine, et les doublons logiques passent ; trop grossière, et du travail légitime est rejeté. La défaillance est silencieuse dans les deux sens. Je testerais la clé avec des cas délibérément gênants, comme deux opérations légitimes sur un même agrégat, avant de m'y fier.

L'idempotence ne couvre pas tout. Elle rend une répétition inoffensive ; elle ne corrige pas un handler qui se trompe dès la première fois, elle n'ordonne pas les messages, et elle n'annule pas un effet pour lequel le fournisseur n'a pas de clé. Un inbox ne protège pas non plus contre un doublon qui a *changé* : si un producteur republie « le même » fait avec un `event_id` différent et qu'aucune clé métier n'existe, rien ne le reconnaît.

Quand ne pas utiliser l'inbox. Si chaque effet du consommateur est naturellement idempotent et ne peut pas devenir périmé, comme l'écrasement d'un champ dénormalisé par le dernier état complet, l'inbox est un coût sans bénéfice, et je m'en passerais en consignant le raisonnement dans le code. Elle est aussi inadaptée à un consommateur qui ne peut pas écrire dans une base de données transactionnelle qui lui soit propre, comme un forwarder sans état. Ce consommateur a besoin que son système en aval fasse la déduplication, avec la clé transmise, ou doit être reconçu pour posséder un stockage.

Erreurs courantes

- **Vérifier et agir en étapes séparées.** C'est tentant parce que le code se lit naturellement : « l'ai-je déjà vu ? non, alors je le fais ». Cela échoue en cas de crash entre les deux et en cas de concurrence. Placez l'écriture d'inbox dans la même transaction que l'effet et laissez la contrainte d'unicité arbitrer.
- **Écrire la ligne d'inbox après l'ack, ou avant le commit de l'effet.** Faire l'ack d'abord perd le message si le travail échoue ensuite. Enregistrer avant le commit de l'effet marque comme fait un travail qui n'a jamais eu lieu. L'effet, la ligne d'inbox et le commit forment une unité ; l'ack suit le commit.
- **Garder l'inbox dans un autre stockage que l'effet.** Un cache ou une base de données séparée paraît moins cher, et recrée le dual write. Gardez-la dans la même base de données que l'état qu'elle protège.
- **Générer une nouvelle clé à chaque tentative.** Une valeur aléatoire créée dans le handler ou dans la boucle de retry du client fait paraître chaque retry nouveau. Dérivez la clé du command, de l'événement ou de l'opération métier, pour qu'elle soit identique à chaque tentative.

- **N'utiliser que l'`event_id` en supposant qu'il couvre tous les doublons.** Il n'intercepte pas deux messages différents pour la même intention. Ajoutez une contrainte sur une clé métier là où le domaine présente ce risque.
- **Choisir une fenêtre de déduplication au jugé.** Une fenêtre plus courte que la rétention du broker, ou que votre pratique de redrive, réadmet sans bruit d'anciens doublons. Dérivez la fenêtre de ces chiffres et consignez la dérivation.
- **Traiter une violation d'unicité comme une erreur générique.** Faire l'ack de toute violation cache de vrais bugs ; faire un retry sur toute violation boucle indéfiniment sur un message déjà traité. Interceptez la violation de l'inbox à part.
- **Oublier l'effet externe.** L'inbox ne peut pas annuler un e-mail ou un débit de carte. Transmettez une clé déterministe au fournisseur, ou choisissez délibérément entre un doublon possible et une perte possible.
- **Publier les événements de suivi directement depuis le handler.** Un crash entre le commit et la publication perd l'événement, et la nouvelle livraison prend le chemin rapide de l'inbox et ne le répète jamais. Écrivez l'événement de suivi dans l'outbox, dans la même transaction que la ligne d'inbox et le changement d'état.
- **Faire l'ack d'un message dont le lease appartient à quelqu'un d'autre.** Un doublon qui voit un lease actif et fait l'ack a supprimé le seul retry si la première tentative échoue ensuite. Remettez-le plutôt en file d'attente avec un délai.

Points clés

- Les doublons viennent du broker, du relay, des retries, du redrive et du replay, ainsi que des clients ; chaque consommateur doit donc être écrit comme si n'importe quel message pouvait arriver deux fois.
- Le check-then-act n'est pas sûr parce que la vérification et l'effet sont des étapes séparées, et qu'un crash ou une tentative concurrente peut se glisser entre les deux.
- La ligne d'inbox et le changement métier doivent faire l'objet d'un seul commit dans une transaction locale, dans la propre base de données

du consommateur, et c'est la contrainte d'unicité sur la clé qui arbitre réellement les doublons concurrents. Un consommateur qui annonce ce qu'il a fait ajoute une ligne d'outbox à cette même transaction.

- Utilisez l'`event_id` comme clé d'idempotence de base pour la consommation d'événements, et ajoutez une clé métier partout où deux messages différents peuvent exprimer la même intention.
- La fenêtre de déduplication doit durer plus longtemps que la rétention du broker, votre pratique de redrive et tout replay prévu, et une contrainte sur une clé métier dans la table du domaine vous protège après la suppression des lignes d'inbox.
- Pour les effets de bord externes, envoyez une clé d'idempotence déterministe dérivée du travail et non de la tentative, gardez la ligne d'inbox locale comme garde à long terme parce que le fournisseur oublie les clés, et là où le système en aval n'a pas de clé, choisissez sciemment entre un doublon possible et une perte possible.
- À la frontière de l'API, l'appelant fournit la clé d'idempotence, et le serveur stocke la clé, un fingerprint de la requête et la réponse, pour qu'une répétition renvoie le premier résultat.

L'idempotence rend inoffensif un message répété, mais elle suppose que le traitement finit par réussir. Certains messages n'y parviendront jamais : ils échouent à chaque tentative, et les retries et les nouvelles livraisons s'accumulent derrière eux. Le Chapitre 8 traite de ces poison messages et des dead letters qu'ils laissent derrière eux.

Chapitre 8 — Poison messages et dead letters

Imaginez Parcelly le lendemain du jour où un nouveau champ, `gift_message`, a été livré sur `OrderPlaced`. Notifications le lit pour construire l'e-mail de confirmation, et la commande de test d'un vendeur porte un message cadeau contenant un caractère que le modèle d'e-mail ne sait pas encoder. Le consommateur lève une exception sur ce seul message. Le message revient. Le consommateur lève une nouvelle exception.

À l'heure du déjeuner, le message a été livré plusieurs milliers de fois. Chaque tentative échoue au même endroit, de la même manière, en quelques millisecondes. Le consommateur est occupé, en bonne santé selon toutes les métriques d'infrastructure, et n'a fait aucun ack utile depuis des heures. Sur un broker à files d'attente, les autres messages sont toujours servis par les workers restants, si bien que les dégâts paraissent limités. Sur un topic à log, une partition a cessé d'avancer parce que le consommateur revient sans cesse sur l'enregistrement en échec au lieu de le dépasser, et chaque commande de cette partition attend derrière lui.

Quelqu'un finit par s'en apercevoir, et quelqu'un propose de déplacer le message ailleurs pour que le travail puisse continuer. C'est la moitié facile de la réponse. La moitié difficile, c'est ce qui se passe ensuite : qui examine ce message, comment décider de ce qu'il signifie, et comment le faire revenir dans le système sans provoquer un second incident. Ce chapitre traite des deux moitiés. Que doit faire un système d'un message qu'il ne peut pas traiter, et comment une équipe empêche-t-elle cette réponse de devenir une décharge ?

Le problème, et pourquoi il est difficile

Un **poison message** est un message qui échoue à chaque fois qu'il est traité. Le mot « chaque » porte la définition. Un message qui échoue parce qu'une base de données était brièvement injoignable n'est pas un poison

message ; il réussira dès que la base de données reviendra. Un poison message échoue parce que quelque chose dans le message, ou dans le code qui le lit, rend la réussite impossible tant qu'un humain ou un déploiement n'a rien changé. Dans le vocabulaire du Chapitre 3, la sémantique de la défaillance dit qu'un nouvel essai ne peut pas aider, et c'est précisément pour cela que faire des retries est un effort gaspillé.

Les causes habituelles se rangent en quelques groupes :

- **Un contenu malformé ou inattendu.** Un champ obligatoire manque, une valeur est hors limites ou un payload ne se parse pas. Les changements de schéma qui n'étaient pas compatibles avec un ancien consommateur produisent ce cas (le Chapitre 12 explique comment l'éviter).
- **Un bug dans le consommateur.** Le message est valide, mais le code gère mal un cas que personne n'a testé, comme un null là où on n'en attendait jamais.
- **Une hypothèse métier violée.** Un événement `PaymentCaptured` arrive chez un consommateur qui n'a aucune trace de l'autorisation correspondante, peut-être parce que des données ont été supprimées, ou parce que des événements se sont croisés en vol et que les retries ont déjà dépassé toute fenêtre de réordonnancement raisonnable. À l'intérieur de cette fenêtre, le même symptôme est transitoire, comme le note le Chapitre 6 et comme le Chapitre 9 le détaille, et il relève des retries plutôt que de la DLQ. Au-delà, le consommateur ne peut pas avancer et ne peut pas inventer l'enregistrement manquant.
- **Un rejet permanent en aval.** Un transporteur externe refuse une adresse comme invalide. Réessayer produit le même refus.

Le Chapitre 6 a introduit la distinction entre défaillances transitoires et permanentes, et ce chapitre reprend là où celui-ci s'arrête. Une politique de retry qui fonctionne correctement finira, à terme, par abandonner un message. Les retries sont bornés par le nombre de tentatives, par le temps ou par un budget. Ce qui se passe à ce moment-là est une décision de conception, et si personne ne la prend, le broker la prend à votre place.

La boucle de nouvelle livraison infinie

Si le consommateur ne fait rien de particulier, le comportement par défaut du broker prend le relais. Un broker à files d'attente livre de nouveau un message sans ack, soit immédiatement après un nack, soit après l'expiration du visibility timeout, et il tient généralement un compteur de livraisons qui augmente à chaque tentative.

Un broker à log fonctionne différemment, et la différence compte. Il n'y a pas de nouvelle livraison par message : le consommateur détient un offset et lit vers l'avant. Refuser de faire un commit d'offset ne pousse pas le broker à rendre l'enregistrement ; la position en mémoire du consommateur continue d'avancer, et l'enregistrement en échec est sauté silencieusement jusqu'à ce que le prochain redémarrage ou le prochain rééquilibrage ramène le consommateur au dernier offset dont le commit est fait. Une boucle sur un log n'existe que parce que le consommateur en construit une : il revient à l'offset en échec et le relit, ou il cesse de faire avancer la partition pendant qu'il fait des retries sur place. Un consommateur qui ne fait ni l'un ni l'autre, et qui fait quand même un commit au-delà de l'échec, perd l'enregistrement sans le moindre signal.

Dans les deux familles, le résultat peut être le même : le message revient, échoue et revient encore, ou, sur un log dont les offsets sont gérés sans soin, il disparaît. Les bons outils de la famille log sont un seek ou une pause explicites, un retry topic qui reçoit l'enregistrement en échec pour que la partition principale puisse avancer, et un dead-letter topic pour les enregistrements abandonnés.

La boucle a trois coûts. Le premier est la capacité gaspillée : les workers passent leur temps sur un travail qui ne peut pas réussir. Le deuxième est le bruit : les logs se remplissent de stack traces identiques, et les vrais problèmes s'y cachent. Le troisième est le coût qui compte le plus, le **head-of-line blocking**. Sur un log, un consommateur traite une partition dans l'ordre, si bien qu'un enregistrement qu'il refuse de sauter bloque tous les enregistrements derrière lui. Sur une file d'attente simple, un poison message ne bloque pas les autres à lui seul, mais une poignée d'entre eux peut occuper tous les workers. Si la file d'attente regroupe les messages pour garantir l'ordre (files FIFO, groupes de messages, sessions), un poison message bloque son groupe, ce qui donne le même rayon d'impact qu'une partition de log, en plus étroit.

Les deux mauvaises échappatoires

Les équipes sous pression se rabattent généralement sur l'une de deux échappatoires, et les deux sont mauvaises pour la même raison : elles cachent la défaillance au lieu de la traiter.

La première consiste à **jeter le message** : attraper l'exception, la journaliser, faire un ack et passer à la suite. Le pipeline s'écoule de nouveau, mais l'événement métier a disparu. Pour `OrderPlaced`, cela signifie une commande qui existe dans Orders et dans aucun autre service. Pour `PaymentCaptured`, cela signifie un grand livre qui ne correspond pas à la réalité. Jeter le message n'est le bon choix que lorsque le message n'a aucune valeur, et un consommateur ne peut pas le savoir au moment de l'échec.

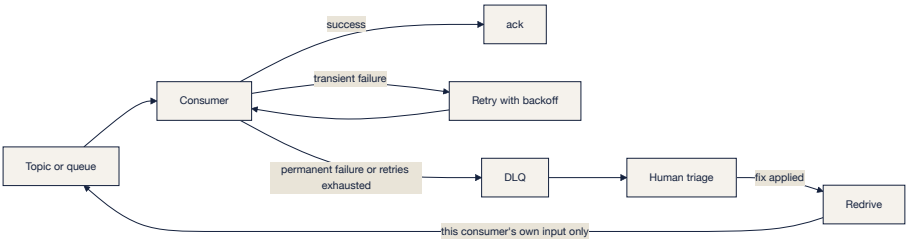
La seconde consiste à **faire des retries indéfiniment** en comptant sur quelqu'un pour corriger la cause. C'est la boucle infinie décrite plus haut, et elle transforme un seul mauvais message en panne pour tout ce qui se trouve derrière lui.

La troisième option se situe entre les deux. Mettre le message de côté, avec assez de contexte pour qu'une personne puisse le comprendre plus tard, et laisser le reste du stream continuer. Cette option, c'est la dead-letter queue.

Le pattern de la dead-letter queue

Une **dead-letter queue (DLQ)** est une destination distincte où un consommateur ou un broker place les messages qu'il a abandonnés. Sur un broker à log, la même idée prend généralement la forme d'un **dead-letter topic** : un topic ordinaire qui reçoit les enregistrements en échec. Lorsqu'un énoncé vaut pour les deux familles, ce chapitre dit DLQ.

Le pattern a un noyau simple. Lorsqu'un message a épuisé ses retries, ou échoue d'une manière que le consommateur classe comme permanente, le consommateur le déplace vers la DLQ et fait un ack de l'original. Le stream continue. Le message en échec n'est pas perdu ; il est mis en parking lot là où on peut l'examiner.



Regardez les deux dernières flèches du diagramme. La DLQ n'est pas une sortie. C'est une salle d'attente dont une porte ramène à l'entrée propre du consommateur en échec, et toute la difficulté de ce chapitre tient dans la distance entre ces deux portes. La porte ne mène pas à un topic partagé : les autres consommateurs ont déjà traité l'événement, et ils ne doivent pas le revoir.

Où se fait le déplacement

Il y a deux façons d'implémenter le déplacement, et le choix détermine ce que vous pouvez stocker.

L'envoi vers la DLQ géré par le broker. De nombreux brokers à files d'attente peuvent être configurés pour qu'un message livré plus d'un certain nombre de fois soit déplacé automatiquement vers une file désignée. Cela coûte peu à mettre en place et fonctionne même lorsque le consommateur plante avant de pouvoir faire quoi que ce soit. Vous obtenez un compteur de livraisons, mais généralement pas la raison de l'échec du consommateur, parce que le broker ne la connaît pas.

L'envoi vers la DLQ géré par le consommateur. Le consommateur attrape la défaillance, construit un enregistrement qui inclut la raison, le publie dans la DLQ, et seulement ensuite fait un ack de l'original. Cela vous permet de classer les défaillances et d'envoyer une erreur permanente vers la DLQ dès la première tentative au lieu de la cinquième. Cela coûte du code dans chaque consommateur, et ce code doit être écrit avec soin : publier d'abord dans la DLQ, faire l'ack ensuite, pour qu'un plantage entre les deux produise un doublon dans la DLQ, jamais une perte.

J'utiliserais les deux là où le broker offre le premier. La règle gérée par le broker n'existe que sur certains brokers à files d'attente, et tous les types de files ne limitent pas les livraisons. Là où elle existe, c'est le filet de sécurité pour les consommateurs qui plantent ou se bloquent d'une manière

que le code ne peut pas attraper, et, associée au chemin géré par le consommateur, elle garantit qu'un poison message ne peut pas boucler indéfiniment, même si votre propre gestion contient un bug.

Les brokers à log n'ont généralement ni compteur de livraisons par message ni DLQ côté broker, et cela laisse un trou. Un message qui fait planter le processus, comme un échec du désérialiseur avant l'exécution du handler, une erreur de mémoire insuffisante ou un plantage natif, n'atteint jamais votre bloc `try`. Le consommateur redémarre à l'offset dont le commit est fait, lit le même enregistrement et plante de nouveau. Le Chapitre 6 a noté que le nombre de tentatives repart de zéro après un plantage. Pour un poison record, c'est tout le problème.

Le remède consiste à compter avant de traiter, pour que le décompte survive au plantage :

```
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)
```

Le compteur a pour clé la partition et l'offset (ou `event_id` si vous pouvez le lire sans désérialiser). Le prix est une écriture durable par enregistrement sur le chemin critique, et une limite qu'il faut fixer assez haut pour qu'un redémarrage ordinaire lors d'un déploiement ne fasse pas passer des enregistrements sains au-dessus. L'alternative est d'accepter qu'un poison record qui fait planter le processus en boucle bloque sa partition jusqu'à ce qu'une personne s'en aperçoive, et c'est pourquoi l'alarme sur le retard décrite plus bas compte.

```
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)
```

L'étape `deadLetter` est celle où l'on capture le contexte nécessaire au triage, et l'ordre de ses deux dernières lignes est l'essentiel.

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

Si `publish` échoue lui-même, `deadLetter` lève une exception et le message ne reçoit pas d'ack. Sur un broker de la famille des files d'attente, cela suffit : le message revient par la nouvelle livraison normale. Sur un broker de la famille log, cela ne suffit pas, parce qu'un offset dont le commit n'est pas fait ne ramène pas l'enregistrement ; le consommateur doit revenir à la position de l'enregistrement en échec, ou mettre la partition en pause, et refaire la publication vers la DLQ avant de lire plus loin. Dans les deux cas, le but est le même mode de défaillance : le message reste dans le système jusqu'à ce qu'il ait un nouveau foyer sûr.

Ce qu'il faut stocker avec une dead letter

Une dead letter sans contexte est une énigme. Une équipe qui trouve un message dans une DLQ six semaines plus tard doit répondre à quatre questions : qu'était-ce, d'où cela venait-il, pourquoi cela a-t-il échoué, et est-il encore sûr de le traiter ? L'enregistrement doit y répondre sans qu'on ait à ouvrir un dépôt de code source.

Champ	Pourquoi il est là
Corps et en-têtes d'origine, octet pour octet	Le redrive doit reproduire le message exactement, `event_id` compris
Nom et position de la source (topic, partition et offset, ou nom de la file)	Situe le message dans l'historique, appuie les décisions liées à l'ordre
Clé d'origine (clé de l'enregistrement ou routing key) et horodatage d'origine	Le redrive doit publier avec la même clé, sinon le message atterrit sur une autre partition et perd son ordre par clé
Entrée du consommateur (la file propre au consommateur en échec ou son topic de redrive)	Le seul endroit où un redrive a le droit de publier
`redrive_count`	Distinct de `attempt_count` : combien de fois ce message est déjà revenu
`error_class` et `error_message`	Regroupe les défaillances par cause ; tronquez les messages longs
Stack trace ou pointeur vers la ligne de log	Permet à un ingénieur de voir où cela a échoué sans le reproduire
`attempt_count`	Distingue « échoué une fois, permanent » de « échoué après de nombreux retries »
`first_failed_at` et `failed_at`	Montre depuis combien de temps le message attend et quand le problème a commencé

Champ	Pourquoi il est là
Nom et version du consommateur	Relie la défaillance à un déploiement ; souvent le chemin le plus rapide vers la cause
`correlation_id`	Relie le message au reste de son flux métier

Deux mises en garde s'appliquent. D'abord, conservez le payload d'origine sans modification. Si vous l'enveloppez dans une enveloppe de votre cru, l'outil de redrive doit l'en extraire exactement, sinon `event_id` et `causation_id` dériveront et les garanties d'idempotence du Chapitre 7 seront perdues. Ensuite, rappelez-vous qu'une DLQ est une copie de données de production, souvent conservée longtemps. Elle hérite des contrôles d'accès et des obligations de rétention de l'original, et les payloads contenant des données personnelles demandent le même soin dans la DLQ qu'ailleurs.

Parking lot ou DLQ

Le mot DLQ sert à désigner deux choses différentes, et les séparer rend l'exploitation plus facile à raisonner.

Un **parking lot** est un lieu de rétention pour les dead letters qui ont besoin d'une décision humaine. La DLQ est l'endroit où le système place un message automatiquement ; le parking lot est l'endroit où une personne place un message délibérément, après l'avoir examiné et avoir conclu qu'un redrive ne le réparera pas. C'est un lieu distinct avec un objectif distinct.

Regardez ce que chacun contient. La DLQ contient des messages qui ont échoué et n'ont pas été examinés. Elle doit se vider : son état attendu est vide ou presque, et tout ce qui y est ancien est un symptôme. Le parking lot contient des messages que quelqu'un a examinés et mis de côté : une commande annulée avant l'arrivée de son événement, un payload qui demande une correction manuelle, un enregistrement qui doit être réconcilié à la main avec une autre équipe. Ces messages n'attendent pas qu'un correctif soit déployé. Ils attendent une décision.

Les mélanger produit une défaillance prévisible. Si les messages décidés et mis en parking lot restent dans la DLQ, la profondeur de la DLQ ne revient jamais à zéro, on relève le seuil d'alerte pour faire taire le bruit, et le pro-

chain vrai poison message arrive sans que personne le remarque. Les garder séparés permet à l'alarme de la DLQ de rester précise.

Un parking lot peut être aussi modeste qu'une table de base de données avec un statut, un responsable, une raison et une date de revue. Il n'a pas du tout besoin d'être une destination du broker, puisque rien ne va le consommer automatiquement. La date de revue est ce qui compte le plus : un message mis en parking lot sans date pour son prochain examen est déjà oublié.

Responsabilité et alarmes

Une DLQ sans responsable est un réservoir qui se remplit en silence. La première décision, avant tout outillage, consiste à savoir qui est responsable de chacune.

Mon choix par défaut est que l'équipe responsable du consommateur est responsable de la DLQ. Elle a écrit le code qui a échoué, elle sait ce que le message aurait dû signifier, et c'est elle qui peut corriger et déployer. Une DLQ par consommateur, ou par groupe de consommateurs et par source, rend cette responsabilité évidente. Une seule DLQ partagée pour toute l'entreprise la rend ambiguë, et l'ambiguïté signifie que personne ne regarde. Là où une équipe plateforme fournit l'outillage, l'équipe plateforme est responsable du mécanisme et l'équipe consommatrice est responsable du contenu.

Sur quoi déclencher une alarme

Déclenchez l'alarme sur l'**arrivée**, pas sur la seule profondeur. Un seul message qui apparaît dans une DLQ normalement vide mérite une notification, parce que la première dead letter est le moment le moins coûteux pour trouver un bug. Un seuil de profondeur seul ne vous dit rien tant que la file n'est pas déjà profonde.

Trois signaux combinés couvrent la plupart des cas :

- **Toute nouvelle arrivée** dans une DLQ censée être vide. Envoyez-la vers un canal qu'un humain lit pendant les heures de travail.
- **Le taux d'arrivée.** Une rafale soudaine signifie généralement un mauvais déploiement ou un changement de schéma incompatible, et justi-

fie de déclencher l'astreinte.

- **L'âge du message le plus ancien.** Ce signal attrape la défaillance lente : la DLQ qui reçoit un message par jour, sur lequel personne n'agit, jusqu'à ce que le plus ancien ait un mois et que le contexte métier ait disparu.

Je ne connais pas le bon seuil d'astreinte pour votre système, et personne ne peut vous le donner sans connaître la valeur métier des messages en question. Une DLQ rattachée aux consommateurs de `PaymentCaptured` mérite un seuil plus bas qu'une DLQ rattachée à Notifications, parce qu'un e-mail perdu est un désagrément et qu'une capture perdue est un écart dans le grand livre. Les seuils doivent refléter cette différence, et la différence doit être écrite à côté de l'alarme.

> **Remarque :** Déclenchez aussi une alarme sur le retard du consommateur. Un consommateur bloqué sur un poison record dans une partition de log ne produit pas forcément de dead letters, parce qu'il n'abandonne jamais. Un retard qui augmente sans aucun débit est le signal que la boucle décrite au début de ce chapitre est en train de tourner.

Redrive: récupérer les messages

Le **redrive** consiste à rendre les dead letters au consommateur qui a échoué sur elles, pour qu'il les traite de nouveau. La cible est l'entrée propre de ce consommateur : son retry topic ou son topic de redrive, ou sa propre file. Ce n'est jamais le topic partagé d'où provenait le message à l'origine, parce que tous les autres consommateurs de ce topic ont déjà traité l'événement, et le republier là le livrerait une seconde fois à chacun d'eux. La clé d'origine voyage avec le message, si bien qu'il reste sur la même partition et garde sa position par clé. Le compteur de tentatives repart de zéro, et le `redrive_count` distinct augmente de un, si bien qu'un message qui revient sans cesse est visible au lieu de paraître nouveau à chaque fois. Le redrive est l'opération qui donne sa valeur à une DLQ, et aussi celle qui a le plus de chances de provoquer un second incident. Le Chapitre 3 a décrit l'ingénieur qui appuie sur un bouton et fait un redrive de 40 000 messages d'un coup. Le remède consiste à traiter le redrive comme une procédure conçue, et non comme un bouton de confort.

Corriger la cause avant le redrive

Un message qui a échoué à cause d'un bug du consommateur échouera de nouveau tant que le bug n'est pas corrigé et déployé. Faire un redrive avant le correctif fait tourner les messages en rond et remplit de nouveau la DLQ avec les mêmes entrées, désormais avec des compteurs de tentatives plus élevés et un historique déroutant. L'ordre du travail est le suivant : comprendre la cause, déployer le correctif, faire un redrive d'un petit échantillon, le regarder réussir, puis faire le redrive du reste.

Le redrive doit pouvoir être répété sans risque

Un message qui fait l'objet d'un redrive est par définition un doublon : le système a déjà vu cet `event_id`, et le consommateur l'a peut-être vu aussi. Si le traitement d'origine a partiellement réussi avant d'échouer, certains effets existent peut-être déjà. Le Chapitre 7 est la raison pour laquelle on y survit. Un consommateur idempotent doté d'un inbox traite le message du redrive comme n'importe quelle autre nouvelle livraison et applique ses effets une seule fois.

Deux points méritent d'être énoncés. Le redrive doit conserver l'`event_id` d'origine, puisque c'est la clé d'idempotence. Un outil de redrive qui génère un nouvel identifiant pour chaque message met l'inbox en échec, et l'effet peut être appliqué deux fois. Et un consommateur qui n'est pas idempotent ne devrait pas du tout être la cible d'un redrive en masse ; corrigez cela d'abord, ou acceptez que chaque message d'un redrive soit une évaluation manuelle du risque.

Le redrive par lots contrôlés

Un outil de redrive sûr a quelques propriétés, aucune exotique :

1. **Un filtre de sélection.** Faire un redrive par classe d'erreur, par fenêtre temporelle ou par `correlation_id`, et pas seulement « tout ».
2. **Un rate limit.** Les messages reviennent vers l'entrée du consommateur à un rythme que celui-ci et ses dépendances peuvent absorber. Un afflux de messages d'un redrive est un retry storm qu'on s'inflige soi-même, et les limites de messages en vol et le délestage décrits dans « Les retries en surcharge : backpressure » (Chapitre 6) s'appliquent au trafic de redrive exactement comme au trafic réel.

3. **Un interrupteur de pause.** Un opérateur peut arrêter le redrive en cours de route si des défaillances réapparaissent.
4. **Un dry-run.** L'outil indique ce qu'il déplacerait avant de déplacer quoi que ce soit.
5. **Un enregistrement d'audit.** Qui a fait un redrive de quoi, quand et pourquoi, conservé quelque part de durable.

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

Remarquez l'ordre à l'intérieur de la boucle : un enregistrement est marqué comme ayant fait l'objet d'un redrive après la réussite de la publication, pas avant. Si l'outil plante en cours de route, quelques messages ont fait l'objet d'un redrive et restent non marqués dans la DLQ, ce qui mène à des doublons, pas à une perte. Un consommateur doté d'un inbox absorbe ces doublons ; un consommateur qui n'en a pas transforme un plantage d'outillage en problème de données. L'enregistrement est marqué, pas supprimé, si bien que la piste d'audit et `redrive_count` survivent ; le nettoyage suit la règle de rétention de « Éviter la décharge » plus bas.

Redrive et ordre des messages

Le redrive casse l'ordre, sauf si vous le concevez pour le préserver. Un message envoyé vers la DLQ a été sauté ; des messages ultérieurs pour le même agrégat ont été traités entre-temps. Quand le message du redrive arrive, il arrive après eux. C'est l'exemple Inventory du Chapitre 6 avec un écart plus long : si `OrderPlaced` est envoyé vers la DLQ au lieu de faire l'objet d'un retry, `OrderCancelled` ne trouve toujours aucune réservation à libérer, et la commande du redrive, des heures plus tard, en crée toujours une que rien ne libérera. Conserver la clé ne répare pas cela : la clé place le message sur la bonne partition, mais la partition est passée à autre chose.

Que cela compte ou non dépend du consommateur. Un consommateur **commutatif**, c'est-à-dire dont l'état final est le même quel que soit l'ordre d'arrivée des événements, n'est pas affecté. Un consommateur qui dépend de transitions d'état l'est. Le Chapitre 9 explique comment détecter et gérer les arrivées dans le désordre avec des numéros de version ou des vérifications de séquence, et un consommateur fondé sur l'état devrait les utiliser avant que quiconque fasse un redrive vers lui. S'il ne le fait pas, la procédure de redrive doit inclure une vérification que le monde n'est pas passé à autre chose depuis que le message a été envoyé vers la DLQ.

Ce qu'une dead letter fait à un stream ordonné

Les streams ordonnés imposent la décision la plus difficile de ce chapitre, parce que le choix qui protège l'ordre est celui qui arrête l'activité.

Prenez un topic à log dont la clé est `order_id`, où Ledger enregistre des écritures. Les consommateurs traitent chaque partition en séquence. Le message 41 pour `order-81724` échoue de façon permanente. Le consommateur a deux options :

- **S'arrêter et attendre.** Bloquer la partition au message 41 jusqu'à ce que quelqu'un corrige la cause. L'ordre est préservé exactement. Toutes les autres commandes dont le hachage tombe sur cette partition s'arrêtent aussi, même si leurs messages sont corrects. Le rayon d'impact est la partition entière.
- **Envoyer vers la DLQ et continuer.** Mettre le message 41 à l'écart, traiter le 42 et les suivants. La partition s'écoule. Mais le 42 dépendait

peut-être du 41, et le consommateur l'a alors traité sur un état qui n'inclut pas le 41.

Aucun des deux choix n'est gratuit. S'arrêter échange la disponibilité contre la justesse ; continuer échange la justesse contre la disponibilité. Aucun réglage ne vous donne les deux, et mieux vaut choisir délibérément, consommateur par consommateur, que d'hériter de ce que le framework fait par défaut.

Ma règle empirique comporte trois parties :

1. **Si le consommateur est insensible à l'ordre ou commutatif**, envoyez vers la DLQ et continuez. Notifications est le cas typique : un e-mail en échec sur une commande n'a aucune incidence sur la suivante.
2. **Si les messages ultérieurs pour la même clé dépendent de celui qui a échoué**, ne continuez pas pour cette clé. Envoyez le message en échec vers la DLQ, puis détournez aussi les messages suivants de la même `partition_key` vers une zone de rétention jusqu'à ce que le premier soit résolu. Les autres clés continuent. Le rayon d'impact reste ainsi limité à un agrégat au lieu d'une partition, au prix d'un mécanisme supplémentaire : le suivi, clé par clé, des clés « bloquées ».
3. **Si le consommateur est un grand livre financier, arrêtez-vous.** Le coût d'un traitement dans le désordre est un enregistrement faux d'une manière difficile à détecter et coûteuse à réparer, alors qu'une partition en pause est bruyante et évidente. La justesse avant la vitesse était la règle énoncée pour Ledger, et elle s'applique ici sous sa forme la plus nette.

La variante par clé de la deuxième règle mérite une esquisse, puisque c'est la plus utile et la moins courante. Elle suppose une petite table `key_block` avec une ligne par clé bloquée (`partition_key`, `blocked_event_id`, `state`), et qu'un seul consommateur à la fois gère le trafic réel d'une clé donnée, ce que vous apporte une partition ou un groupe de messages.

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

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

La libération est une procédure distincte, et un sweep périodique la rend redémarrable.

```
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)
```

Trois détails de cette esquisse décident si elle fonctionne. Le premier est le redrive lui-même. Le message du redrive est précisément celui sur lequel la clé est bloquée, il doit donc contourner le blocage : la vérification compare son `event_id` à celui qui est bloqué. Sans cette exception, le redrive est mis en parking lot derrière les messages qu'il était censé précéder, et la clé ne se débloque jamais.

Le deuxième est la course avec le trafic réel. Pendant que `resolveBlockedKey` vide les messages retenus, de nouveaux messages pour la même clé continuent d'arriver. Ils doivent toujours être retenus, et non traités, jusqu'à ce que l'ensemble des messages retenus soit vide. Cela ne tient que si la décision de mettre en parking lot et la décision de débloquer sont sérialisées : les deux prennent le verrou de ligne de la clé, et la mise en parking lot est écrite dans la même transaction que la vérification.

Si la mise en parking lot était une étape séparée, un message réel pourrait voir une clé bloquée, puis trouver l'ensemble retenu déjà vidé et débloqué, et être mis en parking lot derrière un blocage qui n'existe plus. Personne ne le viderait, et les messages ultérieurs seraient traités avant lui.

Le troisième est un plantage au milieu de la libération. Une fois le message du redrive traité, l'état `resolving` est écrit avant l'ack, si bien que le travail à faire est enregistré dans la base de données et pas seulement dans un message qui ne reviendra jamais. Le sweep reprend toute clé restée dans cet état et continue ; un message traité peu avant le plantage est absorbé par l'inbox. Si un message retenu échoue de façon permanente pendant la libération, la clé se bloque de nouveau, ancrée sur ce message, avec sa propre dead letter, et le reste de l'ensemble retenu attend son redrive. Le blocage par clé est une boucle, pas un événement ponctuel.

L'esquisse n'est pas complète. Un sweep peut chevaucher une libération en cours, si bien que deux résolveurs peuvent travailler sur la même clé. La vérification de l'état sous le verrou empêche chacun d'eux de passer outre un nouveau blocage : dès qu'un résolveur ancre le blocage sur un message en échec, l'autre constate que la clé n'est plus `resolving` et s'arrête. Ce qui reste, c'est un travail en double sur le même message retenu, que l'inbox absorbe ; un lease sur l'état `resolving` économiserait ce travail, pas la justesse. Elle ne pose pas non plus de borne à l'ensemble retenu, et elle laisse de côté la manière dont un opérateur écarte un message retenu qui n'a plus de sens. Considérez-la comme la forme du mécanisme, pas comme un mécanisme fini.

C'est une vraie machinerie avec ses propres modes de défaillance réels, comme un stockage des messages retenus qui grossit sans limite, si bien que je ne la construirais que pour les streams où l'ordre par clé compte et où le taux d'envoi vers la DLQ est assez bas pour que l'ensemble retenu reste petit. C'est aussi un mécanisme qui ne sait pas si les messages retenus ont encore un sens une fois celui en échec résolu. Lorsque les transitions d'état elles-mêmes doivent être validées, et pas seulement mises en séquence, le tampon de versions du Chapitre 9 est l'outil le plus solide : il retient un événement jusqu'à ce que son prédécesseur ait effectivement été appliqué, par numéro de version, et il fonctionne sans enregistrement de dead letter pour l'ancrer.

Éviter la décharge

Une DLQ se transforme en décharge lorsque les messages y entrent plus vite que quiconque ne décide de ce qu'ils signifient. Le symptôme est familier : des milliers d'entrées, de nombreuses erreurs différentes, personne ne sait lesquelles comptent, et une règle non écrite selon laquelle on n'y touche pas. À ce stade, la DLQ n'est plus un mécanisme de sécurité ; c'est un endroit où l'on stocke des données perdues sous un nom poli.

Les décharges ont des causes, et chacune a sa parade.

- **Pas de responsable.** La parade est la règle de responsabilité ci-dessus, mise par écrit.
- **Pas de cadence de triage.** Les dead letters doivent être examinées selon un calendrier, et pas seulement quand une alarme se déclenche. Pour une DLQ à faible volume, une revue hebdomadaire est un point de départ raisonnable.
- **Une rétention qui survit à l'utilité, ou qui expire avant le triage.** Une dead letter concernant une réservation de stock qui a expiré le mois dernier ne peut pas faire l'objet d'un redrive qui ait un sens. Fixez une durée de rétention fondée sur le temps pendant lequel le message reste valable pour le métier, et décidez à l'avance de ce qui se passe à l'expiration : archiver, supprimer avec une entrée d'audit, ou faire remonter. L'erreur inverse est tout aussi courante : une rétention plus courte que la cadence de triage supprime des messages avant que quiconque les ait regardés. Si la DLQ est une file du broker dont la rétention est plus courte que votre intervalle de revue, le broker efface les preuves en premier. Fixez la rétention de la DLQ au-dessus du plus long écart entre deux revues, et lisez la documentation de votre plateforme sur le point de départ de l'horloge de rétention. Si elle part du moment de la mise en file d'origine et non du moment de l'envoi vers la DLQ, un message qui a longtemps attendu avant d'échouer arrive déjà près de l'expiration. C'est une raison de plus pour que l'enregistrement stocke l'horodatage d'origine.
- **Tout est envoyé vers la DLQ.** Si des issues métier courantes, comme un rejet de validation qui fait normalement partie du flux, sont envoyées vers la DLQ, les vraies défaillances s'y noient. Une commande rejetée est un événement métier avec sa propre gestion, pas une dead letter.

- **Des défaillances transitoires envoyées trop tôt vers la DLQ.** Si la politique de retry abandonne au bout de quelques secondes lors d'une brève panne, chaque message en vol pendant cette panne atterrit dans la DLQ. Le remède se trouve au Chapitre 6 : une politique de retry dont les tentatives et la deadline sont dimensionnées pour la panne à laquelle vous comptez survivre.

Je donnerais aussi une issue à chaque dead letter. Le triage se termine par l'une des issues d'une courte liste : un redrive après un correctif, une correction suivie d'un redrive, une mise en parking lot en attente d'une décision manuelle, ou une suppression avec une raison enregistrée. « Toujours là » n'est pas une issue.

Compromis et coûts

Une DLQ déplace une défaillance d'un endroit où elle est bruyante, le consommateur bloqué, vers un endroit où elle est silencieuse. C'est à la fois le bénéfice et le danger. Le stream s'écoule, et le message en échec peut être oublié. Tout ce qui, dans ce chapitre, n'est pas la DLQ elle-même, les alarmes, le responsable, la cadence de revue et la rétention, existe pour contrer ce silence.

Les coûts précis à peser :

- **La charge opérationnelle.** Quelqu'un doit trier, décider et faire le redrive. Une DLQ dont personne n'a le temps de s'occuper est pire que pas de DLQ du tout, parce qu'elle cache le problème derrière l'apparence d'un mécanisme.
- **La gouvernance des données.** Les dead letters contiennent des payloads de production, parfois avec des données personnelles, en dehors du cycle de vie habituel. Elles ont besoin de contrôles d'accès, de règles de rétention et de chemins de suppression.
- **Le risque pour l'ordre.** Envoyer vers la DLQ un message d'un stream ordonné revient à le sauter. À moins de construire un blocage par clé ou d'arrêter la partition, les événements ultérieurs sont appliqués sans lui. Le blocage par clé ajoute lui-même un stockage des messages retenus, un contournement pour le redrive, une boucle de libération qui doit s'exécuter sans danger en parallèle du trafic réel, et un sweep qui reprend une libération interrompue.

- **L'identité stockée.** Un redrive sûr a besoin de la clé d'origine, de la partition et de l'entrée du consommateur sur chaque enregistrement, et d'un `redrive_count` à côté du nombre de tentatives. C'est davantage à stocker et davantage à garder cohérent, et sans cela le redrive ne peut être qu'une supposition.
- **L'investissement en outillage.** Un outil de redrive sûr, avec filtrage, rate limits, issue par enregistrement et piste d'audit, est un petit projet. L'acheter ou le construire fait partie du coût d'une DLQ.

Quand ne pas en utiliser : si un message peut être régénéré ou si sa perte est sans conséquence, une DLQ ajoute plus de processus que de valeur. Un événement de préchauffage de cache ou une mise à jour purement cosmétique peut être jeté, avec une métrique qui compte les messages jetés. Et là où la bonne réponse à une défaillance est de s'arrêter, comme dans l'exemple du grand livre, une DLQ qui laisse le stream continuer joue contre vous. Dans ce cas, arrêtez le consommateur, alertez bruyamment et corrigez la cause.

Le pattern a aussi une limite. Une DLQ contient des messages qui ont échoué de façon visible. Elle ne fait rien pour les messages qui ont été traités de façon incorrecte sans lever d'erreur, ce qui est le territoire du Chapitre 13 et de son travail de réconciliation.

> **Avertissement :** Ne faites jamais l'ack du message d'origine avant que la dead letter soit stockée de façon durable. Si la publication vers la DLQ échoue après l'ack, le message n'existe plus nulle part, et aucune alarme ne vous le dira.

Erreurs courantes

- **Envoyer vers la DLQ sans contexte.** C'est tentant parce que déplacer le corps brut est le code le plus court. Le correctif est l'enregistrement de métadonnées décrit plus haut, y compris les en-têtes d'origine exacts, pour que le redrive et le diagnostic soient tous deux possibles.
- **Faire des retries sur des défaillances permanentes.** Une erreur de parsing ne s'améliore pas avec le temps, et pourtant une politique de retry uniforme la traite comme un timeout. Classez les défaillances, et

envoyez les permanentes directement vers la DLQ au lieu de dépenser le retry budget.

- **Faire un redrive avant de corriger la cause.** La pression pour « faire repartir les messages » est réelle, et le résultat, ce sont les mêmes messages de retour dans la DLQ. Déployez le correctif, faites un redrive d'un échantillon, puis du reste.
- **Faire un redrive de tout d'un coup.** Un afflux de vieux messages dispute les ressources au trafic réel et peut déclencher un nouvel incident. Appliquez un rate limit, procédez par lots et gardez un interrupteur de pause.
- **Déclencher l'alarme sur la seule profondeur.** Cela semble être la métrique évidente, mais elle se déclenche tard. Déclenchez l'alarme sur la première arrivée, sur le taux et sur l'âge du message le plus ancien.
- **Une seule DLQ partagée pour tout.** Elle est peu coûteuse à créer et impossible à prendre en responsabilité. Donnez à chaque consommateur la sienne, et nommez son responsable.
- **Laisser se mélanger messages mis en parking lot et messages non examinés.** La DLQ ne se vide jamais, l'alerte est atténuée, et un vrai poison message passe inaperçu. Gardez un parking lot distinct, avec des responsables et des dates de revue.
- **Ignorer l'ordre lors du redrive.** Un message du redrive arrive après des événements ultérieurs pour le même agrégat. Utilisez les vérifications de version du Chapitre 9, ou confirmez que l'état n'est pas passé à autre chose.
- **Faire un redrive vers le topic partagé.** Cela ressemble à « remettre le message là d'où il vient », et cela livre l'événement une seconde fois à tous les autres consommateurs de ce topic. Faites le redrive vers l'entrée propre du consommateur en échec, avec la clé d'origine.
- **Attendre indéfiniment sur un enregistrement qui fait planter en boucle.** Sur un broker à log, un consommateur qui n'abandonne jamais ne produit ni dead letter ni alarme, hormis le retard. Comptez les tentatives de façon durable, par partition et par offset, pour qu'un redémarrage ne remette pas le décompte à zéro.

Points clés

- Un poison message échoue à chaque fois qu'il est traité, si bien que faire des retries indéfiniment transforme un seul mauvais message en panne pour tout ce qui attend derrière lui.
- Jeter un message en échec et faire des retries indéfiniment sont deux façons de cacher une défaillance ; une dead-letter queue met le message de côté avec son contexte pour que le reste du stream puisse continuer.
- Une dead letter doit porter le corps et les en-têtes d'origine inchangés, sa position dans la source, l'erreur, le nombre de tentatives et la version du consommateur, parce que la personne qui la lira des semaines plus tard n'aura que cet enregistrement.
- Publiez dans la DLQ avant de faire l'ack de l'original, pour qu'un plantage produise un doublon récupérable et jamais un message perdu.
- Chaque DLQ a besoin d'une équipe responsable nommée, d'une alarme sur la première arrivée, le taux et l'âge, et d'une cadence de triage régulière, sinon elle devient une décharge.
- Un redrive n'est sûr que si la cause est corrigée, si l'`event_id` et la clé d'origine sont conservés, si le message va vers l'entrée propre du consommateur en échec et jamais vers un topic partagé, si le consommateur est idempotent, et si les messages reviennent par lots soumis à un rate limit avec un interrupteur de pause.
- Sur un stream ordonné, envoyer un message vers la DLQ impose de choisir entre arrêter la partition et continuer au-delà d'un trou ; choisissez délibérément pour chaque consommateur, arrêtez-vous pour les registres comptables, et ne bloquez par clé que là où la machinerie vaut son coût.

L'envoi vers la DLQ et le redrive s'appuient tous deux sur une question que ce chapitre n'a cessé de repousser : que signifie, pour des messages, arriver dans le bon ordre, et que peut-on construire quand ce n'est pas le cas ? Le Chapitre 9 s'attaque directement à cette question.

Chapitre 9 — L'ordre des messages

Imaginez un jeudi soir chez Parcelly. Un acheteur passe une commande, puis change d'avis en quelques secondes et l'annule. Orders émet `OrderPlaced`, puis `OrderConfirmed`, puis `OrderCancelled`, tous pour `order-81724`, et tous dans le bon ordre. Le relay les publie lui aussi dans cet ordre.

Le topic qui alimente Shipping a été créé avec un réglage par défaut : les messages sont répartis sur six partitions sans tenir compte de leur contenu. `OrderConfirmed` atterrit sur la partition 2 et `OrderCancelled` sur la partition 5. Shipping fait tourner un consommateur par partition. Le consommateur de la partition 5 est inactif et rapide, si bien qu'il lit l'annulation en premier et trouve une expédition qui n'existe pas encore. Il hausse les épaules et fait un ack. Un instant plus tard, le consommateur de la partition 2, qui était occupé par un lot lent, lit la confirmation et planifie une expédition pour une commande qui n'existe plus.

Rien n'a planté, aucun message n'a été perdu, et aucun message n'a été dupliqué. Chaque composant a fait ce pour quoi il était configuré. Un transporteur viendra lundi récupérer un colis que personne n'a commandé. De combien d'ordre un tel système a-t-il besoin, où se perd-il, et que pouvez-vous construire quand vous ne pouvez pas l'avoir ?

Le problème, et pourquoi il est difficile

L'ordre n'est pas une propriété unique qu'un système possède ou non. C'est une chaîne, et chaque maillon peut la rompre indépendamment. Le relay peut publier les lignes dans une séquence différente de celle dans laquelle la base de données a fait les commits (le Chapitre 4 et le Chapitre 5 traitent de ce maillon). Le broker peut répartir des messages liés sur plu-

sieurs partitions ou files. Les consommateurs peuvent traiter en parallèle. Un retry peut placer un message derrière ses successeurs (Chapitre 6), et une dead letter peut le sauter entièrement (Chapitre 8).

Maillon	Comment l'ordre se perd	Où regarder
Producteur et relay	Relays parallèles, commits concurrents, identifiants qui ne suivent pas l'ordre des commits	Chapitres 4 et 5
Broker	Messages liés sur des partitions ou des files différentes	Ce chapitre
Consommateur	Plusieurs workers traitant les messages d'une même partition ou file	Ce chapitre
Gestion des défaillances	Retry topics, nacks, dead letters, redrive	Chapitres 6 et 8
Changement de topologie	Changement du nombre de partitions, rééquilibrages des consommateurs	Ce chapitre

La réponse évidente consiste à exiger un ordre global : chaque événement du système, dans une seule séquence, vu par chaque consommateur. Cela ressemble au choix par défaut sûr. C'est coûteux et, pour la plupart des systèmes, inutile.

Pourquoi l'ordre global coûte cher

Un ordre total unique signifie un point de séquencement unique. Sur un broker à log, c'est une seule partition, lue par au plus un consommateur d'un groupe de consommateurs à la fois. Votre plafond de débit devient la vitesse d'un seul écrivain et d'un seul lecteur. Ajouter de la capacité ne sert à rien, parce que la capacité vient de l'exécution en parallèle et que le parallélisme est précisément ce qu'un ordre total interdit.

Un système qui s'étend sur plusieurs services aggrave la situation. Orders, Payments et Inventory ont chacun leur propre base de données et leur propre outbox. Un ordre global entre tous ces services exigerait un coordinateur qui estampille chaque événement, et ce coordinateur se trouverait sur le chemin d'écriture de chaque service. Vous rachèteriez le couplage que l'architecture était conçue pour supprimer.

Pourquoi l'ordre global est le plus souvent inutile

Demandez-vous ce dont le métier a réellement besoin. Parcelly ne se soucie pas de savoir si `PaymentCaptured` pour `order-81724` est arrivé avant `OrderPlaced` pour `order-90311`. Ces commandes n'ont aucune influence l'une sur l'autre. Ce qui compte, c'est que pour une commande donnée, `OrderPlaced` soit vu avant `OrderConfirmed`, et `OrderConfirmed` avant `OrderCancelled`.

C'est l'**ordre par agrégat**. Un agrégat est la frontière de cohérence dont l'identifiant sert de clé de partition. Si chaque événement concernant un agrégat est vu dans l'ordre où il a été produit, et que les événements concernant des agrégats différents sont libres de s'entrelacer, la plupart des machines à états fonctionnent correctement. C'est l'ordre que vous pouvez vous offrir, et à mon avis c'est l'ordre autour duquel vous devriez concevoir.

Il existe une frontière que même l'ordre par agrégat ne franchit pas. Les événements concernant la même commande, produits par des services différents et transportés sur des topics différents, n'ont aucun ordre les uns par rapport aux autres. `PaymentAuthorized` et `StockReserved` peuvent arriver chez Orders dans n'importe quelle séquence, et c'est exactement pour cela que le flux de commande utilisé tout au long de ce livre ne demande la capture qu'après l'arrivée des deux. Ne considérez pas une clé de partition partagée comme une promesse entre topics.

L'ordre par agrégat

Le mécanisme est simple à énoncer et exigeant à maintenir : choisir une clé, envoyer tout ce qui porte la même clé sur un seul chemin, et traiter ce chemin un message à la fois.

La clé de partition

Sur un broker à log, le producteur hache la `partition_key` pour choisir une partition. Chaque message portant la même clé atterrit sur la même partition, et une partition est une séquence ordonnée. L'enveloppe de ce livre porte `partition_key` pour cette raison, et l'outbox la stocke pour que le relay puisse la fixer sans interroger le service producteur (Chapitre 4).

La garantie est soumise à une condition : **même clé, même partitioner, même nombre de partitions**. Le partitioner, c'est la fonction de hachage plus la manière dont la clé est sérialisée en octets avant le hachage. Si deux producteurs en utilisent de différents, un même `order_id` part vers deux partitions et l'ordre par clé disparaît, sans aucune erreur nulle part. C'est la défaillance polyglotte la plus concrète en matière d'ordre. Les bibliothèques clientes de langages différents ne partagent pas toutes la même fonction de hachage par défaut, si bien qu'un service écrit dans un langage et un relay écrit dans un autre peuvent envoyer le même `order_id` vers des partitions différentes. Deux défenses fonctionnent. Fixer explicitement le partitioner dans chaque producteur, avec le même algorithme et le même encodage de clé. Ou retirer la décision aux producteurs : faire calculer la partition par le relay lui-même à partir de la `partition_key` stockée, pour qu'un seul morceau de code soit responsable de la correspondance. La seconde est naturelle lorsque chaque événement passe déjà par le relay de l'outbox du Chapitre 5.

Les brokers à files d'attente sont différents. Des consommateurs concurrents sur une même file ne vous donnent aucune garantie d'ordre dès qu'ils sont plus d'un, parce que le broker remet le message suivant au premier consommateur libre. Certains brokers à files d'attente proposent une fonction de regroupement qui sérialise les messages partageant un identifiant de groupe, par exemple les groupes de messages des files FIFO d'Amazon SQS, et d'autres proposent un mode où un seul consommateur est actif sur une file à la fois. Ces fonctions marchent, et chacune coûte du parallélisme d'une manière différente. Vérifiez ce que votre broker promet au sujet de la nouvelle livraison et des visibility timeouts avec cette fonction, parce que la garantie comporte souvent des petites lignes. Dites sur quelle famille vous vous appuyez, et ce que cette famille promet réellement.

Choisir la clé

La clé est l'identifiant de l'agrégat, et ce choix dit où réside la cohérence. Pour le cycle de vie des commandes de Parcelly, la clé est `order_id`. Chaque événement qui fait passer une commande d'un état à l'autre la porte, ce qui donne à Shipping, Orders et Ledger une séquence cohérente par commande.

Quelques questions aident lorsque la réponse n'est pas évidente :

- L'état de quelle entité le consommateur modifie-t-il ? Prenez cette entité comme clé.
- Quelle est la plus petite unité à l'intérieur de laquelle deux événements peuvent entrer en conflit ? C'est l'agrégat.
- Le même stream est-il consommé par plusieurs services aux agrégats différents ? Alors soit le producteur choisit le plus fin qu'ils ont en commun, soit chaque consommateur a besoin de sa propre vue.

Inventory montre la tension. Les niveaux de stock appartiennent à un SKU, si bien que les commandes propres à Inventory prennent naturellement `sku` comme clé. Les événements qui vont d'Orders vers Inventory prennent `order_id` comme clé. Un événement de réservation porte les deux identifiants, et Inventory change de clé en interne s'il a besoin de l'ordre par SKU. Choisir une seule clé pour chaque topic du système est tentant, et cela gâche l'intérêt d'avoir des agrégats.

Les partitions chaudes

Une clé de partition ne répartit la charge uniformément que si les clés sont uniformément actives. Elles ne le sont pas. Imaginez chez Parcelly une vente flash au cours de laquelle l'unique produit d'un vendeur reçoit en une heure la plupart des commandes d'une journée. Si Inventory prend `sku` comme clé pour son propre topic, cette seule clé produit l'essentiel du trafic, tout ce trafic atterrit sur une seule partition, et un seul consommateur doit faire le travail de tout le groupe pendant que les autres restent inactifs.

Ajouter des partitions n'aide pas. Une clé donnée correspond toujours à une seule partition, et ajouter des partitions déplace les autres clés autour d'elle. Il existe quatre réponses honnêtes :

- **Réduire l'agrégat.** Si la règle métier le permet, prenez comme clé quelque chose de plus fin, comme un identifiant de réservation plutôt que le SKU. On renonce à l'ordre à l'intérieur du SKU en échange de la répartition, et ce n'est acceptable que si aucune règle n'exige l'ordre par SKU.
- **Investir dans le consommateur, pas dans les partitions.** Rendez chaque message peu coûteux à traiter, pour qu'un seul consommateur

puisse suivre le rythme de la clé chaude.

- **Isoler.** Routez les clés chaudes connues vers un topic ou une partition dédiés, pour qu'elles n'affament pas le trafic ordinaire qui partage la partition.
- **Accepter un ordre relâché pour ce stream.** Ajoutez un suffixe aléatoire à la clé pour répartir la charge, et comptez sur les vérifications de version (plus loin dans ce chapitre) pour rétablir l'ordre chez le consommateur. C'est l'option la plus souple et la plus laborieuse. Notez ce que ce salage apporte et n'apporte pas : il répartit la charge sur le broker, mais la vérification de version sérialise de nouveau la clé sur le verrou de ligne de l'agrégat dans la base de données du consommateur. Le débit ne s'améliore que si le travail coûteux a lieu avant l'étape sérialisée, ou si les opérations commutent.

Je mesurerais l'asymétrie des clés avant de choisir. Une distribution de clés qui paraît uniforme pendant une semaine calme peut être très déséquilibrée un jour de forte activité, et le premier signe est généralement le retard du consommateur sur une seule partition précise (Chapitre 13).

Parallélisme et ordre tirent dans des directions opposées

Un groupe de consommateurs monte en charge en confiant chaque partition à un consommateur, si bien que le nombre de partitions est le plafond du parallélisme. Si Shipping a six partitions, six consommateurs peuvent travailler en parallèle. Un septième resterait inactif.

Les équipes essaient souvent d'augmenter le débit à l'intérieur d'un consommateur en confiant les messages à un pool de threads. Cela paraît inoffensif et casse l'ordre. Deux messages pour la même commande, lus l'un après l'autre, sont confiés à deux workers, et le worker le plus lent termine en second. La partition les a livrés dans l'ordre, et le traitement ne l'a pas préservé.

La façon d'obtenir les deux consiste à paralléliser entre les clés et à sérialiser à l'intérieur d'une clé. Dans le consommateur, hachez la clé vers une lane fixe, et donnez à chaque lane exactement un worker.

```
function pollLoop():
    while running:
        // bounded lanes: a slow lane pauses intake instead of filling
memory
        if anyLaneAbove(HIGH_WATER):
            pause(assignedPartitions)
        else if allLanesBelow(LOW_WATER):
            resume(assignedPartitions)
        for message in poll(maxRecords: POLL_LIMIT):
            // the same key always maps to the same lane, so per-key
order survives
            lanes[hash(message.partition_key) mod
LANE_COUNT].enqueue(message)
        // commits are per partition and happen here, on the polling
thread
        for partition in assignedPartitions:
            commit(partition, contiguousDoneOffset(partition) + 1)

function runLane(lane):
    while running:
        message = lane.dequeue()
        try:
            handleMessage(message)
        on error as failure:
            // in place or aside, as decided in Chapter 6
            decideInPlaceOrAside(message)
        markDone(message.partition, message.offset)
```

Trois détails portent la conception. Le premier est l'offset. Les commits se font par partition, et `contiguousDoneOffset` est l'offset le plus élevé en dessous duquel chaque message de cette partition est terminé. Lorsque les lanes terminent à des vitesses différentes, faire un commit de l'offset de la lane la plus rapide sauterait, après un plantage, des messages qu'une lane plus lente n'avait pas terminés. Le commit a lieu dans la boucle de polling parce que de nombreux clients de log ne peuvent pas être appelés sans risque depuis plusieurs threads. Le prix du parallélisme à l'intérieur du consommateur, c'est cette comptabilité, plus les doublons après un plantage, que gère un inbox du Chapitre 7.

Le deuxième est la borne. Sans elle, une lane lente maintient le point de commit immobile pendant que la boucle de polling continue de remplir la mémoire, et l'écart entre l'offset dont le commit est fait et le travail en vol est exactement le nombre de messages livrés de nouveau après un plantage. Mettre les partitions en pause lorsqu'une lane dépasse son seuil haut transforme la surcharge en retard, qui est visible et sûr, au lieu d'une crois-

sance de la mémoire, qui n'est ni l'un ni l'autre. Le mécanisme, et le choix entre la pause et le délestage, est celui décrit dans « Les retries en surcharge : backpressure » au Chapitre 6.

Le troisième est la défaillance. Si `handleMessage` échoue et que la lane fait un retry sur place, chaque clé qui partage la lane attend derrière, et l'offset dont le commit est fait cesse d'avancer. Mettre le message de côté permet à la lane d'avancer et change l'ordre pour cette clé. C'est la même décision que le Chapitre 6 a prise pour des partitions entières, prise désormais par lane, et la réponse dépend du consommateur de la même manière.

Sur un broker à files d'attente, la même idée apparaît un niveau plus haut. Si vous avez besoin de l'ordre par clé entre des consommateurs concurrents, soit vous utilisez la fonction de regroupement du broker, soit vous acceptez que la file livre dans le désordre et vous gérez cela dans le consommateur. Ne supposez pas qu'une file avec plusieurs consommateurs préserve l'ordre parce que les messages y sont entrés dans l'ordre. Ce n'est pas le cas.

Détecter les arrivées dans le désordre grâce aux versions

Parfois, le broker ne peut pas vous donner l'ordre dont vous avez besoin, à cause d'un retry topic, d'un redrive, de consommateurs parallèles ou d'un changement du nombre de partitions. Dans ces cas, le consommateur doit s'en apercevoir lui-même. Pour cela, les événements doivent porter quelque chose qu'un consommateur peut comparer.

Les horodatages ne sont pas la bonne chose à comparer. Le Chapitre 3 a expliqué que `occurred_at` reflète l'horloge d'une seule machine, et que s'en servir pour décider quelle mise à jour l'emporte permet à un service dont l'horloge avance de gagner tous les conflits qu'il devrait perdre. Utilisez `occurred_at` pour les humains et l'analytique.

Ce qui fonctionne, c'est un **numéro de version** (aussi appelé numéro de séquence) : un compteur que l'agrégat incrémente à chaque changement, écrit dans la même transaction que le changement d'état et transporté avec l'événement. Comme seul le service responsable modifie l'agrégat, et qu'il le fait dans une transaction, les versions forment une séquence sans

trou par agrégat : 1, 2, 3. L'absence de trou exige aussi une règle dans l'autre sens : **chaque incrément de version produit exactement un événement publié**. Une mise à jour interne qui augmente la version sans rien publier laisse un trou permanent, et chaque consommateur attendra un événement qui n'existe pas.

L'enveloppe standard n'a pas de champ pour cela, et je ne modifierais pas l'enveloppe pour un seul chapitre. Je le transporterais dans le payload, sous le nom `order_version` pour les événements d'Orders. C'est la version de l'agrégat (Chapitre 4), le même numéro que celui que stocke la ligne d'outbox. Je laisse la discussion approfondie sur la conception de l'enveloppe au Chapitre 11. Le compteur appartient à un producteur et à un agrégat. Un consommateur ne compare les valeurs de `order_version` qu'entre événements d'Orders pour la même commande, et jamais avec un événement de Payments ou d'Inventory, qui aurait besoin de son propre compteur. Une dépendance entre services ne se résout pas avec une séquence partagée. Elle se résout avec une vérification d'état ou de précondition, comme le fait l'étape de capture lorsqu'elle attend à la fois `PaymentAuthorized` et `StockReserved`. La ligne d'outbox l'obtient gratuitement, puisque le service écrit la nouvelle version, le changement d'état et la ligne d'outbox dans une seule transaction.

La décision du consommateur

Un consommateur qui conserve une vue d'un agrégat conserve aussi la dernière version qu'il a appliquée. Pour chaque événement qui arrive, il compare les versions et emprunte l'un de trois chemins.

Version reçue, comparée à la dernière appliquée	Signification	Action
Égale ou inférieure	Doublon ou périmé	Ignorer, le compter, faire un ack
Supérieure d'exactement un	L'événement suivant	Appliquer, enregistrer la nouvelle version
Supérieure de plus d'un	Un trou : quelque chose manque ou est en retard	Mettre l'événement en tampon, ne pas l'appliquer

```
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)
```

Deux propriétés rendent cela sûr. La vérification de version et la mise à jour de l'état font leur commit de façon atomique, si bien que deux workers traitant la même clé ne peuvent pas appliquer tous les deux la version 4. Et la vérification est idempotente : un doublon est une version périmée comme une autre et il est ignoré, si bien qu'elle complète l'inbox du Chapitre 7 au lieu de la remplacer. `holdEvent` doit lui aussi être idempotent. Sinon, une livraison en double d'une version déjà retenue heurterait la contrainte d'unicité du tampon, ferait échouer la transaction, laisserait le message sans ack et se transformerait en boucle de poison message.

Deux préconditions sont souvent négligées. La première est le début de la séquence. Un consommateur qui commence en cours de stream, comme un nouveau service ou un replay à partir d'un offset ultérieur, voit d'abord la version 5 et attendrait indéfiniment les versions 1 à 4. Initialisez sa vue à partir d'un snapshot ou de l'API du service responsable, avec la version que reflète ce snapshot, et commencez à comparer à partir de là.

La seconde est que la détection des trous ne fonctionne que si le consommateur voit chaque version. Un consommateur qui ne s'abonne qu'à un sous-ensemble des événements d'un agrégat, par exemple seulement `OrderCancelled`, verra les versions 3 et 7 et en conclura, à tort, que les versions 4 à 6 manquent. Pour de tels consommateurs, la règle s'affaiblit en « ne jamais appliquer quelque chose qui n'est pas plus récent que ce que j'ai », ce qui protège toujours contre l'écrasement par des données périmées, mais ne peut pas détecter les trous.

Les pièges du last-writer-wins

La politique de conflit la plus simple veut que la dernière écriture l'emporte. Elle est simple et elle cache trois pièges différents, selon ce que « dernière » signifie.

La dernière arrivée. La vue est écrasée par le message que le consommateur a traité le plus récemment. Un `OrderPlaced` livré de nouveau qui arrive après `OrderConfirmed` remet le statut à `PLACED`, ce qui est la défaillance que le Chapitre 7 a signalée. L'ordre d'arrivée est justement ce que la topologie du broker peut ne pas promettre.

La dernière selon l'horodatage. Mieux en apparence, pire en pratique. Le décalage d'horloge entre services signifie que l'événement gagnant peut être celui dont l'horloge avance. Deux événements écrits à quelques millisecondes d'intervalle sur des machines différentes ont un ordre d'horodatage dénué de sens.

La dernière selon la version. Cela fonctionne, et cela a sa propre limite. Écarter un événement périmé n'est correct que si l'événement plus récent contient déjà tout ce que disait le périmé. C'est vrai pour les événements qui portent un état (« la commande est maintenant `CANCELLED` »). C'est faux pour les événements qui portent des changements, comme « ajouter 4 990 au solde ». Ignorer un événement de changement périmé fait perdre le changement.

Une précision évite que cela contredise le tableau ci-dessus. Avec des versions sans trou appliquées strictement en séquence, une version inférieure est toujours un doublon, parce qu'elle a déjà été appliquée ; l'ignorer est

correct même pour des deltas. Le piège des deltas apparaît lorsqu'un consommateur saute à la version la plus récente et passe par-dessus un trou, puisque les changements sautés sont alors perdus.

Cette distinction décide de ce que vous faites face à une version périmée ou arrivée dans le désordre :

- **Les événements porteurs d'état** (l'état complet, ou un état qui écrase le précédent) : ignorer l'événement périmé est correct, parce que le suivant le remplace.
- **Les événements de delta ou de transition** (incréments, écritures comptables, tout ce qui ne remplace pas) : ne les écartez jamais. Mettez-les en tampon et réordonnez-les, ou traitez-les dans n'importe quel ordre si l'opération est commutative. Ledger en particulier ne doit pas écarter une écriture parce qu'elle est plus ancienne.

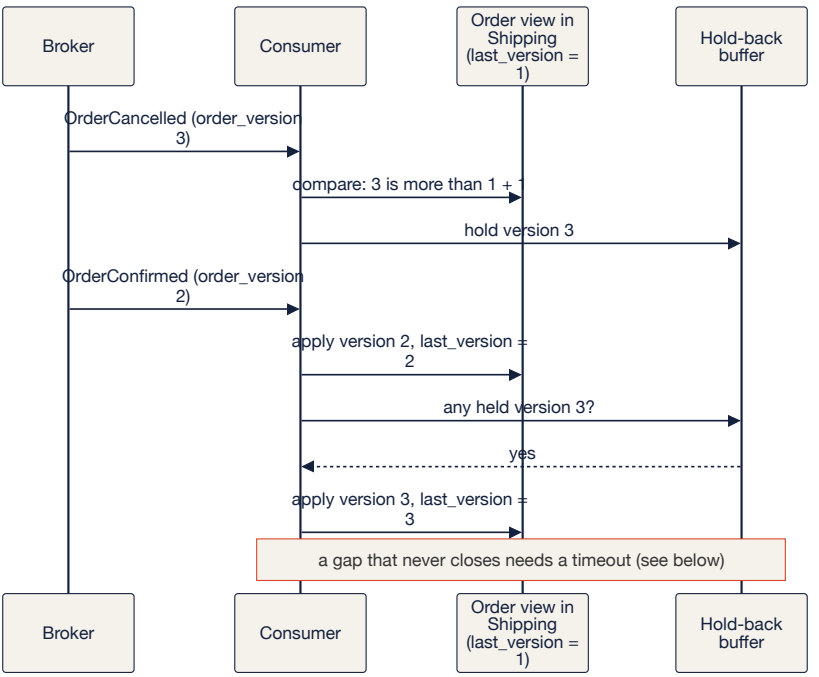
Un piège voisin est celui de l'agrégat supprimé. Si un consommateur supprime sa ligne lorsqu'il voit `OrderCancelled`, un `OrderPlaced` tardif ne trouve aucune ligne, passe la vérification de version sans rien à quoi se comparer, et crée une nouvelle commande. La résurrection est le bug caractéristique de l'ordre ignoré autour des suppressions. Conservez un tombstone : la clé, la dernière version et un marqueur de suppression, au moins aussi longtemps que des événements périmés peuvent encore arriver. Le Chapitre 8 a montré la même résurrection lorsqu'un `OrderPlaced` qui fait l'objet d'un redrive arrive après une annulation, et un tombstone associé à une vérification de version est ce qui l'empêche.

Enfin, ignorer est une décision qui mérite une trace. Un événement périmé écarté en silence est un bug que personne ne peut voir. Comptez les événements ignorés par consommateur et conservez la clé et les versions dans une ligne de log, pour qu'un pic du compteur apparaisse avant qu'un client ne le remarque (Chapitre 13).

Mise en tampon et réordonnancement

Lorsque le consommateur voit un trou, il a deux façons d'avancer. Il peut appliquer l'événement quand même et risquer un état faux, ou il peut retenir l'événement jusqu'à ce que celui qui manque arrive. La seconde option est un **hold-back buffer**.

Le diagramme de séquence ci-dessous montre Shipping recevant deux événements d'Orders pour `order-81724`, la version 3 avant la version 2, comme dans le scénario d'ouverture. Les deux proviennent d'un même producteur et d'un même agrégat, et c'est ce qui donne un sens à la comparaison de leurs versions. Shipping n'agit que sur `OrderConfirmed`, mais pour que cela fonctionne, il lit tout le stream d'Orders pour la commande et enregistre la version de chaque événement, y compris des types qu'il ignore par ailleurs. C'est ce qui rend réel le `last_version = 1` de la vue, et ce qui rend possible la détection des trous. Regardez où la version 3 attend, et ce qui la libère.



L'étape de libération est une boucle. Après avoir appliqué un événement, quel qu'il soit, le consommateur vérifie si le tampon contient la version suivante, l'applique, et recommence jusqu'à ce que le tampon n'ait plus de successeur.

```
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)
```

Le tampon lui-même est une table dont la clé est l'agrégat et la version, si bien qu'il survit à un redémarrage. Un tampon en mémoire disparaît avec le processus, et le broker a déjà considéré ces messages comme ayant reçu un ack.

Le nettoyage demande une règle de plus. Chaque fois que `last_version` fait un saut, par exemple après le remplacement de la vue par un état récupéré, les lignes retenues dont `version <= last_version` sont obsolètes, et le consommateur les supprime dans la même transaction. Sinon, elles restent dans la table pour toujours, et la boucle de vidange ne les atteint jamais.

Le trou qui ne se referme jamais

La mise en tampon est optimiste : elle suppose que l'événement manquant est simplement en retard. Parfois, il est perdu, envoyé vers la DLQ, ou n'a jamais été envoyé à ce consommateur. Sans borne, l'agrégat reste bloqué pour toujours, et l'ensemble des messages retenus grossit.

Chaque tampon a besoin d'un timeout et d'une décision sur ce qui se passe à son expiration. Il y a trois réponses raisonnables, et je choisirais selon le coût de l'erreur :

- **Récupérer l'état manquant.** Demander au service responsable l'état actuel au moyen d'une API et remplacer la vue. Cela fonctionne lorsque le producteur expose une telle API, et cela fait du tampon un chemin rapide doté d'un filet de sécurité.
- **Alerter et attendre.** Un humain examine. C'est le bon choix pour Ledger, où une écriture fausse coûte plus cher qu'une écriture lente.

- **Marquer la vue comme suspecte et laisser la réconciliation la réparer.** C'est le bon choix pour les modèles de lecture qui tolèrent des données périmées (Chapitre 13).

Le blocage par clé du Chapitre 8 est la forme sans versions de ce mécanisme. Là-bas, un marqueur de dead letter bloque la clé, les événements ultérieurs attendent dans un stockage de messages retenus, et un contournement spécial laisse passer l'événement du redrive pour les libérer. Ici, un événement envoyé vers la DLQ crée un trou, les événements ultérieurs attendent dans le tampon, et l'événement du redrive les libère de lui-même, parce qu'il arrive comme `last_version + 1` et que la boucle de vidange fait le reste. Les versions remplacent à la fois le marqueur et le contournement, et c'est pourquoi cette conception est la plus solide là où le producteur peut les fournir.

Les coûts sont réels : une table tampon à dimensionner et à nettoyer, une politique de timeout à ajuster, et une nouvelle catégorie d'agrégats bloqués pour laquelle il faut déclencher l'astreinte de quelqu'un. Si vos événements portent l'état complet et que vos consommateurs peuvent appliquer sans risque « le plus récent l'emporte », le tampon est inutile et je m'en passerais.

Retries, dead letters et ordre des messages

Le Chapitre 6 et le Chapitre 8 se sont chacun terminés par une question d'ordre qu'ils ont laissée pour ici. En bref, chaque mécanisme de gestion des défaillances déplace un message dans le temps, et c'est un réordonnancement.

- **Le retry sur place** sur une partition de log bloque les successeurs : l'ordre est préservé et toute la partition le paie en latence.
- **Un retry topic ou une file à délai** laisse passer les successeurs : le message en échec revient après des événements qu'il précédait.
- **Un visibility timeout** sur un broker à files d'attente renvoie un message dans la file après le timeout, où il entre en concurrence avec des messages envoyés après lui.
- **Une dead letter** saute le message, et un **redrive** le ramène après tout ce qui a suivi.

La vérification de version est ce qui permet de survivre aux options deux à quatre. Prenez l'exemple donné au Chapitre 6 : Inventory reçoit d'Orders `OrderPlaced` puis `OrderCancelled` pour une commande, et `OrderPlaced` est déplacé vers un retry topic après une défaillance transitoire. Avec la version de l'agrégat venant d'Orders, `OrderPlaced` est la version 1 et `OrderCancelled` la version 2. L'annulation arrive en premier, et Inventory n'a encore aucune vue pour la commande, si bien que `last_version` vaut 0 par défaut et que 2 est supérieur à 0 + 1. L'annulation est retenue, et non appliquée à une réservation qui n'existe pas. Lorsque l'`OrderPlaced` dont on a fait un retry arrive comme version 1, il est appliqué, et la boucle de vidange libère l'annulation. La réservation orpheline décrite au Chapitre 6 n'apparaît pas.

Cela ne fonctionne que pour les consommateurs qui conservent des versions et un tampon. Un consommateur qui ne le fait pas doit choisir, comme le Chapitre 6 l'a soutenu, entre bloquer la partition et tolérer le réordonnancement. Et le conseil du Chapitre 8 pour les registres comptables tient toujours : pour un consommateur chez qui une écriture tardive ou manquante signifie un enregistrement financier faux, arrêtez-vous et attendez. Un consommateur en pause est bruyant, et un Ledger qui applique discrètement des écritures dans le désordre ne l'est pas.

> **Avertissement :** Un retry topic combiné à un consommateur sans vérification de version réordonne les événements en silence. Rien n'échoue, aucune alerte ne se déclenche, et l'état est faux. Si vous mettez les défaillances de côté, faites de la vérification de version une partie du même changement.

Changer le nombre de partitions

La partition d'une clé est une fonction de la clé et du nombre de partitions, généralement un hachage modulo ce nombre. Changez le nombre, et la fonction change.

Supposons, à titre d'illustration, qu'une clé ait pour hachage 17. Avec quatre partitions, elle correspond à la partition 1, et avec six partitions, à la partition 5. Un hachage de 20 passe de la partition 0 à la partition 2, tandis que 14 et 15 restent où ils étaient. La plupart des clés se déplacent.

Avec un hachage modulo, une clé ne garde sa partition en passant de quatre à six que si son hachage modulo 12 vaut 0, 1, 2 ou 3, si bien qu'environ deux tiers des clés atterrissent ailleurs. La défaillance n'est pas rare : elle touche chaque clé qui a des messages en vol pendant le changement.

Le danger, c'est le chevauchement. Les messages d'une clé qui se déplace, écrits avant le changement, se trouvent sur l'ancienne partition, toujours non consommés. Les messages écrits après le changement vont sur la nouvelle partition. Deux consommateurs traitent alors la même clé en même temps, sans aucun ordre entre eux. Si un consommateur conserve aussi un état local par clé, comme un cache ou une fenêtre d'agrégation, cet état se trouve sur la mauvaise instance.

Les façons d'y faire face, à peu près par ordre de préférence :

1. **Prévoir assez de partitions dès le départ.** Surdimensionnez par rapport au parallélisme de pointe attendu, parce que ce nombre coûte peu à maintenir et cher à changer. De nombreux brokers à log permettent d'ajouter des partitions mais pas d'en retirer, si bien qu'une erreur dans un sens est définitive.
2. **Vider avant le changement.** Arrêtez les producteurs ou mettez le stream en pause, laissez les consommateurs atteindre la fin des anciennes partitions, puis changez le nombre. L'ordre est préservé au prix d'une fenêtre d'indisponibilité.
3. **Migrer vers un nouveau topic.** Créez un topic avec le bon nombre, effectuez une bascule au cours de laquelle les consommateurs terminent l'ancien topic avant de commencer le nouveau, et retirez l'ancien. C'est l'option la plus sûre et la plus laborieuse.
4. **S'appuyer sur les vérifications de version.** Si chaque consommateur détecte et met en tampon les événements arrivés dans le désordre, on survit à la transition sans pause, et la transition elle-même est aussi un bon test de l'efficacité de la vérification.

Rééquilibrages et révocation

Les rééquilibrages de groupes de consommateurs ont une forme semblable à plus petite échelle, et ils se produisent à chaque déploiement. Lorsqu'une partition passe à un autre consommateur, le travail en vol sur l'ancien peut

chevaucher les premiers messages sur le nouveau. Ce que fait le consommateur lors de la révocation décide de l'ampleur de ce chevauchement.

Lors de la révocation, le consommateur cesse de distribuer de nouveaux messages provenant des partitions révoquées, laisse les lanes terminer ce qu'elles contiennent ou l'abandonne, et fait un commit de l'offset contigu avant de céder la partition. Qu'un rééquilibrage révoque toutes les partitions ou seulement celles qui se déplacent dépend de la stratégie d'affectation de votre client, et une stratégie tout-en-une-fois bloque tout le groupe pendant sa durée ; vérifiez laquelle vous utilisez. Un consommateur lent à remarquer qu'il a perdu une partition continue de traiter comme un zombie pendant que le nouveau propriétaire fait le même travail, alors attendez-vous à un certain chevauchement quoi que vous fassiez.

L'inbox et la vérification de version ne rendent ce chevauchement inoffensif que pour les effets situés dans la propre transaction de base de données du consommateur. Elles ne font rien pour un e-mail ou un appel à l'API d'un transporteur effectués par les deux consommateurs. Ces effets de bord externes ont besoin de leur propre clé d'idempotence, comme le décrit le Chapitre 7, parce que l'inbox protège l'état local et rien d'autre.

Ce dont chaque consommateur de Parcelly a besoin

Tous les consommateurs n'ont pas besoin du même ordre, et les traiter de la même façon est une source à la fois de bugs et de gaspillage. Un tableau, dressé pour Parcelly, montre comment je déciderais.

Consommateur	Besoin d'ordre	Mécanisme	Si un message manque
Notifications	Aucun ; chaque message est un fait indépendant	N'importe quel ordre, pas de clé nécessaire	Envoyer vers la DLQ et continuer
Vue de statut d'Orders	Par commande, porteur d'état	Clé `order_id`, ignorer les versions périmées	Récupérer l'état actuel
Réservations d'Inventory	Par commande, transitions	Clé `order_id`, versions plus hold-back buffer	Retenir la clé, alerter
Shipping	Par commande, `OrderConfirmed` avant `OrderCancelled`	Clé `order_id`, versions plus hold-back buffer	Retenir la clé, alerter

Consommateur	Besoin d'ordre	Mécanisme	Si un message manque
Ledger	Par commande, pas de trous, aucune écriture abandonnée	Clé `order_id`, arrêt sur un trou dans la séquence par commande propre à Payments	S'arrêter et attendre un humain

La ligne Ledger nomme son compteur à dessein. Ledger consomme des événements de Payments, il compare donc une séquence que Payments incrémente par commande, et non `order_version`, qui appartient à Orders. Un compteur n'est comparable qu'entre événements du seul service qui en est propriétaire.

La première ligne est la moins coûteuse et mérite plus de crédit qu'on ne lui en accorde. La meilleure façon de gérer l'ordre est de concevoir le consommateur pour qu'il n'ait pas d'importance. Une opération commutative, comme ajouter un élément à un ensemble ou prendre le maximum de deux valeurs, donne le même résultat dans n'importe quel ordre. Un consommateur construit à partir de telles opérations n'a besoin ni de clé, ni de tampon, ni de version. Lorsque vous pouvez remodeler un handler de cette façon, je le ferais avant de recourir à la machinerie.

Compromis et coûts

Vous échangez du parallélisme contre de l'ordre, et le taux de change est l'agrégat. Une clé plus grossière donne un ordre plus fort et moins de répartition. Une clé plus fine donne de la répartition et un ordre plus faible. Aucune clé ne donne les deux, et la décision appartient à celui qui connaît les règles métier.

Les clés chaudes sont un coût de conception, pas un accident d'exploitation. Une clé de partition choisie parmi les identifiants métier hérite de l'asymétrie du métier. Une vente flash la révèle. Vous pouvez l'atténuer, et vous ne pouvez pas la supprimer sans renoncer à l'ordre pour cette clé.

Les versions exigent de la discipline de la part des producteurs. L'agrégat doit incrémenter la version dans la même transaction que le changement d'état, pour chaque changement, depuis chaque chemin de code. Un seul chemin qui l'oublie laisse un trou que les consommateurs atten-

dront indéfiniment. L'inverse est vrai aussi : chaque incrément doit publier exactement un événement, sinon la séquence a un trou qu'aucun consommateur ne peut refermer. Les backfills, les outils d'administration et les migrations de données sont les endroits où cela échoue, et un consommateur qui commence en cours de stream a besoin d'une vue initialisée avant de pouvoir comparer quoi que ce soit.

Les tampons ajoutent de l'état et une nouvelle défaillance. Une table de retenue doit être dimensionnée, nettoyée, et avoir un timeout, un responsable et une alerte. C'est aussi un endroit de plus où un agrégat bloqué peut se cacher. Pour les consommateurs qui peuvent appliquer « le plus récent l'emporte », passez-vous-en.

Les vérifications de version n'ordonnent pas entre agrégats ni entre topics. Elles protègent la séquence d'un seul agrégat. Si une règle doit savoir qu'un événement du topic A est venu avant un événement du topic B, un numéro de version ne le fournit pas. C'est un problème de workflow, et le Chapitre 10 le traite comme tel.

Quand ne pas s'en soucier. Si le consommateur est commutatif, si les événements portent l'état complet et que « le plus récent l'emporte » est correct, ou si un événement tardif est inoffensif et qu'une réconciliation périodique le réparera, passez-vous de la machinerie. En faire moins est une décision qui a un coût, et elle devrait être consignée.

Erreurs courantes

- **Demander un ordre global.** Cela ressemble au choix par défaut sûr et plafonne le débit à un seul lecteur. Énoncez le besoin réel, l'ordre par agrégat, et concevez pour lui.
- **Prendre comme clé quelque chose qui n'est pas l'agrégat.** Prendre comme clé le type d'événement, le service ou une valeur aléatoire arrive parce que le réglage par défaut le fait pour vous. Le correctif consiste à prendre comme clé l'identifiant de l'entité dont le consommateur modifie l'état.
- **Utiliser les horodatages pour ordonner les événements.** Ils ressemblent à des numéros de version naturels. Le décalage d'horloge rend l'ordre dénué de sens ; transportez plutôt une version issue de l'agrégat.

- **Paralléliser dans un consommateur sans lanes par clé.** Un pool de threads augmente le débit et casse l'ordre de la partition. Hachez la clé vers une lane et donnez à chaque lane un seul worker.
- **Écarter des événements de delta périmés.** Une vérification de version convient aux événements porteurs d'état. Appliquée aux incréments et aux écritures comptables, elle fait disparaître de vrais changements. Classez chaque événement comme état ou delta avant de choisir une politique.
- **Oublier les tombstones.** Supprimer une ligne lors de l'annulation permet à un événement tardif de recréer la commande. Conservez la clé et la dernière version après la suppression aussi longtemps que des événements périmés peuvent arriver.
- **Retenir les événements en mémoire.** Un tampon qui meurt avec le processus perd des événements que le broker considère déjà comme livrés. Rendez-le persistant, dans la même transaction que la vue.
- **Changer le nombre de partitions en production sans plan.** La correspondance change pour environ deux tiers des clés lorsqu'on passe de quatre à six partitions, et l'ordre change en silence avec elle. Videz, migrez, ou protégez-vous avec des versions.
- **Mélanger les partitioners entre producteurs.** Des services écrits dans des langages différents peuvent hacher la même clé différemment. Fixez un seul partitioner partout, ou calculez la partition dans le relay.
- **Des lanes non bornées.** Une lane lente remplit alors la mémoire pendant que le point de commit reste immobile. Bornez chaque lane, mettez les partitions en pause, et faites les commits par partition depuis la boucle de polling.
- **Faire confiance à l'inbox pour les effets externes lors d'un rééquilibrage.** L'inbox ne couvre que la propre base de données du consommateur. Un e-mail ou un appel à un transporteur a besoin de sa propre clé d'idempotence.
- **Supposer que des clés partagées ordonnent les événements entre topics.** Le même `order_id` sur deux topics ne donne aucune séquence entre eux. Faites attendre le consommateur jusqu'à l'arrivée des deux, ou modélisez la dépendance dans un workflow.

Points clés

- L'ordre global vous coûte tout le parallélisme et il est rarement nécessaire ; l'ordre par agrégat est celui pour lequel il vaut la peine de concevoir.
- Une clé de partition qui identifie l'agrégat donne des événements ordonnés par entité sur un broker à log, mais seulement avec le même partitioner et le même nombre de partitions partout ; les brokers à files d'attente ont besoin d'une fonction de regroupement ou d'un consommateur qui tolère le désordre.
- Le parallélisme à l'intérieur d'un consommateur doit être organisé par clé, avec un worker par lane, borné, et avec un commit par partition, sinon il défait silencieusement l'ordre que la partition a livré.
- Les numéros de version incrémentés par l'agrégat dans sa transaction permettent à un consommateur de détecter les événements périmés, en double et manquants, ce que les horodatages ne peuvent pas faire.
- Ignorer un événement périmé est correct pour les événements porteurs d'état et faux pour les deltas et les écritures comptables, qui doivent être mis en tampon ou appliqués dans n'importe quel ordre.
- Les retry topics, le redrive et les changements de partitions réordonnent tous les messages, si bien que tout consommateur qui dépend de l'ordre a besoin d'une vérification de version avant de les utiliser.
- Chaque hold-back buffer a besoin d'un timeout, d'un responsable et d'une alerte, parce qu'un trou qui ne se referme jamais est un agrégat bloqué ; le blocage par clé du Chapitre 8 est la forme sans versions de la même idée.

L'ordre donne à chaque événement une place à l'intérieur d'un agrégat, mais un processus métier s'étend sur plusieurs agrégats et services, et certaines de ses étapes doivent être annulées lorsqu'une étape ultérieure échoue. Le Chapitre 10 se tourne vers les sagas, où l'ordre des étapes est la conception et où la compensation est la façon dont un workflow se rétablit.

PARTIE IV

Workflows et contrats

Chapitre 10 — Sagas et workflows de longue durée

Imaginez un mardi après-midi chez Parcelly. Un acheteur commande une lampe en céramique. Orders émet `OrderPlaced`, Payments autorise la carte, et Inventory réserve la dernière lampe du stock du vendeur. Puis la carte est refusée et Payments émet `PaymentFailed`. Au même moment, un second acheteur regarde une page produit qui affiche « rupture de stock », parce que la réservation de la première commande retient toujours la lampe.

Personne n'a commis d'erreur. Chaque service a fait son travail, chaque transaction locale a fait son commit, et chaque message a été livré. Le problème, c'est que l'opération métier, « vendre cette lampe à cet acheteur », est répartie sur quatre services et plusieurs minutes, et qu'aucune transaction ne la couvre. La moitié s'est produite. L'autre moitié ne se produira pas, et quelqu'un doit rendre la lampe.

Le moment compte. Rien n'a été capturé, donc l'annulation est peu coûteuse : libérer la réservation et laisser la commande se terminer. Si la même lampe avait déjà été payée et que la défaillance était survenue plus tard, la rendre serait un acte différent et plus coûteux, et le chapitre revient sur cette ligne de partage.

Supposez maintenant que la même défaillance survienne pendant que le vendeur retire l'annonce, pendant que l'acheteur modifie l'adresse de livraison, ou pendant que la réponse de la capture se perd en chemin. Ce chapitre porte sur la mécanique qui permet à un flux comme celui-ci de se terminer proprement ou de s'annuler proprement. Comment amener un workflow multiservice à se terminer dans un état que le métier accepte, quand on ne peut pas tenir un verrou sur toute sa durée et que n'importe quelle étape peut échouer ?

Le problème, et pourquoi il est difficile

Au sein d'une seule base de données, une transaction vous donne l'atomicité : soit chaque modification a lieu, soit aucune. Entre services, vous ne l'avez pas. Chaque service possède sa base de données, et le message broker se trouve entre eux. Le Chapitre 4 a montré comment rendre atomiques le changement d'état d'un service et l'événement qu'il publie. Il n'a pas rendu la réaction du service suivant atomique avec le changement du premier, et il ne le peut pas.

L'ancienne réponse était les transactions distribuées, en général le commit en deux phases. Elles tiennent des verrous chez chaque participant jusqu'à ce qu'un coordinateur décide, et si le coordinateur défaille après le vote des participants, ceux-ci attendent en tenant ces verrous jusqu'à son retour. Elles exigent des participants qui prennent en charge le protocole, et elles font du participant le plus lent ou le moins disponible la limite de tous les autres. Un fournisseur de paiement externe ou l'API d'un transporteur ne participe pas du tout à votre transaction. Pour un flux qui comprend un appel à un tiers, cette option est fermée avant même que vous commenciez à la peser.

Une **saga** est l'alternative. Elle découpe une opération métier en une séquence de transactions locales, une par service. Quand une étape échoue, la saga ne fait pas de rollback, parce que les transactions précédentes ont déjà fait leur commit et que d'autres ont peut-être lu leurs résultats. Elle exécute des **compensations** : de nouvelles transactions locales qui annulent sémantiquement les étapes précédentes. Une compensation est une action vers l'avant qui ramène le métier dans un état acceptable, pas une machine à remonter le temps.

Trois choses rendent cela difficile en pratique.

Premièrement, les états intermédiaires sont visibles. Entre « stock réservé » et « paiement capturé », la commande n'est ni confirmée ni annulée, et tout ce qui la lit voit cet état intermédiaire. L'atomicité a disparu ; l'isolation aussi.

Deuxièmement, la compensation est une décision de conception, pas un mécanisme. Certaines étapes ont un inverse net, comme la libération d'une réservation. D'autres n'en ont qu'un approximatif, comme le rem-

boursement d'un paiement capturé, qui coûte des frais et prend des jours. Certaines n'en ont aucun : un e-mail parti reste envoyé.

Troisièmement, la saga elle-même peut défaillir. Le processus qui pilote le workflow peut planter entre deux étapes, un message peut arriver deux fois, une réponse peut ne jamais arriver, et une compensation peut échouer tout comme une action. Tout ce que les chapitres 4 à 9 ont construit pour des messages isolés doit maintenant tenir pour un flux entier.

> **Remarque :** Un test utile avant d'adopter une saga : listez chaque étape avant le pivot (l'étape après laquelle la saga ne fait plus qu'avancer, définie plus loin dans ce chapitre) avec sa compensation, et dites ce qu'elle coûte. Les étapes après le pivot sont censées n'en avoir aucune. Si plusieurs étapes avant lui indiquent « aucune » ou « un ticket de support », la frontière entre ces services est peut-être au mauvais endroit, et déplacer les étapes dans la transaction locale d'un seul service peut coûter moins cher que de les coordonner.

La commande de Parcelly sous forme de saga

Reprenez le flux du happy path et nommez ses étapes comme celles d'une saga. Chaque étape est une transaction locale dans un service, suivie d'un événement écrit dans le transactional outbox de ce service au sein de la même transaction.

Étape	Service	Transaction locale	Compensation
1	Orders	Créer la commande dans un état en attente, émettre `OrderPlaced`	La marquer annulée, émettre `OrderCancelled`
2	Payments	Autoriser la carte, émettre `PaymentAuthorized`	Annuler l'autorisation, émettre `PaymentVoided`
3	Inventory	Réserver le stock, émettre `StockReserved`	Libérer la réservation, émettre `StockReleased`
4	Payments	Capturer le paiement, émettre `PaymentCaptured`	Aucune au sein de la saga : c'est le pivot (un remboursement relève d'un flux d'annulation distinct)

Étape	Service	Transaction locale	Compensation
5	Orders	Confirmer la commande, émettre `OrderConfirmed`	Aucune : on fait des retries jusqu'à ce qu'elle réussisse

Les deux étapes de Payments ne sont pas des transactions de base de données au sens strict. Chacune est un appel à un fournisseur externe avec une clé d'idempotence stable, suivi d'un enregistrement local du résultat, ce qui est le pattern du Chapitre 7. Les garanties de la saga reposent sur cette association, et un timeout sur l'appel au fournisseur porte l'incertitude décrite au Chapitre 3.

Les étapes 2 et 3 s'exécutent en parallèle, et la capture n'est demandée qu'une fois que `PaymentAuthorized` et `StockReserved` sont tous deux arrivés. Shipping réagit à `OrderConfirmed` et Notifications réagit à ce qu'émet Shipping. Nous considérerons pour l'instant ces deux derniers comme extérieurs à la saga et nous reviendrons sur la raison.

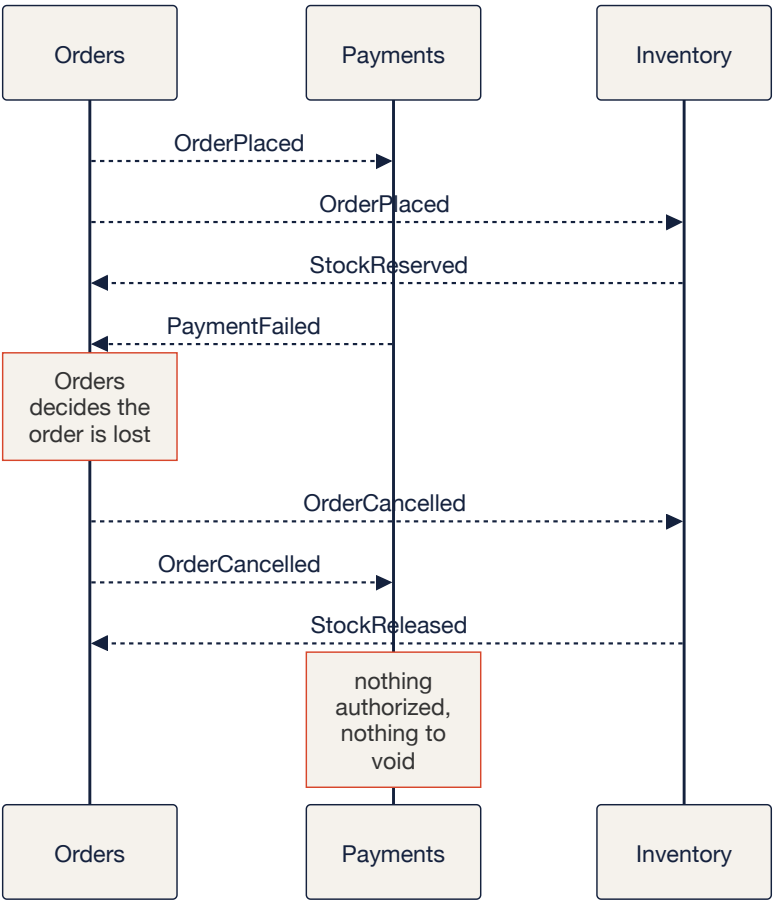
Les branches de défaillance suivent le tableau du flux d'événements. Si `PaymentFailed` arrive, aucune autorisation n'existe, donc la seule compensation consiste pour Inventory à libérer une réservation qui a peut-être été faite, ou peut-être pas, en émettant `StockReleased`. Si `StockReservationFailed` arrive, une autorisation est tenue et rien n'est capturé, donc Payments l'annule et émet `PaymentVoided`. Un remboursement n'apparaît qu'après la capture. Avant la capture, l'annulation est une libération ou un void, et l'appeler remboursement serait faux dans le registre et faux sur le relevé bancaire de l'acheteur.

Dans les deux branches de défaillance, Orders réagit et émet `OrderCancelled`, et les autres services compensent en réponse. Cette phrase constitue toute la conception chorégraphiée. Savoir si c'est toute la conception que vous voulez est la question suivante.

Chorégraphie

Dans une saga **chorégraphiée**, il n'y a pas de responsable central. Chaque service écoute des événements, exécute sa transaction locale et publie l'événement suivant. Le workflow n'existe que comme la somme de ces abonnements.

Le diagramme ci-dessous montre la branche de défaillance du paiement sous forme de chorégraphie. Regardez qui décide quoi : personne ne donne d'instruction, et chaque réaction est déclenchée par un événement auquel le service est déjà abonné.



Les points forts sont réels. Les services restent faiblement couplés : Inventory sait ce que signifie `OrderCancelled` et ne sait rien de Payments. Aucun composant unique ne peut, par sa défaillance, arrêter tous les workflows. Ajouter un consommateur, comme un service d'analytics qui réagit à `OrderConfirmed`, n'exige aucune modification des autres.

Les faiblesses apparaissent à mesure que le flux grandit. La définition du workflow est éparpillée entre des abonnements dans plusieurs bases de code, et on ne peut pas répondre à « que se passe-t-il après `StockReserved` ? » en lisant un seul fichier. Des dépendances cycliques

s'installent : Orders écoute Payments, qui écoute Orders. Chaque nouvelle branche ajoute des événements, et chaque service doit être mis à jour dans un ordre soigneux. Surtout, personne n'est responsable de la question « où en est la commande 81724 en ce moment ? ». La réponse est une jointure entre les logs de plusieurs services, ce qui est le problème que le Chapitre 1 a nommé et auquel le Chapitre 13 revient avec le tracing et la réconciliation.

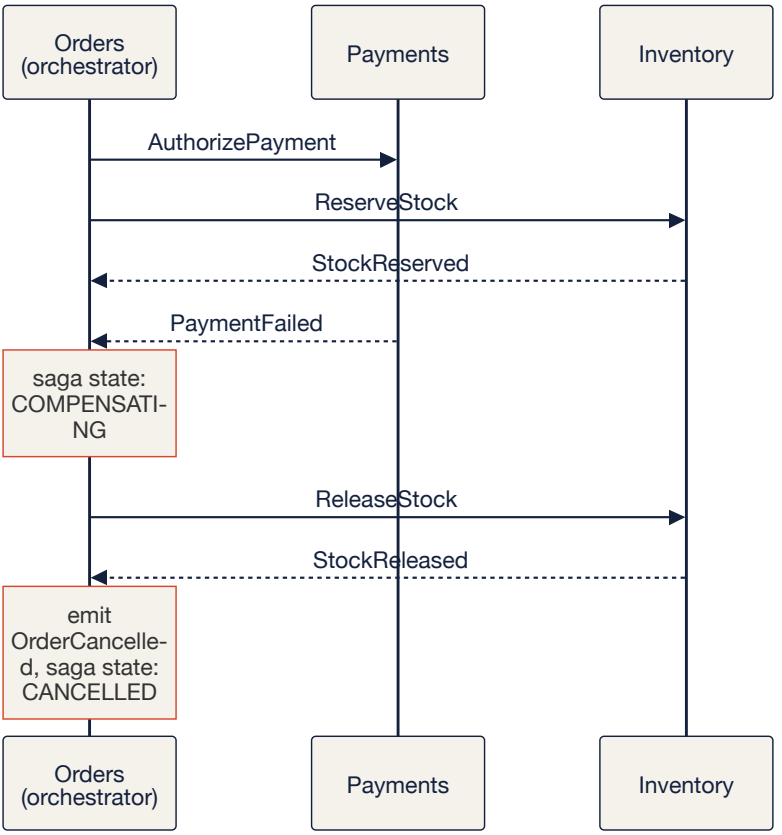
Je trouve que la chorégraphie fonctionne bien quand le flux est court, a peu de branches et qu'aucune étape ne dépend d'une conjonction d'étapes précédentes. Deux ou trois services qui réagissent à un même événement s'y prêtent naturellement. Le flux de commande de Parcelly a déjà franchi cette ligne, parce que la capture exige que deux résultats antérieurs soient arrivés.

Orchestration

Dans une saga **orchestrée**, un composant est responsable du workflow. Il détient l'état, décide de l'étape suivante, envoie des commands aux participants et réagit à leurs réponses. Les participants savent faire leur travail local et rendre compte du résultat ; ils ne savent pas où ils se situent dans le flux plus large.

Cette section introduit deux commands en plus des commands canoniques : `AuthorizePayment` (d'Orders vers Payments) et `ReserveStock` (d'Orders vers Inventory). La confirmation n'a besoin d'aucun command, parce qu'elle est la propre transaction locale de l'orchestrateur. Comme tous les commands de ce livre, ils portent un `command_id` à la place d'un `event_id`, et les réponses sont les événements ordinaires. Dans la version chorégraphiée, Payments et Inventory démarraient sur `OrderPlaced` ; ici, c'est l'orchestrateur qui leur dit de le faire.

Le diagramme montre la même défaillance que précédemment, mais désormais chaque flèche qui sort d'Orders est un command. Regardez comment l'état de la saga vit en un seul endroit et comment les commands de compensation nomment exactement le travail à annuler.



L'orchestrateur est l'endroit où l'on répond à la question « que se passe-t-il ensuite ? ». C'est une machine à états, et la machine à états peut être lue, testée et dessinée. Une nouvelle branche est une modification en un seul endroit. L'orchestrateur sait quelles étapes se sont achevées, de sorte qu'une compensation cible ce qui s'est réellement produit au lieu de tout ce qui aurait pu se produire.

Les coûts en sont l'image inversée. L'orchestrateur connaît chaque participant et devient un point de couplage. Si l'on n'y prend pas garde, la logique métier fuit des services vers lui, et les participants deviennent des appels de procédure distante avec des étapes en plus. C'est aussi un composant qui doit être correct, toujours disponible et lui-même récupérable, car un orchestrateur planté dont l'état est perdu est une saga que personne ne terminera jamais.

À mon avis, un orchestrateur qui vit dans le service responsable du processus métier est la forme la plus défendable. Chez Parcelly, c'est Orders, qui est déjà responsable du cycle de vie de la commande. Un service d'orchestration autonome, responsable des workflows de tout le monde, tend à devenir la chose que chaque équipe doit modifier pour livrer quoi que ce soit.

Choisir entre les deux

Aucun des deux styles n'est meilleur en général. Le choix dépend de la forme du flux et de qui doit répondre aux questions qui le concernent.

Question	Chorégraphie	Orchestration
Où le flux est-il défini ?	Réparti entre les abonnements	Dans une seule machine à états
Couplage entre services	Uniquement par les contrats d'événements	L'orchestrateur connaît chaque participant
Ajouter une étape ou une branche	Toucher plusieurs services	Toucher l'orchestrateur et le participant
Étapes parallèles et jointures	Malaisé ; chaque service suit ce qu'il a vu	Naturel ; l'orchestrateur détient la jointure
« Où en est la commande 81724 ? »	Jointure entre les logs	Lire une seule ligne
Point de défaillance unique	Aucun, mais aucun responsable non plus	L'orchestrateur, qui doit être récupérable
Risque quand il grandit	Dépendances cycliques et cachées	Un god service rempli de logique métier
Bon usage	Flux courts, peu de branches	Flux longs, conditions, timeouts, jointures

Mélanger les deux est normal et judicieux. Une saga de commande orchestrée peut quand même émettre `OrderConfirmed` comme un simple événement auquel Shipping et Notifications réagissent de manière chorégraphiée. La saga doit couvrir la partie du flux qui a besoin d'une annulation coordonnée ; les notifications et l'analytics n'en ont pas besoin, et les y intégrer ne fait qu'ajouter des états.

Compensations et étapes qu'on ne peut pas annuler

Concevez la compensation d'une étape quand vous concevez l'étape, pas quand le premier incident la réclame. Pour chaque étape, notez ce que veut le métier quand la saga échoue après elle.

Quelques règles gardent les compensations honnêtes.

Une compensation est une nouvelle action métier, pas une annulation dans la base de données. `StockReleased` ne supprime pas l'historique de la ligne de réservation. Il enregistre que la réservation a pris fin. `PaymentRefunded` n'efface pas la capture ; le registre contient les deux écritures. C'est pourquoi Ledger est append-only et pourquoi les compensations y apparaissent comme de vrais événements.

Ordonnez les étapes selon la difficulté de les annuler. Placez en premier les étapes faciles à inverser et en dernier celles qui sont difficiles ou impossibles à inverser. L'autorisation et la réservation sont peu coûteuses à annuler, c'est pourquoi elles viennent avant la capture. La capture est le point où un remboursement commence à coûter de l'argent et du temps. Cela mène à l'idée d'une **étape pivot** : le point après lequel la saga ne compense plus et ne fait plus qu'avancer. Chez Parcelly, la capture est le pivot. Avant elle, une défaillance signifie libération et void. Après elle, une défaillance de la confirmation n'est pas un repli mais un retry, parce que le métier détient l'argent de l'acheteur et lui doit une commande.

Certaines actions n'ont pas d'inverse. Un e-mail qui a été envoyé, un transporteur qui a reçu l'expédition, un webhook sur lequel le système d'un vendeur a agi : vous pouvez envoyer un suivi, mais vous ne pouvez pas désenvoyer. Ces étapes ont leur place après le pivot, et leur défaillance doit être traitée par des retries et par une correction, pas par une compensation. C'est aussi pourquoi Notifications est purement réactif et ne fait jamais partie de la saga : ses étapes ne peuvent pas être compensées et ne devraient pouvoir bloquer quoi que ce soit.

Les compensations doivent faire l'objet de retries jusqu'à ce qu'elles réussissent. L'échec d'une action mène à une compensation. L'échec d'une compensation ne mène à rien, sinon à une saga bloquée et à un système incohérent. Traitez une compensation comme quelque chose qui doit à terme s'achever, avec la discipline de retry du Chapitre 6 et le traitement

des poison messages du Chapitre 8. Une compensation qui échoue de façon permanente doit parvenir à un humain par un parking lot, avec l'état de la saga enregistré, et non être abandonnée.

Les compensations doivent être idempotentes et tolérer d'arriver en avance. Le premier point relève du Chapitre 7 : un `ReleaseStock` en doublon ne doit pas libérer deux fois. Le second est plus subtil. Supposez qu'Orders envoie `ReserveStock`, tombe en timeout et commence à compenser en envoyant `ReleaseStock`. La réservation était lente, pas perdue, et la libération peut atteindre Inventory en premier. Si Inventory traite une libération portant sur une réservation inconnue comme un no-op, le `ReserveStock` tardif réussit ensuite et la lampe est retenue pour toujours. La forme sûre consiste pour Inventory à enregistrer la libération au regard de la commande, afin que la réservation ultérieure la voie et refuse. Certains textes appellent cela un tombstone. Quel que soit le nom, la compensation doit laisser une marque que l'action peut lire.

État de la saga et timeouts

Une saga orchestrée a besoin d'un état durable, et celui-ci a sa place dans la même base de données que le service qui l'exécute, afin que les changements d'état rejoignent la même transaction locale que les écritures dans l'outbox et l'inbox.

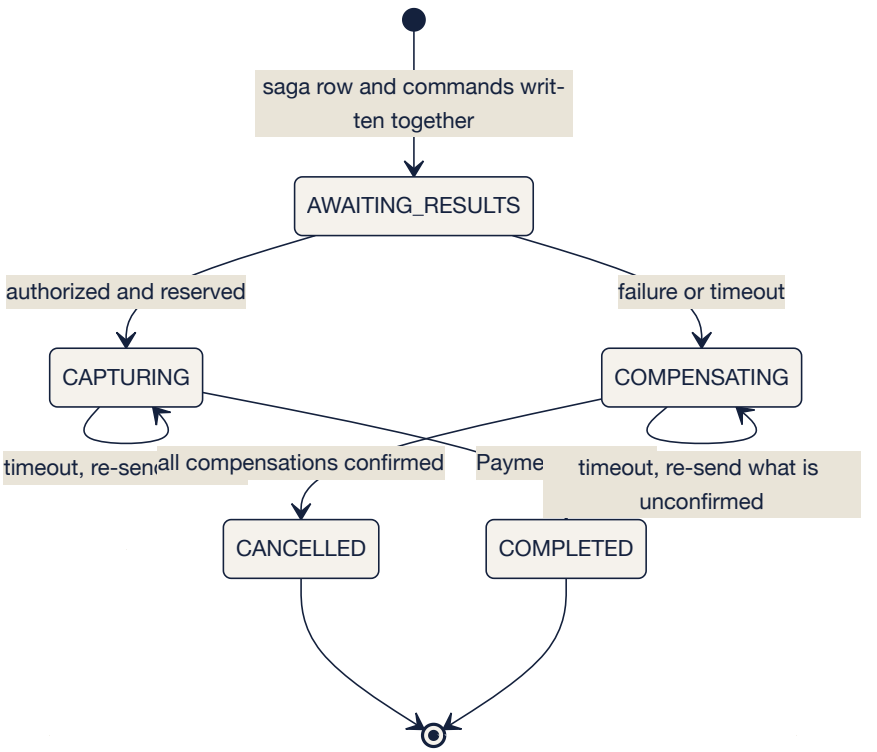
```
table saga_instance:
    saga_id        text primary key
    order_id       text unique not null
    state          text not null      // see below
    authorized     boolean not null
    reserved       boolean not null
    stock_released boolean not null   // compensation confirmed
    auth_voided    boolean not null   // compensation confirmed
    rounds         int not null       // deadlines spent in this state
    deadline_at    timestamp not null
    version        int not null
    updated_at     timestamp not null
```

La colonne `state` est un petit ensemble de positions nommées : `AWAITING_RESULTS`, `CAPTURING`, `COMPLETED`, `COMPENSATING`, `CANCELLED`. La ligne de la saga et les premiers commands sont écrits dans une seule transaction, il n'y a donc pas de position « démarrée » distincte que quiconque pourrait observer. Les deux booléens `authorized` et `reserved` en-

registrent quels résultats sont arrivés, et c'est ainsi que fonctionne la jointure : la capture est demandée quand les deux sont vrais et que l'état est `AWAITING_RESULTS`. Les deux indicateurs `stock_released` et `auth_voided` enregistrent quelles compensations ont été confirmées, de sorte que « toutes les compensations confirmées » est une question à laquelle la ligne peut répondre. Une compensation qui n'est pas nécessaire commence à l'état confirmé : après `PaymentFailed`, il n'y a aucune autorisation à annuler, donc `auth_voided` est vrai à l'entrée dans `COMPENSATING`.

Une contrainte `unique` sur `order_id` empêche deux sagas de s'exécuter pour une même commande. La colonne `version` protège contre deux handlers qui mettraient à jour la même ligne en même temps, et les handlers ci-dessous en dépendent.

Une machine à états dessinée une fois vaut plus que n'importe quelle description en prose. Le diagramme suivant montre les transitions légales ; regardez les sorties qui ne mènent pas à `COMPLETED`, et les deux états qui bouclent sur leur propre timeout.



Remarquez que `CAPTURING` n'a pas de sortie vers `COMPENSATING`. C'est le pivot exprimé dans la machine à états. Une capture dont la réponse n'est jamais venue est renvoyée, et si le fournisseur signale un refus définitif, celui-ci est traité comme un chemin métier distinct et explicite, avec un humain dedans.

Les boucles comptent autant que les sorties. Une saga en `CAPTURING` ou en `COMPENSATING` attend une réponse, et une réponse peut se perdre comme n'importe quel autre message. Si seul `AWAITING_RESULTS` avait une deadline, une saga bloquée dans l'un des deux autres états y resterait sans rien pour la réveiller. Chaque état non terminal a donc une deadline, et la deadline est réinitialisée à chaque transition.

Timeouts

Chaque attente dans une saga a besoin d'une deadline, parce qu'une réponse qui ne vient jamais est aussi une défaillance, et celle qu'on risque le plus d'oublier. Le Chapitre 3 a décrit comment un timeout cache trois possibilités : l'action a échoué, l'action a réussi et la réponse a été perdue, ou l'action n'est pas encore terminée. La saga ne peut pas savoir laquelle, elle a donc besoin d'une politique pour chaque attente, mise par écrit.

Un timeout est aussi un message. Ce chapitre introduit pour cela un événement interne, `SagaTimedOut`, émis par le propre timer de l'orchestrateur. L'implémentation habituelle est un processus timer qui sélectionne les sagas dont la `deadline_at` est dépassée et fournit à chacune un événement `SagaTimedOut` par le même handler que toutes les autres entrées. Ainsi, le timeout emprunte le même chemin que tout le reste, avec la même inbox, les mêmes vérifications d'état et le même outbox.

```
function handleSagaTimedOut(message):
    if inboxContains(message.event_id):
        // fast path only; the unique constraint is the safety net
        ack(message)
        return
    try:
        transaction:
            inboxInsert(message.event_id)
            saga = loadSaga(message.payload.saga_id)
            if saga.state == "AWAITING_RESULTS":
                saga.state = "COMPENSATING"
                saga.rounds = 0
                sendOutstandingCompensations(saga, message.event_id)
            else if saga.state == "COMPENSATING":
                saga.rounds = saga.rounds + 1
                sendOutstandingCompensations(saga, message.event_id)
            else if saga.state == "CAPTURING":
                // same command_id as the first attempt: a repeat, not
new work
                saga.rounds = saga.rounds + 1
                outboxInsert(CapturePayment, saga.order_id,
                             command_id: commandIdFor(saga.saga_id,
"capture"))
            // any other state (COMPLETED, CANCELLED): a stale
timeout, recorded and dropped

            // sets a clock for a non-terminal state; escalates past
MAX_ROUNDS
            armNextDeadline(saga)
            saveSaga(saga, expected_version: saga.version)
        ack(message)
    on UniqueViolation:
        // a concurrent duplicate committed first; the work is done
        ack(message)
    on VersionConflict:
        // another handler changed the row; nothing above committed
        nack(message)
```

La branche `AWAITING_RESULTS` envoie des commands de compensation sans demander quelle étape s'est achevée. C'est délibéré. Au moment du timeout, l'orchestrateur ne sait pas si un `StockReserved` tardif est en route, il demande donc les annulations encore en suspens et compte sur les participants pour traiter « rien à annuler » comme un succès. Le Chapitre 7 explique pourquoi c'est sûr.

`armNextDeadline` fait deux choses au sein de la même transaction. Elle fixe `deadline_at` à `now() + timeoutFor(saga.state)`, si bien qu'aucun état non terminal ne reste sans horloge. Et une fois que `saga.rounds` dépasse `MAX_ROUNDS`, disons cinq, la saga cesse d'être un problème de timing

et devient un problème humain : elle lève une alerte et enregistre l'état de la saga dans le parking lot du Chapitre 8, et les tours continuent. Les sagas terminées ne reçoivent pas de nouvelle deadline. Comme la deadline, le changement d'état et les commands font leur commit ensemble, un plantage ne peut pas laisser une saga qui a avancé mais n'est plus surveillée.

```
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))
```

Une saga en `CAPTURING` ne fait jamais demi-tour. Le pivot est passé, donc les seuls mouvements possibles sont de renvoyer la capture avec la même clé et de demander à une personne.

Choisir la durée d'un timeout n'est pas quelque chose que je peux faire pour vous. Cela dépend du temps que prend le chemin sain le plus lent, du temps qu'un acheteur acceptera d'attendre et de ce que coûte à un vendeur le fait de retenir du stock. Une deadline réglée sur une valeur très courte transforme la lenteur en échec et inonde le système de compensations. Une deadline très longue retient des réservations pendant que des acheteurs regardent une page en rupture de stock.

Les arrivées tardives ont aussi besoin d'une règle. Si `PaymentAuthorized` arrive après que la saga est passée à `CANCELLED`, l'orchestrateur doit émettre à nouveau `VoidAuthorization`, parce que quelqu'un retient des fonds sur une carte pour une commande qui n'existe pas. L'événement tardif prouve qu'une autorisation existe. Le void envoyé au timeout est peut-être encore en transit, il a peut-être été traité juste après que l'autorisation s'est achevée, ou l'appel au fournisseur s'est peut-être exécuté avant que Payments vérifie la marque que laisse le void. Renvoyer le void règle les trois cas.

Ce renvoi ne fonctionne que s'il n'est pas confondu avec une répétition. Ici, deux règles d'identité se rejoignent. Un command dont la répétition ne doit pas provoquer de nouveau travail, comme `CapturePayment`, garde un

seul `command_id` pour la saga et l'étape, de sorte qu'un retry est reconnu et reçoit en réponse le premier résultat. Une compensation qui peut légitimement être émise à nouveau tire son `command_id` du message qui l'a déclenchée : la saga, l'étape et l'`event_id` qui en est la cause. Le void réémis est alors un nouveau command. Il doit aussi être idempotent sur l'état cible, et pas seulement sur le command, si bien que Payments lit `VoidAuthorization` comme « aucune autorisation active ne doit subsister pour la commande 81724 » et le vérifie, au lieu de faire un replay de la réponse en cache « rien à annuler » de la première tentative. Une machine à états qui ignore les événements arrivant dans le mauvais état les perd. Traitez-les explicitement : décidez, par état et par événement, s'il faut appliquer, ignorer ou compenser.

Verrous sémantiques

Comme les sagas renoncent à l'isolation, d'autres flux peuvent lire et modifier une commande en pleine saga. Un acheteur peut modifier l'adresse de livraison pendant que la capture est en attente, un vendeur peut annuler, un agent de support peut rembourser. Les numéros de version du Chapitre 9 aident contre les événements périmés, mais ils n'empêchent pas deux flux d'apporter des modifications métier incompatibles.

Un **verrou sémantique** est un indicateur dans les données qui dit « une saga travaille là-dessus ». Ce n'est pas un verrou de base de données. C'est une valeur de statut que l'autre code est écrit pour respecter. Chez Parcelly, le statut de commande `PENDING` en est un : tant qu'il tient, la commande ne peut être ni modifiée, ni expédiée, ni remboursée, et la page de la commande affiche « en cours de traitement ». De même, la réservation dans Inventory est un verrou sur la lampe, ce qui explique pourquoi le second acheteur a vu « rupture de stock ».

Le verrou sémantique doit être libéré sur chaque chemin : à l'achèvement, à la compensation et au timeout. Une saga qui cale sans libérer son verrou laisse la commande figée, ce qui est une défaillance silencieuse qui fait surface sous forme de tickets de support. Le timeout ci-dessus est le garde-fou, et la réconciliation du Chapitre 13 est la seconde ligne de défense.

Deux autres contre-mesures méritent d'être connues. L'une consiste à rendre les mises à jour commutatives quand c'est possible, de sorte que l'ordre n'ait pas d'importance. L'autre consiste à concevoir une vue prudente : par exemple, montrer au vendeur un niveau de stock qui soustrait déjà les réservations en attente, pour que personne ne promette ce qui est sur le point d'être retenu. Aucune des deux ne supprime la nécessité de décider, pour chaque statut, ce que les autres flux peuvent faire pendant qu'une saga est en cours. Je considère ce tableau, statut par opération autorisée, comme faisant partie de la conception de toute saga.

Outbox et idempotence au sein d'une saga

Le handler de la saga est l'endroit où les chapitres précédents se rejoignent. Il consomme un message, change l'état et émet le command ou l'événement suivant. Fait sans soin, c'est un dual write en double.

La forme qui fonctionne combine les chapitres 4 et 7. Chaque handler exécute une transaction locale qui fait trois choses : enregistrer le message dans l'inbox, mettre à jour la ligne de la saga et écrire les commands ou événements sortants dans le transactional outbox. Le relay publie à partir de là.

```
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)
```

C'est la règle que le Chapitre 7 a introduite pour les consommateurs qui publient : la ligne de l'inbox, le changement d'état et l'insertion dans l'outbox font leur commit dans une seule transaction. Si le processus plante avant le commit, rien ne s'est passé et le broker procède à une nouvelle livraison. S'il plante après le commit et avant l'ack, la nouvelle livraison trouve la ligne de l'inbox, on en fait un ack et le travail est sauté ; un doublon concurrent perd au niveau de la contrainte d'unicité et on en fait un ack de la même manière. Un message périmé ou non pertinent est lui aussi enregistré et fait l'objet d'un ack, de sorte qu'il ne revient jamais. La saga avance une fois par message, quel que soit le nombre de fois où le message est livré.

La vérification `expected_version` est ce qui rend la jointure correcte. `PaymentAuthorized` et `StockReserved` peuvent être traités au même moment par deux workers, et chacun charge une ligne où les deux indicateurs sont faux. Sans la vérification, la seconde écriture écrase la première, et la saga attend dans `AWAITING_RESULTS` jusqu'à ce que son timeout compense une commande en bonne santé. Avec elle, une transaction perd, fait un rollback avec sa ligne d'inbox, et sa nouvelle livraison charge une ligne qui contient déjà l'autre indicateur. Charger la ligne avec un verrou de ligne obtient le même résultat.

Les commands que la saga envoie sont eux-mêmes des messages qui seront livrés at-least-once, le participant a donc besoin de la même protection, et c'est du `command_id` qu'elle provient. Pour une action qui ne doit pas se répéter, dérivez-le de la saga et du nom de l'étape. Cela permet à Payments de reconnaître un `CapturePayment` ayant fait l'objet d'un retry et de répondre avec le premier résultat. Si l'orchestrateur régénère un `command_id` aléatoire à chaque tentative, chaque retry ressemble à un nouveau travail, et l'acheteur peut être débité deux fois. Les compensations qui peuvent être réémises suivent l'autre règle de la section sur les arrivées tardives : l'id provient du message déclencheur, et le participant est idempotent sur l'état cible. Le `causation_id` et le `correlation_id` portés dans l'enveloppe rendent aussi chaque saga traçable : le `correlation_id` est le flux, et chaque `causation_id` en est une étape.

> **Avertissement :** Une compensation envoyée sans clé d'idempotence est aussi dangereuse qu'un paiement envoyé sans. Un `RefundPayment` ayant fait l'objet d'un retry sans clé stable peut rembourser l'acheteur deux fois, et le second remboursement est de l'argent qui ne revient pas.

La place des moteurs de workflow

Tout ce qui précède peut se construire avec une table, un timer et quelques handlers, et pour un flux de la taille de la commande de Parcelly, c'est par là que je commencerais. À un certain point, la mécanique devient la plus grande partie du code : timers durables, retries par étape, versionnement des workflows en cours d'exécution, historique visuel de chaque instance. C'est le problème que traitent les moteurs de workflow.

Cette catégorie d'outils est vaste. Certains sont des bibliothèques ou des services qui persistent la progression du workflow et la reprennent après un plantage, avec un code qui ressemble à un programme séquentiel. D'autres interprètent une définition déclarative d'états et de transitions. Les fournisseurs cloud proposent des versions managées, et plusieurs projets open source existent. Je ne comparerai pas les produits, parce que le paysage change plus vite qu'un livre et que le choix dépend de votre langage, de vos compétences opérationnelles et de votre tolérance à l'exploitation d'un composant critique de plus.

Ce que je demanderais à n'importe quel moteur tient en une courte liste. Où vit l'état du workflow, et pouvez-vous l'inspecter ? Comment rend-il atomiques l'effet de bord d'une étape et sa propre progression, ou vous laisse-t-il le dual write ? Qu'arrive-t-il à un workflow déjà en cours quand vous déployez une définition modifiée ? Comment exprime-t-on les compensations et les timeouts ? Un moteur qui stocke la progression mais exécute votre étape dans une transaction distincte a toujours besoin d'étapes idempotentes ; le moteur déplace le problème, il ne le supprime pas.

Le résumé honnête est qu'un moteur vous épargne la construction et l'exploitation de la machine à états, et vous coûte une dépendance, un modèle à apprendre et une chose de plus qui peut tomber en panne. Pour une ou deux petites sagas, une table coûte moins cher. Pour de nombreux workflows de longue durée avec des étapes humaines et de longues attentes, un moteur se rentabilise.

Compromis et coûts

Une saga échange l'atomicité contre la disponibilité et l'indépendance, et le prix se paie en plusieurs monnaies.

Complexité. Chaque saga multiplie les états. Un flux de trois étapes, chacune avec une action et une compensation, plus les timeouts et les arrivées tardives, a bien plus d'états atteignables que le happy path ne le laisse croire, et chacun exige une décision et un test. Le Chapitre 15 revient sur la manière de les tester.

États intermédiaires visibles. Il n'y a pas d'isolation. Les acheteurs, les vendeurs, les outils de support et les autres services voient une commande à moitié faite. Vous concevez en conséquence avec des verrous séman-

tiques et des statuts clairs, et vous acceptez que certains utilisateurs la voient quand même.

Cohérence à terme, volontairement. Entre une défaillance et la dernière compensation, le système est incohérent, pendant des secondes ou, avec un mauvais timeout, des heures. Ce n'est acceptable que si le métier y a consenti.

Responsabilité opérationnelle. Quelqu'un doit surveiller les sagas bloquées dans un état au-delà de leur deadline, les compensations qui ne peuvent pas s'achever et l'âge de la plus ancienne saga en cours. Sans responsable ni alarme, les sagas échouent en silence.

La compensation n'est pas gratuite. Un remboursement a des frais et un délai, une expédition annulée peut entraîner des frais, et un message à un acheteur disant « votre commande a été annulée » est lui-même un événement visible. Concevez le flux de sorte que la défaillance courante soit peu coûteuse à annuler.

Quand ne pas utiliser de saga : quand les étapes peuvent partager une transaction locale ; quand la défaillance est rare et qu'une correction manuelle coûte moins cher que de construire des compensations, ce qui est un choix légitime pour un processus interne à faible volume ; quand le « workflow » est en réalité un ensemble de réactions indépendantes qui n'ont besoin d'aucune annulation coordonnée, comme envoyer un e-mail ou mettre à jour un index de recherche. Et quand vous ne pouvez pas définir la compensation. Si une étape ne peut être ni annulée ni reportée, la question de conception honnête est de savoir si la frontière entre les services est au mauvais endroit.

Erreurs courantes

- **Construire une saga là où une transaction suffirait.** C'est tentant parce que les services existent déjà et que le diagramme a l'air moderne. Si deux étapes partagent un responsable et une base de données, fusionnez-les en une seule transaction locale.
- **Concevoir le happy path et laisser les compensations pour plus tard.** Plus tard arrive sous la forme d'un incident. Écrivez la compensation à côté de l'étape, et nommez l'événement qu'elle émet.

- **Appeler chaque annulation un rollback ou un remboursement.** Avant la capture, la compensation est une libération ou un void ; un remboursement n'existe qu'après la capture. Utiliser le mauvais terme cache l'état réel dans le registre et sur le relevé de l'acheteur.
- **Laisser une saga attendre indéfiniment.** Une deadline manquante est invisible jusqu'à ce qu'un client appelle. Donnez à chaque attente, dans chaque état non terminal, un timeout et une issue nommée, y compris les attentes de la réponse de capture et des confirmations de compensation.
- **Ignorer les événements qui arrivent dans le mauvais état.** Un `PaymentAuthorized` tardif pour une commande annulée, c'est une carte bloquée. Décidez par état et par événement s'il faut appliquer, ignorer ou compenser.
- **Des compensations non idempotentes.** Une libération ou un remboursement qui, après un retry, agit deux fois provoque un second incident en corrigeant le premier. Donnez aux actions qui ne doivent pas se répéter un `command_id` stable, dérivez l'id d'une compensation réémissible de sa cause, et rendez les participants idempotents sur l'état cible. Un id fixe pour un void renvoyé fait un replay de « rien à annuler » et laisse l'autorisation en place.
- **Mettre tout le métier dans l'orchestrateur.** L'orchestrateur doit décider de l'ordre et de l'issue ; les participants doivent être responsables des règles. Si l'orchestrateur valide les niveaux de stock, il fait le travail d'Inventory.
- **Placer tôt les étapes non compensables.** Envoyer la remise au transporteur avant la capture signifie qu'une défaillance vous laisse avec une expédition et sans argent. Ordonnez les étapes de sorte que celles qui sont difficiles à annuler viennent en dernier.
- **Sauter la libération du verrou.** Un verrou sémantique posé sur `PENDING` et jamais levé au timeout ou à la compensation fige la commande. Libérez-le sur chaque chemin de sortie.

Points clés

- Une saga remplace une transaction distribuée par une séquence de transactions locales et de compensations, et elle renonce pour cela à l'isolation.

- Une compensation est une nouvelle action métier, comme `StockReleased` ou `PaymentVoided`, pas un rollback, et avant la capture ce n'est jamais un remboursement.
- La chorégraphie convient aux flux courts avec peu de branches, l'orchestration aux flux avec des jointures, des timeouts et de nombreuses branches, et les deux sont souvent mélangées, avec l'orchestrateur dans le service responsable du processus.
- Ordonnez les étapes selon la difficulté de les annuler, et traitez la première étape qui ne peut pas être compensée, comme la capture, comme un pivot après lequel la saga ne fait plus qu'avancer.
- Chaque attente dans chaque état non terminal a besoin d'une deadline, chaque état a besoin d'une règle pour les événements tardifs et en doublon, et chaque compensation doit être idempotente et tolérer d'arriver avant l'action qu'elle annule.
- Au sein d'un handler, une seule transaction locale doit réunir l'inbox, l'état de la saga et l'outbox, protégée par une vérification de version pour que des handlers concurrents ne puissent pas s'écraser mutuellement, et un `command_id` fixe pour une action qui ne doit pas se répéter, comme la capture, mais dérivé du message déclencheur pour une compensation qui peut être réémise.
- Un moteur de workflow peut prendre en charge la machine à états, les timers et l'historique, mais il ne supprime pas le besoin d'étapes idempotentes ni d'une responsabilité claire des workflows bloqués.

Une saga suppose que les événements décrivent ce qui s'est passé sous une forme que chaque participant peut lire et à laquelle il peut se fier. Que ce soit le cas dépend de ce que porte chaque événement, de son nom et de son responsable, ce qui est le sujet du Chapitre 11.

Chapitre 11 — Conception des événements

Imaginez que l'équipe Shipping de Parcelly veuille planifier les expéditions plus vite. Aujourd'hui, elle reçoit `OrderConfirmed`, qui porte un `order_id` et pas grand-chose d'autre, puis appelle Orders en HTTP pour récupérer l'adresse de livraison et les lignes de la commande. Orders est lent le lundi matin, donc Shipping ralentit avec lui. Un tech lead propose la correction évidente : mettre toute la commande dans l'événement, pour que Shipping n'ait jamais besoin de demander.

En une semaine, l'événement a grossi. Orders ajoute l'adresse e-mail de l'acheteur, l'adresse complète, le compte de versement du vendeur et un champ appelé `internal_status` qu'utilise son propre écran d'administration. Notifications se met à lire l'e-mail dans l'événement au lieu de sa propre table. L'équipe Ledger remarque que le compte de versement transite désormais par le broker et se trouve dans un topic à longue durée de rétention. Puis Orders renomme `internal_status` pendant un refactoring, et trois consommateurs tombent en panne le même après-midi.

Personne dans cette histoire n'a commis d'erreur de fiabilité au sens des chapitres précédents. L'outbox a fonctionné, les consommateurs étaient idempotents, les retries étaient raisonnables. Le dommage est venu de la conception du message lui-même : ce qu'il dit, ce qu'il porte et qui a le droit de le modifier. Ce chapitre se demande comment concevoir un événement pour qu'il reste utile, sûr et modifiable pendant des années après que le premier consommateur l'a lu.

Le problème, et pourquoi il est difficile

Le Chapitre 1 a souligné que les services d'un système orienté événements sont couplés par le contrat de l'événement : son nom, ses champs et sa signification. Ce couplage est invisible au moment où vous écrivez l'événe-

ment, parce qu'à ce moment-là il y a un producteur et un consommateur, tous deux écrits par des gens qui travaillent à proximité les uns des autres. Le contrat ressemble à un détail interne.

Il cesse d'être interne au moment où un second consommateur s'abonne, et le producteur l'apprend en général tard, ou jamais. Sur un broker à log, un nouveau groupe de consommateurs peut commencer à lire le topic sans prévenir personne. Sur un broker à files d'attente, quelqu'un lie une nouvelle file au même exchange ou au même abonnement de topic. À partir de là, chaque champ du payload est une promesse faite à des gens que vous ne rencontrerez peut-être jamais.

Trois choses rendent cela difficile en pratique.

- **Il est facile d'ajouter aux événements et difficile de leur retirer.** Ajouter un champ coûte une ligne de code. Le supprimer exige de retrouver chaque consommateur, y compris ceux qui se trouvent dans des dépôts que vous ne lisez pas et ceux qui font un replay de l'ancien historique.
- **La pression de conception pousse dans la mauvaise direction.** L'équipe productrice veut livrer, elle émet donc tout ce qu'elle a sous la main. Les équipes consommatrices veulent éviter des appels supplémentaires, elles demandent donc davantage de champs. Personne n'est payé pour dire non.
- **Les erreurs sont durables.** Un événement écrit aujourd'hui peut rester des mois dans un log d'événements et faire l'objet d'un replay dans un consommateur qui n'existe pas encore. Un événement mal nommé ou un champ personnel qui a fuité reste dans le stockage longtemps après la disparition du code qui l'a produit.

La conception des événements n'est donc pas de la décoration. C'est la partie du système où quelques heures de réflexion décident du coût de chaque changement ultérieur.

Événements, commands et documents

La première décision de conception porte sur le type de message que vous envoyez. Trois types sont courants, et ils ont des règles différentes.

Un **événement** énonce un fait qui s'est déjà produit : `PaymentCaptured`. Le producteur ne se soucie pas de qui écoute, et il n'attend rien en retour. Les événements sont nommés au passé, parce qu'ils rapportent quelque chose qui s'est déjà produit ; un événement ultérieur peut le contrebalancer, mais pas le défaire. Un consommateur peut ignorer un événement, et c'est une réponse légitime.

Un **command** demande à un service précis de faire quelque chose : `CapturePayment`. Il a un seul destinataire prévu et l'attente que le destinataire agisse. Un command peut être rejeté, et l'émetteur se soucie en général du résultat. Les commands sont à l'impératif, portent un `command_id` à la place d'un `event_id`, et ne sont jamais appelés événements. Une saga orchestrée, que traite le Chapitre 10, pilote ses participants avec des commands de ce type.

Un **message de document** porte un bloc de données sans dire que quoi que ce soit s'est produit ni que quiconque doit agir, comme un snapshot nocturne du catalogue d'un vendeur. C'est le destinataire qui décide de ce qu'il signifie. Les messages de document sont légitimes, mais c'est le type le plus souvent déguisé en événement, et le déguisement sème la confusion.

Type	Dit	Destinataire	Nommage	Le destinataire peut-il refuser?
Événement	Un fait s'est produit	Tout abonné	Passé : `OrderPlaced`	Oui, en l'ignorant
Command	Fais ceci	Un service nommé	Impératif : `CapturePayment`	Oui, en répondant par un échec
Document	Voici des données	Quiconque le lit	Nom : `CatalogSnapshot`	Sans objet

L'échec courant est le command déguisé en événement. Supposez qu'Orders publie `SendConfirmationEmail`, un nom à l'impératif, et l'appelle un événement parce qu'il transite par le topic. Orders dépend désormais d'un certain comportement de Notifications, mais n'a aucun moyen d'entendre un refus. Si Notifications est en panne, Orders ne le sait pas. Si un second service s'abonne, il se retrouve face à une instruction qui ne lui est pas adressée.

J'appliquerais un test rapide. Demandez-vous si le producteur serait encore correct si personne ne consommait le message. Pour `OrderPlaced`, la réponse est oui : la commande a été passée quoi qu'il arrive. Pour `SendConfirmationEmail`, la réponse est non, parce que le message n'a de sens que si quelqu'un agit. Ce second type a sa place sur un canal où le destinataire est nommé et où l'échec revient à l'émetteur.

C'est aussi en partie une question de direction. Les événements vont du responsable d'un fait vers quiconque s'y intéresse. Les commands vont d'un coordinateur vers un exécutant. Quand le nom d'un événement commence à ressembler à une instruction adressée à un service particulier, la direction s'est inversée.

À quoi ressemble un command

Les deux types échouent aussi différemment, et l'enveloppe doit le refléter. Un événement perdu est silencieux par conception : le producteur n'attendait de personne en particulier qu'il agisse, donc personne ne remarque l'absence. Un command perdu ou en retard, c'est différent. L'émetteur attend un résultat, et la seule chose qui révèle la perte est sa propre deadline, ce qui explique pourquoi la saga du Chapitre 10 met une deadline sur chaque état non terminal. Un command peut aussi arriver après que l'émetteur a abandonné et compensé, le destinataire a donc besoin d'un moyen de le refuser.

Un command porte donc une enveloppe plus petite et différente de celle d'un événement :

```
{
    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` remplace `event_id`, et le Chapitre 10 explique comment il est dérivé : à partir de la saga et de l'étape pour une action qui ne doit pas se répéter, et avec l'`event_id` qui en est la cause pour une compensation qui peut être réémise. `reply_to` et `expects` donnent au résultat un chemin de retour, de sorte qu'un refus est quelque chose que l'émetteur peut entendre. `expires_at` permet à un destinataire d'écarter un command qui arrive après que l'émetteur a cessé d'attendre. Il est facultatif parce qu'il a un coût : il est comparé à l'horloge du destinataire, et le Chapitre 3 a donné les raisons pour lesquelles on ne peut pas se fier à cette horloge à la seconde près. Fixez-le avec une marge, et traitez-le comme une protection contre des commands vieux de plusieurs heures, pas de quelques secondes.

Événements de domaine et événements d'intégration

Au sein d'un service, le code émet souvent des événements sur ses propres agrégats : `OrderLineAdded`, `DiscountApplied`, `AddressCorrected`. Ces **événements de domaine** décrivent le modèle interne, et ils changent chaque fois que le modèle change. Ils sont utiles au sein du service, par exemple pour mettre à jour un modèle de lecture ou déclencher une règle dans la même transaction.

Un **événement d'intégration** est la version qui franchit la frontière du service. Il est choisi délibérément, nommé selon ce que les autres services ont besoin de savoir, et maintenu stable. `OrderPlaced` est un événement d'intégration. Ce n'est pas un dump de ce à quoi ressemblait l'agrégat de la commande à cet instant.

La tentation est de publier les événements de domaine directement sur le broker, parce que la mécanique existe déjà. À mon avis, c'est de loin la manière la plus courante de faire fuiter la structure interne. Pensez à ce qui se passe quand l'équipe Orders scinde un agrégat en deux, ou fusionne deux types de lignes. Si les événements de domaine étaient le contrat public, chaque consommateur casse, alors que rien n'a changé côté métier.

La correction est une étape de traduction dans le service producteur. Les événements de domaine restent libres de changer. Une petite fonction de correspondance décide lesquels d'entre eux deviennent des événements d'intégration, avec quels noms et quels champs. Cette fonction est l'endroit

où vous décidez de ce qu'est le contrat public, et le transactional outbox du Chapitre 4 est un endroit naturel pour l'appeler : l'événement d'intégration est ce qui est écrit dans la ligne de l'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
```

Notez que le nom interne `OrderSubmitted` et le nom public `OrderPlaced` diffèrent. C'est permis, et sain. Le nom public est celui qu'apprennent les autres équipes. Le nom interne peut suivre le code.

Thin events et fat events

Que doit porter le payload ? Les deux extrêmes méritent d'être nommés.

Un **thin event**, parfois appelé notification, porte un identifiant et guère plus : « la commande 81724 a été passée ». Un consommateur qui a besoin de détails rappelle le producteur. Un **fat event** porte assez d'état pour que les consommateurs agissent sans rappeler. Ce style est souvent appelé event-carried state transfer, parce que l'événement déplace un morceau d'état d'un service à un autre.

Chaque côté a un mode de défaillance que les chapitres précédents aident à expliquer.

Les thin events ramènent le couplage synchrone que les événements étaient censés supprimer. Shipping ne peut avancer que si Orders est joignable à cet instant, ce qui est exactement le problème du lundi matin de l'ouverture. Ils créent aussi une course : au moment où le consommateur rappelle, les données ont peut-être changé, et le consommateur lit un état plus récent que celui qui a déclenché l'événement. Si le rappel échoue, le consommateur tombe dans le monde du Chapitre 6, celui des retries contre une dépendance qui peut rester lente longtemps.

Les fat events coûtent ailleurs. Le payload grossit, et avec lui le nombre de champs que vous avez promis de conserver. Les consommateurs se mettent à traiter l'événement comme une réplique des données du producteur, et la réplique vieillit. Un consommateur qui a mis en cache l'adresse issue d'`OrderConfirmed` le lundi peut l'utiliser le vendredi, après que l'acheteur l'a corrigée, à moins qu'il lise aussi l'événement ultérieur. Et tout ce qui se trouve dans le payload est copié vers chaque consommateur et dans chaque segment de log conservé, ce qui augmente l'enjeu pour les champs sensibles.

	Thin	Fat
Payload	Identifiant et type d'événement	Identifiant plus l'état dont les consommateurs ont besoin
Le consommateur a besoin que le producteur soit disponible	Oui, pour récupérer les détails	Non
Risque d'obsolescence	Faible à la lecture, mais course avec les changements ultérieurs	Le consommateur détient un snapshot qui vieillit
Taille du contrat	Petite	Grande
Exposition de la vie privée	Faible	Plus élevée
Replay	Exige les données actuelles du producteur, qui peuvent différer de celles d'alors	Autonome, reproduit ce qui était connu

J'ai tendance à préférer une voie médiane : **porter ce dont le consommateur a besoin pour décider et agir, et rien de ce qu'il pourrait seulement vouloir**. Pour `OrderConfirmed`, Shipping a besoin de l'id de la commande, des lignes qu'il doit emballer et d'une adresse de livraison, ceux-ci ont donc leur place dans l'événement. Il n'a pas besoin du moyen de paie-

ment de l'acheteur, des coordonnées de versement du vendeur ni des règles de remise qui ont produit le total. Une manière utile de trouver la limite consiste à demander à chaque équipe consommatrice ce qu'elle ferait de chaque champ. Un champ sans consommateur nommé ni usage déclaré est candidat à la suppression.

Un second détail compte ici. Si l'événement porte de l'état, dites dans le payload à quel moment cet état appartient. Avec la version de l'agrégat (Chapitre 4), que le Chapitre 9 appelle `order_version` pour les commandes de Parcelly, un consommateur peut dire si le snapshot qu'il détient est plus récent ou plus ancien qu'un second événement ; `occurred_at` n'est qu'une indication approximative, pour les raisons données plus bas. Le Chapitre 9 explique comment détecter une arrivée dans le désordre avec de tels numéros, ce chapitre note donc seulement qu'un fat event sans version invite le piège du last-writer-wins qui y est décrit. Ne confondez pas ce numéro avec le `schema_version` de l'enveloppe : la version de l'agrégat compte les changements d'une commande et avance à chaque changement, tandis que `schema_version` nomme la version cassante de la forme du payload et avance rarement (Chapitre 12).

Le choix n'a pas besoin d'être global. Une équipe peut commencer thin pour les événements rarement utilisés et ajouter des champs quand un vrai consommateur le demande. Commencer thin et grandir coûte moins cher que commencer fat et réduire, parce que réduire implique un changement cassant, que le Chapitre 12 traite en détail.

Nommer les événements

Les noms d'événements de Parcelly utilisés tout au long du livre suivent un petit ensemble de règles, et cette section les énonce avec leurs raisons.

- **Au passé**, parce que l'événement est un fait : `PaymentCaptured`, pas `CapturePayment`, qui est le command.
- **Sujet en premier, en PascalCase** : l'agrégat vient en premier, puis ce qui s'est passé. `ShipmentDispatched`, pas `DispatchedShipment`. Triés par ordre alphabétique, les événements portant sur un même agrégat se regroupent dans les catalogues et les tableaux de bord.
- **Langage métier, pas langage technique.** `OrderConfirmed` dit quelque chose qu'un product owner comprend. `OrderRowUpdated` dit

quelque chose d'une table.

- **Ni technologie ni consommateur dans le nom.** `OrderPlacedKafkaEvent` fait fuiter le broker. `OrderPlacedForShipping` fait fuiter un consommateur, et le consommateur suivant devra se demander si l'événement lui est destiné.
- **Pas de préfixe de service, sauf si deux services émettent des faits sur le même nom.** Supposez qu'Orders et Shipping puissent tous deux signaler un `DeliveryDelayed` : l'un signifie que le vendeur n'a pas remis le colis, l'autre que le transporteur est en retard. Les consommateurs ne peuvent pas les distinguer par leur nom, la qualification devient donc nécessaire (`OrderDeliveryDelayed`, `ShipmentDeliveryDelayed`). L'exemple est illustratif ; les services de Parcelly de ce livre n'émettent pas les deux. Sinon, le producteur apparaît déjà dans le champ `producer`.

La règle de nommage la plus importante est celle qui porte sur le sens. `OrderUpdated` est un nom qui échoue au test du fait : il dit que quelque chose a changé et rien d'autre. Un consommateur doit comparer le nouvel état à l'ancien pour apprendre ce qui s'est passé, et il ne peut pas distinguer une adresse modifiée d'un total modifié. Préférez des noms qui énoncent le fait métier : `OrderCancelled`, `ShipmentAddressCorrected`. Quand vous ne trouvez pas de nom métier, c'est souvent le signe que l'événement décrit un changement de base de données plutôt qu'un changement métier.

Une autre règle est la stabilité. Une fois que des consommateurs dépendent du nom d'un événement, le renommer est un changement cassant qui coûte autant que de supprimer un champ. Passez les 10 minutes de plus sur le nom avant la première version.

L'enveloppe

Chaque événement de ce livre porte la même enveloppe : `event_id`, `event_type`, `schema_version`, `occurred_at`, `producer`, `correlation_id`, `causation_id`, `partition_key` et `payload`. L'enveloppe est la partie du message sur laquelle l'outillage peut s'appuyer sans

comprendre le métier. Un relay, un handler de dead letters, un système de tracing et un outil de replay peuvent tous travailler avec elle sans analyser le payload.

Chaque champ répond à une question que pose un composant :

Champ	La question à laquelle il répond	Qui la pose
`event_id`	Ai-je déjà vu exactement cet événement ?	Inbox, déduplication (Chapitre 7)
`event_type`	Comment interpréter le payload ?	Routage, désérialisation
`schema_version`	Quelle version cassante de la forme le payload a-t-il ?	Consommateurs, schema registry (Chapitre 12)
`occurred_at`	Quand le fait s'est-il produit, tel que le producteur l'a vu ?	Réconciliation, analytics, débogage
`producer`	Quel service est responsable de ce fait ?	Responsabilité, support, routage des questions
`correlation_id`	À quel flux métier cela appartient-il ?	Support, réconciliation, recherche dans les logs
`causation_id`	Qu'est-ce qui a directement causé ceci ?	Reconstitution de la chaîne
`partition_key`	Quel agrégat cet événement concerne-t-il ?	Broker, ordre (Chapitre 9)
`payload`	Quels sont les détails ?	Logique métier

Deux points demandent de l'attention. `occurred_at` est l'horloge du producteur, et le Chapitre 3 a expliqué pourquoi il ne faut pas ordonner les événements selon elle. Il vous dit à peu près quand le fait s'est produit, et il ne sert pas de séquence. La `partition_key` est l'identifiant de l'agrégat, et son choix a fait l'objet du Chapitre 9.

`correlation_id` n'est pas un trace id. Le contexte de trace (W3C Trace Context) voyage dans les en-têtes à côté de l'enveloppe, et le Chapitre 13 montre comment il est capturé dans l'outbox. Les traces sont échantillonnées et expirent ; `correlation_id` est la clé métier qui survit aux deux, et c'est celle par laquelle cherche un ingénieur du support.

L'enveloppe d'un langage à l'autre

Les services de Parcelly ne sont pas tous écrits dans le même langage. Orders et Payments tournent sur la JVM, Notifications est en Node, et les jobs de réconciliation de l'équipe Ledger sont en Python. La liste des champs ci-dessus dit ce que contient l'enveloppe. Elle ne dit pas comment chaque champ s'écrit sur le fil, et c'est là qu'un second langage casse un contrat qui semblait correct dans le premier.

Champ	Type sur le fil	Règle
`event_id`, `correlation_id`, `causation_id`, `partition_key`	string	Les identifiants sont des chaînes, même quand ils ont l'air numériques. Un id de 64 bits sous forme de nombre JSON est arrondi silencieusement par un consommateur JavaScript
`event_type`	string	Un nom stable et neutre vis-à-vis du langage, comme `PaymentCaptured`, jamais un nom de classe ou de package, et jamais un en-tête de type qu'un framework remplit par défaut
`schema_version`	integer	Petit, et un entier dans tous les langages
`occurred_at`	string	RFC 3339 en UTC avec un suffixe `Z`, à la précision fixe de la milliseconde, sans décalage. Les parseurs sont en désaccord sur tout le reste
Montants dans le `payload`	integer ou string	Des unités mineures entières (`total_minor`) dans une plage documentée, ou une chaîne décimale. Jamais un flottant. Les nombres JavaScript ne sont exacts que jusqu'à 2 ^ 53 - 1, donc un champ qui peut dépasser cette valeur voyage sous forme de chaîne
Champs de type énumération	string	Traitez l'ensemble comme ouvert : un consommateur doit gérer une valeur qu'il n'a jamais vue (Chapitre 12)

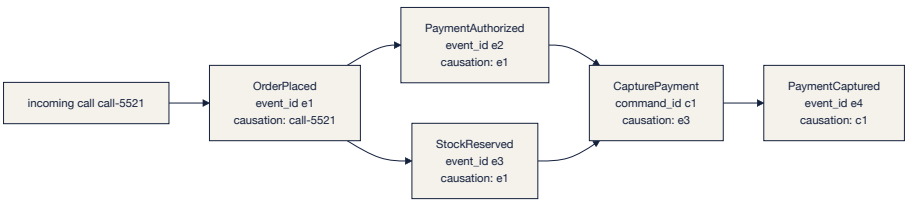
Deux règles de placement complètent le contrat. L'enveloppe vit dans le corps du message, et les en-têtes du broker ne font que refléter ce que dit déjà le corps, parce que les en-têtes sont la partie qui se perd quand un message est copié, envoyé vers la DLQ ou relayé vers un autre broker. Et le type de contenu est déclaré une seule fois, dans le schéma, et non deviné message par message. Ce qu'il advient des champs qu'un consommateur ne connaît pas, et la manière dont les formats diffèrent pour les décoder, est une question de compatibilité qui relève du Chapitre 12.

Corrélation et causalité

Les deux identifiants qui déroutent le plus souvent les lecteurs sont `correlation_id` et `causation_id`. Ils répondent à des questions différentes.

Le `correlation_id` est partagé par chaque événement d'un même flux métier. Quand un acheteur passe la commande 81724, tout ce qui suit porte `correlation_id: "order-81724"` : l'autorisation, la réservation, la capture, l'expédition. Vous cherchez par lui quand un acheteur écrit pour demander ce qui s'est passé.

Le `causation_id` pointe vers le seul message qui a directement causé celui-ci. C'est l'`event_id` d'un événement, ou le `command_id` d'un command. Il ne se répète pas au long du flux. Chaque saut le fixe à l'identifiant de son déclencheur immédiat, et suivre la chaîne à rebours reconstitue le chemin.



Regardez les deux dernières étapes. `CapturePayment` est déclenché par deux événements, et Orders doit en choisir un pour `causation_id`, ici celui qui a complété la condition. L'ambiguïté est réelle, et je mettrais la règle par écrit pour que tous les producteurs choisissent de la même manière. L'événement `PaymentCaptured` nomme le command, pas l'événement qui le précède, ce qui explique pourquoi l'enveloppe admet les deux types d'identifiants dans ce champ.

Chaque événement du flux partage `correlation_id: "order-81724"`. Le premier événement d'un flux fixe l'identifiant de corrélation, soit en le dérivant de l'agrégat, soit en reprenant celui de la requête entrante. Les événements suivants le copient sans le modifier.

Ce livre conserve tout du long l'id `order-81724`, à la portée de l'agrégat, et c'est un choix qui a une conséquence. Un id à la portée de l'agrégat est simple, mais chaque flux sur cette commande se fond sous lui : un remboursement trois semaines plus tard partage l'id avec le passage en caisse

initial, et on ne peut pas distinguer les deux. L'alternative est un id **à la portée du flux**, généré à neuf pour chaque flux (`checkout-81724-1`, `refund-81724-1`), avec `order_id` gardé consultable dans le payload. Il coûte une chose de plus à générer et à propager, et il vaut ce coût quand un agrégat peut démarrer un second flux que vous devez distinguer du premier. Le reste de ce chapitre tient dans les deux cas. Copier est tout le travail, et un consommateur qui oublie de le copier rompt la piste pour tout ce qui est en aval.

```
function buildFollowUpEvent(trigger, eventType, payload):
    // trigger is the message being handled: an event or a command
    return {
        event_id:       newEventId(),
        event_type:     eventType,
        schema_version: currentVersionOf(eventType),
        occurred_at:    now(),
        producer:       thisServiceName,
        correlation_id: trigger.correlation_id,  // copied, never
regenerated
        causation_id:   identifierOf(trigger),   // event_id or
command_id
        partition_key:  aggregateIdOf(payload),  // this event's
aggregate, not the trigger's
        payload:        payload
    }
```

La `partition_key` du déclencheur ne peut être copiée que si les deux messages concernent le même agrégat. Un événement de suivi portant sur un paiement ou une expédition qui hériterait de la clé de la commande serait ordonné par rapport au mauvais stream (Chapitre 9). Parcelly utilise la commande comme clé pour presque tout, ce qui explique pourquoi l'erreur est difficile à repérer.

Le prix d'une erreur ici se paie au pire moment. Pendant un incident, la question est « qu'est-il arrivé à cette commande à travers cinq services ? ». Sans `correlation_id` cohérent, un ingénieur fouille cinq logs à la main. Le Chapitre 13 montre comment le tracing et la réconciliation utilisent ces champs, le travail de ce chapitre consiste donc à s'assurer qu'ils sont présents et corrects dès le premier événement.

Ce qui n'a pas sa place dans un événement

Un événement quitte la base de données de son producteur et entre dans un système qui a d'autres règles d'accès, une autre rétention et de nombreux lecteurs. La question de conception est donc aussi une question sur ce que vous êtes prêt à copier dans ce système.

Données personnelles. Un événement contenant une adresse e-mail, un numéro de téléphone ou une adresse de livraison complète est stocké partout où le broker le stocke, peut-être pendant toute la fenêtre de rétention, et lu par chaque consommateur. Dans un log d'événements, supprimer plus tard les données d'une personne est malaisé, parce que le log est append-only et que les consommateurs ont déjà copié les champs. Certaines équipes répondent par des schémas de chiffrement ou des topics séparés. Je commencerais plus tôt : mettez un identifiant dans l'événement, comme `buyer_id`, et laissez un consommateur qui a un besoin légitime récupérer les champs personnels auprès du service responsable à ce moment-là, sous réserve des règles d'accès de ce service. Cela penche vers des thin events pour les champs sensibles, et je pense que l'échange en vaut la peine. Si un fat event a vraiment besoin d'une adresse, comme c'est le cas pour Shipping, l'événement doit la porter délibérément, pour ce public, avec une durée de rétention choisie en sachant qu'elle s'y trouve.

Quels champs comptent comme des données personnelles, et quelles obligations en découlent, dépend de votre juridiction et de vos conseillers juridiques. Je ne connais pas vos contraintes. L'habitude de conception est la même partout : chaque champ sensible d'un événement doit avoir une raison nommée.

Secrets et credentials. Les tokens, les clés et les numéros de compte de versement n'ont pas leur place dans un payload. Un message est journalisé, copié vers des destinations de dead letters, affiché dans des outils d'administration et joint à des tickets. Partez du principe que chaque payload sera lu par une personne.

Détails internes. Noms de tables, codes de statut internes, feature flags, ids de base de données qui n'ont de sens qu'à l'intérieur d'un service, et valeurs dont la signification dépend du code du producteur.

`internal_status`, dans le scénario d'ouverture, en est l'exemple. Un champ de ce genre devient une promesse publique dès que quelqu'un le lit.

Instructions destinées à des consommateurs précis. Un indicateur comme `skip_email_if_gift` dit à Notifications comment se comporter. L'événement doit énoncer le fait, et c'est le consommateur qui décide quoi en faire.

> **Avertissement :** Tout ce qui entre dans un log d'événements doit être traité comme permanent et largement lisible. Un secret ou un champ personnel écrit une seule fois est copié vers des destinations de dead letters, des sauvegardes et des outils de replay, et le supprimer partout est bien plus difficile que de ne pas l'y mettre.

Une courte checklist aide lors de la revue d'un nouvel événement : chaque champ a-t-il un consommateur qui en a besoin, chaque champ est-il quelque chose que le producteur est prêt à maintenir stable, et le champ pourrait-il nuire à quelqu'un s'il était lu par une personne qui ne devrait pas le voir ?

Granularité

Quelle taille doit avoir un événement ? Il y a deux erreurs opposées.

Trop fin. Publier un événement `FieldChanged` pour chaque mise à jour de colonne produit un flot de bruit. Les consommateurs doivent reconstituer le sens à partir de nombreux messages, et l'ordre de ces messages commence à compter d'une manière contre laquelle le Chapitre 9 a mis en garde. Un changement qui était un seul fait métier, l'acheteur qui corrige une adresse, arrive désormais sous forme de quatre événements qu'on peut voir dans un état à moitié appliqué.

Trop grossier. Un événement `OrderChanged` qui se déclenche à chaque modification et porte toute la commande oblige chaque consommateur à comparer le nouvel état à l'ancien, et à deviner quel en était le sens métier. Cela signifie aussi que chaque consommateur se réveille à chaque changement, y compris ceux qui ne l'intéressent pas.

J'utiliserais le fait métier comme unité : un événement par décision ou occurrence que quelqu'un du métier nommerait. Un acheteur qui passe une commande est un fait. Un paiement capturé en est un autre. Quand une transaction produit plusieurs faits, publiez plusieurs événements par la même écriture dans le transactional outbox, pour qu'ils fassent leur commit ou leur rollback ensemble.

Un test pratique consiste à se demander si un consommateur pourrait réagir de manière utile à cet événement seul. Si la réponse est « seulement après l'arrivée des trois événements suivants », la granularité est trop fine. Si la plupart des consommateurs ignorent la plus grande partie du payload, ou si le nom de l'événement a besoin d'un « et » pour le décrire, il est peut-être trop grossier et cache peut-être deux faits.

Responsabilité et contrat public

Chaque événement a exactement un responsable : le service responsable de l'agrégat que l'événement décrit. Payments est responsable de `PaymentCaptured`. Orders est responsable d'`OrderPlaced`. Les consommateurs peuvent lire l'événement, mais seul le responsable le définit. Un consommateur qui veut une forme différente le demande au responsable, et ne publie pas sa propre variante sous le même nom.

Traiter un événement comme un **contrat public** a des conséquences pratiques, et je les inscrirais dans les accords de travail de l'équipe.

- **Le responsable documente l'événement** : son nom, sa signification, quand il est émis, quand il ne l'est pas, chaque champ avec son type et sa signification, et les garanties sur l'ordre et la clé de partition.
- **L'événement a une version et un processus de modification.** Le Chapitre 12 décrit comment faire évoluer les schémas sans casser les consommateurs. La règle de base a sa place ici : le responsable ne modifie pas un événement publié sans connaître les conséquences pour ses lecteurs.
- **Les consommateurs se déclarent.** Un court registre de qui consomme quoi transforme un abonnement silencieux en dépendance connue. Il n'a pas besoin d'être sophistiqué. Une liste dans le dépôt qu'une pull request doit mettre à jour suffit pour commencer.

- **Les consommateurs ne lisent que ce dont ils ont besoin.** La règle du tolerant reader du Chapitre 12 commence ici : ignorez les champs que vous n'utilisez pas, pour que les changements additifs ne vous cassent pas.
- **L'émission fait partie du comportement.** Un service qui cesse d'émettre `OrderCancelled` dans un chemin de code a modifié son interface publique tout autant qu'un service qui supprime un endpoint HTTP.

Les mots « contrat » et « responsable » pèsent tous deux. Un contrat peut être testé. Un responsable peut être appelé en astreinte. Si personne ne peut dire qui est responsable d'un événement, personne ne peut dire qui a le droit de le modifier, et le scénario d'ouverture se répète.

Compromis et coûts

La discipline coûte de la vitesse. Faire correspondre les événements de domaine à des événements d'intégration, nommer avec soin, écrire la documentation et tenir une liste des consommateurs ralentissent tous la première version. Le bénéfice arrive plus tard, sous forme de changements moins coûteux. Pour un prototype avec un producteur et un consommateur écrits par la même personne, le cérémonial peut ne pas être rentable, et je le garderais léger jusqu'à ce qu'une deuxième équipe apparaisse.

Les fat events échangent le couplage dans le temps contre le couplage à la forme. Les consommateurs n'ont plus besoin que le producteur soit joignable, mais ils dépendent des champs qu'il copie. Un payload plus gros signifie un contrat plus gros, davantage à maintenir stable et davantage à protéger.

Les thin events sont peu coûteux à publier et coûteux à consommer. Chaque consommateur paie un rappel, et le producteur doit servir cette charge de lecture, qui culmine en général au même moment que l'événement. Si l'API du producteur est lente, le thin event déplace le problème du producteur vers chaque consommateur.

Des enveloppes soignées coûtent des octets et de l'attention. Neuf champs sur chaque message s'additionnent à fort volume, et chaque producteur doit les remplir correctement. L'alternative est de reconstituer l'in-

formation manquante pendant un incident, ce qui coûte plus cher. L'importance de ce point dépend de votre volume de messages. L'approche habituelle consiste à garder l'enveloppe petite et le payload léger.

Les règles au niveau du fil coûtent de la flexibilité. Des chaînes pour les ids et les montants, un format d'horodatage fixe et une enveloppe dans le corps rendent un système polyglotte ennuyeux, au bon sens du terme. Elles impliquent aussi des conversions aux frontières, et une équipe qui veut des types natifs dans son propre langage paie cette conversion sur chaque message.

L'expiration d'un command coûte une dépendance à l'horloge. `expires_at` protège contre les commands périmés, mais il est évalué sur l'horloge du destinataire. La marge que vous ajoutez est le temps pendant lequel un command véritablement en retard est encore accepté.

Les événements anonymisés ou réduits à un identifiant limitent ce que les consommateurs peuvent faire seuls. Un consommateur qui doit contacter l'acheteur a besoin d'un appel au responsable ou d'une copie locale des données qu'on lui a donné une raison de détenir. C'est une friction, et c'est une friction choisie.

Quand ne pas mettre autant de structure autour d'un événement. Un événement purement interne à un service, consommé par la même base de code et jamais conservé, n'a besoin ni d'enveloppe, ni de version, ni de liste de responsables. Réservez le traitement complet aux événements qui franchissent une frontière, et en particulier à ceux qui survivent à un déploiement.

Erreurs courantes

- **Publier directement les événements de domaine.** C'est tentant parce que la mécanique existe, et le résultat est que les refactorings internes deviennent des changements cassants. Ajoutez une correspondance explicite vers des événements d'intégration et traitez cette correspondance comme le contrat.
- **Appeler un command un événement.** Le passé est retiré ou ajouté pour la forme, et le producteur finit par dépendre d'un destinataire

précis sans moyen d'entendre un refus. Envoyez les commands à un destinataire nommé et attendez un résultat en retour.

- **Les événements `Updated`.** `OrderUpdated` ne dit rien de ce qui a changé, donc chaque consommateur compare les états et devine. Nommez le fait métier, et si aucun nom n'existe, demandez-vous si l'événement est nécessaire.
- **Déverser tout l'agrégat dans le payload.** Cela épargne une conversation et crée un contrat public de la taille de la table de la base de données. Demandez à chaque consommateur ce qu'il fait de chaque champ, et coupez ce que personne n'utilise.
- **Mettre des données personnelles et des champs internes parce qu'ils étaient sous la main.** Corrigez cela au moment de la revue avec une checklist, car après la publication les données sont déjà dans un stockage que vous ne contrôlez pas.
- **Encoder selon les valeurs par défaut du langage.** Un nom de classe comme `event_type`, des ids en nombres, des montants en flottants, des horodatages avec le décalage que le runtime a choisi. Cela fonctionne jusqu'à ce que le second langage le lise. Mettez les règles du fil par écrit une fois, comme le fait le tableau ci-dessus.
- **Copier la `partition_key` du déclencheur.** C'est juste quand les deux messages concernent un même agrégat et faux pour un événement de suivi qui porte sur un autre. Dérivez la clé de l'agrégat propre à l'événement.
- **Régénérer `correlation_id` à chaque saut.** Quelqu'un écrit un helper qui crée un nouvel identifiant par défaut. Copiez-le depuis le déclencheur et testez que chaque consommateur le fait.
- **Utiliser `occurred_at` comme ordre.** Un horodatage est une lecture d'horloge, et les horloges ne sont pas d'accord. Utilisez la clé de partition et un numéro de version comme le décrit le Chapitre 9.
- **Pas de responsable.** Un événement émis par « la plateforme » ou par deux services sous un même nom n'a personne qui puisse approuver une modification. Attribuez un seul service responsable par type d'événement.

Points clés

- Un événement énonce un fait, un command demande une action à un destinataire nommé, et un document porte des données ; mélanger ces trois types mène à un couplage caché et à des refus perdus, et un command a besoin d'un chemin de réponse et d'une expiration facultative parce que la perte d'un command n'est remarquée que par la deadline de l'émetteur.
- Les événements de domaine décrivent le modèle interne d'un service, tandis que les événements d'intégration sont le contrat public choisi délibérément, et une correspondance explicite entre eux empêche un refactoring de devenir un changement cassant.
- Les thin events gardent les contrats petits mais réintroduisent le couplage à l'exécution et les courses lors des rappels, tandis que les fat events suppriment le rappel mais agrandissent le contrat ; portez donc ce dont les consommateurs ont besoin pour agir et rien de ce qu'ils pourraient seulement vouloir.
- Nommez les événements au passé avec le sujet en premier, utilisez le langage métier, et évitez les noms du type `Updated` qui cachent le fait.
- L'enveloppe existe pour que l'outillage puisse dédupliquer, router, tracer et ordonner sans lire le payload ; son encodage sur le fil (ids en chaînes, horodatages RFC 3339 en UTC, montants sûrs, noms de type neutres vis-à-vis du langage) est ce qui la garde lisible dans tous les langages, et `correlation_id` est copié tout au long du flux tandis que `causation_id` pointe vers le seul déclencheur direct.
- Gardez les données personnelles, les secrets, l'état interne et les instructions propres à un consommateur hors des payloads, parce que tout ce qui se trouve dans un log d'événements est largement copié et difficilement supprimé.
- Chaque événement a un seul service responsable, une signification documentée et une liste de consommateurs connus, et c'est ce qui transforme un message en contrat qu'on peut modifier à dessein.

Un événement bien conçu doit tout de même changer, parce que le métier changera, et que les consommateurs se mettront à jour à des rythmes différents pendant que les anciens événements restent dans le log. Le Cha-

pitre 12 se tourne vers l'évolution des schémas : comment modifier un contrat dont d'autres équipes dépendent déjà sans les casser.

Chapitre 12 — Évolution des schémas

Imaginez que l'équipe Payments de Parcelly veuille prendre en charge les captures partielles. Un vendeur expédie la moitié d'une commande de deux articles, et l'acheteur doit être débité de cette moitié maintenant et du reste plus tard. L'événement `PaymentCaptured` porte un seul champ, `amount_minor`, qui jusqu'ici a toujours signifié « la commande entière ». L'équipe le renomme en `captured_minor` et ajoute `remaining_minor`, ce qui est plus propre et dit ce que cela signifie. Les tests du dépôt Payments passent, et le déploiement part tard un jeudi.

Ledger ne se trouve pas dans ce dépôt. Il lit `amount_minor` pour passer une écriture en partie double, ne trouve rien et rejette chaque nouvel événement comme mal formé. Notifications lit le même champ pour un reçu et imprime un montant vide. Un job de reporting qui fait un replay des événements du mois dernier rencontre désormais deux populations dans le même topic, les anciens noms et les nouveaux, et son parseur n'en gère qu'une.

Rien de tout cela n'est un bug de Payments. Le code a fait ce que l'équipe a demandé. L'échec tient au fait qu'un événement est lu par des programmes que son auteur ne rencontre jamais, et que certains d'entre eux lisent des événements écrits il y a des mois. Comment changer un contrat dont vous ne contrôlez pas les lecteurs, alors que les anciennes écritures sont toujours dans le log ?

Le problème, et pourquoi il est difficile

Un schéma d'événement est un contrat entre un producteur et chaque consommateur, présent et futur. Une signature de fonction dans votre propre base de code est aussi un contrat, mais un compilateur vérifie les deux côtés et vous pouvez changer les deux dans un seul commit. Un sché-

ma d'événement n'a pas de compilateur entre les deux côtés. Producteur et consommateurs se déploient à des jours différents, appartiennent à des équipes différentes, et le broker accepte tous les octets qui arrivent.

Trois propriétés rendent cela plus difficile que le versionnement ordinaire d'une API.

Premièrement, **les événements survivent au code qui les a écrits**. Sur un broker à log, les événements restent pendant la période de rétention configurée, et certains systèmes les conservent indéfiniment. Sur un broker à files d'attente, la fenêtre est plus courte, mais un backlog ou une dead-letter queue (DLQ) peut retenir des messages pendant des jours. Un consommateur déployé aujourd'hui peut devoir lire quelque chose écrit sous un schéma d'il y a un an.

Deuxièmement, **le replay fait du passé une entrée vivante**. Le replay, défini au Chapitre 2, est la façon dont vous reconstruisez une projection, faites le backfill d'un nouveau service ou récupérez après un bug. Il remet au code d'aujourd'hui tous les schémas historiques à la fois. Un consommateur qui ne gère que la forme la plus récente fonctionne jusqu'au jour où quelqu'un fait un replay, et il échoue alors d'abord sur les événements les plus anciens. La forme n'en est que la moitié : le replay applique aussi les règles métier et les effets de bord d'aujourd'hui à des faits anciens, comme envoyer à nouveau un reçu ou appeler à nouveau le fournisseur de paiement. Un consommateur reconstruit par replay doit savoir qu'il est en train de faire un replay, par un mode explicite ou une plage d'offsets, supprimer ses effets externes et appliquer des règles valables pour des faits historiques. Le Chapitre 7 traite du volet déduplication.

Troisièmement, **vous ne pouvez pas mettre tout le monde à niveau d'un coup**. Les déploiements progressifs font cohabiter l'ancienne et la nouvelle version d'un même service pendant des minutes ou des heures. Un changement correct une fois que tout a été mis à niveau reste un incident pendant la mise à niveau.

L'approche évidente consiste à traiter le schéma comme un détail interne et à corriger les consommateurs après coup. Cela fonctionne tant qu'il y a un consommateur et une équipe. Cela cesse de fonctionner quand le producteur ne sait pas combien de consommateurs existent, ce qui est préci-

sément la situation que la conception orientée événements cherche à créer. Le Chapitre 11 a soutenu qu'un événement publié est un contrat public. L'évolution des schémas est la façon de tenir cette promesse dans le temps.

La compatibilité, définie

Les mêmes mots sont employés de façon approximative à bien des endroits, voici donc les définitions qu'utilise ce livre.

- **Compatibilité vers l'arrière :** un consommateur utilisant le nouveau schéma peut lire des événements écrits avec l'ancien schéma. Nouveaux lecteurs, anciennes données.
- **Compatibilité vers l'avant :** un consommateur utilisant l'ancien schéma peut lire des événements écrits avec le nouveau schéma. Anciens lecteurs, nouvelles données.
- **Compatibilité totale :** les deux à la fois.
- Les variantes **transitives** de chacune vérifient le nouveau schéma par rapport à toutes les versions antérieures, pas seulement la plus récente.

Le sens est facile à inverser mentalement, alors je pose plutôt une question concrète : « si je déploie ce changement, qui casse ? »

Compatibilité	Qui lit quoi	Ordre de déploiement sûr	Protège contre
Vers l'arrière	Nouveau lecteur, anciennes données	Les consommateurs d'abord, puis les producteurs	Le replay de l'historique, les backlogs en retard
Vers l'avant	Ancien lecteur, nouvelles données	Les producteurs d'abord, puis les consommateurs	Les consommateurs qu'on ne peut pas mettre à niveau à la demande
Totale	Les deux sens	N'importe quel ordre	Les parcs mixtes, les consommateurs inconnus
Aucune	Rien n'est garanti	Coordonné, avec un plan de migration	Rien ; voir la section sur la migration

Une réserve sur les noms. Les registries et les bibliothèques de sérialisation emploient ces mots avec de petites différences. Lisez les définitions de l'outil que vous utilisez avant de vous fier à un réglage, et vérifiez quelle

variante votre registry applique par défaut : une règle non transitive ne suffit pas pour un log avec replay. Testez avec de vrais événements d'exemple de chaque version.

Pour un log avec replay, la compatibilité vers l'arrière par rapport à tout l'historique est la propriété dont vous avez besoin, car un consommateur qui fait un replay peut rencontrer n'importe quelle version jamais publiée. Pour une file d'attente à rétention courte, une vérification par rapport à la version précédente peut suffire. C'est une décision de rétention autant qu'une décision de schéma.

Changements sûrs et changements incompatibles

Un changement est **additif** quand les anciens consommateurs peuvent l'ignorer et que les nouveaux consommateurs savent faire face à son absence. Un changement est **incompatible** quand un lecteur, ancien ou nouveau, ne peut plus interpréter un événement. Les verdicts ci-dessous supposent que les lecteurs se comportent comme décrit dans la section sur le tolerant reader.

Changement sur `PaymentCaptured`	Verdict habituel	Pourquoi
Ajouter un champ optionnel (`payment_method`)	Sûr	Les anciens lecteurs l'ignorent ; les nouveaux gèrent son absence
Ajouter un champ obligatoire sans valeur par défaut	Incompatible	Les nouveaux lecteurs ne peuvent pas interpréter les anciens événements
Supprimer un champ optionnel	Sûr seulement si personne ne le lit	Les lecteurs qui en dépendent ne voient rien
Supprimer un champ obligatoire	Incompatible	Les anciens lecteurs l'attendent
Renommer un champ	Incompatible	Se lit comme une suppression plus un ajout ; quelques formats peuvent résoudre un renommage avec des alias
Changer un type (`int` vers `string`)	Incompatible	Les parseurs échouent ou convertissent en silence
Élargir un type numérique	Dépend du format	Certains formats promeuvent, d'autres non

Changement sur `PaymentCaptured`	Verdict habituel	Pourquoi
Changer le sens d'un champ	Incompatible, et le pire de tous	Rien n'échoue ; les résultats sont faux
Ajouter une valeur à une énumération	Risqué	Les anciens lecteurs peuvent la rejeter ou la traiter mal

Deux lignes méritent qu'on s'y attarde.

Changer le sens est le changement dangereux

Renommer un champ casse bruyamment. Ledger échoue sur le premier événement, une alerte se déclenche et quelqu'un lit la stack trace. Changer ce que signifie un champ, en laissant intacts son nom et son type, casse en silence. Imaginez que Payments décide que `amount_minor` signifiera désormais « montant capturé dans cet événement » au lieu de « montant total de la commande ». Chaque parseur réussit toujours. Ledger enregistre des montants partiels comme s'il s'agissait de totaux, et l'erreur fait surface des semaines plus tard lors d'une réconciliation (Chapitre 13), sous la forme d'un écart que personne ne sait expliquer.

Aucun outil de schéma ne le détecte, puisque le schéma est inchangé. Seul un humain qui relit le changement le voit. Ma règle : si le sens change, c'est un nouveau champ ou un nouveau type d'événement, jamais une modification de l'ancien.

Les énumérations sont un contrat caché

Ajouter une valeur à une énumération est additif dans le schéma et souvent pas dans les consommateurs. Un consommateur qui associe `payment_method` à une branche de traitement, avec une branche `else` qui suppose « carte », traitera mal un nouveau `"bank_transfer"` sans aucune erreur. Les producteurs peuvent documenter que les énumérations sont ouvertes. Les consommateurs peuvent traiter les valeurs inconnues comme inconnues et les orienter vers une erreur visible ou une valeur par défaut sûre, au lieu de les ranger dans un cas connu.

Schema registries et formats

Un **schema registry** est un service qui stocke les schémas utilisés par un système, leur attribue des identifiants et peut vérifier un schéma proposé par rapport aux précédents. Il fait deux métiers qu'il vaut la peine de distinguer.

1. **Consultation.** Un message porte, ou désigne, l'identifiant du schéma avec lequel il a été écrit. Un consommateur récupère ce schéma et décode les octets. C'est ce qui rend praticables les formats binaires compacts : les noms de champs ne voyagent pas avec chaque message.
2. **Contrôle d'accès.** Avant qu'un producteur n'enregistre un nouveau schéma, le registry le compare aux versions antérieures selon une règle de compatibilité choisie et le rejette s'il enfreint la règle.

C'est le second métier que je défendrais le plus fermement. Les règles qui vivent dans une checklist de relecture sont sautées le jour où l'équipe est débordée. Une vérification dans le pipeline de déploiement, elle, n'est jamais débordée.

Subjects, topics et types d'événements

Un registry vérifie la compatibilité par **subject**, et le fait qu'un subject soit un topic ou un type d'événement décide si plusieurs types d'événements peuvent partager un topic. Ce partage est ce dont a besoin l'ordre par agrégat entre types d'événements (Chapitre 9) : les `PaymentAuthorized` et `PaymentCaptured` d'un même paiement ne restent dans l'ordre que s'ils voyagent sur la même partition du même topic. Les exemples de Parcelly supposent un topic par type d'agrégat, comme `payments.payment-events`, portant plusieurs types d'événements avec l'id d'agrégat pour clé, et une compatibilité configurée par type d'événement plutôt que par topic. Le coût est que les consommateurs filtrent sur `event_type`, et qu'un topic versionné (Stratégie 3) déplace d'un coup tous les types d'événements de cet agrégat. Un topic par type d'événement est plus simple à configurer et renonce à l'ordre entre types.

Trois familles de formats

Ce livre est agnostique quant au format, il s'agit donc d'une orientation, pas d'une comparaison de produits. Avro, Protobuf et JSON Schema diffèrent par l'allure des données sur le fil, ce qui détermine ce qu'il est sûr de changer.

Famille	Sur le fil	Champs identifiés par	Conséquence pratique	Piège
Style Avro	Binaire compact, sans noms de champs	Le nom, résolu entre le schéma de l'écrivain et celui du lecteur	Le lecteur a besoin du schéma de l'écrivain ; les valeurs par défaut portent la compatibilité	Les alias permettent à un lecteur de résoudre un champ renommé, donc un renommage n'est pas toujours incompatible. Un champ absent est rempli avec la valeur par défaut du lecteur, donc l'absence est invisible
Style Protobuf	Binaire compact	Le numéro de champ	Renommer est bon marché ; réutiliser un numéro est dangereux	Bon marché seulement sur le fil binaire : ProtoJSON et le code généré utilisent les noms, donc un renommage casse le source et le JSON. Un scalaire proto3 sans présence explicite décode un champ absent comme 0
JSON Schema	JSON texte	Le nom ; schéma optionnel à l'exécution	Facile à déboguer ; la compatibilité est une convention sauf si elle est imposée	Avec un modèle fermé (`additionalProperties: false`), ajouter un champ casse les anciens lecteurs, et les registries jugent différemment les modèles ouverts et fermés

Trois idées valent pour tous.

- **Les valeurs par défaut rendent les ajouts sûrs, si la valeur par défaut est honnête.** Un nouveau champ avec une valeur par défaut définie permet à un nouveau lecteur de traiter un ancien événement. Sans elle, un ajout devient une rupture. Pour un champ dont le lecteur doit pouvoir détecter l'absence, comme un montant, la valeur par défaut doit être « pas de valeur » : une union Avro de `null` et du type avec `null` par défaut, ou `optional` en proto3, ou un type wrapper. Une valeur par défaut de 0 décode un ancien événement comme une capture de zéro, et le lecteur ne peut pas faire la différence. Le lecteur vérifie alors la présence, pas la valeur.

- **Une identité ne doit pas être réutilisée.** Qu'un champ soit identifié par son nom ou par son numéro, donner un nouveau sens à une ancienne identité est une rupture. Retirez les identifiants ; ne les recyclez pas.
- **Les champs inconnus doivent être tolérés.** Qu'un décodeur conserve, ignore ou rejette les champs qu'il ne connaît pas dépend du format et de ses réglages. Renseignez-vous avant de concevoir en fonction de cela.

Si vous utilisez du JSON sans langage de schéma, vous obtenez quand même l'essentiel du bénéfice en gardant les fichiers de schéma sous contrôle de version, en validant dans les tests du producteur et en conservant de vrais événements d'exemple pour chaque version. Le registry ajoute la consultation centralisée et l'application des règles. Il n'ajoute pas la réflexion.

Où vit la version

L'enveloppe standard porte `schema_version` sous forme d'entier à côté de `event_type`, et un consommateur lit la paire pour savoir quelle forme attendre. La garder dans l'enveloppe signifie qu'un routeur, un compteur de métriques ou un outil de DLQ peut la voir sans décoder le payload. « Combien d'événements `schema_version: 1` arrivent encore ? » devient une requête bon marché au lieu d'un projet.

Deux numéros sont faciles à confondre ici, alors ce chapitre leur donne des noms distincts. La **version incompatible** est le `schema_version` de l'enveloppe. Elle ne bouge que pour un changement qu'un ancien lecteur ne peut pas gérer : un renommage, une suppression, un changement de type ou un changement de sens. La **révision du registry** est l'identifiant que le registry attribue à chaque schéma enregistré, et elle bouge à chaque changement enregistré, additifs compris. Un consommateur rejette un événement dont il ne connaît pas la version incompatible. Il lit un événement avec une révision du registry plus récente mais une version incompatible connue comme la forme la plus récente qu'il connaît, et ignore ce qu'il ne reconnaît pas. Aucun des deux n'est la version d'agrégat du Chapitre 4, qui compte les changements d'une commande et n'a rien à voir avec la forme du payload.

Si chaque changement additif incrémentait `schema_version`, chaque consommateur pas encore mis à niveau rejetterait chaque nouvel événement, et la compatibilité vers l'avant aurait disparu. Le prix de cette séparation, ce sont deux numéros à ne pas confondre. Un exemple de cette même ligne est BabelQueue (babelqueue.com), un standard d'enveloppe open source et indépendant du langage qui fige son enveloppe à `schema_version` 1, n'autorise de nouveaux champs qu'optionnels et ignorables, exige une incrémentation de version pour un renommage, une suppression ou un changement de type, fait rejeter par les consommateurs une version qu'ils ne connaissent pas, et fait en sorte que les champs inconnus ne cassent pas le traitement et soient préservés quand le message est publié à nouveau.

Un événement tel que Payments le publie pendant l'étape de dual-publish de la migration ci-dessous :

```
{
  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"
  }
}
```

Compatibilité entre langages

Un schéma peut être parfaitement compatible selon les règles du registry et casser malgré tout entre deux langages, car les règles décrivent la structure et la rupture se situe dans la façon dont un runtime la lit. Le Chapitre 11 a fixé l'encodage de l'enveloppe. Voici les pièges dans le payload et dans ce qu'un consommateur fait des champs qu'il ne connaît pas.

Les champs inconnus sont perdus à la réémission. Supposons qu'un consommateur Node de Notifications désérialise chaque `PaymentCaptured` dans une classe dotée de `order_id`, `captured_minor` et `currency`, et copie l'événement vers un topic destiné aux partenaires. Quand Payments ajoute plus tard `payment_method`, la copie le perd en silence. Rien n'échoue. Le champ n'est tout simplement jamais là. Les décodeurs binaires Protobuf conservent en général les champs inconnus ; beaucoup de mappers JSON typés ne le font pas. Règle : un saut qui fait suivre un message préserve les champs inconnus, en transportant le payload brut ou une map générique, ou bien il publie un nouvel événement à lui sous son propre contrat.

Les nombres. `captured_minor` tient largement dans l'entier de n'importe quel langage. Un identifiant ou un compteur int64 ne tient pas dans un nombre JSON JavaScript au-delà de 2 ^ 53 - 1, et le consommateur l'arrondit sans erreur. Règle : documentez la plage de chaque champ numérique, et envoyez sous forme de chaîne tout ce qui peut la dépasser, ainsi que tout décimal.

Les horodatages. Les types Avro `timestamp-millis` et `timestamp-micros` contiennent le même genre de valeur à des échelles différentes. Un écrivain en microsecondes lu en millisecondes donne une date éloignée de dizaines de milliers d'années, qui se parse quand même. Règle : fixez l'unité dans le schéma et ne la changez jamais ; un changement d'unité est une version incompatible.

Les énumérations. Une valeur inconnue est traitée différemment selon le code généré : certains la conservent, d'autres la ramènent à une valeur par défaut, d'autres lèvent une exception. Règle : traitez toute énumération comme ouverte et décidez dans le consommateur de ce que signifie « inconnu ».

Null et absent. Beaucoup de décodeurs ne les distinguent pas, et certains comblent le vide avec une valeur par défaut. Règle : pour les champs dont l'absence compte, utilisez les formes sensibles à la présence décrites dans le point sur les valeurs par défaut plus haut.

Aucun de ces points n'exige une version précise d'une bibliothèque précise. Tous exigent qu'une personne teste avec un consommateur dans chaque langage qu'utilise Parcelly, un coût que les tests de contrat du Chapitre 15

peuvent porter.

Stratégies de versionnement

Il y a trois façons d'introduire un changement, de la moins coûteuse à la plus coûteuse. Je les essaierais dans l'ordre et ne monterais d'un échelon que lorsque l'échelon inférieur ne peut pas absorber le changement.

Stratégie 1: ajouter un champ optionnel

Si le changement est quelque chose de nouveau que les anciens lecteurs peuvent ignorer, ajoutez un champ optionnel. Ajouter `payment_method` à `PaymentCaptured` est le cas d'école. Cela enregistre une nouvelle révision du registry, et la règle de compatibilité décide si le registry l'accepte. Cela ne change pas `schema_version`, puisqu'aucun ancien lecteur ne perd quoi que ce soit. Incrémenter la version incompatible pour un changement additif ferait rejeter les nouveaux événements par chaque consommateur pas encore mis à niveau, ce qui ruine tout l'intérêt d'ajouter le champ de façon compatible.

Cette stratégie ne peut pas exprimer un renommage, un changement de type, la suppression d'un élément obligatoire ou un changement de sens. Pour ceux-là, montez d'un cran.

Stratégie 2: introduire un nouveau type d'événement

Quand le sens change, dites-le avec un nouveau nom. Supposons que Parcelly considère une capture partielle comme un fait métier différent d'une capture complète. Le modèle honnête est un nouvel événement, nommé selon les règles du Chapitre 11. Ce chapitre introduit `PaymentPartiallyCaptured` comme événement supplémentaire.

L'ancien événement garde son sens et ses consommateurs continuent de fonctionner. Le coût est que les deux événements existent désormais, et qu'un consommateur qui veut « toutes les captures » doit s'abonner aux deux. C'est le bon compromis quand le nouveau fait est conceptuellement différent, et le mauvais quand il s'agit du même fait sous une meilleure forme. Ce second cas est un problème de représentation, et un upcaster y convient mieux.

Stratégie 3: un topic versionné

Quand un changement incompatible touche tout le stream, mettez la nouvelle forme sur un nouveau topic, par exemple `payments.payment-events.v2`. Les consommateurs migrent à leur propre rythme, et l'ancien topic est retiré quand son dernier lecteur part. Avec l'organisation décrite plus haut, le nouveau topic porte tous les types d'événements de l'agrégat.

C'est l'option la plus claire et la plus coûteuse. Les clés de partition, l'ordre et les offsets des deux topics n'ont aucun rapport, donc un consommateur qui passe de l'un à l'autre doit décider où commencer à lire et comment éviter de traiter deux fois un fait. Les règles d'ordre du Chapitre 9 s'appliquent à chaque topic séparément, et rien n'ordonne les événements d'un topic à l'autre. Sur un broker à files d'attente, l'équivalent est une seconde file d'attente avec ses propres bindings, sa gestion des dead letters et ses alertes. Je n'opterais pour un topic versionné que pour une vraie rupture qu'un upcaster ne peut pas combler, et seulement quand je peux nommer qui supprimera l'ancien.

L'upcasting et le tolerant reader

Deux techniques permettent aux consommateurs de faire face à plusieurs versions sans obliger le producteur à se figer.

L'upcasting

Un **upcaster** convertit un événement écrit sous un ancien schéma vers le schéma courant avant que la logique métier ne le voie. Le code métier est écrit pour une seule forme, et la conversion de chaque version historique vit en un seul endroit.

```
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
```

Le commentaire de la branche version 1 porte le vrai travail. Un upcaster encode une affirmation sur le passé, et cette affirmation doit être vraie. Si certains anciens événements étaient en pratique partiels, l'upcaster présente discrètement un montant partiel comme la capture complète. Testez les upcasters avec de vrais événements archivés, pas avec des événements inventés.

`UnsupportedSchemaVersion` a besoin de sa propre classification. Pour le code actuel, c'est permanent, puisque faire un retry ne change rien, mais un déploiement du consommateur le résout. L'événement est donc retenu pour un redrive après la mise à niveau du consommateur (Chapitre 8), et non envoyé au parking lot, où il attendrait un humain qui n'a rien à décider.

Le côté consommateur, dans le handler avant la logique métier, est l'endroit le plus simple pour commencer. Réécrire l'historique dans un nouveau topic est utile quand de nombreux consommateurs auraient sinon besoin du même upcaster. Réécrire les événements stockés sur place est une autre décision : les systèmes event-sourced s'y refusent souvent, parce que le log fait office de preuve. Je respecterais cela sauf raison légale ou de sécurité, et je le mènerais alors comme une migration de données, avec un plan.

Le tolerant reader

Un **tolerant reader** ne prend dans un événement que ce dont il a besoin et ignore le reste. Il lit `order_id` et `captured_minor` et ne se soucie pas qu'il y ait aussi un `payment_method`. Il est libéral dans ce qu'il accepte et strict sur ce dont il dépend.

```
function readPaymentForLedger(event):
    payload = upcastPaymentCaptured(event)

    if not hasField(payload, "order_id") or
       not hasField(payload, "captured_minor") or
       not hasField(payload, "currency"):
        // a missing dependency will fail on every attempt,
        // so it is a poison message, not a retry case
        raise InvalidEvent

    return {
        order_id: payload.order_id,
        amount:   payload.captured_minor,
        currency: payload.currency
    }
```

La tolérance a deux faces. Ignorer les champs inconnus est sûr quand les producteurs ajoutent des choses. Substituer discrètement zéro à un montant manquant est dangereux. Le lecteur ci-dessus refuse de continuer quand un champ dont il dépend est absent, et il s'interroge sur la présence, pas sur la valeur. Ce n'est honnête que si le décodeur sait distinguer un champ absent d'un zéro, d'où l'importance des valeurs par défaut vues plus haut. Ce refus est une classification : l'événement ne peut jamais réussir, il prend donc le chemin des échecs permanents du Chapitre 6 et finit en dead letter, comme le décrit le Chapitre 8.

> **Remarque :** La tolérance est un accord entre les deux côtés. Le producteur promet d'ajouter et de ne pas changer le sens. Le consommateur promet d'ignorer ce qu'il ne comprend pas et de refuser ce qu'il ne peut pas faire sans risque. Si l'un des deux manque à sa part, l'autre ne peut pas sauver le système.

Le registry vérifie le producteur avant qu'il ne publie. Le tolerant reader protège le consommateur après qu'une erreur est passée au travers. Je ne compterais pas sur l'un à la place de l'autre.

Migrer un changement incompatible

Parfois, une rupture ne peut pas être évitée : l'ancienne forme est fausse, une réglementation impose de retirer un champ, ou deux services doivent être remodelés ensemble. La réponse consiste à transformer un déploiement incompatible en plusieurs déploiements compatibles. C'est le pattern **expand and contract** appliqué aux événements : élargir le contrat pour prendre en charge les deux formes, migrer chaque lecteur, puis resserrer en supprimant l'ancienne forme.

Un plan de migration pour `PaymentCaptured`

Disons que Parcelly doit vraiment renommer `amount_minor` en `captured_minor`, et que rien d'autre ne change dans l'événement. La valeur garde son sens : le montant capturé pour la commande entière. C'est un pur renommage, et le plan ci-dessous n'est sûr que pour cette raison. Les captures partielles sont un changement de sens et sont traitées à part, après les étapes.

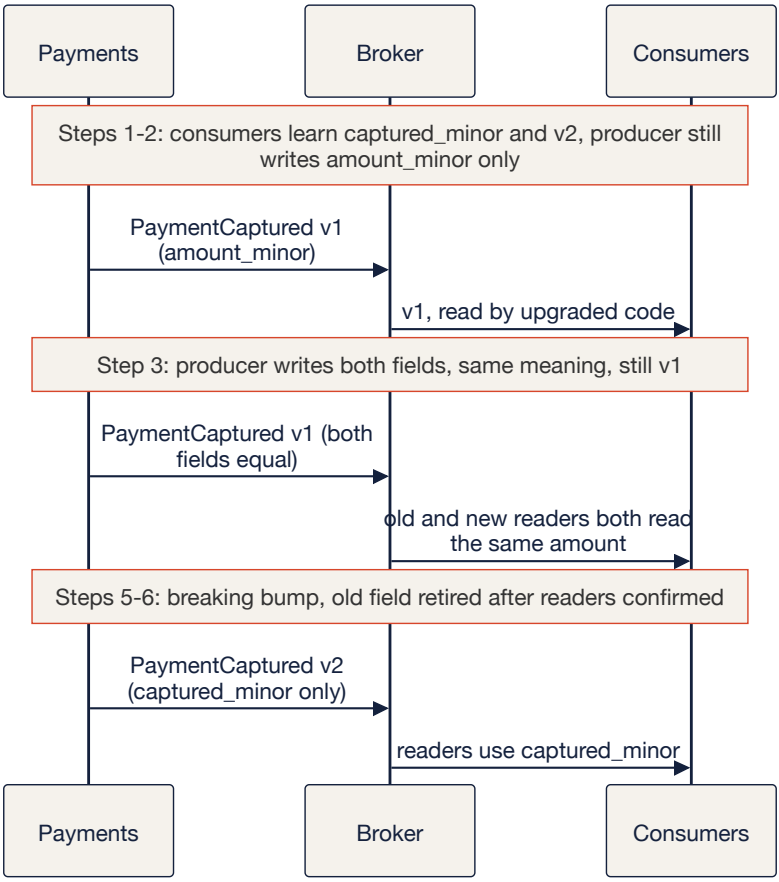
1. **Enregistrer l'ajout.** La nouvelle révision du registry conserve `amount_minor` et ajoute `captured_minor` comme champ optionnel dont la valeur par défaut est « pas de valeur », et non 0. Sous une règle de compatibilité vers l'arrière, le registry l'accepte. `schema_version` reste à 1, puisqu'aucun ancien lecteur ne perd quoi que ce soit.
2. **Mettre à niveau les consommateurs d'abord.** Chaque consommateur apprend à lire `captured_minor` et à se rabattre sur `amount_minor` quand il est absent. Il apprend aussi à lire la version incompatible 2, qui n'existe pas encore, afin que l'incrémentation de l'étape 5 ne trouve personne pour la rejeter. Les nouveaux lecteurs existent avant les nouvelles données.
3. **Faire un dual-publish.** Payments écrit les deux champs dans chaque événement, toujours en `schema_version` 1, ou, si le changement exige un nouveau type d'événement ou un nouveau topic, publie l'ancienne et la nouvelle forme du même fait. Ce n'est pas le dual write du Chapitre 4 : les deux messages proviennent de la même transaction du transactional outbox, donc la garantie habituelle tient.
4. **Observer.** Mesurez qui lit encore l'ancien champ ou l'ancien topic. Les métriques du producteur sur `schema_version` montrent ce qui est encore écrit ; l'activité des groupes de consommateurs sur l'ancien topic,

la liste des consommateurs du Chapitre 11 et les tests de contrat du Chapitre 15 montrent qui le lit encore. Un consommateur qui ne bouge jamais est une conversation à avoir, pas un chiffre à ignorer.

5. **Cesser d'écrire l'ancienne forme.** Payments ne publie plus que `captured_minor` et incrémente `schema_version` à 2, car c'est la seule étape à laquelle un ancien lecteur ne peut pas survivre. Un consommateur qui a manqué l'étape 2 rejette l'événement, et celui-ci part dans la DLQ pour un redrive après la mise à niveau de ce consommateur (Chapitre 8). L'étape 4 existe pour que cette liste reste vide. `amount_minor` reste dans le schéma comme déprécié jusqu'à ce que le plus ancien événement qui le porte soit sorti de la rétention.
6. **Resserrer.** Supprimez l'ancien champ et retirez l'ancien topic s'il y en avait un. Ne supprimez le fallback des consommateurs qu'une fois que le plus ancien événement sans `captured_minor` est sorti de la rétention, ou déplacez-le dans un upcaster permanent.

Les captures partielles restent hors de ce plan. Dès qu'une capture peut être partielle, `amount_minor` n'a plus de valeur honnête. S'il porte le montant capturé, son sens a changé sur place, et un lecteur qui a sauté l'étape 2, qu'il soit inconnu ou ramené par un rollback, enregistre le montant partiel comme s'il s'agissait du total de la commande, sans erreur. S'il porte le total de la commande, ce même lecteur enregistre trop. Le registry laisse passer les deux versions, car la structure est inchangée. C'est la leçon : une vérification structurelle de compatibilité ne peut pas voir un changement de sens. Placez le nouveau sens dans un nouveau champ, un nouveau type d'événement (Stratégie 2, `PaymentPartiallyCaptured`) ou une nouvelle version majeure sur un nouveau topic. Ensuite, ne publiez aucune capture partielle tant que chaque lecteur n'est pas passé à la nouvelle forme, et ne les publiez par la suite que sous la nouvelle forme.

L'ordre est le cœur du plan. Pour un changement compatible vers l'arrière, les consommateurs passent en premier ; pour un changement compatible vers l'avant, les producteurs peuvent passer en premier. Quand vous ne savez pas lequel s'applique, suivez le plan tel qu'il est écrit et gardez les deux champs un certain temps.



Le dual-publish entre topics

Quand le plan utilise un topic versionné, l'étape 3 publie le fait deux fois, une fois sur chaque topic. Un consommateur qui lit les deux ne doit pas traiter deux fois le même fait métier. Je donnerais aux copies des valeurs `event_id` distinctes, puisque ce sont des messages distincts, et je porterais l'id de l'original dans un champ documenté. Quel que soit votre choix, la clé d'idempotence du consommateur pour ce fait doit être définie délibérément. Le Chapitre 7 traite des clés ; ce qu'il faut retenir ici, c'est qu'une migration est précisément le moment où les hypothèses d'idempotence sont mises à l'épreuve.

> **Avertissement :** Pendant un dual-publish, ne laissez jamais un consommateur lire à la fois l'ancien et le nouveau stream et appliquer les deux. Un consommateur qui a deux sources pour un même fait a besoin d'une règle de déduplication unique, écrite noir sur blanc, faute de quoi il passera deux fois une écriture au grand livre.

Rollbacks et historique

Le rollback fait partie du plan. Si des événements en version 2 sont déjà publiés, faire un rollback du producteur ne les retire pas du log. Un consommateur ramené par un rollback à un code qui ne sait pas lire la version 2 échoue alors sur des événements qui existent déjà. Les consommateurs doivent donc savoir lire la version 2 avant qu'un producteur ne l'écrive, et la cible de rollback d'un consommateur doit savoir lire toutes les versions présentes dans la rétention, version 2 comprise. Un code qui ne lit que la version 1 est une cible de rollback à éviter, même s'il s'agissait de la dernière release avant le changement.

La compatibilité vers l'arrière gère les anciens événements sans y toucher. Si vous voulez supprimer le code de fallback, vous avez trois options : garder l'upcaster jusqu'à l'expiration des anciens événements, réécrire les événements stockés dans un nouveau topic, ou faire un snapshot de l'état ailleurs et laisser l'historique s'éteindre. Chacune coûte quelque chose : du code permanent, une migration à laquelle il faut faire confiance, ou la capacité de faire un replay. Je ne sais pas laquelle votre système devrait choisir sans savoir à quelle fréquence vous faites des replays et combien de temps vos régulateurs attendent que vous conserviez les enregistrements.

Déprécier des événements

Les événements sont plus faciles à ajouter qu'à supprimer, car un consommateur dont vous ignorez l'existence peut dépendre de n'importe lequel d'entre eux. Un cycle de vie raisonnable comporte quatre étapes.

1. **Annoncer.** Marquez l'événement comme déprécié dans le registry ou le catalogue d'événements, nommez son remplaçant, donnez une date de retrait et un contact.

2. **Mesurer.** Trouvez qui le lit encore : appartenance aux groupes de consommateurs sur un log, bindings sur une file d'attente, consultations du registry. Traitez la liste comme un indice, pas comme une preuve, car un job par lot mensuel peut être absent aujourd'hui et présent la semaine prochaine.
3. **Cesser de produire.** Après la date, arrêtez de publier.
4. **Retirer.** Supprimez le schéma et le topic ou la file d'attente une fois que la rétention a fait vieillir le dernier événement et que les lecteurs sont partis.

Deux mises en garde. Une dépréciation sans échéance ne se termine jamais, alors mettez la date par écrit et tenez-la. Et un événement déprécié dont on fait un replay après son retrait échouera, alors décidez de ce que le replay doit faire pour la période concernée avant de le retirer.

La responsabilité compte. L'équipe du producteur est responsable de l'événement et de sa dépréciation, comme l'a décrit le Chapitre 11. Les consommateurs sont responsables de leurs migrations. Une politique qui laisse le producteur décider seul, ou qui laisse les consommateurs refuser indéfiniment, échouera. La politique utile, c'est un préavis du producteur et une date tenue par les consommateurs.

Compromis et coûts

- **Un registry est un service de plus à faire tourner.** Il se trouve sur le chemin du déploiement de chaque producteur et, selon la configuration, du décodage des consommateurs. Il exige de la disponibilité, un contrôle d'accès et un responsable, et les clients ont besoin d'un plan pour quand il est injoignable.
- **Les règles de compatibilité ralentissent les producteurs, délibérément.** Une équipe qui aurait pu renommer un champ en un après-midi mène désormais une migration en six étapes. C'est le prix des déploiements indépendants.
- **Les upcasters s'accumulent.** Chaque version historique ajoute une branche et un test, et le code ne se simplifie pas tant que les anciens événements ne sont pas sortis de la rétention. Une rétention indéfinie signifie des upcasters indéfinis.

- **Les tolerant readers masquent les erreurs.** Un consommateur qui ignore les champs inconnus ne remarque pas non plus quand un producteur commence à envoyer quelque chose dont il devrait se soucier. La tolérance sans surveillance transforme des échecs bruyants en dérive silencieuse.
- **Le dual-publish multiplie le trafic et le travail de déduplication** pendant toute la durée de la migration.
- **Deux numéros de version au lieu d'un.** La version incompatible et la révision du registry empêchent les changements additifs de faire échouer les consommateurs, et coûtent à une équipe l'effort de les distinguer dans les tableaux de bord et les conversations.
- **Les numéros de version ne sont pas de la documentation.** `schema_version: 4` dit que quelque chose a changé, pas quoi ni qui est concerné. Un journal des changements tenu avec le schéma vaut plus que le numéro.
- **Les règles inter-langages coûtent en confort.** Des chaînes pour les grands nombres, la préservation du payload brut et des unités d'horodatage fixes signifient du code de conversion supplémentaire dans chaque langage, et un test par langage.

Quand ne pas utiliser ces patterns

Ne construisez pas toute la machinerie pour un événement qui a un producteur, un consommateur et une équipe qui déploie les deux ensemble. Un déploiement coordonné coûte moins cher, et le risque contre lequel vous vous assureriez n'existe pas. Revenez-y quand un second consommateur apparaît, surtout s'il appartient à une autre équipe.

Ne traitez pas le mode strict d'un registry comme un substitut à la conception. Une suite d'ajouts sûrs peut laisser un événement avec 40 champs optionnels et aucun objectif clair. Si un événement ne cesse de grossir, ce sont probablement deux événements, et les règles de granularité du Chapitre 11 s'appliquent.

Ne créez pas un topic versionné pour chaque changement. Un système qui le fait finit avec de nombreux streams à moitié retirés et personne qui sache lequel est le bon.

Erreurs courantes

- **Traiter un renommage comme un petit changement.** C'est tentant, parce que le nouveau nom est plus propre et que les tests passent. Pour chaque lecteur, c'est une suppression plus un ajout. Ajoutez le nouveau nom à côté de l'ancien, migrez les lecteurs, puis supprimez l'ancien.
- **Changer le sens d'un champ sur place.** Le nom et le type restent, donc aucun outil ne proteste. Donnez au nouveau sens un nouveau champ ou un nouveau type d'événement.
- **Ajouter un champ obligatoire sans valeur par défaut.** Cela ressemble à un ajout, mais les anciens événements ne l'ont pas. Rendez-le optionnel avec une valeur par défaut, ou introduisez-le avec une nouvelle version et un upcaster.
- **Tester la compatibilité seulement par rapport à la dernière version.** Le replay lit tout ce qui est dans la rétention. Vérifiez par rapport à chaque version encore présente, et gardez des événements d'exemple de chacune dans les tests des consommateurs (Chapitre 15).
- **Mettre à niveau les producteurs avant les consommateurs.** L'équipe du producteur est responsable du changement et elle est prête la première. Pour un changement compatible vers l'arrière, les consommateurs passent en premier. Confirmez l'ordre avant le déploiement.
- **Échouer bruyamment sur les champs inconnus.** Un parseur strict transforme chaque changement additif en incident. Ignorez ce que vous n'utilisez pas ; ne rejetez que ce dont vous dépendez.
- **Mettre en silence une valeur par défaut à un champ manquant.** Substituer zéro à un montant manquant masque un défaut, et un décodeur qui remplit 0 le masque avant même que votre code ne tourne. Utilisez une valeur par défaut « pas de valeur » pour ces champs, vérifiez la présence et levez une erreur pour que le chemin d'échec retienne l'événement pour un humain.
- **Incrémenter `schema_version` à chaque changement.** Chaque changement additif devient alors un échec franc pour chaque consommateur pas encore mis à niveau. N'incrémentez la version incompatible que pour les changements qu'un ancien lecteur ne peut pas gérer, et laissez la révision du registry compter le reste.

- **Faire suivre à travers une classe typée.** La copie perd chaque champ que la classe ne déclare pas, et rien ne s'en plaint. Préservez les champs inconnus à la réémission, ou publiez un nouvel événement sous votre propre contrat.
- **Réutiliser un numéro de champ ou un nom de topic.** Un numéro recyclé lit les anciennes données comme le nouveau champ, et un topic réutilisé mélange deux formes. Retirez les identifiants.
- **Déprécier sans date ni responsable.** L'événement s'attarde et l'équipe suivante a peur de le supprimer. Donnez à chaque dépréciation une date, un remplaçant et une personne qui répond aux questions.
- **Faire un rollback d'un consommateur sans vérifier le schéma.** Vérifiez que la cible du rollback sait lire toutes les versions présentes dans la rétention.

Points clés

- Un schéma d'événement est un contrat avec des lecteurs que vous ne contrôlez pas et avec des données écrites il y a longtemps ; la compatibilité vers l'arrière signifie que les nouveaux lecteurs peuvent lire les anciennes données, vers l'avant que les anciens lecteurs peuvent lire les nouvelles données, et un log avec replay a besoin de la compatibilité vers l'arrière par rapport à toutes les versions encore dans la rétention.
- Ajouter un champ optionnel est sûr, tandis que les renommages, les changements de type, les nouveaux champs obligatoires et les changements de sens sont incompatibles, et un changement de sens est le pire parce que rien n'échoue bruyamment ; seul un changement incompatible fait bouger `schema_version`, et la révision du registry compte le reste.
- Un schema registry mérite sa place en imposant la compatibilité dans le pipeline de déploiement, mais il vérifie la structure et non le sens, donc un humain relit toujours ce qu'un champ est censé dire.
- Préférez la stratégie de versionnement la moins coûteuse qui absorbe le changement : d'abord un champ optionnel, un nouveau type d'événement quand le sens correspond à un fait différent, et un topic versionné seulement quand tout le stream doit casser.

- Les upcasters et les tolerant readers permettent aux consommateurs de faire face à plusieurs versions, à condition que l'upcaster encode une affirmation vraie sur le passé et que le lecteur vérifie la présence, pas la valeur, et refuse ce qu'il ne peut pas interpréter sans risque.
- Entre langages, les ruptures silencieuses sont les champs inconnus perdus, les grands entiers, les unités d'horodatage, les enums fermés et la distinction entre null et absent ; préservez donc ce que vous ne connaissez pas et mettez la représentation par écrit.
- Migrez un changement incompatible comme une suite de changements compatibles, avec l'incrémentation à la seule étape à laquelle un ancien lecteur ne peut pas survivre, et planifiez les rollbacks dans le même document.

Un contrat qui évolue sans risque n'est rassurant que si vous pouvez voir ce qui circule dans le système, et c'est là que commence le Chapitre 13.

PARTIE V

Exploiter et prouver

Chapitre 13 — Observabilité et réconciliation

Imaginez un jeudi matin chez Parcelly. Le support transmet un ticket : un acheteur a payé une commande il y a deux jours, son relevé de carte montre le débit, et la page de la commande indique toujours « En attente de confirmation ». Le support ouvre l'écran d'administration d'Orders et voit `PAYMENT_PENDING`. Il interroge l'équipe Payments, qui trouve une ligne `PaymentCaptured` dans sa base de données et un outbox propre. Il interroge l'équipe Orders, dont les logs du consommateur ne montrent aucune erreur pour cette commande. Le tableau de bord de chacun est au vert.

Trois ingénieurs cherchent alors dans les logs de trois services par numéro de commande. Une ligne de log utilise `order-81724`, une autre `81724`, et le relay de Payments ne journalise que son id de ligne interne. Au bout d'une heure, ils trouvent l'événement dans l'outbox de Payments, marqué comme publié. Au bout d'une autre heure, ils apprennent que le topic du broker qui alimente Orders avait un groupe de consommateurs qui a pris du retard pendant un déploiement. La rétention du broker a supprimé une plage de messages que le groupe n'avait pas lus, et la politique de reset d'offset du groupe était réglée pour sauter à la position la plus récente, si bien qu'il a repris après le trou sans lever d'erreur. L'événement existait, a été publié et n'a jamais été lu.

Rien de tout cela n'était un bug de code au sens où un relecteur le signalerait. Le système a fait ce pour quoi il avait été construit, et personne ne pouvait le voir faire. Pire, le ticket existe parce qu'un acheteur s'en est aperçu avant toute alarme, et personne ne sait combien d'autres commandes sont dans le même cas. Ce chapitre pose deux questions. Que doit voir une équipe pour répondre en quelques minutes à « où en est cette commande ? », et comment une équipe trouve-t-elle les commandes dont personne ne s'est encore plaint ?

Le problème, et pourquoi il est difficile

Dans un système synchrone, une défaillance arrive en général avec une stack trace attachée à une requête. Dans un système orienté événements, le travail d'un même flux métier est réparti entre plusieurs services, plusieurs bases de données, un broker et un certain nombre de moments dans le temps. Aucun processus ne détient à lui seul toute l'histoire, et aucun log ne la contient à lui seul.

Trois propriétés rendent cela difficile. Premièrement, **les sauts sont découplés dans le temps**. Le producteur termine et oublie, et le consommateur peut s'exécuter des secondes ou des heures plus tard, dans un autre processus, sans rien de la pile d'appels du producteur. Les outils qui suivent une requête à travers des appels imbriqués la perdent au premier message.

Deuxièmement, **les pires défaillances sont silencieuses**. Le Chapitre 3 a décrit les messages non lus qu'un broker supprime quand un consommateur accumule un retard supérieur à la rétention. Le Chapitre 8 a décrit des messages traités de façon incorrecte sans lever d'erreur. Dans les deux cas, rien n'échoue, donc rien ne déclenche d'alerte. Un système ne peut remarquer un événement qui n'est jamais arrivé qu'en sachant qu'il aurait dû arriver.

Troisièmement, **des composants en apparence sains peuvent former un flux cassé**. Le relay tourne, le broker répond, le consommateur ne signale aucune erreur, et une commande reste bloquée. La santé d'un composant répond à « ce processus est-il vivant ? », et le métier demande « cette commande est-elle allée au bout ? ». Ce sont deux questions différentes, et c'est dans l'écart entre elles que travaille ce chapitre.

Le Chapitre 3 a décrit la sémantique d'une défaillance sous forme de trois questions : ce qui peut échouer, comment vous l'observeriez, et quelle garantie y survit. Ce chapitre porte sur la deuxième question, et sur ce qu'il faut faire quand la réponse s'avère être « vous ne l'observeriez pas ».

L'approche a deux moitiés. L'observabilité vous donne les moyens de suivre un flux et de voir la pression monter. La **réconciliation**, qui compare les visions de la vérité de deux systèmes et répare les différences, vous donne un moyen de trouver ce qu'aucun signal n'a signalé. Une

équipe qui n'a que la première moitié voit les défaillances bruyantes. Une équipe qui n'a que la seconde trouve les problèmes avec des jours de retard. Je construirais les deux.

Transporter le contexte à travers les sauts asynchrones

Chaque événement porte déjà deux identifiants issus du Chapitre 11 : `correlation_id`, partagé par tous les événements d'un même flux métier, et `causation_id`, qui désigne le déclencheur direct. Ils forment l'ossature de l'histoire. Le tracing distribué ajoute un second fil, qui enregistre la chronologie et la relation parent-enfant entre unités de travail.

Une **trace** est un arbre de spans chronométrés, chacun étant une unité de travail dans un processus. À l'intérieur d'un processus, la bibliothèque de tracing conserve le span courant dans un état ambiant. À travers un message, cet état disparaît, sauf si vous le mettez dans le message. L'approche standard consiste à sérialiser le contexte de trace, c'est-à-dire un trace id, un span id et un indicateur d'échantillonnage, dans les en-têtes du message. W3C Trace Context définit un format commun pour cela, avec deux en-têtes, `traceparent` et `tracestate`, et la plupart des bibliothèques de tracing le suivent. La capacité des en-têtes et leur propagation diffèrent selon le broker et le pont : certains limitent le nombre ou la taille des en-têtes, et certains ponts les suppriment, alors vérifiez le chemin que vos messages empruntent réellement.

L'endroit où le contexte est capturé compte plus qu'il n'y paraît. Le Chapitre 4 a mis une colonne `headers` dans l'outbox précisément pour cela. Le relay s'exécute plus tard, dans un autre processus, sans span ambiant, donc un relay qui injecterait le contexte au moment de la publication attacherait sa propre trace au lieu de celle qui a passé la commande. Le contexte doit être capturé au moment où la ligne d'outbox est écrite, dans la transaction du producteur.

```
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)
```

Côté consommateur, il y a deux façons de relier le nouveau span à l'ancien. Un span de consommateur peut être l'enfant du span du producteur, ce qui dessine un seul arbre. Ou bien il peut démarrer une nouvelle trace et conserver un **link** vers le contexte du producteur. Le bon choix dépend du cas.

Cas	Relation	Pourquoi	Ce que cela coûte
Message unique, consommé rapidement	Enfant du span du producteur	Une seule cascade de la requête à l'effet	Si la livraison est tardive, la trace reste ouverte pendant tout le délai et les durées sont trompeuses
Lot ou fan-in	Nouvelle trace, un link par producteur	Un span n'a qu'un seul parent, donc un consommateur de lot ne peut pas être l'enfant de douze producteurs	Le backend doit prendre en charge les links, et quelqu'un doit les parcourir
Fan-out	Un span de producteur, un span de consommateur enfant par abonné	L'arbre reste un arbre	La trace grossit à chaque abonné ajouté
Replay, redrive ou livraison tardive	Nouvelle trace avec un link	Un parent vieux de plusieurs heures étirerait la trace d'origine	La décision d'échantillonnage est reprise, donc une trace de producteur échantillonnée peut pointer vers une trace de consommateur non échantillonnée

Je choisirais quand même les links par défaut, parce que le retard et la nouvelle livraison sont le cas normal pour lequel ce livre planifie. Le prix, c'est la cascade unique de bout en bout que la plupart des interfaces de

tracing dessinent pour la relation parent-enfant. Pour un consommateur qui s'exécute toujours dans les instants qui suivent son producteur, la relation parent-enfant est le meilleur choix, et je la prendrais.

```
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
```

Remarquez que le span enregistre `event_id` et la tentative de livraison. Les brokers à base de log ne tiennent aucun compte des livraisons, et seuls certains brokers à base de file d'attente en exposent un, alors décidez d'où vient ce nombre avant de vous y fier. Quand un message fait l'objet d'une nouvelle livraison, vous obtenez un second span avec le même `event_id`. La paire vous dit que le message a été livré deux fois, et le champ `outcome` décrit plus bas vous dit s'il a été appliqué deux fois.

Le tracing a ses limites, et je me garderais de m'appuyer sur lui pour la garantie. Les backends échantillonnent, donc la trace de la commande qui vous intéresse peut ne pas exister. Les en-têtes de contexte peuvent être supprimés par un pont, une copie en dead letter ou un producteur écrit à la main. Une façon d'éviter de dépendre des en-têtes est de porter l'identifiant de trace dans le corps : l'enveloppe BabelQueue (babelqueue.com), un standard open source et neutre vis-à-vis du langage, porte dans le corps du message un `trace_id` que le premier producteur génère, que les producteurs suivants copient sans le modifier et qui survit à l'envoi vers la DLQ, de sorte que la piste tient même quand un saut supprime les en-têtes. Une trace qui s'arrête à une ligne d'outbox n'atteint jamais un consommateur qui n'a pas lu la ligne. C'est pourquoi les identifiants de l'enveloppe comptent plus que la bibliothèque de tracing : `correlation_id` doit apparaître dans chaque ligne de log et chaque ligne de base de données qui traite l'événement, et alors il fonctionne même quand le tracing est désactivé.



Lisez le diagramme en cherchant ce qui ne change pas : le `correlation_id` reste le même à chaque saut, tandis que chaque service stocke un contexte de trace neuf dans son propre outbox.

Reconstituer l'histoire d'une commande

Une fois les identifiants en place, la question « où en est la commande 81724 ? » devient une requête. L'implémentation la moins coûteuse est une convention plutôt qu'un produit : chaque service écrit une ligne de log structurée par événement traité et par événement publié, avec `correlation_id`, `event_id`, `event_type`, `causation_id`, le résultat et l'horodatage comme champs. Chercher `correlation_id = "order-81724"` dans ces logs vous donne chaque saut dans chaque service.

```
log {
    at: "2026-03-14T09:41:09Z", service: "orders",
    action: "consumed", outcome: "applied",
    event_type: "PaymentCaptured", event_id: "3f2a9c1e-…",
    occurred_at: "2026-03-14T09:41:07Z",
    correlation_id: "order-81724", causation_id: "b81d77f0-…",
    delivery_attempt: 1, lag_ms: 1840
}
```

Le champ `outcome` mérite qu'on s'y arrête. Utilisez un petit ensemble fermé : `applied`, `duplicate_skipped`, `stale_ignored`, `failed_retry` et `dead_lettered`. Le Chapitre 9 a soutenu qu'un événement périmé ignoré mérite une trace, et le Chapitre 7 traite de la vérification de l'inbox qui produit `duplicate_skipped`. Une fois ces résultats devenus des champs, « combien d'événements ce consommateur a-t-il écartés comme périmés au cours de la dernière heure ? » est un comptage et non une enquête.

Trier par `occurred_at` donne une chronologie, et suivre `causation_id` donne un arbre : `OrderPlaced` a causé `PaymentAuthorized`, qui avec `StockReserved` a causé `CapturePayment`, et ainsi de suite. Dans le scénario d'ouverture, l'arbre aurait montré `PaymentCaptured` publié par Payments et aucune ligne `consumed` dans Orders. Le premier saut manquant

est l'endroit où chercher. C'est plus utile que n'importe quelle ligne de log isolée : un trou dans la chaîne est la preuve que quelque chose a été perdu, alors qu'une longue chaîne montre seulement que le flux a progressé.

Si vous exploitez une saga du Chapitre 10, sa ligne `saga_instance` est une seconde source. Elle répond à « quel est l'état selon l'orchestrateur ? », et les logs répondent à « que s'est-il passé en réalité ? ». Quand les deux divergent, vous avez trouvé quelque chose qui mérite un ticket.

Je tiendrais aussi un **journal d'événements** pour les flux les plus importants : un enregistrement append-only des enveloppes d'événements, chacune avec un hash de son payload au lieu du payload lui-même, écrit par chaque service à côté de son inbox. Cela coûte du stockage et un peu de charge d'écriture, et en échange vous pouvez répondre à des questions après que le broker a oublié les événements, notamment savoir si un événement reconstruit correspond encore à l'original (voir la section sur la réparation plus bas). Gardez une rétention assez longue pour que votre réconciliation la plus lente puisse s'exécuter.

Les métriques qui comptent

Les métriques vous renseignent sur la pression avant qu'un message isolé ne tourne mal. La liste de celles qui sont utiles dans un système orienté événements est courte, et la plupart ont déjà été évoquées dans les chapitres précédents. Je commencerais par celles-ci.

Signal	Ce qu'il mesure	Pourquoi il compte
Retard du consommateur (messages)	À quelle distance de la tête se trouve le consommateur	Taille du backlog ; utile pour la capacité
Âge du message, côté consommateur	Temps écoulé depuis la production du plus ancien message non traité	Ce que vit le métier
Âge de l'outbox	Âge de la plus ancienne ligne non publiée	Santé du relay (Chapitre 5)
Latence de bout en bout	Temps entre le premier `occurred_at` du flux et l'effet dans le consommateur final	La vraie deadline du flux

Signal	Ce qu'il mesure	Pourquoi il compte
Taux de retry et nombre de tentatives	Part des messages qui nécessitent plus d'une tentative	Une dégradation masquée par les retries (Chapitre 6)
Arrivées en DLQ et âge	Nouveaux dead letters, et le plus ancien non examiné	Les défaillances qui exigent un humain (Chapitre 8)
Comptes d'ignorés et de doublons	Événements périmés écartés, doublons sautés	La pression sur l'ordre et les nouvelles livraisons
Taux de vidange et temps de rattrapage	Rythme de traitement face au rythme d'arrivée ; retard divisé par le taux de vidange net	Si un backlog se résorbe avant sa deadline ou la limite de rétention (Chapitre 6)

Le retard, et pourquoi l'âge du message est préférable

Le **retard du consommateur** en messages est le chiffre que la plupart des tableaux de bord montrent en premier, et c'est le plus facile à mal lire. Un retard de 10 000 messages ne veut pas dire grand-chose en soi : si le consommateur en vide 5 000 par seconde, il a deux secondes de retard, et s'il en vide 10 par seconde, il a un quart d'heure de retard. Le retard en messages répond à une question de capacité. Le retard en temps répond à la question métier.

L'**âge du message** est le temps écoulé depuis la production du plus ancien message encore en attente. Certains brokers à base de file d'attente exposent directement l'âge du plus ancien message ; sur les autres, déduisez-le d'un horodatage du plus ancien message en attente. Sur un broker à log, déduisez-le de l'horodatage de l'enregistrement situé à l'offset du consommateur dont le commit est fait. Dans les deux cas, le nombre est en secondes, il peut donc être comparé à une deadline choisie par le métier.

L'horodatage que vous soustrayez compte, parce que les candidats mesurent des choses différentes. Le temps d'ajout du broker mesure combien de temps un message a attendu au broker, c'est-à-dire la santé du consommateur. `occurred_at` mesure le temps écoulé depuis que le fait s'est produit, y compris le temps qu'une ligne a passé dans l'outbox à attendre le relay, c'est-à-dire la vision du délai qu'a le métier. Le temps de création du producteur dépend de la configuration : selon celle-ci, un broker à log stocke soit ce temps, soit le temps d'ajout, et un relay qui republie une ancienne ligne obtient un temps de création neuf, sauf s'il en fixe un explici-

tement. J'utiliserais le temps d'ajout pour la gauge d'âge du consommateur et `occurred_at` pour la latence de bout en bout. Utiliser `occurred_at` pour la gauge fait paraître un consommateur sain très en retard dès que le relay republie une ancienne ligne. L'âge d'un groupe est le maximum sur ses partitions, et les alertes doivent utiliser ce maximum.

Un piège mérite d'être nommé. Une partition sans nouveau trafic a un retard nul et, avec un calcul naïf, un âge nul ou un âge qui croît indéfiniment. Décidez de ce que rapporte une partition inactive, et testez-le. La même fonction doit résister à la défaillance de rétention décrite plus bas : un offset dont le commit est fait tombé avant le début du log pointe vers des données supprimées, donc cette vérification passe en premier et fait l'objet d'un signalement à part. Un consommateur bloqué sur un poison record, comme au Chapitre 8, montre un débit nul et un âge croissant, et c'est cette combinaison qui doit déclencher l'astreinte.

```
function reportConsumerAge(partition):
    committed = committedOffset(partition)
    head = headOffset(partition)
    if committed >= head:
        // nothing waiting: report zero, and report liveness
separately
        emitGauge("message_age_seconds", 0, partition)
        return
    logStart = logStartOffset(partition)
    if committed < logStart:
        // retention already deleted unread data: report how much, as
a
        // gauge that follows the condition and not how often we
checked
        emitGauge("unread_offsets_lost", logStart - committed,
partition)
        // the gauge clears at the next commit, so also leave a record
        logLostRange(partition, committed, logStart)
        committed = logStart
    record = readRecordAt(partition, committed)
    // append time: how long this record has waited at the broker
    emitGauge("message_age_seconds", now() - appendTime(record),
partition)
```

La rétention est la deadline que vous ne voyez pas

L'âge du message a une seconde fonction, moins évidente. Sur un broker à log, un consommateur dont le retard dépasse la période de rétention perd des messages en silence : le broker supprime des données qui n'ont jamais été lues, et le consommateur reprend là où sa politique de reset d'offset le

place. Sur un broker à files d'attente, la même chose se produit quand les messages expirent. Le Chapitre 3 a nommé cette défaillance, et aucune erreur ne l'accompagne.

Cette politique de reset est une décision, et chaque option a un prix. Revenir au plus ancien offset disponible fait un replay de tout ce qui reste, donc la perte est partielle et le consommateur doit absorber un afflux d'anciens messages. Sauter au dernier offset fait avancer le consommateur et rend la perte silencieuse, ce qui s'est produit dans le scénario d'ouverture. Échouer, pour que le consommateur s'arrête et attende une personne, transforme une perte silencieuse en incident. Je choisirais l'échec pour tout flux où un trou coûte de l'argent, et j'accepterais que le consommateur reste arrêté jusqu'à ce que quelqu'un décide. Le Chapitre 14 traite de la façon de se remettre du trou une fois qu'on l'a remarqué : faire un reset à un horodatage et faire un replay, avec les effets de bord supprimés.

La défense est arithmétique. Comparez l'âge du message à la période de rétention ou d'expiration, et déclenchez une alerte quand l'âge en atteint une fraction, disons la moitié. À ce stade, la deadline est encore devant, et une personne a le temps d'ajouter des consommateurs, de mettre un producteur en pause ou d'allonger la rétention. Une fois la deadline passée, la seule preuve est l'absence d'événements, et c'est précisément ce que la réconciliation sert à détecter. Je mettrais le ratio sur le même tableau de bord que l'âge lui-même, parce qu'un retard acceptable pour le métier et fatal pour la rétention est une situation réelle.

Pour un connecteur CDC, le signal équivalent est la part du log de la base de données que le connecteur n'a pas encore consommée, comparée à ce que la base de données accepte de conserver pour lui. Un connecteur arrêté plus longtemps que la durée pendant laquelle la base de données conserve son log peut avoir besoin d'un nouveau snapshot pour récupérer, et c'est une panne à part entière.

Latence, retries et le reste

La **latence de bout en bout** a besoin d'une définition avant d'avoir besoin d'un graphique. Je mesurerais depuis l'`occurred_at` du premier événement d'un flux jusqu'au moment où le commit de l'effet final est fait, par exemple de `OrderPlaced` à `OrderConfirmed`. Rapportez des percentiles, pas une moyenne : la queue lente, ce sont les acheteurs qui écrivent au

support. Le consommateur final ne voit que l'enveloppe qu'il a consommée, et cette enveloppe porte l'heure de son propre fait, pas le début du flux. Propagez l'heure de début du flux dans un en-tête, ou retrouvez-la par `correlation_id` dans le journal d'événements ou dans le service propriétaire. Étiquetez ensuite la mesure avec le `event_type` final. Les deux horodatages proviennent des horloges de services différents, attendez-vous donc à un certain décalage d'horloge dans le résultat ; il est faible face à une deadline métier, et c'est une raison de ne pas alerter sur des différences inférieures à la seconde.

Le **taux de vidange** est la forme observable du backpressure. Divisez le retard par la différence entre le rythme auquel le consommateur traite les messages et le rythme auquel ils arrivent, et vous obtenez le temps de rattrapage. Si les arrivées dépassent le traitement, le retard ne se résorbe jamais, et c'est le chiffre sur lequel alerter, bien avant que l'âge n'atteigne la limite de rétention. Le Chapitre 3 a défini le backpressure et le Chapitre 6 traite de la façon de borner le travail qu'un consommateur accepte.

Le **taux de retry et le nombre de tentatives** font apparaître le problème que le Chapitre 6 a appelé les défaillances cachées : une dépendance dégradée depuis des semaines pendant que les retries l'absorbent discrètement. L'histogramme du nombre de tentatives est plus informatif que le nombre d'erreurs, car un message qui réussit toujours à la troisième tentative ne produit aucune erreur.

Les **arrivées en DLQ** et l'âge du plus ancien dead letter non examiné viennent du Chapitre 8. Ajoutez les **comptes de doublons et d'événements périmés** des chapitres 7 et 9. Un compte de doublons qui augmente juste après un déploiement est normal, puisque l'attribution du travail se déplace, et un compte qui augmente à une heure stable désigne des handlers lents. Sur un broker à files d'attente, la cause habituelle est un visibility timeout plus court que la durée d'exécution du handler. Sur un broker à log, c'est un rééquilibrage, ou un consommateur qui dépasse son timeout de polling et perd ses partitions.

Des alertes qui ont un sens

L'échec de la plupart des dispositifs d'alerte n'est pas un manque de données. C'est que les données déclenchent l'astreinte pour des causes plutôt que pour des effets, et qu'après quelques semaines de bruit, plus personne ne fait confiance aux appels d'astreinte. Je tiendrais les alertes à une seule exigence : si celle-ci se déclenche à trois heures du matin, une personne nommée peut dire quoi faire.

Alertez sur des **symptômes liés à une deadline métier**. « La plus ancienne ligne d'outbox non publiée a plus de deux minutes » est un symptôme : quelque chose est bloqué entre un changement d'état et son annonce. « Le CPU du consommateur dépasse 80 % » est une cause, et cela peut compter ou non. Utilisez les métriques de cause dans les tableaux de bord où quelqu'un enquête, et réservez l'astreinte aux symptômes.

Donnez à chaque alerte **un responsable et un seuil assorti d'une raison**. Le Chapitre 8 a fait la même remarque pour les DLQ : un e-mail perdu et une capture perdue méritent des seuils différents, et la différence doit être écrite à côté de l'alarme. La raison compte parce que les seuils dérivent, et qu'un chiffre sans raison énoncée est relevé la première fois qu'il dérange.

Alertez sur **l'absence autant que sur la présence**. Un relay qui plante n'émet aucune erreur ; il n'émet rien. Un groupe de consommateurs qui a perdu tous ses membres montre du retard, mais un topic qui a complètement cessé de recevoir des événements ne montre ni retard ni erreur. Pour les flux au rythme régulier, comme les commandes pendant les heures ouvrées, une alerte telle que « aucun événement `OrderPlaced` observé depuis 10 minutes » attrape un silence côté producteur que tous les autres signaux manquent. Le seuil doit suivre le profil du trafic, et pour les flux sans rythme, c'est le mauvais outil.

Enfin, deux niveaux suffisent à la plupart des équipes : un appel d'astreinte pour tout ce qui menace une deadline ou perd des données, et un ticket pour la pression lente, comme une DLQ de messages de faible valeur qui grossit. Un troisième niveau a tendance à être ignoré.

> **Avertissement :** Relever un seuil d'alerte pour faire taire le bruit, c'est ainsi qu'une équipe échange une gêne contre un angle mort. Si une alerte se déclenche trop souvent, corrigez sa cause ou sa définition. Ne relevez pas le chiffre jusqu'à ce qu'elle cesse de se déclencher.

La réconciliation

Les métriques et les traces observent ce que fait le système. Elles ne peuvent rien vous dire d'un événement qui n'est jamais entré dans le système, d'un handler qui a fait le commit d'un état erroné sans erreur, ou d'une réparation que quelqu'un a appliquée à la main. Pour ces cas, vous comparez le système à un enregistrement indépendant de ce qui devrait être vrai. C'est la réconciliation.

Un job de réconciliation prend deux visions des mêmes faits et rapporte les différences. Les paires sont en général évidentes dès qu'on les cherche :

- **Orders face à Payments :** chaque commande à l'état `PAID` a un paiement capturé, et chaque paiement capturé appartient à une commande qui en a connaissance.
- **Payments face à Ledger :** chaque `PaymentCaptured` a un `LedgerEntryPosted` correspondant pour le même montant dans la même devise.
- **Vos enregistrements face à ceux d'un fournisseur :** le rapport de règlement du fournisseur de paiement face à vos propres enregistrements de capture. Le Chapitre 7 renvoyait ici pour les effets qui ne peuvent pas être rendus idempotents, là où la réconciliation est la dernière ligne de défense.
- **Un modèle de lecture face à sa source :** un index de recherche ou une vue dont le Chapitre 9 disait qu'ils pouvaient être marqués comme suspects et réparés plus tard.

Trois types de différences couvrent la plupart des cas : **manquant** (la source l'a, le côté dérivé non), **périmé** (les deux l'ont mais le côté dérivé diverge, en général parce qu'il est en retard) et **en trop** (le côté dérivé a quelque chose que la source ne connaît pas). Chacun pointe vers une cause différente. Manquant suggère des événements perdus ou jamais

consommés, périmé suggère un problème d'ordre ou de retard, ou de temps à autre un handler qui a écrit une mauvaise valeur, et en trop suggère un bug ou une modification manuelle.

Le point qui demande du soin, c'est que les événements en transit ressemblent exactement à des écarts. Un paiement capturé il y a quatre secondes et pas encore dans Ledger, c'est normal. Un job qui le signale produira du bruit et apprendra aux gens à l'ignorer. Le job ne compare donc que des enregistrements plus anciens qu'un **délai de grâce** choisi d'après votre latence de bout en bout normale, et il confirme un écart une seconde fois avant de le signaler.

```
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)
```

Plusieurs détails de ce bloc portent l'exactitude. La fenêtre est bornée et la limite est appliquée, donc le job lit des données qui ont eu le temps de se stabiliser. Chaque lot est une recherche par clé plutôt qu'une jointure entre bases de données, parce que les services de Parcelly ne lisent pas les tables les uns des autres : le job récupère donc l'autre côté via l'interface propre à ce service ou un extrait publié. Le job ne fait qu'enregistrer les écarts et ne modifie rien. Et comme il fait un upsert dans une table d'écarts ayant pour clé l'élément et le type, l'exécuter deux fois, ou après un crash au milieu d'une fenêtre, produit la même table, sans pile croissante de lignes en double. La même fonction clôt les écarts une fois que les deux côtés concordent, et c'est l'étape sur laquelle s'appuie le code de réparation ci-dessous. Un paiement enregistré deux fois dans Ledger est repéré par la passe aller, qui compte les correspondances. C'est la passe retour qui trouve une entrée « en trop » sans capture derrière elle, comme une écriture pour un paiement qui n'a jamais été capturé, ce qu'une vérification à sens unique ne trouverait jamais.

Choisissez la cadence selon les dégâts que cause un écart. Payments face à Ledger est financier et mérite une exécution fréquente, avec une comparaison complète quotidienne comme filet de sécurité. Un index de recherche peut se contenter d'une exécution nocturne. La santé du job lui-même compte aussi : déclenchez une alerte s'il ne s'est pas terminé dans la fenêtre attendue, car une réconciliation qui s'est arrêtée en silence est pire que pas de réconciliation du tout, puisque les gens croient qu'elle surveille.

> **Remarque :** Un job de réconciliation est un consommateur des données qu'il vérifie, et il peut se tromper. Traitez d'abord un écart comme une piste d'enquête. Les premières semaines d'un nouveau job se passent surtout à trouver des faux positifs dus au timing, aux arrondis ou à une règle de comparaison qui ne correspond pas au métier.

Réparation et backfill

Trouver un écart, c'est la moitié du travail. L'autre moitié consiste à modifier l'état pour qu'il concorde, sans créer un second écart.

La règle à laquelle je m'en tiendrais est que **les réparations passent par le même chemin que les changements normaux**. S'il manque à Ledger une entrée pour un paiement capturé, n'insérez pas la ligne avec une console de base de données. Le Chapitre 3 a traité les actions de l'opérateur comme faisant partie du modèle de défaillance, et une ligne modifiée à la main n'émet aucun événement, donc tous les autres consommateurs de ce fait restent dans l'erreur, et la réconciliation suivante signale une incohérence que vous avez vous-même créée. Republiez plutôt le fait, ou envoyez le command qui le provoque, via l'outbox du producteur, afin que les consommateurs le traitent comme tout le reste.

Une réparation peut faire l'objet d'un retry, être interrompue ou être lancée deux fois par deux ingénieurs, et elle doit rester sûre dans chaque cas. L'idempotence du Chapitre 7 n'y répond que si l'événement de réparation porte l'`event_id` **d'origine**. Un événement de réparation avec un nouvel id est un événement différent pour chaque consommateur. Si le premier événement était seulement retardé, il arrive après la réparation, et Ledger enregistre le paiement deux fois. La réparation republie donc le fait qui existe déjà : la même enveloppe, avec les mêmes `event_id` et `causation_id`, et l'inbox traite comme doublon la copie qui arrive en second, quelle qu'elle soit.

Cela exige que le producteur conserve l'id d'origine. Le plus simple est de stocker `event_id` sur l'enregistrement métier (ici, la capture) au moment où la ligne d'outbox est écrite, avec un hash du payload. Si la ligne d'outbox existe encore, elle est marquée comme non publiée et le relay l'envoie à nouveau. Si elle a été purgée, l'enveloppe est reconstruite à partir de l'enregistrement avec l'id stocké.

La reconstruction a une limite. L'enregistrement est l'état courant, et l'événement est un fait du passé. Si l'enregistrement a changé entre-temps, par exemple parce qu'un remboursement partiel a ajusté le montant, un payload reconstruit à partir de lui diffère de l'original tout en portant le même `event_id`. C'est pourquoi le hash est stocké : la réparation reconstruit le payload, compare son hash avec celui qui est stocké, et s'arrête pour un humain s'ils diffèrent. Un enregistrement sans id ni hash stocké, en général écrit avant que l'habitude n'existe, doit se rabattre sur un id dérivé de sa clé métier et sur des champs qui ne peuvent pas changer, et le consom-

mateur doit alors aussi être protégé par une contrainte d'unicité sur cette clé, comme le décrit le Chapitre 7. J'ajouterais de toute façon cette contrainte à Ledger, comme seconde ligne de défense.

Une republication fait aussi du **fan-out**. Le topic livre l'enveloppe à chaque abonné, pas seulement à Ledger. Un abonné dont l'inbox contient encore l'`event_id` la saute. Un abonné dont la ligne d'inbox a été purgée, parce que l'événement est plus ancien que sa fenêtre de déduplication (Chapitre 7), l'applique une seconde fois, et Notifications pourrait envoyer à nouveau le même e-mail. Il y a donc deux chemins sûrs. Si l'événement est plus récent que la plus courte fenêtre de déduplication parmi tous les abonnés, republiez-le via l'outbox du producteur comme ci-dessus. S'il est plus ancien, ne livrez la réparation qu'au consommateur qui l'a manqué, via la file d'attente ou le topic de réparation propre à ce consommateur, ou en faisant un reset d'offset de ce seul groupe, et comptez sur une contrainte de clé métier comme protection. Le Chapitre 8 applique le même principe au 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
```

Deux détails portent la sécurité. Marquer à nouveau comme non publiée une ligne non publiée est sans danger quand on le répète, et la violation d'unicité due à une seconde insertion concurrente est traitée comme « déjà pris en charge » plutôt que comme une erreur. L'enveloppe republiée garde aussi son `causation_id` d'origine, ainsi la chaîne de causalité reste véridique, et le lien vers l'écart vit dans l'enregistrement de réparation. Le dernier commentaire est le plus important. La fonction de réparation ne marque pas l'écart comme résolu. C'est la réconciliation suivante qui le fait, quand les deux côtés concordent. Cela évite qu'une réparation publiée puis perdue soit signalée comme corrigée.

La réparation en masse, qu'on appelle en général **backfill**, ajoute l'échelle. Un backfill de 30 000 événements vers un consommateur en production entre en concurrence avec le trafic normal et peut pousser la latence du

flux au-delà de sa deadline. Exécutez-le par petits lots, avec un rate limit et un interrupteur de pause, comme le Chapitre 8 le recommandait pour le redrive, et surveillez l'âge du message et le retard du consommateur pendant qu'il tourne. Sur un broker à log, vous préférerez peut-être un replay délibéré vers un groupe de consommateurs séparé qui reconstruit une vue dérivée, puis bascule. C'est plus propre quand le consommateur est une projection, et c'est une erreur quand le consommateur a des effets de bord comme l'envoi d'e-mails.

La réparation via le producteur convient à une poignée d'enregistrements. Quand les dégâts sont plus étendus, comme un groupe de consommateurs qui a sauté tout un trou de rétention, une base de données restaurée ou un cluster de brokers perdu, republier enregistrement par enregistrement cesse d'être le bon outil. Le Chapitre 14 traite de la reprise par replay, des sinistres et de la migration de broker.

Chaque réparation a besoin d'un enregistrement : qui l'a lancée, quels écarts elle couvrait, ce qu'elle a publié et quand. La piste d'audit est ce qui permet à l'ingénieur suivant d'apprendre que le pic de doublons à 14:10 était un backfill, et c'est ainsi qu'un régulateur ou l'équipe finance apprend plus tard qu'une entrée de Ledger était une correction.

Les invariants métier

La réconciliation compare deux systèmes. Une **vérification d'invariant** compare un système à une règle que le métier tient pour vraie. Les invariants sont souvent moins coûteux à évaluer et attrapent des problèmes qu'aucun second système ne pourrait attraper, comme un bug qui fait que deux systèmes s'accordent sur la mauvaise réponse.

Pour Parcelly, quelques exemples montrent la forme :

- Aucune commande n'est `CONFIRMED` sans paiement capturé.
- Aucune réservation ne reste bloquée au-delà de sa deadline.
- Aucune instance de saga ne reste en `AWAITING_RESULTS` après son `deadline_at`.
- Chaque paiement capturé a exactement une entrée Ledger, ni zéro ni deux.
- La somme des entrées Ledger d'une commande s'équilibre à zéro.

Chacune est une requête qui ne doit renvoyer aucune ligne. Exécutez-la selon un planning, et traitez chaque ligne renvoyée comme un écart doté d'un responsable. L'exemple de la saga est celui que le Chapitre 10 avait laissé ouvert : le handler de timeout est la première protection contre une commande figée, et une requête d'invariant sur les instances bloquées est la seconde, parce que le handler qui aurait dû se déclencher peut lui-même avoir échoué.

```
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)
```

Prendre pour clé l'invariant et l'élément signifie qu'une violation qui persiste est mise à jour et non dupliquée, et `first_seen_at` vous dit depuis combien de temps un problème est ouvert. Cet âge est lui-même une métrique sur laquelle il vaut la peine d'alerter.

J'écrirais les invariants avec les gens du produit et de la finance plutôt que pour eux. Les ingénieurs savent ce que le système peut faire, et ils ne savent pas toujours ce que le métier considère comme un défaut. Une règle comme « un remboursement ne dépasse jamais la capture » est évidente pour la finance et peut n'être écrite nulle part dans le code.

Compromis et coûts

L'observabilité et la réconciliation ne sont pas gratuites, et elles ne fonctionnent pas toutes seules.

La télémétrie a un prix. Des logs structurés pour chaque événement traité multiplient le volume de logs, et des labels à forte cardinalité comme `order_id` sur les métriques peuvent submerger un backend de métriques. Gardez les identifiants dans les logs et les traces, et limitez les labels de métriques à de petits ensembles comme le service, le topic et le type d'événement. Échantillonnez les traces, et acceptez que la trace dont vous avez besoin puisse ne pas exister.

Un journal d'événements duplique des données. Il en vaut la peine pour les flux où « que s'est-il passé ? » a un poids juridique ou financier. Pour un flux de faible valeur comme une notification cosmétique, les logs suffisent, et le journal est un coût sans lecteur.

La réconciliation lit des données de production. Un job de comparaison peut charger la base de données source, et un job négligent sur le découpage en lots peut ralentir le service qu'il protège. Exécutez-le sur une réplique quand vous en avez une, et bornez la taille de ses lots et son rythme. Il ajoute aussi une dépendance : si l'interface de l'autre côté est indisponible, le job ne peut pas s'exécuter.

Les alertes coûtent de l'attention. Chaque appel d'astreinte entame la crédibilité du suivant. Un petit ensemble d'alertes ayant chacune un vrai responsable fait plus qu'un grand ensemble que personne ne lit.

La réparation comporte son propre risque. Un mécanisme de réparation est un chemin pour modifier l'état de production en dehors du flux normal des requêtes. Sans audit, sans approbations pour les types dangereux et sans conception idempotente, il remplace une classe de problèmes par une pire.

Quand ne pas construire tout cela : un système avec un service et un consommateur, des enjeux faibles et des flux courts n'a besoin ni d'un journal d'événements, ni de tables d'invariants, ni d'une réconciliation nocturne. Des logs structurés avec `correlation_id` et une alerte sur l'âge du message sont le minimum que je maintiendrais à n'importe quelle échelle. Le reste grandit avec le coût d'une défaillance silencieuse. Si un événement perdu coûte un écart dans le grand livre, construisez la réconciliation en premier. S'il coûte un résultat de recherche périmé, construisez-la en dernier.

Erreurs courantes

- **Alerter sur la santé, pas sur le flux.** Des graphiques de CPU et de mémoire au vert disent que les processus sont vivants et ne disent rien de l'aboutissement des commandes. C'est tentant, parce que les métriques d'infrastructure existent déjà. Ajoutez les alertes sur les symptômes : âge du message, âge de l'outbox et violations d'invariants.

- **Ne surveiller le retard qu'en messages.** Un nombre de messages n'est associé à aucune deadline. Convertissez-le en âge, et comparez l'âge à la deadline métier et à la période de rétention.
- **Injecter le contexte de trace dans le relay.** Le relay n'a pas de span ambiant, il attache donc le mauvais, ou aucun. Capturez le contexte quand la ligne d'outbox est écrite et transmettez-le sans le modifier.
- **Des identifiants incohérents.** Un service journalise `order-81724`, un autre `81724`. La recherche par corrélation échoue alors précisément quand on en a besoin. Définissez le format une fois dans la conception des événements et imposez-le avec un helper de logging partagé. Gardez ce même format pour le trafic synthétique : le Chapitre 15 marque les sondes avec un en-tête séparé et jamais avec un `correlation_id` spécial.
- **Utiliser un seul horodatage pour tous les âges.** Le temps de création du producteur, le temps d'ajout du broker et `occurred_at` mesurent des délais différents. Utilisez le temps d'ajout pour la gauge d'âge du consommateur et `occurred_at` pour la latence de bout en bout.
- **Republier une réparation vers tous les abonnés.** Un consommateur dont la ligne d'inbox a été purgée applique à nouveau l'événement. Comparez l'événement à la plus courte fenêtre de déduplication, ou ne livrez la réparation qu'au seul consommateur qui l'a manqué.
- **Réconcilier sans délai de grâce.** Le job signale chaque événement en transit comme un écart, l'équipe apprend à l'ignorer, et la seule vraie découverte se perd dans le bruit. Ne comparez que des enregistrements stabilisés et revérifiez avant de signaler.
- **Réparer en modifiant des lignes à la main.** Cela corrige une table, n'émet aucun événement et laisse dans l'erreur chaque consommateur en aval. Réparez via le producteur et l'outbox, pour que le changement circule comme n'importe quel autre.
- **Une réparation qui n'est pas idempotente.** Une seconde exécution duplique la correction. Republiez avec l'`event_id` d'origine, afin que l'inbox reconnaisse aussi bien une seconde exécution qu'un original tardif.
- **Laisser la réconciliation se dégrader.** Le job casse après un changement de schéma, personne ne le remarque, et l'équipe se croit cou-

verte. Déclenchez une alerte quand une exécution est en retard ou échoue, et passez en revue le backlog d'écarts selon un planning.

Points clés

- Un flux réparti entre plusieurs services a besoin d'identifiants qui survivent à chaque saut : `correlation_id` dans chaque ligne de log et chaque ligne de base de données, et un contexte de trace stocké dans la ligne d'outbox pour que le relay transmette le bon.
- Le tracing aide, mais il est échantillonné et peut être perdu, et le choix entre links et parent-enfant est un compromis au cas par cas ; les identifiants de l'enveloppe doivent donc suffire à eux seuls pour reconstituer l'histoire d'une commande.
- L'âge du message, mesuré par rapport à une deadline métier et à la rétention du broker, est plus utile que le retard en messages, et c'est le signal qui vous avertit avant qu'un consommateur en retard ne perde des événements en silence ; la politique de reset d'offset décide si une telle perte est silencieuse ou bruyante.
- Alertez sur des symptômes liés à des deadlines, donnez à chaque alerte un responsable et une raison énoncée pour son seuil, et prévoyez des alertes sur le silence, car les défaillances qui perdent des données ne produisent souvent aucun signal.
- La réconciliation trouve ce qu'aucune métrique ne signale : elle compare des enregistrements stabilisés entre deux visions de la vérité, classe les différences en manquantes, périmées ou en trop, et les enregistre sans rien modifier.
- Les réparations doivent emprunter le chemin normal, en republiant l'`event_id` d'origine pour qu'un original tardif et une réparation ne puissent pas prendre effet tous les deux, avec une piste d'audit, et un écart est clos par la comparaison réussie suivante, non par la réparation elle-même. Une republication atteint chaque abonné, donc les anciens événements ne vont qu'au consommateur qui les a manqués.
- Les vérifications d'invariants métier, comme l'absence de commande confirmée sans paiement capturé, attrapent des défauts que deux systèmes d'accord entre eux masqueraient.

L'observation et la réconciliation vous disent que le système se comporte mal, et la réparation corrige une poignée d'enregistrements. Le Chapitre 14 traite de ce qu'il faut faire quand les dégâts dépassent ce qu'une réparation peut atteindre : reprendre par replay, survivre à un sinistre et passer à un autre broker.

Chapitre 14 — Reprise, sinistres et migration de brokers

Imaginez un mardi chez Parcelly. Le cluster de broker managé qui transporte chaque événement de la marketplace devient injoignable. La page de statut du fournisseur indique que la zone qui l'héberge est défaillante, et qu'un cluster de remplacement sera disponible dans l'heure. Quand il l'est, les topics existent et sont vides, ou bien ils contiennent des données jusqu'à un point situé quelque part dans les dernières minutes. Personne ne peut dire exactement où se trouve ce point.

Les services vont bien. Orders, Payments, Inventory, Shipping et Ledger ont chacun leur propre base de données, et chacune est intacte. Chaque service a un relay qui retente sans relâche sa publication, et les outboxes se remplissent. Les consommateurs sont inactifs, en attente d'un broker qui n'est pas là. Pendant ce temps, l'équipe doit décider de ce qu'elle fera à son retour, et les options semblent toutes plausibles. Pointer les consommateurs vers le nouveau cluster et les laisser tourner. Tout republier depuis les outboxes. Restaurer la dernière sauvegarde de quelque chose. Chaque option est juste pour un stockage et fausse pour un autre.

Ce chapitre traite du lendemain du jour où un stockage tourne mal, disparaît ou est remplacé volontairement. Il pose une seule question. Quand le broker, la base de données ou la région n'est plus ce qu'il était, lesquelles des garanties que vous avez construites tiennent encore debout, et comment vous en servir pour revenir à un état auquel vous pouvez vous fier ?

Le problème, et pourquoi il est difficile

Les chapitres précédents se défendaient contre des défaillances qui laissent les stockages intacts : un crash entre deux étapes, un consommateur lent, un doublon, un poison message. Une reprise est différente, parce que ce qui a mal tourné, c'est la mémoire elle-même. Un stockage est reve-

nu en arrière dans le temps, ou a perdu une partie de son historique, ou se fait remplacer par un autre. Chaque défense de ce livre repose sur un stockage qui se souvient de quelque chose, si bien que la question devient : quelle mémoire a survécu, et laquelle non ?

C'est difficile pour deux raisons. D'abord, les stockages tombent en panne indépendamment et dans des directions différentes. Une base de données restaurée à il y a six heures et un broker qui a perdu ses deux dernières minutes sont en désaccord d'une manière qu'aucun des deux ne peut détecter. Chacun est cohérent avec lui-même. Le Chapitre 3 appelait cela perdre un stockage entier, et notait que rien ne lève d'erreur au début.

Ensuite, le travail de reprise est rare, et le travail rare se fait mal sous pression. Les opérations qui font reculer un groupe de consommateurs, les réglages qui empêchent un replay d'envoyer une deuxième vague d'e-mails, l'ordre dans lequel les services redémarrent : ce sont autant de décisions peu coûteuses à prendre à l'avance et coûteuses à improviser.

La thèse de ce chapitre est la thèse du livre dans sa forme la plus nue. Le broker n'est pas ce qui survit. Ce qui survit à un cluster perdu, à une base de données restaurée ou à un changement de produit, c'est la sémantique des défaillances que vous avez conçue autour de lui : le transactional outbox (Chapitre 4) qui se souvient de ce qui n'a jamais été confirmé, l'inbox qui reconnaît ce qui a déjà été fait, l'identifiant stable, le contrat et l'habitude de comparer deux vues de la vérité. Si ces éléments sont en place, le broker est remplaçable. S'ils ne le sont pas, c'est le broker qui portait la justesse du système, et personne ne le savait.

Reprendre par le replay

Le **replay**, défini au Chapitre 2, est la relecture délibérée de l'historique depuis un log d'événements. Il ne s'applique qu'à la famille des logs, car une file d'attente supprime ce qu'elle livre. Le Chapitre 15 utilise le replay comme technique de test. Ici, c'est un outil de reprise, et il a une procédure.

La situation à laquelle il convient est celle qui ouvrait le Chapitre 13 : un groupe de consommateurs a pris du retard, la rétention a supprimé des messages qu'il n'avait pas lus, et le groupe a poursuivi au-delà d'un trou. Si

les données sont encore retenues, la première reprise à envisager n'est pas de reproduire les événements un par un depuis le producteur. C'est de ramener la position du groupe avant le trou et de relire.

Réinitialiser un groupe à un instant donné

La position d'un groupe de consommateurs est stockée sous forme d'un offset par partition. La faire reculer est une action d'administration sur le broker, et la plupart des brokers à base de log peuvent cibler soit un offset explicite, soit un horodatage, que le broker résout en premier offset situé à cet instant ou après. Je préfère un horodatage. Un offset est un nombre qui a un sens pour une partition d'un topic sur un cluster, et l'horodatage est ce que vous savez : le déploiement a commencé à 09:40, donc revenez à 09:30. Consultez la documentation de votre broker pour savoir comment il résout un horodatage, car l'heure du producteur et l'heure d'ajout du broker ne sont pas toujours celle qu'il utilise.

La procédure est courte et l'ordre compte.

1. Arrêtez le groupe. Réinitialiser les offsets d'un groupe en cours d'exécution échoue ou se fait écraser par le commit suivant, selon le broker.
2. Notez les offsets actuels avant de changer quoi que ce soit. C'est le seul chemin de retour si le reset était erroné.
3. Choisissez la cible avec une marge. Remontez plus tôt que ce qui vous semble nécessaire, car les doublons coûtent peu et un trou, non.
4. Réinitialisez à la cible, redémarrez le groupe et surveillez l'âge du plus ancien message non traité (Chapitre 13) pendant qu'il se vide.
5. Continuez à surveiller jusqu'à ce que le groupe soit revenu au head, puis réconciliez, car le reset ne répare que ce que le log contenait encore.

Choisir une marge n'est sûr que grâce à l'inbox (Chapitre 7), qui transforme la relecture des messages que le groupe a bel et bien traités en no-ops. Un groupe dont les consommateurs ne sont pas idempotents ne devrait pas être réinitialisé, quoi que dise la procédure.

Limiter le replay à un seul groupe de consommateurs

Un log permet à 10 groupes de lire le même topic, et chacun a ses propres offsets. C'est la propriété qui permet de cibler le reset sans risque. Réinitialisez le groupe qui a manqué les événements, et aucun autre groupe ne s'en aperçoit. Réinitialisez le plus petit ensemble de groupes qui ont manqué quelque chose, et nommez-les dans le plan avant de l'exécuter. Le consommateur de `OrderPlaced` de Payments garde sa position pendant que le groupe d'Inventory relit la dernière journée.

Une seconde forme de la même discipline est un groupe séparé pour le replay. Si une projection doit être reconstruite, par exemple la vue de recherche d'Orders, démarrez un nouveau groupe sous un autre nom qui lit depuis le début vers une table neuve, et basculez le lecteur quand il a rattrapé son retard. L'ancien groupe ne bouge jamais, il n'y a donc rien à défaire si la reconstruction tourne mal. Cela coûte du stockage et un second consommateur pendant un temps, ce qui est un petit prix pour un rollback gratuit.

Neutraliser les effets de bord

Un replay applique le code d'aujourd'hui aux faits d'hier. Si un handler envoie des e-mails, appelle le fournisseur de paiement ou notifie un transporteur, le replay lui fait recommencer. Le Chapitre 12 a soulevé ce point, et la règle qu'il donnait est le point de départ : un consommateur reconstruit par replay doit savoir qu'il est en train de faire un replay.

La première idée qui vient est un drapeau dans le message, un en-tête qui dit « ceci est un replay ». Cela ne fonctionne pas, car un message dont on fait un replay est le message original. Le broker contient les octets que le producteur a écrits, et rien ne les marque comme lus pour la deuxième fois. Le mode appartient au consommateur, pas au message. Le groupe qui fait un replay tourne comme un déploiement séparé avec une valeur de configuration telle que `mode: replay`, ou bien la couche d'effets vérifie le nom du groupe.

Une fois le mode connu, il existe quatre options pour les effets eux-mêmes, et chacune a un prix.

- **Placer un gate sur les effets en bordure.** Chaque appel externe passe par une seule fonction qui vérifie le mode et, en replay, ne fait rien.

C'est la conception la plus claire et la moins coûteuse à auditer, mais elle ne fonctionne que si aucun handler ne contourne le gate. Un nouveau handler qui appelle directement un fournisseur la met en échec, la règle a donc besoin d'un test (Chapitre 15).

- **Mettre des clés d'idempotence sur les appels externes.** Si l'appel de capture porte une clé dérivée de `event_id`, le fournisseur reconnaît la seconde tentative. Le Chapitre 7 a traité ce point. Cela ne protège que tant que le fournisseur se souvient de la clé ; c'est donc une seconde ligne de défense, et non celle sur laquelle compter pour un replay du mois dernier.
- **Suspendre l'outbox des événements dérivés.** Un handler dont on fait un replay écrit de nouvelles lignes dans son propre outbox dans la même transaction, et ces lignes partent vers le service suivant. Si les nouveaux événements portent des identifiants neufs, les consommateurs en aval les traitent comme de nouveaux faits. Soit vous dérivez l'identifiant de chaque événement sortant de façon déterministe à partir de l'`event_id` d'entrée et du nom du handler, afin que la seconde exécution produise les mêmes identifiants et que les inboxes en aval les absorbent, soit vous retenez le relay du service qui fait un replay jusqu'à avoir vérifié ce qu'il a produit.
- **Faire un replay dans un autre environnement.** Pour une projection sans effets, une nouvelle table est la réponse la plus propre, comme ci-dessus. Pour un handler avec effets, un déploiement différent dont la couche d'effets est reliée à un stub permet de tester le replay avant de l'exécuter pour de vrai.

> **Avertissement :** Réinitialiser un groupe qui envoie des e-mails, débite des cartes ou appelle des transporteurs, sans gate ni clés, transforme chaque événement historique en une nouvelle action. L'inbox n'aide pas si vous le réinitialisez aussi, et pour les effets externes il ne l'aurait jamais pu, car le fournisseur ne fait pas de rollback avec votre base de données. Décidez de la méthode de neutralisation avant de déplacer les offsets, pas après la première plainte.

La fenêtre de déduplication et l'inbox

L'inbox change ce que fait un replay, et cela joue pour vous ou contre vous selon ce que vous voulez.

Si la base de données du consommateur est intacte et que la plage que vous relisez se trouve dans sa fenêtre de déduplication (Chapitre 7), l'inbox reconnaît chaque `event_id` qu'il a déjà traité et le saute. C'est ce que vous voulez après un trou : le groupe ne traite que ce qui manque, et le reste coûte peu. C'est aussi pourquoi un replay peut réparer le consommateur qui n'a jamais vu un événement, sans déranger ceux qui l'ont vu.

Si la plage est plus ancienne que la fenêtre, les lignes de l'inbox ont disparu. Le consommateur applique les anciens événements une seconde fois. Pour un consommateur protégé par une contrainte sur une clé métier, comme une écriture de registre par capture, la seconde application échoue sans dommage. Pour un consommateur qui n'en a pas, l'effet se produit deux fois. La fenêtre de déduplication fait donc partie du plan de reprise : avant un replay, indiquez l'événement le plus ancien de la plage, la fenêtre la plus courte parmi les groupes qui le verront, et ce qui protège le consommateur là où les deux ne se recouvrent pas. Le Chapitre 7 a donné la règle selon laquelle la fenêtre doit durer plus longtemps que tout replay que vous prévoyez. Voici le plan.

L'inbox peut aussi être l'obstacle. Un consommateur avec un bug a écrit un état erroné. Vous corrigez le bug et faites un replay, et chaque message est sauté, car chaque `event_id` est dans l'inbox. Ce cas arrive un peu plus loin.

Replay, réparation et redrive

Trois outils font de nouveau circuler des messages, et ils diffèrent par ce sur quoi ils agissent.

Outil	Ce sur quoi il agit	Chapitre	Adapté quand
Réparation	Une poignée d'enregistrements, republiés via l'outbox du producteur avec l'`event_id` d'origine	13	Quelques éléments manquent ou sont erronés, et le producteur peut dire lesquels
Redrive	Des dead letters, renvoyés au consommateur qui les a rejetés	8	Les messages ont échoué de façon visible et la cause est corrigée

Outil	Ce sur quoi il agit	Chapitre	Adapté quand
Replay	Une plage du log, relue par un groupe	14	De nombreuses clés sont touchées, les données sont encore en rétention, et le consommateur est idempotent

La question qui les distingue est de savoir qui sait ce qui ne va pas. Avec la réparation, le producteur ou un job de réconciliation nomme les éléments précis. Avec le redrive, c'est le système lui-même qui a mis les messages de côté. Avec le replay, personne n'a de liste : le dommage est une plage de temps, et la façon la moins coûteuse de couvrir une plage est de tout relire et de laisser l'inbox trier ce qui a été fait. Le replay échange la précision contre la couverture.

La rétention est une limite de la reprise

Le replay ne remonte que jusqu'où le log conserve des données. La rétention, un réglage choisi il y a longtemps pour le coût du disque, est donc aussi la limite de la reprise la moins coûteuse dont vous disposez. Une panne qui dure plus longtemps que la rétention, ou un bug découvert après elle, vous fait passer du chemin bon marché à la republication depuis les producteurs et à la réconciliation.

Je fixerais la rétention en fonction du temps dont l'équipe a besoin pour remarquer et corriger un problème, pas en fonction du seul disque. Si un job de réconciliation tourne chaque jour et qu'un correctif met un jour à être livré, une rétention d'un jour ne laisse rien. La rétention interagit aussi avec le côté des coûts : une rétention plus longue augmente le nombre d'anciens schémas qu'un replay peut rencontrer (Chapitre 12), et elle prolonge la durée de vie des données personnelles dans le log. Je ne connais pas le bon chiffre pour votre système. Il dépend de la vitesse à laquelle vous remarquez, de la durée de la réparation et de ce que le log a le droit de contenir.

Un consommateur qui s'est trompé sans échouer

Le Chapitre 3 a une ligne pour un bug de consommateur sans erreur : le handler a réussi et a écrit le mauvais résultat. C'est la reprise qui a le plus besoin du replay, et celle où l'inbox fait obstacle.

Imaginez que le consommateur de `PaymentCaptured` de Ledger ait converti les montants avec le mauvais exposant pour une devise, pendant une semaine. Chaque message a reçu un ack. Rien n'a échoué, et l'inbox contient chaque `event_id`. La réconciliation (Chapitre 13) finit par signaler que les totaux de Ledger pour cette devise ne concordent pas avec Payments. Le correctif est livré, et maintenant les lignes erronées existent et doivent être corrigées.

Faire un replay du topic dans le même consommateur ne fait rien, car l'inbox dit que chaque événement est traité. Il existe deux voies honnêtes, et le choix dépend de ce qu'est Ledger.

Si l'état du consommateur est dérivé et peut être jeté, comme une projection, reconstruisez-le. Démarrez un nouveau groupe vers un nouveau stockage avec un inbox vide, faites un replay depuis le premier événement touché, vérifiez le résultat par rapport à la source avec la réconciliation et basculez. L'état erroné est jeté, pas corrigé.

Si l'état du consommateur est un enregistrement qui ne doit pas être réécrit, comme dans un registre append-only, le replay aide quand même, mais pas pour écraser. Lisez la plage touchée avec le code corrigé dans un mode correction. Pour chaque événement, calculez ce que l'écriture aurait dû être, comparez-la à ce qui est comptabilisé, net des corrections antérieures, et passez une écriture de correction pour la différence, avec un identifiant dérivé de l'`event_id` d'origine et du nom propre de la correction, afin qu'exécuter la correction deux fois soit un no-op. La contrainte d'unicité sur l'identifiant propre de l'écriture de registre fait ce travail ; la correction n'a donc besoin d'aucune ligne d'inbox, et la boucle traite une clé en double comme « déjà corrigé » et passe à l'événement suivant.

Les écritures erronées restent, comme historique. Les corrections sont de nouveaux faits qui disent pourquoi. C'est plus lent et plus explicite qu'une reconstruction, et c'est la seule forme qu'un auditeur accepte.

```
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
```

Le fil conducteur est que le replay trouve la plage touchée à peu de frais, et que la décision sur ce qu'il faut faire de chaque événement appartient à la sémantique propre du consommateur.

Scénarios de sinistre

Un sinistre est une défaillance où la perte n'est pas un message mais un stockage. La manière utile d'organiser les possibilités est de partir du stockage qui est revenu en arrière dans le temps. Rédigez chaque scénario selon le pattern utilisé au Chapitre 3 : ce qui a cassé, comment cela se manifeste et quelle garantie tient encore. Le dernier des trois est le plus utile, car il nomme ce sur quoi vous comptez, et un flux qui n'a rien à mettre à cet endroit est exposé dans ce scénario. Avant les scénarios, un mot sur ce que vous visez.

Des objectifs pour chaque stockage

Deux nombres décrivent une reprise. L'**objectif de point de reprise (RPO)** est la quantité de données que vous pouvez vous permettre de perdre, mesurée en temps : une sauvegarde toutes les six heures signifie un RPO pouvant aller jusqu'à six heures. L'**objectif de temps de reprise**

(RTO) est la durée pendant laquelle vous pouvez vous passer du service. Les deux sont des décisions, et l'erreur habituelle est d'en prendre une seule pour tout le système.

Ils diffèrent parce que les stockages diffèrent. Pour les stockages de ce chapitre, une ébauche ressemble à ceci.

Stockage	Ce que coûte une perte	Voie de retour habituelle	Ce que cela détermine
Une base de données de service	Un état qui n'existe nulle part ailleurs, sauf si un événement le porte	Sauvegarde plus log shipping	RPO par base de données, fixé par la valeur métier des données
L'outbox	Les événements pas encore publiés	Même base de données, donc même RPO	Toute ligne perdue est un événement qui n'a jamais existé
L'inbox	La mémoire de ce qui a été traité	Même base de données que les effets	Recule avec eux (Chapitre 3)
Le broker	Les événements en transit et l'historique retenu	Réplication, ou republication depuis les producteurs	Le RPO est fixé par la réplication, le RTO par la vitesse de vidange des consommateurs
Une projection dérivée	Rien, si elle peut être reconstruite	Replay	Le RTO est la durée du replay, le RPO est nul si le log contient tout
Un fournisseur externe	Des effets hors de votre contrôle	Ses enregistrements, et la réconciliation	En dehors de vos deux objectifs

L'intérêt du tableau est que le RTO de la projection reconstruite par replay est un calcul de vidange. Si le log contient un mois et que le groupe peut traiter l'équivalent d'une journée par heure, la reconstruction prend environ 30 heures, et c'est son RTO (supposez ces chiffres ; les vôtres viennent de la mesure, comme au Chapitre 13). Une équipe qui n'a jamais mesuré son taux de vidange ne connaît pas son RTO.

Je lierais aussi les objectifs au modèle de défaillance du Chapitre 3, flux par flux. Le flux de capture de paiement a une tolérance différente de celle du flux de notification, et la tolérance de Ledger pour un trou est plus proche de zéro. Un objectif uniforme finit par être trop strict pour les notifications et trop lâche pour le registre.

Restaurer une base de données depuis une sauvegarde

Supposons que la base de données d'Orders soit restaurée à un point situé six heures en arrière. Qu'est-ce que cela fait à la mécanique de ce livre ?

L'**outbox** recule avec l'état. Les lignes écrites pendant ces six heures ont disparu, avec les commandes qu'elles annonçaient, même si Payments et Inventory ont peut-être déjà agi sur les événements. Les lignes qui n'étaient pas publiées au point de restauration mais l'ont été plus tard dans l'ancienne chronologie redeviennent non publiées, si bien que le relay les publie une seconde fois. C'est un doublon sans danger, car l'`event_id` est le même.

Il existe un danger plus aigu. Si les identifiants de commande proviennent d'une séquence, la séquence restaurée est rembobinée, et la prochaine commande recevra un identifiant déjà attribué, à une autre commande, dans la chronologie perdue. Payments détient désormais une capture pour `order-81724` qui appartient à un acheteur qu'Orders a oublié, et la nouvelle `order-81724` appartient à quelqu'un d'autre. Pour cette raison, je préférerais des identifiants aléatoires opaques pour les agrégats. Si vous devez garder une séquence, avancez-la au-delà de la plus haute valeur jamais émise avant d'accepter du trafic, un nombre que vous pouvez lire dans les services en aval.

L'**inbox** d'un consommateur dont la base de données est restaurée recule en même temps que les effets. Le Chapitre 3 a souligné que c'est correct : les messages traités avant le point de restauration sont toujours dans l'inbox et leurs effets ont survécu, et les messages postérieurs sont oubliés aux deux endroits. Cette propriété ne tient que lorsque l'inbox et les effets vivent dans la même base de données.

Ce qui ne recule pas, c'est l'**offset** du consommateur, qui vit dans le broker. Le groupe est positionné au head, au-delà de six heures d'événements que la base de données restaurée a oubliés. Le consommateur ne les reverra jamais à moins que quelqu'un ne fasse reculer le groupe ; le plan de restauration inclut donc un reset de ce groupe.

La cible demande du soin, car le point de restauration est une heure sur l'horloge de la base de données, alors qu'un reset par horodatage travaille sur les heures des événements dans le log. Au moment de la restauration, le groupe n'était pas au head. Il était à une certaine position, et l'événe-

ment à cette position avait été produit quelque temps auparavant, jusqu'à l'âge du message du groupe (Chapitre 13). La cible est donc l'heure de l'événement à la position du groupe à ce moment-là, moins une marge. Si la cible est fixée à partir du seul point de restauration, un groupe qui avait une heure de retard saute l'heure qu'il n'avait pas encore traitée quand la base de données a reculé.

Lisez la position dans l'historique de l'âge des messages ou dans les offsets enregistrés, et gardez cet historique en dehors de la base de données que vous êtes sur le point de restaurer. Certaines équipes stockent l'offset dans la même base de données que les effets, dans la même transaction, afin que les deux reculent ensemble. Cela supprime l'étape manuelle et retire au broker la tenue des offsets, ce qui est un coût à peser.

Les effets externes restent où ils étaient. La capture que Payments a envoyée au fournisseur dans la chronologie perdue existe dans les enregistrements du fournisseur et pas dans la base de données de Payments. C'est là que la réconciliation (Chapitre 13) est indispensable. Comparez vos enregistrements avec ceux du fournisseur, et réparez les différences.

L'ordre dans lequel le système redémarre compte, et il vaut la peine de l'écrire comme une procédure.

```
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")
```

Les étapes sont l'essentiel. Mettre d'abord les deux côtés en pause empêche le service restauré d'aggraver son état incohérent, et réconcilier avant de reprendre suit le Chapitre 3. Le reset transforme la restauration en problème de replay, avec un ajout.

Un Orders restauré qui relit les heures perdues produit de nouvelles lignes d'outbox, dont un command `CapturePayment`, et un `command_id` neuf paraîtrait nouveau à l'inbox de Payments et pourrait capturer deux fois. Les identifiants des événements dérivés viennent donc de l'`event_id` d'entrée, comme dans la section sur les effets de bord. Un command déclenché par deux événements, comme l'est `CapturePayment`, tire plutôt son identifiant de la clé métier, par exemple `order_id` plus le type de command (Chapitre 7), car celui des deux qui est lu en dernier peut changer lors d'un replay. Le relay est retenu jusqu'à ce que quelqu'un ait vérifié ce qui a été produit.

Perdre des données du broker

Un broker peut perdre des données sans perdre le cluster. Dans la famille des logs, une réplique qui était en retard peut être promue leader, et les messages qu'elle n'avait pas disparaissent. Le niveau d'ack et le nombre minimal de répliques synchronisées décident si l'on a dit au producteur qu'un tel message était en sécurité (Chapitre 2). Certains brokers autorisent ou interdisent par configuration la promotion d'une réplique en retard, et je chercherais la valeur par défaut du vôtre pour la noter. La rétention peut faire de même en silence, comme l'a montré le Chapitre 13.

Ce que vous voyez : des positions de consommateurs qui sautent ou sont signalées hors plage, un topic dont l'offset de fin est plus bas qu'avant et, après coup, des constats de réconciliation de type « manquant ». L'ack du broker est une preuve avec une limite, et cette limite est le nombre de copies qui existaient à ce moment-là.

Ce qui survit : l'outbox, si vous avez gardé les lignes. Un relay qui purge les lignes juste après la publication n'a aucune mémoire de ce qu'il a envoyé. Un relay qui garde les lignes publiées pendant une période, disons quelques jours à titre d'illustration, peut marquer comme non publiées les lignes de la fenêtre touchée et les renvoyer. La republication conserve l'`event_id` d'origine, si bien que les consommateurs qui ont bien reçu un événement traitent la seconde copie comme un doublon. C'est la même étape que la réparation du Chapitre 13, appliquée à une plage, et elle comporte la même précaution sur le fan-out : les événements plus anciens que la fenêtre de déduplication d'un abonné doivent être livrés au seul consommateur qui les a manqués, pas à tout le monde.

Si les lignes de l'outbox ont disparu, les enregistrements propres du producteur sont la source suivante : reconstruisez l'enveloppe à partir de l'enregistrement métier avec son `event_id` stocké et le hash de son payload, et arrêtez-vous pour un humain là où le hash diffère. Et si aucun des deux n'existe, l'événement ne peut pas être récupéré et la réconciliation continuera à le signaler. À ce stade, la bonne question est une question métier : le fait manquant est-il reconstruit à partir d'un autre système, ou accepté comme perdu et expliqué ? Concevoir de façon que cette question soit rarement posée, c'est ce qu'achète le fait de garder les lignes de l'outbox quelque temps.

Perdre une région

Une perte de région combine les deux cas précédents, et ajoute un problème qui lui est propre. Supposons que Parcelly fasse tourner un cluster de secours dans une seconde région, alimenté par un outil de réplication qui copie les topics en continu, et que les bases de données se répliquent de façon asynchrone.

La première propriété à énoncer est que **la réplication asynchrone a un RPO**, et c'est le retard de réplication au moment de la défaillance. Tout ce qui avait été écrit dans la région principale et pas encore copié a disparu du point de vue du cluster de secours. C'est la mesure de ce que vous perdez, et c'est un nombre qui varie de minute en minute. Il mérite aussi une alerte, contre une limite déclarée, de la même manière que le Chapitre 13 traite l'âge des messages : vous surveillez votre RPO.

La deuxième propriété est que **les offsets ne voyagent pas**. Le cluster de secours détient une copie des messages, mais cette copie a ses propres offsets, attribués quand la réplique l'a écrite. La position enregistrée par le commit d'un consommateur dans le primary ne dit rien du cluster de secours. Certains outils de réplication proposent une traduction des offsets ; je vérifierais ce que le vôtre garantit, et je ne concevrais pas une reprise qui dépende de son exactitude.

Le plan portable consiste à démarrer chaque groupe sur le cluster de secours à partir de l'heure de l'événement à la position qu'il avait atteinte quand le primary est tombé, moins une marge, et à laisser l'inbox absorber le recouvrement. C'est la règle de la restauration de base de données. Un groupe qui avait 40 minutes de retard démarre 40 minutes en arrière. Les événements qui n'ont jamais été répliqués ne sont pas du tout sur le cluster de secours, et la fenêtre de l'outbox ci-dessous les couvre.

Puisque les informations de commit propres au primary sont perdues avec la région, l'historique de l'âge des messages doit être conservé en dehors. S'il a disparu, rabattez-vous sur le point le plus ancien situé dans les fenêtres de déduplication des consommateurs et acceptez les doublons supplémentaires. Un seuil d'alerte ne suffit pas, car l'alerte était peut-être en train de se déclencher à ce moment-là.

La troisième propriété est que **les doublons au failover sont certains, pas possibles**. Un consommateur qui redémarre plus tôt que sa vraie position traitera des événements deux fois. Un relay qui était en pleine publication quand la région est tombée peut publier une seconde fois depuis la base de données survivante.

Les trous sont l'autre face : des événements dont le primary a fait l'ack et que le cluster de secours n'a jamais reçus. Si la base de données du producteur a survécu et qu'elle est plus récente que le broker de secours, elle contient des lignes marquées publiées que le cluster de secours n'a jamais vues. Pour celles-ci, élargissez la fenêtre de republication de l'outbox comme dans le cas de la perte du broker : marquez comme non publiées les lignes du dernier intervalle de retard de réplication, plus une marge, et renvoyez-les. Le retard de réplication dimensionne cette fenêtre, et seulement cette fenêtre. Les doublons ne coûtent rien au consommateur. Les trous coûtent ce que vous essayez de protéger.

Si la base de données du producteur est elle-même perdue avec la région et restaurée depuis une réplique plus ancienne, le cas se ramène à une restauration depuis une sauvegarde, avec les séquences rembobinées et les commandes oubliées qui vont avec.



Lisez le diagramme depuis la gauche. Les deux étapes qui traitent l'inconnu, la fenêtre de republication et la marge d'horodatage, penchent toutes deux vers la duplication. Chaque étape qui suit existe pour rendre la duplication sans danger.

Migration de brokers

La migration est un sinistre que vous planifiez. Une équipe passe d'un broker à un autre pour une question de coût, une fin de support, une offre managée ou un changement de pattern d'accès, et le déplacement a les mêmes ingrédients que les sinistres ci-dessus : des données d'un côté, un nouvel endroit vide de l'autre, des positions qui ne se transfèrent pas, et une période pendant laquelle deux systèmes vivent en même temps. La différence, c'est que vous avez le temps d'être prudent, et qu'aucune urgence ne vous force à improviser.

Les outils du livre font d'une migration un changement maîtrisé. La raison est que plusieurs d'entre eux ont été placés exprès entre l'application et le broker.

Le relay comme point de bascule unique

Avec un transactional outbox (Chapitre 4), un service producteur ne parle jamais au broker. Il écrit dans sa propre base de données, et le relay (Chapitre 5) déplace les lignes vers le broker. Le choix du broker est donc encodé dans un seul processus. Changer la destination des événements, c'est changer ce vers quoi le relay publie, et le code du producteur ne change pas. C'est un changement plus petit qu'un changement réparti sur tous les services, un bénéfice secondaire d'un pattern que le Chapitre 4 recommande pour une autre raison.

Le relay possède aussi l'enveloppe. Il prend `event_id` dans la ligne de l'outbox et le place dans le message qu'il publie. Sur le nouveau broker, le même événement a le même `event_id`, et c'est cette égalité qui rend sûr le reste de la migration.

Dual-publish

La première étape la plus sûre consiste pour le relay à publier chaque ligne vers les deux brokers. Pendant un temps, chaque événement existe sur l'ancien broker, où les consommateurs d'aujourd'hui le lisent, et sur le nouveau, où personne ne le lit encore.

Le dual-publish est lui-même un dual write, et le Chapitre 4 explique pourquoi ceux-ci sont dangereux. Ici le risque est plus faible, car la ligne de l'outbox est la source de vérité et un crash laisse la ligne à renvoyer.

La question de conception est de savoir ce qu'attend le relay. S'il ne marque une ligne comme publiée qu'après le succès des deux publications, un shadow broker injoignable laisse des lignes non publiées et arrête le flux réel dont dépendent les consommateurs, pour un broker que personne ne lit. Suivez donc les deux cibles séparément. La ligne est publiée, au sens du Chapitre 5, quand l'ancien broker l'a reçue, et un second marqueur enregistre le nouveau broker. Une passe, dans l'ordre des `id`, alimente le nouveau broker avec les lignes auxquelles manque le second marqueur, ce qui conserve l'ordre par clé, et la purge attend les deux. Le

coût est une colonne et une passe de plus, et en échange une panne du nouveau broker apparaît comme un écart croissant dans la comparaison et non comme une panne de Parcelly.

Pendant cette phase, l'ancien broker reste celui auquel les consommateurs font confiance. Le nouveau broker est un shadow, et sa seule tâche est d'être comparé à l'original.

Mise en miroir et bridges

Une alternative au dual-publish dans le relay consiste à copier les messages entre brokers avec un outil : un outil de mise en miroir pour passer de log à log, ou un bridge ou un connecteur pour passer de log à file d'attente et de file d'attente à log. Cela ne demande aucune modification du relay et transporte aussi l'historique qui existait avant le début de la migration, ce que ne fait pas le dual-publish.

Deux mises en garde. Un miroir ou un bridge est un consommateur et un producteur ; il est donc at-least-once et produit des doublons, comme toute paire consommateur-producteur. Et il doit transmettre `event_id`, dans le corps du message ou dans un en-tête, sans le modifier. Un bridge qui génère son propre identifiant transforme chaque événement copié en nouvel événement, et l'inbox ne peut en reconnaître aucun. Testez cela sur un échantillon avant de vous y fier. Savoir si un miroir conserve le partitionnement, les clés et les en-têtes, et dans quel ordre d'une partition à l'autre, dépend de l'outil, et la réponse se trouve dans sa documentation, pas dans ce livre.

Les offsets ne se traduisent pas

L'offset d'un groupe de consommateurs sur l'ancien broker n'a aucun sens sur le nouveau. Même entre deux brokers du même type, la copie a sa propre numérotation. Entre une file d'attente et un log, il n'y a pas d'offset du tout d'un côté.

La réponse est celle à laquelle la section sur la perte de région est arrivée : choisir un point de départ sûr et s'appuyer sur l'idempotence. Pour chaque groupe, choisissez un horodatage tel que tout événement que le groupe n'avait pas fini de traiter sur l'ancien broker ait été publié après lui. Prenez le retard du groupe au moment de la bascule, convertissez-le en temps et

ajoutez une marge. Démarrez le nouveau groupe à cet horodatage sur le nouveau broker. Il relit des événements que l'ancien groupe a déjà traités, et l'inbox les saute.

La fenêtre de déduplication s'applique ici comme à un replay : l'événement le plus ancien que le nouveau groupe relira doit se trouver dans la fenêtre de chaque consommateur.

Ordre des messages et partitions

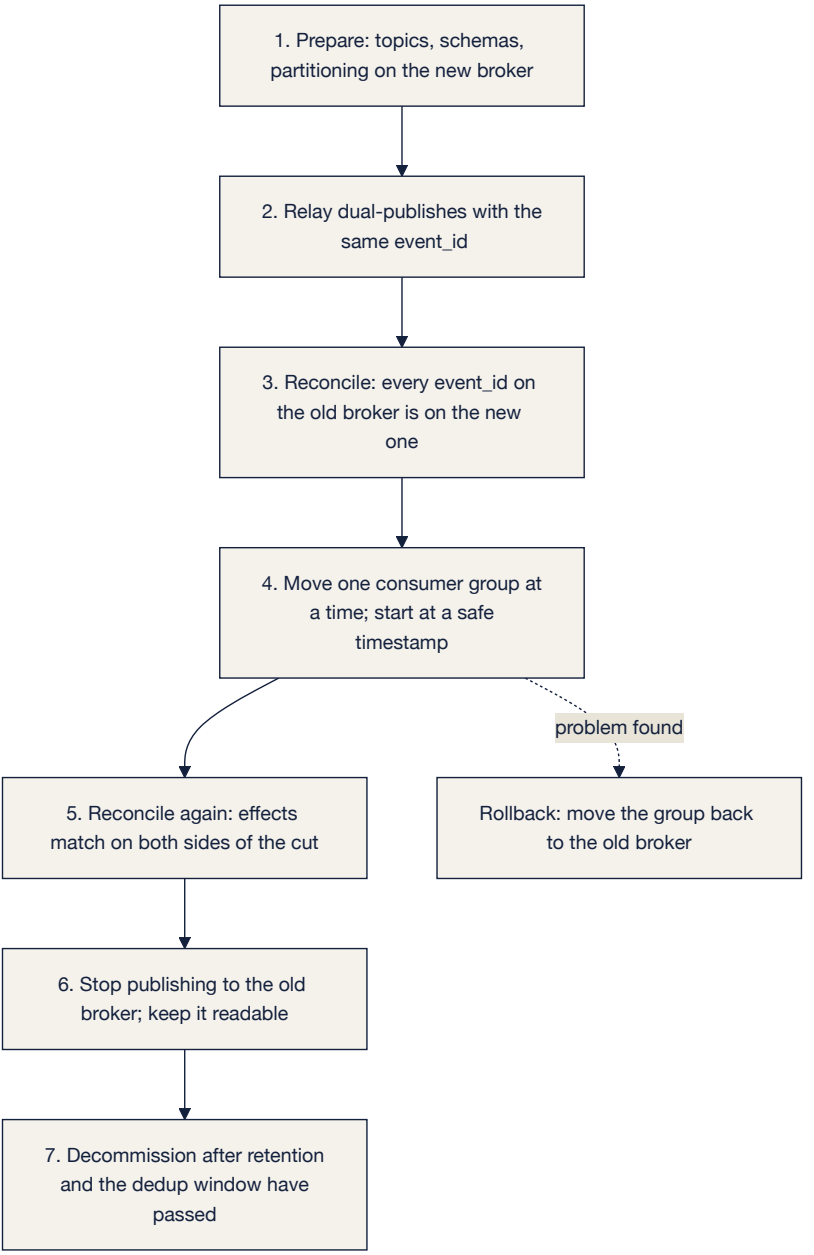
Le Chapitre 9 a donné la condition de l'ordre par clé : même clé, même partitionneur, même nombre de partitions. Une migration change au moins l'un des trois. Le nouveau broker peut utiliser un partitionneur différent, si bien qu'un même `order_id` atterrit sur une partition où il n'avait jamais atterri, et le nouveau topic peut être créé avec un nombre de partitions différent, parce que c'est le moment où l'on reconsidère ce nombre.

Deux conséquences en découlent. Calculez la partition dans le relay à partir de la `partition_key` stockée, comme le suggérait le Chapitre 9, afin qu'un seul morceau de code possède la correspondance, et testez que les événements d'une clé atterrissent sur une seule partition dans le nouveau broker. Et planifiez la bascule de façon qu'aucune clé n'ait d'événements en transit aux deux endroits. Si un consommateur commence à lire le nouveau topic alors que l'ancien topic contient encore des événements non traités pour le même `order_id`, les nouveaux événements peuvent être traités avant les anciens : de nouveau `OrderCancelled` avant `OrderConfirmed`, la défaillance du Chapitre 9. Il existe deux façons de l'éviter. Vider l'ancien backlog jusqu'à zéro avant de démarrer le nouveau groupe, ce qui suspend le traitement pendant la durée de la vidange. Ou rendre les consommateurs tolérants par conception, avec des événements porteurs d'état et une version d'agrégat qui rejette tout ce qui est plus ancien, comme le décrivait le Chapitre 9. Je viderais d'abord pour tout flux où l'ordre compte et où la pause est supportable.

La bascule

L'ordre des opérations est la partie sur laquelle on pose des questions : les consommateurs d'abord, ou les producteurs d'abord ? Avec le dual-publish, la question se dissout en grande partie, car les producteurs écrivent déjà

vers les deux avant qu'un seul consommateur ne bouge. L'ordre devient celui-ci.



La réconciliation encadre le déplacement. Avant, comparez l'ensemble des valeurs `event_id` sur les deux brokers sur une fenêtre : tout ce qui est sur l'ancien broker et manque sur le nouveau est un défaut du dual-publish, trouvé alors que personne ne dépend du nouveau broker. Après, comparez

les effets : un groupe déplacé à 14:00 devrait avoir produit le même état que celui qu'il aurait produit sur l'ancien broker, et c'est la réconciliation entre services du Chapitre 13 qui le vérifie. Une migration déclarée terminée parce que les consommateurs sont en bonne santé a seulement vérifié que rien n'a planté.

Déplacez les groupes de consommateurs un par un, en commençant par le moins critique, disons Notifications, et en terminant par Ledger. Chaque groupe est une petite étape réversible séparément.

Rollback

Une migration sans plan de rollback est une porte à sens unique. Avec cette conception, le plan est presque gratuit, tant qu'une condition tient : l'ancien broker continue de recevoir chaque événement. Tant que le relay fait du dual-publish, l'ancien broker est complet, et faire un rollback signifie donc repointer le groupe vers l'ancien broker à un horodatage avec une marge. L'inbox absorbe les événements que le groupe a traités au nouvel endroit.

Cette condition explique pourquoi l'étape six arrive tard. Une fois que le relay cesse d'écrire vers l'ancien broker, un rollback exige d'inverser le flux, le relay écrivant de nouveau vers l'ancien et les consommateurs de l'ancien broker repartant du point où il s'est arrêté. C'est encore possible parce que l'outbox contient les lignes, et c'est plus lent. Le rollback est aussi borné par la rétention sur l'ancien broker et par la fenêtre de déduplication ; je ne le mettrais donc hors service qu'une fois les deux écoulées depuis le dernier événement qu'il a reçu. Retirer l'ancien broker le jour où le dernier consommateur migre supprime le rollback au moment même où les problèmes du nouveau broker ont le plus de chances d'apparaître.

Entre file d'attente et log

Quand la migration traverse les familles, la sémantique change, et un plan qui la traite comme un changement d'adresse est faux.

Aspect	De la file d'attente au log	Du log à la file d'attente
Historique	Gagné : le replay devient possible (et la rétention devient une décision)	Perdu : pas de replay, la reprise repose donc sur l'outbox et les producteurs

Aspect	De la file d'attente au log	Du log à la file d'attente
Ack	L'ack par message devient un commit d'offset ; un message lent peut désormais bloquer sa partition (Chapitre 9)	Un commit d'offset devient un ack par message, avec visibility timeouts et nouvelle livraison
Ordre des messages	L'ordre de groupe, là où la file d'attente le propose, devient un ordre par partition ; choisissez la clé	L'ordre par partition doit être recréé, par exemple avec des groupes de messages là où la file d'attente les prend en charge
Dead letters	Le redrive est quelque chose que vous construisez ou configurez ; le log n'a pas de dead-letter queue (DLQ) intégrée, seulement un dead-letter topic que vous créez (Chapitre 8)	Les fonctionnalités DLQ de la file d'attente apparaissent, et ses limites et sa rétention diffèrent
Délais et expiration	La livraison différée et l'expiration par message peuvent ne pas exister et nécessiter un substitut	Le délai et l'expiration peuvent être natifs ; la reprise fondée sur la rétention disparaît

Chaque ligne est un endroit où un comportement sur lequel vos consommateurs comptaient peut changer en silence. J'écrirais la liste des comportements dont dépend chaque consommateur avant de choisir la cible, et je testerais ceux qui comptent, comme un poison message qui ne doit pas bloquer sa partition, lors d'une répétition (Chapitre 15).

Répéter la reprise

Rien de tout cela ne vaut grand-chose si la première fois que vous l'exécutez, c'est sous pression. Une restauration qui n'a jamais été faite est une croyance. Une procédure de replay qui n'a jamais tourné sur un volume réaliste n'a pas de durée.

Deux répétitions valent leur coût. Un **test de restauration** prend une sauvegarde, la restaure dans un environnement jetable, vérifie ce que la restauration fait à l'outbox, à l'inbox et aux offsets, puis exécute la procédure de redémarrage et la chronomètre. Un **game day** retire volontairement quelque chose dans un endroit sûr, un broker, une zone, une base de données de consommateur, et fait reprendre l'équipe en n'utilisant que le runbook. Le Chapitre 15 explique comment concevoir ces exercices. La raison a sa place ici : seule l'exécution de la reprise montre quelles étapes vous avez oubliées.

Un runbook pour les scénarios de ce chapitre doit répondre à une courte liste de questions, écrites et non gardées dans la tête de quelqu'un. Le Chapitre 16 les intègre à ses conseils sur les runbooks.

- Quel stockage est touché, et lequel est revenu en arrière dans le temps ?
- Quel est le point de reprise, et comment a-t-il été déterminé ?
- Quels groupes de consommateurs doivent être réinitialisés, à quel horodatage et avec quelle marge ?
- Quels effets doivent être neutralisés pendant le replay, et par quel mécanisme ?
- Chaque plage à relire se trouve-t-elle dans la fenêtre de déduplication de chaque consommateur touché ?
- Qu'est-ce qui est mis en pause, dans quel ordre, et qui le reprend ?
- Quelles réconciliations confirment que la reprise a fonctionné, et qui les lit ?
- Comment annuler l'ensemble si cela tourne mal ?

Compromis et coûts

La rétention est un budget de reprise. Une rétention plus longue garde la reprise bon marché disponible plus longtemps, aux coûts énumérés plus haut. Une rétention plus courte vous pousse à republier depuis les producteurs.

Garder les lignes de l'outbox coûte du stockage et un job de nettoyage. Les mêmes lignes qui rendent possible une fenêtre de republication sont celles qu'une conception plus sobre supprime immédiatement. Avec change data capture (CDC) et suppression immédiate, la fenêtre n'existe pas (Chapitre 5). Si vous voulez cette sécurité, gardez les lignes publiées pendant une période déclarée et surveillez la table.

Le replay est un instrument grossier. Il relit tout ce qui se trouve dans la plage, y compris ce qui allait bien, et il repose sur le fait que chaque consommateur est idempotent et que chaque effet passe par un gate. Pour quelques enregistrements connus, la réparation est moins coûteuse et plus sûre. Le replay gagne sa place quand le dommage est étendu et que personne ne détient de liste.

La neutralisation des effets de bord est du code qui doit être testé. Un gate qui existe mais qu'un nouveau handler contourne est pire que rien, car il crée de la confiance. Le coût est un test permanent et une règle de revue de code.

Une migration avec dual-publish coûte double. Deux brokers tournent en même temps, le relay travaille davantage et suit deux cibles, et les doublons du côté ancien augmentent. En échange, vous obtenez une vérification avant que quiconque dépende du nouveau broker, et un rollback. Si le système tolère une fenêtre de maintenance, une bascule directe coûte moins cher : arrêtez le relay, videz les anciens consommateurs, pointez le relay vers le nouveau broker et démarrez les consommateurs là-bas. Avec un outbox, cela demande une interruption mais aucune perte, car les lignes attendent. Elle renonce à la vérification et au rollback, et je ne la choisirais pas pour le registre.

Quand ne pas construire tout cela. Un flux où un trou coûte peu, comme un flux interne de métriques, n'a pas besoin de fenêtre de republication, de runbook ni de game day. Adaptez la reprise au coût de la défaillance, et notez quels flux vous avez décidé de laisser non couverts. Leur absence du plan doit être une décision.

Erreurs courantes

- **Réinitialiser un groupe sans vérifier ses effets de bord.** C'est tentant, car le reset tient en une seule opération et les données sont là. Le résultat est une deuxième vague d'e-mails et une deuxième tentative de débit. Décidez comment les effets sont neutralisés avant de déplacer le moindre offset.
- **Réinitialiser trop de groupes.** Tout faire repartir de zéro donne une impression d'exhaustivité. Chaque groupe que vous réinitialisez relit l'historique et porte le risque de ses propres effets. Ne réinitialisez que les groupes qui ont manqué quelque chose.
- **S'appuyer sur un en-tête de replay.** C'est tentant, car un drapeau sur le message ressemble au signal le moins coûteux. Un message dont on fait un replay est le message original et ne porte aucun signe qu'il est relu. Le mode appartient au déploiement du consommateur ou à la couche d'effets.

- **Faire un replay dans un consommateur avec l'inbox du bug.** Après un bug de consommateur, l'inbox saute chaque message comme déjà traité. Reconstruisez un état dérivé dans un stockage neuf, ou passez des corrections explicites pour un état append-only.
- **Un seul objectif de reprise pour tout le système.** C'est tentant, car c'est un seul nombre sur lequel s'accorder. Il est trop strict pour les notifications et trop lâche pour Ledger. Fixez les objectifs par flux et par stockage.
- **Restaurer une base de données et reprendre aussitôt.** Le service est en bonne santé et les tableaux de bord sont au vert, alors on reprend le trafic. Le service restauré publie alors un état rembobiné, réutilise des identifiants et manque des événements que ses offsets ont sautés. Mettez en pause, réconciliez, réinitialisez les offsets, puis reprenez.
- **Réutiliser des identifiants issus de séquences après une restauration.** Les séquences sont le choix par défaut et rien ne semble anormal jusqu'à ce que deux commandes partagent un identifiant. Une séquence rembobinée distribue un identifiant qui appartient déjà à une autre commande. Utilisez des identifiants opaques, ou avancez la séquence au-delà de la plus haute valeur jamais émise.
- **Faire confiance à la traduction des offsets.** C'est tentant, car l'outil la met en avant et elle épargnerait l'arithmétique des horodatages. Un outil de réplication qui copie les messages peut ou non faire correspondre exactement les offsets. Prévoyez une reprise qui fonctionne sans elle, avec les horodatages et l'inbox.
- **Un bridge qui génère de nouveaux identifiants.** Un bridge est souvent configuré pour envelopper chaque message dans sa propre enveloppe, et cela passe un test avec un seul consommateur. Les événements copiés paraissent nouveaux, l'inbox ne reconnaît donc rien et chaque doublon devient un effet. Transmettez `event_id` sans modification et testez-le.
- **Mettre l'ancien broker hors service avec la migration du dernier consommateur.** On a l'impression que la migration est terminée. Cela supprime le rollback exactement au moment où il a le plus de chances d'être nécessaire. Attendez que la rétention et la fenêtre de déduplication soient écoulées.

- **Ne jamais restaurer une sauvegarde.** Le job de sauvegarde est au vert. La première restauration est alors la vraie. Restaurez régulièrement dans un environnement jetable et chronométrez l'opération.

Points clés

- Quand un stockage est perdu, restauré ou remplacé, ce qui survit est la sémantique des défaillances qui l'entoure, c'est-à-dire l'outbox, l'inbox, l'`event_id` stable, le contrat et la réconciliation, et le broker lui-même est remplaçable.
- Le replay est un outil de reprise pour la famille des logs : arrêtez un groupe de consommateurs, notez ses offsets, réinitialisez-le à un horodatage mesuré à partir de l'endroit où se trouvait réellement le groupe, avec une marge, et laissez l'inbox absorber les doublons, tant que les données sont encore dans la rétention.
- Un message dont on fait un replay ressemble à l'original ; le consommateur qui fait un replay doit donc connaître son mode et faire passer ses effets par un gate, dériver les identifiants en aval de façon déterministe ou retenir son outbox, et ne jamais s'appuyer sur un en-tête.
- La fenêtre de déduplication fait partie de chaque plan de reprise, car une plage plus ancienne que la fenêtre réadmet des doublons, et un bug de consommateur transforme l'inbox, de protection, en obstacle qui exige une reconstruction ou des corrections explicites.
- Les objectifs de point de reprise et de temps de reprise appartiennent à chaque stockage et à chaque flux, et une base de données restaurée rembobine son outbox, son inbox et ses séquences tandis que les offsets restent là où le broker les a laissés ; la procédure de redémarrage met donc en pause, réconcilie, réinitialise les offsets et seulement ensuite reprend.
- La réplication asynchrone entre régions a un RPO égal à son retard et transporte les données mais pas les offsets des consommateurs ; un failover démarre donc chaque groupe à l'heure de l'événement correspondant à sa propre position au moment de la défaillance, dimensionne une fenêtre de republication de l'outbox d'après le retard de réplication, et traite les doublons comme certains et les trous comme le vrai risque.

- Une migration de broker utilise le relay comme point de bascule unique, fait du dual-publish avec le même `event_id`, déplace un groupe de consommateurs à la fois depuis un horodatage sûr, réconcilie avant et après, et garde l'ancien broker lisible jusqu'à ce que le rollback ne soit plus nécessaire.

Une procédure de reprise qui n'a jamais été exécutée est une hypothèse, tout comme le code censé faire survivre un système à une défaillance. Le Chapitre 15 explique comment tester les systèmes orientés événements, y compris comment répéter les restaurations, les failovers et les replays que ce chapitre a décrits.

Chapitre 15 — Tester les systèmes orientés événements

Imaginez l'équipe Inventory chez Parcelly, le vendredi qui précède un long week-end. Sa suite est au vert : 400 tests, quelques minutes d'exécution, une couverture que personne ne conteste. Le handler de `OrderPlaced` a un test qui construit un événement, appelle le handler et vérifie que le stock a baissé de deux. Il passe à chaque fois. Les tests utilisent un substitut en mémoire du broker, qui livre chaque message une fois, dans l'ordre, immédiatement.

Le samedi, un déploiement redémarre les consommateurs. Les handlers tournent plus lentement que d'habitude pendant qu'ils montent en charge, et l'un d'eux dépasse son visibility timeout ; le broker remet donc le même message `OrderPlaced` à un second consommateur pendant que le premier le traite encore. Les deux lisent l'inbox, les deux n'y trouvent rien, et les deux réservent le stock. La contrainte d'unicité qui aurait dû faire échouer l'un d'eux manquait dans le vrai schéma, et aucun test ne l'a remarqué, car les tests tournaient sur une fausse base de données écrite à la main qui n'appliquait jamais de contraintes et n'exécutait jamais deux handlers à la fois. La suite ne s'était jamais trompée sur ce qu'elle vérifiait. Elle n'avait jamais vérifié la seule chose qui comptait.

Pendant ce temps, un test de la suite d'Orders échoue une exécution sur 20. L'équipe a cessé de le regarder. Il attend deux secondes qu'un événement arrive, et parfois l'événement en met trois.

Les deux histoires ont la même racine : les tests décrivent un système qui n'existe pas. Ce chapitre répond à une question pratique. Comment tester du code dont les pires bugs n'apparaissent qu'en présence de duplication, de délai et de réordonnancement, sans construire une suite lente, flaky ou ignorée ?

Le problème, et pourquoi il est difficile

Le Chapitre 1 soutenait que le happy path est celui que vous pouvez tester sans effort et celui auquel vous êtes enclin à croire. Les chapitres 2 à 14 ont ensuite construit des défenses contre des défaillances que le happy path ne montre jamais. Les défenses sont du code, et un code jamais exercé sous la défaillance qu'il traite est un espoir. Le Chapitre 3 a introduit un modèle de défaillance avec un champ « vérifié par » pour chaque entrée. C'est dans ce chapitre que ce champ se remplit.

Trois propriétés rendent ces tests difficiles.

Les défaillances sont des entrelacements, pas des entrées. Un test de fonction fournit une entrée et vérifie une sortie. Le bug de double réservation exige un timing particulier : deux livraisons qui se chevauchent dans le temps, ou un commit suivi d'un crash et d'une nouvelle livraison. Aucune valeur d'entrée ne le déclenche. Le test doit contrôler le moment où les choses se produisent, ce qui signifie que le harnais de test doit posséder le comportement du broker, l'horloge et les points de crash.

L'asynchronie cache le moment de vérité. Après l'exécution d'un handler, l'effet peut se produire dans un autre service quelques secondes plus tard. Un test ne peut pas faire d'assertion immédiatement, et il ne peut pas savoir combien de temps attendre. Dormir pendant une durée fixe est faux dans les deux sens : trop court, le test est flaky ; trop long, la suite est lente.

Les fakes mentent exactement aux endroits qui comptent. L'idempotence repose sur une contrainte d'unicité. L'atomicité repose sur une transaction. L'ordre repose sur l'attribution des partitions. Un fake qui saute tout cela vous donne une suite au vert et aucune information. Le prix du réalisme est la vitesse, et tout l'art consiste à décider où le payer.

Adapter la pyramide de tests

La pyramide de tests classique tient toujours : beaucoup de tests rapides en bas, peu de tests lents en haut. Ce qui change, c'est le rôle de chaque couche, et la défaillance que chaque couche est chargée de détecter.

Couche	Ce qui tourne	Réel ou fake	Détecte
Tests de handler	Un handler contre une vraie base de données	Vraie base de données, pas de broker	Mauvais effet, inbox manquante, atomicité cassée, mauvaise gestion des doublons et de l'ordre
Tests de contrat	Les formes d'événements entre producteur et consommateur	Aucun service	Changements de schéma qui cassent un consommateur
Tests de composant	Un service avec un vrai broker, un outbox et un relay	Vrai broker, autres services remplacés par des stubs	Câblage : de l'outbox au relay puis au topic, acks, retries, envoi en dead letter
Tests de flux	Quelques services ensemble, une défaillance injectée	Tout est réel, à petite échelle	Comportement entre services, comme les compensations et les timeouts
Sondes en production	Des événements synthétiques dans le système réel	Réel	Problèmes de déploiement et de configuration qu'aucun environnement de préproduction ne montre

Deux choix du tableau méritent chacun une phrase. La couche des handlers utilise un vrai moteur de base de données, car l'inbox, l'écriture dans l'outbox et la vérification de version dépendent toutes de contraintes et de transactions qu'un fake devrait réimplémenter, et c'est en les réimplémentant que la suite d'Inventory s'est fait berner. Le broker, au contraire, est laissé de côté à cette couche. Un handler est une fonction qui, à partir d'un message et d'un état, produit un nouvel état et quelques messages sortants. Si vous pouvez l'appeler directement, vous pouvez le tester rapidement.

La forme de la pyramide découle aussi de l'endroit où vivent les défaillances. La plupart des défenses de ce livre sont de la logique côté consommateur ou côté producteur, que la couche des handlers atteint à peu de frais. Seul le câblage a besoin d'un broker, et là le nombre de tests est faible, car chacun prouve une propriété de la configuration, pas de la logique métier.

Tester les handlers

Un test de handler doit énoncer le contrat dans le langage du livre : étant donné ce message et cet état, l'effet se produit une fois et l'événement sortant est écrit dans l'outbox. Puis le même test reçoit un second acte, où le message arrive de nouveau.

```
function testReservationIsIdempotent():
    db = freshDatabase()
    seedStock(db, "sku-17", available: 5)
    message = buildEvent("OrderPlaced", order_id: "order-1",
                         lines: [{ sku: "sku-17", qty: 2 }])

    handleOrderPlaced(db, message)
    // same event_id: the broker redelivered it
    handleOrderPlaced(db, message)

    assert stockAvailable(db, "sku-17") == 3
    assert reservationCount(db, "order-1") == 1
    assert outboxCount(db, "StockReserved", "order-1") == 1
```

La dernière assertion est celle que les équipes oublient. Un doublon qui ne réserve pas deux fois mais émet `StockReserved` deux fois cause quand même des dégâts en aval, et vérifier l'outbox le détecte.

Ce test couvre les doublons séquentiels, que traite la vérification du chemin rapide dans l'inbox. Le cas dangereux est le doublon concurrent, où deux workers passent tous deux la vérification. Pour le tester, le harnais exécute deux handlers en même temps et utilise une barrière pour les retenir tous deux juste avant la transaction, afin que l'entrelacement soit forcé au lieu d'être espéré.

```
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
```

Ce test échoue si quelqu'un supprime la contrainte d'unicité et s'appuie sur la vérification lecture-puis-insertion, ce qui est précisément la régression la plus difficile à repérer en revue. Le Chapitre 7 a expliqué pourquoi la contrainte est le filet de sécurité. Un test comme celui-ci rend l'explication exécutable.

Les points de crash fonctionnent de la même manière. Si le handler appelle `crashPoint("before-ack")` là où l'ack aurait lieu, un test peut armer ce point et y provoquer un crash simulé.

```
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
```

Lisez l'assertion dans le bloc `on`. Elle vérifie que le commit a survécu au crash, ce qui est la moitié de l'histoire. L'autre moitié est la nouvelle livraison qui suit. Un test de crash qui ne vérifierait que la seconde moitié passerait aussi pour un handler qui n'a jamais fait de commit.

Injecter des doublons, du réordonnancement et du délai

Les tests individuels encodent les cas auxquels vous avez pensé. Le modèle de défaillance contient aussi ceux auxquels vous n'avez pas pensé ; j'ajouterais donc une couche de tests qui perturbent tout un stream d'événements et vérifient que le résultat ne change pas.

L'idée est une propriété de convergence. Prenez un stream enregistré et correctement ordonné pour un agrégat. Calculez l'état final attendu en repliant directement les événements, sans aucun broker. Puis livrez les mêmes événements au handler selon une planification perturbée, certains mélangés à l'intérieur d'une fenêtre et certains dupliqués, et vérifiez que l'état final est égal à l'état attendu.

```
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
```

Exécutez-le pour quelques centaines de graines en CI. Deux détails le gardent honnête. La graine est journalisée, si bien qu'un échec sur la graine 4172 est un échec que vous pouvez relancer sur un portable au lieu d'une anecdote. Et le résultat de référence provient d'un autre chemin de code que le handler, si bien qu'un bug du handler ne peut pas apparaître aussi dans l'attendu.

Le délai est la troisième perturbation. Un message retardé est un message qui arrive après ses successeurs, ou après le déclenchement d'un timeout. C'est le temps qu'il faut contrôler ici : les handlers et les sagas qui lisent directement l'horloge système ne peuvent pas être testés pour les timeouts sans attendre. Je passerais l'horloge comme dépendance, et je laisserais le test la faire avancer. Le Chapitre 10 a énuméré les états dans lesquels une saga peut se trouver. Un test de saga devient alors un tableau : pour chaque état, livrez chaque événement susceptible d'arriver, y compris les retardataires, et vérifiez l'état obtenu et la compensation émise.

État de la saga	Événement qui arrive	Résultat attendu
`AWAITING_RESULTS`	`PaymentAuthorized`	Reste en `AWAITING_RESULTS`, en attente du stock
`AWAITING_RESULTS`, paiement retenu	Le timeout se déclenche	`COMPENSATING` ; les commands `ReleaseStock` et `VoidAuthorization` sont écrits dans l'outbox

État de la saga	Événement qui arrive	Résultat attendu
`COMPENSATING` ou `CANCELLED`	`StockReserved` tardif	Aucune transition de la saga. Un `StockReserved` tardif signifie qu'Inventory a réservé avant que `ReleaseStock` ne lui parvienne, et ce command libère donc le stock. Si la libération était arrivée en premier, son tombstone aurait refusé la réservation (Chapitre 10)
`COMPLETED`	`PaymentCaptured` en double	Ignoré

La troisième ligne est celle qui compte. L'arrivée tardive après un timeout est le cas que les sagas traitent mal, car le happy path et le chemin du timeout ont chacun été testés seuls et la collision ne l'a jamais été. Le tableau doit être écrit par quelqu'un qui se demande « quoi d'autre pourrait arriver dans cet état ? » et non par quelqu'un qui suit le diagramme de flux.

> **Remarque :** Si vos tests reposent sur une planification aléatoire, conservez quelques graines en échec comme cas de régression fixes une fois qu'elles sont trouvées. Un bug reproduit une fois doit rester reproduit.

Tests de contrat et vérifications de compatibilité

Les tests de handler construisent leurs propres événements, ce qui signifie qu'ils testent le consommateur par rapport à l'idée que le consommateur se fait du producteur. Quand le producteur modifie un champ, rien dans la suite du consommateur ne le remarque. Le Chapitre 12 a décrit les règles de compatibilité et un schema registry qui les applique. Cela couvre la structure. Les tests de contrat couvrent la partie que le registry ne peut pas voir : ce dont chaque consommateur dépend réellement.

Dans un **contrat piloté par le consommateur**, chaque consommateur note les champs et les types qu'il lit dans un événement, et le pipeline du producteur vérifie que son schéma les fournit toujours. Le sens compte. Le producteur ne peut pas savoir quels champs sont porteurs, et le consommateur, si. Un contrat est donc un petit fichier qui appartient au consommateur et qui est stocké là où la CI du producteur peut le lire.

```
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)
```

Il faut deux moitiés. Le pipeline du producteur exécute `verifyContracts` contre chaque contrat enregistré. Le pipeline du consommateur décode chaque événement d'exemple de ses propres contrats avec son vrai désérialiseur et exécute le handler dessus. Ensemble, elles détectent un producteur qui cesse de fournir un champ et un consommateur qui se met à lire un champ qu'il n'a jamais déclaré. La seconde détection ne tient que si l'exemple est construit à partir du contrat et ne contient que les champs déclarés ; un exemple copié depuis l'événement complet du producteur laisse le consommateur lire un champ non déclaré et passer quand même.

Les exemples doivent couvrir chaque version encore en rétention, comme le recommandait le Chapitre 12. Le replay lit tout ce qui est là, et un consommateur qui ne gère que la forme la plus récente échoue sur le premier ancien événement. C'est aussi là que la vérification de compatibilité du registry a sa place dans la CI : un changement de schéma du producteur doit faire échouer le build avant de pouvoir être enregistré, pas après avoir atteint le topic.

Soyez honnête sur la limite. Un contrat prouve que `amount_minor` existe et est un entier. Il ne prouve pas qu'il signifie toujours « le montant en unités mineures ». Changer le sens sans changer la structure passe chaque test

de contrat, et seul un relecteur humain, ou un test comportemental dans le consommateur, le remarque. Je ne traiterais pas des contrats au vert comme une affirmation sur le sens.

Assertions en cohérence à terme

Un test de bout en bout publie quelque chose puis fait une assertion sur un effet ailleurs. L'effet finit par se produire ; l'assertion a donc besoin d'une forme qui le tolère. La forme standard est un polling avec une deadline.

```
function eventually(check, timeout, interval):
    deadline = now() + timeout
    while true:
        observation = check()
        if observation.satisfied:
            return
        if now() >= deadline:
            // report what was last seen, not only that time ran out
            raise AssertionTimeout(observation.description)
        sleep(interval)
```

La structure compte plus qu'il n'y paraît. La vérification s'exécute une fois de plus quand la deadline est atteinte, si bien qu'une condition remplie pendant le dernier sommeil passe quand même, l'intervalle est court par rapport au timeout, et le message d'échec porte la dernière observation. Un message qui dit « timed out » ne vous apprend rien. Un message qui dit « order-1 stuck at AWAITING_RESULTS, 1 held event » vous dit où regarder.

Les tests asynchrones flaky ont une courte liste de causes, et la plupart se corrigent à la cause.

- **Les sommeils fixes.** Remplacez-les par `eventually`. Le timeout devient un plafond et le test se termine dès que la condition est vraie.
- **Des timeouts réglés pour un portable rapide.** Fixez le plafond généreusement, car il ne coûte rien quand la condition est remplie rapidement, et gardez-le séparé des attentes de performance. Un budget de latence est un autre test, dans un autre environnement.
- **L'état partagé entre tests.** Un message resté dans un topic partagé fait voir au test suivant un événement qu'il n'a pas publié. Donnez à

chaque test son propre correlation id et filtrez sur lui, ou son propre topic ou sa propre file d'attente.

- **Un ordre supposé là où aucun n'est promis.** Si un test fait une assertion sur l'ordre de deux événements provenant de partitions différentes, il vérifie quelque chose que le broker n'a jamais promis (Chapitre 9).
- **Vérifier que rien ne s'est produit.** « Aucun second e-mail n'a été envoyé » ne peut pas devenir un succès par polling, car attendre plus longtemps pourrait encore en faire apparaître un. Préférez amener le système à la quiescence, puis faire l'assertion. La quiescence est une propriété de tout le chemin, pas d'un seul consommateur : l'outbox est vide, le relay est inactif, le retard de chaque consommateur concerné est nul, aucun retry ni minuteur n'est planifié, et les files de retry et de délai sont vides. Ce n'est qu'alors qu'il ne reste rien qui puisse produire l'effet, y compris un saut en aval dont le message est encore en transit.

Pour le dernier cas, le harnais a besoin d'un moyen de dire « exécute jusqu'à ce que plus rien ne soit en attente ». Dans un test de handler ou de composant, le harnais peut pomper le consommateur jusqu'à ce que la file soit vidée et que l'horloge injectée n'ait plus de minuteurs à déclencher. C'est à la fois plus rapide et plus déterministe qu'attendre le temps d'horloge, et c'est la principale raison pour laquelle je garde la plupart des tests comportementaux à la couche des handlers et n'utilise le temps réel que là où c'est le câblage qui est testé.

Injection de défaillances et chaos

Tout ce qui précède teste du code isolément. La question suivante est de savoir si le système assemblé se comporte comme l'affirme le modèle de défaillance. L'injection de défaillances introduit délibérément une défaillance dans un environnement en marche et vérifie la réponse du système.

Partez du modèle de défaillance du Chapitre 3 et choisissez des entrées, pas des outils. Pour chacune, écrivez l'hypothèse dans les termes de l'entrée elle-même : « si un consommateur est tué entre le commit et l'ack, la nouvelle livraison ne change rien ». Puis injectez-la : tuez le processus à ce moment-là, mettez le relay en pause, retardez les réponses du broker, cou-

pez la connexion entre un service et sa base de données, redémarrez un nœud du broker en pleine publication. Ensuite, vérifiez les invariants, pas les logs : le stock correspond aux réservations, le registre a une écriture par capture, aucune ligne de l'outbox n'est bloquée.

Ajoutez à cette liste des exercices de reprise, car une défense jamais exercée est un espoir. Une sauvegarde jamais restaurée en est l'exemple classique. Restaurez-en une dans un environnement jetable et vérifiez ce qu'une restauration fait à l'outbox, à l'inbox et aux offsets des consommateurs, ce que traite le Chapitre 14 : des lignes d'outbox ressuscitées, des lignes d'inbox rembobinées si bien que les doublons ne sont plus reconnus, des offsets qui ne correspondent plus. Faites un failover vers un cluster de secours, ou répétez-le, et comparez le temps de reprise et les données perdues avec les objectifs de reprise choisis pour ce flux au Chapitre 14. Et testez le rattrapage : mettez un consommateur en pause assez longtemps pour accumuler un backlog, relancez-le pendant que le trafic continue d'arriver, et vérifiez que le backlog se résorbe avant la deadline du flux et avant l'expiration de la rétention. Cela vérifie le taux de vidange du Chapitre 13 et les limites sur le travail en cours du Chapitre 6.

J'exécuterais cela par couches de rayon d'impact croissant. Les tests de flux dans un environnement dédié injectent des défaillances selon un calendrier en CI, petites et reproductibles. Un game day périodique dans un environnement plus grand les injecte comme exercice d'équipe, en partie pour trouver des bugs et en partie pour savoir si quelqu'un remarque quoi que ce soit. Les alertes du Chapitre 13 sont tout autant testées que le code. Si vous avez tué un consommateur et qu'aucune alerte ne s'est déclenchée, vous avez trouvé un bug dans la supervision.

Le chaos en production est une étape suivante possible, pas un point de départ. Il n'a de sens que lorsque le système a montré qu'il survivait à la même défaillance en préproduction, que le rayon d'impact peut être borné et que quelqu'un peut arrêter l'expérience instantanément. Une équipe qui n'a pas tout cela ne devrait pas commencer par là, et je n'aurais pas honte de rester longtemps aux game days. La valeur d'une pratique d'injection de défaillances réside dans le cycle hypothèse, injection et vérification, pas dans le caractère spectaculaire de l'injection.

> **Avertissement :** Une injection de défaillances qui partage un broker, une base de données ou un compte chez le fournisseur de paiement avec du trafic réel est une panne avec un plan. Isolez l'environnement, ou limitez l'injection au trafic synthétique, avant d'éteindre quoi que ce soit.

Le replay comme test de régression

Un log d'événements est un enregistrement de ce qui s'est réellement passé, et le replay, la relecture délibérée de l'historique, le transforme en suite de régression. Prenez une tranche d'événements réels, faites-les passer par la nouvelle version d'un consommateur dans un environnement jetable, et comparez l'état dérivé avec ce que l'ancienne version a produit. Les différences sont soit voulues, comme pour une correction de bug, soit des régressions. Le replay a aussi un usage opérationnel, la reprise après un trou ou une restauration par relecture de l'historique, et le Chapitre 14 le traite. Cette section ne porte que sur son usage pour tester.

Cela détecte ce que les fixtures écrites à la main ne détecteront jamais : des formes d'événements d'il y a trois versions de schéma, des séquences que personne n'a imaginées, un champ toujours présent dans les tests et nul dans une petite fraction de l'historique. C'est aussi ce qui se rapproche le plus d'un test de l'affirmation du Chapitre 12 selon laquelle les anciens événements se décodent toujours.

Une procédure praticable comporte cinq étapes.

1. **Sélectionner une tranche.** Une fenêtre de temps, ou l'historique complet d'un échantillon d'agrégats. Un échantillon qui inclut chaque version de schéma, plus les incidents connus, vaut mieux qu'un échantillon aléatoire.
2. **La masquer.** Les événements dont on fait un replay contiennent des données d'acheteurs. Anonymisez-les avant qu'ils ne quittent leur périmètre d'origine, ou placez l'environnement jetable sous les mêmes contrôles que la production.
3. **Désactiver les effets et les événements émis.** Le consommateur qui fait un replay ne doit pas débiter de cartes, envoyer d'e-mails ni appeler de transporteurs. Les effets vont vers des stubs qui enregistrent ce

qu'on leur a demandé de faire, et les enregistrements des stubs font partie de la sortie. Son outbox et son relay doivent publier vers un broker jetable, ou être capturés, et ne jamais pointer vers les topics de production, sans quoi le replay republie l'historique dans les consommateurs réels.

4. **Exécuter l'ancienne et la nouvelle version sur la même tranche.** Comparez l'état dérivé et les événements émis, agrégat par agrégat.
5. **Classer chaque différence.** Voulue, inoffensive ou régression. Une différence que personne ne sait expliquer est une régression jusqu'à preuve du contraire.

Le replay s'applique à la famille des logs, où l'historique est encore là. Sur une file d'attente classique, les messages consommés ont disparu. L'équivalent est un échantillon capturé : une dérivation qui copie une fraction des messages, anonymisés, vers un stockage que vous contrôlez. C'est un outil plus faible, car la capture ne vaut que ce que vaut l'échantillonnage, mais il sert le même objectif.

Le replay a aussi une limite de coût. Les gros historiques demandent un temps et un stockage réels, et l'outillage de comparaison est quelque chose que votre équipe doit construire et maintenir en état de marche. Je réserverais le replay de l'historique complet aux changements qui touchent la façon dont les événements sont interprétés, et j'utiliserais une tranche plus petite pour les livraisons courantes.

Vérifier en production

Aucun environnement de préproduction ne reproduit la forme du trafic, les données, la configuration et les bizarreries de la production. Certaines défaillances, comme une permission manquante sur un topic après un déploiement, un secret périmé ou une file d'attente mal routée, n'apparaissent que dans le système réel. La réponse n'est pas de renoncer aux tests de préproduction, mais d'ajouter une dernière couche qui tourne là où se trouvent les défaillances.

Un **événement synthétique** est un élément de trafic injecté délibérément et explicitement étiqueté, qui parcourt le vrai pipeline et est vérifié à l'arrivée. Pour Parcelly, c'est une commande passée toutes les quelques minutes par une sonde, qui circule de `OrderPlaced` jusqu'à `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)
```

L'étiquette est le mécanisme de sécurité, et c'est un en-tête `synthetic` dédié, pas un préfixe sur `correlation_id`. Un identifiant qui contrôle aussi le comportement est un piège, et le Chapitre 13 demande un format d'identifiant uniforme, `order-81724`, pour que la recherche par corrélation fonctionne partout. Le prix est qu'un saut qui oublie de copier l'en-tête transforme une sonde en trafic réel sans la moindre erreur. Chaque service le copie avec le même utilitaire que celui qui copie `correlation_id`, la sonde utilise aussi un compte acheteur synthétique dédié que Payments et Shipping reconnaissent indépendamment, et un test supprime volontairement l'en-tête et vérifie qu'aucun débit ni aucune expédition réels n'en résultent.

Chaque service reconnaît le drapeau et change ses bords, pas sa logique centrale : Payments parle au sandbox du fournisseur, Shipping saute la remise au transporteur, Notifications supprime l'envoi, et Ledger comptabilise sur un compte de test que la finance exclut des rapports. Ces branches sont du code qui ne tourne que pour les sondes ; une erreur dans ces branches est donc un bug que seule la production expose. Inventory a besoin du même traitement, sans quoi chaque sonde réserve du stock réel : réservez sur un SKU synthétique dédié, ou sautez la réservation pour les commandes étiquetées. Je testerais les branches elles-mêmes à la couche des handlers, et je testerais la sonde en cassant volontairement quelque chose et en vérifiant qu'elle échoue.

Le nettoyage comporte un piège de plus. Une sonde qui dépasse son timeout n'a pas forcément échoué à terminer ; ses événements peuvent encore être en transit, et un `StockReserved` ou un `PaymentAuthorized` tardif peut arriver après l'annulation. C'est pourquoi la sonde est enregistrée d'emblée et chaque service se fonde sur le drapeau `synthetic` : un sweep périodique peut trouver les sondes plus anciennes que le budget et annuler tout ce qu'elles ont laissé derrière elles, et l'annulation est elle-même idempotente, si bien que l'exécuter deux fois est sans danger.

Une sonde a aussi besoin d'un budget, et le budget a besoin d'un responsable. Une sonde en échec déclenche l'astreinte, si bien que le budget doit être assez généreux pour éviter les fausses alertes et assez serré pour avoir un sens. Le Chapitre 13 traite de la latence de bout en bout et de la manière d'alerter dessus. La sonde est la source la moins coûteuse de cette mesure, car elle exerce tout le chemin, y compris les parties qu'aucune commande réelle n'a touchées au cours de la dernière heure.

La vérification en production inclut les jobs de réconciliation du Chapitre 13. Une exécution de réconciliation est un test d'invariant sur des données réelles, exécuté en continu, et elle détecte les défaillances que personne n'a pensé à simuler. Je la compte comme la dernière couche de la pyramide.

Compromis et coûts

Le réalisme coûte du temps. Les vraies bases de données, les vrais brokers et les défaillances injectées rendent les tests plus lents. La suite qui était rapide l'était parce qu'elle sautait ce qui comptait. Je budgéterais explicitement les couches lentes : tests de handler en secondes, tests de composant en quelques minutes, tests de flux et injection de défaillances selon un calendrier plutôt qu'à chaque commit.

Les tests de contrat ajoutent de la coordination. Chaque consommateur possède des fichiers qui vivent dans le pipeline du producteur, et quelqu'un doit les tenir à jour. Pour un système à deux services et une équipe, un test partagé du schéma peut suffire. Les tests de contrat commencent à rapporter quand les équipes productrices et consommatrices déploient indépendamment, et ils rapportent peu avant cela.

Les planifications aléatoires ajoutent du travail d'investigation. Une graine en échec est à la fois une énigme et un vrai défaut. Les équipes qui n'aiment pas cela finissent par réduire le nombre de graines jusqu'à ce que le test ne trouve plus rien. Le remède est de journaliser les graines et de conserver des régressions fixes, pas d'abandonner la technique.

Le chaos et les sondes ont un rayon d'impact. Les deux touchent des systèmes réels ou partagés. Quand ne pas les utiliser : quand le système ne survit pas encore à la défaillance isolément, quand personne ne peut arrêter l'exécution immédiatement, ou quand le trafic étiqueté ne peut pas être tenu à l'écart de l'argent réel et des acheteurs réels.

Le replay coûte cher à mettre en place. L'infrastructure de masquage, de stubs d'effets et de comparaison est un projet en soi. Un petit système avec un historique court tire davantage des tests de handler et des points de crash. Je ne sais pas où se situe votre seuil de rentabilité, car il dépend de la fréquence à laquelle les consommateurs changent et du coût d'une mauvaise interprétation de l'historique, mais une équipe qui a déjà eu un incident dû à un ancien événement inattendu peut généralement le deviner.

Erreurs courantes

- **Tester contre un fake de broker qui ne se comporte jamais mal.** C'est tentant, car il est rapide et les premiers tests passent. Faites dupliquer, retarder et réordonner le fake, ou mettez la défaillance dans les tests de handler par des appels explicites, comme dans ce chapitre.
- **Simuler les contraintes de la base de données.** Un fake écrit à la main n'applique ni l'unicité, ni l'isolation, ni l'atomicité, si bien que les tests passent que la défense existe ou non. Utilisez le vrai moteur, dans un conteneur ou une instance jetable, pour tout test qui dépend d'une contrainte.
- **Dormir pendant une durée fixe.** Cela ressemble à la façon la plus simple d'attendre. Faites du polling sur la condition avec un plafond et rapportez la dernière observation en cas d'échec.
- **Ne jamais tester le crash entre commit et ack.** Cela semble exotique, et pourtant cela arrive à chaque déploiement. Ajoutez un point de crash à chaque handler et un test qui l'arme.

- **Ne tester que l'état final, pas l'outbox.** Un doublon qui ne change pas l'état peut quand même émettre un second événement. Faites une assertion sur ce qui a été écrit dans l'outbox.
- **Des tests de contrat qui vérifient la présence mais pas le caractère obligatoire.** Un champ optionnel dans le schéma du producteur peut être absent. Faites dire « required » au contrat et vérifiez-le.
- **Des tests aléatoires sans graines.** Un échec qui ne peut pas être reproduit finit supprimé. Journalisez la graine et conservez les graines en échec comme cas fixes.
- **Faire confiance à une sauvegarde jamais restaurée.** Le job de sauvegarde signale un succès, si bien que la défense semble testée. Restaurez-la régulièrement dans un environnement jetable et vérifiez ensuite l'outbox, l'inbox et les offsets.
- **Faire tourner des sondes sur de l'argent réel.** Un trafic synthétique qui atteint le vrai fournisseur de paiement ou un vrai transporteur n'est pas une sonde. Routez le trafic étiqueté vers des sandboxes en bordure, et testez ce routage.

Points clés

- Les tests qui supposent un broker livrant chaque message une fois, dans l'ordre, testent un système qui n'existe pas ; chaque défense de ce livre a donc besoin d'un test qui applique la défaillance contre laquelle elle protège.
- Les tests de handler ont leur place sur un vrai moteur de base de données, car l'inbox, l'outbox et les vérifications de version dépendent de contraintes et de transactions que les fakes ne reproduisent pas.
- Les doublons, le réordonnancement et le délai se testent au mieux comme une propriété de convergence sur un stream perturbé, avec la graine journalisée pour que tout échec puisse être reproduit.
- Les contrats pilotés par le consommateur détectent les ruptures structurelles avant le déploiement, mais ils ne voient pas un changement de sens ; une revue humaine de ce qu'un champ est censé dire reste donc partie intégrante du processus.
- Remplacez les sommeils fixes par du polling sur une condition avec un plafond, faites vos assertions sur la quiescence plutôt que sur le si-

lence, et injectez l'horloge pour que les timeouts puissent être testés sans attendre.

- L'injection de défaillances doit partir des entrées du modèle de défaillance et vérifier les invariants ensuite, en n'élargissant le rayon d'impact qu'à mesure que le système prouve qu'il survit à chaque étape, et elle doit inclure des exercices de reprise pour les restaurations, les failovers et le rattrapage.
- Le replay de l'historique réel et les sondes synthétiques en production détectent ce que les fixtures écrites à la main et les environnements de préproduction ne peuvent pas détecter, et ils ont besoin de masquage, de stubs d'effets et de trafic étiqueté pour être sûrs.

Un système testé de cette manière mérite un peu plus de confiance, jamais une confiance totale, et cela laisse ouverte la question de ce qu'une équipe doit vérifier avant de faire confiance à une conception. Le Chapitre 16 rassemble le livre en une checklist de fiabilité, et se termine sur ce que la checklist ne peut pas faire.

Chapitre 16 — Une checklist de fiabilité et réflexions finales

Imaginez une revue de conception chez Parcelly, un jeudi tranquille. L'équipe s'apprête à ajouter un nouveau flux : les vendeurs pourront proposer des remboursements partiels, et Payments émettra un nouvel événement pour cela. Le diagramme est propre. Quelqu'un a tracé les flèches, nommé l'événement et listé les consommateurs. La réunion est prévue pour 30 minutes, et on dirait qu'elle se terminera en 20.

Puis un tech lead pose une question simple : « Que se passe-t-il si le remboursement est émis et que l'événement n'est jamais publié ? » La salle se tait, non pas parce que personne ne connaît le pattern, mais parce que personne ne l'avait demandé. L'outbox existe ailleurs dans le code. Le consommateur idempotent existe ailleurs aussi. Savoir si ce nouveau flux les utilise est une autre affaire, et le diagramme ne le dit pas.

C'est ainsi que la fiabilité s'érode d'ordinaire. L'équipe connaît les patterns. Ce qui s'efface, c'est l'habitude de vérifier, pour chaque nouveau flux, que chaque pattern est réellement en place. La question de ce dernier chapitre est donc pratique : que demanderiez-vous, flux par flux, pour savoir si le système peut tenir ses promesses un mauvais jour, et qui doit être responsable des réponses ?

Le problème, et pourquoi il est difficile

La fiabilité est une propriété des jonctions, et les jonctions ne sont visibles dans aucun fichier pris isolément. Un producteur peut être correct, un consommateur peut être correct, et le flux entre les deux peut quand même perdre des événements, parce que le passage de relais n'a jamais été examiné. Le Chapitre 1 a avancé cet argument au début du livre, et les chapitres suivants ont traité les jonctions une à une.

La difficulté, désormais, est que ce savoir est réparti sur 15 chapitres. Personne n'en garde autant en tête pendant une revue. Sous la pression d'une deadline, les questions sautées sont celles dont la réponse est « ça marche jusqu'au jour où ça ne marche plus » : la fenêtre de déduplication, le responsable des dead letters, la vérification de compatibilité, l'alerte qui se déclenche sur l'âge et non sur le nombre.

Une checklist est la réponse habituelle, et elle a des limites connues. Elle ne peut pas réfléchir. Quelqu'un qui ne comprend pas un item peut quand même le cocher, et une liste complétée donne alors une fausse assurance. J'utiliserais la liste ci-dessous comme point de départ d'une conversation, et non comme une barrière qu'un formulaire peut franchir. Chaque item renvoie au chapitre qui explique pourquoi il est là, si bien qu'un lecteur en désaccord peut retrouver le raisonnement et contester la prémisse.

Traitez chaque item comme une question sur un flux précis, comme le flux de commande de Parcelly, et non sur le système en général. « Nous utilisons un outbox » n'est pas une réponse. « Orders écrit `OrderPlaced` dans son outbox dans la même transaction que la ligne de commande » en est une.

La méthode derrière la liste

La liste n'est pas une idée nouvelle. C'est la thèse du livre appliquée à une revue : les systèmes orientés événements fiables se construisent autour de la **sémantique des défaillances**, et non des message brokers — pour une jonction donnée, ce qui peut échouer, comment vous l'observeriez et quelle garantie y survit.

Cela fait trois questions, et chaque item ci-dessous est l'une d'elles posée à propos d'une jonction. Un item qui nomme une défaillance demande ce qui peut échouer. Un item qui nomme une métrique, une alerte ou un test demande comment vous l'observeriez. Un item qui nomme un réglage, une contrainte ou un pattern demande quelle garantie y survit. Une jonction dont les items ne répondent qu'à l'une des trois est à moitié terminée, et une jonction sans réponse à la deuxième question échouera sans prévenir personne.

La checklist

La liste est regroupée selon l'endroit où vit la défaillance. La colonne de droite indique le chapitre à relire quand une réponse est « non » ou « je ne suis pas sûr ».

Producteur

Vérification	Chapitre
Le changement d'état et l'événement sont écrits dans une seule transaction locale, sans publication directe dans le chemin métier.	4
La ligne d'outbox porte l'enveloppe complète, et l'`event_id` est créé au moment de l'écriture et n'est jamais régénéré.	4
L'équipe a consigné par écrit que la publication est at-least-once et que les doublons en sont le prix.	4
Le relay publie d'abord et marque ensuite, et sélectionne par état, non par un curseur d'id croissant.	5
Plusieurs relays, s'il y en a, réservent les lignes de façon sûre et gardent les lignes d'un même agrégat sur un seul worker.	5
L'âge de la plus ancienne ligne non publiée est mesuré, et seules les lignes publiées sont purgées.	5

Si une équipe ne pouvait répondre qu'à une seule question en revue, je voudrais que ce soit la première, parce que toutes les garanties ultérieures supposent que l'événement existe tout court. Lorsqu'un flux n'a pas besoin d'outbox, parce que ses étapes peuvent partager une transaction locale, l'équipe doit le dire et dire pourquoi (Chapitre 1).

Broker

Vérification	Chapitre
Pour chaque événement important, l'équipe a consigné ce que « publié » signifie : les réglages de réplication et de flush derrière un ack du producteur.	2, 3
L'ack ou le commit d'offset a lieu après l'effet, de sorte que la sémantique est at-least-once et non un at-most-once accidentel.	2
Les visibility timeouts, les session timeouts et le temps de retry dans le handler sont réglés ensemble, afin qu'un travail lent ne soit pas relivré pendant qu'il s'exécute encore.	2, 6

Vérification	Chapitre
La clé de partition identifie l'agrégat, et l'équipe sait quel ordre offre sa famille de brokers.	9
Chaque producteur, dans chaque langage, utilise le même partitioner et le même encodage de clé, ou bien le relay calcule lui-même la partition.	9
La rétention dépasse la plus longue panne réaliste plus le temps nécessaire pour la remarquer.	3, 13
Les consommateurs s'arrêtent proprement lors d'un déploiement, afin que le rééquilibrage ne soit pas une source routinière de doublons.	3

La plupart de ces items sont des réglages, pas du code. Ils restent à leurs valeurs par défaut jusqu'à ce qu'un incident révèle ce qu'étaient ces valeurs, et à ce moment-là ils sont difficiles à retracer.

Consommateur

Vérification	Chapitre
La ligne d'inbox et le changement métier font leur commit dans une seule transaction locale, avec une contrainte d'unicité sur la clé.	7
La fenêtre de déduplication dure plus longtemps que la rétention, le redrive et le replay planifié, et une contrainte sur la clé métier couvre la table du domaine au-delà.	7
Les effets de bord externes portent une clé d'idempotence déterministe dérivée du travail, et non de la tentative.	7
Les défaillances sont classées comme transitoires, permanentes ou inconnues, et les retries utilisent un exponential backoff avec jitter, une deadline et un budget partagé.	6
Le travail en cours est plafonné, la concurrence baisse quand la dépendance ralentit, et le consommateur se met en pause au lieu d'échouer, si bien que la surcharge se manifeste sous forme de retard ; une limite à l'entrée rejette ou diffère le nouveau travail avec un signal clair pour l'appelant.	3, 6
Un poison message finit dans une DLQ avec le corps, les en-têtes, la position source, l'erreur, le nombre de tentatives et la version du consommateur, publié avant que l'original ne reçoive son ack.	8
Un consommateur bloqué sur un enregistrement est détecté, par un retard sans débit et par un compteur de crashs par position, et ne tourne pas en boucle jusqu'à ce que quelqu'un s'en aperçoive.	8, 13

Vérification	Chapitre
Le redrive préserve l'`event_id` d'origine, s'exécute par lots à débit limité et dispose d'un interrupteur de pause.	8
L'ordre par agrégat est protégé par des lanes à clé, des vérifications de version et des hold-back buffers dotés de timeouts.	9
Chaque étape de workflow avant le pivot a une compensation idempotente, et chaque attente a une deadline.	10

C'est le groupe le plus long, parce que la plupart des patterns du livre aboutissent ici. S'il paraît lourd, c'est exact : la livraison at-least-once fait porter le fardeau au consommateur.

Contrat

Vérification	Chapitre
Chaque événement énonce un fait, est nommé au passé et a un seul service propriétaire.	11
L'événement d'intégration est un contrat public délibéré, dérivé de l'événement de domaine, sans données personnelles ni secrets dans le payload.	11
Une commande est nommée à l'impératif, a un seul destinataire prévu, porte un `command_id`, dispose d'un chemin de réponse pour qu'un refus parvienne à l'émetteur, et n'est jamais appelée événement ; l'émetteur attend avec une deadline qui lui est propre.	10, 11
`correlation_id` est recopié tout au long du flux et `causation_id` pointe vers le déclencheur direct.	11
Les types sur le fil de l'enveloppe, comme les horodatages, les identifiants et les montants, sont fixés pour chaque langage qui la lit ou l'écrit.	11
Chaque changement est classé comme compatible ou cassant, et la compatibilité est vérifiée dans le pipeline de déploiement.	12
Un changement de sens est relu par un humain, parce qu'aucun outil ne le détecte.	12
Les consommateurs ignorent les champs qu'ils n'utilisent pas et tolèrent les valeurs qu'ils n'ont jamais vues, un saut qui transmet un message préserve les champs inconnus, et un événement dont ils ne connaissent pas le `schema_version` est rejeté ou mis de côté.	12
Les changements cassants migrent comme une suite d'étapes non cassantes, avec des rollbacks planifiés dans le même document, et les consommateurs qui lisent encore l'ancienne version sont connus.	11, 12

Opérations

Vérification	Chapitre
Le contexte de trace et `correlation_id` survivent à chaque saut asynchrone, si bien que l'histoire d'une commande peut être reconstruite à partir de son identifiant.	13
Le retard du consommateur, l'âge des messages, l'âge de l'outbox, le taux de retry et les arrivées en DLQ sont mesurés, et les alertes se déclenchent sur l'âge et le taux.	13
Des jobs de réconciliation comparent régulièrement la source de vérité à l'état dérivé, et les réparations sont relues avant d'être exécutées.	13
Les invariants métier, comme « chaque paiement capturé a une écriture au registre », sont vérifiés directement.	13
Les handlers sont testés avec des doublons, du réordonnancement et du délai, et les tests de contrat s'exécutent en CI.	15
L'injection de défaillances et des tests de régression fondés sur le replay existent pour les flux qui comptent le plus.	15
Chaque stockage a un objectif de point de reprise (RPO) et un objectif de temps de reprise (RTO) déclarés, et l'équipe sait quel stockage revient en arrière dans le temps lors d'une restauration.	14
Une procédure de replay nomme les groupes de consommateurs, l'horodatage et sa marge, les effets de bord qu'elle supprime, et la fenêtre de déduplication dans laquelle chaque plage dont on fait un replay doit tenir.	7, 14
Un plan de migration de broker utilise le relay comme unique point de bascule, fait du dual-publish avec déduplication côté consommateur, et comporte des étapes de cutover et de rollback.	14
Les restaurations et les game days ont été répétés, et les runbooks ci-dessous ont été écrits à partir de ce que la répétition a montré.	14, 15
Un modèle de défaillances écrit nomme des responsables, des tests et des dates de revue.	3

Questions pour une revue de conception

La checklist vérifie une conception qui existe. Ces questions aident à en façonner une qui n'existe pas encore. Je les poserais dans cet ordre, parce que chaque réponse contraint la suivante.

1. **Quel est le fait, et qui en est le propriétaire ?** Si vous ne pouvez pas nommer l'événement au passé ni nommer un seul service propriétaire,

la conception n'est pas prête pour un broker (Chapitre 11).

2. **Faut-il seulement un événement ?** Si les étapes peuvent partager une transaction locale, ou si l'appelant a besoin d'une réponse immédiate, un appel synchrone peut être le meilleur outil (Chapitre 1).
3. **Que se passe-t-il entre le commit et la publication ?** Demandez la fenêtre de crash, pas le pattern. « Nous publions juste après le commit » est une perte qui attend un déploiement (Chapitre 4).
4. **Que se passe-t-il quand ce message arrive deux fois, et quand il arrive en retard ?** Ce sont des défaillances différentes avec des correctifs différents (chapitres 7 et 9).
5. **Quelle est la borne de « à terme » ?** Nommez une durée, une alerte et un état visible par l'utilisateur pour l'attente (Chapitre 1).
6. **Que se passe-t-il quand ce message ne peut jamais réussir ?** Demandez où il aboutit, qui le voit et comment il revient (Chapitre 8).
7. **Que se passe-t-il quand le consommateur, ou la dépendance derrière lui, est plus lent que le producteur ?** Demandez ce qui est plafonné, ce qui se met en pause et quel travail attend, et qui est responsable de ce choix (chapitres 3 et 6).
8. **Si une étape ultérieure échoue, qu'est-ce qui annule les précédentes ?** Nommez la compensation et le pivot après lequel plus rien n'est annulé (Chapitre 10).
9. **Comment changerons-nous ce contrat dans un an ?** Si personne n'y a réfléchi, la réponse sera improvisée pendant un incident (Chapitre 12).
10. **Comment saurons-nous qu'il est cassé, et comment saurons-nous qu'il est réparé ?** La détection et la vérification sont des questions distinctes (chapitres 13 et 15).
11. **Si un stockage, le broker ou une région est perdu, combien perdons-nous, et combien de temps avant d'être de retour ?** Demandez un RPO et un RTO par stockage, ainsi que le replay ou la restauration qui suit (Chapitre 14).
12. **Qui reçoit l'appel d'astreinte, et que fait-il en premier ?** Si la réponse est un nom, vous avez un responsable. Si c'est « l'équipe », peut-être pas.

Certaines n'ont pas de réponse nette un jour donné, et c'est normal. « Nous ne savons pas encore, et voici comment nous allons le découvrir » est une meilleure issue qu'une réponse assurée que personne n'a vérifiée.

Le volet organisationnel

Chaque pattern de ce livre a une moitié humaine. Un outbox a besoin de quelqu'un qui surveille l'âge de sa plus ancienne ligne. Une DLQ a besoin de quelqu'un qui la lit. Un schema registry a besoin de quelqu'un qui décide ce que « compatible » veut dire. Quand la moitié humaine manque, le pattern existe toujours sur le diagramme mais ne fait plus son travail.

Responsabilité

Je consignerais trois types de responsabilité.

- **La responsabilité des événements.** Chaque événement a un seul service et une seule équipe propriétaires, qui documentent ce que l'événement signifie et qui le consomme (Chapitre 11).
- **La responsabilité opérationnelle.** Chaque DLQ, parking lot, rapport de réconciliation et alerte a un responsable nommé. Une alerte dont personne n'est responsable est une notification, et les notifications sont ignorées.
- **Les flux inter-services.** Le flux de commande de Parcelly traverse six services. Une équipe doit être responsable du flux dans son ensemble, y compris de ses timeouts et de ses workflows bloqués, même quand aucun service pris isolément ne l'est (Chapitre 10).

Runbooks

Un runbook est ce que vous écrivez en supposant que le lecteur est fatigué et n'a pas vu le système depuis six mois. Pour chaque alerte, dites ce qu'elle signifie, ce qu'il faut regarder en premier, quelles actions sont sûres, lesquelles ne le sont pas, et qui appeler si les premières étapes n'aident pas.

Certains runbooks sont liés à une alerte, d'autres à un événement si rare que personne dans l'équipe ne l'a vécu récemment. Le Chapitre 14 ajoute ce second type. Sa section sur la répétition de la reprise liste huit ques-

tions auxquelles un runbook de reprise doit répondre, par écrit et non dans la tête de quelqu'un :

1. Quel stockage est touché, et lequel est revenu en arrière dans le temps ?
2. Quel est le point de reprise, et comment a-t-il été déterminé ?
3. Quels groupes de consommateurs doivent être réinitialisés, à quel horodatage et avec quelle marge ?
4. Quels effets doivent être supprimés pendant le replay, et par quel mécanisme ?
5. Chaque plage à relire se trouve-t-elle dans la fenêtre de déduplication de chaque consommateur touché ?
6. Qu'est-ce qui est mis en pause, dans quel ordre, et qui le reprend ?
7. Quelles réconciliations confirment que la reprise a fonctionné, et qui les lit ?
8. Comment annuler l'ensemble si les choses tournent mal ?

Le tableau liste les événements pour lesquels j'écrirais un runbook. La dernière colonne indique à quelles questions parmi les huit je m'attendrais à ce que chacun réponde. C'est un point de départ, et le système d'une équipe ajoutera ses propres questions.

Événement	L'action dangereuse	Questions
Un consommateur bloqué sur un poison message (Chapitre 8)	Sauter la position sans en garder trace, ou relever la limite de retries jusqu'à ce que le message passe	6, 7, 8
Un consommateur dont le retard approche la rétention (chapitres 13 et 14)	Laisser les messages expirer sans être lus, ou faire un reset vers l'avant sans que personne ne l'ait décidé	2, 3, 5, 7
Reset d'offset et replay (Chapitre 14)	Faire un replay avec les effets de bord actifs, ou au-delà de la fenêtre de déduplication	3, 4, 5, 6, 8
Redrive (Chapitre 8)	Renvoyer toute une DLQ d'un coup vers une dépendance qui peine encore	5, 6, 8
Réparation et backfill (Chapitre 13)	Exécuter un script non relu contre l'état de production	4, 7, 8

Événement	L'action dangereuse	Questions
Restauration d'une base de données depuis une sauvegarde (Chapitre 14)	Reprendre les consommateurs sans réinitialiser leurs offsets, ou réutiliser des identifiants que la séquence rembobinée distribue à nouveau	1, 2, 3, 4, 5, 6, 7, 8
Perte de données du broker ou failover (Chapitre 14)	Reprendre des groupes à des positions qui ne signifient plus ce qu'elles signifiaient	1, 2, 3, 5, 6, 7
Évacuation d'une région (Chapitre 14)	Promouvoir un standby sans savoir quel était son retard	1, 2, 3, 6, 7, 8
Cutover d'une migration de broker (Chapitre 14)	Déplacer les consommateurs avant que les copies issues du dual-publish soient dédupliquées, ou sans retour possible	3, 5, 6, 8

Deux d'entre eux méritent un soin particulier. Le redrive vient en premier, parce qu'un redrive négligent transforme une défaillance contenue en déluge (Chapitre 8). La réparation et le backfill viennent en second, parce qu'un script qui corrige un état dérivé est un changement en production et mérite une revue, un dry-run et une trace de ce qu'il a fait (Chapitre 13).

Astreinte

L'astreinte pour un système orienté événements diffère de l'astreinte pour un service requête-réponse sur un point qui compte : la défaillance est souvent silencieuse. Un service synchrone renvoie des erreurs et un graphique vire au rouge. Un système orienté événements peut ne rien renvoyer du tout, parce que le message est posé dans une file d'attente, en attente. Les alertes sur l'âge, le retard et le travail non traité sont le signal principal, et l'ingénieur d'astreinte a besoin d'un accès suffisant pour regarder dans la file d'attente, la DLQ et l'outbox sans attendre d'autorisation.

> **Remarque :** Entraînez-vous à la réparation avant d'en avoir besoin. Une équipe qui a déjà fait un redrive d'une DLQ de test, en journée, sous le regard d'un collègue, gérera le vrai cas autrement qu'une équipe qui a seulement lu le runbook. Le Chapitre 14 fait la même remarque pour les restaurations et les game days, et le Chapitre 15 explique comment concevoir les exercices.

Compromis et coûts

Une checklist a des coûts, et le premier est l'attention. Une liste de 43 items appliquée à chaque flux est trop longue, et une équipe qui les applique tous à un flux de notification à faibles enjeux conclura que le problème, c'est la checklist. Je pondérerais la liste par le coût de la défaillance. Un flux qui déplace de l'argent ou modifie le stock mérite la liste entière. Un e-mail de rappel mérite l'idempotence et un responsable de la DLQ, et peut-être pas grand-chose d'autre.

Le deuxième coût est la fausse assurance. Une liste complétée dit que quelqu'un a examiné chaque item, pas que le système est correct. Les chapitres 13, 14 et 15 existent parce que des conceptions bien réfléchies échouent quand même, d'une manière que leurs auteurs n'avaient pas imaginée. Traitez la checklist comme la couche de défense la moins chère, pas comme la dernière.

Le troisième coût est la dérive. Une liste écrite une fois devient le reflet de ce que l'équipe croyait à un moment donné. Revisitez-la après chaque incident, et supprimez les items qui ont cessé de mériter leur place. Une liste courte que l'on utilise vaut mieux qu'une liste longue que l'on signe.

Quand ne pas l'utiliser : n'utilisez pas la checklist pour remplacer le jugement, et ne l'exigez pas pour des prototypes jetables. Si un flux doit être supprimé le mois prochain, le coût de la liste complète dépasse le coût de la défaillance. Dites-le explicitement, et consignez la date à laquelle le prototype soit disparaît, soit fait l'objet d'une vraie revue.

Erreurs courantes

- **Traiter la checklist comme une barrière.** Une barrière peut s'auditer en comptant les coches, et c'est là la tentation. Le résultat est un formulaire rempli et une conception non discutée. Utilisez les items comme des questions, et consignez les réponses qui ont surpris quelqu'un.
- **Vérifier des patterns plutôt que des flux.** « Nous avons un outbox » donne l'impression d'avancer. Demandez si ce flux l'utilise, et nommez la table et la transaction.

- **Tout appliquer partout.** Cela paraît rigoureux, puis mène à la lassitude et à l'abandon. Pondérez la liste par le coût de la défaillance, et dites quels items vous sautez et pourquoi.
- **Remettre les items humains à plus tard.** Les responsables, les runbooks et les dates de revue semblent moins urgents que le code, et ce sont les premières choses qui manquent quand une alerte se déclenche la nuit. Attribuez-les pendant la revue de conception elle-même.
- **Ne jamais revisiter les réponses.** Un flux qui a passé la revue il y a un an a gagné depuis des consommateurs, des versions et des personnes. Mettez une date de revue sur les réponses, comme le Chapitre 3 le suggère pour le modèle de défaillances.
- **Croire que la liste est complète.** Votre système a des modes de défaillance que ce livre ne nomme pas. Ajoutez-les à votre propre liste quand vous les rencontrez.

Points clés

- La fiabilité s'érode plus souvent par des questions sautées que par des patterns inconnus, si bien qu'un ensemble fixe de questions posées à chaque nouveau flux protège une conception mieux que l'expertise présente dans la salle.
- Chaque item de la checklist est l'une des trois questions du livre, ce qui peut échouer, comment vous l'observeriez et quelle garantie y survit, posée à propos d'une jonction, si bien que chaque item doit recevoir une réponse pour un flux précis et non pour le système en général.
- Pondérez la liste par le coût de la défaillance : les flux qui déplacent de l'argent ou du stock méritent chaque item, et les flux à faibles enjeux méritent les quelques items qui empêchent les effets en double et la perte silencieuse.
- Chaque pattern de ce livre a une moitié humaine, si bien que les responsables, les runbooks, les destinataires des alertes et les dates de revue ont leur place dans la même revue de conception que le code.
- Les défaillances orientées événements sont silencieuses, si bien que l'astreinte dépend d'alertes sur l'âge, le retard et le travail non traité, et que les événements rares, une restauration, un replay, la perte d'une

région et une migration de broker, ont besoin de runbooks qui ont réellement été exécutés.

- Une checklist complétée montre que quelqu'un a réfléchi à chaque item, pas que le système est correct, et c'est pourquoi la réconciliation et les tests existent comme couches supplémentaires.
- La fiabilité n'est pas une fonctionnalité que l'on livre une fois, mais une discipline que l'on continue de pratiquer, par de petites questions, des réponses écrites et des réparations répétées avant d'en avoir besoin.

Parcelly ne devient jamais fiable comme une fonctionnalité devient terminée. Le système devient plus fiable chaque fois que quelqu'un demande ce qui se passe quand le message arrive deux fois, consigne la réponse et lui attribue un responsable. Aucun broker n'offre cela, et aucune bibliothèque ne le livre. C'est une habitude d'attention, entretenue par des personnes qui savent que chaque système qu'elles construisent sera un jour malmené par un réseau, un déploiement ou un collègue fatigué, et qui ont décidé de concevoir pour ce jour-là sans drame. Voilà ce que je vous demanderais de retenir de ce livre : non pas un ensemble de patterns, mais l'habitude de poser la question suivante.

Annexe: Glossaire et référence rapide des patterns

Cette annexe comporte deux parties. La Partie A est un glossaire des termes employés dans le livre, par ordre alphabétique, avec le chapitre où chaque terme est défini ou utilisé de la façon la plus complète. La Partie B est une référence rapide des patterns, présentée sous forme d'un tableau que vous pouvez parcourir pendant une revue de conception.

Aucune des deux parties ne remplace les chapitres. Une définition ici est un rappel, et une ligne du tableau est un renvoi. Le raisonnement, les coûts et les cas où un pattern ne devrait pas être utilisé se trouvent dans le chapitre indiqué dans la dernière colonne.

A. Glossaire

ack (acknowledgment). Le signal par lequel le consommateur indique au broker qu'un message a été traité et peut être retiré de la file d'attente, ou que la position du consommateur peut avancer. La forme négative est un **nack**. Chapitre 2.

âge du message. Le temps écoulé depuis la production du plus ancien message encore en attente, comparé à une deadline métier et à la période de rétention. Chapitre 13.

agrégat. La frontière de cohérence qui possède une clé de partition. Les événements d'un même agrégat sont ordonnés les uns par rapport aux autres ; les événements d'agrégats différents ne le sont pas. Chapitre 9.

at-least-once. Une sémantique de livraison dans laquelle un message est livré une ou plusieurs fois, si bien que des doublons sont possibles. Elle tient tant que le message est encore conservé et que le consommateur ne fait l'ack qu'après l'effet. Un consommateur dont le retard dépasse la rétention perd quand même des messages en silence. C'est le choix par défaut du livre. Le Chapitre 2 la définit ; le Chapitre 3 traite de la perte.

at-most-once. Une sémantique de livraison dans laquelle un message est livré zéro ou une fois, si bien qu'un message peut être perdu mais n'est jamais livré deux fois. Chapitre 2.

backfill. Réparation ou reconstruction en masse qui injecte un grand ensemble d'événements dans un consommateur actif, exécutée par petits lots à débit limité. Chapitre 13.

backpressure. Le fait de borner le travail en cours à chaque étape afin que la charge reste finie. Dans un système qui passe par un broker, elle apparaît sous forme de limites sur l'admission d'un consommateur, comme le prefetch ou le nombre d'enregistrements par poll, et sous forme de retard que le broker absorbe. La pression n'atteint un producteur que par le contrôle de flux du broker ou par un throttling explicite. Le Chapitre 3 la place dans le modèle de défaillances ; le Chapitre 6 en donne les mécanismes.

causation_id. Le champ de l'enveloppe qui pointe vers l'événement ou le command qui a directement causé celui-ci, là où `correlation_id` nomme le flux entier. Chapitre 11.

change data capture (CDC). La lecture du journal de transactions d'une base de données pour transformer les changements dont le commit est fait en messages. Dans le contexte de l'outbox, un relay fondé sur le CDC lit les lignes d'outbox dans le journal au lieu de faire du polling sur la table. Chapitre 5.

circuit breaker. Une protection autour d'une dépendance qui arrête les appels lorsque les défaillances franchissent un seuil, puis laisse passer des appels d'essai après une attente. Chapitre 6.

clé de partition. L'identifiant d'agrégat utilisé pour router les événements vers une partition et ainsi préserver l'ordre par agrégat. Chapitre 9.

clé d'idempotence. La clé qu'un consommateur ou une API utilise pour reconnaître un travail répété. Pour la consommation d'événements, c'est normalement l'`event_id`. Chapitre 7.

clé métier. Un identifiant dérivé du sens du travail, comme un identifiant de paiement associé à une opération, plutôt que de l'enveloppe. Elle protège contre l'arrivée du même travail dans deux messages différents. Chapitre 7.

cohérence à terme. La propriété selon laquelle des répliques ou des services convergent vers le même état si les mises à jour s'arrêtent, sans borne sur le temps que cela prend. Chapitre 1.

command. Une demande de faire quelque chose, nommée à l'impératif, comme `CapturePayment`. Un command est adressé à un seul service et peut être refusé. Ce n'est pas un événement. Chapitre 11.

commit d'offset. L'enregistrement par lequel le consommateur indique, sur un broker à log, qu'il a traité tout ce qui précède une position. Il remplace l'ack par message. Chapitre 2.

compatibilité vers l'arrière. Un consommateur qui utilise le nouveau schéma peut lire des événements écrits avec l'ancien schéma : nouveaux lecteurs, anciennes données. Chapitre 12.

compatibilité vers l'avant. Un consommateur qui utilise l'ancien schéma peut lire des événements écrits avec le nouveau schéma : anciens lecteurs, nouvelles données. Chapitre 12.

compensation. L'étape qui annule sémantiquement une étape antérieure d'une saga, comme libérer du stock ou annuler une autorisation. Le livre ne l'appelle pas un rollback. Chapitre 10.

consommateur. Un processus qui lit des messages depuis un topic ou une file d'attente et agit en conséquence. Chapitre 2.

consommateur idempotent. Un consommateur dont le traitement répété d'un même message a le même effet qu'un traitement unique. Chapitre 7.

consommateurs concurrents. Plusieurs consommateurs qui partagent une même file d'attente, de sorte que chaque livraison va à l'un d'eux. Un message relivré peut atteindre un autre consommateur. C'est l'équivalent, dans la famille des files, d'un groupe de consommateurs. Chapitre 2.

contexte de trace. Le trace id et le span id qui situent une unité de travail dans une trace distribuée, capturés dans la ligne d'outbox et envoyés dans les en-têtes du message afin que le span du consommateur puisse se lier à celui du producteur. Chapitre 13.

contrat piloté par le consommateur. Un fichier dont un consommateur est propriétaire, qui liste les champs et les types qu'il lit dans un événement, et qui est vérifié dans le pipeline du producteur. Chapitre 15.

correlation_id. Le champ de l'enveloppe partagé par tous les événements d'un même flux métier. Son compagnon `causation_id` pointe vers l'événement ou le command qui a directement causé celui-ci. Chapitre 11.

cutover. L'étape d'une migration de broker au cours de laquelle un groupe de consommateurs commence à lire depuis le nouveau broker, franchie un groupe à la fois après la réconciliation des deux brokers et avec un rollback planifié. Chapitre 14.

dead letter. Un message mis de côté parce que son traitement a échoué. Employé comme modificateur, le terme prend un trait d'union : dead-letter queue. Chapitre 8.

dead-letter queue (DLQ). Une destination séparée où un consommateur ou un broker place les messages auxquels il a renoncé. Sur un broker à log, la même idée est un **dead-letter topic**. Chapitre 8.

deadline. Le moment après lequel un résultat n'est plus utile. Les retries doivent s'arrêter avant. Chapitre 6.

délai de grâce. L'âge qu'un enregistrement doit atteindre avant qu'un job de réconciliation ne le compare, afin que les événements encore en transit ne soient pas signalés comme des écarts. Chapitre 13.

différé. La classe de retry pour une défaillance qui ne dit rien sur le message, parce que la dépendance est connue pour être indisponible, comme lorsqu'un circuit breaker est ouvert. Un message différé ne consomme aucune tentative et n'est jamais envoyé vers la DLQ pour cette raison. Le consommateur se met en pause et le message attend. Chapitre 6.

doublon. Un message traité ou livré plus d'une fois. Chapitre 2.

dual write. La mise à jour de deux systèmes, comme une base de données et un broker, sans transaction partagée. Chapitre 4.

dual-publish. La publication du même fait sous deux formes ou vers deux destinations pendant une migration : les deux champs ou les deux topics pendant un changement de schéma (Chapitre 12), les deux brokers pendant une migration de broker (Chapitre 14). Dans une migration de broker, le relay conserve un second marqueur pour la nouvelle cible, afin que l'ancien broker reste complet et que le nouveau puisse être réconcilié avec lui.

enveloppe. L'ensemble standard de champs qui accompagne chaque événement, comme `event_id`, `event_type`, `schema_version`, `occurred_at`, `correlation_id` et `causation_id`, séparé du payload. Chapitre 11.

étape pivot. Le point d'une saga après lequel le workflow ne compense plus et ne fait plus qu'avancer. Chapitre 10.

événement. Un message dont le contenu est un fait déjà survenu, nommé au passé. Chapitre 11.

événement d'intégration. Un événement publié comme un contrat public délibéré, dérivé d'un événement de domaine et exempt de données personnelles et de secrets. Chapitre 11.

événement synthétique. Du trafic étiqueté injecté dans le pipeline actif par une sonde et vérifié en fin de flux. Chapitre 15.

event_id. Le champ de l'enveloppe qui identifie un événement. Il est créé lors de l'écriture de la ligne d'outbox et n'est jamais régénéré, et c'est la clé d'idempotence habituelle pour la consommation. Chapitres 4 et 7.

évolution des schémas. Le fait de changer la forme des événements au fil du temps sans casser les lecteurs et les écrivains existants. Chapitre 12.

exactly-once. Employé dans le livre uniquement comme un effet qualifié : livraison at-least-once plus un consommateur idempotent, ce que le livre appelle aussi effectively-once. Ce n'est pas une garantie de livraison. Le traitement exactly-once d'un broker est réel à l'intérieur de la frontière de ce broker et s'y arrête. Chapitre 2.

expand and contract. Un pattern de migration qui transforme un changement cassant en plusieurs étapes compatibles : prendre en charge les deux formes, migrer chaque lecteur, puis retirer l'ancienne forme. Chapitre 12.

failover. Une bascule prévue vers une autre copie d'une partition de broker, d'une base de données ou d'un cluster quand la copie active tombe en panne. Elle peut perdre ce qui n'avait pas encore été répliqué. Chapitres 3 et 14.

fat event. Un événement qui porte assez d'état pour que les consommateurs puissent agir sans rappeler le producteur. L'opposé est un **thin event**. Chapitre 11.

fenêtre de déduplication. La période pendant laquelle un consommateur se souvient des ids traités. Un message dont la ligne d'inbox a été supprimée peut être traité à nouveau. Chapitre 7.

file d'attente. Une structure de broker dans laquelle chaque message est consommé et reçoit son ack individuellement. Chapitre 2.

groupe de consommateurs. Un ensemble de consommateurs qui se partagent les partitions d'un topic, de sorte que chaque partition est lue par un seul membre à la fois. Chapitre 2.

head-of-line blocking. Un enregistrement qu'un consommateur refuse de sauter bloque tous les enregistrements situés derrière lui dans la même partition ou le même stream. Chapitre 8.

hold-back buffer. Une table durable dans laquelle un consommateur retient un événement dont le prédécesseur n'est pas arrivé, et le libère quand le trou se comble ou qu'un timeout expire. Chapitre 9.

inbox. La table des ids de messages traités côté consommateur, écrite dans la même transaction que les effets du handler. Chapitre 7.

jitter. Un aléa ajouté au délai d'un retry afin que de nombreux clients ne fassent pas leurs retries au même rythme. « Exponential backoff avec jitter » est l'expression consacrée. Chapitre 6.

journal d'événements. Un enregistrement append-only des enveloppes d'événements, sans payloads, tenu par un service à côté de son inbox afin de pouvoir répondre à des questions après que le broker a oublié les événements. Chapitre 13.

link (trace). Une référence d'un span du consommateur vers le contexte de trace du producteur, utilisée à la place d'une relation parent-enfant quand la livraison est tardive, répétée ou faite par lots. Chapitre 13.

log d'événements. La séquence durable et append-only des événements. Chapitre 2.

message. Une unité de données envoyée à travers un broker : un événement, un command ou un document. Chapitre 11.

message broker. L'infrastructure qui transporte les messages entre les services. Deux familles apparaissent dans le livre : à base de log et à base de file d'attente. Chapitre 2.

mise en miroir. La copie de messages entre brokers à l'aide d'un outil : un outil de mise en miroir entre deux logs, ou un pont ou un connecteur entre un log et une file d'attente. Elle transporte aussi l'historique antérieur à une migration et peut alimenter un cluster en standby, et les offsets de la copie lui sont propres. Chapitre 14.

modèle de défaillances. La liste écrite des jonctions d'un système et, pour chacune, ce qui peut échouer, comment on l'observerait et quelle garantie y survit, avec des responsables, des tests et des dates de revue. Chapitre 3.

nouvelle livraison. Le fait que le broker livre un message une nouvelle fois, par exemple après un ack manqué ou un visibility timeout expiré. Le replay est une autre chose. Chapitre 2.

objectif de point de reprise (RPO). La quantité de données que vous pouvez vous permettre de perdre, mesurée en temps. Une sauvegarde toutes les six heures signifie un RPO pouvant aller jusqu'à six heures, et chaque stockage a le sien. Chapitre 14.

objectif de temps de reprise (RTO). La durée pendant laquelle vous pouvez vous passer du service. Pour une projection dont on fait un replay, c'est un calcul de vidange. Chapitre 14.

offset. La position d'un consommateur dans une partition d'un log d'événements. Chapitre 2.

outbox. La table écrite dans la même transaction locale que le changement d'état, et depuis laquelle un relay publie les événements. Le terme complet est **transactional outbox**. Chapitre 4.

outbox du consommateur. Un outbox écrit par un consommateur qui publie un message de suite, dans la même transaction que sa ligne d'inbox et son changement d'état. Un crash avant le commit ne laisse rien derrière lui, et un crash après est absorbé par l'inbox lors de la nouvelle livraison. Le Chapitre 7 introduit la règle ; le Chapitre 10 l'applique aux étapes de saga.

parking lot. Un lieu de stockage pour les dead letters qui nécessitent une décision humaine, distinct de la DLQ elle-même. Chapitre 8.

partition. Une unité d'un topic à base de log, ordonnée et lue par un seul consommateur d'un groupe à la fois. Chapitres 2 et 9.

poison message. Un message qui échoue chaque fois qu'il est traité. Sur un broker à log, un poison message qui fait planter le processus laisse le consommateur bloqué sur une position ; il est donc détecté par un retard sans débit et par un compteur par position. Chapitre 8.

producteur. Un processus ou un service qui publie des messages, et le propriétaire des événements qu'il émet. Chapitre 11.

réconciliation. La comparaison des visions de la vérité de deux systèmes et la réparation des différences. Chapitre 13.

redrive. Le fait de rendre les dead letters au consommateur qui a échoué sur elles, sur sa propre entrée et jamais sur le topic partagé d'où elles venaient. La clé et l'`event_id` sont conservés, le compteur de tentatives repart de zéro, et `redrive_count` enregistre le tour. Chapitre 8.

rééquilibrage. La réattribution des partitions quand des membres rejoignent ou quittent un groupe de consommateurs, ou quand un membre est jugé ne plus répondre. Deux consommateurs peuvent brièvement travailler sur la même partition. Chapitre 3.

relay. Le processus qui déplace les lignes d'outbox vers le broker. Chapitre 5.

replay. La relecture délibérée de l'historique depuis un log d'événements. Le Chapitre 2 le définit ; les chapitres 7, 12, 13, 14 et 15 l'utilisent.

reset d'offset. Le déplacement de la position dont le commit est fait d'un groupe de consommateurs, généralement vers l'arrière jusqu'à un horodatage avec une marge, afin que l'historique soit relu. Les offsets appartiennent à un cluster et ne se transposent pas à un autre. Chapitre 14.

retard du consommateur. La distance qui sépare un consommateur de la tête d'une partition ou d'une file d'attente, mesurée en messages ou en temps. Le Chapitre 2 le définit ; les chapitres 3 et 13 l'utilisent.

rétention. La durée pendant laquelle un broker conserve un message, qu'il ait été lu ou non : une limite de temps ou de taille sur un log, une expiration sur une file d'attente. Elle borne le replay, le redrive et la durée pendant laquelle un consommateur en retard peut rester arrêté. Le Chapitre 2 la définit ; les chapitres 3, 13 et 14 l'utilisent.

retry budget. Une limite sur les retries exprimée en part du trafic global plutôt que par appel. Chapitre 6.

retry storm. Une vague de retries qui ajoute de la charge à une dépendance en difficulté et la maintient à terre. Chapitre 6.

révision du registry. L'identifiant qu'un schema registry attribue à chaque schéma enregistré. Elle change à chaque modification enregistrée, y compris les ajouts, et c'est un nombre différent de la **version du schéma** et de la **version de l'agrégat**. Chapitre 12.

saga. Un workflow de longue durée composé de transactions locales avec des compensations, soit orchestré par un coordinateur, soit chorégraphié au moyen d'événements. Chapitre 10.

schema registry. Un service qui stocke les schémas, attribue des identifiants et vérifie un schéma proposé par rapport aux précédents. Chapitre 12.

sémantique des défaillances. Pour une jonction donnée, ce qui peut échouer, comment vous l'observeriez et quelle garantie y survit. C'est la thèse du livre. La Préface l'énonce, le Chapitre 3 l'applique sous forme de modèle de défaillances, et le Chapitre 16 la transforme en checklist de revue.

tolerant reader. Un consommateur qui ne prend dans un événement que les champs dont il a besoin et ignore le reste. Chapitre 12.

tombstone. La clé, la dernière version et un marqueur de suppression qu'un consommateur conserve après avoir supprimé la ligne d'un agrégat, afin qu'un événement tardif ne puisse pas la recréer. Chapitre 9.

topic. Un stream nommé de messages, dans l'une ou l'autre famille de brokers. Chapitre 2.

upcaster. Du code qui convertit un événement écrit selon un ancien schéma vers le schéma actuel avant que la logique métier ne le voie. Chapitre 12.

vérification de version. La comparaison, par un consommateur, de la version contenue dans un événement avec la dernière version qu'il a appliquée pour cet agrégat, afin d'ignorer les événements périmés et de détecter les trous. Chapitre 9.

vérification d'invariants. Une requête planifiée portant sur une règle que le métier tient pour vraie, comme le fait que chaque paiement capturé ait exactement une écriture dans Ledger. Chaque ligne renvoyée est un écart. Chapitre 13.

verrou sémantique. Un indicateur de statut dans les données qui signale qu'une saga travaille sur un enregistrement, et que le reste du code est écrit pour respecter. Ce n'est pas un verrou de base de données. Chapitre 10.

version de l'agrégat. Un compteur des changements d'un agrégat, stocké dans la ligne d'outbox et porté dans le payload, comme `order_version` pour Orders. Il ordonne les événements d'un agrégat et ne dit rien de la forme du payload. C'est un nombre différent de la **version du schéma** et de la **révision du registry**. Le Chapitre 4 la définit ; le Chapitre 9 l'utilise.

version du schéma. L'entier `schema_version` dans l'enveloppe, que le Chapitre 12 appelle aussi la version cassante. Elle ne change que pour une modification qu'un ancien lecteur ne peut pas gérer, et c'est un nombre différent de la **révision du registry** et de la **version de l'agrégat**. Chapitre 12.

visibility timeout. Sur un broker à files d'attente, le temps pendant lequel un message livré reste masqué avant d'être relivré s'il n'a pas reçu d'ack. Chapitre 2.

B. Référence rapide des patterns

Chaque ligne nomme la défaillance que le pattern traite, dans les termes du livre, et ce qu'il vous coûte. Un pattern déplace une défaillance vers un endroit où vous pouvez la gérer ; il ne supprime pas la défaillance. Le Chapitre 16 transforme ces lignes en checklist de revue.

Pattern	Défaillance traitée	Coûts	Chapitre
Transactional outbox	Dual write entre une base de données et un broker	Une table supplémentaire, un relay à exploiter, des doublons reportés sur les consommateurs	4
Relay par polling	Déplacer les lignes d'outbox vers le broker avec peu d'infrastructure	Délai de polling, charge sur la base de données, conception soignée du curseur	5
Change data capture	Publier les lignes d'outbox depuis le journal de la base de données avec un faible délai	Poids opérationnel, failover du connecteur, rétention du journal à gérer	5
Exponential backoff avec jitter	Des retries qui surchargent une dépendance en cours de rétablissement	Latence supplémentaire, réglage des délais et des limites de tentatives	6
Retry budget	Des retries qui amplifient une panne	Certains appels échouent vite alors qu'ils auraient pu réussir	6
Backpressure et contrôle de flux	Une dépendance ou un consommateur qui reçoit plus de travail qu'il ne peut en absorber	Retard, consommateurs en pause et travail rejeté, une décision métier sur ce qui attend	6
Circuit breaker	Des appels répétés à une dépendance défaillante	Un état à régler ; des appels rejetés tant que le breaker est ouvert	6
Retry topic ou file à délai	Des délais de retry trop longs pour être attendus dans le handler	Plus de topics, plus de pièces mobiles, un ordre plus difficile	6
Consommateur idempotent avec inbox	Les doublons issus de la livraison at-least-once	Des écritures en table à chaque message, une rétention à gérer	7
Clé métier	Le même travail qui arrive dans des messages différents	Une conception de clé par opération, et des contraintes d'unicité	7
Clé d'idempotence transmise en aval	Des effets de bord externes en double	Nécessite la prise en charge par le service en aval	7
Dead-letter queue	Des poison messages coincés dans une boucle de nouvelles livraisons	Une pile qui a besoin d'un responsable, d'alarmes et d'un chemin de réparation	8
Parking lot	Des dead letters qui nécessitent une décision humaine	Du travail manuel et une règle de responsabilité claire	8

Pattern	Défaillance traitée	Coûts	Chapitre
Redrive	Récupérer les dead letters une fois la cause corrigée	Doublons, réordonnancement et charge sur le flux principal	8
Outbox du consommateur	Un consommateur qui change d'état et publie un message de suite, en perdant l'un des deux	Un outbox côté consommateur et un relay à exploiter	7, 10
Clé de partition par agrégat	Des événements désordonnés au sein d'un même agrégat	Partitions chaudes, limites au parallélisme	9
Vérification de version côté consommateur	Des événements périmés ou désordonnés qui écrasent un état plus récent	Un champ de version et une décision pour chaque trou	9
Hold-back buffer	Des trous dans une séquence qui doit être appliquée dans l'ordre	Mémoire, timeouts et le trou qui ne se comble jamais	9
Saga (orchestrée ou chorégraphiée)	Un workflow métier qui traverse plusieurs services sans transaction unique	Des états, des timeouts et des compensations à concevoir et à exploiter	10
Compensation	Annuler des étapes de saga terminées après une défaillance ultérieure	Certaines étapes ne peuvent pas être annulées ; l'étape pivot marque la limite	10
Verrou sémantique	Des modifications concurrentes d'un enregistrement sur lequel une saga travaille	Des opérations bloquées pendant l'exécution de la saga	10
Types sur le fil fixés par champ entre langages	Le même champ de l'enveloppe lu différemment par deux runtimes	De la discipline dans chaque langage et des tests qui traversent les langages	11
Enveloppe d'événement standard	Des événements que l'outillage et les consommateurs ne peuvent ni tracer ni router	De la discipline chez chaque producteur	11
Thin event ou fat event	Choisir entre rappels et état transporté	Thin : couplage au moment de la lecture. Fat : contrat plus large et données copiées	11
Schema registry avec vérifications de compatibilité	Des producteurs qui cassent les consommateurs en silence	Un service à exploiter et un processus à suivre	12
Tolerant reader et upcaster	Des lecteurs qui cassent face à de nouveaux champs ou d'anciennes versions	Du code de conversion conservé pour chaque version historique	12

Pattern	Défaillance traitée	Coûts	Chapitre
Expand and contract	Un changement cassant qui ne peut pas être évité	Plusieurs déploiements, du dual-publish, une migration plus longue	12
Contexte de trace et correlation id	Perdre l'histoire d'un flux à travers des sauts asynchrones	Une propagation dans chaque producteur et chaque consommateur	13
Réconciliation et réparation	Une divergence silencieuse entre un système source et l'état dérivé	Des jobs de comparaison, un outillage de réparation et la relecture des résultats	13
Journal d'événements	Des questions sur des événements que le broker a oubliés	Une table écrite à chaque message et une rétention à gérer	13
Backfill	Un état dérivé qui doit être reconstruit ou complété à partir des données existantes	Des lots à débit limité et un interrupteur de pause, et la charge sur un consommateur actif	13
Vérifications d'invariants	Une casse au niveau métier qu'aucune métrique technique ne montre	Définir les invariants et être responsable des alertes	13
Replay par reset d'offset	Un consommateur qui s'est trompé, ou un stockage revenu en arrière dans le temps, et qui a de nouveau besoin de l'historique	Une fenêtre de déduplication qui couvre la plage, des effets de bord supprimés, et la rétention comme limite	14
Objectifs de point et de temps de reprise par stockage	Des sinistres gérés par l'improvisation, sans perte ni indisponibilité convenues	Un chiffre à mesurer et à répéter pour chaque stockage	14
Réplication inter-clusters pour la reprise	La perte d'un cluster de brokers ou d'une région	Un standby à exploiter, un retard de réplication qui constitue le RPO, et des offsets qui ne se transposent pas	14
Dual-publish et cutover	Passer à un autre broker sans perdre d'événements ni dupliquer d'effets	Un second marqueur dans l'outbox, de la réconciliation et une migration plus lente	14
Tests de contrats pilotés par le consommateur	La dérive des contrats entre équipes	L'entretien des tests et un accord sur qui les exécute	15

Pattern	Défaillance traitée	Coûts	Chapitre
Tests de handlers et de perturbation	Des handlers jamais éprouvés face aux doublons, au délai ou au réordonnancement	Une infrastructure de test sur une vraie base de données et des suites plus lentes	15
Injection de défaillances	Des entrées du modèle de défaillances jamais éprouvées dans un système en marche	Un environnement dédié, un rayon d'impact à borner	15
Test de régression fondé sur le replay	Des changements qui cassent le comportement sur l'historique réel	Des données d'échantillon sélectionnées et anonymisées à entretenir	15
Événements synthétiques en production	Des défaillances silencieuses sur le chemin actif	Des données de test qui doivent être isolées des vraies données métier	15

Comment lire le tableau

Beaucoup de lignes vont par paires : un pattern qui corrige une défaillance et un second pattern qui corrige la défaillance introduite par le premier. L'outbox crée des doublons, donc le consommateur a besoin d'un inbox. Les retries créent de la charge, donc ils ont besoin d'un budget. Une dead-letter queue crée une pile, donc elle a besoin d'un responsable et d'une procédure de redrive.

Cet enchaînement est la forme même du livre. Si vous adoptez une ligne et sautez sa partenaire, la défaillance s'est déplacée mais n'a pas disparu. À mon avis, le tableau est le plus utile lu de droite à gauche : partez de la défaillance que votre système a aujourd'hui, trouvez la ligne qui la nomme, et lisez la colonne des coûts avant de lire le chapitre.

À PROPOS DU LIVRE

Les systèmes orientés événements fiables se construisent autour de la sémantique des défaillances, et non des message brokers.

À chaque jonction, là où un composant confie du travail à un autre, le livre pose les trois mêmes questions : ce qui peut échouer, comment vous l'observeriez et quelle garantie y survit. La livraison at-least-once est traitée comme un fait plutôt que comme un défaut ; les handlers sont donc idempotents, et les doublons, les messages perdus, les retries et l'arrivée dans le désordre sont prévus à la conception au lieu d'être espérés disparus. En suivant Parcelly, le livre parcourt le transactional outbox, les relays et le change data capture, les retries, l'idempotence, les dead letters, l'ordre, les sagas, l'évolution des schémas, l'observabilité et la reprise, chacun avec le coût qu'il comporte.

À PROPOS DE L'AUTEUR

Muhammet Şafak développe des logiciels depuis 2008. Pendant de longues années, Muhammet a occupé des rôles actifs dans les domaines de la santé, de la finance et des télécommunications, et défend l'idée que le développement doit partir du problème, non de la technologie. Depuis 2024, Muhammet conçoit et développe des systèmes à fort trafic capables de monter en charge de bout en bout.

Muhammet accompagne en mentorat des équipes de développement logiciel de petite et moyenne taille, mène des projets open source et consigne ses décisions par écrit, avec leurs raisons et leur coût.

Les systèmes fiables ne se construisent pas avec des composants fiables. Ils se construisent autour des composants qui défaillent.

I ACCÈS

II ÉVÉNEMENTS

III AGENTS