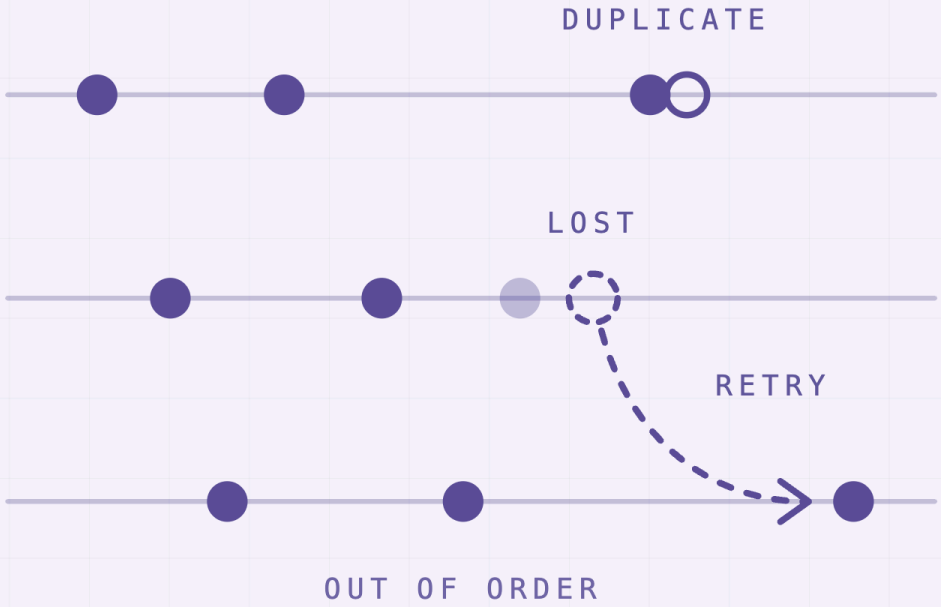




Güvenilir Event-Driven Sistemler Tasarlamak

Güvenilir Event-Driven Sistemler Tasarlamak

Mesajlaşma, retry, idempotency, outbox, ordering ve schema evolution

Muhammet Şafak

İçindekiler

Önsöz	5
KISIM I — TEMELLER	12
Bölüm 1 — Happy Path'in Yalanı	13
Bölüm 2 — Mesajlaşma Semantiği	25
Bölüm 3 — Pratik Bir Hata Modeli	42
KISIM II — GÜVENİLİR ŞEKİLDE YAYINLAMAK	60
Bölüm 4 — Dual-Write Problemi ve Transactional Outbox	61
Bölüm 5 — Relay'ler, Polling ve Change Data Capture	80
KISIM III — GÜVENİLİR ŞEKİLDE TÜKETMEK	100
Bölüm 6 — Retry'ı Doğru Yapmak	101
Bölüm 7 — Idempotency	125
Bölüm 8 — Poison Message'lar ve Dead Letter'lar	147
Bölüm 9 — Ordering	169
KISIM IV — İŞ AKIŞLARI VE KONTRATLAR	189
Bölüm 10 — Saga'lar ve Uzun Süren İş Akışları	190
Bölüm 11 — Event Tasarımı	211
Bölüm 12 — Schema Evolution	230
KISIM V — İŞLETMEK VE KANITLAMAK	251
Bölüm 13 — Observability ve Reconciliation	252
Bölüm 14 — Recovery, Felaketler ve Broker Migration	274
Bölüm 15 — Event-Driven Sistemleri Test Etmek	298
Bölüm 16 — Bir Güvenilirlik Kontrol Listesi ve Kapanış Düşünceleri	315
Ek: Sözlük ve Örüntülere Hızlı Başvuru	327

Önsöz

Parcelly'de bir salı öğleden sonrasını düşünün; var olmayan bir pazar yeri bu. Checkout akışı aylardır sağlıklı çalışıyor. Bir alıcı sipariş veriyor, Orders servisi siparişi veritabanına yazıyor ve message broker'a bir `OrderPlaced` event'i gidiyor. Payments kartı yetkilendiriyor. Inventory stoku rezerve ediyor. Shipping paketi planlıyor. Bütün dashboard'lar yeşil.

Sonra Orders'ta bir deploy ilerlemeye başlıyor. Her instance bir siparişi aynı şekilde işliyor: siparişi veritabanına commit ediyor, ardından `OrderPlaced`'i message broker'a yayınlıyor. Birkaç saniye boyunca bazı instance'lar kapanırken diğerleri ayağa kalkıyor. Bir instance bir siparişi commit ediyor ve publish çağrısından önce durduruluyor. Sipariş güvenli saklanmış durumda, ama event hiçbir zaman dışarı çıkmıyor. Hiçbir şey gürültüyle çökmüyor, hiçbir alert tetiklenmiyor. Ertesi sabah bir alıcı verilmiş görünen bir siparişe bakıyor; oysa Payments hiçbir kartı yetkilendirmemiş, Inventory tek bir ürünü bile rezerve etmemiş. Orders işin yürüdüğünü sanıyor, sistemin geri kalanı ise bundan hiç haberdar değil.

Kimse alışıldık anlamda bir bug yazmadı. Her kod satırı söylediğini yaptı. Sistem satırların arasındaki boşlukta bozuldu: bir veritabanı commit'i ile bir ağ çağrısı arasında, bir mesajın teslim edilmesi ile anlaşılması arasında, "gönderdik" ile "onlar yaptı" arasında.

Bu kitap o boşlukla ilgili.

Bu kitap neden var

Event-driven mimarinin bir itibar sorunu var ve bunu iki taraftan da hak ediyor. Savunucuları, bağımsız ölçeklenen ve koordinasyon olmadan evrilen, gevşek bağlı servisleri anlatıyor. Eleştirmenleri ise bir sipariş verildiğinde ne olduğunu kimsenin söyleyemediği dağınık bir spaghettiyi anlatıyor. İkisi de aynı sistemi farklı günlerde tarif ediyor.

İki günü birbirinden ayıran şey nadiren broker seçimi, framework ya da mimari diyagram oluyor. Ayıran şey, gösterişsiz sorulardan oluşan kısa bir liste. Bu mesaj iki kez teslim edilirse ne olur? Hiç gelmezse ne olur? Kendisini izlemesi gereken mesajdan sonra gelirse ne olur? Consumer her seferinde hata verirse ne olur? Producer bir alanı değiştirmek istediğinde yüz consumer hâlâ eski kodu çalıştırıyorsa ne olur?

Bu sorular bu kitabın tezidir. Güvenilir event-driven sistemler message broker'ların değil, **failure semantics**'in etrafında kurulur: belirli bir birleşim yeri için neyin başarısız olabileceği, bunu nasıl gözlemleyeceğiniz ve hangi garantinin ayakta kalacağı. Birleşim yeri, bir bileşenin işi başka bir bileşene devrettiği her yerdir; örneğin bir veritabanı commit'inin ardından gelen bir publish ya da bir teslimatın ardından gelen bir etki. Buradaki örüntüler ağ ve veritabanı yüzünden var, herhangi bir ürün yüzünden değil; broker da kendi birleşim yerleri olan bir bileşen daha.

Ekipler bu sorularla genellikle tek tek, production'da, sistemin bozulduğu sırayla karşılaşır. Her cevap ayrı ayrı öğrenilir ve çıkarılan dersler nadiren tek bir yerde toplanır. Bir broker'ın dokümantasyonu size nasıl publish ve consume edeceğinizi söyler. Bu iki fiilin çevresinde doğru bir servisin neye benzediği konusunda ise çok daha sessizdir.

Bu kitap o yer olmaya çalışıyor. Event-driven sistemleri güvenilir kılan örüntülere dair bir rehber: transactional outbox, idempotent consumer'lar, backoff ve jitter ile retry'lar, backpressure, dead-letter queue'lar, ordering stratejileri, saga'lar, özenli event tasarımı, schema evolution, observability, kaybolan veriden ve broker değişiminden kurtarma ve test. Her birini, uyulacak bir kural olarak değil, bedeli olan bir karar olarak ele almaya çalıştım. Parcelly'nin ödemeleri için doğru olan bir örüntü, bildirimleri için israf olabilir; önemli olduğunda kitap bunu söyleyecek.

Kitabın ne olmadığı konusunda da açık olmak istiyorum. Belirli bir broker için bir tutorial değil. Şimdiye kadar adı konmuş her enterprise integration pattern'ın kataloğu da değil. Kitabı izlemenin sisteminizi doğru kılacağı sözünü vermiyor. Güvenilirlik sonradan eklenen bir özellik değildir; nasıl ele alacağınıza bilerek karar verdiğiniz bir hata kümesi ile bilerek kabul etmeye karar verdiğiniz başka bir hata kümesidir. Amacım bu kararları görünür kılmak.

Bu kitap kimin için

Kitap, event'ler üzerinden haberleşen servisleri inşa eden ya da işleten **backend ve platform mühendisleri** ile bu servislerin işler ters gittiğinde nasıl davranması gerektiğine karar vermek zorunda olan **tech lead'ler** için yazıldı. Gece boyunca büyüyen bir dead-letter queue'ya baktıysanız, bir consumer'ın idempotent olması gerekip gerekmediğini tartıştıysanız ya da event'lerini kimsenin tam olarak açıklayamadığı bir topic'i devraldıysanız, aklımdaki okur sizsiniz.

Mesajlaşma konusunda uzman olmanız gerekmiyor. Şunlara hâkim olmanız yeterli:

- İlişkisel veritabanları ve transaction'lar: commit'in ne olduğu, bir unique constraint'in ne yaptığı ve isolation'ın kabaca ne anlama geldiği.
- Ağ üzerinden konuşan servisler ve timeout ile HTTP tarzı request ve response'un temel kelime dağarcığı.
- Message broker'ın genel fikri: producer'lardan mesaj kabul edip bunları consumer'lara ileten bir şey.

Belirli bir ürün deneyimine ihtiyacınız yok. Bir bölüm Kafka, RabbitMQ ya da Amazon SQS'ten söz ettiğinde, bunu bir sistem ailesinin örneği olarak yapar ve belirli bir sürümde değil, genel olarak geçerli olan davranışı anlatır. Hiçbirini kullanmadıysanız kitap yine de anlam taşıyacaktır. Birini derinlemesine kullandıysanız, bildiklerinizin bir kısmının o ürüne özgü olduğunu göreceksiniz; kitap hangi kısımların olduğunu işaretlemeye çalışacak.

Her gün kod yazmayan bir tech lead iseniz, her bölümün açılış senaryosunu, trade-off bölümünü ve temel çıkarımlarını okuyarak karar verme değerinin büyük kısmını alabilirsiniz. Pseudocode, transaction sınırının tam olarak nerede durduğunu görmesi gerekenler için orada.

Kitap nasıl düzenlendi

Kitap beş kısımdan oluşuyor; ilk okumada sırayla okunmaları, sonrasında ise herhangi bir sırayla kullanılmaları amaçlanıyor.

Kısım I, Temeller (Bölüm 1 ila 3). Happy path'i parçalarına ayırarak başlıyoruz. Bölüm 1, basit görünen bir akışın nasıl bir düzine kararı gizlediğini gösteriyor. Bölüm 2, mesajlaşma garantilerinin gerçekte ne anlama geldiğini ve at-least-once delivery'nin neden gerçekçi taban olduğunu netleştiriyor. Bölüm 3, şeylerin bozulma biçimleri için, timeout'lardan ve çökmelerden hep başarısız olan mesajlara ve bütün bir depolamanın kaybına kadar, küçük bir kelime dağarcığı olan pratik bir hata modeli kuruyor. Kitabın geri kalanı bunu ortak dili olarak kullanıyor.

Kısım II, Güvenilir Şekilde Yayınlamak (Bölüm 4 ve 5). Bölüm 4, açılış sahnesindeki tam boşluk olan dual-write problemini ele alıyor ve transactional outbox'ı tanıtıyor. Bölüm 5, outbox satırlarını broker'a taşıyan mekanizmayı, polling relay'leri change data capture ile karşılaştırarak ve her birinin size neye mal olduğunu göstererek işliyor.

Kısım III, Güvenilir Şekilde Tüketmek (Bölüm 6 ila 9). Bu en uzun kısım, çünkü tekrarlanan etkilerin, takılan queue'ların ve kaybolan güncellemelerin çoğu consumer'larda yaşanıyor. Bölüm 6 retry'ları ele alıyor: ne zaman retry edilir, ne zaman edilmez, backoff'un neden jitter'a ihtiyacı olur ve bir consumer'ın kendi retry'ları yüzünden aşırı yüklenmesi nasıl önlenir. Bölüm 7 idempotency ve inbox üzerine. Bölüm 8 poison message'lar ve dead letter'larla ilgileniyor. Bölüm 9 ordering'e ve genellikle ondan ne kadar az şey talep etmeyi göze alabileceğinize dönüyor.

Kısım IV, İş Akışları ve Kontratlar (Bölüm 10 ila 12). Bölüm 10, servislere yayılan ve bir transaction paylaşamayan iş akışları için saga'ları tanıtıyor. Bölüm 11, event tasarımının kendisiyle ilgili: içlerine ne girer, neler dışarıda kalır, nasıl adlandırılırlar ve bir envelope'un birkaç dilde yazılıp okunurken nasıl ayakta kaldığı. Bölüm 12, producer'ların ve consumer'ların farklı hızlarda değişebilmesini sağlayan sessiz disiplin olan schema evolution'ı ele alıyor.

Kısım V, İşletmek ve Kanıtlamak (Bölüm 13 ila 16). Bölüm 13, sisteminizin ne yaptığını görmekle ve reconciliation ile, yani bir sistemin görüşünü bir diğerkinkiyle karşılaştırma alışkanlığıyla ilgili. Bölüm 14, veri kaybından, bir region kaybından ve broker değişiminden sonra sistemin nasıl kurtarılacağını anlatıyor. Bölüm 15, en kötü bug'ları yalnızca duplication, gecikme ve reordering altında ortaya çıkan kodun nasıl test edileceğini işliyor. Bölüm 16 her şeyi bir güvenilirlik kontrol listesinde topluyor ve kitabı kapatıyor.

Ek, bir sözlük ve örüntüler için bir hızlı başvuru içeriyor; böylece bir bölümü yeniden okumadan bir şeye bakabilirsiniz.

1'den 15'e kadar her bölüm aynı biçimi izliyor. Kısa bir senaryoyla açılıyor, problemin neden zor olduğunu açıklıyor, örüntüyü sunuyor ve ardından en değerli bulduğum iki bölüme geçiyor: örüntüyü ne zaman kullanmamanız gerektiğini söyleyen **Trade-off'lar ve maliyetler** ile **Sık yapılan hatalar**. Her bölüm beş ila yedi temel çıkarımla bitiyor. Vaktiniz kısıtlıysa, bu üç kısım sizi beklediğinizden daha ileri götürecektir.

Bölümleri seçerek okuyabilirsiniz. Bugünkü derdiniz büyüyen bir dead-letter queue ise doğrudan Bölüm 8'e gidin ve gerektikçe referansları izleyerek Bölüm 6 ve 7'ye dönün. Yeni bir servis tasarlıyorsanız, publish koduna ilk satırı yazmadan önce Kısım II'yi okumanızı öneririm, çünkü outbox'la başlamak, sonradan eklemekten çok daha kolaydır.

Parcelly ile tanışın

Kitap boyunca örnekleri tek bir kurgusal sistem taşıyor: bağımsız satıcıların mal listelediği ve alıcıların sipariş verdiği çevrimiçi bir pazar yeri olan **Parcelly**. Altı servisi var.

- **Orders** sipariş yaşam döngüsünün sahibi ve iş akışlarının çoğunun başladığı yer.
- **Payments** parayı yetkilendirir ve tahsil eder, harici bir ödeme sağlayıcısıyla konuşur.
- **Inventory** stoku takip eder ve geçici rezervasyonları tutar.
- **Shipping** sevkiyatları planlar ve paketleri kargo firmalarına teslim eder.
- **Notifications** e-posta, push mesajı ve SMS gönderir; başka hiçbir şeyi asla bloke etmemelidir.
- **Ledger** para hareketlerinin yalnızca eklemeli, çift girişli kaydını tutar.

Her servis kendi ilişkisel veritabanına sahip ve kimsenin veritabanını okumuyor. Bir message broker üzerinden, `OrderPlaced`, `PaymentAuthorized`, `StockReserved`, `PaymentFailed` ve `ShipmentScheduled` gibi adlara sahip event'ler değiş tokuş ederek

haberleşiyorlar. Event'ler tutarlı bir biçim izliyor; bir tanımlayıcı, bir tür, bir schema sürümü, correlation alanları ve bir payload taşıyan bir envelope. Böylece bölümler onlardan tek bir ağızdan söz edebiliyor.

Bilerek bir pazar yeri seçtim. Para almak, mal rezerve etmek ve paket taşımak; kayıp bir mesajın ya da tekrarlanan bir etkinin maliyetinin çabucak hissedildiği bir alan. Tekrarlanan bir e-posta can sıkıcıdır. Tekrarlanan bir capture ise bir iade, bir destek talebi ve bir özürdür. Kaybolan bir rezervasyon, fazla satılmış bir üründür. Zararsızdan pahalıya uzanan bu eğim, her akışın ne kadar güvenilirlik mekanizmasını hak ettiğine karar vermek için tam olarak ihtiyaç duyduğunuz şey.

Parcellary kurgudur, olayları da öyle. Her bölümü açan senaryolar bir hata biçimini göstermek için uydurulmuştur ve her zaman kurgu olarak çerçevelenir. Birinci tekil şahsı yargı için kullanacağım; örneğin "burada bir unique constraint eklerdim" ya da "bence bu, karmaşıklığa değmez" gibi, ama akıl yürütmeyi savaş hikâyeleri gibi süslemeyeceğim. Kitap bir şeyin olduğunu söylediğinde bunun bir anekdottan değil, bu sistemlerin çalışma biçiminden kaynaklandığına güvenebilmelisiniz.

Kitap ayrıca bilerek **broker'dan bağımsız**. Bir noktanın broker'ınızın log-based olup olmamasına bağlı olduğu yerlerde, yani consumer'ların sıralı bir partition'da bir offset'i takip ettiği ya da mesajların tek tek ack'lendiği ve redelivery edildiği queue-based bir yapıda, hangisi olduğunu söyleyeceğim. Önemli olmadığı yerlerde aksini iddia etmeyeceğim.

Kodu nasıl okumalı

Bu kitaptaki kod **pseudocode**'dur. Hiçbir dile ait değil, derlenmez ve içindeki hiçbir şey gerçek bir kütüphaneyi çağırır. Bu bilinçli. Her kod parçasının amacı, bir transaction'ın nerede başlayıp bittiğini, bir mesajın nerede ack'lendiğini ve bir adım yarıda başarısız olduğunda ne olduğunu göstermek. Bu olgular her dilde aynı, sözdizimi ise onların yoluna çıkmaktan başka bir işe yaramaz.

Bloklar kısa, en fazla 30 satır. Bir tablo schema'sının işe yaradığı yerlerde SQL benzeri bir notasyonla yazılmış. Diyagramlar Mermaid kullanıyor ve her birini, neye bakmanız gerektiğini söyleyen bir cümle tanıtıyor. Bir

örüntüyü kendi stack'inize çeviriyorsanız, pseudocode'u bir implementasyon olarak değil, ordering ve atomicity için bir şartname olarak ele alın.

Kesinlik üzerine bir not

Bu kitapta önerilerde bulunacağım ve bazılarının arkasında sıkı duracağım. Örneğin, bir event'in kaybı iş için önemliyse event'leri düz bir dual write ile yayınlamam. Diğer öneriler zevk ya da bağlam meselesi ve hangisinin hangisi olduğunu söylemeye çalışacağım.

Dürüst cevabın "duruma bağlı; neye bağlı olduğu da şu" olduğu yerler de olacak. Retry budget'ları, dead-letter saklama süresi, partition sayıları ve doğru ordering düzeyi, işiniz ve ekibiniz hakkında göremediğim olgulara bağlı. Bu yerlerde bir sayı uydurmak yerine değişkenlerin adını vereceğim. Bir bölümü bitirdiğinizde kendi sisteminizde hangi soruları soracağınızı biliyorsanız bunu başarı sayacağım.

Güvenilirlik işinin talihsiz bir özelliği var: başarılı olduğunda görünür hiçbir şey olmaz. Siparişler karşılanır, defterler tutar ve kimse size teşekkür etmek için mesaj yazmaz. Umarım bu kitap, o türden bir sessizliği kurmanıza yardımcı olur.

Happy path ile ve ona güvenmemek için nedenlerle başlayalım.

KISIM I

Temeller

Bölüm 1 — Happy Path'in Yalanı

Parcelly'yi ilk yılında düşünün. Orders, Payments'ı HTTP üzerinden çağırıyor, cevabı bekliyor, sonra Inventory'yi çağırıp yine bekliyor, ardından Shipping'i çağırıyor. Kod yukarıdan aşağı okunuyor, hata bir exception ve bir hata sayfası gören alıcı tekrar deneyebiliyor. Ödeme sağlayıcısı yavaşsa yavaşlıyor, Inventory kapalıysa çöküyor; ama ne olduğu konusunda kimsenin kafası karışık değil.

Sonra ekip event'lere geçmeye karar veriyor. Orders `OrderPlaced`'i bir message broker'a publish edecek, diğer servisler de kendi zamanlarında buna tepki verecek. Demo harika. Bir alıcı "Siparişi ver"e tıklıyor ve bir saniye içinde stok rezerve ediliyor, kart tahsil ediliyor, sipariş onaylanıyor ve bir sevkiyat planlanıyor. Herkes alkışlıyor.

Şimdi production'daki ilk ayı gözünüzün önüne getirin. Bir alıcının kartından para çekiliyor, ama sipariş sonsuza dek "pending"de kalıyor. Bir başka sipariş iki kez kargolanıyor. Üçüncüsünün, eşleşen bir ödemesi olmayan bir ledger kaydı var ve hangi servisin haklı olduğunu kimse söyleyemiyor. Destek ekibi "81724 numaralı siparişe ne oldu?" diye soruyor ve dürüst cevap, dört servisin her birinin bunun bir parçasını bildiği, hiçbirinin hepsini bilmediği.

Demoda yanlış olan hiçbir şey yoktu. Eksikti, çünkü demo, her adımın başarılı olduğu bir hikâyledir. Bu bölümün cevapladığı soru, ekibin request-response'tan ayrılırken neyi bıraktığı ve kimse onu yeniden inşa etmeye başlamadan önce "güvenilir"in ne anlama geldiği.

Problem ve neden zor olduğu

Happy path, her bileşen ayaktaiken, her ağ çağrısı ulaşırken, her mesaj bir kez ve sırasıyla teslim edilirken ve her consumer başladığı işi bitirirken gerçekleşen event dizisidir. Beyaz tahtaya çizdiğiniz yol ve bir demonun işlettiği yoldur. Aynı zamanda aylarca çalışan bir sistemde en nadir görülen yoldur.

Hatalar egzotik değil. Deploy'lar process'leri yeniden başlatır. Bağlantılar bir yazma işleminin ortasında kopar. Bir broker bir replica'ya failover yapar. Bir consumer beklenenden uzun sürer ve mesajı başkasına verilir. Bunların her biri yoğun bir sistemde her gün bir yerlerde olur. Kötü bir production ayının olağan biçimi, bir avuç sıradan olayın happy path için yazılmış kodla karşılaşmasıdır.

Bunu zorlaştıran şey, event-driven sistemlerin hataları görünür kılan şeyi ortadan kaldırmasıdır. Senkron bir çağrıda çağırılan taraf bekler ve sonuç aynı bağlantı üzerinden döner: bir sonuç ya da bir hata, onu doğuran kodun yanında. Event-driven bir tasarımda producer bir mesajı broker'a verir ve yoluna devam eder. Üç servis ve on saniye sonra bir şey ters giderse producer çoktan gitmiştir ve kanıt henüz kimsenin ilişkilendirmediği log'lara dağılmıştır.

Bariz tepki, her parçayı güvenilir kılmak: iyi bir broker, iyi test edilmiş servisler, her çağrıda retry. Her biri yardımcı olur, hiçbirisi yeterli değildir, çünkü canınızı yakan hatalar bileşenlerin arasındaki boşluklarda yaşar. Veritabanına yazılan ama hiç publish edilmeyen ya da işlenen ama hiç ack'lenmeyen bir mesaj, iki bileşenden birindeki bir bug değildir. Bir birleşim yerinin özelliğidir. Bu kitabın büyük kısmı birleşim yerleri hakkındadır.

Her birleşim yeri için kitap aynı üç soruyu sorar: ne başarısız olabilir, bunu nasıl gözlemlersiniz ve hangi garanti ayakta kalır. Cevaplara birleşim yerinin **failure semantics**'i diyeceğim. Bir broker'ın dokümantasyonu size broker'ın kendi birleşim yerlerinin semantiğini verir. Çevresindeki birleşim yerlerini, yani veritabanınız ile publish çağrınız ya da bir teslimat ile etkisi arasındakileri çözmek size düşer ve Bölüm 3 bunun nasıl yapılacağını gösteriyor.

Senkron ve asenkron coupling

Event'lere geçiş genellikle "decoupling" olarak anlatılır; teslim ettiğinden fazlasını vaat eden bir sözcük. Coupling'in biçim değiştirdiğini söylemek daha doğrudur.

Senkron bir çağrı iki servisi **zaman**, **availability** ve **bilgi** bakımından birbirine bağlar. Çağrılan taraf şu an çalışıyor olmalıdır, çağıranın latency'si çağrılanınkini içerir ve çağırılan, çağrılanın kim olduğunu ve ne

cevap vereceğini bilir. Faydası basit bir hata modelidir: çağırın çağırının başarısız olduğunu bilir ve hemen retry etmeye, fallback'e geçmeye ya da hata bildirmeye karar verebilir. Maliyeti ise zincirin en zayıf halkası kadar erişilebilir olmasıdır. Yavaş bir kargo API'si yavaş bir checkout'a dönüşür.

Bir broker üzerinden giden mesaj, zaman ve availability coupling'ini ortadan kaldırır. Inventory yeniden başlarken Orders bir sipariş kabul edebilir, çünkü broker mesajı Inventory dönene kadar tutar. Bir analitik servisi gibi yeni bir consumer, Orders değişmeden dinlemeye başlayabilir. Bu kazanç gerçektir ve bu tarzı benimsemenin nedeni de budur.

Ama coupling yok olmadı, yer değiştirdi. Servisler artık **event'in kontratı** üzerinden birbirine bağlı: adı, alanları ve anlamı. **Broker'ın davranışı** üzerinden bağlılar: durability, ordering ve delivery konusunda ne vaat ettiği. Ve birbirlerinin zamanlamasına dair kimsenin yazıya dökmediği varsayımlar üzerinden bağlılar. Bölüm 11 ve Bölüm 12 kontratı, Bölüm 2 ise broker'ı ele alıyor.

Kısacası, senkron sistemler gürültüyle ve zincir hâlinde bozular. Asenkron sistemler sessizce ve parça parça bozular.

Temporal decoupling'in bedeli

Temporal decoupling, producer ile consumer'ın aynı anda aktif olmak zorunda olmaması demektir. Event'lerle satın aldığınız ana şey budur; dolayısıyla bedeli konusunda kesin olmaya değer.

Producer artık sonucu öğrenmiyor

Orders, Payments'ı senkron çağırdığında, alıcıya cevap vermeden önce ödemenin işe yarayıp yaramadığını öğreniyordu. Event'lerle Orders `OrderPlaced`'i publish eder ve "alındı" diye cevap verir. Ödemenin başarılı olup olmadığı sonraya kalan bir sorudur ve cevap, gelirse, başka bir event olarak gelir. Sipariş artık bir ara duruma, "verildi ama henüz onaylanmadı" durumuna sahiptir ve sistemin, bu durum sürdüğü sürece alıcının ne göreceğine ve planlanandan uzun sürerse ne olacağına karar vermesi gerekir.

Kimse akışın tamamına sahip değil

Senkron sürümde diziye tek bir fonksiyon barındırıyordu ve iş sürecini tek bir yerde okuyabiliyordunuz. Event-driven sürümde süreç, servislere dağılmış tepkilerin toplamıdır. Payments OrderPlaced'e tepki verir, Orders PaymentAuthorized ve StockReserved'a tepki verir, Shipping OrderConfirmed'a tepki verir. Akış yalnızca ortaya çıkan bir özellik olarak vardır ve kimse nerede tanımlandığını gösteremez.

Her şey çalışırken bu sorun değildir, bir şey çalışmadığında pahalıdır; çünkü "bu sipariş nerede takıldı?" sorusunun bakılacak tek bir yeri yoktur. Bölüm 10 buna saga'larla geri dönüyor, Bölüm 13 ise soruyu cevaplayabilmek için neleri gözlemlemeniz gerektiğini ele alıyor.

Hatalar exception olmaktan çıkıyor

Bir çağrı zincirinde bir hata, bir şey onu ele alana kadar stack boyunca yukarı yayılır. Bir mesaj akışında stack yoktur. Başarısız olan bir consumer mesajı ack'leyip kaybedebilir, reddedip yeniden teslim ettirebilir ya da bir insan için kenara ayırabilir. Her biri, hatayı hiç görmeyen servisler için sonuçları olan bir politika karardır. Bence senkron bir sistemde hata yönetimi bir kontrol akışı meselesi, asenkron bir sistemde ise bir tasarım meselesidir; çünkü yönetim zamana, process'lere ve bazen ekiplere yayılır.

Zaman bir değişken oluyor

"Publish edildi" ile "işlendi" arasında, siz bir sınır koymadıkça üst sınırı olmayan bir boşluk vardır. Bu boşlukta bir ürün tükenebilir, bir alıcı iptal edebilir, bir fiyat değişebilir. Bir consumer, publish edildiğinde doğru olan bir event'in işlendiğinde hâlâ anlamlı olup olmadığına karar vermek zorundadır.

Not: Event'lerin geçmiş zamanla adlandırılmasının nedeni budur. OrderPlaced, geçmiş bir ana dair bir olgudur. Onu hemen şimdi harekete geçme talimatı olarak ele alan bir consumer, sessizce hiç zaman geçmediğini varsaymıştır. Bir command, örneğin CapturePayment, bir servisten bir şey yapmasını ister ve reddedilebilir; bir event ise zaten olmuş bir şeyi bildirir. Bölüm 11 bu ayrımı geliştirir.

"Eventually"nin gerçekte anlamı

Eventual consistency açıklamalarının çoğu tek cümledir: yeni güncelleme yapılmazsa tüm replica'lar aynı değere yakınsar. Bu doğrudur ve bir checkout akışı kuran biri için neredeyse işe yaramazdır, çünkü "eventually"nin ne kadar uzun olduğunu ve beklerken ne yapılacağını dışarıda bırakır.

Eventually bir süre değildir

200 milisaniyede yakınsayan bir sistem ile gece batch'inden sonra yakınsayan bir sistem, ikisi de eventually consistent'tır. Bir onay sayfasına bakan alıcı için bunlar farklı ürünlerdir. Önemli olan her hop için, kabaca da olsa, bir sayı isterdim. Diyelim ki Orders, capture'dan sonraki birkaç saniye içinde onaylanmış bir sipariş göstermeli, Notifications ise bir dakika sürebilir. Bu rakamlar örnektir ve sizinkiler farklı olacaktır, ama "eventually"yi test edilebilen ve üzerine alert kurulabilen bir gereksinime dönüştürürler. Sınırı olmayan bir gecikme bir tasarım tercihi değildir. Tercihin yokluğudur.

Eventually bir garanti değildir

Yakınsama, yalnızca önemli olan her mesaj eninde sonunda teslim edilirse, her consumer onu işlerse ve hiçbir adım onu sessizce düşürmezse gerçekleşir. Bir mesaj, bir veritabanı commit'i ile bir broker publish'i arasında kaybolursa görünümler yakınsamaz. Bir insan ya da bir onarım işi fark edene kadar farklı kalırlar. "Eventually" koşullu bir vaattir ve koşul, sistemin kaybı tespit edilebilir ve onarılabilir kılan mekanizmalara sahip olmasıdır. Bir broker seçmek bunları sağlamaz. Bu kitabın II. ve III. Kısımları kaybın çoğunu önler, Bölüm 13 ise geri kalanını tespit edip onarır.

Beklerken ne yapmalı

Tutarsızlık penceresi boyunca diğer okuyucular bir ara durum görür. Alıcının kartından para çekildikten bir an sonra, tek bir siparişin şu anlık görüntüsünü düşünün:

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

Her servis kendi verisi konusunda haklı. Birleşim tutarsız görünüyor ve birkaç saniye için normal. Yalnızca öyle kaldığında problem olur. Faydalı operasyonel soru "servisler şu anda tutarlı mı?" değil, "bu sipariş ne zamandır bu durumda ve bu, izin verdiğimizden uzun mu?" sorusudur.

Güvenilirliğin çalışan bir tanımı

"Güvenilir" o kadar gevşek kullanılıyor ki iki mühendis bir sistemin güvenilir olması gerektiği konusunda anlaşıp zıt şeyleri kastedebilir. Kontrol edilebilir bir tanım istiyorum. Event-driven sistemler için dört özellik kullanıyorum. Bunlar neyin korunduğunu söyler; yukarıdaki üç soru ise bir birleşim yerini bu özelliklere karşı nasıl kontrol edeceğinizdir.

Loss yok. Bir iş olgusunu temsil eden her event, ona ihtiyaç duyan her consumer'a ulaşır. Loss en zarar verici hatadır, çünkü sessizdir: sistem çalışmaya devam eder ve tek belirti, bir alıcı şikâyet edene kadar kimsenin fark etmediği bir yokluktur. Loss olmaması, broker'ın değil, yolun tamamının bir özelliğidir. Producer mesajı saklayamadıysa ya da consumer işi bitirmeden ack'lediyse durable bir broker işe yaramaz. Bölüm 2 broker'ların neyi vaat ettiğini inceliyor, Bölüm 4 ve Bölüm 5 ise producer tarafını ele alıyor.

Zararlı duplication yok. Bazı mesajlar birden fazla kez teslim edilecektir. Bu belirli bir broker'ın kusuru değil, mesaj kaybetmemenin bedelidir: bir gönderen mesajın ulaşp ulaşmadığını söyleyemediğinde güvenli hamle onu yeniden göndermektir. Güvenilir bir sistem duplicate'leri transport katmanında önlemeye çalışmaz. Onları zararsız hâle getirir. "Zararlı" sözcüğü önemlidir. İki özdeş e-posta bir can sıkıntısıdır; aynı ödemenin ikinci bir capture'ı ise mali bir olaydır. Bölüm 7 ana araç olan idempotency'yi ele alıyor.

Sınırlı gecikme. Her event, işin tolere ettiği bir süre içinde işlenir ya da sistem bunun işlenmediğini yüksek sesle söyler. Doğru biçimde ama üç gün geç teslim edilen bir mesaj, birçok amaç için kayıp bir mesajdır. Sınırlı demek hızlı demek değildir. Bir sınırın var olduğu, birinin onu seçtiği ve ihlalinin bir alert ürettiği anlamına gelir. Bir ödeme onayı saniyelere ihtiyaç duyabilir; haftalık bir ekstre saatleri tolere edebilir. Bölüm 6 retry ve backoff'u, Bölüm 13 ise lag ve alerting'i ele alıyor.

Explainability. Herhangi bir sipariş için, herhangi bir zamanda, birisi ne olduğunu yeniden kurabilir: hangi event'ler üretildi ve tüketildi, her servis neye karar verdi ve neden. Bu en az gösterişli özelliktir ve diğer üçünün korunup korunamayacağını belirler. Bir mesajı kaybeden ama tam olarak nerede kaybettiğini gösterebilen bir sistem onarılabilir. Kimsenin izini süremediği bir biçimde yanlış davranan bir sistem değildir. Bölüm 13, bunun için gereken tanımlayıcıları, kayıtları ve karşılaştırmaları ele alıyor.

Dördü birbirine karşı çeker; bu yüzden yalnızca birini içeren bir tanım tehlikelidir. Sınırsız retry'larla loss olmamasının peşine düşmek, sınırlı gecikmeyi tehdit eder. Gecikmeyi sınırlamak için yavaş mesajları düşürmek, loss olmamasını tehdit eder. Duplicate'lerden kaçınmak için retry etmeyi reddetmek loss'u geri getirir. Güvenilirlik, akış başına seçilen ve yazıya dökülen bilinçli bir dengedir.

Happy path production'da neden başarısız olur

Elimizde tanım varken Parcelly'nin ilk ayına dönün ve hataları özelliğe göre sıralayın. Hasarın çoğunu birkaç mekanizma açıklar.

Parcelly'nin gördüğü	Risk altındaki özellik	Altta yatan neden	Bölüm
Sipariş kabul edildi, ama downstream'de hiçbir şey olmuyor	Loss yok	Event bir veritabanına commit edildi ama hiç publish edilmedi	4
Sipariş iki kez kargolandı	Zararlı duplication yok	Yeniden teslim edilen mesaj tekrar işlendi	7
Bir kötü mesaj bir partition'ı bloke ediyor	Sınırlı gecikme	Bir mesaj her seferinde başarısız oluyor ve sonsuza dek retry ediliyor	8
Sipariş "verildi"den önce "onaylandı"	Explainability	İlişkili event'ler sırasız işlendi	9

Parcellly'nin gördüğü	Risk altındaki özellik	Altta yatan neden	Bölüm
Ledger ödemelerle uyuşmuyor	Explainability	Ortak tanımlayıcı yok, görünümüleri karşılaştırmanın yolu yok	13

Dördüncü satır sorunlu olanıdır. Reordering hiçbir şeyi duplicate etmez. Durumu bozar; örneğin eski bir güncelleme daha yeni bir güncellenmenin üzerine yazdığında. Dört özelliğin hiçbiri bunu doğrudan adlandırmaz. Ben bunu explainability altında sınıflandırıyorum, çünkü belirti kimsenin açıklayamadığı bir geçmişti ve durum bozulmasının kendisini Bölüm 9 ele alır.

İlk satır bir adım adım inceleme hak ediyor, çünkü balayını en sık bitiren hata odur.

```
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))
```

Bu blok güvenli değildir ve bozuk taban çizgi olarak gösterilmiştir. Veritabanı commit'i ve broker publish'i, ortak bir transaction'ı olmayan iki sistemdir; kitap buna **dual write** der. Process iki satır arasında durursa sipariş vardır ve kimse bunu duymaz. Satırlar ters çevrilirse hiç commit edilmemiş bir sipariş için event publish edilebilir. İki sıralama da doğru değildir ve ikisi de process'i tam yanlış anda öldürmeyen her testi geçer.

Bu nadir bir bug değildir. Hiçbir zaman bir hata modeli olmayan bir tasarımın varsayılan davranışdır. Bölüm 3 modeli, Bölüm 4 düzeltmeyi veriyor.

Kitabın haritası

Kitabın geri kalanı, tanımın ortaya çıkardığı soruları, bir ekibin genellikle karşılaştığı sırayla cevaplıyor.

Soracağınız soru	Bölüm
Broker gerçekte neyi vaat ediyor, neyi etmiyor?	2
Ne bozulabilir ve bozulduğunda ne göreceğim?	3
Veritabanımı değiştirip bir event'i kaybetmeden nasıl publish ederim?	4
Publish edilen event'ler veritabanından broker'a nasıl ulaşır?	5
İşleri daha kötü hâle getirmeden nasıl retry ederim?	6
Tekrarlanan işlemeyi nasıl zararsız hâle getiririm?	7
Asla başarılı olamayacak bir mesajla ne yapmalıyım?	8
Sıra ne zaman önemlidir ve onu nasıl korurum?	9
Servisler arasında işi nasıl koordine eder ve geri alırım?	10
Bir event neyi içermeli ve ne diye adlandırılmalı?	11
Bir event'i consumer'ları bozmadan nasıl değiştiririm?	12
Olup bitenleri nasıl görür, ölçer ve onarırım?	13
Kaybolan veriden, kaybolan bir region'dan ya da bir broker değişiminden nasıl kurtarırım?	14
Hata yollarını nasıl test ederim?	15
Canlıya çıkmadan önce neyi kontrol ederim?	16

Bu bölümün açtığı Kısım I, kelime dağarcığını ve hata modelini kuruyor. Kısım II güvenilir şekilde yayınlamayı, Kısım III güvenilir şekilde tüketmeyi, Kısım IV iş akışlarını ve kontratları, Kısım V ise sistemi işletmeyi ve kanıtlamayı ele alıyor. İlk geçişte kısımları sırayla okuyun. Bugün belirli bir problemle uğraşıyorsanız tablo bir dizindir.

Trade-off'lar ve maliyetler

Event'lere geçmek yanlış değildir ve bu bölüm ona karşı bir argüman değildir. Geçiş gözler açık yapmak için bir argümandır, çünkü maliyet gerçektir ve belirli yerlerde ödenir.

State karmaşıklığıyla ödersiniz. Senkron akışın iki sonucu vardı: tamamlandı ya da başarısız oldu. Event-driven akışın birçok ara durumu vardır ve her birinin adlandırılması, kullanıcılara gösterilmesi, izlenmesi

ve sonunda çözülmesi gerekir. Birinin bu state machine'in sahibi olması gerekir.

Operasyonel beceriyle ödersiniz. Bir çağrı zincirinde hata ayıklamak bir stack trace gerektirir. Bir mesaj akışında hata ayıklamak correlation id'leri, lag dashboard'ları ve zamanlama üzerine akıl yürütebilme becerisi gerektirir. Bunlara sahip olmayan bir ekip, incident'ler seyrekleşse bile bir incident'i anlama süresinin uzadığını görür. Maliyet nöbetçi kişiye düşer.

Test eforuyla ödersiniz. Happy path'i test etmek az efor gerektirir. Hata yolları, tekrarlanmış mesajlar, sırası bozulmuş mesajlar ve elverişsiz anlarda yeniden başlatmalar gerektirir. Bölüm 15 bunu yavaş, güvenilmez bir test paketi olmadan nasıl yapacağınızı gösteriyor, ama efor sıfır değildir.

Basit şeylerde latency ile ödersiniz. Tek bir fonksiyon çağrısı olan bir işlem, bir broker üzerinden birkaç hop'a dönüşür. Çağırının cevaba hemen ihtiyacı olduğunda, örneğin fiyat göstermeden önce bir indirim kodunu doğrularken, senkron bir çağrı daha iyi bir araçtır.

İki adım birlikte başarılı olmak ya da başarısız olmak zorundaysa ve aynı serviste ve veritabanında yaşıyorsa event'leri kullanmazdım. Yerel bir transaction, bu kitaptaki herhangi bir örüntüden daha basit, daha güçlü ve daha ucuzdur. Ekibin bir broker'ı ve etrafındaki monitoring'i işletecek kapasitesi yoksa ya da sistem, tek bir veritabanı olan tek bir deploy edilebilir birimin işi gördüğü kadar küçükse de tereddüt ederdim. Birçok iyi sistem iki tarzı karıştırır: cevap gerektiren bir soru için senkron bir çağrı, başkalarının önemseyebileceği bir olgu için bir event. Sistem başına değil, etkileşim başına seçerdim.

Dört özellikli tanımın da bir sınırı var. Bir şartname değil, bir mercektir. İşiniz için doğru gecikme sınırının ne olduğunu ya da belirli bir consumer için hangi duplicate'lerin zararlı olduğunu söylemez. Sayılarınızı bilmiyorum ve bu cevaplar nadiren yalnızca mühendislikten gelir. Sunabileceğim şey, onları sorma alışkanlığıdır.

Sık yapılan hatalar

- **Yalnızca happy path'i test etmek.** Cazip, çünkü özellik isteği onu anlatıyordu ve hızlı geçiyor. Akış diyagramındaki her oku hata

hakkında bir soru olarak ele alın ve her cevap için en az bir test yazın (Bölüm 15).

- **Broker'ı güvenilirlik katmanı saymak.** Broker'lar durability'yi reklam eder, bu da bunu cazip kılar. Bir broker, kabul ile teslim arasındaki kesiti kapsar ve en çok zarar veren hatalar bunun öncesinde ve sonrasında yer alır. Broker'ın neyi vaat ettiğini listeleyin, sonra vaadin etrafındaki boşlukları listeleyin (Bölüm 2 ve Bölüm 3).
- **"Eventually"yi bir tasarım olarak kullanmak.** Bir karar gibi geliyor ve ne kadar süreceği gibi daha zor sorudan kaçınıyor. Önemli olan her asenkron hop'a bir sınır ve bir alert iliştin ve kullanıcının beklerken ne gördüğünü söyleyin.
- **Duplicate'lerin ele alınamayacak kadar nadir olduğunu varsaymak.** Sessiz bir test ortamında hiç görünmezler. Consumer'ları, her mesajın birden fazla kez gelebileceği varsayımıyla tasarlayın (Bölüm 7).
- **Veritabanına commit etmeyi ve broker'a publish etmeyi iki düz adım olarak yapmak.** Doğal okunuyor ve her demoda çalışıyor. Düzeltme, Bölüm 4'teki transactional outbox'tır.
- **Güvenilirliği uptime olarak tanımlamak.** Uptime ölçmek ucuzdur, ama bir servis bütün ay ayakta kalıp yine de mesaj kaybedebilir ya da onları geç işleyebilir. Dört özelliği doğrudan ölçün: ne kayboldu, ne tekrarlandı, işler ne kadar gecikti ve bunları açıklayabildiniz mi.

Temel çıkarımlar

- Happy path, her adımın başarılı olduğu bir hikâyedir ve yalnızca ona karşı test edilen bir sistem, uzun süre nadiren geçerli olan tek durumda çalıştığını göstermiştir.
- Senkron çağrılardan event'lere geçmek coupling'i ortadan kaldırmaz; onu zaman ve availability'den kontratlara, broker davranışına ve yazılmamış zamanlama varsayımlarına kaydırır.
- Event-driven hatalar sessizdir ve servislere yayılır; bu yüzden hata modeli olmayan bir tasarım birleşim yerlerini production'dan öğrenme eğilimindedir.
- "Eventually", süresi eklenmemiş koşullu bir vaattir; bu yüzden önemli olan her asenkron hop'un bir sınırı, bir alert'e ve bekleme dönemi için

tanımlanmış, kullanıcıya dönük bir duruma ihtiyacı vardır.

- Güvenilir bir sistemde loss yoktur, zararlı duplication yoktur, gecikme sınırlıdır ve explainability vardır; bunlar birbirine karşı takas edildiği için her akışın bilinçli, yazıya dökülmüş bir dengeye ihtiyacı vardır.
- Bir veritabanı commit'ini bir broker publish'inin izlemesi gibi bileşenler arasındaki birleşim yerleri bileşenlerin kendisinden daha fazla zarara yol açar ve bu kitabın büyük kısmı o birleşim yerleri hakkındadır; üç soruyla incelenirler: ne başarısız olabilir, bunu nasıl gözlemlersiniz ve hangi garanti ayakta kalır.
- Event'ler her zaman doğru araç değildir: adımlar yerel bir transaction paylaşabiliyorsa ya da çağırının hemen bir cevaba ihtiyacı varsa senkron bir çağrı daha basit ve daha güçlüdür.

Bir sonraki bölüm broker'ın kendisine bakıyor. Boşlukları etrafında tasarım yapabilmemiz için, bir queue ya da log bir mesajın teslim edildiğini söylediğinde tam olarak neyi vaat ettiğini ve neyi sessizce size bıraktığını bilmemiz gerekiyor.

Bölüm 2 — Mesajlaşma Semantiği

Parcelly'de bir indirim döneminin ortasında, bir cuma akşamını düşünün. Notifications servisi aylardır "siparişiniz onaylandı" e-postalarını şikâyet almadan gönderiyor. Biriken mesajlar arttıkça autoscaling consumer ekliyor, birkaç dakika sonra da bazılarını yeniden kaldırıyor. Akşamın sonunda destek ekibinde iki ticket var. Bir alıcı aynı onay e-postasını üç kez almış. Bir diğeri, siparişi onaylanmış olmasına rağmen hiç almamış.

Ekip broker dashboard'unu açıyor ve hiçbir şeyin bozuk olmadığını görüyor. Her offset commit edilmiş. Her consumer sağlıklı. Grafikler düz. Yine de aynı sistem bir alıcı için duplicate, bir diğeri için sessizlik üretti ve iki sonuç da dokümantasyonla uyumlu.

Biri broker'ın "at-least-once delivery" vaat ettiğini söylüyor, bir başkası bunu broker'ın hallettiğini varsaymış olduğunu. İki söz de doğru, ama hiçbirini iki alıcıya olanı açıklamıyor. Notifications'ın log tabanlı bir topic'i tek bir consumer group olarak okuduğunu, client'ının offset'leri otomatik commit ettiğini ve group her boyut değiştirdiğinde bir rebalance çalıştığını varsayın. Bu iki olgu iki ticket'ı da üretebilir ve bu bölüm bunun nasıl olduğunu gösteriyor.

Bunu açıklamak için ekibin net bir resme ihtiyacı var: broker neyi vaat ediyor, neyi etmiyor ve bir mesajın hayatının hangi kısımları uygulamaya ait? Bir mesaj, producer ile tamamlanmış bir yan etki arasında gerçekte nelerden geçiyor ve her garanti nerede başlayıp nerede bitiyor?

Problem ve neden zor olduğu

Çoğu mühendis broker ile en basit tarifi üzerinden tanışır: bir mesaj publish edersiniz, bir consumer onu alır. Bu tarif üç ayrı devri gizler ve bu kitaptaki her güvenilirlik sorusu bunlardan birinde yaşar.

İlk devir producer'dan broker'a. İkincisi broker'dan consumer'a. Üçüncüsü consumer'dan etkiye: veritabanı yazması, e-posta, bir kargo firmasına yapılan çağrı. Broker ilk ikisi hakkında çok şey söyleyebilir, üçüncüsü hakkında neredeyse hiçbir şey söyleyemez, çünkü üçüncüsü broker'ın göremediği koddaki ve sistemlerde gerçekleşir.

Zorluk şurada: her devir bir acknowledgment ile biter ve acknowledgment, bir tarafın bir ağ üzerinden diğerine geçmişle ilgili yaptığı bir beyandır. "Bunu aldım" ya da "Bunu bitirdim" der. "Ve başka hiçbir şey ters gidemez" demez. Bir acknowledgment kaybolduğunda gönderen, işin yapıp yapılmadığını söyleyemez. Bu belirsizlik, üç delivery semantiğinin köküdür ve kelime dağarcığını edindikten sonra ona döneceğiz.

İkinci bir zorluk, "broker" sözcüğünün, önemli şekillerde farklı davranan iki aileyi kapsamasıdır. **Queue**, her mesajın tek bir consumer'a teslim edildiği ve tek tek ack'lendiği bir broker yapısıdır. **Event log** ise consumer'ların kendi hızlarında okuduğu ve bir pozisyon kaydettiği, kalıcı, yalnızca sona ekleme yapılan bir event dizisidir. Bir aile için doğru olan ifadeler çoğu zaman diğeri için yanlıştır. Kitap, ayrımın önemli olduğu her yerde aileyi adıyla anar ve bu ayrımın kurulduğu yer bu bölümdür.

Üçüncü zorluk, bir ürünün dokümantasyonundaki garantilerin kapsamlı olmasıdır. "En az bir kez teslim edilir" (at least once), broker'ın kendi sınıрыyla ilgili bir beyandır. Consumer'ın mesajı aldıktan sonra yaptığı şeyi kapsamaz ve producer'ın publish ederken seçtiği ayarlara bağlıdır. Garantiyi kapsamı olmadan okumak, bir ekibin yukarıdaki cuma akşamına varmasının yoludur.

Queue'lar ve log'lar

Queue modeli

Bir queue'da broker her mesajın durumunu izler. Bir mesaj ya bekliyordur, ya bir consumer'a teslim edilmiştir, ya da ack'lenmiş ve gitmiştir. Aynı queue'dan birkaç consumer okuyabilir; bu düzene **competing consumers** denir, çünkü her mesaj aynı anda bunlardan birine teslim edilir ve sıradaki mesaj için yarışlar.

Bir consumer bir mesajı ack'lediğinde broker onu siler. Dolayısıyla retention tasarım gereği kısadır: bir mesaj, işlenene ya da süresi dolana kadar yaşar. Zaten ack'lenmiş bir mesajı queue'dan isteyebilmenin yerleşik bir yolu yoktur, çünkü queue onu saklamaz.

Sonuç olarak queue, bir kez yapıp unutulması gereken işler için iyi bir seçimdir. "Bu fatura PDF'ini oluştur" gibi bir Parcelly işi doğal bir adaydır: işi yap, sonra mesajın başka bir işe yaramaz. Maliyeti, geçmişin mevcut olmamasıdır. Bir bug bir saat boyunca yanlış metin gönderirse, olguları taşıyan mesajlar gitmiştir ve onarım başka bir yerden başlamak zorundadır.

Log modeli

Bir event log'da broker mesaj başına durum izlemez. Mesajları partition'lar içinde sırayla saklar ve onları, kimin okuduğundan bağımsız bir retention politikasına göre tutar. **Partition**, bir topic'in sıralı bir alt dizisidir ve **topic** adı olan bir mesaj akışıdır. Her mesajın, partition içindeki konumu olan bir **offset**'i vardır.

Consumer'lar ne kadar okuduklarının kaydını tutar. Bu konuma da offset denir ve onu consumer ya da onun adına broker kaydeder. **Consumer group**, bir topic'in işini paylaşan consumer kümesidir: her partition, grupta aynı anda tek bir consumer'a atanır. Farklı group'lar aynı topic'i birbirinden bağımsız okur, her birinin kendi offset'leri vardır.

Log'un tanımlayıcı özelliği, okumanın tüketmemesidir. On group aynı topic'i okuyabilir ve yeni bir group en baştan başlayabilir. Geçmişin bilinçli olarak yeniden okunması olan **replay**, kitabın bu eylem için ayırdığı terimdir ve yalnızca log veriyi tuttuğu için mümkündür. Parcelly'nin Ledger'ı bundan yararlanan bir consumer için iyi bir örnektir: entry'lerin oluşturulma biçiminde bir bug bulunursa Ledger servisi düzeltilebilir ve eski event'lere yeniden yönltililebilir.

Maliyet, log'un defter tutma işini consumer tarafına itmesidir. Klasik log modelinde broker, mesajları tek tek izlemediği için, başarısız olan tek bir mesajı yeniden teslim ederken sıradakine geçemez. Offset 500'deki mesajı işleyemeyen bir consumer, 501'e güvenle geçmeden önce ne yapacağına karar vermek zorundadır.

Aralarında seçim yapmak

Konu	Queue ailesi	Log ailesi
İzleme birimi	Her mesaj	Partition başına bir pozisyon
Retention	Ack'lenene ya da süresi dolana kadar	Okuyuculardan bağımsız, süreye ya da boyuta göre
Birkaç bağımsız okuyucu	Okuyucu başına mesajın bir kopyası gerekir	Doğal, group başına bir pozisyon
Geçmişin replay'i	Mevcut değil	Retention içinde mevcut
Tek bir başarısız mesaj	Kendi başına yeniden teslim edilir, diğerleri devam eder	Ele alınmazsa partition'ı bloke eder
Ordering	Depolamada olsa olsa FIFO; competing consumer'lar ve redelivery arasında kaybolur	Partition içinde güçlü

Bir aileyi aracın popülerliğine göre seçmezdim. Üç soruyla başladım: geçmişini yeniden okuması gereken biri var mı, birkaç consumer'ın aynı olgulara ihtiyacı var mı ve aynı aggregate için ordering önemli mi? Bunlardan herhangi birine evet, log'a işaret eder. Geçmişe ihtiyaç duymayan ve tek tek mesajların bağımsız retry'larına ihtiyaç duyan saf bir iş queue'su, queue'ya işaret eder.

Birçok ekip ikisini de çalıştırır. Diğer servislerin durum kurduğu event'ler bir log'a, bir kez gerçekleşmesi gereken görevler bir queue'ya gider. Bu meşru bir tasarımdır ve iki ayrı semantik kümesini işletmenin ve anlamanın maliyetiyle gelir. Bölüm 9, ordering'in her ailede ne anlama geldiğine geri döner.

Push, pull ve teslimat akışı

Broker'lar, teslimatı kimin başlattığı konusunda da farklılaşır. **Push** modelinde broker, mesajlar hazır oldukça consumer'lara gönderir. **Pull** modelinde consumer, hazır olduğunda mesaj ister.

Queue tabanlı broker'lar genellikle push yapar (RabbitMQ consumer'lara teslim eder, Amazon SQS consumer'ları ise poll eder, yani bu aile içinde bile değişir) ve bir consumer'ın aynı anda tutabileceği ack'lenmemiş mesaj sayısına bir sınır koyar. Bu sınıra çoğu zaman prefetch count ya da

window denir. Log tabanlı broker'lar genellikle pull yapar: consumer, mevcut offset'inden itibaren sıradaki batch'i ister. İki model de her iki ailede bulunabilir, bu yüzden bunu bir yasa değil bir eğilim olarak ele alırdım.

Fark, yoğunlukla consumer yavaş olduğunda ne olduğu açısından önemlidir. Sınırsız bir push modelinde broker göndermeye devam eder, consumer'ın belleği henüz başlamadığı mesajlarla dolar ve bu mesajlar ne başka bir consumer için kullanılabilir ne de bitmiş olduğu bir durumda bekler. Pull modelinde consumer kendi hızını kontrol eder ve yavaş bir consumer, aşırı yüklenmiş bir process olarak değil, büyüyen consumer lag olarak görünür. **Consumer lag**, bir consumer'ın bir partition'ın ya da queue'nun başından mesaj ya da süre olarak ne kadar geride olduğudur.

Pratik kural, bir consumer'ın, işi üzerindeki hakkının süresi dolmadan bitirebileceğinden fazla iş tutmaması gerektiğidir. Bölüm 3, yavaş consumer'ları ve backpressure'ı başlı başına hatalar olarak ele alır, Bölüm 6 ise yükün nasıl sınırlanacağını anlatır. Şimdilik "teslim edildi" ile "işleniyor"un farklı durumlar olduğunu ve aralarındaki boşluğun zamanın aktığı yer olduğunu hatırlayın.

Acknowledgment, redelivery ve offset'ler

Queue'da acknowledgment

Acknowledgment, genellikle **ack** olarak kısaltılır; consumer'ın bir mesajla işini bitirdiğine ve broker'ın onu unutabileceğine dair beyanıdır. **Nack**, yani negatif acknowledgment, consumer'ın mesajı işleyemediğini söyler. Broker'a ve ayarlara bağlı olarak bir nack, mesajı queue'ya geri döndürür, düşürür ya da bir dead-letter queue'ya (DLQ) gönderir.

Anahtar tasarım sorusu, ne zaman ack yapılacağıdır. Makul yalnızca iki konum vardır.

- **İşlemeden önce ack.** Broker mesajı hemen unuttur. Consumer işin ortasında çökerse mesaj kaybolur. Bu, at-most-once davranışı üretir.
- **İşledikten sonra ack.** Broker, iş bitene kadar mesajı tutar. Consumer işi bitirdikten sonra ama ack'lemeden önce çökerse mesaj geri gelir ve iş iki kez yapılır. Bu, at-least-once davranışı üretir.

İki özelliği birden veren bir konum yoktur, çünkü acknowledgment ile iş, iki ayrı sistemdeki iki ayrı eylemdir ve bir çökme ikisinin arasına düşebilir.

Visibility timeout ve redelivery

Bir queue, bir consumer'ın çöktüğüne nasıl karar verir? Bunu doğrudan gözlemleyemez. Bazı broker'lar bağlantıyı izler ve kapanan bir bağlantıyı hata sayar. Diğerleri bir **visibility timeout** kullanır: bir mesaj teslim edildiğinde, belirli bir süre boyunca diğer consumer'lara görünmez olur. Consumer süre dolmadan ack'lerse mesaj silinir. Ack'lemezse mesaj yeniden görünür olur ve başka birine teslim edilir. Bu ikinci teslimat bir **redelivery**'dir.

Timeout, işlemenin ne kadar süreceğine dair bir tahmindir ve yanlış tahminin iki yönü de bir şeye mal olur. Çok kısaysa, yavaş ama canlı bir consumer, hâlâ çalışırken mesajının ikinci bir consumer'a verildiğini görür. Şimdi iki consumer aynı mesajı işliyor ve ilk consumer'ın sonunda göndereceği ack başarısız olabilir, yok sayılabilir ya da broker'a bağlı olarak başarılı olup ikinci consumer hâlâ çalışırken mesajı silebilir. Bu son durumda ikinci denemenin başarısız olması mesajı kaybettirir. Çok uzunsa, gerçek bir çökme mesajı o sürenin tamamı boyunca gizli bırakır ve gecikme, alıcı için latency olarak ortaya çıkar.

Parcelly'nin Inventory consumer'ının bir rezervasyonu normalde bir saniyenin çok altında bitirdiğini ve birinin timeout'u 30 saniyeye ayarladığını varsayalım. Bu sayı örnek amaçlıdır. Bir rezervasyonun 40 saniye sürmesine yol açan bir veritabanı aksaması, artık aynı **OrderPlaced**'in ikinci bir teslimatını üretecek ve Inventory stoğu iki kez rezerve etmeye çalışacaktır. Broker'da hiçbir şey bozuk değil. Timeout bir varsayımı kodladı ve varsayım geçerli olmaktan çıktı.

Bazı sistemler, bir consumer'ın çalışırken timeout'u uzatmasına izin verir. Bu, meşru biçimde uzun işlerde yardımcı olur ve yanlış yapılabilecek yeni bir şey ekler: asla bitirmeyeceği bir mesaj için sonsuza kadar uzatan bir consumer.

Log'da offset'ler

Bir log'da mesaj başına ack yoktur. Bunun yerine consumer bir **offset commit** yapar: "Bu pozisyondan önceki her şeyi işledim" diye kaydeder. Bir yeniden başlatma ya da yeniden atamadan sonra bir sonraki consumer,

commit edilmiş offset'ten okur.

Aynı iki konum burada da vardır. Offset'i işlemeyen önce commit edin; bir çökme, son tamamlanan mesajdan sonra gelen ve commit edilmiş pozisyona kadar uzanan mesajları kaybeder. İşledikten sonra commit edin; bir çökme, son commit ile çökme arasındaki mesajları tekrarlar. Gerçek consumer'lar batch'ler hâlinde commit eder, dolayısıyla bir çökmeden sonra tekrarlanan mesaj sayısı bir değil, bir batch'tir.

Birçok client kütüphanesi otomatik commit yapabilir. Bu pratiktir ve aynı zamanda delivery semantiğinizi belirlemenin en az açık yoludur, çünkü client'a bağlı olarak commit, işlemenizin bittiği andan bağımsız bir anda gerçekleşebilir. İşleme başka bir thread'e devredilmişse client, hiç tamamlanmamış işi commit edebilir. Bence önemli bir consumer, ekibin kodda gösterebileceği ve üzerinde akıl yürütebileceği bir noktada, açıkça commit etmelidir.

Consumer group'lar ve rebalancing

Consumer'lar bir consumer group'a katıldığında ya da ondan ayrıldığında broker, partition'ları aralarında yeniden dağıtır. Buna **rebalancing** denir. Bir rebalance sırasında, bir consumer tarafından okunmakta olan bir partition başka birine atanır ve yeni sahip, son commit edilmiş offset'ten başlar.

Log tabanlı group'lar, queue'daki visibility timeout'un yaptığı tahmini başka bir biçimde yapar. Yapılandırılmış bir aralık içinde broker'a dönmeyen consumer ölü sayılır, partition'ları yeniden atanır ve commit edilmemiş işini yeni sahip tekrar okur. Hâlâ canlı olan yavaş bir consumer, rebalance duplicate'lerinin gündelik sebebidir.

Açılış senaryosundaki duplicate'lerin kaynağı budur. 100'den 150'ye kadar olan mesajları işlemiş ama yalnızca 120'ye kadar commit etmiş bir consumer düşünün. Bir deploy onu yeniden başlatır ve partition, 120'den okuyan ikinci bir consumer'a geçer. 120'den 150'ye kadar olan mesajlar iki kez işlenir ve sistemin hiçbir parçası arıza yapmamıştır. Rebalancing, her deploy'da, scale-up'ta ve çökmede olabilen normal bir olaydır; dolayısıyla redelivery nadir bir kaza değil, gündelik bir olaydır.

Sessizliğin sebebi bunun ayna görüntüsüdür. Client, consumer'ın yalnızca bir arka plan gönderici'ye devrettiği mesajlar için offset commit ettiyse ve process, gönderici bitirmeden durdurulduysa, sonraki sahip bu mesajların ardından başlar. Bu mesajlar bir daha okunmaz ve hiçbir log satırı bunu söylemez. Aynı rebalance bu yüzden bir alıcının mesajını tekrarlayıp bir diğerininkini düşürebilir.

İlgili bir tehlike daha var: kısa bir süre için eski sahip, partition'ı kaybettiğini henüz bilmeyebilir ve üzerinde çalışmaya devam ediyor olabilir. O zaman iki consumer aynı partition'ı aynı anda işleyebilir. İyi tasarlanmış client'lar bunu önlemeye çalışır, ama önlemenin kusursuz olmasına bağlı bir consumer tasarlamazdım.

Delivery semantiği

Mekanik yerine oturduğuna göre, üç delivery semantiği, ack'i nereye koyduğunuzun sonuçlarıdır.

At-most-once

At-most-once, her mesajın sıfır ya da bir kez teslim edildiği anlamına gelir. İşlemeden önce ack'leyerek ya da başarısız bir publish'i yeniden denemeyerek elde edilir. Bir mesaj kaybolabilir ve asla duplicate olamaz.

Bir mesajı kaybetmenin maliyeti, duplicate'leri ele almaktan düşükse makul bir seçimdir. Bir Parcelly kargo firmasından gelen periyodik "mevcut konum" ping'i bir örnektir: sonraki ping, kaybolanı önemsiz kılar.

PaymentCaptured için yanlış seçimdir; orada kaybolan mesaj, paranın hareket ettiği ve kimsenin bunu kaydetmediği anlamına gelir.

At-least-once

At-least-once, her mesajın bir ya da daha fazla kez teslim edildiği anlamına gelir. İşledikten sonra ack'leyerek ve bir acknowledgment gelene kadar yeniden deneyerek elde edilir. Bir mesaj, broker onu tuttuğu sürece asla kaybolmaz ve yeniden teslim edilebilir.

Kitabın üzerine kurulduğu semantik budur. Üçü arasında hatanın görünür ve kurtarılabılır olduğu tek semantiktir: duplicate, tespit edip yok sayabileceğiniz bir şeydir; kaybolan bir mesaj ise günlerce fark

etmeyebileceğiniz bir şeydir. Bedeli, her consumer'ın bir **duplicate**'e, yani birden fazla kez işlenen ya da teslim edilen bir mesaja, tahammül etmesi gerektirir. Bölüm 7 bunun nasıl yapılacağıyla ilgilidir.

"Exactly-once"

Delivery, güvenilir bir ağ üzerinden yapılan fiziksel bir eylemdir ve alıcının acknowledgment'ı, diğer her mesaj gibi kaybolabilir. Ack almamış bir gönderenin iki seçeneği vardır: tekrar göndermek, ki bu duplicate riski taşır, ya da göndermemek, ki bu kayıp riski taşır. Hiçbir protokol bu seçimi yalnızca iki taraf için ortadan kaldırmaz.

Yine de exactly-once gerçek ve yararlı bir sonuçtur ve bir delivery garantisi değil, bir etkidir. At-least-once delivery, tekrarlanan işlenmesi bir kez işlenmesiyle aynı etkiye sahip olan bir consumer ile buluştuğunda, birleşik sistem her mesaj tam bir kez ele alınmış gibi davranır. Tarif, at-least-once delivery artı bir **idempotent consumer**'dır.

Bazı broker'lar exactly-once processing'i reklam eder ve bu iddia, bir sınır içinde meşrudur. Log tabanlı bir broker, girdi, çıktı ve offset commit'in hepsi aynı broker'ın içinde yaşadığında bir okuma-işleme-yazma döngüsünü atomik yapabilir. Bu sınırın içinde bir çökme her şeyi birlikte geri alır. Garanti ayrıca her downstream okuyucunun yalnızca commit edilmiş veriyi okumasını gerektirir ve bir yeniden başlatmadan sonra eski instance'ın dışarıda bırakılabilmesi için kararlı bir producer kimliğine dayanır. Sınır, yan etkilerinizin başladığı yerde biter. Consumer bir ilişkisel veritabanına yazdığı, bir ödeme sağlayıcısını çağırdığı ya da bir e-posta gönderdiği anda broker'ın transaction'ı artık etkiyi kapsamaz ve duplicate problemi geri döner.

Bu nedenle kitabın varsayılanı at-least-once artı idempotent consumer'lar olarak kalır. Herhangi bir broker ve herhangi bir yan etkiyle çalışır, etkinleştirilmiş bir özelliğe bağlı değildir ve farklı bir ürüne geçişte de ayakta kalır; bu, Bölüm 14'ün ele aldığı bir geçiştir. Bütün bir pipeline'ı tek bir broker'ın transactional sınırı içinde tutabiliyorsanız, özellik işi azaltabilir; ben yine de consumer'ları ona bağımlı olmayacak şekilde tasarladım.

Not: Producer'lar da, bir consumer devreye girmeden önce duplicate yaratabilir. Bir publish acknowledgment'ını beklerken timeout'a düşen bir producer genellikle yeniden publish eder ve broker artık bir event'in iki kopyasını tutuyor olabilir. Her iki kopya da aynı event_id'yi taşır, yeter ki producer onu event oluşturulurken bir kez atamış olsun ve retry için yeni bir tane üretmemiş olsun. Bazı broker'lar bir producer'ın kendi retry'larını, bir producer oturumu içinde sequence number ile ya da bir zaman penceresi içinde deduplication id ile deduplicate eder. Bu, tek bir bağlantı içindeki retry'ı kapsar. Yeniden başlayıp tekrar publish eden bir process'i, örneğin bir outbox relay'ini kapsamaz; consumer'ın hâlâ event_id'yi kontrol etmesinin sebebi budur. Bölüm 4, event'leri kaybetmeden nasıl publish edileceğini, Bölüm 7 ise duplicate'lerin nasıl tanınacağını ele alır.

Broker'ın gerçekte neyi garanti ettiği

Yukarıdaki her şey broker'dan consumer'a olan tarafı anlatır. Producer'dan broker'a olan tarafın kendi sınırları vardır ve bunlar ürün adından çok yapılandırmaya bağlıdır.

Producer acknowledgment'ları

Bir producer publish ettiğinde broker, farklı anlarda acknowledgment verebilir ve ayarları birbirinden ayırmanın yararlı yolu, producer "tamam" cevabını duyduğunda mesajı kaç node'un tuttuğuna bakmaktır.

Acknowledgment yoksa producer, depolama hakkında hiçbir şey öğrenmez. Alıcı node'un acknowledgment'ında, mesajı diske flush edip etmediğinden bağımsız olarak, bir node mesajı kabul etmiştir. Replication sonrasındaki acknowledgment'ta mesaj, gereken sayıda node'da vardır. Diske flush etmek ayrı bir sorudur ve aşağıda gelir.

Producer'ın seçimi, bir çökmenin neler yapabileceği açısından önemlidir. İlk iki ayarda, ack verdikten hemen sonra ölen bir broker node'u mesajı da beraberinde götürebilir. Replicated acknowledgment'ta cevap daha geç gelir ve kayıp penceresi daha küçüktür. Bedeli, her publish için daha yüksek latency ve yazmayı tamamlamaya yetecek sayıda node'un erişilebilir olmasına bağlıdır.

Queue ailesinde publisher confirm'ler, log ailesinde ise acknowledgment seviyeleri sunan broker'lar aynı fikri tarif ediyor. Beklediğiniz acknowledgment'ın sizin yapılandırmanızda gerçekte ne anlama geldiğini kontrol edin. Bence bu, bir ekibin bir kez okuyup not alabileceği en değerli dokümantasyon sayfalarından biridir.

Replication ve fsync

Replication, tek bir node'un kaybına karşı korur: mesaj birkaç node'da vardır. Her şeyin birlikte başarısız olmasına karşı korumaz ve garantisi, producer'a "tamam" denmeden önce kaç kopyanın var olması gerektiğine bağlıdır. Hem acknowledgment seviyesini hem de bunun gerektirdiği minimum replica sayısını kontrol edin, çünkü diğerleri geride kaldığında "tüm replica'lar" tek bir replica anlamına gelebilir. Ayrıca bir failover'ın güncel olmayan bir replica'yı terfi ettirip ettirmeyeceğine bakın, çünkü bu, producer'a güvende olduğu söylenen veriyi kaybettirebilir.

Disk hakkında, ilgili ve çoğu zaman yanlış anlaşılan bir nokta var. Bir dosyaya yazmak normalde veriyi işletim sisteminin cache'ine koyar ve program, genellikle fsync denen senkron bir flush ile zorlamadıkça, veri fiziksel diske daha sonra ulaşır. Flush'ı her mesajda zorlamak yavaştır, bu yüzden birçok broker bunu her yazmada yapmaz. Boşluğu kapatmak için replication'a güvenirler ve flush'ı bir takvime göre yaparlar. Belirli bir broker'ın acknowledgment'ının verinin diskte olduğunu ima edip etmediği yapılandırmasına bağlıdır ve bunu varsaymazdım.

Bu ayarlardan hangisinin sisteminize uyduğunu bilmiyorum; bu, kaybolan bir event'in neye mal olduğuna bağlıdır. Kaybolan bir `PaymentCaptured` ile kaybolan bir "sayfa görüntülendi" farklı problemlerdir ve onları farklı yapılandırmak makul olurdu.

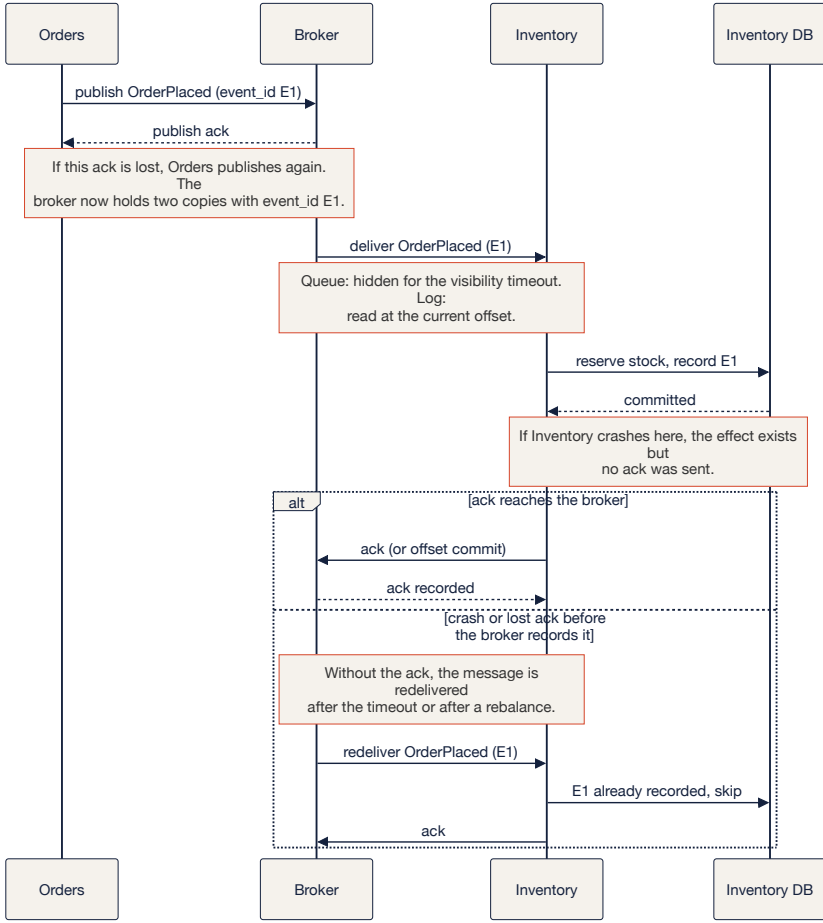
Garantinin sınırı

Bunları bir araya getirdiğinizde sınır netleşir. Bir broker, kabul ettiği bir mesajın tutulacağını ve retention ya da expiry sınırları içinde ve yalnızca yukarıdaki acknowledgment ile replication ayarlarının izin verdiği ölçüde kalıcı olarak consumer'lara sunulacağını vaat edebilir. Bir mesajın broker'a ilk etapta ulaşacağını vaat edemez, çünkü bu, publish'i timeout'a düştüğünde producer'ın ne yaptığına bağlıdır. Bir consumer'ın etkisinin tam bir kez gerçekleşeceğini de vaat edemez.

İki boşluk uygulamanın elinde kalır. Birincisi producer tarafında: Orders veritabanına kaydedilen bir sipariş ile broker'a publish edilen bir `OrderPlaced` event'i iki ayrı yazmadır ve bir çökme ikisinin arasına düşebilir. Bölüm 1 buna dual write adını verdi ve Bölüm 4 onu ortadan kaldırır. İkincisi consumer tarafında, etki ile acknowledgment arasında. Bölüm 7 bu boşluğu kapatır. Bu bölümü açan üç devir açısından bakıldığında broker, ilk ikisi için failure semantics'i belirtir; elinizde kalan iki boşluk ise birinci devirden önceki dual write ile üçüncü devirdeki etki ile acknowledgment arasındaki boşluktur.

Bir mesajın hayatı

Aşağıdaki diyagram, tek bir `OrderPlaced` mesajını Orders servisinden Inventory'deki tamamlanmış bir etkiye kadar izler. Bir hatanın göndereni cevapsız bıraktığı, notlarda işaretlenmiş iki yere bakın: kaybolan publish acknowledgment'ı ve kaybolan consumer acknowledgment'ı. Bölümün anlattığı her şey bu tek dizinin içinde gerçekleşir.



Diyagramın **else** dalı, bölümün asıl noktasıdır. Redelivery bir hata değildir ve hiçbir bileşen yanlış davranmamıştır. Stoğun iki kez rezerve edilmesini engelleyen tek şey, Inventory'nin E1'i rezervasyonla aynı transaction'da kaydetmesi ve ikinci teslimatta tekrar kontrol etmesidir.

Bu özelliğe sahip bir consumer'ın pseudocode olarak görünümü şöyledir. Consumer'ın kendi veritabanında, işlenmiş **event_id** değerlerinden oluşan bir tablo olan bir **inbox** kullanır ve acknowledgment'ı commit'ten sonraya koyar.

```

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)

```

At-least-once'ı güvenli kılan tasarım budur ve burada yalnızca ack'in nereye ait olduğunu göstermek için tanıtılmıştır. Bölüm 7, yalnızca inbox kontrolünün yetmediği durumlar dahil, bunu ayrıntısıyla kurar.

Trade-off'lar ve maliyetler

Semantik seçmek, hangi hatayı yönetmeyi tercih edeceğinizi seçmektir. Seçeneklerin hiçbirisi hatayı ortadan kaldırmaz ve her biri onu başka bir yere taşır.

At-least-once, her consumer'da idempotency işine mal olur. Her consumer'ın tekrarlanan işi tanımanın bir yoluna ihtiyacı vardır; bu genellikle kendi veritabanındaki bir inbox tablosudur. Bu, depolama, hot path üzerinde bir unique constraint ve bir temizlik politikası demektir. Bir düzine consumer'ı olan bir ekip için bir düzine kez uygulanacak bir örüntüdür. Bir consumer'ın etkisi, bir durumu sabit bir değere ayarlamak gibi doğal olarak idempotent olduğunda inbox gereksiz olabilir ve yine de nedenini yazardım.

At-most-once, bütünlüğe mal olur. Basit ve hızlıdır ve yalnızca birisi önceden, kayıp bir mesajın duplicate'ten ucuz olduğuna karar verdiğinde kabul edilebilir. Bu karar açık olmalı ve kaydedilmelidir. Hatanın olağan

biçimi, bunu fark etmeden, auto-acknowledgment ayarları aracılığıyla varsayılan olarak seçen bir ekiptir.

Bir broker içindeki transactional processing, taşınabilirliğe mal olur.

Sınırı içinde duplicate ele almayı azaltabilir ve tasarımı bir ürünün özelliklerine, her katılımcının bu sınırın içinde kalmasına ve onu işletmenin operasyonel çabasına bağlar. Bir ilişkisel veritabanına ya da harici bir API'ye geçtiğinizde zaten idempotency'ye ihtiyacınız olur. Özelliği reddetmezdim ve doğruluğu yalnızca onun üzerine kurmazdım.

Güçlü producer acknowledgment'ları latency'ye ve availability'ye mal olur. Birkaç replica'yı beklemek her publish'i yavaşlatır ve çok az node'a erişilebildiğinde yazmanın başarısız olmasına yol açar. Kaybolmaması gereken event'ler için bu trade-off genellikle doğrudur. Yüksek hacimli, düşük değerli telemetri için olmayabilir.

Log'lar consumer tarafı defter tutmaya, queue'lar geçmişe mal olur.

Bir log seçerseniz, offset commit politikası ve bir partition'ı bloke eden tek bir mesajın ele alınması sizin işinizdir. Bir queue seçerseniz replay'den vazgeçersiniz. Tek tek mesajların bağımsız retry'larına ihtiyaç duyduğunuzda ve geçmişe ihtiyaç duymadığınızda, bir log için hak ettiğinden fazla mekanizma olabilir.

Bu bölümün tavsiyesinin ne zaman uygulanmayacağı: tek bir producer'ı, tek bir consumer'ı olan ve ara sıra kayba tahammül eden bir sistem, örneğin dahili bir metrik akışı, ne bir inbox'a ne de güçlü replication'a ihtiyaç duyar. Garantilerin maliyeti, hatanın maliyetiyle eşleşmelidir. Hata, daha zayıf bir garanti seçmek değildir; hata, bunu yaptığınızı bilmemektir.

Sık yapılan hatalar

- **"Teslim edildi"yi "işlendi" saymak.** Cazip, çünkü dashboard teslimatları ve acknowledgment'ları gösterir ve sağlıklı görünen sayılar bunlardır. Teslim edildi, broker'ın mesajı devrettiği anlamına gelir. İşlendi, etkinin commit edildiği anlamına gelir. İkisini de izleyin ve ack'inizi etkiyi takip ettirin.
- **İş bitmeden ack'lemek.** Cazip, çünkü redelivery'yi önler ve queue'yu temiz gösterir. Her consumer çökmesini kayıp bir mesaja çevirir. Etki

commit edildikten sonra ack'leyin ve ortaya çıkan duplicate'leri ele alın.

- **Offset auto-commit'i varsayılan olarak açık bırakmak.** Cazip, çünkü bir demoda çalışır ve kod gerektirmez. Client'a bağlı olarak işlemenizle ilgisiz bir anda commit edebilir, böylece bir çökme rastgele bir dilimi kaybedebilir ya da tekrarlayabilir. Etkiden sonra, açıkça commit edin.
- **Visibility timeout'u sezgiyle ayarlamak.** Cazip, çünkü varsayılan genellikle iyi görünür. Yavaş ama sağlıklı bir çalışmadan kısa bir timeout, eşzamanlı duplicate işlemeye yol açar. Gözlemleyebildiğiniz en yavaş meşru işleme süresine ve bir paya dayandırın ve uzun işler için bilinçli olarak uzatın.
- **Bir broker'ın exactly-once özelliğinin veritabanınızı kapsadığına inanmak.** Cazip, çünkü dokümantasyon bu ifadeyi kullanır. Garanti, broker'ın kendi sınırıyla kapsamlıdır. Dışındaki her şey, örneğin veritabanınızdaki bir satır ya da bir e-posta, idempotency gerektirir.
- **Publish edilmiş bir mesajın saklanmış bir mesaj olduğunu varsaymak.** Cazip, çünkü publish çağrısı döndü. Acknowledgment seviyesine bağlı olarak mesaj yalnızca tek bir node'un belleğinde var olabilir. Producer'ınızın neyi beklediğini bilin ve yazın.
- **Rebalancing'in nadir olduğunu varsaymak.** Cazip, çünkü sessizce gerçekleşir. Deploy'lar, scale olayları ve çökmelerin hepsi, group protokolüne ve üyelik ayarlarına bağlı olarak bir rebalance tetikleyebilir ve her rebalance bir batch mesajı tekrarlayabilir. Redelivery her gün oluyormuş gibi tasarlayın, çünkü yanılmanın maliyeti bir duplicate etkidir.

Temel çıkarımlar

- Bir mesaj üç devirden geçer: producer'dan broker'a, broker'dan consumer'a ve consumer'dan etkiye; ve bir broker'ın garantileri yalnızca ilk ikisini kapsar.
- Queue'lar her mesajı izler ve acknowledgment'ta siler; log'lar ise mesajları okuyuculardan bağımsız tutar ve her consumer group'un kendi offset'ini tutmasına izin verir, replay'i mümkün kılan da budur.
- Acknowledgment ya da offset commit'i nereye koyduğunuz semantiği belirler: işlemeyen önce at-most-once, işledikten sonra at-least-once

verir ve hiçbir konum hem kaybı hem de duplicate'leri önlemez.

- Redelivery istisnai değil rutindir, çünkü visibility timeout'ların süresi dolar, consumer'lar çöker ve rebalance'lar (ki deploy'lar bunlara sık sık yol açar) batch'leri tekrarlar.
- Exactly-once, at-least-once delivery ve bir idempotent consumer'dan kurulan bir etkidir ve bir broker'ın kendi exactly-once processing'i o broker'ın sınırında durur.
- Bir producer acknowledgment'ı, arkasındaki replication ve flush ayarları kadar güçlüdür; bu yüzden ekip, her önemli event için "publish edildi"nin ne anlama geldiğini bilmeli ve kaydetmelidir.
- Her semantik seçenek hatayı ortadan kaldırmak yerine taşır ve doğru seçim, geriye kalan hatayı yönetmeyi göze alabileceğinize karar verdiğiniz seçimdir.

Mesajlaşma semantiği, broker'ın normal bir günde neyi yapıp neyi yapmayacağını söyler. Bölüm 3, anormal günlere döner ve gerçekte neyin bozulduğuna dair pratik bir model kurar: network partition'lar, çökmeler, yavaş consumer'lar, saatler ve insanlar.

Bölüm 3 — Pratik Bir Hata Modeli

Parcelly'de Payments'ı bir capture'ın ortasında düşünün. Orders, Payments'a bir `CapturePayment` komutu göndermiş ve Payments parayı capture etmek için harici ödeme sağlayıcısını çağırmış. Sağlayıcının bağlantısı sessizleşiyor. Otuz saniye sonra Payments'ın HTTP client'ı pes ediyor ve bir timeout fırlatıyor.

Payments şimdi cevaplayamayacağı bir soruyla karşı karşıya. Sağlayıcı, bağlantı kopmadan önce ödemeyi capture etti mi, yoksa isteği hiç almadı mı? İki geçmiş de tamamen aynı gözlemi üretiyor: sessizlik, sonra bir timeout. Payments yeniden denerse ve ilk capture başarılı olmuşsa alıcıdan iki kez tahsilat yapılır. Payments hata bildirirse ve ilk capture başarılı olmuşsa, Orders zaten ödenmiş bir siparişi iptal eder.

Bu hikâyede egzotik hiçbir şey yok. Bug yok, kesinti raporu yok, hata yapan bir mühendis yok. Bir ağ, ağların yaptığını yaptı ve sistemin, iki dünyadan hangisinde yaşadığını söylemenin hiçbir yolu yoktu.

Bir event-driven sistemde ters giden şeylerin çoğu bu biçimdedir: bir adım gerçekleşti ya da gerçekleşmedi ve kanıt eksik. Bu bölüm, kanıtın kaybolma yollarını adlandırıyor; böylece her biri için bilinçli olarak tasarım yapabilirsiniz. Cevapladığı soru şu: Parcelly'ninki gibi bir sistemde tam olarak ne bozulabilir, bozulduğunda ne göreceksiniz ve her hata bu kitapta nerede ele alınıyor?

Problem ve neden zor olduğu

Hata modeli, ters gidebileceğini varsaydığınız şeylerin ve gidemeyeceğini varsaydığınız şeylerin yazılı listesidir. Her sistemin bir tane vardır, kimse yazmış olsun ya da olmasın. Yazılmamış sürüm genellikle iki madde içerir: "servis çöker" ve "ağ yavaştır". Bunlar gerçektir, ama kendini belli eden hatalardır.

Basit vaka, tespit edebildiğiniz bir hatadır; reddedilen bir bağlantı ya da bir veritabanı hatası gibi. Her biri size adımın tamamlanmadığını söyler.

Zor vaka, sizi belirsiz bırakan hatadır. Bir timeout "işlem başarısız oldu" demez. "Beklemeyi bıraktım" der. İşlem başarısız olmuş, başarılı olmuş ya da hâlâ çalışıyor ve siz vazgeçtikten sonra başarılı olmak üzere olabilir. Bunu hissetmek için bir timeout ve karşı uçta bir yan etki yeter.

Bu yüzden bariz yaklaşım, yani happy path için kod yazıp her çağrının etrafına bir `try` ve bir `retry` eklemek, ayakta kalmaz. `Retry`, geçmiş bilmeden gelecek hakkında verilen bir karardır. Önceki deneme başarılı olduysa `retry` bir `duplicate`'tir. `Retry` etmeyi reddederseniz bir mesaj kaybolabilir. İki seçim de kendi başına yanlış değildir; her biri, içinde bulunduğunuz dünyadan ayırt edemediğiniz bir dünyada yanlıştır.

İkinci bir neden var. Hatalar birleşir: bir `deploy` bir `consumer`'ı yeniden başlatırken bir `broker failover` olur ve bir `producer` `retry` eder. Onları tek tek listelemek yine de doğru başlangıçtır, ama `production` onları birlikte teslim eder.

Dolayısıyla bu bölümün işi sınıflandırmadır. Her hata için Bölüm 1'deki üç soruyu soracağım: bu birleşim yerinde ne başarısız olabilir, bunu nasıl gözlemlersiniz ve hangi garanti ayakta kalır. Aşağıdaki maddelerde "gerçekte doğru olan" birincisini, "gördükleriniz" ikincisini, her maddenin sonunda adı geçen örüntü ise üçüncüsünü yanıtlar. Bu üç cevap birlikte bir hatanın semantiğidir ve kitabın geri kalanı bunları sormaya devam eder.

Önemli on hata

Aşağıdaki liste, servislerden, ilişkisel veritabanlarından ve bir `message broker`'dan kurulu bir sistem için başlayacağım kümedir. Tam değildir ve hiçbir liste tam değildir.

Network partition'lar ve timeout belirsizliği

Açılış senaryosu örnek vakadır. İki taraf bir ağla bağlıdır ve ağ trafiği düşürür, geciktirir ya da `reorder` eder. **Network partition**, iki makine grubunun birbirine hiç erişemediği uç biçimdir. Timeout gündelik biçimdir ve çok daha yaygındır.

Her tarafın gözlemleyebildikleri asimetriktir. Çağırın yalnızca cevabın gelmediğini görür. Çağrılan işi yapmış ve cevabı kaybetmiş olabilir ya da isteği hiç görmemiş olabilir. Aynısı Parcelly'deki her hop için geçerlidir: harici bir kargo firmasını çağırın bir servis, broker'a gönderen bir producer, broker'a ack veren bir consumer.

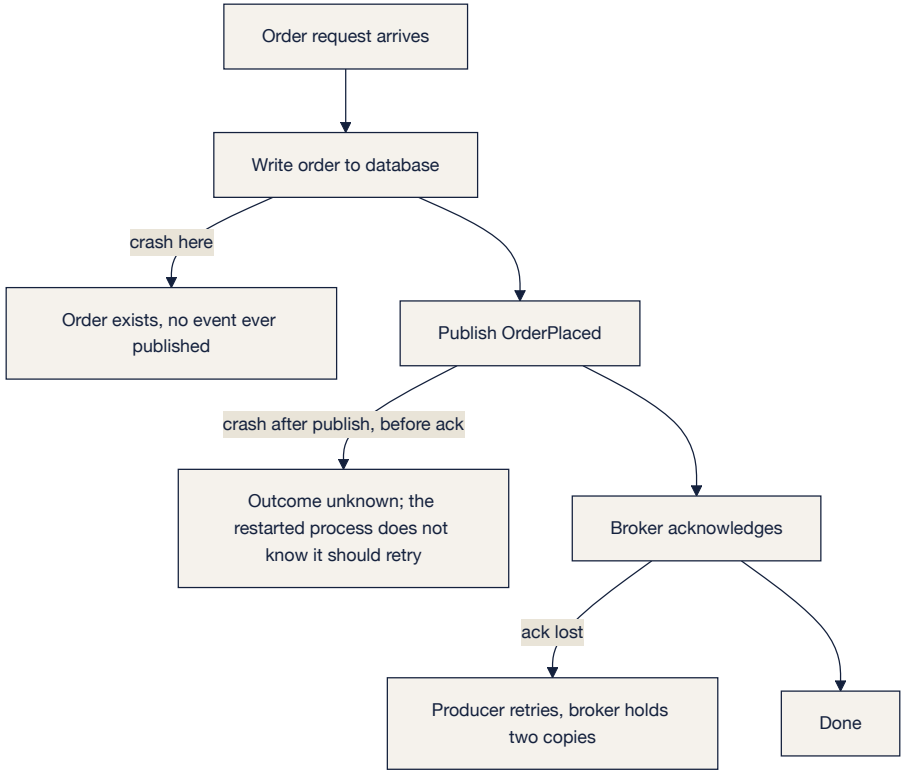
Buna verilebilecek tek dürüst cevap, işlemi tekrar edilmesi güvenli hâle getirmektir. Payments, sağlayıcıya her capture ile sabit bir idempotency key gönderirse ve sağlayıcı tekrarlanan bir key'i aynı request olarak ele alırsa, timeout sonrası retry zararsız olur. Tasarım, "oldu mu?" sorusunun cevabına bağımlı olmaktan çıkar, çünkü iki cevap da aynı nihai duruma götürür. Bölüm 6, ne zaman ve nasıl retry edileceğini, Bölüm 7 ise retry'ı güvenli kılan idempotency'yi ele alır.

Bazı harici sistemler böyle bir key sunmaz. O zaman tekrar harekete geçmeden önce sisteme ne olduğunu sorarsınız; bu bir reconciliation adımıdır ve Bölüm 13 onu ele alır.

Producer çökme pencereleri

Orders'ın yeni bir siparişi ele aldığını düşünün. Siparişi kendi veritabanına kaydetmeli ve `OrderPlaced`'i broker'a publish etmelidir. Bunlar iki sistemdir ve hiçbir transaction ikisini kapsamaz. Bu, Bölüm 1'deki dual write'tır ve ilginç olan, her çökme penceresinin geride ne bıraktığıdır.

Process iki yazma arasında çökebilir ve yazmaların sırası, hangi hasarı alacağınızı belirler.



Diyagram, önce veritabanına yazan sıralamayı çizer ve üç pencere gösterir: publish'ten önce bir çökme, ondan sonra sonucu bilinmeyen bir çökme ve kaybolan bir acknowledgment. Veritabanı commit'i önce gelirse ve process publish etmeden ölürse sipariş vardır ve downstream'de kimse onu duymaz. Payments hiç authorize etmez, Inventory hiç rezerve etmez ve alıcı hiç gelmeyen bir onayı bekler. Hiçbir şey hata vermez. Sipariş orada durur.

Diyagramın çizmediği ters sıranın kendi penceresi vardır. Önce publish ederseniz, commit'ten önceki bir çökme daha kötü bir durum bırakır: hiç doğru olmamış bir olguyu duyuran bir event. Downstream servisler var olmayan bir sipariş üzerinde hareket eder. Ve iki sırada da, broker'ın acknowledgment'ı dönüş yolunda kaybolursa producer retry eder ve broker aynı event'in iki kopyasını tutuyor olabilir.

Dışarıdan gördüğünüz şey sessizliktir: event'siz siparişler, siparişsiz event'ler ve ara sıra duplicate'ler. Log'lardaki hiçbir şey "kayboldu" demez. Bölüm 4, durum değişikliğini ve publish etme niyetini tek bir yerel

transaction yaparak pencereyi ortadan kaldıran transactional outbox'ı tanıtır. Bölüm 5, outbox satırlarının ardından broker'a nasıl yolculuk ettiğini ele alır.

Broker failover

Broker'lar, bir makine verileri de beraberinde götürmeden ölebilsin diye replicate edilir. Bir partition'ın ya da queue'nun lider kopyasını tutan makine ölünce bir diğeri devralır. Bu bir **failover**'dır ve bir arıza değil, tasarlanmış bir davranıştır.

Önemli olan, producer'a failover'dan önce ne söylendiğidir. Bir lider ölmeden birkaç an önce ack'lenen bir mesajın hayatta kalıp kalmadığı, broker'ın nasıl ack verecek şekilde yapılandırıldığına bağlıdır: hiç beklemeden, alıcı node mesajı kabul ettikten sonra (diske flush etmiş olsun ya da olmasın) ya da gereken sayıda replica onu aldıktan sonra. En güçlü ayarlar latency'ye ve throughput'a mal olur. En zayıf ayarlar, bir failover'da ack'lenmiş mesajları kaybedebilir. Bu, broker aileleri ve yapılandırmalar arasında farklılık gösterir; bu yüzden burada belirli bir ürün için belirli bir davranış öne sürmeyeceğim; nokta, garantinin bir sınırı olduğu ve sınırın birinin seçtiği bir ayar olduğudur.

Failover sırasında producer'lar da hata ve timeout görür, retry eder ve eski liderin zaten kabul ettiği mesajların duplicate'lerini publish edebilir. Bölüm 2, broker garantilerinin nerede durduğunu anlatır. Buradaki kural, "broker tarafından ack'lendi"yi kanıt değil, güçlü bir delil olarak ele almaktır.

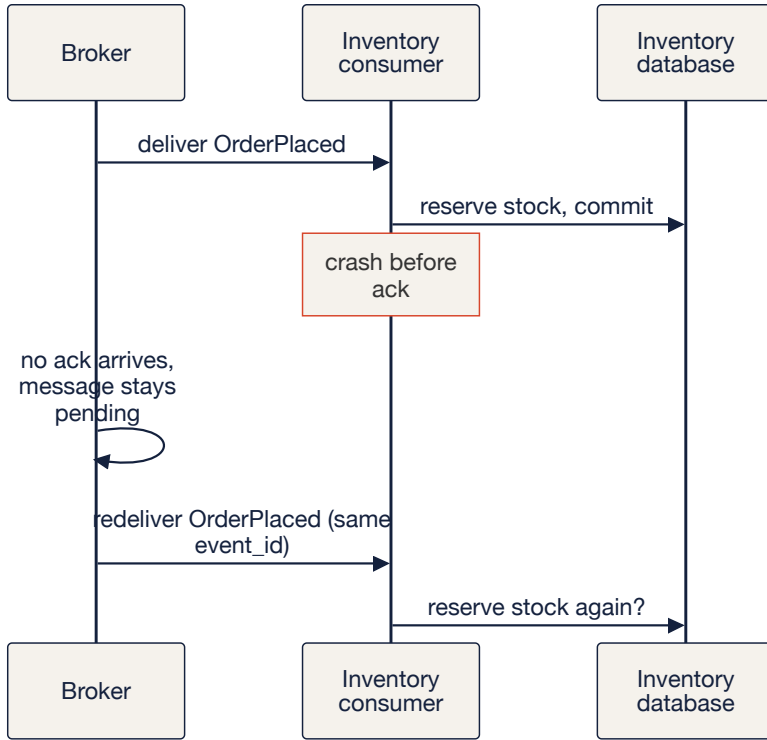
Gördükleriniz: bir publish hatası patlaması, consumer throughput'unda bir düşüş, ardından bir redelivery dalgası. Bir mesaj kaybolduysa hiçbir şey görmezsiniz; Bölüm 13'ün reconciliation'ı bu yüzden önemlidir.

Consumer çökmesi: işlendi ama ack'lenmedi

Şimdi consumer tarafı. Inventory `OrderPlaced`'i alır, veritabanında stoğu rezerve eder ve commit eder. Sonra process, mesajı ack'leyemeden ölür.

Broker'ın bakışından mesaj hiç bitmemiştir. Queue tabanlı bir broker'da broker kapanan bağlantıyı fark eder ya da visibility timeout dolar ve mesaj yeniden teslim edilir. Log tabanlı bir broker'da consumer'ın offset'i hiç

commit edilmemiştir, dolayısıyla partition'ın sıradaki sahibi eski pozisyonundan başlar ve mesajı tekrar okur. İki ailede de ikinci teslimat, zaten yapılmış bir iş için gelir.



Diyagramın son satırı problemin tamamıdır. Her teslimatta koşulsuz olarak stok rezerve eden bir consumer, onu iki kez rezerve eder. Duplicate'lerin olağan kaynağı budur ve kitabın varsayılanının at-least-once delivery artı idempotent consumer'lar olmasının nedeni de budur. Bir consumer farklı noktalarda çökebilir ve her noktanın farklı bir sonucu vardır:

```

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.

```

Bu, Bölüm 2'deki handler'ın, çökme noktalarını gösterecek şekilde kısaltılmış sürümüdür. A çökme noktasının etkisi hiçbir şeydir. C çökme noktasının etkisi hiçbir şeydir. İlginç olan B'dir ve onu yalnızca, işlediği şeyi işle aynı transaction'da kaydeden bir consumer atlatır. Bölüm 7 bunu düzgünce kurar.

Gördükleriniz: iki rezervasyon, iki e-posta, iki ledger entry gibi duplicate yan etkiler ve hiçbir yerde hata yok, çünkü her işleme denemesi başarılı oldu.

Yavaş consumer'lar ve backpressure

Her hata bir çökme değildir. Yalnızca yavaş olan bir consumer da aynı ölçüde zarar verebilir ve hiçbir şey bozuk olmadığı için fark etmek daha zordur.

Notifications'ın, request başına 100 milisaniye yerine 10 saniye almaya başlayan bir e-posta sağlayıcısını çağırdığını varsayın. Notifications hâlâ çalışır, ama her mesaj artık bir worker'ı 100 kat daha uzun meşgul eder. Mesajlar bitirildiklerinden daha hızlı gelir, bu yüzden Bölüm 2'nin tanımladığı consumer lag istikrarlı biçimde büyür.

Yavaşlık, ardından ikincil hataları tetikler. Queue tabanlı bir broker'da, visibility timeout'unu aşan bir mesaj, ilk worker hâlâ onu işlerken yeniden belirir ve ikinci bir worker aynı mesaja başlar. Duplicate'ler zaten yavaş

olan bir sisteme yük ekler. Log tabanlı bir broker'da, broker'a zamanında geri dönmeyen bir consumer ölü ilan edilebilir ve partition'ları yeniden atanır; bu da bu listedeki bir sonraki hatayı başlatır.

Backpressure, yükün sonlu kalması için her aşamada devam eden işi sınırlama disiplini. Broker aracılı bir sistemde farkı broker emer, bu da temporal decoupling'in faydasıdır, ama buffer sınırlıdır ve iki farklı şekilde tükenir. Broker'ın diski dolduğunda producer'ları yavaşlatabilir ya da bloke edebilir, hatta hizmet vermeyi tamamen durdurabilir; broker'ınızın bunlardan hangisini yaptığı, olacağı günden önce öğrenilmesi gereken bir şeydir. Hangisi olursa olsun producer bunu görür ve bu gerçek backpressure'dır: gürültülü ve görünür. Bir consumer bir log'un retention süresinden daha fazla geride kaldığında ya da queue'daki bir mesaj expiry'sinden uzun yaşadığında broker, hiç okunmamış mesajları siler. Bu gecikme değildir. Sessiz kayıptır: hata verilmez, hiçbir consumer başarısız olmaz ve okunmamış `PaymentCaptured` event'leri gitmiştir. İki sınırdan çok önce bile bir backlog, capture edilmeden süresi dolan bir ödeme authorization'ı gibi, bir iş deadline'ını ihlal etmiş olabilir.

Broker'ın buffer'ı sorunu yalnızca ertelediği için yükün aynı anda birkaç yerde sınırlanması gerekir. Bir consumer, uçustaki mesaj sayısını ya da prefetch ettiği batch'i sınırlayabilir. Worker concurrency'sini sınırlayabilir; böylece yavaş bir e-posta provider'ı worker'ların hepsini değil, sabit bir sayısını meşgul eder. Yavaş dependency'ye yönelik çağrılarını rate-limit edebilir ya da dependency toparlanana kadar tüketimi tamamen duraklatabilir. Bir queue'nun, açık bir overflow politikasıyla bir uzunluk limiti olabilir; böylece neyin düşürüleceğini ya da reddedileceğini broker'a bırakmak yerine siz seçersiniz.

Producing tarafı ve topoloji de limitler koyar. Producing tarafında outbox, publish'leri reddeden bir broker'ı emer; çünkü satırlar, relay onları teslim edebilene kadar veritabanında bekler. Log tabanlı bir broker'da ise partition sayısı, bir group'taki kaç consumer'ın paralel çalışabileceğini sınırlar; dolayısıyla instance eklemek, önceden belirlenmiş bir sayıda fayda sağlamayı bırakır. Bu limitlerin her biri throughput'tan stabilite için ödün verir ve her birinin bir metric'e ihtiyacı vardır; yoksa sizi geride tutanın o olduğunu bilemezsiniz.

Gördükleriniz: yükselen lag ve mesaj yaşı, büyüyen bir backlog, tırmanan redelivery sayıları. Bölüm 6 backoff'u, uçustaki iş limitlerini ve tüketimi duraklatmayı, Bölüm 8 zamanında işlenemeyen mesajlarla ne yapılacağını, Bölüm 13 ise sizi erkenden uyaran lag ve yaş metric'lerini ele alır.

Her seferinde başarısız olan mesajlar ve her mesaj için başarısız olan dependency'ler

Şimdiye kadar her hata bir şans meselesiydi: bir çökme, kaybolan bir cevap, kötü bir an. Bazı hatalar ise deterministiktir. **Poison message**, zamanlama ne olursa olsun her denemede başarısız olan bir mesajdır. Inventory'ye, kataloğunda bulunmayan bir SKU'yla bir `OrderPlaced` geldiğini varsayın. Inventory bir hata fırlatır, mesaj yeniden teslim edilir, hata tekrarlanır ve bir sonraki denemede hiçbir şey öncekinden farklı değildir.

Neden mesajın dışında da olabilir. Payments'ın provider'ı çökmüşse her capture başarısız olur ve kesinti sürdüğü sürece her mesaj zehirli görünür. İki durum zıt tepkiler gerektirir. Hiçbir zaman başarılı olmayacak bir mesaj ana akıştan hızla çıkmalı, bir **dead-letter queue (DLQ)**'ya, yani işlenemeyen mesajların bir insanı ya da bir onarım işini beklediği ayrı bir yere gitmelidir. Bir dependency çöktüğü için başarısız olan bir mesaj ise hiç dead-letter'a gönderilmemelidir; çünkü toparlanmadan sonra başarılı olacaktır ve bir kesinti sırasında bütün bir backlog'u dead-letter'a göndermek geçici bir sorunu manuel bir soruna çevirir.

Arada ters giden şeyler bir örüntü izler. Bir poison message üzerindeki retry'lar, başarı şansı olmadan denemeleri ve kapasiteyi tüketir. Log tabanlı bir broker'da ya da sırayı koruyan bir queue'da böyle tek bir mesaj, arkasındaki her şeyi bloke edebilir. Tek bir başarısız dependency'ye karşı retry eden çok sayıda consumer, toparlanmaya çalışan şeye yük bindirir; bu bir retry storm'dur. Bir kod bug'ı ise hiçbir hata vermeden de başarısız olabilir: consumer ack'ler, döner ve yanlış değeri yazar; böylece hata yolundaki hiçbir şey çalışmaz.

Gördükleriniz: log'larda aynı `event_id`'nin tekrar tekrar görünmesi, hiç düşmeyen redelivery sayıları, takılı kalmış bir partition, bir kesinti sırasında büyüyen bir DLQ ve sessiz bug için, bir reconciliation kontrolü uyuşmazlık bulana kadar hiçbir şey. Bölüm 6 hangi hataların retry etmeye değer olduğunu nasıl ayırt edeceğinizi, Bölüm 8 poison message'ları ve

dead-letter yolunu, Bölüm 13 ve 14 ise hata vermemiş bir bug'ın hasarını bulmayı ve onarmayı ele alır. Her sınıflandırmanın bedeli yanlış olabilmektedir; bu yüzden kararı görünür ve geri alınabilir tutun.

Güvenemeyeceğiniz saatler ve timestamp'ler

Her makinenin bir saati vardır ve hiçbir iki saat tam olarak uyuşmaz. Kayarlar, senkronizasyonla düzeltilirler ve bir düzeltme saati geriye alabilir. Duraklatılmış bir sanal makine, uyuduğu süre kadar yanlış bir saatle uyanır.

Event'ler için sonuç spesifikidir. `occurred_at`, bir şeyin tek bir makineye göre ne zaman olduğuna dair bir olgudur. Orders'dan gelen bir event'in `occurred_at`'ini Payments'tan gelenle karşılaştırmak, event'lerin sırası kadar saatleri hakkında da bir şey söyler. Hangi güncellemenin kazanacağına karar vermek için timestamp'e göre sıralarsanız, saati birkaç saniye ileri olan bir servis, kaybetmesi gereken her çatışmayı kazanır.

Timeout'ların da ilgili bir tuzağı vardır. Duvar saati zamanı olarak ifade edilen bir deadline, "bu authorization 14:00'e kadar geçerli", 14:00 fikri düzenleyenden farklı olabilen bir makine tarafından karşılaştırılır. Tek bir makinede süreleri ölçmek için yalnızca ileri giden bir monotonic clock doğru araçtır. Makineler arası ordering için duvar saati zamanını hiç kullanmazdım.

Event'leri güvenilir biçimde sıralayan şey sistemin yapısıdır: log tabanlı bir broker'da aggregate ile key'lenmiş bir partition, ki bu, her key için tek bir producer yolu olduğu ve partition sayısı değişmediği sürece o key üzerindeki event'ler için toplam bir sıra verir, ya da consumer'ların karşılaştırabileceği, aggregate üzerinde açık bir versiyon numarası. Bölüm 9 ordering'i derinlemesine ele alır. `occurred_at`'i insanlar ve analitik için bilgi olarak ele alın ve doğruluk için bir girdi olarak ondan çekinin.

Gördükleriniz: sırasız uygulanan güncellemeler, bir etkinin nedeninden önce gelir görüldüğü log'larda "imkânsız" diziler ve yalnızca tek bir host'ta ortaya çıkan bug'lar.

Deploy'lar ve rebalance'lar

Bir deploy planlı bir hatadır. Instance'lar durdurulur ve başlatılır ve arada iş, yarı yolda kesilir. Deploy süreciniz bir sonlandırma sinyali gönderirse ve consumer, işlenmekte olan mesajlarla çıkarsa, "işlendi ama ack'lenmedi" durumunu bilerek, tekrar tekrar, her release'te yaratmış olursunuz.

Rebalance, bunun log ailesindeki sürümüdür ve bir deploy olmadan da olabilir. Üyeler bir consumer group'a katıldığında ya da ayrıldığında ya da bir üye yanıt vermiyor sayıldığında partition'lar yeniden atanır. Devir sırasında iki consumer kısa süre aynı partition'a sahip olduklarına inanabilir ve yavaş olan bir consumer, partition'ı başka bir yere taşıdıktan sonra mesajları işlemeye devam edebilir. Queue ailesindeki karşılığı, iş sürerken visibility timeout'ları dolan competing consumers'tır.

Hasar örüntüsü, deploy zamanında bir duplicate patlaması ve bazen partition'lar yeniden atanırken kısa bir duraksamadır. Bütün instance'lar boyunca hızla art arda ilerleyen bir deploy, arka arkaya birkaç rebalance tetikleyebilir ve en kötü durumda consumer'lar işi yapmaktan çok yeniden atamakla zaman harcar. İki önlem bunu azaltır, ama anlamamız gereken daha fazla yapılandırma getirirler: yalnızca taşınması gereken partition'ların alındığı cooperative reassignment ve yeniden başlatılan bir instance'ın eski partition'larını tam bir yeniden dağıtım olmadan geri alabilmesi için kararlı bir üye kimliği.

Consumer'ların çıkmadan önce işlenmekte olan mesajları bitirerek ya da serbest bırakarak zarifçe durmasını sağlayın ve zarif yolun bazen başarısız olacağını varsayın; böylece idempotent consumer gerçek savunma olarak kalır. Rolling deploy'ları, sıkı bir döngüde yeniden atamaları tetiklemeyecek şekilde aralıklandırırız da. Gördükleriniz: release zamanlarıyla örtüşen duplicate'ler ve lag sığramaları.

İnsan hataları

Son kategori, çoğu mimari dokümanının dışarıda bıraktığıdır. Bu sistemleri insanlar işletir ve insanlar, genellikle önceki bir hatayı onarırken, baskı altında hata yapar.

Parcelly'de 40.000 dead letter bulan ve hepsini redrive etmek için düğmeye basan nöbetçi mühendisi düşünün. Neden düzeltildiyse akıp geçerler. Düzeltilmediyse hemen yeniden başarısız olurlar ve mühendisin

iyileştirmeye çalıştığı sistemi su altında bırakırlar. Bazıları manuel bir geçici çözümle zaten ele alındıysa, birinin elle yaptığı işin duplicate'leridir.

Diğer biçimler de aynı ölçüde sıradandır. Biri bir müşterinin önünü açmak için bir satırı doğrudan düzenler, outbox'ı atlar, dolayısıyla hiçbir event publish edilmez ve downstream servisler birbirinden ayrışır. Biri, consumer'lar okuyamadan değişmiş bir payload'la bir producer deploy eder. Biri yanlış ortamda bir queue'yu purge eder. Bunlar, onarım araçları geniş ve güvenceler eksik olduğunda olanlardır.

Operatör eylemlerini hata modelinin parçası olarak ele alır ve onları diğer her bileşen gibi tasarladım: bir pause switch'iyile küçük batch'ler hâlinde redrive, event'lerin yayılması için normal değişikliklerle aynı yoldan geçen onarım eylemleri ve her manuel eylemin bir audit trail'e yazılması. Bölüm 8 dead letter'ları ve redrive'ı, Bölüm 12 payload değişikliklerini, Bölüm 13 ise bir insan düzeltmesinin ters gittiğini öğrenme yolunuz olan reconciliation'ı ele alır.

Gördükleriniz: birinin bir şeye dokunmasından hemen sonra ani bir sıçrama ve açıklayacak başarısız bir mesaj olmaksızın servisler arasında uyuşmayan veri.

Bütün bir store'u kaybetmek

Şimdiye kadarki her hata, store'ların hayatta kaldığını varsaydı: veritabanı commit ettiğini tutar ve broker ack'lediğini saklar. Son madde bu varsayımı bırakır. Bir cluster ya da bir region kaybedilir. Bir broker, replication'ının kapsadığından fazla veri kaybeder. Bir consumer'ın veritabanı, altı saat önce alınmış bir backup'tan geri yüklenir. Bir producer'ın veritabanı, son birkaç yüz sipariştin önceki bir noktaya geri yüklenir.

Bunlar tek bir hata değildir; çünkü hasar, hangi store'un zamanda geriye gittiğine bağlıdır. Inventory'nin veritabanı geri yüklenirse, geri yükleme noktasından sonra yapılan rezervasyonları unuttur ve broker, onları yeniden yaratacak event'leri artık tutmuyor olabilir. Orders'ın veritabanı geri yüklenirse, downstream servislerin zaten üzerinde işlem yaptığı siparişleri unuttur ve Payments, Orders'ın artık tanımadığı siparişler için capture'lar tutar. Hiçbir taraf hata vermez. Her biri kendi içinde tutarlıdır ve diğeri hakkında yanlıştır.

Bir özellik yardımcı olur. Inbox, etkilerle aynı veritabanında yaşadığı için bir geri yükleme ikisini de aynı ana geri götürür. Geri yükleme noktasından önce işlenen mesajlar hâlâ inbox'tadır, dolayısıyla bir replay onları atlar; bu doğrudur, çünkü etkileri de hayatta kalmıştır. Geri yükleme noktasından sonra işlenen mesajlar risk altındaki mesajlardır ve onları replay etmek güvenlidir: inbox onları artık hatırlamaz ve etkileri onunla birlikte kaybolmuştur.

Dış etkiler istisnadır. Ödeme provider'ındaki bir capture, veritabanınızla birlikte geri alınmaz; bu yüzden bu çağrıların kendi idempotency key'lerine ihtiyacı vardır, yoksa replay alıcıdan ikinci kez tahsilat yapar. Kural, tüketimi sürdürmeden önce reconcile etmektir: her servisin neye inandığını karşılaştıran, farkları onarın ve ancak ondan sonra akışın çalışmasına izin verin.

Gördükleriniz: başlangıçta hiçbir şey ve sorun da budur. Servisler sağlıklıdır ve dashboard'lar yeşildir, oysa toplamalar aralarında uyuşmaz. Bölüm 13 uyuşmazlığı bulan reconciliation'ı, Bölüm 14, Recovery, Felaketler ve Broker Migration ise nasıl restore edeceğinizi, replay edeceğinizi ve devam edeceğinizi ele alır. Bunun için tasarım yapmak, gerçekten test edilmiş backup'lara, recovery sürenizden uzun süren bir retention'a ve ihtiyaç duymadan önce bir restore'u prova etme disiplinine mal olur.

Hata haritası

İşte aynı malzeme tek bir yerde. Tasarım review'larınızın yanında tutun.

Hata	Ne görürsünüz	Gerçekte doğru olan	Nerede ele alınır
Network partition ya da timeout	Çağırın cevap alamaz	İşlem çalışmış da olabilir, çalışmamış da	Bölüm 6, Bölüm 7, Bölüm 13
Yazma ile publish arasında producer çökmesi	Event'siz siparişler ya da siparişsiz event'ler	Durum ve event uyuşmuyor	Bölüm 4, Bölüm 5
Kaybolan broker acknowledgment'tı	Broker'da duplicate event'ler	İlk publish başarılı oldu	Bölüm 7

Hata	Ne görürsünüz	Gerçekte doğru olan	Nerede ele alınır
Broker failover	Publish hataları, sonra redelivery'ler; nadiren sessiz kayıp	Garanti, acknowledgment ayarlarına bağlı	Bölüm 2, Bölüm 13
İşten sonra consumer çökmesi	Duplicate yan etkiler, hata yok	İş yapıldı, mesaj hâlâ pending	Bölüm 7
Yavaş consumer	Büyüyen lag ve mesaj yaşı	Consumer canlı, geride kalıyor	Bölüm 6, Bölüm 8, Bölüm 13
Geride kalırken retention ya da expiry	Hiçbir şey; eksik event'ler hiç gelmez	Okunmamış mesajlar broker tarafından silindi	Bölüm 8, Bölüm 13
Poison message ya da başarısız dependency	Aynı mesajın yeniden teslimi; takılı partition; kesinti sırasında büyüyen DLQ	Hata deterministik ya da neden mesajın dışında	Bölüm 6, Bölüm 8
Hatasız consumer bug'u	Hiçbir şey; toplamalar sonradan uyuşmaz	Handler başarılı oldu ve yanlış sonucu yazdı	Bölüm 13, Bölüm 14
Clock skew	Sırasız güncellemeler, imkânsız diziler	Timestamp'ler nedensellik uyuşmuyor	Bölüm 9
Deploy ya da rebalance	Release zamanında duplicate'ler ve lag	İşin sahipliği yarı yolda taşındı	Bölüm 7, Bölüm 13
Operatör hatası	Manuel bir eylemden sonra sıçrama; sessiz ayrışma	Bir onarım, durumu normal yolun dışında değiştirdi	Bölüm 8, Bölüm 13
Bütün bir store'u kaybetmek	Başlangıçta hiçbir şey; restore sonrası servisler uyuşmaz	Bir store zamanda geriye gitti ya da verisi yok	Bölüm 13, Bölüm 14

İki şey öne çıkıyor. Birincisi, duplicate'ler "Ne görürsünüz" sütununda üç farklı neden için görünür (kaybolan acknowledgment, consumer çökmesi, deploy) ve redelivery'ler dördüncüsü için. Tek bir örüntünün, idempotent consumer'ın, neden bütün bir bölümü hak ettiği budur. İkincisi, son sütun sürekli Bölüm 13 ve 14'e dönüyor. Bazı hatalar oluştuğları anda önlenemez ve hatta tespit bile edilemez; bu yüzden sistemin, gerçeğin iki görünümünün uyuşmadığını sonradan fark etmenin bir yoluna da ihtiyacı vardır.

Hata modelini yazıya dökmek

Bir hata modeli yalnızca yazılı, güncel ve paylaşılmışsa yararlıdır. Bence pratik, tek tek herhangi bir maddeden daha değerlidir, çünkü yazma eylemi "burada ne varsayıyoruz?" sorusunu zorlar ve sürprizler varsayımlarda yaşar.

Onu, mimari açıklamanın yanında, repository'de kısa bir doküman olarak tutardım; her hata için bir madde, her seferinde aynı alanlarla. Alanlar üç soruyu yansıtır: "where" ve "we assume" neyin başarısız olabileceğini, "observable as" bunu nasıl göreceğinizi, "defense" ise hangi garantinin ayakta kalacağını söyler. Kullanılabilir bir madde şöyle görünür:

```
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
```

Önemli olan alanlar son üçüdür. "Verified by", maddeyi bir inanıştan test edilebilir bir iddiaya çevirir ve Bölüm 15 bunu nasıl test edeceğinizi gösterir. "Owner", birinin page edildiği anlamına gelir. "Last reviewed", modelin çürüdüğünü kabul eder: servisler eklenir, broker'lar değiştirilir ve geçen yıl doğru olan bir varsayım şimdi doğru olmayabilir.

Not: Model, olamayacağını varsaydıklarınızı da belirtmelidir. "Veritabanının commit edilmiş bir transaction'ı asla kaybetmediğini varsayıyoruz" meşru bir maddedir. Varsayımı yazıp, yanlış olsa ne yapacağınızı bilmek, onu örtük bırakmaktan iyidir.

Modeli üç yerde kullanın. Tasarım review'larında, her yeni akışı listeye karşı yürüyün ve hangi hatalara maruz kaldığını sorun. Incident review'larında, incident'in hangi maddeye ait olduğunu bulun ve hiçbirisi değilse bir tane ekleyin. Testte, her maddeyi bir senaryo yapın. Modelde bulunup hiçbir testte bulunmayan bir hata, bir umuttur.

Trade-off'lar ve maliyetler

Bir hata modeli yazmak zamana ve bilmediğinizi kaydetmenin rahatsızlığına mal olur. Bir servisi ve bir queue'su olan küçük bir ekibin bu listedeki her maddeye ihtiyacı yoktur. En çok acıtacak üç dört hatayla başlayın.

Modelin de bir sınırı vardır. Bu kitaptaki her savunma, bazı hataların kapsam dışı olduğunu varsayar: veritabanının dayanıklı olduğunu, broker'ın replication'ının yapılandırıldığı gibi çalıştığını. Bu varsayımlar makuldür ve her biri modelin yanlış olabileceği bir yerdir. Onları listelemek kabul edilebilir olup olmadıklarına karar vermenizi sağlar; görünmezken bunu yapamazsınız. Kendi kodunuzdaki bir bug ise farklıdır. Kapsam içindedir; tespit edilmesi ve kurtarılması gereken bir hata olarak, çünkü olacaktır ve modelin, bunu nasıl fark edeceğinizi ve yazdığı şeyi nasıl onaracağınızı söylemesi gerekir.

40 maddelik bir model okunmaz ve üç maddelik bir model vakaları kaçırır. Yeni bir ekip üyesinin tek oturuşta okuyabileceği bir liste hedeflerdim.

Her hata için azami güçte tasarım yapmak da hedef değildir. Bir broker failover'ından sıfır kayıpla sağ çıkmak latency'ye ve paraya mal olur ve çift bir hatadan sağ çıkmak daha fazlasına mal olur. Ara sıra iki kez gönderilen bir Notifications e-postası, asla duplicate olmaması gereken bir Ledger entry ile aynı savunmayı hak etmez. Modelin işi, riske göre harcama yapmanıza izin vermektir. Bir hatanın iş maliyetini adlandıramıyorsam, onu önlemek için teknik bir maliyet ödemekte tereddüt ederdim.

Model sistemi güvenilir yapmaz. Varsayımlarınızı denetlenebilir yapar.

Sık yapılan hatalar

- **Yalnızca çökmeleri modellemek.** Çökmeyi modellemek cazip, çünkü temiz ve tespit edilebilir. Maliyetli hatalar, timeout'lar ve yavaş consumer'lar gibi belirsiz olanlardır. Çözüm: her çağrı için, cevap gelmediğinde çağıranın ne yaptığını yazın.
- **Bir timeout'u hata saymak.** "Cevap yok"u "olmadı" diye okumak doğal geliyor. Çözüm: onu bilinmeyen olarak ele alın ve işlemi, tekrarlanması güvenli ya da kontrol edilebilir olacak şekilde tasarlayın.

- **Broker'ın acknowledgment'ına kanıt olarak güvenmek.** "Acknowledged" kelimesi kesin gibi duyulur. Çözüm: acknowledgment ayarınızın gerçekte ne garanti ettiğini bilin ve etmediği durumlar için reconcile edin.
- **Timestamp'e göre sıralamak.** `occurred_at` envelope'da hemen orada ve ona göre sıralamak tek satır. Çözüm: bir partition key içinde aggregate versiyonuna göre sıralayın ve timestamp'leri insanlar için tutun.
- **Deploy'ları yok saymak.** Release'ler hata gibi değil, rutin gibi hissettirir. Çözüm: her deploy'u planlı bir consumer çökmesi sayın ve mesajlar işlenirken shutdown'ı test edin.
- **Operatörleri dışarıda bırakmak.** Onarım araçları, ürün kodundan daha az review alır. Çözüm: redrive'ı, replay'i ve manuel düzenlemeleri, batch limitleri, pause switch'leri ve audit trail'lerle modele koyun.
- **Geri yüklenmiş bir veritabanını temiz bir başlangıç saymak.** Bir restore, incident'in sonu gibi hissettirir. Çözüm: her restore'dan sonra, tüketim sürmeden önce diğer servislerle reconcile edin ve geri alınmamış dış etkileri kontrol edin.
- **Değişmeyecek bir hatayı retry etmek.** Retry'ler titiz hissettirir. Çözüm: zamanlamaya bağlı hataları mesaja bağlı olanlardan ayırın ve ikinci türü erkenden dead-letter yoluna gönderin.
- **Modeli bir kez yazmak.** Bir tasarım aşamasında yazılan doküman bir anlık görüntüdür. Çözüm: bir owner atayın, incident'lerden sonra gözden geçirin ve her maddeye tarih koyun.

Temel çıkarımlar

- Bir timeout "beklemeyi bıraktım" demektir, "başarısız oldu" değil; bu yüzden bilinmeyen hangi yöne çözülmüşse çözülsün doğru kalan işlemler tasarlayın.
- Her birleşim yerinin kendi çökme pencereleri vardır ve hasar sessizdir. Bir dual write, event'siz siparişler ya da var olmayan siparişler hakkında event'ler bırakır ve commit ettikten sonra ama ack'lemeden önce çöken bir consumer her redelivery'de bir duplicate alır; bu da idempotent consumer'ları bir gereklilik yapar.

- Bir broker'ın acknowledgment'ı, sınırı replication ve flush ayarlarıyla belirlenen bir delildir; bu yüzden sınırı bilin ve dışında kalanı reconcile etmeyi planlayın.
- Yavaş consumer'lar ve deploy'lar, hiçbir bileşen bozuk olmadan duplicate'lere ve duraksamalara yol açar; bu yüzden yükü sınırlayın, lag'i ve mesaj yaşını ölçün ve consumer'ları zarıfçe durdurun.
- Her seferinde başarısız olan bir mesaj ya da her mesaj için başarısız olan bir dependency, şans eseri bir hatadan farklı bir tepki gerektirir; bu yüzden retry etmeden önce sınıflandırın ve dead-letter'ı toparlanmayacak olanlara saklayın.
- Duvar saati timestamp'leri makineler arasında uyuşmaz; bu yüzden event'leri bir partition key içinde aggregate versiyonuna göre sıralayın ve `occurred_at`'i insanlar için kullanın.
- Operatörler ve restore'lar sistemin parçasıdır: geri yüklenmiş bir store zamanda geriye gitmiştir, bu yüzden tüketimi sürdürmeden önce reconcile edin ve onarımlar bir sonraki incident'e dönüşmesin diye owner'ları, testleri ve review tarihlerini adlandıran yazılı bir hata modeli tutun.

Kaldırılacak ilk hata, producer'daki hatadır; orada bir durum değişikliği ve event'i kimse fark etmeden birbirinden ayrılabilir. Bölüm 4, o pencereyi kapatan transactional outbox ile Kısım II'yi başlatır.

KISIM II

Güvenilir Şekilde Yayınlamak

Bölüm 4 — Dual-Write Problemi ve Transactional Outbox

Parcelly'nin satıcı promosyonlarını devreye almasının üzerinden birkaç hafta geçtiğini düşünün. Bir destek mühendisi, tamamı şu olan bir ticket açıyor: "Alıcı, 81724 numaralı sipariş için ödeme adımını tamamladığını söylüyor ama satıcıya hiç kargo etiketi ulaşmamış. Sipariş 'verildi' görünüyor." Mühendis Orders veritabanına bakıyor ve satırı buluyor: durum `PLACED`, oluşturulma saati 14:02:11. Sonra Payments'a bakıyor, bu sipariş için hiçbir authorization yok. Inventory'de de rezervasyon yok. Parcelly'nin geri kalanı açısından bu sipariş yok.

14:02 civarındaki Orders loglarını arıyor ve bir deploy işareti buluyor. Servis, siparişi commit ettikten 200 milisaniye sonra, `OrderPlaced`'i yayımlamadan önce yeniden başlatılmış. Kod tam olarak yazıldığı şeyi yapmış: önce kaydet, sonra yayımla. Hiçbir hata fırlatılmamış, çünkü hatayı fırlatacak süreç çoktan gitmişti. Hızlı bir sorgunun, aynı dakika içinde aynı durumda 11 sipariş daha gösterdiğini varsayalım.

Ekip 11 siparişi elle düzeltiyor ve runbook'a bir not ekliyor. Kimse bundan memnun değil. Kod, bir reviewer'ın işaretleyeceği türden bir şekilde yanlış değil; yine de bir süreç yanlış anda öldüğünde iş kaybedebiliyor. Daha büyük bir makine bunu değiştirmez, daha dikkatli bir geliştirici de değiştirmez. Bu bölümün sorusu şu: bir servis, veritabanı ile broker'ı kapsayan bir transaction olmadan, kendi veritabanını değiştirip bunu sistemin geri kalanına birbirinden ayrılamayacak biçimde bildirebilir mi?

Problem ve neden zor olduğu

Bölüm 1 dual write'ı tanımlamıştı: ortak bir transaction olmadan güncellenen iki sistem. Orders, ilişkisel veritabanına bir satır yazmalı ve broker'a bir mesaj yayımlamalı. Veritabanı, ilkinin atomikliğini garanti

eder. Broker, bazı konfigürasyonlarda, ikincisinin durability'sini garanti eder. İkisinin de gerçekleşeceğini ya da ikisinin de gerçekleşmeyeceğini garanti eden hiçbir şey yok.

Bölüm 3, producer'ın crash pencerelerini bir diyagramda çizmişti. Burada onları bir tablo olarak ele almak istiyorum, çünkü iki yazmanın sırası çoğu ekibin fark etmeden verdiği bir tasarım kararıdır ve her seçimin arkasında bekleyen bir hata vardır.

Her sıralama bozulur

İki bariz sırayı alın ve olası her hata noktası için, sonrasında dünyanın nasıl görüldüğünü sorun. Bu, veritabanı ile broker arasındaki birleşim yerinin failure semantics'ini yazıya dökmekle aynı alıştırmaadır: ne başarısız olabilir, bunu nasıl gözlemlersiniz ve hangi garanti ayakta kalır. Aşağıdaki tablo, producer için bu alıştırmanın ta kendisidir.

Sıra	Hata noktası	Veritabanının durumu	Broker'ın durumu	Hasar
Commit, sonra publish	Commit'ten sonra, publish'ten önce crash	Sipariş kaydedildi	Event yok	Kayıp event. Downstream siparişi hiç öğrenmez
Commit, sonra publish	Publish başarısız olur ya da timeout'a düşer, süreç hayatta kalır	Sipariş kaydedildi	Bilinmiyor	Çağırın retry etmeli ya da vazgeçmeli. Vazgeçmek event'i kaybettirir
Commit, sonra publish	Broker kabul eder, ack kaybolur	Sipariş kaydedildi	Event var	Retry bir duplicate yayımlar
Publish, sonra commit	Publish'ten sonra, commit'ten önce crash	Sipariş yok	Event var	Hayalet event. Downstream hiçbir zaman gerçek olmamış bir olguya göre hareket eder
Publish, sonra commit	Commit başarısız olur (constraint, deadlock, timeout)	Sipariş yok	Event var	Aynı hayalet event; bu kez producer'da görünür bir hata da var

Sıra	Hata noktası	Veritabanının durumu	Broker'ın durumu	Hasar
Transaction içinde publish, en son commit	Publish'ten sonra commit başarısız olur	Sipariş yok	Event var	Transaction bloğu koda yanlış bir güvenlik hissi verir, broker ise bu bloğun parçası değildir

Son satır bir yoruma değer, çünkü en güvenli görünen o. Birçok framework, bir veritabanı transaction bloğunun içinden publish yapmanıza izin verir. Sanki publish transaction'ın parçasıymış gibi okunur. Değildir. Broker çağrısı hemen gerçekleşir ve geri alınamaz; commit ise sonradan yine gerçekleşir. Bu, daha iyi girintilenmiş publish-then-commit sıralamasıdır.

Bunun ayna görüntüsü, birçok framework'ün sunduğu "after commit" callback'idir; transaction commit olduktan sonra çalışan bir hook. Ekipler çoğu zaman bunun güvenli olduğuna inanır, çünkü publish yalnızca commit edilmiş veri için gerçekleşir. Oysa bu, tablonun ilk satırıdır: önce commit, sonra publish, niyet de süreç belleğinde tutuluyor. Commit ile hook arasındaki bir crash event'i kaybettirir ve event'in hiç borç olduğunu kaydeden bir şey yoktur.

Tablonun yakalamadığı dördüncü bir biçim de var: yavaş bir broker. Publish transaction'ın içinde yapılıyorsa, veritabanı broker'ı beklerken lock'ları ve bir bağlantıyı elinde tutar. Bir dakika boyunca yavaş olan bir broker, bir dakika boyunca yavaş olan bir veritabanına dönüşür. Birbirinden bağımsız olması gereken iki sistemin availability'sini birbirine bağlamış olursunuz.

Retry neden yardımcı olmaz

Akla yatkın ilk onarım, publish etrafına bir retry döngüsü koymaktır. Retry'lar, sürecin hayatta kalıp yeniden deneyebildiği tablonun ikinci satırında işe yarar. Sürecin gittiği birinci satırda hiçbir işe yaramazlar ve üçüncü satırdaki duplicate'i yaratırlar. Bölüm 6, retry'ları düzgünce ele alıyor. Şimdilik şuna dikkat edin: retry döngüsü sürecin belleğinde yaşar, bizim ilgilendiğimiz hata ise sürecin ortadan kaybolmasıdır.

İkinci cazip onarım, veritabanını broker ile karşılaştırıp eksik olanı yeniden yayımlayan periyodik bir job'dır. Bazı ekipler bunu kuruyor; bence makul bir emniyet ağı ve Bölüm 13 bunu reconciliation olarak yeniden ele alıyor. Ana mekanizma olarak ise iki sorunu var. Broker'a nadiren iş anahtarıyla sorgu atılabilir, dolayısıyla "eksik olan" pahalı bir tahmine dönüşür. Üstelik job'ın kendisi de aynı dual write'ı yapar; yani aynı şekillerde başarısız olabilir, yalnızca daha seyrek ve daha fazla kafa karışıklığıyla.

Neden distributed transaction değil

Sorun, ortak transaction'ı olmayan iki sistemse, klasik cevap onlara bir tane vermektir. Veritabanları ve transaction manager'lar için XA olarak standartlaştırılmış **two-phase commit (2PC)**, birkaç resource manager'ı koordine eder. İlk fazda her katılımcı prepare eder; yani commit edebileceğine söz verir ve lock'larını tutar. İkinci fazda bir coordinator her katılımcıya commit etmesini, ya da biri prepare edememişse rollback yapmasını söyler.

Kâğıt üzerinde bu pencereyi kapatır. Pratikte bir broker ile bir servis veritabanı için nadiren cevaptır; nedenleri çoğunlukla operasyoneldir.

- **Destek düzensizdir.** Popüler broker'ların çoğu bir transaction-manager arayüzünü hiç uygulamaz. Uyguladığında da garantiler ve hata yönetimi veritabanınıninkinden farklıdır ve iki kural setini birden devralırsınız.
- **Coordinator karar verirken katılımcılar lock tutar.** Coordinator iki faz arasında crash olursa, prepare edilmiş transaction'lar in-doubt durumunda kalır. Satırlar, biri ya da bir şey onları çözene kadar kilitli kalır. In-doubt penceresi, tam da kimsenin prova etmediği türden nadir ve uzun bir hatadır.
- **Coordinator, çalıştırmanız, güvenceye almanız ve kurtarmanız gereken bir bileşen olur.** Kendi durable log'larına ve doğru olması gereken bir recovery prosedürüne ihtiyaç duyar.
- **Availability çarpılır.** Veritabanının, broker'ın ve coordinator'ın birlikte sağlam olmasını gerektiren bir transaction, ancak üçü de sağlamken kullanılabilir. Broker'ı cazip kılan bağımsızlık gider.
- **Throughput ve latency zarar görür.** İki round trip, her katılımcıda zorunlu yazmalar ve ağ çağrıları boyunca tutulan lock'lar birikir.

- **Dış dünyaya uzanmaz.** Bir e-posta sağlayıcısı ya da bir ödeme sağlayıcısı sizin transaction'ınıza katılmaz. Broker için 2PC'yi benimseyen bir ekip bile her dış çağrı için başka bir cevaba ihtiyaç duyar.

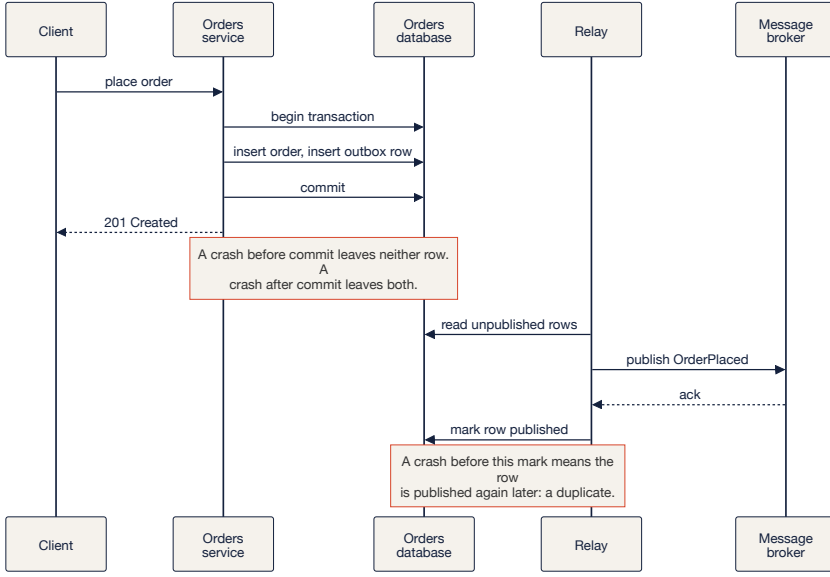
2PC'yi her durumda dışlamam. İki kaynağı da kontrol eden, olgun bir transaction manager çalıştıran ve in-doubt transaction'ları çözecek operasyonel becerilere sahip bir ekip bunu çalıştırabilir. Her biri kendi veritabanına sahip servislerden bir marketplace kuran ve XA arayüzü sunmayan bir broker kullanan çoğu ekip için maliyet yüksek, fayda dardır. İstedikleri şey bir distributed transaction'dan daha küçüktür: state değişikliğinin ve publish etme niyetinin birlikte commit olmasını isterler. Bu yerel bir problemdir ve yerel transaction'lar iyi anlaşılmış bir konudur.

Transactional outbox

Fikir, dual write'ı tek bir yazmaya ve sonradan yapılan, tekrarlanabilir bir taşımaya dönüştürmektir. Servis, request içinde broker'a publish etmek yerine, event'i tarif eden bir satırı kendi veritabanındaki bir tabloya, state değişikliğiyle aynı yerel transaction içinde yazar. Bu tablo **transactional outbox**'tır. Bundan sonra kitap ona outbox diyor.

Relay adlı ayrı bir süreç, outbox satırlarını okur ve broker'a publish eder. Bölüm 5 relay'i anlatıyor. Bu bölümde onu, her satırı en az bir kez eninde sonunda teslim eden bir fonksiyon olarak ele alıyor ve bunun bize ne kazandırdığına bakıyoruz.

Mantık kısa. Sipariş ve outbox satırı ya birlikte commit olur ya da hiçbiri olmaz, çünkü ikisi de tek bir veritabanında ve tek bir transaction'da yaşar. Satır var olduktan sonra publish etme niyeti durable'dır. Commit'ten sonraki bir crash artık event'i kaybettirmez, çünkü servis ya da yerine geçen servis yeniden başladığında satır hâlâ oradadır. Commit'ten önceki bir crash ne sipariş ne satır bırakır, dolayısıyla hayalet event de olmaz.



Diyagramdaki iki nota bakın. Birincisi kapattığımız pencereyi gösteriyor. İkincisi tuttuğumuz pencereyi gösteriyor: relay publish edip, bunu kaydetmeden önce crash olabilir. Outbox duplicate'leri ortadan kaldırmaz; hatayı sınırlı olduğu ve ele alındığı bir yere taşır. Buna aşağıda döneceğim.

Tablo

Outbox, kitabın geri kalanında kullanılan şekli izler. Her envelope alanını taşır; böylece relay, üreten servise hiçbir şey sormadan event'i kurabilir. Bu bölüm envelope'un ötesine dört kolon ekliyor; her birinin bir gerekçesi var.

```

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

```

Bu tercihlerin birkaçının savunulması gerekir.

- **id ve event_id farklı şeylerdir.** event_id, consumer'ların idempotency key olarak kullanacağı bir UUID'dir ve event iki kez yayımlandığında değişmemelidir. id ise relay'in satırları bulmak için kullanabileceği, veritabanının atadığı bir sequence number'dır. Commit sırası hakkında bir şey söylemez ve bir sonraki kısım bunun nedenini gösteriyor.
- **aggregate_version ordering'in doğruluk kaynağıdır.** Aggregate'in version'ıdır ve satırı yazan transaction içinde artırılır. Parcellly siparişlerinde bu order_version'dır. Aggregate başına sıra gerektiren consumer'lar bu sayıyı karşılaştırır; id'yi ya da geliş sırasını değil. Bölüm 9, bir consumer'ın bunu nasıl kullandığını gösteriyor. Standart envelope'ta bunun için bir alan yoktur; bu yüzden relay event'i kurarken kolonu payload'a order_version olarak kopyalar ve consumer'lar onu oradan okur.
- **aggregate_type ve partition_key ayırdır.** partition_key, broker'ın ordering için kullandığı şeydir ve order-81724 gibi aggregate id'sidir. aggregate_type, relay'e ve tabloyu okuyan insana

anahtarın ne tür bir şeye işaret ettiğini söyler; ayrıca satırları farklı topic'lere yönlendirmenin kullanışlı bir yoludur.

- **created_at, occurred_at'ten ayırdır.** Birincisi satırın yazıldığı zamandır, ikincisi olgunun gerçekleştiği zaman. Genellikle ikisi milisaniyeler içindedir. Bir servis geç öğrendiği bir olguyu kaydettiğinde, örneğin event'ten sonra gelen bir kargo taraması, ayırırlar. Relay-lag metric'i **created_at**'i kullanmalıdır, çünkü lag iş alanının değil tablonun bir özelliğidir. Saat tek bir saat olsun diye, onu uygulama kodunda **now()** çağırmak yerine bir kolon default'u ile veritabanına doldurtun. Yine de bu insert zamanıdır, commit zamanı değil: uzun süren bir transaction, satırı relay'in gördüğünden daha eski gösterir ve lag'in bu kısmı relay'in kusuru değildir.
- **headers transport kaygılarını tutar.** Trace context ve yönlendirme ipuçları buraya girer; böylece payload iş olgusu olarak kalır. Bir alanın veri mi yoksa transport ipucu mu olduğuna karar veremeyen bir ekip, onu payload'a eklemekten önce Bölüm 11'i okumalı.
- **published_at nullable'dır ve null anlamlıdır.** **published_at**'i olmayan bir satır, relay'in broker'a hâlâ borçlu olduğu bir satırdır. Bazı ekipler bunun yerine **PENDING**, **PUBLISHED** ve **FAILED** gibi değerlere sahip bir **status** kolonu kullanır. İki tasarım aynı bilgiyi taşır. Ben timestamp'i tercih ediyorum, çünkü ne zaman olduğunu da söyler; relay'in ne kadar geç kaldığını sorarken bu işe yarar. Ayrıca yanlış yapılabilecek şey daha azdır. Kalıcı olarak başarısız olan satırları park etmenin bir yoluna ihtiyacınız olursa, **failed_at** kolonunu o zaman ekleyin, daha önce değil.

O an, duyduğunuzdan daha erken gelir. Broker'ın hiçbir zaman kabul etmeyeceği bir satır, örneğin aşırı büyük bir payload ya da serializer'ın reddettiği bir payload, biri onu onarana kadar kendi anahtarını ve en basit relay'de tüm tabloyu bloke eder. Bu ilk olduğunda bir **failed_at** kolonuna (ya da status kolonlarını seçtiyseniz bir **FAILED** status'una) ve park edilmiş satırlar için bir alert'e ihtiyacınız olur. Bölüm 5, relay'in bunu nasıl kullandığını gösteriyor.

Relay'in ihtiyaç duyduğu tek index, yayımlanmamış satırlar üzerinde, insert sırasıyla tutulan indextir. Onsuz relay, sonsuza kadar büyüyen bir tabloyu tarar. Temizliği Bölüm 5 ele alıyor.

Aynı transaction içinde yazmak

Üreten kod kısadır ve kısalığı asıl meseledir. Aşağıdaki fonksiyon, Orders'ın request yolundaki değişikliğin tamamıdır.

```
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
```

Üç ayrıntı görüldüğünden daha önemli.

Birincisi, fonksiyondaki hiçbir şey broker ile konuşmaz. Bir broker kesintisi sipariş vermeyi başarısız kılmaz ve yanıt vermesi bir dakika süren bir broker, bir veritabanı transaction'ını açık tutmaz. Bu, availability açısından gerçek bir kazanç: Orders, broker kapalıyken de sipariş kabul edebilir ve outbox, relay onu boşaltabilene kadar büyür.

İkincisi, `event_id` commit'ten önce oluşturulur ve satırda saklanır. Bu event'in sonraki her yeniden yayımı aynı id'yi taşır. Consumer'ların dayandığı özellik budur ve id'yi relay'de, publish anında üretmenin neden bir hata olacağı da budur.

Üçüncüsü, payload transaction'ın içinde, transaction'ın yazdığı state'ten kurulur. Payload sonradan siparişi yeniden okuyarak kurulsaydı, eşzamanlı bir güncelleme onu değiştirebilirdi ve event, o anda var olmayan bir

olguyu tarif ederdi. Event, olanın bir anlık görüntüsüdür ve bu görüntü, değişikliğin yapıldığı yerde alınır.

Pseudocode, sipariş insert'ünün ve outbox insert'ünün aynı veritabanı bağlantısını ve aynı transaction'ı kullandığını varsayar. Bu, düşündüğünüzden daha sık kazara bozulur. Outbox için ikinci bir bağlantı açan ya da outbox insert'ünü auto-commit eden bir data-access katmanı, dual write'ı sessizce geri getirir; bu kez iki tablo arasında. Bunun testi sizin test paketinizde yer almalı: sipariş insert'ünden sonra bir hata zorlayın ve hiçbir outbox satırının bulunmadığını doğrulayın.

Commit sırası, id sırası ve aggregate lock'u

Yeni bir siparişin rakibi yoktur; bu yüzden `placeOrder` bir lock gerektirmez. Mevcut bir siparişteki her sonraki değişiklik gerektirir ve lock'un önce alınması gerekir. `OrderCancelled` emit eden `cancelOrder` aşağıda.

```
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
```

Version'ı güvenilir kılan lock'tur ve tek bir aggregate için id sırasına anlam kazandıran da yalnızca odur. Lock olmadan, aynı sipariş üzerindeki iki transaction'ı düşünün. B transaction'ı outbox satırını insert eder ve id 101'i alır, sonra A'nın tuttuğu sipariş satırında bloke olur. A, id 102'yi insert eder ve commit olur; B sonradan commit olur. Commit sırası A, sonra B'dir; id sırası ise B, sonra A. İki satırı tek bir poll'da gören bir relay onları ters sırayla publish eder. Lock önce alındığında B, A commit olana kadar insert edemez; dolayısıyla tek bir aggregate'in id'leri onun commit sırasını izler.

Öyle bile olsa consumer'ları `id`'ye bağımlı kılmazdım. Farklı aggregate'lerin satırları iç içe geçer, paralel bir relay batch'ler arasında reordering yapar ve Orders veritabanının dışında kimse `id`'yi göremez. `aggregate_version`, event ile birlikte yolculuk eder; payload'da `order_version` olarak ve bu siparişe ilgili olguların hangi sırayla gerçekleştiğini söyler. Bölüm 5, bir relay'in neyi koruyup neyi koruyamayacağına geri dönüyor; Bölüm 9 ise consumer tarafını gösteriyor.

Relay'in borçlu olduğu şey

Relay'in sözleşmesi bilerek küçük tutulmuştur. Yayımlanmamış satırları seçer, her birini broker'a publish eder ve broker onayladıktan sonra yayımlandı olarak işaretler. En basit sürüm aşağıda; Bölüm 5, onun yok saydığı üretim kaygılarını ele alıyor: batching, locking, ordering ve failover gibi.

```
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())
```

Döngünün içindeki yorum, garantinin tamamıdır. Satır yalnızca broker onayladıktan sonra işaretlenir; çağrının ack'i beklemesinin ve bir hatanın batch'i durdurmasının nedeni de budur. Publish ile işaretleme arasındaki bir crash, satırı yayımlanmamış bırakır. Bir sonraki turda relay onu yeniden publish eder. Alternatif, önce işaretleyip sonra publish etmek, hatayı duplicate yerine kayıp event hâline getirirdi. Outbox daha güvenli hatayı bilerek seçer.

Outbox neyi garanti eder

Outbox size üç şey verir ve bir şeyi elinizden alır.

- **State ile niyet arasında atomiklik.** State değişikliği ve publish etme kararı birlikte commit olur. Outbox satırı olmayan commit edilmiş bir sipariş yoktur, commit edilmiş siparişi olmayan bir outbox satırı da yoktur.
- **Niyetin durability'si.** Transaction commit olduktan sonra event, veritabanı kadar durable'dır. Relay kapalı olabilir, broker kapalı olabilir, producer değiştirilebilir ve satır bekler.
- **At-least-once publication.** Relay ve broker eninde sonunda toparlanırsa, relay satırları bir id cursor'ıyla değil state'e göre bulursa, temizlik yalnızca yayımlanmış satırları silerse ve broker'ın hiçbir zaman kabul etmeyeceği bir satır sonsuza kadar bloke etmeye bırakılmayıp park edilir ve ona alert kurulursa (bunların hepsi Bölüm 5'te), commit edilmiş her satır eninde sonunda yayımlanır. Bir satır birden fazla kez yayımlanabilir.
- **Garanti edilmeyen: exactly-once publication.** Relay, uygulamanın yapamadığı aynı nedenle, broker'a publish etmeyi ve veritabanında bir satırı işaretlemeyi atomik olarak yapamaz. Pencere request yolundan relay'e taşındı; orada daha küçük, gözlemlenebilir ve onarılabilir.

Bu özet konusunda dikkatli olmak istiyorum. Outbox, event'lerin asla kaybolmadığı iddiası değildir. Event üretmesi gereken her state değişikliği için uygulama outbox üzerinden yazdığı sürece, dual write'ın producer tarafındaki kayıp penceresinin kapandığı iddiasıdır. Tabloyu elle güncelleyen ya da outbox'ı atlayan bir yoldan güncelleyen bir geliştirici pencereyi yeniden açar. Bölüm 3 bunu bir insan hatası olarak listelemiştir ve outbox'ı olan bir ekibin hâlâ event kaybetmesinin en yaygın yolu budur.

Duplicate, consumer'ın problemi olur

Publication at-least-once olduğu için, outbox ile yayımlanan bir event'in her consumer'ı duplicate'lere dayanıklı olmalıdır. Bu mühendislikle ortadan kaldırılacak bir kusur değil, garantinin bedelidir ve bedeli kimin ödediği konusunda net olmakta fayda var.

Producer ekibi hafif kısmı yapar. Bir tablo ekler, bir satır yazar ve bir relay çalıştırır. Consumer ekipleri daha zor kısmı yapar, çünkü daha önce gördükleri bir `event_id`'yi tanımalı ve etkisini tekrarlamaktan kaçınmalıdırlar. Payments iki kez authorize etmemeli. Notifications aynı e-postayı iki kez göndermemeli. Ledger aynı kaydı iki kez işlememeli.

Uyarı: Idempotent consumer'lar olmadan bir outbox benimsemek, kayıp event problemini duplicate etki problemiyle takas etmektir. Çifte tahsilat gibi bazı event'ler için ikinci problem daha pahalıdır. İkisini birlikte yayına alın ve bir consumer'ın tekrarlanan işi nasıl tanıdığı için Bölüm 7'ye bakın.

Bunu producer ekiplerine de açık sözlerle söyledim: outbox, consumer'larınızla yaptığınız bir sözleşmedir. Aynı olgunun her yeniden yayımında `event_id` sabit kalmalıdır ve sözleşme örtük bırakılmamalı, event dokümantasyonuna yazılmalıdır.

Kısa bir hata turu

Tasarımı Bölüm 3'teki pencerelere karşı denetlemek için Orders'ı crash'ler boyunca yeniden yürütün.

1. Süreç, transaction commit olmadan önce ölür. Sipariş yok, outbox satırı yok, event yok. Client bir hata ya da timeout görür ve request'i retry edebilir; bu, API katmanına ait ayrı bir problemdir ve Bölüm 7'deki idempotency tartışmasında ele alınır.
2. Süreç, commit'ten hemen sonra ölür. Sipariş ve satır ikisi de var. Relay satırı bir sonraki turunda publish eder. Client response'unu almamış olabilir, ama sistem tutarlıdır.
3. Relay, publish ettikten sonra ve işaretlemekten önce ölür. Event broker'dadır ve satır yayımlanmamış durumdadır. Bir sonraki tur onu yeniden publish eder. Consumer'lar aynı `event_id` ile bir duplicate görür.
4. Broker kullanılamıyor. Relay publish edemez, satırları yayımlanmamış bırakır ve daha sonra tekrar dener. Orders request kabul etmeye devam eder. Lag büyür ve bir alert de büyümeli.
5. Servis, commit sırasında bağlantısını kaybeder ve sonucu bilmez. Ya sipariş ve satırı vardır ya da ikisi de yoktur ve servis hangisi olduğunu söyleyemez. Tahmin yürütmemelidir. Güvenli cevap, client'ın retry'nin API düzeyinde bir idempotency key taşımasıdır; böylece veritabanı hangi durumda olursa olsun retry tek bir siparişte ve tek bir event'te birleşir (Bölüm 7).

Her durum ya tutarlılıkla ya da bir duplicate ile biter; hiçbir hayalet event ya da sessiz bir kayıpla bitmez. Outbox'ın iddia ettiği şeyin tamamı budur ve onu haklı çıkarmaya yeter.

Bir duplicate outbox'ın upstream'inde durur ve hiçbir downstream savunması onu görmez. Bir client, kaybolan bir response'tan sonra retry ettiğinde ve API'nin kendi anahtarı olmadığına, Orders ikinci bir sipariş ve yeni bir `event_id` ile ikinci bir `OrderPlaced` oluşturur. O zaman her consumer'ın inbox'ı iki farklı event görür, çünkü gerçekten farklıdır. Outbox işini iki kez yapmıştır. Çözüm API'dedir; Bölüm 7'deki Idempotency-Key yaklaşımıyla.

Alternatifler ve komşular

Dual write'ı durdurmanın tek yolu outbox değil. İki tasarım daha adil bir karşılaştırmayı hak ediyor, çünkü bir ekip makul bir şekilde bunlardan birini tercih edebilir.

Kendini dinlemek

Listen-to-yourself örüntüsünde servis önce veritabanına yazmaz. Event'i broker'a publish eder, sonra kendi event'ini consume eder ve state değişikliğini handler'da uygular. Broker tek yazma olur ve veritabanı, servisin duyduklarının bir projection'ı olur.

Bu, tek bir yazma olduğu için dual write'ı ortadan kaldırır. Servisin şeklini değiştirir. Client'ın request'i, state var olmadan döner; dolayısıyla sistem beklemenin bir yolunu sunmuyorsa çağırana kendi yazdığını okuyamaz. Doğrulama publish'ten önce yapılmalıdır, çünkü bir event yayımlandıktan sonra bir olgudur ve geçersiz olan bir compensation olmadan sonradan reddedilemez. Ayrıca event'i uygulayan handler'ın kendisi de idempotent olmalıdır; bu da aynı consumer yükünü geri getirir.

Bunu zaten event'lerle düşünen ve kendi okumalarında eventual consistency'ye dayanabilen servisler için düşünürdüm. Alıcının tarayıcısının, düğmeye tıkladıktan hemen sonra siparişi görmeyi beklediği Orders için düşünmezdim. Kendi yazdığını okuma boşluğu, ekibin sonunda geçici çözümlerle çözeceği bir ürün problemidir.

Event sourcing

Event sourcing'de event'ler doğruluk kaynağıdır. Servis, aggregate başına event dizisini saklar ve mevcut state bunlardan türetilir. Event store'a yazmak state değişikliğinin kendisidir; dolayısıyla ayrı bir event ile tutarlı tutulacak bir şey yoktur. Broker'a publish etme o zaman event store'dan okur; bu, başka bir adla bir relay'dir.

Event sourcing, veritabanının ne olduğunu değiştirerek dual write'ı çözer. Ayrıca tam bir geçmiş ve projection'ları yeniden kurma yeteneği verir. Bedeli farklı bir programlama modeli, hiçbir zaman rahatça yeniden yazamayacağınız saklı event'lerin schema evolution'ı ve tüm ekip için bir öğrenme eğrisidir. Önemli bir mimari taahhüttür ve yalnızca bir outbox'tan kaçınmak için benimsemezdim. Zaten kullanıyorsanız, yine de bir relay'e ihtiyacınız var ve bu bölümde at-least-once publication hakkında söylenen her şey geçerli.

Seçenekleri karşılaştırmak

Seçenek	Neyi ortadan kaldırır	Neye mal olur	En iyi uyduğu durum
Commit, sonra publish	Hiçbir şeyi	Crash'te sessiz kayıp	Event kaybı kabul edilebilir, örneğin best-effort metric'ler
2PC / XA	Pencereyi, teorik olarak	Lock'lar, coordinator, düzensiz destek	İki kaynağı da kontrol ediyor ve recovery'yi kendiniz yürütüyorsunuz
Transactional outbox	Producer tarafı penceresini	Ek bir tablo, bir relay, duplicate'ler	State ilişkisel bir veritabanında yaşıyor ve event'ler kaybolmamalı
Listen to yourself	İkinci yazmayı	Kendi okumalarında eventual consistency, erken doğrulama	Servis zaten event-first
Event sourcing	Ayrı state yazmasını	Farklı bir state modeli	Geçmişin kendisi bir gereksinim

State'ini zaten ilişkisel bir veritabanında tutan ve bunu başkalarına bildirmesi gereken bir servis için outbox en küçük değişikliktir. Kitabın çoğunun onu varsaymasının nedeni bu.

Trade-off'lar ve maliyetler

Outbox'ı tarif etmek ucuz, çalıştırmak bedava değil. Bunlar, bir ekip taahhüt vermeden önce önüne koyacağım maliyetler.

Daha fazla hareketli parça. Bir tablo, bir relay süreci ve izlenecek bir lag var. Bir şey bozulduğunda bakılacak yeni bir yer var. Relay, birinin sahiplendiği bir bileşendir ve deploy, alert ve runbook'a ihtiyaç duyar. Bölüm 5 ayrıntıyı veriyor.

Ek veritabanı yükü. Her event, iş yazmasıyla aynı transaction içinde bir insert'tür ve her publish bir okuma ile bir güncellemedir. Transaction'lar biraz uzar ve yazma hacmi artar. Çoğu servis için artış, işin geri kalanına kıyasla küçüktür. Çok yüksek event hacimleri yazan bir servis, benim "küçük" dememe güvenmek yerine bunu kendi donanımında ölçmeli.

Eklenen latency. Event, transaction commit olduktan ve relay satırı fark ettikten sonra broker'a ulaşır. Polling ile gecikme, polling aralığına bağlıdır. Log tabanlı bir yaklaşımla daha kısa olabilir ve Bölüm 5 ikisini karşılaştırıyor. Her iki durumda da outbox, event'in request'ten birkaç milisaniye içinde broker'a ulaşması gereken akışlar için doğru seçim değildir ve seçmeden önce latency bütçenizi bilmelisiniz.

Tasarım gereği duplicate'ler. Yukarıda ele alındı. Bir consumer idempotent yapılamıyorsa, outbox onun için hiçbir şeyi iyileştirmez ve ekibin, o consumer'ın bu hâliyle var olup olmaması gerektiğine karar vermesi gerekir.

Ordering yalnızca kısmen korunur. Relay satırları `id`'ye göre okur ve her transaction, outbox satırını insert etmeden önce aggregate satırını lock'lamadıkça `id` sırası commit sırası değildir. Bu lock, tek bir aggregate'in yazıcılarını serileştirir; bu da sıcak aggregate'lerde başlı başına bir maliyettir. Paralel relay'ler sırayı daha da kötüleştirir ve lock aggregate'ler arasında yardımcı olmaz. `aggregate_version` kolonu consumer'lara denetleyecekleri bir şey verir ve her yazmada bir version artışına mal olur. Outbox size sırayı zorlayabileceğiniz bir yer verir, bedava sıra vermez. Bölüm 5 ve Bölüm 9 buna geri dönüyor.

Büyüme ve temizlik. Yayınlanmış satırlar birikir. Silme ya da arşivleme olmadan tablo, relay'in sorgusu yavaşlayana kadar büyür. Retention bir karardır ve reconciliation ile etkileşir, çünkü eski satırlar aynı zamanda

nelerin yayımlandığının kanıtınızdır.

Yalnızca transaction destekleyen bir veritabanındaki state için çalışır.

State, çok satırlı transaction'lı olmayan bir store'da yaşıyorsa, store'a özgü bir change stream ya da event-sourced bir tasarım gibi başka bir yaklaşıma ihtiyacınız var. Atomikliği yalnızca tek bir doküman içinde sunan bir store'un kendine özgü bir varyantı vardır: bekleyen event'leri aggregate dokümanının içinde tutun ve relay'in onları oradan boşaltmasına izin verin.

Ne zaman kullanılmaz

Şu durumlarda outbox eklemeyiz:

- event best-effort ise ve kaybı kabul edilebilirse, örneğin bir metric ya da düşük değerli bir analitik ipucu
- servisin kendine ait durable state'i yoksa; yani event'in yanında commit edilecek bir şey yoksa
- sistem, state'e de onun consumer'larına da tek bir sürecin sahip olduğu kadar küçükse ve doğrudan bir çağrı daha basitse
- ekip consumer'ları henüz idempotent yapamıyorsa; bu durumda önce onu düzeltirdim, çünkü outbox bu açığın riskini yükseltir

Not: "Outbox kullanıyoruz", tek bir servisin producer yolu hakkında bir ifadedir. Consumer'ların güvenli olup olmadığı, ordering'in tutup tutmadığı ya da event'lerin iyi tasarlanıp tasarlanmadığı hakkında hiçbir şey söylemez. Onu bir kümedeki tek bir örüntü olarak görün, güvenilirlik mührü olarak değil.

Sık yapılan hatalar

- **Outbox satırını ayrı bir transaction'da yazmak.** Bir framework hook'u ya da bir event bus soyutlaması bağlantıyı gizlediği için cazip gelir. O zaman iki yazma, aynı veritabanının iki tablosu arasında bir dual write'a dönüşür ve pencere geri gelir. Tek bir bağlantı ve tek bir transaction kullanın ve rollback'i test edin.
- **"After commit" hook'undan publish etmek.** Yalnızca commit edilmiş veri için tetiklendiği ve tablo gerektirmediği için cazip gelir. Niyetin

bellekte tutulduğu commit-then-publish'tir; dolayısıyla ikisi arasındaki bir crash event'i kaybettirir. Satırı transaction içinde yazın ve publish'i relay'e bırakın.

- **Outbox satırını, aggregate'i lock'lamadan önce insert etmek.** Kod, tek yazıcılı her testte çalışır. Eşzamanlılık altında, tek bir sipariş için id sırası ile commit sırası ayrışır ve relay onun event'lerini ters sırayla publish edebilir. Önce aggregate satırını lock'layın, version'ını artırın, sonra insert edin.
- **Request yolundan publish edip outbox'ı yedek olarak tutmak.** Kemer ve pantolon askısı gibi hissettirir. İki publisher, farklı zamanlamalı duplicate'ler ve hakkında akıl yürütemeyeceğiniz bir ordering üretir. Tek publisher relay olsun.
- **event_id'yi relay'de üretmek.** Zararsız görünür, çünkü mesajı kuran relay'dir. O zaman yeniden yayımlanan event yeni bir id alır ve consumer'lar onu duplicate olarak tanıyamaz. Id'yi satır yazıldığında üretin.
- **Satırı, broker onaylamadan önce yayımlandı olarak işaretlemek.** Duplicate'leri azaltır ve daha derli toplu hissettirir. Duplicate'i kayıp bir event'e çevirir ki bu daha kötü bir hatadır. Onaydan sonra işaretleyin ve duplicate'i kabul edin.
- **Payload'ı commit'ten sonra kurmak.** Event'i doldurmak için aggregate'i yeniden okumak, üreten kodu basit tutuyor gibi görünür. Arada başka bir transaction veriyi değiştirebilir ve event artık olguyla örtüşmez. Payload'ı transaction'ın yazdığından kurun.
- **Bazı kod yollarının outbox'ı atlmasına izin vermek.** Bir migration script'i, bir admin aracı ya da elle çalıştırılan bir SQL düzeltmesi state'i değiştirir ve hiçbir şey emit etmez. Önemli olan her state değişikliğini aynı fonksiyondan geçirin ve elle yapılan onarımları, üretmeleri gereken event'ler açısından gözden geçirin.
- **Temizliği unutmak.** Tablo aylarca çalışır, sonra relay'in sorgusu en kötü anda yavaşlar. Retention'a, tabloyu oluştururken karar verin.
- **Outbox'ı güvenilirlik hikâyesinin sonu saymak.** Tek bir pencereyi kapatır. Consumer'ların hâlâ idempotency'ye, retry'lara ve hiç başarılı olmayan mesajlar için bir yere ihtiyacı var.

Temel çıkarımlar

- Dual write, daha iyi bir sıra seçilerek güvenli hâle getirilemez: commit-then-publish event kaybeder, publish-then-commit event uydurur ve transaction bloğu içindeki publish hâlâ ikincisidir.
- Two-phase commit ilke olarak pencereyi kapatabilir, ancak bir servis veritabanı ve bir broker için genellikle lock'lar, coordinator operasyonları ve availability açısından problemin değerinden fazlasına mal olur.
- Transactional outbox, event'i state değişikliğiyle aynı transaction'da bir satır olarak yazarak dual write'ı tek bir yerel transaction ile tekrarlanabilir bir taşımaya dönüştürür.
- Outbox satırı tam event envelope'unu, yazma anında oluşturulan sabit bir `event_id`'yi (böylece her yeniden yayım tanınabilir) ve aggregate'in lock'u altında artırılan bir `aggregate_version`'ı taşır, çünkü `id` sırası commit sırası değildir.
- Outbox, state ile niyetin atomikliğini, durable niyeti ve at-least-once publication'ı garanti eder; bir mesajın tam olarak bir kez yayımlanacağını garanti etmez.
- Duplicate'ler garantinin bedelidir; dolayısıyla idempotent consumer'ları olmayan bir outbox, bir hatayı başka bir hatayla takas eder.
- Listen-to-yourself ve event sourcing de dual write'ı ortadan kaldırır, ancak servisin state'ini nasıl okuduğunu ve sakladığını değiştirirler; bu yüzden onları salt bir relay'den kaçınmak için değil, verdikleri şey için seçerdim.

Publish etmek, taşımanın yalnızca yarısı: bir şeyin bu satırları okuyup broker'a kaybetmeden, sırasını bozmadan ve onu boğmadan taşınması gerekiyor. Bölüm 5, relay'i, polling ile change data capture'ı ve outbox'ın üretimde nasıl çalıştırılacağını ele alıyor.

Bölüm 5 — Relay'ler, Polling ve Change Data Capture

Parcelly'de bir Çarşamba sabahı düşünün. Orders ekibi transactional outbox'ı iki hafta önce yayına aldı. Her `OrderPlaced` event'i artık siparişin kendisiyle aynı yerel transaction içinde `outbox` tablosuna yazılıyor ve Bölüm 4'teki dual-write penceresi kapandı. Ekip problemi çözülmüş sayıyor ve bir sonraki ticket'a geçiyor.

Relay, her saniye uyanan, gördüğü son `id`'den büyük `id`'li satırları okuyan, onları publish eden ve yeni en yüksek `id`'yi hatırlayan küçük bir süreç. Tek bir instance olarak çalışıyor ve deploy edildiği günden beri kimse onu düşünmedi.

Sonra rutin bir deploy, relay'i yanlış anda değiştiriyor. Eski instance, broker'a 200 satırlık bir batch publish etmiş ama yeni pozisyonunu henüz kaydetmemiş. Yeni instance son kaydedilen pozisyondan başlıyor, aynı 200 satırı okuyor ve onları yeniden publish ediyor. Inventory, zaten işlediği siparişler için ikinci kez stok rezerve ediyor. Bir hafta sonra bir temizlik job'ı ekleniyor ve relay'in henüz okumadığı satırları sessizce siliyor. Ondan sonraki ay, uzun süren bir transaction geç commit oluyor ve satırları tablodan hiç çıkmıyor.

Bu hataların hiçbirisi outbox örüntüsünün kendisinde değil. Hepsi, satırları broker'a taşıyan kısımda. O kısmı, geç başarısız olacak, hızlı toparlanacak ve hâlâ üretebileceği duplicate'ler konusunda dürüst olacak şekilde nasıl çalıştırırsınız?

Problem ve neden zor olduğu

Transactional outbox size tek bir garanti verir: sipariş commit edildiyse, event'ini publish etme niyeti de onunla birlikte commit edilmiştir. Yolculuğun ikinci yarısı hakkında hiçbir şey söylemez. Birinin satırları

okuması, broker'a vermesi ve bunu yaptığını hatırlaması gerekir. Bu biri **relay**'dir; outbox satırlarını broker'a taşıyan süreç.

Bir relay üç şey yapar: publish edilmesi gereken satırları bulur, onları publish eder ve bunu yaptığını kaydeder. Bunlar üç ayrı adımdır ve broker ile veritabanı bir transaction paylaşmaz. Producer'dan kaldırdığımız aynı dual-write problemi, daha dar bir pencereyle relay'in içinde yeniden belirir.

Crash pencerelerini listelemek için yeterince az. Tablo, relay'in failure semantics'idir: ne başarısız olabilir, bunu nasıl gözlemlersiniz ve hangi garanti ayakta kalır.

Crash anı	Satırın durumu	Sonuç
Publish'ten önce	Yayımlanmamış	Satır yeniden alınır, event bir kez yayımlanır. Güvenli.
Publish'ten sonra, işaretlemeden önce	Yayımlanmamış, ama event broker'da	Satır yeniden alınır, event iki kez yayımlanır. Bir duplicate.
İşaretlemeden sonra	Yayımlanmış	Yapılacak bir şey yok. Güvenli.

Üç adımın ortadaki satırı kapatan hiçbir sıralaması yok. Relay önce işaretleyip sonra publish ederse, arada olan bir crash event'i kaybettirir; outbox'ın var olma nedeni olan hata budur. Bu yüzden relay önce publish eder, sonra işaretler ve ikisi arasındaki bir crash'in duplicate üreteceğini kabul eder. Bu, outbox bölümünün yaptığı takasın aynısıdır: at-least-once delivery ve duplicate yönetiminin consumer'lara itilmesi (Bölüm 7).

Dolayısıyla ilk tasarım sorusu, duplicate'lerden nasıl kaçınılacağı değil. Pencerenin nasıl küçük tutulacağı, satırların makul bir sırayla çıktığından nasıl emin olunacağı ve relay'in geride kaldığının bir müşteriden önce nasıl görüleceği. İkinci tasarım sorusu, satırları okumak için iki mekanizma arasında bir seçim: tabloyu polling etmek ya da veritabanının kendi transaction log'unu okumak. Her biri zorluğu başka bir yere taşır.

Polling relay

Bir **polling relay**, outbox tablosunu belirli aralıklarla sorgular, bulduğu her şeyi publish eder ve bu satırları yayımlandı olarak işaretler. Çalışan en basit mekanizmadır ve rakamlar aksini söylemedikçe ben buradan

başlardım.

Temel döngü

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

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

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

`selectUnpublished`'in arkasındaki sorgu, çoğu polling relay'in yanlış yaptığı yerdir; bu yüzden daha yakından bakmaya değer. Bariz şekil, "bana en son işlediğimden büyük `id`'li satırları ver" olur. Verimli görünür ve güvensizdir.

Id üzerindeki bir cursor neden güvenli değil

Satır id'leri genellikle transaction commit olduğunda değil, insert çalıştığında atanır. İki Orders transaction'ı düşünün. A transaction'ı 101 id'li bir outbox satırı insert eder ve commit etmeden önce yavaş bir fonksiyonu çağırarak bir süre harcar. B transaction'ı 102 id'li satırı insert eder ve hemen commit eder. Relay poll eder, 102'yi görür, publish eder ve cursor'ını 102'ye taşır. Sonra A commit olur. 101 numaralı satır artık cursor'ın gerisinde kalmıştır ve hiçbir zaman okunmayacaktır.

Hata sessizdir. Hiçbir şey hata vermez, hiçbir log satırı belirmez ve event yayımlanmaz. Bu, outbox'ın ortadan kaldırmak için kurulduğu kaybın aynısıdır; bir optimizasyonla yeniden ortaya çıkmıştır.

Benim tercih ettiğim çözüm, bir pozisyonu hatırlamayı bırakıp bunun yerine state'e göre sorgulamak: `published_at`'i null olan satırları `id`'ye göre sıralı seçin. Geç commit olan bir satır, bir sonraki poll geldiğinde hâlâ yayımlanmamıştır; dolayısıyla bulunur. Bedeli, yayımlanmamış satırlar üzerinde bir index ve o sorgunun ucuz kalacağı kadar küçük kalması

gereken bir tabludur. Veritabanınız destekliyorsa, yalnızca yayımlanmamış satırları kapsayan kısmi bir index, tablonun ne kadar geçmiş tuttuğundan bağımsız olarak sorguyu hızlı tutar. Desteklemiyorsa, bu bölümün ilerisindeki retention işi daha çok önem kazanır. Kısmi bir index maliyetin tamamını ortadan kaldırmaz: eski satır versiyonlarını tutan veritabanlarında her satır insert edilir, güncellenir ve sonra silinir; bu çalkantı, engine'in temizlemesi gereken ölü satırlar bırakır.

State tabanlı okuma kaybı giderir. Sırayı gidermez; o ayrı bir sorudur ve sırayı korumak bölümü ona cevap veriyor.

Batching ve polling aralığı

İki ayar relay'in davranışını şekillendirir: poll başına kaç satır aldığı ve yapacak bir şey olmadığında ne kadar uyuduğu.

Daha büyük bir batch, sorgunun ve işaretleme güncellemesinin maliyetini yayar ve throughput'u artırır. Aynı zamanda crash penceresini genişletir: 500 satırlık bir batch'in yarısında ölen bir relay, yeniden publish edilecek 500'e kadar event bırakır. Daha küçük bir batch, crash'ten sonra daha az yeniden publish eder ve aşağıda anlatılan satır sahiplenme ile lock'ları daha kısa süre tutar. Diyelim ki 50 ila 200 satır makul bir ilk tahmin; ayar yapmadan önce ölçün.

Polling aralığı, latency için bir tabandır. Bir saniyelik aralıkla bir event, relay ona bakmadan önce neredeyse bir saniye bekleyebilir. Aralığı kısaltmak latency'yi düşürür ve veritabanının karşılaması gereken boş sorgu sayısını artırır. Parcelly için, `OrderPlaced`'in commit edilmesi ile publish edilmesi arasındaki bir iki saniyelik gecikme alıcılar için görünmezdir. Ekran başında bir insanın beklediği bir akış için öyle olmayabilir.

Publish'ten sonra işaretleme

İşaretleme adımı kendi özenini hak eder. Her satırı publish edin, broker'ın acknowledgment'ını toplayın ve yalnızca broker'ın onayladığı satırları işaretleysin. Publish'i onaylanmamış bir satırı, çağrı hatasız dönmüş olsa bile işaretlemeysin; çünkü birçok client kütüphanesi asenkron gönderir ve hatayı sonradan bildirir.

```

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

```

Bu fonksiyon tabloya hiç dokunmaz. Broker'ın onayladığı id'leri döndürür ve bu bölümdeki her relay tam olarak o id'leri işaretler, başka hiçbirini değil. Başarısız olan satır ve batch'te ondan sonra gelen her satır yayımlanmamış kalır ve bir sonraki poll'da yeniden okunur. Publish etme ve işaretleme bilerek ayrı tutulur; böylece işaretleme, gerçekleştiği tek yerde bir lock ya da lease ile korunabilir. Her event, aggregate type'ının topic'ine, mesaj anahtarı olarak `partition_key` ile gider; Bölüm 4'ün kullandığı yönlendirmenin aynısı.

`break` bilinçli bir tercihtir ve aşağıda yeniden gündeme gelecek olan `ordering` ile ilgilidir. Başarısız bir satırı atlayıp ondan sonraki satırları publish etmek daha hızlıdır ve bir aggregate'in event'lerinin sırasını değiştirebilir. Durmak daha yavaştır ve güvenlidir. Ben dururdum ve relay'in takıldığını monitoring'in bana söylemesine izin verirdim. Bu taslak sadelik için tüm batch'i durdurur; yalnızca etkilenen anahtarı bloklamak, aşağıda sırayı korumak başlığı altında anlatılan inceltmedir.

Bir aktif relay mi, birkaç tane mi

En basit topoloji, tüm işi yapan tek bir relay instance'ıdır. Ordering davranışı hakkında akıl yürütmek daha kolaydır, çünkü `id` sırasıyla okur ve bu sırayla publish eder; ancak aşağıdaki sırayı korumak bölümünde `id` ile commit sırası hakkındaki uyarıya bakın. Zayıflığı `availability`'dir: o instance ölürse, başka biri başlayana kadar hiçbir şey publish edilmez. Birçok sistem için birkaç saniye içinde yeniden başlatma kabul edilebilirdir ve onu yeniden başlatan bir supervisor ile tek bir relay doğru cevaptır.

Tek bir relay yetişemediğinde ya da publish etmedeki bir boşluk kabul edilemez olduğunda, birkaç tane çalıştırırınız. Şimdi yeni bir probleminiz var: iki relay aynı satırı aynı anda publish etmemeli. Standart araç, skip semantiğine sahip bir row lock'tur. Birçok ilişkisel veritabanı, başka bir transaction'ın zaten kilitlediği satırları atlayarak güncelleme için satır seçmenin bir yolunu sunar; buna genel olarak SKIP LOCKED tarzı locking denir. Relay bir transaction içinde bir batch'i sahiplenir ve diğer relay'ler o satırların üzerinden geçer.

```
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
```

Bu çalışır ve çoğu zaman gözden kaçan maliyetler taşır. Birincisi, relay broker ile konuşurken transaction açık kalır; yani yavaş bir broker, uzun süre tutulan lock'lar ve büyüyen bir açık transaction yığını demektir. Eşzamanlı okuyucular için eski satır versiyonlarını tutan veritabanlarında, uzun süre açık kalan bir transaction ölü satırların temizlenmesini de geciktirir; bu da tam olarak en çok çalkalanan tabloya zarar verir. İkincisi, transaction publish'ten sonra abort olursa, kopan bir bağlantı ya da başarısız bir işaretleme yüzünden, publish ettiği her satır yeniden publish edilir. Daha kısa bir alternatif, satırları bir sahiplenme zamanıyla damgalayan kısa bir claim transaction'ı, hiçbir transaction dışında yapılan bir publish ve ayrı bir mark transaction'ıdır; ölü bir relay'in satırları yeniden sahiplenilebilir olsun diye de bir claim timeout'u eklenir. Bunun bedeli, hakkında akıl yürütülecek bir state'tir. Üçüncüsü, paralel relay'ler global ordering'i kaybeder. Relay A 1 ila 100 numaralı satırları sahiplenirken relay B 101 ila 200'ü sahiplenebilir ve B önce bitirebilir. Farklı siparişlerle ilgili event'ler için bunun önemi yok. Aynı siparişle ilgili iki event için var.

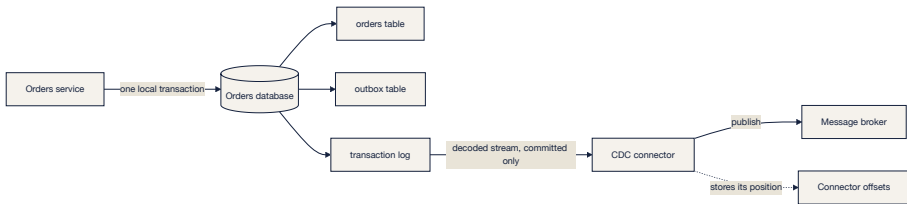
Çare, işi satıra göre değil partition_key'e göre bölmektir. Her relay'e bir anahtar kümesi atayın; örneğin partition_key'i sabit sayıda bucket'a hash'leyin ve her bucket'ı aynı anda tam olarak bir relay'e verin. Bir bucket

içinde satırlar tek bir worker tarafından sırayla işlenir, bucket'lar arasında ise paralel çalışır. Bu, log tabanlı bir broker'daki partition'larla aynı fikirdir; bir adım önceye uygulanmış hâli. "Tam olarak" ifadesi aşağıdaki failover bölümündeki lease'e ihtiyaç duyar ve bir rebalance sırasında bucket devri, anahtar başına sıranın en çok risk altında olduğu andır, çünkü eski ve yeni sahip aynı anahtar için birlikte publish edebilir. Bölüm 9, gerçekçi olarak koruyabileceğiniz sıranın neden anahtar başına sıra olduğunu açıklıyor.

Change data capture

İkinci mekanizma outbox tablosunu hiç sorgulamaz. **Change data capture (CDC)**, bir veritabanının kendi transaction log'unu okur ve commit edilmiş satır değişikliklerini bir change record akışına dönüştürür. Debezium'un tanıdık bir örnek olduğu bir CDC connector'ı, o log'a bağlanır, outbox'a bir satır insert edildiğini fark eder ve broker'a bir mesaj publish eder.

Tablo rolünü korur, çünkü transactional garanti hâlâ onu state değişikliğiyle aynı transaction'da yazmaktan gelir. Değişen, onu kimin okuduğudur. Hiçbir uygulama süreci polling yapmaz. Veritabanı kendi recovery'si ve replikasyonu için her commit edilmiş değişikliği zaten sıralı bir log'a kaydeder ve connector o log'u bir replica'nın okuyacağı gibi okur.



Transaction log'dan broker'a giden yola bakın. Veritabanının connector'lara sunduğu change stream (PostgreSQL ona logical decoding der, MySQL ise binary log'unu sunar) yalnızca commit edilmiş transaction'ları içerir ve bunlar commit sırasıyla görünür. Bu, polling bölümündeki geç commit problemini tümüyle ortadan kaldırır: log zaten commit'lerin toplam bir sırasındır; dolayısıyla geç commit olan bir satır, doğru yerde, geç görünür.

Nasıl davranır

Tasarımdan birkaç özellik çıkar.

Ordering, commit sırasını izler. Event'ler, transaction'ların commit olduğu sırayla görünür. Satırın `partition_key`'ini mesaj anahtarı olarak kullanarak publish eden bir connector, log tabanlı bir broker'da anahtar başına sırayı korur; yeter ki içindeki producer, birlikte uçuştaki olan request'leri bir retry yeniden sıralayamayacak şekilde konfigüre edilmiş olsun. Client'ınızın ayarlarını kontrol edin; aynı uyarı asenkron publish eden bir polling relay için de geçerlidir.

Latency düşük ve sabittir. Bir polling aralığı yoktur. Connector bir değişikliği commit'ten kısa süre sonra görür; dolayısıyla gecikme genellikle küçüktür ve bir şeyin ne sıklıkla uyandığına bağlı değildir.

Connector kendi pozisyonunu takip eder. Log'da ne kadar ilerlediğini kaydeder ve yeniden başlatmadan sonra oradan devam eder. Connector'lar pozisyonu genellikle her event'ten sonra değil periyodik olarak kaydeder; bu yüzden bir crash, son kaydedilen pozisyondan beri her event'i yeniden publish edebilir. CDC de polling gibi at-least-once'tır. Duplicate penceresi taşındı, kaybolmadı.

Published bayrağı isteğe bağlıdır. Connector'ın yayımlanmamış satırları bulması gerekmediği için, tabloda relay'in hatırı için bir `published_at` kolonuna gerek yoktur. CDC kullanan birçok ekip her outbox satırını insert edildikten kısa süre sonra siler. Insert zaten log'a kaydedilmiştir; dolayısıyla connector onu yine görür. Tablo minicik kalır ve temizlik problemi büyük ölçüde ortadan kalkar.

Envelope'un şekillendirilmesi gerekir. Her bir change record bir satırı tarif eder, bir event'i değil. Connector ya da bir transform adımı, outbox kolonlarını standart envelope'a eşlemelidir: `event_id`, `event_type`, mesaj anahtarı olarak `partition_key`, gövde olarak `payload`. Çoğu connector tam da bu amaç için routing ve alan eşlemesini destekler. Eşleme mekanik olsun diye outbox kolonlarını envelope ile hizalı tutun.

Neye mal olur

CDC bedava değildir ve maliyetler operasyoneldir.

Artık bir connector çalıştırıyorsunuz, genellikle bir connector framework'ünün içinde, ayrıca onun bağımlı olduğu broker altyapısıyla birlikte. Birinin nasıl başarısız olduğunu öğrenmesi gerekir.

Veritabanının log'unu connector'a açması gerekir; bu da genellikle belirli bir loglama seviyesini etkinleştirmek ve veritabanının connector adına tuttuğu bir replikasyon pozisyonu oluşturmak anlamına gelir. Connector durursa ve kimse fark etmezse, veritabanı atamayacağı log segmentlerini tutmaya devam eder ve disk dolar. Bu, CDC ailesinde bilinen bir hata biçimidir ve ayrıntılar veritabanına göre değişir. "Connector kapalı" durumunu bir page olarak ele alırdım, çünkü hasar broker ile sınırlı değildir.

Outbox tablosunda ve connector'ın izlediği herhangi bir tabloda yapılacak schema değişiklikleri planlanmalıdır. Bir kolon yeniden adlandırması ya da tip değişikliği eşlemeyi bozabilir.

İlk kurulum, snapshot'lar ve uzun kesintilerden sonra connector yeniden başlatmaları da prova edilmeyi hak eder. Davranış ürüne ve veritabanına bağlıdır ve bunu bir diyagramdan varsaymazdım.

Not: CDC bir teknikler ailesidir ve ürünler log'u nasıl okudukları, pozisyonları nasıl sakladıkları ve uzun bir kesintiden sonra ne olduğu konusunda farklılaşır. Bu bölümü genel şekil olarak değerlendirin ve çalıştırdığınız sürümlerin tam davranışı için connector'ınızın ve veritabanınızın dokümantasyonunu okuyun.

Log gittiğinde

"Disk dolar", bir takasın yarısıdır ve diğer yarısı daha kötüdür. Veritabanı ya connector'ın pozisyonu için sınırsız log tutar, ki bu diski doldurur, ya da tuttuğunu bir üst sınırla kısıtlar ya da log'ların süresini bir takvime göre doldurur, ki bu diski korur. Birçok veritabanı seçmenize izin verir. Üst sınır koyarsanız ve connector bu sınırdan daha uzun süre kapalı kalırsa, kaydedilmiş pozisyonu artık var olmayan bir log'a işaret eder. Boşluktaki event'ler kaybolur ya da connector bir snapshot'tan yeniden başlamak zorunda kalır.

Bir re-snapshot'ın bir outbox'a ne yaptığı, daha önceki bir karara bağlıdır. Tablo hâlâ satır tutuyorsa, snapshot hepsini yeniden emit eder; yani tutulan her satır her consumer için bir duplicate olur. CDC'de yaygın olduğu gibi satırlar insert'ten hemen sonra silindiyse, snapshot hiçbir şey bulmaz ve boşluktaki event'ler tablodan hiçbir biçimde geri alınamaz. Bu

sessiz bir kayıptır: hata yok, duplicate yok, Inventory'ye hiç ulaşmamış bir sipariş. Tabloyu minicik tutan silme, kanıtı da ortadan kaldıran aynı silmedir.

Veritabanı primary'sinin bir failover'ı üçüncü bir durum ekler. Connector'ın pozisyonunun yeni primary'ye taşınıp taşınmadığı veritabanına ve sürüme göre değişir ve bunu bir diyagram okuyarak değil, bir failover'ı prova ederek öğrenirdim.

Bundan üç alışkanlık çıkar. Retention üst sınırını bilerek seçin; diskin, ne kadar uzun bir kesintiden sağ çıkabileceğinizle takası olarak ve sayıyı yazıya geçirin. Headroom için alert kurun; yani connector'ın lag'inin retention penceresinin ne kadarını çoktan kullandığı için, pencere bitmeden önce. Ve kesintiden önce, boşluktaki event'leri nasıl geri alacağınıza karar verin; örneğin sipariş state'ini yayımlananlarla karşılaştıran bir reconciliation job'ından (Bölüm 13). Bölüm 14, hasar verildikten sonraki recovery'yi ele alıyor.

Polling ile CDC arasında seçim yapmak

Hiçbir mekanizma soyut olarak daha iyi değil. Tablo, nerede farklılaştıklarını gösteriyor.

Konu	Polling relay	CDC connector
Hareketli parçalar	Sahibi olduğunuz tek küçük süreç	Connector, framework'ü, veritabanı log ayarları
Latency	Alttan poll aralığıyla sınırlı	Genellikle düşük ve sabit
Ordering kaynağı	Sorgu sırası artı kendi özeniniz	Log'dan gelen commit sırası
Geç commit olan transaction'lar	State tabanlı sorguyla güvenli	Log tarafından ele alınır
Veritabanındaki yük	Tekrarlanan sorgular, row lock'lar	Log okumaları, tutulan log segmentleri
Crash sonrası duplicate'ler	Mümkün, pencere işaretlenmemiş batch'tir	Mümkün, pencere kaydedilmemiş pozisyonur
Tablo büyümesi	Retention işi gerekir	Satırlar hemen silinebilir
En çok zarar veren hata	Sessiz lag, takılı satırlar	Durmuş connector, tutulan log

Konu	Polling relay	CDC connector
Gereken beceriler	Sıradan uygulama becerileri	Veritabanı log iç yapısı, connector operasyonları

Yeni başlayan bir ekip için varsayılanım bir polling relay. Birkaç düzine satır koddur, ekip böyle bir servisi nasıl çalıştıracağını ve debug edeceğini zaten biliyor ve hata biçimleri sorguladığı tabloda görünür. Şu üç şeyden biri doğru olduğunda CDC'ye geçerdim: polling latency'si çok yüksek ve veritabanını yormadan düşürülemiyor, outbox hacmi polling sorgularını ölçülebilir bir maliyet hâline getiriyor ya da zaten bir CDC platformu çalıştırıyorsunuz ve birinin sahibi var.

CDC'yi yalnızca daha ciddi görüldüğü için benimsemezdim. Hiç bir replikasyon log'u işletmemiş ve sipariş akışını onun üzerine koyan bir ekip, anladığı bir problemi anlamadığı bir problemle takas etmiş olur. Terside geçerli: CDC'yi başka amaçlarla zaten çalıştıran bir ekip, yanında bir polling relay yazmaktan az kazanç sağlar.

Relay'i işletmek

Mekanizmayı seçmek için ilk yarısı. İkinci yarısı, onu sağlıklı tutan iştir.

Retention ve temizlik

Bir polling relay, geride yayımlanmış satırlar bırakır. Kendi hâline bırakılırsa tablo sınırsız büyür, yayımlanmamış satırların sorgusu yavaşlar ve veritabanı kimsenin okumadığı satırlara emek harcar. Temizlik tasarlanmalıdır, sonradan eklenmemelidir.

İki seçim önemlidir: yayımlanmış satırların ne kadar tutulacağı ve nasıl silineceği. Bir süre tutmanın değeri var. Bir consumer eksik bir event bildirdiğinde, event'in hiç yayımlanıp yayımlanmadığını öğrenmek için ilk bakılan yer outbox'tır. Diyelim ki birkaç gün makul bir başlangıç noktası. Doğru süre, ekibinizin eksik event'leri ne kadar çabuk fark ettiğine bağlıdır; bu, depolamayla değil insanlar ve alert'lerle ilgili bir sorudur.

```
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)
```

Job'ın lock'ları uzun süre tutmaması ya da replica'ları boğmaması için küçük batch'ler hâlinde silin. Tablo büyükse ve veritabanınız destekliyorsa, zamana göre partition'lamak ve tüm partition'ları düşürmek satır satır silmekten daha ucuzdur. Ne seçerseniz seçin, purge koşulu `published_at is not null`'i içermelidir. Yalnızca yaşa göre silen bir temizlik, relay'in geride kaldığı gün hiç yayımlanmamış event'leri silecektir. Açılıştaki senaryodaki hata budur ve bir duplicate değil, bir veri kaybıdır.

Uyarı: Outbox satırlarını asla yalnızca yaşa göre purge etmeyin. Retention sürenizden daha uzun süre takılı kalan bir relay'in yayımlanmamış satırları silinir ve o event'ler sonsuza kadar gider. Yalnızca yayımlandı olarak işaretlenmiş satırları purge edin ve yaşanan yayımlanmamış satırlar için alert kurun.

CDC ile temizlik farklıdır ama yok değildir. Satırlar insert'ten hemen sonra silinebilir; dolayısıyla tablo küçük kalır. Artık büyüyen şey tutulan log'dur ve izlenecek şey connector'ın ona karşı lag'idir. Hemen silmek ayrıca, daha önceki kaybolmuş log bölümünün açıkladığı gibi, bir re-snapshot'ın hiçbir şeyi geri alamayacağı anlamına gelir; bu yüzden hemen silmeyi, yayımlanmış satırların kısa bir süre tutulmasıyla tartın.

Relay'i monitor etmek

Duran bir relay dışarıdan görünmez. Orders hâlâ başarılı olur, outbox satırları kabul eder ve diğer her servis sessizce event'siz kalır. Kimse hata almaz. Bu yüzden relay'in kendi sinyallerine ihtiyacı var ve lansmandan önce üçünü devreye alırdım.

1. **En eski yayımlanmamış satırın yaşı.** Bu tek başına en iyi sayıdır, çünkü kullanıcıların deneyimlediği şeyi ölçer: en yavaş event'in ne kadardır beklediğini. Teknolojiye değil, iş alanına bağlı bir eşik aşıldığında alert verin. Bir alıcının onay e-postası bir dakika içinde gelmesi gerekiyorsa, beş dakikalık en eski yayımlanmamış yaş bir incident'tir.
2. **Yayımlanmamış satır sayısı.** Büyüyen bir backlog, en eski satır hâlâ genç olsa bile relay'in producer'dan yavaş olduğunu söyler.
3. **Publish hata oranı ve batch süresi.** Bunlar ilk ikisini açıklar. Her poll'da aynı satırı retry eden bir relay, sabit bir hata sayısı ve büyüyen bir yaş gösterir.

Bir CDC connector'ı için aynı sorular farklı birimlerle geçerlidir: connector'ın pozisyonunun log'un başının ne kadar gerisinde olduğu, çalışıp çalışmadığı ve veritabanının onun için ne kadar log tuttuğu. Bölüm 13 bu sinyallere ve lag metric'lerinin kaçırdıklarını yakalayan reconciliation'a geri dönüyor.

Failover ve çifte publish

Bir relay kendi yeniden başlatmalarından sağ çıkmalıdır ve bu, iki instance'ın kısa süre aynı anda çalıştığı durumu da içerir. Deploy'lar, eski instance'ı ölü gösteren ağ bölünmeleri ve eski süreç çıkmadan bir süreci yeniden başlatan orchestrator'lar bu örtüşmeyi üretir.

Tek aktifli bir relay için bir lease kullanın: veritabanında, aktif relay'in kısa aralıklarla yenilediği ve sahibi durursa süresi dolan bir satır ya da lock. Süre dolduktan sonra bir standby devralır. İnsanları şaşırtan ayrıntı şudur: eski sahip lease'inin dolduğunu bilmeyebilir. Uzun bir garbage collection ya da takılmış bir ağ yüzünden duraklatılıp uyanabilir ve artık sahibi olmadığı bir batch'i publish edebilir.

```

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

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

```

Artan token bir fencing token'dır ve fikir yalnızca korunan kaynağın onu kontrol ettiği yerde işe yarar. Veritabanı kontrol edebilir: bayat bir token taşıyan bir commit'in başarısız olması sağlanabilir ve Bölüm 7 tam olarak bunu consumer'lar ve API'ler için kullanıyor. Broker çoğu zaman edemez; dolayısıyla token, bayat bir relay'in publish etmesini engellemez. İşaretleme başka bir konudur. Bayat sahip yalnızca broker'ın onayladığı satırları işaretler; yani işaretleri doğrudur ve onları geri almak yalnızca daha fazla duplicate yaratırdı. Bir lease'in sınırlaması gereken şey eşzamanlı publish'tir, çünkü birlikte publish eden iki sahip duplicate üretir ve bir anahtarın daha eski bir satırını daha yeni bir satırın arkasına koyabilir.

Dört şey bunu sınırlar. Lease TTL'ini bir batch'in en kötü durum süresinin çok üzerinde tutun ve batch'leri küçük tutun, çünkü lease döngü başına bir kez yenilenir ve tek başına yavaş bir broker onu aşabilir. Taslağın yaptığı gibi her publish'ten önce kalan lease'i kontrol edin. Log tabanlı bir broker epoch ile producer fencing destekliyorsa onu kullanın; böylece eski sahibi

broker'ın kendisi reddeder. Ve eski sahip uyandığında, başka bir şey publish etmeden önce, lease satırını lock'layan bir transaction içinde veritabanına sorarak öğrenmesine izin verin. Lock tutmayan bir kontrol bir check-then-act'tir ve aynı şekilde başarısız olur.

Dürüst sınır budur. Failover çifte publish üretebilir ve hiçbir lease düzeni bunu ortadan kaldırmaz. Lease'in size kazandırdığı, pencerenin kısa ve nadir olmasıdır. Zararı ortadan kaldıran şey, her iki publish'te de sabit olan `event_id`'yi kullanarak ikinci kopyayı aynı event olarak ele alan bir consumer'dır. Bu yüzden relay `event_id`'yi outbox satırından almalı ve publish ederken asla yenisini üretmemelidir. Her gönderimde yeni bir tanımlayıcı üreten bir relay, zararsız bir duplicate'i görünürde iki ayrı event'e dönüştürür. Bazı broker'lar producer'ın verdiği bir deduplication id'sini kabul eder ve bir zaman penceresi içindeki tekrarları atar; sizinki öyleyse, onu `event_id`'ye ayarlayın. Pencereyi bedavaya daraltır ve consumer'ın inbox'ının yerini tutmaz. Bölüm 7, tüketen tarafı ele alıyor.

Sırayı korumak

Outbox, event'lerin sırasını yalnızca relay'in koruduğu ölçüde korur ve relay de yalnızca tablonun tuttuğu sırayı koruyabilir. Bölüm 4, `id` sırasının commit sırası olmadığını gösterdi: bir transaction bir `id` alabilir, bir lock'ta bekleyebilir ve daha sonraki bir `id` almış bir transaction'dan sonra commit olabilir. `id`, insert sırasıdır, commit sırası değil.

Bir polling relay için bundan iki sonuç çıkar. Bir anahtarın daha düşük bir `id`'si, aynı anahtarın daha yüksek bir `id`'si zaten publish edildikten sonra commit olursa, ikisi ters sırayla çıkar ve geç kalan satırı kurtaran state tabanlı sorgu, onu geç publish eden şeyin ta kendisidir. İkisi, id'leri ters olarak tek bir poll'da görünür hâle gelirse, tek bir batch içinde ters sırayla çıkarlar. Dolayısıyla "`id` sırasıyla" publish eden bir relay, iki koşuldan biri sağlanmadıkça, aggregate başına event'leri aggregate'in değiştiği sırayla teslim etmez.

Birincisi, producer'ların outbox satırını insert etmeden önce aggregate satırını lock'lamasıdır; böylece aynı anahtarlı transaction'lar serileşir ve id'leri commit sıralarını izler (Bölüm 4). İkincisi, sıranın veride taşınmasıdır: consumer'ların karşılaştırdığı ve Bölüm 9'un kullandığı `aggregate_version` kolonu. İkisini de yapardım. Lock, relay için ucuz bir sigortadır ve version, relay'den sonraki her şeyden sağ çıkan şeydir. Bir

CDC connector'ı ilk problemden kaçınır çünkü log commit sırasında, ancak yine de partition'lar arasında size yardımcı olmaz; dolayısıyla version yararlı kalır.

Bu durumda bile relay'in kendi seçimleri önemlidir. Bunların her biri sahip olduğu sırayı bozabilir:

- Başarısız bir satırı atlayıp aynı anahtarın sonraki satırlarını publish etmek.
- Paralel relay'leri anahtara göre bölmeden çalıştırmak.
- Başarısız bir publish'i bant dışında retry etmek; böylece sonraki bir satır önce ulaşır.
- Bir aggregate'in event'leri için farklı mesaj anahtarları kullanmak.

Güvenli varsayılan, anahtar başına head-of-line blocking'dir: `order-81724` için bir satır publish edilemiyorsa, `order-81724` için sonraki satırları publish etmeyi durdurun ve diğer anahtarların satırlarının devam etmesine izin verin. Bu, bloke olan tek aggregate için latency'ye mal olur ve onun sırasını korur. Sorun hiçbir zaman publish edilemeyecek bir satırsa, örneğin serializer'ın reddettiği bir payload, birinin onu onarması gerekir ve yukarıdaki monitoring onun bunu öğrenme yoludur.

Önceki taslağın ne yaptığı konusunda net olun. Tüm batch üzerinde bir `break` ve en eski önce gelen bir sorguyla, yayımlanamayan tek bir satır her poll'da ilk sırada döner ve her anahtarı sonsuza kadar durdurur. Anahtar başına bloklama iki ekleme gerektirir; `publishAcked`'in `break`'inin yerini alan bir inceltme olarak gösteriliyor: relay bu batch'te hangi anahtarların başarısız olduğunu hatırlar ve tekrar tekrar başarısız olan bir satır park edilir; böylece sorgu onu döndürmeyi ve anahtarının sonraki satırlarını döndürmeyi bırakır.

```

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

```

Select, park edilmiş satırı olan her anahtarı hariç tutar; böylece o anahtarın sonraki satırları bekler ve diğer anahtarların satırları akar. Park etmek, Bölüm 4'ün `failed_at` kolonunun gerektiği andır; bir `attempts` sayacıyla birlikte, böylece geçici bir hata satır park edilmeden önce birkaç kez retry edilir. Park edilmiş satır, `failed_at`'i set edilmiş olandır. Park edilmiş her satır için alert kurun, çünkü park edilmiş bir satır bloke olmuş bir aggregate'tir ve bir insanın onu onarması ya da atması gerekir. Maliyet iki kolon, index'lenmesi daha zor bir sorgu ve ilk incident'tan önce birinin yazması gereken bir onarım prosedürüdür.

Sıra, relay'den sonra da consumer'da ve broker'da kaybolur. Üzerine düşeni yapan bir relay size korunmaya değer bir dizi verir; bunun yolun geri kalanından sağ çıkacağının garantisini değil. Bölüm 9, sırayı broker'dan handler'a kadar izliyor.

Trade-off'lar ve maliyetler

Her iki mekanizma da sadeliği operasyonel erişimle takas eder ve ikisi de kapatamayacağınız bir duplicate penceresi bırakır.

Bir polling relay, ekibinizin zaten anladığı kodun içinde kalmak karşılığında veritabanı yüküne ve latency'ye mal olur. Zayıf noktaları, büyüyen bir tablodaki yayımlanmamış satırların sorgusu, birkaç relay çalıştığında lock davranışı ve kimse en eski satırı izlemediğinde sessiz

lag'dir. Sisteminiz mütevazı bir hızda publish ediyorsa ve bir saniyelik gecikme kabul edilebilirse, polling relay sahip olma maliyeti açısından yenilmesi zor bir seçenektir.

CDC, daha düşük latency, commit sırasıyla okuma ve küçük kalan bir tablo karşılığında yeni bir bileşene ve yeni becerilere mal olur. Zayıf noktaları, veritabanının log tutmasına yol açan durmuş bir connector, durmuş bir connector altında süresi dolup event kaybettiren sınırlı bir log, eşlemeleri bozan schema değişiklikleri ve nadiren prova edilen recovery prosedürleridir. Connector'ın sahibi olacak kimseniz yoksa, mimari diyagram ne kadar çekici görünürse görünsün, onu benimsemezdim.

Bir outbox relay'ini hiç kullanmamanın zamanı: bir event kayıpsız yitirilebiliyorsa, örneğin kaçırılan bir mesajın hiçbir şeye mal olmadığı bilgilendirici bir bildirim, commit'ten sonra doğrudan publish kabul edilebilir ve daha ucuz olabilir. Event'leriniz, state ile outbox arasında yerel bir transaction sunmayan bir veritabanındaki state'i yansıtmak zorundaysa, relay bunu onaramaz ve farklı bir tasarıma ihtiyacınız vardır.

İkisinde de ortak olan iki maliyet var. Hiçbiri size exactly-once publishing vermez: publish ile işaretleme arasındaki ya da publish ile pozisyon kaydı arasındaki bir crash'ten sonraki duplicate, tasarımın parçasıdır. Ve relay ile producer envelope üzerinde anlaşamıyorsa hiçbiri işe yaramaz; bu yüzden outbox kolonlarını ve envelope alanlarını uyumlu tutun.

Sık yapılan hatalar

- **State tabanlı bir sorgu yerine id üzerinde bir cursor kullanmak.** Yayınlanmamış satırlar için tarama yapmaktan kaçındırdığı için cazip gelir. Geç commit olan transaction'lar o zaman cursor'ın gerisinde kalır ve hiç yayınlanmaz. `published_at is null` ile sorgulayın ya da zaten commit'e göre sıralayan log tabanlı bir mekanizma kullanın.
- **Satırları broker onaylamadan önce işaretlemek.** Önce işaretlemek duplicate'leri kaldırıyor gibi görünür. Bir crash'i kayıp bir event'e çevirir. Publish edin, acknowledgment'ı bekleyin, sonra işaretleyin.
- **Relay'de yeni bir event tanımlayıcısı üretmek.** Gönderen relay olduğu için zararsız hissettirir. O zaman consumer'ın inbox'ı yeniden yayınlanmış bir event'i aynıysa olarak tanıyamaz. `event_id`'yi satırdan kopyalayın.

- **Yalnızca yaşa göre purge etmek.** Tabloyu küçük tutar ve relay'in sağlıklı olduğu her testi geçer. Relay sağlıklıysa yayımlanmamış event'leri siler. Yalnızca yayımlanmış satırları purge edin ve yaşlı yayımlanmamış olanlar için alert kurun.
- **Throughput'u yüksek tutmak için başarısız bir satırı atlamak.** Aynı aggregate'in sonraki satırları o zaman onu geçer. Bunun yerine anahtar başına bloklayın, sürekli başarısız olan satırı park edin ve nedeni düzeltin.
- **id sırasının commit sırası olduğuna inanmak.** Tek yazıcılı her testte tutar. Eşzamanlı yazıcılar altında daha düşük bir id daha geç commit olabilir ve relay çifti ters sırayla publish eder. Outbox insert'ünden önce aggregate satırını lock'layın ve aggregate_version taşıyın.
- **Lease'in broker'ı fence'leyeceğine güvenmek.** Lease'i dolduktan sonra uyanan bayat bir relay yine de publish eder, çünkü broker token'ı kontrol etmez. Batch'leri TTL içinde tutun, her publish'ten önce lease'i kontrol edin ve gerisini consumer'ların emmesine bırakın.
- **Kimsenin seçmediği bir retention üst sınırıyla CDC çalıştırmak.** Connector bir hafta sonu boyunca kapalı kalıp pozisyon gidene kadar varsayılan bir ayrıntı gibi hissettirir. Üst sınırı seçin, headroom için alert kurun ve bir re-snapshot'ın outbox'ınıza ne yaptığını bilin.
- **Birkaç relay'i anahtara göre bölmeden çalıştırmak.** Row lock'lar çifte işi önler ve sıra hakkında hiçbir şey söylemez. Anahtar bucket'larını tek worker'lara atayın.
- **Relay'i monitor etmemek.** Outbox producer'ı güvenilir kılar; bu da relay'in hatasını geriye kalan tek hata yapar ve durmuş bir relay, sessiz bir sistemden tamamen ayırt edilemez. En eski yayımlanmamış satırın yaşı için alert kurun.
- **Bir connector kesintisini broker problemi saymak.** CDC'de durmuş bir connector, veritabanının log'unu tutmasına yol açar. Onun için page verin ve yalnızca lag'i değil, diski de izleyin.

Temel çıkarımlar

- Bir relay üç ayrı şey yapar: bulur, publish eder ve işaretler; broker ile veritabanı bir transaction paylaşmaz, dolayısıyla publish ile işaretleme arasındaki bir crash her zaman olası bir duplicate bırakır.

- Önce publish edin, ikinci olarak işaretleyin: diğer sıra bir crash'i kayıp bir event'e çevirir ki outbox'ın ortadan kaldırmak için var olduğu hata budur.
- Bir polling relay, yayımlanmamış satırları artan bir id cursor'ıyla değil, state'e göre seçmelidir; çünkü transaction'lar id'leri ayırdıklarından farklı bir sırayla commit olur; yine de aggregate başına sıra, ancak producer'lar önce aggregate'i lock'layıp bir `aggregate_version` taşırsa tutar.
- Birkaç polling relay çalıştırmak için, satırları skip-locked tarzı locking ile sahiplenin ve `partition_key`'e göre bölün; böylece bir aggregate'in satırları her zaman tek bir worker tarafından sırayla işlenir.
- Change data capture, commit edilmiş değişiklikleri veritabanı log'undan commit sırasıyla okur ve polling'i ve geç commit problemini ortadan kaldırır; bedeli, ekibinizin sahiplenmesi gereken bir connector, dolu bir disk ile kaybolan event'ler arasında bir log-retention takası ve becerilerdir.
- Latency, yük ya da mevcut bir CDC platformu aksini söylemedikçe bir polling relay ile başlayın ve seçimi sadeliğin operasyonel erişimle takası olarak görün.
- En eski yayımlanmamış satırın yaşını monitor edin, yalnızca yayımlanmış satırları purge edin, hiç publish edilmeyen satırları park edin, `event_id`'yi outbox satırından kopyalayın ve failover'dan sonra duplicate bekleyin; çünkü lease eşzamanlı publish'i sınırlar ve geri kalanı zararsız kılacak tek şey idempotent bir consumer'dır.

Relay artık her event'i en az bir kez, korunmaya değer bir sırayla ve bazen iki kez teslim ediyor. Sıradaki problem broker'ın öbür tarafında duruyor: işi başarısız olduğunda bir consumer ne yapmalı; Bölüm 6 bunu retry'larla ele alıyor.

KISIM III

Güvenilir Şekilde Tüketmek

Bölüm 6 — Retry'ı Doğru Yapmak

Parcelly'de bir satıcı promosyonunun başladığı bir Salı sabahı düşünün. Orders bir `CapturePayment` command'ı gönderiyor ve Payments bunu harici ödeme sağlayıcısına iletiyor. Sağlayıcı zorlanıyor ve yavaş yanıt veriyor. Payments'ın çağrıları timeout'a düşmeye başlıyor; Payments da mesajı başarısız sayıyor ve broker onu yeniden teslim ediyor. Her Payments instance'ı aynı şeyi, aynı anda, aynı sabit bir saniyelik bekleyle yapıyor.

Bir dakika içinde sağlayıcı, promosyonun normal yükünü değil, normal yük artı geri dönen her başarısız çağrıyı, tekrar tekrar, senkronize dalgalar hâlinde alıyor. Birkaç saniye sürebilecek olan sağlayıcı yavaşlığı, retry'lar gelmeye devam ettiği sürece sürüyor. Bu arada Payments'ın consumer'ları başarılı olmayacak mesajları retry etmekle meşgul; sağlıklı mesajlar onların arkasında kuyruğa giriyor. O yığının bir yerinde, sağlayıcıda aslında başarılı olmuş birkaç capture yeniden gönderiliyor.

Burada kimse kötü bir satır kod yazmadı. Her instance mantıklı olanı yaptı: başarısız oldu, bekledi, yeniden denedi. Hasar, kimsenin koordine etmediği bin mantıklı kararın toplamından geldi.

Bu bölüm, retry'yı bir refleks olmaktan çıkarıp bir karar hâline getirmekle ilgili. Cevapladığı soru şu: bir çağrı başarısız olduğunda, yeniden deneyip denemeyeceğinize, ne zaman, ne sıklıkta ve hangi katmanda deneyeceğinize nasıl karar verirsiniz ki retry sistemin toparlanmasını engellemek yerine ona yardım etsin?

Problem ve neden zor olduğu

Retry, yeniden deneme sözüdür; başarı sözü değil. Bu bölümde bir **attempt**, herhangi bir denemedir; dolayısıyla attempt sayısı bir artı retry sayısına eşittir. Bariz yaklaşım, "çağrıyı bir döngüye sar ve birkaç kez retry et", dört ayrı nedenle başarısız olur ve bunlar birbirini büyütür.

Birincisi, **her hata retry etmeye değmez**. Kart reddedildiği için reddedilen bir request, onuncu attempt'te de reddedilir. Onu retry etmek kapasiteyi boşa harcar ve birinin gerçek problemi fark ettiği anı geciktirir. Retry döngüsünün baktığı hatanın ne türden olduğunu bilmesi gerekir ve çoğu döngü bilmez.

İkincisi, **retry'lar yükü en kötü anda artırır**. Bir dependency aşırı yüklendiğinde, bozulduğunda ya da yeniden başlarken başarısız olur. Retry'lar tam da o dependency'ye yönelmiş ek request'lerdir. Her çağırın üç kez retry ederse, yani toplamda dört attempt yaparsa, zorlanan bir servis en az dayanabileceği anda normal trafiğinin dört katına kadarını görür.

Üçüncüsü, **retry'lar senkronize olur**. Birlikte başarısız olan çağırınlar birlikte retry eder. Sabit bir gecikme, tek bir hatayı periyodik bir sıçramaya dönüştürür, çünkü her çağırın aynı anda uyanır. Bu thundering herd'dür ve gecikmenin yalnızca uzunluğa değil, rastgeleliğe de ihtiyaç duymasının nedeni budur.

Dördüncüsü, **retry, ilk sonucu bilinmeyebilecek bir eylemi tekrarlar**. Bölüm 3, bir timeout'un request'in işlenip işlenmediğini söylemediğini göstermişti. Bu yüzden bir timeout'tan sonra retry etmek, işi iki kez yapma riskini taşır. Idempotency (Bölüm 7) olmadan retry, gerçekleşmeyi bekleyen bir duplicate'tir.

Bu bölümdeki geri kalan her şey, bu dört problemten birine verilen bir cevaptır. Sınıflandırma birincisini cevaplar. Jitter'lı backoff ve retry budget ikinci ile üçüncüyü cevaplar. Bölüm 7'nin kurduğu idempotency dördüncüyü cevaplar ve onun yokluğunu ele alan aşağıdaki bölüm, neden atlanamayacağını açıklar.

Hataları sınıflandırmak

Nasıl retry edeceğinize karar vermeden önce, edip etmeyeceğinize karar verin. Bölüm 1, herhangi bir birleşim yeri için sorulacak üç soru vermişti: ne başarısız olabilir, bunu nasıl gözlemlersiniz ve hangi garanti ayakta kalır. Cevapların toplamı bir hatanın semantiğidir ve bir retry kararı, o semantiğin iş başında hâlidir. Her hatayı, yeniden denemenin yardımcı

olup olamayacağı hakkında ne söylediğine bağlı olarak üç sınıftan birine ayırmayı ve ardından mesaj hakkında hiçbir şey söylemeyen dördüncü bir sinyal eklemeyi faydalı buluyorum.

Transient hatalar

Transient hata, nedeninin kendiliğinden ortadan kalkması beklenen hatadır. Bir bağlantı sıfırlanması, bir timeout, bir `503 Service Unavailable`, veritabanında bir deadlock kurbanı, bir throttling response'u, broker'da bir leader seçimi. Request yanlış değildi. Dünya kısa süreliğine yoldaydı.

Doğru cevap bir gecikmeyle retry etmektir, çünkü nedenin ortadan kalkmasına izin veren şey gecikmedir.

Kalıcı hatalar

Kalıcı hata, nedeni request'te ya da veride yattığı için her attempt'te yinelenen hatadır. Doğrulamadan geçemeyen bir payload, gelmesi gerektiğinden çok sonra bile eksik kalan bir kayda referans (event-driven bir sistemde eksik bir kayıt transient olabilir, çünkü onu oluşturan event geç gelebilir; Bölüm 9 bu durumu ele alıyor), bir `400 Bad Request`, yetersiz bakiye nedeniyle reddedilen bir kart, schema sürümü consumer'ın anlamadığı bir event.

Retry etmek boşa emektir ve bazen zararlıdır. Doğru cevap durmak, nedenini kaydetmek ve mesajı bir insanın ya da bir onarım job'ının ilgilenebileceği bir yere göndermektir. Kalıcı hataların nereye gittiğini Bölüm 8 ele alıyor. Özellikle kart reddini not edin: Payments için bu hiç hata değildir. `PaymentFailed` event'ine dönüşen beklenen bir iş sonucudur ve hiçbir retry söz konusu değildir.

Bilinmeyen hatalar

Bilinmeyen hata, sınıfını söyleyemediğiniz hatadır. Timeout klasik örnektir; kimsenin öngörmediği bir mesajla gelen beklenmedik bir exception da öyle. Timeout transient olabilirdi. Exception, her attempt'te başarısız olacak, consumer'daki bir bug olabilir.

Bilinmeyen gerçek bir sınıftır, taksonomideki bir boşluk değil ve ona nasıl davrandığınız bir policy kararıdır. Benim varsayılanım, bilinmeyen hataları transient saymak, ancak bilinen transient olanlardan daha sıkı bir limitle; çünkü bilinmeyen bir hatayı yanlışlıkla retry etmenin maliyeti birkaç boş attempt iken, onu yanlışlıkla bırakmanın maliyeti kaybedilen iştir. Her katmanın attempt'leri tükendiğinde, bilinmeyen bir hata kalıcı bir hata gibi ele alınır: tam geçmişiyse dead letter'lara gider.

Bir attempt'in sonucu bilinmiyorsa, yeniden harekete geçmeden önce sorun. Bir retry'dan önce ve kesinlikle bir compensation'dan önce, sağlayıcıya operasyonun durumunu idempotency key'i ile sorun. Capture gerçekleştiyse doğru devam adımı bir void değil, bir refund'dır. Gerçekleşmediyse bir retry ya da bir void güvenlidir. Her iki yönde tahmin yürütmek, bir alıcının Parcelly'nin iptal ettiğini sandığı bir sipariş için tahsilat yapılmasıyla karşılaşmasının yoludur.

Ertelenen: dependency'nin kapalı olduğu biliniyor

Dördüncü sinyal, mesaj hakkında bir hüküm değildir. Bir circuit breaker (aşağıda anlatılıyor) open olduğunda, bir çağrının başarısız olması request hakkında hiçbir şey söylemez: çağrı dependency'ye hiç ulaşmadı. Bu sınıfa **deferred** diyorum. Deferred bir mesaj bir attempt tüketmez, hiçbir limite sayılmaz ve bu yüzden asla dead-letter'a gönderilmez. Consumer çekmeyi bırakır ya da partition'ı pause eder ve mesaj, dependency'yi yeniden çağırmaya değer hâle gelene kadar olduğu yerde bekler.

Bu ayrım önemlidir, çünkü alternatifi aritmetiktir. 30 saniye boyunca open kalan bir breaker, aralarında bir saniye olan üç handler içi attempt'e izin veren bir handler'ın önünde, sağlıklı bir mesajın tüm bütçesini kabaca ilk üç saniyede yakmasına ve hiç sahip olmadığı bir problem yüzünden DLQ'ya düşmesine yol açar.

Sınıflandırmayı açık hâle getirmek

Sınıflandırma, okunabilen ve gözden geçirilebilen bir yerde yaşamak zorundadır. Bir hata sinyalini bir sınıfa ve bir ele alış biçimine eşleyen tablo, bir consumer'daki en değerli küçük artefaktlardan biridir. Payments'ın sağlayıcıya yaptığı çağrılar için örnek bir sürüm şöyle görünür:

Sinyal	Sınıf	Ele alış
Connection reset, DNS failure	Transient	Backoff ile retry
Timeout on a capture call	Bilinmeyen	Aynı idempotency key ile retry, sıkı limit
429, throttling response	Transient	Retry, verilmişse sağlayıcının bekleme ipucuna uy
5xx from the provider	Transient	Backoff ile retry
400, validation failure	Kalıcı	Retry etme, dead-letter'a gönder
Card declined	Kalıcı, iş sonucu	PaymentFailed emit et, retry etme
Unrecognized error	Bilinmeyen	Sıkı limitle retry, sonra dead-letter
Circuit breaker open	Deferred	Attempt tüketilmez; tüketimi pause et, breaker izin verince devam et

İki uyarı. Birincisi, status code'lar ipucudur, garanti değil; hatalı biçimlendirilmiş bir request için 500 dönen bir sağlayıcı, onu sizin için yanlış sınıflandırmıştır ve bunu o sağlayıcıyla deneyimleyerek öğrenmeniz gerekecektir. İkincisi, tablo hiç görmediği herhangi bir sinyal için varsayılan olarak bilinmeyene dönmeli, asla kalıcıya ya da transient'e değil. Kimsenin sınıflandırmadığı yeni bir hata hiçbir şeyin kanıtı değildir.

Sınıflandırıcının kendisi küçüktür:

```
// 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"
```

Jitter'lı exponential backoff

Retry etmeye karar verdikten sonra sıradaki soru ne zaman olduğudur. Hemen retry etmek transient bir nedene ortadan kalkması için zaman tanımaz. Sabit bir gecikmeyle retry etmek ona zaman tanır, ama her çağıranı uyum içinde tutar.

Exponential backoff, gecikmeyi her attempt'te büyütür: diyelim bir saniye, sonra iki, sonra dört, bir tavana kadar. Erken retry'lar hızlıdır; bu, bir anda ortadan kalkan hatalara uyar. Sonraki retry'lar yavaştır; bu, gerçekten kapalı olan bir dependency'yi rahatlatır. Tavan, gecikmenin iş alanının tolere edebileceğinin ötesine büyümesini engeller.

Jitter, her gecikmeye rastgelelik ekler. Onsuz, aynı anda başarısız olan çağırılar aynı gecikmeleri hesaplar ve uyum içinde retry eder; sürü backoff'tan bozulmadan çıkar. Onunla retry'lar bir aralığa yayılır ve dependency bir duvar yerine bir leke gibi dağılmış request'ler görür.

Birkaç jitter şeması var. İkisini adıyla bilmeye değer. **Full jitter**'da gecikme, sıfır ile o attempt için üstel tavan arasında rastgele bir değerdir. **Equal jitter**'da gecikmenin yarısı sabit, yarısı rastgeledir; böylece minimum bekleme asla sıfıra yakın olmaz. Full jitter yükü en iyi yayar, equal jitter ise beklemede bir taban garanti eder. Full jitter ile başladım ve equal jitter'a yalnızca dependency çok hızlı retry'lardan zarar görüyorsa geçerdim.

```

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

```

Bu döngü ele aldığı hatalar altında doğrudur, ancak sonraki bölümlerin ele aldığı iki açıdan eksiktir. Çağırının ne kadar zamanı kaldığını bilmez ve tüm sistemin zaten ne kadar retry yaptığını bilmez.

Sağlayıcı, ne zaman yeniden deneneceğine dair bir ipucu gönderirse, örneğin bir throttling response'undaki bekleme değeri, ona uyun. Sağlayıcı kendi toparlanmasını sizin formülünüzden daha iyi bilir. İpucunu hesaplanan gecikmenizin altında bir taban olarak ele alın, onun yerine geçen bir şey olarak değil ve jitter'ı koruyun; böylece aynı ipucuna sahip çağırılar birlikte geri dönmez. Bir taban yine de deadline'ın ötesinde uyumak için bir ruhsat değildir: ipucu kalan süreden uzunsa retry etmeyi bırakın ve bir sonraki katmanın karar vermesine izin verin.

Deadline'lar ve retry budget'ları

Jitter'lı backoff, retry'ların aralığını belirler. İki ek limit, kaç tane olabileceğini belirler.

Deadline'lar

Deadline, sonucun artık yararlı olmadığı andır. Request-driven bir akışın doğal bir deadline'ı vardır: alıcı bir checkout sayfasına bakıyor ve 10 saniye sonra ayrılacak. Arka plandaki bir consumer'ın deadline'ı daha yumuşaktır: taşıyıcının kesme saatinden sonra gelen bir kargo etiketi, hiç gelmeyen kadar geçtir.

Attempt'leri sayan ama zamanı yok sayan retry mantığı, yararlılığın çok ötesine geçebilir. Her biri 30 saniyelik timeout'lu ve aralarında büyüyen gecikmeler olan üç attempt, dakikalarca beklemektir. Retry edilen her operasyona toplam bir zaman bütçesi vermeyi ve attempt sayısı ne derse desin, bir sonraki gecikme onu aşacağında retry etmeyi bırakmayı tercih ediyorum.

```
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)
```

Deadline da yol almalıdır. Orders bir akışa 10 saniye verip Payments'ı çağırıyorsa, Payments kendi taze 10 saniyesini başlatmamalı, kalanı iletmelidir. Aksi hâlde her katman bir önceki katmanın sabrı içinde retry eder ve toplam, kullanıcının bekleyeceği süreyi sessizce aşar.

Retry budget'ları

Çağrı başına bir attempt limiti, birçok çağrı aynı anda başarısız olurken bir dependency'yi korumaz. Her çağrı üç retry limitinin içinde kalır ve dependency yine de normal trafiğinin dört katını alır.

Retry budget, retry'ları çağrı başına değil, toplam trafiğin bir payı olarak sınırlar. Yaygın bir biçim token bucket'tır: her başarılı çağrı bir token'ın bir kesrini ekler, her retry bir tane harcar ve bucket boşaldığında çağrılar retry etmeden hızlıca başarısız olur. Kararlı durumda retry'lar request'lerin küçük bir kesridir. Yaygın bir kesintide budget birkaç an içinde tükenir ve sistem kendi hatasını büyütmeyi bırakır.

Sayılar sizin seçiminizdir ve dependency'nin ne kadar ek yükü emebileceğine bağlıdır. Bir rakam önermeyeceğim, çünkü vereceğim her rakam sizin dependency'niz hakkında bir tahmin olurdu. Önemli olan, budget'ın var olması, bir süreçteki çağırılar arasında paylaşılması ve tükenmesinin bir dashboard'da görünür olması. Süreç başına tutulan, oran tabanlı bir budget genellikle yeterlidir, çünkü her sürecin kendi oranını sınırlaması toplamı zaten sınırlar. Instance'lar arasında paylaşılan bir budget bir seçenektir, kesin bir iyileştirme değil: dağıttık bir sayaç gerektirir ve bu sayaç hata yolunda yeni bir dependency'dir, kendine ait bir hata moduyla.

Bedeli, retry'da başarılı olacak bazı çağrılarının bunun yerine başarısız olmasıdır. Budget'ın amacı budur: bireysel başarıyı sistemin hayatta kalmasıyla takas eder ve bunu bilerek yapmalıdır.

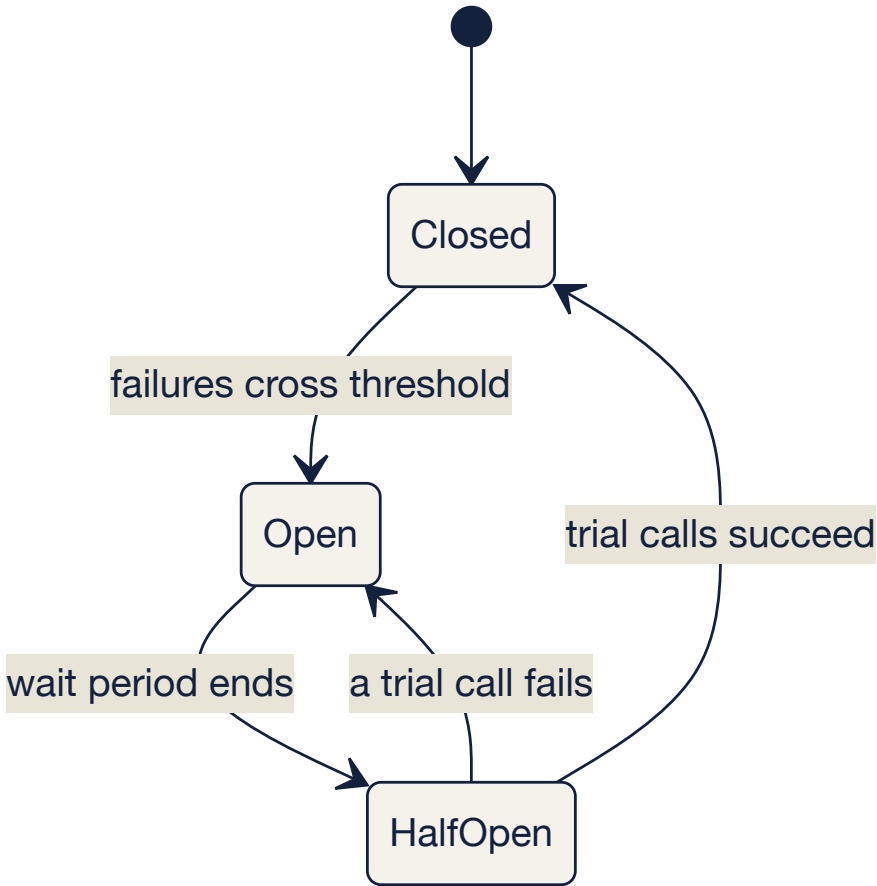
Retry storm'lar ve circuit breaker

Retry storm, retry etmenin başarısız olan bir dependency üzerindeki baskın yük hâline gelmesiyle olan şeydir. Parcelly'nin açılış senaryosu bir örnektir. Üç bileşen gerektirir: yavaş ya da başarısız olan bir dependency, retry eden çağırılar ve çağırılara durmalarını söyleyen hiçbir mekanizmanın olmaması. Katmanlar durumu daha da kötüleştirir. Orders,

Payments'a üç attempt yaparsa ve Payments sağlayıcıya üç attempt yaparsa, tek bir başarısız capture sağlayıcıya dokuz çağrıya dönüşebilir. Üç attempt'lik bir katman daha eklenince 27 olur.

İki savunma doğrudan storm'u hedefler. Yukarıdaki retry budget, sistemin ne kadar retry yapacağını sınırlar. Circuit breaker, çağrı yapıp yapmaması gerektiğini sınırlar.

Circuit breaker, bir dependency'ye yapılan çağrılarının sonuçlarını izler ve üç durumu vardır. Normal işleyişte closed'dur ve çağrılar geçer. Hatalar bir eşiği aştığında open olur ve çağrılar dependency'ye dokunmadan hemen başarısız olur; bu, dependency'ye toparlanması için alan tanır. Bir beklemeden sonra half-open'a geçer ve birkaç deneme çağrısının geçmesine izin verir. Başarılı olurlarsa closed olur. Olmazlarsa yeniden open olur.



Breaker ve retry policy'si birbirini tamamlar ve farklı soruları cevaplar. Backoff, tek bir çağrıda bir sonraki attempt'ten önce ne kadar bekleneceğine karar verir. Breaker, çağrılar arasındaki örüntüden, attempt yapmaya hiç değip değmeyeceğine karar verir. Open durumdaki bir breaker, retry döngüsünün bir sonraki attempt'ini anlık bir hataya çevirir. Bu hata transient değildir ve bilinmeyen de değildir: yukarıdaki sınıflandırmadan gelen **deferred** sinyalidir. Retry döngüsü onu saymamalı, ona göre backoff yapmamalı ve onun üzerinde spin etmemelidir. Kontrolü, neyi yapmayı bırakacağına karar veren consumer'a geri verir.

Bir consumer'ın, request-driven bir çağırının sahip olmadığı ek bir seçeneği vardır. Breaker open olduğunda, bir consumer mesajları tüketip her birini başarısız kılmak yerine broker'dan mesaj çekmeyi bırakmalı ya da partition'ı pause etmelidir. Broker'da kalan mesajlar, retention'ın izin verdiği sürece güvendedir; depolamaya mal olurlar ve consumer lag olarak görünürler. Tüketilen, retry edilen ve başarısız olan mesajlar attempt sayılarını yakar ve hiçbir zaman kendilerine ait olmayan bir problem yüzünden dead-letter'a düşebilir. Pause etmek, breaker'ın consumer tarafındaki biçimidir ve bu bölümdeki diğer şeylerin çoğundan önce onu kurardım.

Pause etmek iki takip sorusu doğurur. Birincisi, hiç trafik akmazken half-open durumunu kimse test etmez. Deneme çağrılarını, deneme olduklarını bilen bir şeyin yapması için concurrency bir ile devam edin ya da sentetik bir probe çağrısı gönderin. İkincisi, log tabanlı bir broker'da pause edilmiş bir consumer poll döngüsünü canlı tutmalıdır; partition'ları pause eder ama polling'i durdurmaz. Aksi hâlde group consumer'ı ölü sayar ve partition'larını yeniden atar (aşağıda anlatılan liveness penceresi).

Breaker'ın bedeli, yanlış başarısız olabilmesidir. Çok hassas eşikler kısa bir aksamada open olur ve transient bir hatayı kesintiye çevirir. Çok gevşek eşikler çok geç open olur. Breaker ayrıca, bir akış sabaha karşı üçte durduğunda birinin anlaması gereken state ekler: ilk soru, dependency'nin mi kapalı olduğu yoksa yalnızca breaker'ın mı open olduğudur.

Aşırı yük altında retry'lar: backpressure

Breaker, başarısız olmuş bir dependency'ye tepki verir. **Backpressure** ondan önceki aşama için disiplindir: dependency yavaş, doygun ya da toparlanıyorken ve sistem ona ne kadar iş vereceğine karar vermek zorundayken. Bölüm 3 onu hata modelinin içine yerleştirir. Bu kısım mekanizmanın kendisidir.

Açılış senaryosuna dönün. Sağlayıcının yavaşlığı bir bug'dan gelmedi. Emebileceğinden fazla işten geldi ve her Payments instance'ı onu beslemeye devam etti. Uçuştaki iş üzerinde limiti olmayan bir consumer tam olarak böyle davranır: çekmeye, çağrı başlatmaya ve zaten fazla ağır olan yığına eklemeye devam eder. Aşağıdaki limitler, kabaca bu sırayla koyacağım limitlerdir.

- **Uçuştaki işi sınırlayın.** Bir consumer'ın aynı anda tuttuğu mesaj sayısına bir tavan koyun. Queue tabanlı bir broker'da bu prefetch ya da in-flight sayısıdır. Log tabanlı bir broker'da poll başına kayıt sayısı artı çalışan handler sayısıdır. Tavan olmadan limit, consumer'ın belleği ve broker'ın redelivery saati olur ve ikisi de sizin seçtiğiniz bir limitten daha kötü limitlerdir.
- **Dependency yavaşlayınca concurrency'yi kısın.** Çağrı başına 50 milisaniyede doğru olan concurrency, çağrı başına beş saniyede yanlıştır, çünkü aynı sayıda slot artık kat kat fazla bekleyen iş tutar. Latency yükseldiğinde eşzamanlı çağrı sayısını düşürün ve latency toparlandığında yavaş yavaş yeniden yükseltin.
- **Başarısız kılmak yerine pause edin.** Dependency doygunken, open bir breaker'da olduğu gibi çekmeyi bırakın. Pause edilmiş bir consumer lag olarak görünür; bu görünür ve güvenlidir. Çeken ve başarısız kılan bir consumer attempt yakar.
- **Canlı trafiğe retry'lardan önce hizmet edin.** Retry'lar ve redrive, aynı dependency için ilk attempt'lerle yarışır. Kapasite kısıtlı olduğunda ilk attempt'ler kazanmalıdır, çünkü retry, sistemin zaten zorlandığını bildiği bir iştir. Retry topic ve redrive consumer'larını daha düşük concurrency ile çalıştırın ya da ana akışın backlog'u varken beklemelerine izin verin.
- **Girişte atın.** Yukarı akışta bir yerde kabul edilen işe bir limit olmak zorundadır; yoksa dependency toparlanırken queue sınırsız büyür.

Yeni işi, hizmet veremeyen bir dependency'nin arkasında süresiz sıraya almak yerine, çağırana net bir sinyalle reddedin ya da erteleyin.

Bu limitler ve retry budget'ı tek bir fikrin iki yüzüdür. Budget, retry'ın eklediği ek işi sınırlar; backpressure ise ek olsun olmasın uçustaki işi sınırlar. Hiçbiri tek başına yetmez: uçustaki işe tavan koymayan bir budget, yavaş bir dependency'nin her slot'u ilk attempt'lerle doldurmasına izin verir ve budget'sız bir uçustaki iş tavanı slot'ların retry'larla dolmasına izin verir.

Bedeli lag ve reddetmedir. Yük atan bir sistem bazı çağıranlara bilerek hayır der ve pause eden bir sistem bazı mesajları geç teslim eder. İkisi de throughput'u hayatta kalmayla takas etmektir ve "hangi iş bekler" sorusunun cevabının sahibi, sayıları ayarlayan mühendis değil, akışın iş sahibidir. Bölüm 8 aynı rate-limit fikrini redrive'a uygular ve Bölüm 9 uçustaki iş sınırını anahtarlı lane'lere uygular.

Retry'lar nerede yaşar

Bir mesaj birkaç katmanda retry edilebilir ve katmanlar farklı davranır. Onları bilmeden karıştırmak, bir mesajın kimsenin amaçladığından çok daha fazla retry edilmesinin ya da hiç edilmemesinin yoludur.

Uygulama seviyesinde retry

Consumer handler'ın içinde retry eder: yukarıdaki kod, sağlayıcıya yapılan bir çağrının etrafına sarılmış. Retry'lar çalışırken mesaj unacknowledged kalır. Bu katman ucuz ve hızlıdır ve milisaniyeler ya da saniyeler içinde ortadan kalkan hatalar için iyidir. Sınırları, beklerken consumer'ın kapasitesini tutması, bir consumer crash'inin retry state'ini kaybettirmesi (broker yeniden teslim eder ve attempt sayısı baştan başlar; bu yüzden süreci çökerten bir mesaj, Bölüm 8'in anlattığı tepite ihtiyaç duyar) ve uzun beklemlerin broker'ın kendi timeout'larını aşma riski taşımasıdır.

Queue tabanlı bir broker'da unacknowledged mesaj **visibility timeout** boyunca gizlenir ve ardından yeniden teslim edilir. Handler içi retry'lar bu timeout'tan uzun sürerse, ilk kopya hâlâ retry edilirken broker yeniden teslim eder ve artık iki kopya çalışıyordur. Log tabanlı bir broker'da, poll'lar arasında çok uzun zaman harcayan bir consumer, consumer group'u tarafından ölü sayılabilir ve partition'ları yeniden atanabilir; sonuç

aynıdır. Uyacağım kural: handler içi retry'ların en uzun toplam süresi broker'ın timeout'unun rahatça içinde kalmalı ve daha uzun olan her şey farklı bir katmana ait olmalıdır.

Broker redelivery

Consumer mesajı başarısız kılar ve broker onu yeniden teslim eder. İki broker ailesi burada farklılaşır ve fark önemlidir.

Queue tabanlı bir broker'da redelivery yerleşiktir. Consumer mesajı nack eder ya da visibility timeout'un dolmasına izin verir ve mesaj, hemen ya da yapılandırılmış bir gecikmeden sonra, herhangi bir competing consumer'ın alabileceği queue'ya döner.

Log tabanlı bir broker kendiliğinden yeniden teslim etmez. Consumer'ın okuma pozisyonu poll ettikçe ilerler ve commit edilen offset yalnızca yeniden başlatılan bir consumer'ın nereden devam edeceğini kaydeder (Bölüm 2 bunu ele alıyor). Bir kaydı işleyememek ve offset'ini commit etmemek bu yüzden kendi başına hiçbir şey yapmaz: consumer bir sonraki kaydı okur ve sonra commit ederse, bu commit başarısız kaydı da kapsar ve o sessizce atlanır. Başarısız kayıt yalnızca bir yeniden başlatmadan, bir rebalance'tan ya da consumer'ın açık bir adımından sonra yeniden okunur.

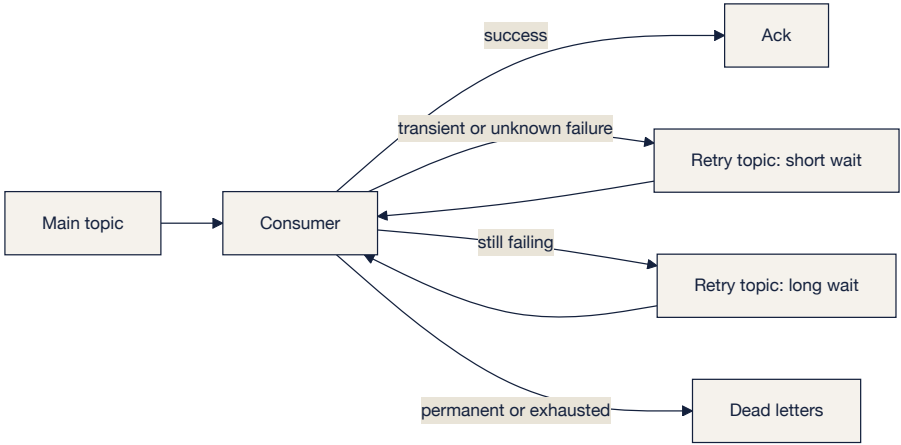
Bu yüzden bir log üzerinde consumer üç eylemden birini seçmek zorundadır. Başarısız offset'e geri seek edebilir ya da partition'ı pause edip yerinde retry edebilir; bu sırayı korur ve arkasındaki her şeyi durdurur. Kaydı bir **retry topic**'e publish edip devam edebilir. Ya da bir dead-letter topic'e yazıp devam edebilir. Her biri, broker'ın sizin için yaptığı bir şey değil, consumer'ın kodunda bilinçli bir karardır.

Broker redelivery durable'dır: consumer crash'inden sağ çıkar. Aynı zamanda kabadır. Birçok broker, backoff olmadan, hemen ya da sabit bir takvimle yeniden teslim eder ve attempt sayısı yalnızca consumer'ın okuyup üzerinde işlem yapması gereken bir delivery sayacı olarak bulunabilir. Onu diğer katmanların altındaki emniyet ağı olarak görün, ana retry policy'si olarak değil.

Retry topic'ler ve delay queue'lar

Bir hata, handler içi retry'ın tutabileceğinden daha uzun bir bekleme gerektirdiğinde, yaygın bir yapı mesajı ana akıştan çıkarıp daha sonra geri gelmesine izin vermektir. Bir mesajı başarısız kılan consumer, onu artırılmış bir attempt sayısı ve bir not-before zamanıyla bir **retry topic'e** (ya da retry queue'ya) publish eder ve orijinali ack eder. Aynı bir consumer ya da farklı bir topic'teki aynı consumer, zaman geldiğinde onu alır. Gecikmeli teslimi ya da mesaj başına gecikmeleri destekleyen broker'lar bunu doğrudan sağlar. Desteklemedikleri yerde yaygın bir düzenleme, her biri sabit bir beklemeye sahip, backoff'un her adımı için bir tane olan bir retry topic merdivenidir.

Bekleme süreleri bir sebeple topic başına sabittir. Bir log'da consumer, vadesi henüz gelmemiş bir kaydı atlayamaz: kayıt hazır olana kadar başta beklemek zorundadır. Bir topic'teki her kaydın bekleme süresi aynıysa topic zaten vade sırasındadır ve başta beklemek ek bir maliyet getirmez. Kısa ve uzun gecikmeleri karıştıran bir topic, kısa olanları uzun olanların arkasında tutardı. Mesaj başına not-before zamanları yalnızca yerel gecikmeli teslimi olan broker'larda işler.



Bu katman ana akışı serbest bırakır; dolayısıyla başarısız bir mesaj artık arkasındaki mesajları tutmaz. Yeniden başlatmalardan sağ çıkar, çünkü state broker'da yaşar. Ayrıca size bakacak bir yer verir: retry topic'inin derinliği, toparlanan bir dependency'yi bekleyen iş miktarının doğrudan ölçüsüdür.

Daha fazla mekanizmaya mal olur. Oluşturulacak, adlandırılacak, monitor edilecek ve güvenceye alınacak daha fazla topic vardır. Yeniden publish, başarısız olabilen bir produce operasyonunun kendisidir ve bu, Bölüm 1'deki dual write'ı geri getirir; bu yüzden sıra önemlidir: önce retry topic'e publish edin, sonra orijinali ack edin. Süreç ikisi arasında ölürse mesaj iki kez var olur. Bu, consumer'ın zaten tolere etmesi gereken bir duplicate'tir.

Üç kural, merdivenin korumak için var olduğu şeye zarar vermesini önler:

- **Topic'i consumer başına sahiplenin.** Bir retry topic tek bir consumer group'a ya da subscription'a aittir. Başka bir group onu okuyabiliyorsa, o group asla başarısız olmadığı bir mesajı yeniden işler.
- **Kimliği koruyun.** Yeniden publish edilen mesaj orijinal key'i, `event_id`'yi ve iş header'larını korur. Yeni bir key ordering'i bozar, yeni bir `event_id` inbox'ı bozar.
- **Retention'ı beklemeye göre boyutlandırın.** Bir retry topic'indeki retention, en uzun beklemeyi artı üzerinde beklediğiniz lag'i aşmalıdır; yoksa mesaj beklerken expire olur.

En önemli maliyet, sonraki bölümün ele aldığı **ordering**'dir.

Katmanları seçmek

Pratikte bir consumer üçünü de kullanır. Kısa bir uygulama seviyesi retry'ı, bir iki saniye içinde ortadan kalkan connection reset ya da deadlock gibi aksamaları emer. Bir retry topic ya da delay queue, dakikalar gerektiren hataları kapsar. Broker'ın redelivery'si crash'leri kapsar. Bir dead-letter queue (DLQ) zinciri sonlandırır; attempt'leri tükenmiş ya da hiç başarılı olamayacak olanı alır; tasarımını Bölüm 8 ele alıyor.

Katmanlar birlikte, üç bağımsız değil tek bir toplam attempt budget'ıyla tasarlanmalıdır. Her katman üç attempt'e izin veriyorsa, bir mesaj 27 kez denenebilir ve bunu kimse planlamamıştır. Bu bölümün sonundaki policy çarpımı açık hâle getirir ve bir tavan koyar.

Retry'lar ve ordering

Bir mesajı retry etmek, mesajların işlenme sırasını değiştirir. Bu, yukarıdaki katmanlamadan çıkar ve hasar verene kadar fark edilmez.

Aynı sipariş için önce `OrderPlaced`'i, sonra `OrderCancelled`'i bu sırayla alan `Parcelly`'nin `Inventory consumer`'ını düşünün. `OrderPlaced`'in işlenmesinin `transient` olarak başarısız olduğunu ve `consumer`'ın onu bir `retry topic`'e taşıyıp `OrderCancelled`'i işlemeye devam ettiğini varsayın. İptal, serbest bırakacak bir rezervasyon bulamaz, çünkü `Inventory`'nin bildiği kadarıyla henüz hiçbirisi yoktur. Sonra `retry` edilen `OrderPlaced` gelir ve hiçbir şeyin serbest bırakmayacağı bir rezervasyon yaratır. Nihai state yanlıştır ve her bireysel adım makul görünmüştür.

İki dürüst seçenek var ve her birinin bir bedeli var.

- **Bloke et ve yerinde retry et.** Log tabanlı bir broker'da `consumer`, başarısız olan `offset`'e geri seek eder ya da `partition`'ı `pause` eder ve arkasındaki hiçbir şeye dokunmadan önce onu `backoff` ile `retry` eder. O `partition` için sıra korunur. Bedeli, tüm `partition`'ın tek bir mesajın arkasında beklemesidir; dolayısıyla tek bir başarısız mesaj o `partition`'daki her `aggregate`'i durdurabilir. Sıranın `throughput`'tan daha önemli olduğu durumda doğru seçim budur ve bir kalıcı hatayı hızla sınıflandırmanın neden bu kadar önemli olduğu da budur: bir `partition`'ın başındaki bir `poison message`, arkasındaki her şeyi bloke eder.
- **Kenara al ve reordering'i kabul et.** Başarısız mesaj bir `retry topic`'e gider ve `consumer` devam eder. `Throughput` korunur. Bedeli, `consumer`'ın sıra dışı geliş, `version` kontrolleri gibi tekniklerle ya da aynı `aggregate`'in sonraki her mesajını da öncekisi başarılı olana kadar park ederek dayanması gerektiğidir. Bölüm 9 bu teknikleri ele alıyor.

`Stream` başına seçerdim. Bir `aggregate`'in mesajlarının sırayla uygulanması gereken bir `event stream`'i için yerinde `retry` eder; hızlı sınıflandırmaya ve bir `dead-letter` çıkışına yatırım yapardım. Mesajlar arasındaki sıranın önemli olmadığı, bağımsız olgulardan oluşan bir `stream` için kenara almak ucuzdur ve akışı hareketli tutar.

Gruplama özelliği olmayan `queue` tabanlı broker'larda, `competing consumer`'lar arasındaki `ordering` baştan zayıftır; dolayısıyla `retry`'lar bir garantiyi bozmaktan çok mevcut bir özelliği biraz daha kötüleştirir. Yine de sırayı varsayan bir `consumer`, onu sağlaması için bir `queue`'ya güvenmemelidir. Bir gruplama özelliği kullanıyorsanız (`FIFO queue`'lar,

message group'lar, session'lar), trade-off bir log'dakiyle aynıdır: başarısız bir mesaj, yerinde bir retry'ın bir partition'ı bloke etmesi gibi, başarılı olana ya da dead-letter'a gönderilene kadar group'unu bloke eder.

Idempotency olmadan retry'lar

Her retry, daha önceki sonucu bilinmeyebilecek bir işe yapılan tekrarlanmış bir denemedir. Önceki attempt hiçbir şey yapmadan başarısız olduysa, retry zararsızdır. İş yaptıktan sonra başarısız olduysa, retry işi iki kez yapar.

Bu nadir bir uç durum değil. Açılış senaryosunu düşünün. Payments, 4.990 minor unit EUR'u capture etmek için sağlayıcıyı çağırıyor. Sağlayıcı parayı capture ediyor ve response'unu gönderiyor, response kayboluyor. Payments bir timeout görüyor. Timeout bilinmeyen olarak sınıflandırılıyor ve policy retry etmeyi söylüyor. İkinci capture request'i çıkıyor. Alıcıdan bir kez mi iki kez mi tahsilat yapılacağı artık tamamen, bir şeyin ikinci request'i bir tekrar olarak tanınır kılıp kılmadığına bağlı.

O şey bir **idempotency key**'dir: request ile birlikte gönderilen, alıcının bu işi daha önce gördüğünü tanımasını sağlayan sabit bir tanımlayıcı. Sağlayıcı çağrısı için, aynı capture'ın her attempt'inde aynı key'dir; böyle key'leri destekleyen bir sağlayıcı bunu tek bir capture sayar. Event'leri consume etmek için normalde `event_id`'dir ve işle aynı transaction içinde inbox'a karşı kontrol edilir. Bölüm 7 ikisini de kuruyor.

Uyarı: Bir retry policy'si yalnızca idempotent bir operasyonun üzerinde güvenlidir. Kart tahsil etmek, e-posta göndermek ya da stok rezerve etmek gibi etkisi olan herhangi bir çağrıda retry'ları etkinleştirmeden önce, ikinci attempt'i neyin zararsız kıldığını belirleyin. Hiçbir şey kılmiyorsa, otomatik retry'ların doğru sayısı sıfırdır ve hata bir insana gitmelidir. At-least-once bir broker, handler ne isterse istesin bir crash ya da timeout'tan sonra yeniden teslim edeceği için, "sıfır"ı handler'ın uygulaması gerekir: bu mesajı daha önce gördüğünü tespit etmeli ya da etkiden önce ack edip at-most-once'ı kabul etmeli ve ardından hatayı bir insana yönlendirmelidir.

Bağımlılık iki yönlüdür. Idempotency retry'ları güvenli kılan şeydir ve retry'lar idempotency'yi gerekli kılan şeydir. Idempotent consumer'lar olmadan agresif retry eden bir sistem, hata oranıyla orantılı bir hızda duplicate etkiler üretir; yani sistemin zaten baskı altında olduğu ve kimsenin temizlik yapacak vakti olmadığı anlarda.

Daha ince bir durum da var. Bir operasyon aynı key için idempotent olup yine de key değiştiği için retry etmek açısından güvensiz olabilir. Bir handler her attempt'te yeni bir tanımlayıcı üretiyorsa, örneğin her çağrıda yeni bir UUID, sağlayıcı her seferinde ayrı request'ler görür. Key, `event_id` ya da sipariş gibi sabit girdilerden bir kez türetilmeli ve her attempt'te yeniden kullanılmalıdır.

Parcellary için bir retry policy'si

Parçaları bir araya getirmek için, Payments'ın `CapturePayment` consumer'ı için yazacağım şey şu. Sayılar örnektir, öneri değil ve sizin dependency'nizin kendi sayılarına ihtiyacı olacak.

```
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
```

Bunu, her birinin bir gerekçesi olan bir karar kümesi olarak okuyun. Handler içi katman kısa aksamaları emer. Retry topic'ler, dakikalarca kapalı kalan bir sağlayıcıyı kapsar. Deadline, işin ne zaman yapmaya değmez hâle geldiğini söyler ve bu kararın sahibi mühendis değil, iş alanıdır. Bayat bir capture bir `OrderCancelled`'a dönüşmeyi gerektirebilir, ama compensation sağlayıcıda gerçekte ne olduğuna bağlıdır. Son attempt timeout'a uğradıysa capture başarılı olmuş olabilir ve bu durumda doğru devam adımı bir `PaymentVoided` değil, bir `PaymentRefunded`'dir. Policy'nin önce durumu sormasının nedeni budur. Bir kez türetilen key, tüm yapının bir alıcıdan iki kez tahsilat yapmasını engelleyen şeydir.

Policy'deki iki satır ikinci bir bakışı hak ediyor. `maxTotalAttempts` vardır, çünkü katmanlar çarpılır: üç handler içi attempt, çarpı bir ilk geçiş artı üç retry topic adımı, bir crash'ten sonraki herhangi bir redelivery sayılmadan önce, sağlayıcıya yapılan 12'ye varan çağrıdır. 8'lik tek bir toplam (örnektir) birinin gerçekten sahip çıktığı sayıdır. Ve tüm retry ufku, burada iki saat, sağlayıcının idempotency key'leri hatırladığı süreden kısa olmalıdır. Sağlayıcılar key'leri sınırlı bir pencere boyunca tutar ve pencereden sonra gelen bir retry, eski bir key taşıyan yeni bir request'tir.

Bölüm 8'in redrive'ı ölü bir capture'ı geri koyduğunda attempt sayacı yeniden başlar, ama ayrı bir redrive sayacı kalır; böylece geçmiş kaybolmaz ve sürekli geri gelen bir mesaj, öyle biri olarak görünür.

Neyin eksik olduğuna dikkat edin: tek bir "retry sayısı" ayarı. Katmanlı bir sistemde bu tek sayı en az dört kararı gizler.

Trade-off'lar ve maliyetler

Retry'lar, bir sistemin görünür güvenilirliğini artırmanın en ucuz yollarından biridir ve en kolay aşırı kullanılanlardan biridir. Maliyetleri belirgindir.

Latency. Her retry bekleme süresi ekler. Backoff ve jitter yükü yayar ama başarıya kadar geçen süreyi de uzatır. Dakikalarca retry eden bir akış doğru biçimde tamamlanabilir ve yine de 10 saniye sonra ayrılan alıcı için işe yaramaz olabilir. Bir insanın beklediği yerde, uzun sessiz bir retry yerine hızlı bir başarısızlığı ve net bir mesajı tercih ederdim.

Amplification. Retry'lar yükü çarpar ve katmanlı retry'lar onu yeniden çarpar. Her serviste makul olan bir policy zincirde tehlikeli olabilir. Birinin uçtan uca attempt budget'ının sahibi olması gerekir, çünkü hiçbir tek servis onu göremez.

Kapasite. Yerinde retry eden bir consumer, beklerken slot'unu işgal eder. Uzun bir kesintide tüm consumer'lar beklemeye geçebilir ve sistem, başarılı olacak işte bile ilerlemeyi bırakır. Bir breaker aracılığıyla tüketimi pause etmek, uçustaki işi backpressure ile sınırlamak ve uzun beklemleri retry topic'lere taşımak olağan savunmalardır.

Gizli hatalar. Eninde sonunda başarılı olan bir retry dışarıdan bir başarı gibi görünür. Retry'ları ve attempt'leri saymazsanız, bir dependency haftalarca, onu sessizce emen bir policy'nin arkasında bozulabilir. Bölüm 13 neyin ölçüleceğini ele alıyor: attempt sayıları, retry oranı ve en eski retry edilmiş mesajın yaşı bir dashboard'da yer almalı.

Karmaşıklık. Retry topic'ler, delay queue'lar ve budget'lar, birinin kurması, adlandırması, monitor etmesi ve açıklaması gereken altyapıdır. Tek bir consumer'ı ve tek bir dependency'si olan küçük bir sistem için backoff'lu bir handler içi retry ve bir dead-letter çıkışı ihtiyacınız olan her şey olabilir ve bir retry topic merdiveni, problemin hak ettiğinden fazla mekanizma olabilir.

Ne zaman retry etmemeli. Kalıcı hataları retry etmeyin. Idempotent olmayan, etkisi olan operasyonları retry etmeyin. Deadline'ın ötesinde retry etmeyin. Bir circuit breaker dependency'nin kapalı olduğunu söylerken retry etmeyin ve o mesaja bunun için bir attempt yazmayın. Zaten başka bir katmanın retry'ının içinde duran bir katmanda, çarpımı hesaplamadıysanız retry etmeyin. Ve bir karardan kaçınmanın yolu olarak retry etmeyin: sürekli geri gelen bir hata bilgidir ve doğru cevap çoğu zaman durup bakmaktır.

Sık yapılan hatalar

Her şeyi aynı şekilde retry etmek

Tek bir döngü en kısa kod olduğu için cazip gelir. Kart reddini connection reset gibi ele alır ve başarılı olamayacak hatalara kapasite harcar. Önce sınıflandırın ve her sınıfa kendi ele alış biçimini verin.

Jitter'sız sabit gecikmeler

Sabit bir bekleme öngörülebilir ve basit hissettirir. Birlikte başarısız olan her çağırını tekrarlayan bir sıçramaya senkronize eder. Jitter'lı exponential backoff kullanın ve bence başlangıç noktası olarak full jitter.

Attempt'leri saymak ama zamanı saymamak

Bir attempt limiti güvenlik sınırı gibi görünür. Uzun timeout'lar ve büyüyen gecikmelerle yine de yararlılığın dakikalarca ötesine uzanabilir. Tüm operasyon için bir deadline belirleyin ve bir sonraki gecikme onu aşacağında durun.

Her katmanda retry etmek

Her katmanın sahibi makul bir retry ekler ve çarpım hiçbirine görünmez. Zincir için tek bir toplam attempt budget'ına karar verin ve retry'ları en az maliyetli oldukları yere koyun.

Breaker open iken retry etmek

Döngü çalışıyordu diye döngüyü sürdürmek cazip gelir. Ölü bir dependency'ye karşı mesaj çeken ve onları başarısız kılan bir consumer, mesajların attempt'lerini tüketir ve onları yanlışlıkla dead-letter'a gönderebilir. Open bir breaker'ı attempt'e mal olmayan bir deferred sinyali olarak ele alın, tüketimi pause edin ve mesajların broker'da beklemesine izin verin.

Bir consumer'ın limitsiz çekmesine izin vermek

Sınırsız consumer varsayılandır ve maksimum throughput gibi hissettirir. Yavaş bir dependency'ye karşı her slot'u bekleyen işle doldurur ve retry storm'u besler. Uçuştaki mesajları sınırlayın, latency yükseldiğinde concurrency'yi düşürün ve ilk attempt'lerin retry'lara karşı kazanmasına izin verin.

Broker'ın timeout'larını göz ardı etmek

Handler içi retry'lar handler'a yerel hissettirir. Visibility timeout'u ya da group'un liveness penceresini aşarlarsa, broker yeniden teslim eder ya da yeniden atar ve işin iki kopyası aynı anda çalışır. Toplam handler içi süreyi broker'ın limitlerinin çok içinde tutun.

Sıradaki kalması gereken bir mesajın yanından retry etmek

Başarısız bir mesajı retry topic'e taşımak, durmuş bir partition için hızlı bir çözümdür. O aggregate'in event'lerini reorder eder. Önce ordering gereksinimine karar verin, sonra bloke etmek ile kenara almak arasında seçim yapın.

Idempotent olmayan etkileri retry etmek

Retry döngüsünü eklemek bir öğleden sonra sürer ve idempotency ertelenir. Başarılı bir çağrıdan sonraki ilk timeout bir duplicate etki üretir. İlk retry etkinleştirilmeden önce, ikinci attempt'i neyin zararsız kıldığını belirleyin.

Her attempt'te yeni bir key üretmek

Her request benzersiz biçimde tanımlandığı için özen gibi görünür. Amacı boşa çıkarır, çünkü alıcı bir tekrarı tanıyamaz. Key'i sabit girdilerden bir kez türetin ve yeniden kullanın.

Retry'ları saymamak

Yalnızca nihai hatalar loglanırsa, bozulan bir dependency retry'lar yetersiz kalana kadar görünmez kalır. Her attempt'i, sınıfı ve gecikmesiyle kaydedin.

Temel çıkarımlar

- Retry, yeniden deneme sözüdür, başarı sözü değil; bu yüzden herhangi bir hatada ilk soru, yeniden denemenin yardımcı olup olamayacağıdır.
- Hataları transient, kalıcı ya da bilinmeyen olarak sınıflandırın, sınıflandırmayı bir tablo olarak yazın ve tanınmayan hataları daha sıkı bir limite bilinmeyen sayın.
- Jitter'lı exponential backoff kullanın, tüm operasyonu bir deadline ile sınırlayın ve retry'ların başarısız bir dependency üzerindeki baskın yük olmaması için bir retry budget paylaşın.
- Bir circuit breaker çağrı yapıp yapmayacağına karar verir; open bir breaker attempt'e mal olmayan bir deferred sinyalıdır ve bir consumer için en güçlü biçimi tüketimi pause etmektir. Backpressure aynı işi

daha erken yapar: uçustaki işi sınırlar ve canlı trafiğin retry'lara ve redrive'a karşı kazanmasına izin verir.

- Retry'lar handler'da, broker redelivery'sinde ve retry topic'lerde yaşayabilir ve tek bir attempt budget'ı altında birlikte tasarlanmalıdır; handler içi süre broker'ın timeout'larının içinde tutulmalıdır.
- Başarısız bir mesajı kenara almak, aggregate'inin event'lerini reorder eder; yerinde bloke etmek ile reordering'e dayanmak arasında stream'in ordering gereksinimine göre seçim yapın.
- Bir retry policy'si yalnızca idempotent operasyonların üzerinde güvenlidir; idempotency key bir kez türetilir ve her attempt'te yeniden kullanılır.

Retry'lar ne sıklıkta yeniden deneyeceğinize karar verir, ama iki kez denemenin zararsız olup olmadığına karar veremez. Bölüm 7 diğer yarıyı kuruyor: bir duplicate'i felaketten bir non-event'e çeviren idempotent consumer.

Bölüm 7 — Idempotency

Parcelly'nin Ledger servisini, consumer'ları tek tek yeniden başlatılırken bir rolling deploy'un ortasında düşünün. Bir consumer bir `PaymentCaptured` event'i okudu, 4.990 minor unit EUR'luk bir ledger kaydı yazdı ve mesajı ack etmek üzereyken süreç kapanma sinyali alıyor. Ack makineden hiç çıkmıyor. Broker, at-least-once delivery'nin vaat ettiği şeyi tam olarak yaparak aynı mesajı onu isteyen bir sonraki consumer'a verir.

İkinci consumer event'i okur, olağandışı bir şey görmez ve ikinci bir ledger kaydı yazar. Ledger append-only'dir ve yapısı gereği doğrudur; dolayısıyla alıcının iki kez ödeme yaptığını sadakatle ve kalıcı olarak kaydeder. Hiçbir şey çıkmadı. Hiçbir alert tetiklenmedi. Her bileşen tasarlandığı gibi davrandı ve defterler yanlış.

Notifications'ın da aynı öğleden sonranın kendi versiyonu var. Yavaş bir handler visibility timeout'unu aştığı için mesaj yeniden teslim edilir ve bir alıcı "kargonuz planlandı" e-postasını üç kez alır. Bu, utanç verici bir durum. Ledger'daki ise pahalı.

Önceki bölümler duplicate'lerin normal olduğunu ortaya koydu: broker yeniden teslim eder, relay bir failover'dan sonra iki kez publish eder ve Bölüm 6'daki retry'lar tasarım gereği duplicate'tir. Bu bölüm, o bölümlerin açık bıraktığı soruyu yanıtlıyor. Aynı mesaj iki kez geldiğinde, ikinci gelişin hiçbir şeyi değiştirmemesi için bir consumer tam olarak ne yapmalı?

Problem ve neden zor olduğu

Hedef mütevazı görünüyor: her mesajı, dünyanın görebildiği kadarıyla, bir kez işlemek. Zorluk şurada: consumer, hatırlamadan bir mesajı daha önce görüp görmediğini bilemez ve hatırlamak da yanlış anda başarısız olabilecek bir yazma işlemidir.

Duplicate'ların nereden geldiğiyle başlayın, çünkü liste broker'dan daha uzun.

Kaynak	Duplicate nasıl ortaya çıkar
Broker redelivery	Ack kayboldu, visibility timeout doldu ya da bir rebalance, offset commit edilmeden partition'ı taşıdı
Relay	Bir relay bir satırı publish etti, published olarak işaretlemeyen çıktı ve yeniden başladıktan sonra tekrar publish etti (Bölüm 5)
Uygulama retry'ı	Bir consumer bir çağrıyı tekrarladı ya da bir producer, ilk denemesi aslında başarılı olmuş bir publish'i tekrarladı (Bölüm 6)
Redrive ve replay	Biri dead letter'ları geri taşıdı ya da bir event log'unu daha eski bir offset'ten bilerek yeniden okudu
İnsanlar ve client'lar	Bir alıcı "Öde" düğmesine çift tıkladı, bir mobil uygulama kararsız bir ağda retry yaptı, bir webhook gönderen iki kez teslim etti

Bunların çoğu *aynı* mesajı, aynı `event_id` ile iki kez üretir. Birkaçı ise aynı niyeti ifade eden iki farklı mesaj üretir ve bu ayırım, bir key seçerken önem kazanacak.

Cazip çözüm, harekete geçmeden önce bakmaktır. Bu mesajın işlenip işlenmediğini sor, işlenmediyse işle. Bunu **check-then-act** anti-pattern'i olarak adlandırmak istiyorum, çünkü doğru hissettiriyor ve sessizce başarısız oluyor.

```
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)
```

Bunu hata altında okuyun. Süreç `postLedgerEntry`'den sonra ve `inboxInsert`'ten önce ölürse kayıt vardır ama inbox bunu bilmez; dolayısıyla redelivery kaydı yeniden post eder. İki yazmanın sırasını değiştirin, hata da tersine döner: `inboxInsert`'ten sonra ve ledger yazımından önce gelen bir çökme, işlenmemiş mesajı işlenmiş olarak işaretler ve etki sonsuza dek kaybolur. Hiçbir sıra güvenli değildir, çünkü iki yazma tek bir atomik eylem değildir.

Çökmelerle ilgisi olmayan ikinci bir hata daha var. İki consumer aynı mesajı hemen hemen aynı anda alabilir; örneğin yavaş bir ilk deneme visibility timeout'unu aştığında. İkisi de kontrolü çalıştırır, ikisi de hiçbir şey bulmaz ve ikisi de devam eder. "Check" ile "act" arasındaki pencere küçüktür ve yeterli hacimde rutin olarak isabet alır.

Dolayısıyla gerçek problemin iki yarısı var. Nelerin işlendiğinin hafızası etkiyle atomik olarak yazılmalı ve bu hafıza eşzamanlı denemeler arasında hakemlik yapabilmeli. Bölümün geri kalanı ikisini de kuruyor; ikisine de ihtiyaç duymadığınız durumla başlıyor.

Doğal ve zorlama idempotency

Bir operasyon, iki kez uygulanması sistemi bir kez uygulanmasıyla aynı durumda bıraktığında **idempotent**dir. Bazı operasyonlar biçimleri gereği idempotenttir ve herhangi bir mekanizma kurmadan önce bunları aramayı tercih ederim.

Bir değer set etmek idempotenttir: `set order.status = "PAID"` 10 kez çalışabilir ve sipariş yine `PAID`'dir. Bir satırı id'siyle silmek idempotenttir. Primary key'i event'ten türettiğiniz bir satır insert edip çakışmayı yok saymak idempotenttir. Bir aggregate'in tam yeni durumunu yazmak ("sevkiyat artık `DISPATCHED`, şu takip numarasıyla") idempotenttir. Buna **doğal idempotency** diyorum: duplicate'in fazladan bir etkisi olmaz; bunun nedeni operasyonun ne olduğudur, sizin eklediğiniz bir şey değil.

Diğer operasyonlar değildir. `balance = balance + 4990` iki kez uygulanınca iki para yatırma olur. `send_email` iki e-posta gönderir. `append a ledger entry` iki kayıt ekler. `increment the counter of units sold` birimi iki kez sayar. Bir ödeme sağlayıcısının `capture endpoint`'ini key olmadan çağırmak iki kez tahsilat yapabilir. Bunlar için idempotency'nin eklenmesi gerekir; ben buna **zorlama idempotency** diyorum: işin yapıldığını kaydedersiniz ve işi yeniden yapmadan önce kayda bakarsınız.

Doğal idempotency'nin iki dürüst sınırı var. Birincisi, sessizce sıraya bağımlı olmasıdır. `status = "PAID"` set etmek idempotenttir, ama daha eski bir event'in gecikmiş bir duplicate'i daha yeni bir status'un üzerine yazabilir: `OrderConfirmed`'dan sonra gelen yeniden teslim edilmiş bir `OrderPlaced`, status'u tekrar `PLACED`'e çeker. Operasyon idempotenttir ve

sistem yine de yanlıştır. Bölüm 9 ordering'i ve version kontrollerini ele alıyor; şimdilik şuna dikkat edin: "bir değere set etmek", yalnızca değer stale olamıyorsa duplicate'lere karşı güvenlidir.

İkinci sınır, olduğundan fazla tahmin etmenin cazip olmasıdır. "Yalnızca bir satırı güncelleyen" bir handler çoğu zaman bir audit kaydı da yazar, bir takip event'i yayınlar ya da bir sayacı artırır ve bunların her biri kılık değiştirmiş bir increment'tir. Bir consumer'ı gözden geçirirken tüm etkilerini listeler ve her biri için ayrı ayrı ikinci bir çalışmadan sağ çıkıp çıkmadığını sorarım. Tek bir idempotent olmayan etki bile inbox'a ihtiyaç duymak için yeterlidir.

Pratik kuralım, etki izin verdiği yerde doğal idempotency'yi tercih etmek ve handler'a bunun nedenini yazmaktır. Etki bir sayaç, bir append, bir e-posta ya da harici bir çağrıysa zorlama türünü kullanın. Şüphede kalırsanız yine de inbox'ı kullanın: maliyeti küçük ve öngörülebilir, yanılmanın maliyeti ise değil.

Idempotency key'i seçmek

Idempotency key, consumer'ın tekrarlanan işi tanımak için kullandığı şeydir. Tabloyu kurmadan önce içine neyin gireceğine karar vermeniz gerekir ve bu seçim hangi duplicate'leri yakalayabileceğinizi belirler.

event_id

Event tüketimi için varsayılan, envelope'taki `event_id`'dir. Event başına benzersizdir ve en önemlisi **redelivery boyunca sabittir**: broker'ın size verdiği mesajın her kopyası aynı değeri taşır. Producer onu, event outbox'a yazılırken bir kez atar (Bölüm 4); dolayısıyla iki kez publish eden bir relay aynı `event_id`'yi iki kez publish eder. Bir dedup kontrolünün tam olarak ihtiyaç duyduğu özellik budur.

`event_id` bir *mesajı* tanımlar. "Bu belirli olguyu daha önce ele aldım mı?" sorusuna yanıt verir ve bundan geniş bir şeye değil.

Business key

Bazen iki farklı mesaj aynı niyeti tarif eder. Orders'ın bir timeout'tan sonra bir `CapturePayment` command'ını retry ettiğini düşünün; bir bug ya da bir tasarım tercihi yüzünden her retry yeni bir `command_id` ile yeni bir

command olarak yazılıyor. Payments artık, ikisi de "81724 numaralı sipariş için ödemeyi capture et" anlamına gelen, farklı id'li iki command alıyor. Mesaj id'sine göre deduplication ikisini de yakalamaz.

Çare bir **business key**'dir: zarftan değil, işin anlamından türetilen bir tanımlayıcı. Capture için `payment_id` artı operasyon olabilir, örneğin `capture:payment-5521`. Bir ledger kaydı için ödeme id'si artı kayıt türü olabilir. Bir business key, kaç mesaj istediğine bakılmaksızın "bu iş birimi yalnızca bir kez gerçekleşebilir" der.

Boyut	event_id / command_id	Business key
Aynı mesajın redelivery'sini yakalar	Evet	Evet
Aynı niyet için iki mesajı yakalar	Hayır	Evet
Kim atar	Producer, otomatik olarak	Operasyon başına bir insan kararı
Risk	Mantıksal duplicate'leri kaçırır	Fazla kaba bir key meşru işi reddeder
Nerede saklanır	Genel inbox tablosu	Çoğu zaman domain tablosunda bir unique constraint

Sondan ikinci satırdaki risk somut bir örneği hak ediyor. Refund'lar için business key yalnızca `payment_id` ise, aynı ödemenin ikinci, meşru bir kısmı refund'ı duplicate olarak reddedilir. Key, iki isteği *gerçekten farklı* kılan her şeyi içermelidir: burada belki çağırının sağladığı bir refund idempotency key'i. Key'i seçmek, domain'inizde "aynı iş" in ne demek olduğuna karar vermektir; bu bir tasarım kararıdır, bir framework ayarı değil.

Bana göre katmanlar birlikte iyi çalışır. Her consumer için temel olarak genel bir inbox'ta `event_id`'yi kullanın ve iki farklı mesajın makul biçimde aynı şeyi ifade edebileceği her yerde domain tablosuna bir business-key unique constraint'i ekleyin. Inbox sizi broker'a ve relay'e karşı korur. Constraint sizi sistemin geri kalanına, bug'ları dahil, karşı korur.

Inbox pattern'i

Inbox, consumer tarafındaki işlenmiş mesaj id'lerinin tablosudur. Bölüm 4, producer'a bir state değişikliği ve event'inin birlikte commit edilmesi için bir transactional outbox verdi. Inbox onun ayna görüntüsüdür: consumer, state değişikliğini ve mesajı ele aldığıнын kaydını birlikte, kendi veritabanındaki tek bir local transaction içinde commit eder.

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

`consumer` kolonu, bir servisin tek bir tabloya karşı birkaç bağımsız consumer çalıştırmasına izin verir; böylece Ledger'ın posting consumer'ı ile reporting consumer'ı kendi geçmişlerini ayrı ayrı hatırlar. Primary key'in (`consumer, event_id`) çifti olmasının nedeni budur. Yalnızca `event_id` olsaydı, bir mesajı ele alan ilk consumer, ikinci consumer'ın insert'ünün başarısız olmasına yol açardı ve ikinci consumer'ın etkisi duplicate'miş gibi atlanırdı. Tablo başına tek consumer'ı olan bir servis kolonu bırakıp `event_id` üzerinden key'leyebilir; bu, kitabın başka yerlerinde kullanılan daha basit biçimdir.

Handler, inbox satırını ve business değişikliğini aynı transaction içinde yazar.

```
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)
```

Hataları adım adım izleyin. Süreç commit'ten önce ölürse ne ledger kaydı ne de inbox satırı vardır; dolayısıyla redelivery mesajı baştan işler. Commit'ten sonra ve ack'ten önce ölürse ikisi de vardır ve redelivery fast path'i izler: kontrol satırı bulur, mesaj ack edilir ve hiçbir şey yazılmaz. Etki ile hafızası asla uyuşmazlık yaşayamaz, çünkü tek bir bütün olarak commit edilirler.

En üstteki kontrol bir optimizasyondur. Yaygın duplicate durumunda bir transaction'dan tasarruf sağlar ve doğruluktan bir şey kaybetmeden kaldırılabilir. Güvenlik ağı, transaction içindeki unique constraint'tir; gerçek hakemlik orada yapılır.

Bir transaction'ın içinde yazmaların sırası atomikliği değiştirmez; dolayısıyla iki yazmayı ters çevirmeye dair uyarı yalnızca yukarıdaki güvensiz sürüm için geçerlidir. Sıra maliyeti değiştirir. Bu versiyonda eşzamanlı bir duplicate, insert'te başarısız olmadan önce bütün işi yapar ve bazı veritabanlarında bir violation transaction'ın tamamını abort eder; yani handler transaction'ın içinde toparlanamaz. Ben önce sahiplenirdim: inbox satırını "on conflict do nothing" semantiğiyle insert edin, bir satırın insert edilip edilmediğine bakın ve edilmediyse etkiyi atlayın. O zaman duplicate key üzerinde bekler, hiçbir iş yapmaz ve hiçbir zaman hata fırlatmaz. Sonraki kısmın örneği bu biçimi kullanıyor.

ack üzerine bir not, çünkü bu terim iki aile arasındaki bir farkı gizliyor. Queue tabanlı bir broker'da tek bir mesajı acknowledge eder. Log tabanlı bir broker'da ise genellikle bir offset'in commit edilmesidir; bu da "buraya kadar her şey" anlamına gelir ve bir partition'ı paralel işleyen bir consumer, bitirmediği bir mesajın ötesine commit edip onu kaybedebilir. Inbox duplicate'lara karşı korur, buna karşı değil; Bölüm 9 bunu ordering ile paralelliğin bulunduğu yerde ele alıyor.

Bir consumer'dan çıkan takip event'leri

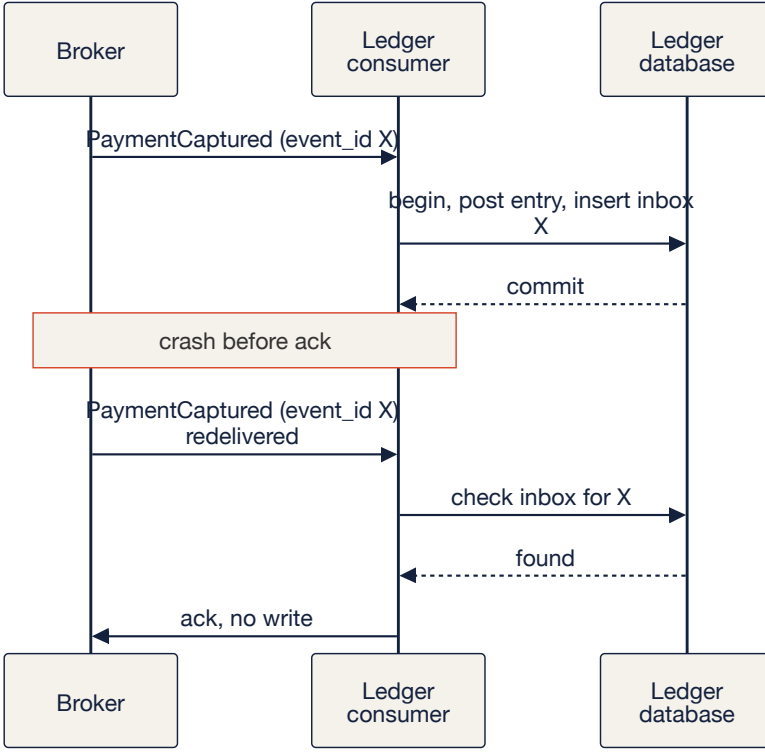
Birçok consumer kendi state'ini değiştirmekle kalmaz: ne yaptığını duyurur. Ledger bir kayıt işler ve LedgerEntryPosted yayınlar. Bu takip event'i de diğer her etki gibi bir etkidir ve bir çökmeden tam olarak bir kez sağ çıkmalıdır; ne sıfır kez ne de iki kez.

Takip event'ini commit'inin hemen ardından doğrudan yayınlayan bir handler, çökmede onu kalıcı olarak kaybeder. Redelivery inbox fast path'ini izler, satırı bulur, ack eder ve publish'i atlar. State doğrudur ve event hiç var olmamıştır. Çözüm, Bölüm 4'teki outbox'ın tüketen tarafa uygulanmasıdır: inbox satırı, state değişikliği ve outbox insert'ü tek bir local transaction içinde commit edilir.

```
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)
```

Süreç commit'ten önce ölürse hiçbir şey yoktur ve redelivery hepsini yapar. Commit'ten sonra ölürse takip event'i outbox'ta durur ve relay onu yayınlar; redelivery ise inbox satırını bulur ve hiçbir şey yapmaz. Consumer'ın outbox'ı, producer'ınkiyle aynı at-least-once publication'a sahiptir; dolayısıyla `LedgerEntryPosted`'i tüketen kişinin kendi inbox'ına ihtiyacı vardır. Bölüm 10, bir saga adımı bir event'e tepki verip bir sonrakini yayınladığında bu kural üzerine kurulur.

Inbox'ın, koruduğu state ile *aynı veritabanında* yaşamasının nedeni de budur. İşlenmiş mesaj kaydı bir cache'te ya da başka bir veritabanında dursaydı, ortak bir transaction'ı olmayan iki yazmaya, yani Bölüm 4'teki dual write'in consumer tarafındaki haline dönmüş olurduk. Inbox, servisin veri deposunun dışından üzerine eklenebilecek bir teknik değildir.



Ortakdaki nota bakın: çökme penceresi commit ile ack arasındadır ve redelivery'yi no-op'a çeviren şey inbox satırıdır.

Eşzamanlı duplicate'ler

Yukarıdaki sequence, duplicate'in ilk deneme bittikten sonra geldiğini varsayıyor. Eşzamanlılık daha zordur ve sizi kurtaran şey unique constraint'tir.

İki consumer'ın aynı mesajı aynı anda işlediğini gözünüzün önüne getirin. İkisi de bir transaction başlatır. İkisi de işi yapar. İkisi de inbox satırını insert etmeye çalışır. Veritabanı yalnızca birinin commit etmesine izin verir; diğeri bir unique-constraint violation alır, transaction'ı rollback olur ve ledger kaydı onunla birlikte yok olur. Kaybeden sonra mesajı ack etmeden bırakır ya da yeniden kontrol ettikten sonra ack eder; her iki durumda da ledger'da tek bir kayıt vardır.

Mekanizmanın tamamı bu ve neyi gerektirmediğini söylemeye değer: dağıtık lock yok, iki consumer arasında koordinasyon yok, zamanlama hakkında varsayım yok. Veritabanının constraint'i, her iki denemenin de geçmek zorunda olduğu tek bir hakemlik noktasıdır.

```
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
```

Bir unique violation'ın ack mi edilmesi yoksa retry mı edilmesi gerektiği, bilinçli verilmeye değer küçük bir karardır. Violation inbox key'indeyse, mesaj commit edilmiş bir transaction tarafından ele alınmıştır ve ack etmek doğrudur. `applyChange` içindeki başka bir constraint'teyse hata bambaşka bir anlama gelir ve ack etmek gerçek bir bug'u gizler. Bana göre handler bunları ayırt etmelidir; bunu, yukarıdaki kodun yaptığı gibi, ihlal edilen constraint'in adını kontrol ederek yapabilir.

Daha uzun süren handler'lar ve devam eden iş

Yukarıdaki pattern, handler'ın tamamını tek bir transaction içinde tutar. İş birkaç yazmadan ibaretse bu sorun değildir. Handler yavaş bir şeyi çağırıyorsa rahatsız edicidir, çünkü transaction açık kalır, lock'lar tutulur ve siz beklerken visibility timeout dolabilir. Süresi dolan mesaj ikinci bir consumer'a yeniden teslim edilir ve o da aynı yavaş işi paralel olarak başlatır.

İki sağlam yanıt var. Birincisi, transaction'ı kısa tutmaktır: yavaş harici işi transaction'ın *dışında* yapın, bir sonraki bölümdeki external-effects yaklaşımını kullanın ve transaction'ın içine yalnızca local state değişikliğini ve inbox satırını koyun. İkincisi, iş gerçekten sahiplenilmek zorundaysa, sahiplenmeyi bir status ile açık hale getirmektir.

```

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)

```

Bir consumer satırı bir lease ile `processing` olarak insert eder ve bu sahiplenme, yavaş iş başlamadan önce kendi kısa transaction'ında commit edilir. Canlı bir lease bulan bir duplicate mesajı ack etmemelidir, çünkü ilk deneme hâlâ başarısız olabilir ve acknowledgement tek retry'ı silerdi. Mesajı, kalan lease'ten daha uzun bir gecikmeyle yeniden kuyruğa alır ve bir sonraki teslimde tekrar bakar. Süresi dolmuş bir lease bulan, daha yeni bir lease token ile devralabilir ve orijinal sahip local değişikliklerini commit etmeden önce bu token'ı kontrol etmelidir; `done` bulan ise ack eder.

Token yalnızca onu kontrol edebilen şeyleri korur. Local transaction edebilir: inbox satırını lock'lar, token'ı karşılaştırır ve daha yenisi varsa rollback olur. Yavaş harici çağrı geri alınamaz; bu yüzden etki, sonraki bölümdeki deterministik key'i alır ve uyanıp çağrıyı tekrarlayan eski bir sahip downstream tarafından tanınır. Bu, Bölüm 5'teki relay lease'inden gelen fencing fikridir ve aynı sınıra sahiptir: bir fence, yalnızca korunan kaynak token'ı kontrol ettiği yerde çalışır.

Bu şema işe yarar ve birinin üzerinde düşünmesi gereken bir state machine ekler; bir worker lease'i elinde tutarken ölürse ne olacağı sorusu da buna dahildir. Buna ancak ilk yaklaşım mümkün olmadığında başvururum ve lease'i kısa ve yenilenebilir tutarım.

Uyarı: Duplicate'ları dışarıda tutmak için önceden yapılmış bir SELECT'e ya da bir row lock'a güvenmeyin. İkinci bir consumer'ın birincinin commit edilmemiş işini görüp görmediği veritabanının isolation level'ına bağlıdır ve bir seviyede doğru olan kontrol başka bir seviyede başarısız olur. Isolation level'ı serializable'a yükseltmek de constraint ihtiyacını ortadan kaldırmaz: sessiz bir duplicate'i bir serialization failure ile değiştirir; handler bunu retry edilebilir saymalı ve baştan yeniden çalıştırmalıdır. Lock'lar contention'ı azaltır; garanti, transaction içindeki unique constraint'tir.

Dedup window'lar ve retention

Her id'yi sonsuza dek tutan bir inbox sonsuza dek büyür. Bir noktada eski satırları silmek isteyeceksiniz ve silme, garantiyi bir ada layık bir biçimde değiştirir: **dedup window**. Inbox satırı silinmiş bir mesaj, yeniden ortaya çıkarsa tekrar işlenebilir.

Peki pencere ne kadar uzun olmalı? Bir duplicate'in orijinalden sonra makul biçimde gelebileceği en uzun süreyi kapsamalıdır. Bu sayıyı sizin sisteminiz için bilmiyorum ve kısa bir listedeki etkenlere bağlı.

- **Broker retention'ı ve redelivery ufku.** Dört gün boyunca yeniden teslim eden bir queue ya da bir hafta saklayan bir log, duplicate'lerin o kadar geç gelebileceği anlamına gelir.
- **Dead-letter ve redrive pratiği.** Bir ekip dead letter'ları başarısız olduktan bir hafta sonra redrive edebiliyorsa, pencere bunu aşmalıdır (Bölüm 8).
- **Replay.** Bir event log'unu eski bir offset'ten yeniden okumak, replay ettiğiniz kadar eski event'lerin duplicate'lerini üretir. Ya pencere yeterince uzundur ya da replay, bilerek sıfırlanmış bir consumer'a karşı çalıştırılır; bu, önceden tasarladığınız bir prosedürdür.
- **Relay failover'ı.** Bir failover'dan sonraki çift publish'ler genellikle orijinale zaman olarak yakındır; dolayısıyla seçeceğiniz herhangi bir pencerenin rahatça içinde kalır.

Yaygın bir biçim, broker'ın kendi retention'ından rahatça daha uzun bir saklama süresidir, örnek olarak birkaç hafta; batch'ler halinde silen zamanlanmış bir job ile birlikte. Inbox tablosunu zamana göre partition'lamak silmeyi ucuzlatır, çünkü bütün bir partition'ı drop etmek milyonlarca satırı silmekten çok daha az maliyetlidir.

Unutmayın, inbox tek savunma hattı değildir. Domain tablosundaki bir business-key constraint'i veri kadar yaşar; dolayısıyla inbox satırı gittikten sonra gelen duplicate'lere karşı korur. İki katmanı da kullanmanın bir diğer nedeni budur: inbox yaygın duruma karşı ucuz ve sınırlı hafızalı koruma sağlar, constraint ise kuyruğu kapsar.

Sistem bunu kabul edebiliyorsa, bir zaman penceresine alternatif vardır: key'i monotonik bir şeyden türetin. Kural ilk bakışta görüldüğünden daha dardır. Bölüm 9'un delta'lardan ayırdığı state taşıyan event'ler için

geçerlidir; yani her event'in bir aggregate'in tam yeni state'ini taşıdığı durumlar. Bunlarda consumer, uygulanan en yüksek aggregate version'ını saklar ve daha yeni olmayan her şeyi reddeder. Offset tek bir partition içindeki bir konumdur, version ise tek bir aggregate'in özelliğidir; dolayısıyla ikisi birbirinin yerine kullanılamaz: bir offset, başka partition'larda yaşayan aggregate'ler hakkında hiçbir şey söylemez.

Delta event'ler için "daha yeni değilse reddet" yanlıştır, çünkü atlanan bir version kaybolmuş bir değişiklik demektir. Böyle bir consumer `last + 1` version'ını uygular, daha sonraki bir version'ı boşluk dolana kadar buffer'lar ve `last`'a eşit ya da ondan küçük her şeyi duplicate sayar; Bölüm 9 buffer'lamayı ve sınırlarını anlatıyor. Her iki durumda da hafıza TTL'siz sınırlıdır ve Bölüm 9'un incelediği ordering garantilerine bağlıdır. Bir version sayacı olan aggregate'lere uyar, birkaç partition'dan beslenen consumer'lara uymaz.

Harici yan etkiler

Inbox, kendi veritabanınızda yaşayan etkileri korur. Birçok etki orada yaşamaz. Bir e-posta gönderdiğiniz anda sisteminizden çıkar ve ödeme sağlayıcısına yapılan bir çağrı, transaction'ınız sonradan commit etsin etmesin bir kartı tahsil eder. Hiçbir local transaction bunları geri alamaz; dolayısıyla atomiklik hilesi uygulanmaz.

Payments'ın bir `CapturePayment` command'ını ele aldığını düşünün. Harici sağlayıcıyı çağırmalı ve ardından sonucu kaydetmelidir.

```
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)
```

Sağlayıcı çağrısı ile transaction arasında bir çökme olursa, para hareket etmiştir ve Payments'ın kaydı yoktur. Redelivery'de handler sağlayıcıyı tekrar çağırır. Alıcının iki kez tahsil edilip edilmeyeceği artık tamamen sağlayıcıya bağlıdır.

Key'i downstream'e iletmek

Çözüm, idempotency key'i etkiyi gerçekleştiren sisteme taşımaktır; böylece o sistem tekrarı tanıyabilir. Ödeme sağlayıcıları bunu yaygın olarak destekler: request'le birlikte bir key gönderirsiniz ve aynı key'li ikinci bir request yeniden harekete geçmek yerine ilk sonucu döndürür. Header'ın ya da alanın ayrıntıları sağlayıcıya göre değişir; kesin davranışı ve key'leri ne kadar süre hatırladığını öğrenmek için sağlayıcınızın dokümantasyonunu okumanızı öneririm.

En çok önem taşıyan ayrıntı, *key'i nasıl seçtiğinizdir*. Key **işten deterministik olarak türetilmeli, deneme başına üretilmemelidir**.

Handler içinde oluşturulan rastgele bir key, her retry'da yeni bir request'le aynı görünür ve hiçbir şeyi korumaz.

```
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
```

Şimdi hataları yeniden izleyin. Sağlayıcı çağrısından sonra ve commit'ten önce bir çökme, parayı hareket etmiş ve Payments'ı habersiz bırakır.

Yeniden teslim edilen command aynı kodu aynı key ile çalıştırır, sağlayıcı orijinal sonucu döndürür ve transaction onu kaydeder. Response'un

kaybolduđu, sađlayıcı çağrısı sırasındaki bir çökme de aynı durumdur: retry aynı key'i taşıır ve sađlayıcı ya capture'ı bir kez gerçekleştirir ya da önceki sonucu döndürür. Aynı command üzerinde yarışan iki consumer aynı anda aynı key'i gönderir ve ikincisini "in progress" diye reddeden bir sađlayıcı işi başarısız kılmış olmaz: handler bu yanıtı retry edilebilir sayar ve bir sonraki teslimde tekrar bakar.

Sađlayıcının hafızası ile inbox aynı koruma deđildir. Bir sađlayıcı key'leri sınırlı bir süre hatırlar ve ne kadar süre olduđu için dokümantasyonunu okurdum. Bu süreden sonra gelen bir redelivery ikinci bir capture yapardı. Yukarıda anlatılan dedup window'dan en az o kadar uzun tutulan local inbox satırı, handler'ın sađlayıcıyı hiç çağırmasını sađlayan şeydir; dolayısıyla sađlayıcının key retention'ı, güvendiđiniz pencere asla olmaz.

İş mesaja göre deđil iş kuralına göre key'leniyorsa, burada da iş kuralından türetilmiş bir key kullanın; örneğin ödeme id'si artı operasyon. O zaman id'leri farklı iki command bile sađlayıcıya tek bir request olarak ulaşır.

Downstream'in key'i olmadığında

Her downstream bir key kabul etmez. Düz SMTP'de idempotency key yoktur ve birçok e-posta ve SMS sađlayıcısı güvenilir biçimde sunmaz. O zaman iki kusurlu sonuç arasında bir seçimle karşı karşıya kalırsınız ve bunu etki başına açıkça yapmanızı öneririm.

Kaydetmeden önce gönderirseniz, bir çökme duplicate bir e-postaya yol açabilir. Göndermeden önce kaydederseniz, bir çökme eksik bir e-postaya yol açabilir. Bunlar etki için at-least-once ve at-most-once davranışlarıdır. "Kargonuz planlandı" bildirimi için nadir bir duplicate ucuzdur ve kaybolan bir mesaj bir destek talebidir; bu yüzden önce gönderir ve ara sıra olan tekrarı kabul ederim. Deđer kazandıran tek kullanımlık bir kupon kodu için diđer yöne eğilir ve eksik bir gönderimi tespit edip onaran bir yol eklerim.

Bazı etkiler, *niyeti* local transaction içinde kaydedip etkiyi ayrı bir worker'dan gerçekleştirerek daha güvenli hale getirilebilir; bu, command'lara uygulanmış outbox fikridir. Sađlayıcı sınırındaki problemi ortadan kaldırmaz, çünkü worker çağrı ile işaretleme arasında yine çökebilir. Ancak problemi, görebileceđiniz, bilinçli olarak retry edebileceđiniz ve reconcile edebileceđiniz tek bir yere taşıır. Bölüm 13

reconciliation'ı ele alıyor: kayıtlarınızı sağlayıcının kilerle karşılaştırmak ve farkları onarmak; bu, tam olarak idempotent yapılamayan etkiler için son savunmadır.

Idempotency-Key ve HTTP API'ları

Aynı problem kendi ön kapınızda da karşınıza çıkar. Bir alıcının telefonu "sipariş ver" gönderir, ağ response'u düşürür ve uygulama retry yapar. Orders, client söylemedikçe bir retry'ı ikinci bir sipariştan ayıramaz.

Yaygın kullanılan konvansiyon bir **Idempotency-Key** request header'ıdır: **client**, her mantıksal operasyon için benzersiz bir değer üretir ve onun her retry'ında aynı değeri gönderir. Bu, daha önce verilen tanımın diğer yarısıdır: event tüketiminde key normalde producer'ın atadığı **event_id**'dir, client'a dönük request'lerde ise çağırana tarafından sağlanır. Sunucunun işi, tekrarlanan bir key'in ilk request gibi davranmasını sağlamaktır.

Bir sunucunun key'den fazlasına ihtiyacı vardır. Key'i, request'in bir fingerprint'i ve ürettiği response ile birlikte saklar.

```

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

```

Fingerprint kontrolü önemlidir. Bir key'i farklı bir body için yeniden kullanan client'ın bir bug'u vardır ve ilk response'u sessizce döndürmek bunu gizlerdi. In-progress durumu eşzamanlılık durumu için önemlidir: tek bir key'le gelen iki eşzamanlı request'in ikisi de çalışmamalıdır. Row lock yalnızca var olan bir satırın devralınmasını sıralar; ilk sahiplenmeye unique constraint hakemlik eder ve kaybeden insert'ün yakalanıp 409 ile yanıtlanmasının nedeni budur.

Lease crash durumu için önemlidir ve token'ı yavaş durum için önemlidir. Lease'ini aşacak kadar duraklayan bir sahip uyanıp işi bitirmeye çalışabilir, ancak `requireLeaseToken` daha yeni token'ı görür ve sipariş ile outbox satırları dahil bütün transaction'ı rollback olur. Burada token bir fence işlevi görür, çünkü korunan kaynak olan local veritabanı onu transaction içinde kontrol eder; bu, Bölüm 5'teki relay lease'yle aynı fikirdir. Lease süresi yalnızca çakışmayı kısaltır; eski sahibin commit'ini başarısız kılan şey token'dır. Eski çağırana bir hata alır ve retry'ı saklanmış response'u alır. Her iki kontrol de, Bölüm 3'ün saatler hakkında verdiği nedenlerle, uygulama sunucusunun değil, veritabanının saatini, `dbNow()` 'u kullanır. Key'i çağırana göre kapsamlamak, bir client'ın bir başkasıyla çakışmasını önler.

Pseudocode başarı yolunu gösteriyor ve hata yolunun da bir kurala ihtiyacı var. Request, 4xx döndüren bir validation hatası gibi deterministik bir nedenle başarısız olursa sunucu o response'u tamamlanmış response olarak saklar. Aynı key'le yapılan bir retry o zaman aynı 4xx'i alır ve değişmiş olabilecek veriye karşı validation'ı tekrar çalıştırmaz. Bir timeout ya da erişilemeyen bir bağımlılık gibi geçici bir nedenle başarısız olursa sahiplenme serbest bırakılır ya da süresinin dolmasına bırakılır; böylece client'ın retry'ı işi çalıştırabilir. Geçici bir hatayı nihai yanıt olarak cache'lemek, bir aksaklığı o key için kalıcı bir hataya çevirirdi.

`(caller, key)` üzerindeki unique constraint, inbox'ın primary key'yle aynı hakemlik rolünü oynar. Pattern aynı pattern'dir: atomik bir sahiplenme, saklanmış bir sonuç ve tekrar için deterministik bir yanıt. API event üretiyorsa, sipariş satırı, outbox satırları ve idempotency kaydının tamamlanması tek bir local transaction içinde yazılabilir; böylece retry edilen bir request ne ikinci bir sipariş oluşturur ne de ikinci bir `OrderPlaced` yayınlar. Saklanan key'i en az client'ların retry yapabileceği süre kadar tutun; bu, broker'lar hakkında değil client davranışı hakkında bir karardır.

Belirtmeye değer bir bağlantı: client'ın sağladığı key event'e akabilir. Orders onu event header'larına koyabilir ya da `OrderPlaced`'in `event_id`'sini ondan türetebilir; böylece downstream consumer'lar alıcının dokunuşundan ledger kaydına kadar tek bir kimlik görür. `event_id` türetilirse bu, çağırana ile key'den birlikte deterministik biçimde gelmelidir; örneğin ikisi üzerinden bir name-based UUID ile; böylece tesadüfen aynı

key'i seçen iki çağırın asla bir kimliği paylaşmaz. Bölüm 11, bir event'e nelerin ait olduğunu tartışıyor; key'in oraya ait olup olmadığı, idempotency'nin request'i ne kadar takip etmesini istediğinize bağlı.

Trade-off'lar ve maliyetler

Inbox her mesaja bir yazma ekler. Tüketilen her mesaj, hot path üzerinde, aynı transaction içinde bir fazladan insert'e mal olur. Bir avuç yazma yapan bir consumer için ek yük küçüktür. Çok yüksek hacimli, çok ucuz bir consumer için, örneğin bir read-model cache'ini güncelleyen için, baskın maliyet olabilir ve orada doğal idempotency çoğu zaman daha iyi seçimdir.

Inbox depolaması ve temizliği süregelen bir iştir. Tablo trafikle büyür ve retention politikası sahip olduğunuz ve yeniden gözden geçirdiğiniz bir karardır. Fazla kısa bir pencere eski duplicate'leri yeniden içeri alır; fazla uzun bir pencere depolamaya ve index boyutuna mal olur. Birinin temizlik job'unu izlemesi gerekir, çünkü durduğunda tablo, bir performans problemi olana kadar sessizce büyür.

Hot path üzerindeki bir unique constraint bir contention noktası olabilir. İyi dağılmış bir key ile ucuzdur. Kümelenen bir key ile ya da tek bir hot aggregate'e vuran birçok consumer ile kuyruk oluşturabilir. Tabloyu consumer ve event id'ye göre key'lemek ve handler transaction'ını kısa tutmak bunu yönetilebilir kılar.

Business key'ler bir modelleme karardır ve yanlış olanı görmek zordur. Fazla ince olursa mantıksal duplicate'ler geçer; fazla kaba olursa meşru iş reddedilir. Hata her iki yönde de sessizdir. Güvenmeden önce key'i, bir aggregate üzerinde iki meşru operasyon gibi, bilerek zor vakalarla test etmeyi öneririm.

Idempotency her şeyi kapsamaz. Bir tekrarı zararsız kılar; ilk seferde yanlış olan bir handler'ı düzeltmez, mesajları sıralamaz ve sağlayıcının key'i olmayan bir etkiyi geri almaz. Bir inbox ayrıca *değişmiş* bir duplicate'e karşı da korumaz: bir producer "aynı" olguyu farklı bir `event_id` ile yeniden publish ederse ve hiçbir business key yoksa, hiçbir şey onu tanımaz.

Inbox'ı ne zaman kullanmamalı. Consumer'ın her etkisi doğal olarak idempotentse ve stale olamıyorsa, örneğin denormalize edilmiş bir alanı en son tam state ile ezmek gibi, inbox faydasız bir maliyettir; ben onu atlar ve gerekçeyi koda yazardım. Uygun olmadığı bir yer de, kendi transactional veritabanına yazamayan bir consumer'dır, örneğin state tutmayan bir forwarder. Böyle bir consumer'ın downstream'inin key iletilerek deduplicate etmesi gerekir ya da bir depoya sahip olacak şekilde yeniden tasarlanması gerekir.

Sık yapılan hatalar

- **Kontrol etmeyi ve harekete geçmeyi ayrı adımlarda yapmak.** Cazip, çünkü kod doğal okunuyor: "bunu gördüm mü? hayır, o zaman yap." İkisi arasındaki bir çökmede ve eşzamanlılık altında başarısız olur. Inbox yazmasını etkiyle aynı transaction'a koyun ve unique constraint'in hakemlik etmesine izin verin.
- **Inbox satırını ack'ten sonra ya da etki commit edilmeden önce yazmak.** Önce ack etmek, iş sonradan başarısız olursa mesajı kaybettirir. Etki commit edilmeden önce kaydetmek, hiç gerçekleşmemiş işi yapılmış olarak işaretler. Etki, inbox satırı ve commit tek bir bütündür; ack commit'i izler.
- **Inbox'ı etkiden farklı bir depoda tutmak.** Bir cache ya da ayrı bir veritabanı daha ucuz hissettirir ve dual write'ı yeniden yaratır. Onu koruduğu state ile aynı veritabanında tutun.
- **Deneme başına yeni bir key üretmek.** Handler'ın ya da client'ın retry döngüsünün içinde oluşturulan rastgele bir değer, her retry'ı yeni gösterir. Key'i command'dan, event'ten ya da business operasyonundan türetin; böylece her denemede aynı olur.
- **Yalnızca `event_id` kullanıp tüm duplicate'leri kapsadığını varsaymak.** Aynı niyet için iki farklı mesajı yakalamaz. Domain'in böyle bir riski olduğu yerde bir business-key constraint'i ekleyin.
- **Dedup window'u tahminle seçmek.** Broker'ın retention'ından ya da redrive pratiğinizden kısa bir pencere, eski duplicate'leri sessizce yeniden içeri alır. Pencereyi bu sayılardan türetin ve türetmeyi yazın.
- **Bir unique violation'ı genel bir hata saymak.** Her violation'ı ack etmek gerçek bug'ları gizler; her violation'ı retry etmek ele alınmış bir mesajda sonsuza dek döner. Inbox violation'ını ayrıca yakalayın.

- **Harici etkiyi unutmak.** Inbox bir e-postayı ya da bir kart tahsilatını geri alamaz. Sağlayıcıya deterministik bir key iletin ya da olası bir duplicate ile olası bir kayıp arasında bilinçli olarak karar verin.
- **Takip event'lerini doğrudan handler'dan yayınlamak.** Commit ile publish arasındaki bir çökme event'i kaybettirir; redelivery ise inbox fast path'ini izler ve onu asla tekrarlamaz. Takip event'ini, inbox satırı ve state değişikliğiyle aynı transaction içinde outbox'a yazın.
- **Lease'i başkasına ait bir mesajı ack etmek.** Canlı bir lease gören ve ack eden bir duplicate, ilk deneme sonradan başarısız olursa tek retry'ı ortadan kaldırmıştır. Onun yerine bir gecikmeyle yeniden kuyruğa alın.

Temel çıkarımlar

- Duplicate'lar broker'dan, relay'den, retry'lardan, redrive ve replay'den ve client'lardan gelir; bu yüzden her consumer, herhangi bir mesaj iki kez gelebilirmiş gibi yazılmalıdır.
- Check-then-act güvensizdir, çünkü kontrol ile etki ayrı adımlardır ve bir çökme ya da eşzamanlı bir deneme ikisinin arasına girebilir.
- Inbox satırı ve business değişikliği, consumer'ın kendi veritabanında tek bir local transaction içinde commit edilmelidir ve eşzamanlı duplicate'lere gerçekte hakemlik eden şey key üzerindeki unique constraint'tir. Ne yaptığını duyuran bir consumer, aynı transaction'a bir outbox satırı ekler.
- Event tüketimi için temel idempotency key olarak `event_id`'yi kullanın ve iki farklı mesajın aynı niyeti ifade edebildiği her yerde bir business key ekleyin.
- Dedup window, broker'ın retention'ını, redrive pratiğinizi ve planladığınız her replay'i aşmalıdır; domain tablosundaki bir business-key constraint'i ise inbox satırları silindikten sonra sizi korur.
- Harici yan etkiler için, denemeden değil işten türetilmiş deterministik bir idempotency key gönderin, sağlayıcı key'leri unuttuğu için local inbox satırını uzun vadeli koruma olarak tutun ve downstream'in key'i olmadığında olası bir duplicate ile olası bir kayıp arasında bilinçli olarak seçim yapın.

- API sınırında idempotency key'i çağıran sađlar ve sunucu key'i, bir request fingerprint'ini ve response'u saklar; böylece bir tekrar ilk sonucu döndürür.

Idempotency, tekrarlanan bir mesajı zararsız kılar, ancak işlemin eninde sonunda başarılı olacağını varsayar. Bazı mesajlar asla olmayacak: her denemede başarısız olurlar ve retry'lar ile redelivery'ler onların arkasında yığılır. Bölüm 8, bu poison message'larla ve geride bıraktıkları dead letter'larla ilgileniyor.

Bölüm 8 — Poison Message'lar ve Dead Letter'lar

Parcelly'yi, `OrderPlaced` üzerinde yeni bir alanın, `gift_message`'in yayına girdiği sabahın ertesinde düşünün. Notifications onu onay e-postasını oluşturmak için okuyor ve bir satıcının test siparişi, e-posta şablonunun encode edemediği bir karakter içeren bir gift message taşıyor. Consumer bu tek mesajda exception fırlatıyor. Mesaj geri geliyor. Consumer yine exception fırlatıyor.

Öğle yemeğine kadar mesaj birkaç bin kez teslim edilmiş oluyor. Her deneme aynı yerde, aynı biçimde, milisaniyeler içinde başarısız oluyor. Consumer meşgul, her altyapı metriğine göre sağlıklı ve saatlerdir yararlı hiçbir şeyi ack etmedi. Queue tabanlı bir broker'da diğer mesajlar kalan worker'lar tarafından hâlâ işleniyor; dolayısıyla hasar küçük görünüyor. Log tabanlı bir topic'te ise bir partition hareket etmeyi bıraktı, çünkü consumer başarısız kaydın üzerinden geçmek yerine sürekli ona geri seek ediyor ve o partition'daki her sipariş onun arkasında bekliyor.

Sonunda biri fark ediyor ve biri, iş devam edebilsin diye mesajı başka bir yere taşımayı öneriyor. Yanıtın kısa yarısı bu. Zor yarısı bundan sonra olanlar: o mesaja kim bakacak, ne anlama geldiğine nasıl karar verecek ve ikinci bir incident'a yol açmadan sisteme nasıl geri dönecek? Bu bölüm iki yarıyı da ele alıyor. Bir sistem işleyemediği bir mesajla ne yapmalı ve bir ekip bu yanıtın bir çöplüğe dönüşmesini nasıl önler?

Problem ve neden zor olduğu

Poison message, her işlendiğinde başarısız olan bir mesajdır. "Her" kelimesi tanımı taşır. Bir veritabanına kısa süre ulaşamadığı için başarısız olan bir mesaj poison değildir; veritabanı geri döndüğünde başarılı olacaktır. Poison bir mesaj, mesajla ya da onu okuyan kodla ilgili bir şey, bir insan ya da bir deployment bir şeyi değiştirene kadar başarıyı imkânsız

kıldığı için başarısız olur. Bölüm 3'ün sözlüğüyle, hatanın semantiği yeniden denemenin yardımcı olamayacağını söyler ve onu retry etmenin boşa emek olmasının nedeni de tam olarak budur.

Olağan nedenler birkaç gruba ayrılır:

- **Hatalı biçimlendirilmiş ya da beklenmeyen içerik.** Zorunlu bir alan eksik, bir değer aralığın dışında ya da bir payload parse edilmiyor. Eski bir consumer ile uyumlu olmayan schema değişiklikleri bunu üretir (Bölüm 12 bundan nasıl kaçınılacağını ele alıyor).
- **Consumer'daki bir bug.** Mesaj geçerlidir, ama kod kimsenin test etmediği bir durumu yanlış ele alır; örneğin hiç beklenmeyen bir yerde null gelmesi.
- **İhlal edilmiş bir business varsayımı.** Bir `PaymentCaptured` event'i, eşleşen authorization kaydı olmayan bir consumer'a ulaşır; belki veri silindiği için, belki de event'ler yolda birbirini geçtiği ve retry'lar makul her reordering penceresini çoktan aştığı için. O pencerenin içinde aynı belirti geçicidir; Bölüm 6'nın belirttiği ve Bölüm 9'un derinlemesine ele aldığı gibi, DLQ'ya değil retry'lara aittir. Pencerenin ötesinde consumer devam edemez ve eksik kaydı uyduramaz.
- **Kalıcı bir downstream reddi.** Harici bir carrier bir adresi geçersiz diye reddeder. Tekrar denemek aynı reddi üretir.

Bölüm 6, geçici ve kalıcı hatalar arasındaki ayrımı tanıttı ve bu bölüm onun bittiği yerden devam ediyor. Doğru çalışan bir retry politikası eninde sonunda bir mesajdan vazgeçecektir. Retry'lar deneme sayısı, süreyle ya da bütçeyle sınırlıdır. O noktada ne olacağı bir tasarım kararıdır ve kimse vermezse broker sizin yerinize verir.

Sonsuz redelivery döngüsü

Consumer özel bir şey yapmazsa broker'ın varsayılan davranışı devreye girer. Queue tabanlı bir broker, ack edilmemiş bir mesajı ya bir nack'ten hemen sonra ya da visibility timeout dolduğunda yeniden teslim eder ve genellikle her denemeyle artan bir redelivery sayısı tutar.

Log tabanlı bir broker farklı çalışır ve fark önemlidir. Mesaj başına redelivery yoktur: consumer bir offset tutar ve ileri doğru okur. Bir offset'i commit etmemek, broker'ın kaydı geri vermesini sağlamaz; consumer'ın bellekteki konumu ilerlemeye devam eder ve başarısız kayıt, bir sonraki

restart ya da rebalance consumer'ı son commit edilmiş offset'e geri sarana kadar sessizce atlanır. Bir log üzerindeki döngü yalnızca consumer onu kurduğu için vardır: başarısız offset'e geri seek eder ve onu tekrar okur ya da yerinde retry yaparken partition'ı ilerletmeyi durdurur. İkisini de yapmayan ve yine de başarısızlığın ötesinde commit eden bir consumer, kaydı hiçbir sinyal vermeden kaybeder.

Her iki ailede de sonuç aynı olabilir: mesaj geri gelir, başarısız olur ve tekrar geri gelir ya da dikkatsiz offset yönetimine sahip bir log'da kaybolur. Log ailesi için doğru araçlar, açık bir seek ya da pause; ana partition ilerleyebilsin diye başarısız kaydı alan bir retry topic'i ve vazgeçilen kayıtlar için bir dead-letter topic'idir.

Döngünün üç maliyeti vardır. Birincisi boşa giden kapasitedir: worker'lar zamanlarını başarılı olamayacak işe harcar. İkincisi gürültüdür: log'lar birbirinin aynı stack trace'lerle dolar ve gerçek problemler aralarında saklanır. Üçüncüsü, en önemli olanıdır: **head-of-line blocking**. Bir log'da consumer bir partition'ı sırayla işler; dolayısıyla atlamayacağı bir kayıt, arkasındaki her kaydı bloke eder. Sade bir queue'da tek bir poison message kendi başına diğerlerini bloke etmez, ama bir avuç poison message her worker'ı meşgul edebilir. Queue mesajları ordering için gruplandırıyor (FIFO queue'lar, message group'lar, session'lar), bir poison message group'unu bloke eder; bu, bir log partition'ıyla aynı etki alanıdır, yalnızca daha dardır.

İki kötü kaçış

Baskı altındaki ekipler genellikle iki kaçıştan birine başvurur ve ikisi de aynı nedenle yanlıştır: hatayı ele almak yerine gizlerler.

Birincisi **mesajı düşürmektir**: exception'ı yakala, log'la, ack et ve devam et. Pipeline yeniden akar, ama business event gitmiştir. `OrderPlaced` için bu, `Orders`'ta olan ve başka hiçbir serviste olmayan bir sipariş demektir. `PaymentCaptured` için gerçeklikle uyuşmayan bir ledger demektir. Düşürmek yalnızca mesaj hiçbir değer taşımadığında doğru seçimdir ve bir consumer bunu başarısızlık anında bilemez.

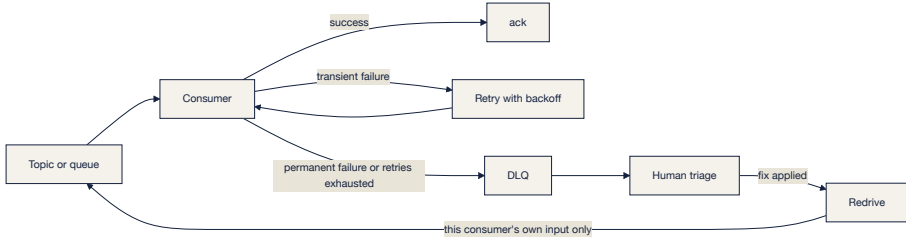
İkincisi **sonsuz dek retry etmek** ve birinin nedeni düzeltereğine güvenmektir. Bu, yukarıdaki sonsuz döngüdür ve tek bir kötü mesajı arkasındaki her şey için bir outage'a çevirir.

Üçüncü seçenek bunların arasında durur. Mesajı, bir insanın onu sonradan anlayabileceği yeterli bağlamla kenara koyun ve stream'in geri kalanının devam etmesine izin verin. Bu seçenek dead-letter queue'dur.

Dead-letter queue pattern'i

Dead-letter queue (DLQ), bir consumer ya da broker'ın vazgeçtiği mesajları yerleştirdiği ayrı bir hedeftir. Log tabanlı bir broker'da aynı fikir genellikle bir **dead-letter topic**'idir: başarısız kayıtları alan sıradan bir topic. Bir ifade her iki aile için geçerli olduğunda bu bölüm DLQ der.

Pattern'in basit bir çekirdeği var. Bir mesaj retry'larını tükettikten sonra ya da consumer'ın kalıcı olarak sınıflandırdığı bir biçimde başarısız olduktan sonra, consumer onu DLQ'ya taşır ve orijinali ack eder. Stream devam eder. Başarısız mesaj kaybolmaz; incelenebileceği bir yere park edilir.



Diyagramdaki son iki oka bakın. DLQ bir çıkış değildir. Başarısız olan consumer'ın kendi input'una geri açılan bir kapısı olan bir bekleme odasıdır ve bu bölümün bütün zorluğu o iki kapı arasındaki mesafede yatar. Kapı paylaşılan bir topic'e açılmaz: diğer consumer'lar event'i zaten işledi ve onu bir daha görmemelidir.

Taşıma nerede yapılır

Taşımayı uygulamanın iki yolu var ve seçim neyi saklayabileceğinizi etkiler.

Broker tarafından yönetilen dead-lettering. Birçok queue tabanlı broker, belirli bir sayıdan fazla teslim edilen bir mesajın otomatik olarak belirlenmiş bir queue'ya taşınacağı şekilde yapılandırılabilir. Kurması ucuzdur ve consumer, bir şey yapabilmekten çökse bile çalışır. Size bir delivery count verir, ama genellikle consumer'ın neden başarısız olduğunu vermez, çünkü broker bunu bilmez.

Consumer tarafından yönetilen dead-lettering. Consumer hatayı yakalar, nedeni içeren bir kayıt oluşturur, onu DLQ'ya publish eder ve ancak ondan sonra orijinali ack eder. Bu, hataları sınıflandırmanıza ve kalıcı bir hatayı beşinci denemede değil ilk denemede dead-letter etmenize olanak tanır. Her consumer'da kod gerektirir ve dikkatle yazılmalıdır: önce DLQ'ya publish edin, sonra ack edin; böylece ikisi arasındaki bir çökme DLQ'da bir duplicate üretir, asla bir kayıp üretmez.

Broker birincisini sunuyorsa ikisini de kullanırdım. Broker tarafından yönetilen kural yalnızca bazı queue tabanlı broker'larda vardır ve her queue türü teslimatı sınırlamaz. Var olduğu yerde, kodun yakalayamadığı biçimlerde çöken ya da takılan consumer'lar için güvenlik ağıdır ve consumer tarafından yönetilen yolla birlikte, kendi ele alışınızda bir bug olsa bile poison bir mesajın sonsuza dek dönemeyeceği anlamına gelir.

Log tabanlı broker'ların genellikle ne mesaj başına bir delivery sayacı ne de broker tarafında bir DLQ'su vardır ve bu bir boşluk bırakır. Süreci çökerten bir mesaj, örneğin handler çalışmadan önce bir deserializer hatası, bir out-of-memory hatası ya da bir native crash, `try` bloğunuza hiç ulaşmaz. Consumer commit edilmiş offset'ten yeniden başlar, aynı kaydı okur ve yeniden çöker. Bölüm 6, attempt sayısının bir crash'ten sonra baştan başladığına dikkat çekmişti. Poison bir kayıt için sorunun tamamı budur.

Çare, sayının crash'ten sağ çıkması için işlemeyen önce saymaktır:

```
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)
```

Sayaç partition ve offset ile anahtarlanır (deserialize etmeden okuyabiliyorsanız `event_id` ile). Bedeli, sıcak yolda kayıt başına durable bir yazma ve sıradan bir deploy restart'ının sağlıklı kayıtları üzerine

itmeyeceği kadar yüksek ayarlanması gereken bir limittir. Alternatif, crash döngüsündeki bir poison kaydın, biri fark edene kadar partition'ını durdurmasını kabul etmektir ve aşağıda anlatılan lag alarminin önemli olmasının nedeni budur.

```
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)
```

`deadLetter` adımı, triage için gereken bağlamın yakalandığı yerdir ve son iki satırının sırası asıl noktadır.


```

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)

```

`publish` kendisi başarısız olursa `deadLetter` hata fırlatır ve mesaj ack edilmez. Queue ailesi bir broker'da bu yeterlidir: mesaj normal redelivery yoluyla geri döner. Log ailesi bir broker'da yeterli değildir, çünkü commit edilmemiş bir offset kaydı geri getirmez; consumer başarısız kaydın konumuna geri seek etmeli ya da partition'ı pause etmeli ve daha ileri okumadan önce dead-letter publish'ini retry etmelidir. Her iki durumda da hedef aynı hata modudur: mesaj, güvenli bir yeni yuvası olana kadar sistemde kalır.

Bir dead letter ile neler saklanmalı

Bağlamı olmayan bir dead letter bir bulmacadır. Altı hafta sonra bir DLQ'da bir mesaj bulan bir ekibin dört soruyu yanıtlaması gerekir: ne idi, nereden geldi, neden başarısız oldu ve hâlâ işlemek güvenli mi? Kayıt, bir kaynak depoyu açmadan bunları yanıtlamalıdır.

Alan	Neden orada
Orijinal body ve header'lar, byte byte	Redrive, <code>event_id</code> dahil mesajı tam olarak yeniden üretmelidir

Alan	Neden orada
Kaynak adı ve konumu (topic ve partition ve offset ya da queue adı)	Mesajı geçmişte konumlandırır, ordering kararlarını destekler
Orijinal key (record key ya da routing key) ve orijinal timestamp	Redrive aynı key ile publish etmelidir; yoksa mesaj başka bir partition'a düşer ve key başına sırasını kaybeder
Consumer input'u (başarısız olan consumer'ın kendi queue'su ya da redrive topic'i)	Bir redrive'ın publish edebileceği tek yer
redrive_count	attempt_count'tan ayrı: bu mesajın şimdiden kaç kez geri geldiği
error_class ve error_message	Hataları nedene göre gruplar; uzun mesajları truncate edin
Stack trace ya da log satırına bir işaretçi	Bir mühendisin yeniden üretmeden nerede başarısız olduğunu görmesini sağlar
attempt_count	"Bir kez başarısız oldu, kalıcı" ile "çok sayıda retry'dan sonra başarısız oldu" arasında ayrım yapar
first_failed_at ve failed_at	Mesajın ne kadardır beklediğini ve problemin ne zaman başladığını gösterir
Consumer adı ve version'ı	Hatayı bir deploy'a bağlar; çoğu zaman nedene giden en hızlı yoldur
correlation_id	Mesajı business akışının geri kalanına bağlar

İki uyarı geçerli. Birincisi, orijinal payload'u değiştirmeden tutun. Onu kendi envelope'unuza sararsanız, redrive aracı onu tam olarak açmalıdır; yoksa event_id ve causation_id kayar ve Bölüm 7'deki idempotency garantileri kaybolur. İkincisi, bir DLQ'nun production verisinin bir kopyası olduğunu ve çoğu zaman uzun süre tutulduğunu unutmayın. Orijinalin erişim kontrollerini ve retention yükümlülüklerini devralır ve kişisel veri içeren payload'lar DLQ'da da başka her yerdeki kadar özen gerektirir.

Parking lot ve DLQ

DLQ kelimesi iki farklı şey için kullanılır ve onları ayırmak operasyonu akıl yürütmeyi kolaylaştırır.

Parking lot, bir insan kararı gerektiren dead letter'lar için bir bekletme yeridir. DLQ, sistemin bir mesajı otomatik olarak koyduğu yerdir; parking lot ise bir kişinin bir mesaja baktıktan ve onun bir redrive ile düzelmeyeceğine karar verdikten sonra bilerek koyduğu yerdir. Aynı bir amacı olan ayrı bir yerdir.

Her birinin ne içerdiğini düşünün. DLQ, başarısız olmuş ve incelenmemiş mesajları tutar. Boşalmalıdır: beklenen durumu boş ya da boşa yakındır ve içindeki her eski şey bir belirtidir. Parking lot, birinin incelediği ve kenara koyduğu mesajları tutar: event'i gelmeden iptal edilmiş bir sipariş, elle düzeltilmesi gereken bir payload, başka bir ekiple elle reconcile edilmesi gereken bir kayıt. Bu mesajlar deploy edilecek bir düzeltmeyi beklemiyor. Bir kararı bekliyor.

Bunları karıştırmak öngörülebilir bir hata üretir. Karara bağlanmış ve park edilmiş mesajlar DLQ'da kalırsa DLQ'nun derinliği asla sıfıra dönmez, gürültüyü durdurmak için alert eşiği yükseltilir ve bir sonraki gerçek poison message fark edilmeden gelir. Onları ayrı tutmak DLQ alarımının keskin kalmasını sağlar.

Bir parking lot, bir status, bir sahip, bir neden ve bir gözden geçirme tarihi olan bir veritabanı tablosu kadar mütevazı olabilir. Hiç bir broker hedefi olması gerekmez, çünkü onu otomatik olarak tüketecek bir şey yoktur. En önemli olan gözden geçirme tarihidir: bir sonraki bakışı için tarihi olmayan park edilmiş bir mesaj zaten unutulmuştur.

Sahiplik ve alarmlar

Sahibi olmayan bir DLQ, sessizce dolan bir giderdir. Herhangi bir araçtan önce ilk karar, her birinden kimin sorumlu olduğudur.

Varsayılanım, consumer'ın sahibi olan ekibin DLQ'nun da sahibi olmasıdır. Başarısız olan kodu onlar yazdı, mesajın ne anlama gelmesi gerektiğini onlar biliyor ve düzeltip deploy edebilecek olanlar da onlar. Consumer başına ya da consumer group ve kaynak başına bir DLQ, sahipliği açık hale getirir. Bütün şirket için tek bir paylaşımlı DLQ onu belirsiz kılar ve belirsizlik kimsenin bakmaması demektir. Araçları bir platform ekibi sağlıyorsa, platform ekibi mekanizmanın, tüketen ekip içeriğinin sahibidir.

Neye alarm kurulmalı

Varışa alarm kurun, yalnızca derinliğe değil. Normalde boş olan bir DLQ'da tek bir mesajın belirmesi bir bildirimi hak eder, çünkü ilk dead letter bir bug'ı bulmak için en ucuz andır. Tek başına bir derinlik eşiği, queue zaten derin olana kadar size hiçbir şey söylemez.

Üç sinyal birlikte çoğu durumu kapsar:

- Boş olması gereken bir DLQ'ya **herhangi bir yeni varış**. Bunu, bir insanın mesai saatlerinde okuduğu bir kanala yönlendirin.
- **Varış hızı**. Ani bir patlama genellikle kötü bir deploy ya da uyumsuz bir schema değişikliği anlamına gelir ve birini page etmeyi haklı çıkarır.
- **En eski mesajın yaşı**. Bu, yavaş hatayı yakalar: günde bir mesaj alan, kimsenin işlem yapmadığı ve en eski mesaj bir aylık olana, business bağlamı gidene kadar süren DLQ.

Sisteminiz için doğru paging eşiğini bilmiyorum ve söz konusu mesajların business değerini bilmeden kimse size söyleyemez. `PaymentCaptured` consumer'larına bağlı bir DLQ, `Notifications`'a bağlı olandan daha düşük bir eşiği hak eder, çünkü kaybolan bir e-posta bir rahatsızlık, kaybolan bir capture ise bir ledger tutarsızlığıdır. Eşikler bu farkı yansıtmalı ve fark alarının yanına yazılmalıdır.

Not: Consumer lag'e de alarm kurun. Bir log partition'ında poison bir kayıta takılı kalan bir consumer, hiç vazgeçmediği için mutlaka dead letter üretmez. Throughput olmadan yükselen lag, bu bölümün başında anlatılan döngünün çalıştığının sinyalidir.

Redrive: mesajları geri almak

Redrive, dead letter'ları yeniden işlenmek üzere, onlarda başarısız olan consumer'a geri vermek anlamına gelir. Hedef o consumer'ın kendi input'udur: retry ya da redrive topic'i, ya da kendi queue'su. Hiçbir zaman mesajın aslen geldiği paylaşılan topic değildir, çünkü o topic'in diğer her consumer'ı event'i zaten işlemiştir ve oraya yeniden publish etmek onu hepsine ikinci kez teslim ederdi. Orijinal key mesajla birlikte yolculuk eder; böylece mesaj aynı partition'da kalır ve key başına konumunu korur.

Attempt sayacı sıfırdan yeniden başlar ve ayrı `redrive_count` bir artar; böylece sürekli geri gelen bir mesaj, her seferinde yeniymiş gibi görünmek yerine görünür olur. Redrive, bir DLQ'yu değerli kılan operasyondur ve aynı zamanda ikinci bir incident'a yol açma ihtimali en yüksek olandır. Bölüm 3, bir düğmeye basıp 40.000 mesajı bir anda redrive eden mühendisi anlattı. Çare, redrive'ı bir kolaylık düğmesi olarak değil, tasarlanmış bir prosedür olarak ele almaktır.

Redrive'dan önce nedeni düzeltin

Bir consumer bug'ı yüzünden başarısız olan mesaj, bug düzeltilip deploy edilene kadar yeniden başarısız olur. Düzeltmeden önce redrive etmek mesajları bir daire içinde taşır ve DLQ'yu aynı kayıtlarla, şimdi daha yüksek attempt count'larla ve kafa karıştırıcı bir geçmişle yeniden doldurur. İşin sırası şöyledir: nedeni anlayın, düzeltmeyi deploy edin, küçük bir örneği redrive edin, başarılı olduğunu izleyin, sonra geri kalanını redrive edin.

Redrive tekrarlanabilir derecede güvenli olmalı

Redrive edilen bir mesaj tanım gereği bir duplicate'tir: sistem bu `event_id`'yi daha önce görmüştür ve consumer da görmüş olabilir. Orijinal işleme başarısız olmadan önce kısmen başarılı olduysa bazı etkiler zaten var olabilir. Bunun atlatılabilir olmasının nedeni Bölüm 7'dir. Inbox'lı idempotent bir consumer, redrive edilen mesajı herhangi bir redelivery gibi ele alır ve etkilerini bir kez uygular.

İki nokta belirtmeye değer. Redrive orijinal `event_id`'yi korumalıdır, çünkü idempotency key odur. Her mesaj için yeni bir tanımlayıcı üreten bir redrive aracı inbox'ı boşa çıkarır ve etki iki kez uygulanabilir. Ve idempotent olmayan bir consumer toplu redrive'ın hedefi olmamalıdır; önce onu düzeltin ya da redrive edilen her mesajın elle yapılan bir risk değerlendirmesi olduğunu kabul edin.

Redrive'ı kontrollü batch'lerle yapmak

Güvenli bir redrive aracının birkaç özelliği vardır; hiçbiri sıra dışı değildir:

1. **Bir seçim filtresi.** Yalnızca "her şey" değil, error class'a, zaman aralığına ya da `correlation_id`'ye göre redrive.

2. **Bir rate limit.** Mesajlar consumer'ın input'una, onun ve bağımlılıklarının sindirebileceği bir hızda döner. Redrive edilen mesajların bir seli, kendi kendine yaptığınız bir retry storm'dur ve "Aşırı yük altında retry'lar: backpressure" (Bölüm 6) bölümündeki uçustaki iş limitleri ve yük atma, redrive edilen trafiğe tıpkı canlı trafiğe olduğu gibi uygulanır.
3. **Bir pause anahtarı.** Hatalar yeniden belirirse bir operatör redrive'ı ortasında durdurabilir.
4. **Bir dry run.** Araç, herhangi bir şeyi taşımadan önce neyi taşıyacağını raporlar.
5. **Bir audit kaydı.** Kimin neyi, ne zaman ve neden redrive ettiği, dayanıklı bir yerde tutulur.

```
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"
```

Döngünün içindeki sıraya dikkat edin: bir kayıt, publish başarılı olduktan sonra redrive edilmiş olarak işaretlenir, önce değil. Araç ortada çökerse birkaç mesaj redrive edilmiş ve DLQ'da hâlâ işaretlenmemiş durur; bu kayba değil duplicate'lere yol açar. Inbox'lı bir consumer bu duplicate'leri sindirir; olmayan, bir araç çökmesini veri problemine çevirir. Kayıt

silinmez, işaretlenir; böylece audit trail ve `redrive_count` korunur. Temizlik aşağıdaki "Çöplüğü önlemek" bölümündeki retention kuralını izler.

Redrive ve ordering

Redrive, ona göre tasarlamadıkça ordering'i bozar. Dead-letter edilen bir mesaj atlandı; aynı aggregate için sonraki mesajlar bu arada işlendi. Redrive edilen mesaj geldiğinde onlardan sonra gelir. Bu, Bölüm 6'daki Inventory örneğinin daha uzun bir aralıkla hâlidir: `OrderPlaced` retry edilmek yerine dead-letter edilirse, `OrderCancelled` yine serbest bırakılacak bir rezervasyon bulamaz ve saatler sonra redrive edilen yerleştirme, yine hiçbir şeyin serbest bırakmayacağı bir tane oluşturur. Key'i korumak bunu onarmaz: key mesajı doğru partition'a koyar, ama partition ilerlemiştir.

Bunun önemli olup olmadığı consumer'a bağlıdır. **Commutative** olan, yani event'ler hangi sırada gelirse gelsin nihai state'in aynı olduğu bir consumer etkilenmez. State geçişlerine bağlı bir consumer etkilenir. Bölüm 9, sıra dışı gelişin version numaraları ya da sequence kontrolleriyle nasıl tespit edilip ele alınacağını anlatıyor ve state tabanlı bir consumer, kimse ona redrive etmeden önce bunları kullanmalıdır. Kullanmıyorsa redrive prosedürü, mesaj dead-letter edildiğinden beri dünyanın ilerlemediğine dair bir kontrol içermelidir.

Bir dead letter'ın ordered bir stream'e yaptığı

Ordered stream'ler bu bölümün en zor kararını zorunlu kılar, çünkü ordering'i koruyan seçim, business'ı durduran seçimdir.

`order_id` ile key'lenmiş, Ledger'ın kayıt post ettiği log tabanlı bir topic düşünün. Consumer'lar her partition'ı sırayla işler. `order-81724` için 41 numaralı mesaj kalıcı olarak başarısız olur. Consumer'ın iki seçeneği vardır:

- **Dur ve bekle.** Partition'ı, biri nedeni düzeltene kadar 41 numaralı mesajda tutun. Ordering tam olarak korunur. O partition'a hash'lenen diğer her sipariş de, mesajları sağlam olsa bile, durur. Etki alanı bütün partition'dır.

- **Dead-letter et ve devam et.** 41 numaralı mesajı park edin, 42 ve sonrasını işleyin. Partition akar. Ama 42, 41'e bağlı olabilir ve consumer onu o zaman 41'i içermeyen bir state'e karşı işlemiş olur.

İki seçenek de bedava değil. Durmak, availability'yi correctness ile takas eder; devam etmek correctness'ı availability ile takas eder. İkisini birden veren bir ayar yoktur ve framework varsayılanı ne ise onu miras almaktansa, consumer başına bilerek seçmek daha iyidir.

Pratik kuralımın üç parçası var:

1. **Consumer sıraya duyarsız ya da commutative ise**, dead-letter edin ve devam edin. Notifications tipik durumdur: bir siparişteki başarısız bir e-postanın bir sonrakiyle ilgisi yoktur.
2. **Aynı key için sonraki mesajlar başarısız olana bağlıysa**, o key için devam etmeyin. Başarısız mesajı dead-letter edin, sonra aynı `partition_key` için izleyen mesajları da ilki çözülene kadar bir bekletme alanına yönlendirin. Diğer key'ler devam eder. Bu, etki alanını bir partition yerine bir aggregate'te tutar; bedeli ek bir mekanizmadır: "bloke" key'lerin key başına izlenmesi.
3. **Consumer bir finansal ledger ise, durun.** Sıra dışı işlemin maliyeti, tespit etmesi zor ve onarması pahalı biçimde yanlış bir kayıttır; duraklatılmış bir partition ise gürültülü ve açıktır. Ledger için belirtilen kural olan speed yerine correctness, burada en yalın haliyle uygulanır.

İkinci kuraldaki key başına varyant, en kullanışlı ve en az yaygın olduğu için bir taslağı hak ediyor. Taslak, bloke edilmiş key başına bir satırı olan küçük bir `key_block` tablosunu (`partition_key`, `blocked_event_id`, `state`) ve belirli bir key'in canlı trafiğini aynı anda tek bir consumer'ın ele aldığını varsayar; bir partition ya da bir message group'un size verdiği şey budur.


```

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)

```

Serbest bırakma ayrı bir prosedürdür ve periyodik bir sweep onu yeniden başlatılabilir kılar.

```

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)

```

Bu taslaktaki üç ayrıntı, işe yarayıp yaramayacağını belirler. Birincisi redrive'ın kendisidir. Redrive edilen mesaj, key'in bloke olduğu mesajın ta kendisidir; bu yüzden bloğu bypass etmesi gerekir: kontrol onun `event_id`'sini bloke olanla karşılaştırır. Bu istisna olmadan redrive, öncesinde gelmesi gereken mesajların arkasına park edilir ve key asla açılmaz.

İkincisi, canlı trafikle yarışır. `resolveBlockedKey` held mesajları boşaltırken, aynı key için yeni mesajlar gelmeye devam eder. Held küme boşalana kadar yine de held edilmeli, işlenmemelidir. Bu yalnızca park etme kararı ile temizleme kararı serileştirilmişse geçerlidir: ikisi de key'in satır lock'unu alır ve park, kontrolle aynı transaction içinde yazılır. Park ayrı bir adım olsaydı, canlı bir mesaj bloke bir key görebilir, sonra held

kümenin zaten boşaltılıp temizlendiğini bulabilir ve artık var olmayan bir bloğun arkasına park edilebilirdi. Onu kimse boşaltmazdı ve sonraki mesajlar onun önünde işlenirdi.

Üçüncüsü, serbest bırakmanın ortasında bir crash'tir. Redrive edilen mesaj işlendikten sonra `resolving` state'i ack'ten önce yazılır; böylece yapılacak iş yalnızca asla geri gelmeyecek bir mesajda değil, veritabanında kayıtlıdır. Sweep, o state'te kalan her key'i alır ve devam eder; crash'ten kısa süre önce işlenen bir mesajı inbox sindirir. Serbest bırakma sırasında held bir mesaj kalıcı olarak başarısız olursa, key o mesaja çapalanmış olarak, kendi dead letter'ıyla yeniden bloke olur ve held kümenin geri kalanı onun redrive'ını bekler. Key başına blok bir döngüdür, tek seferlik bir olay değil.

Taslak eksiksiz değildir. Bir sweep, canlı bir serbest bırakmayla çakışabilir; dolayısıyla iki resolver aynı key üzerinde çalışabilir. Lock altındaki state kontrolü, ikisinden birinin yeniden bloğu geçmesini engeller: bir resolver bloğu başarısız bir mesaja çapaladığında, diğeri key'in artık `resolving` olmadığını görür ve durur. Geriye kalan, aynı held mesaj üzerinde yinelenen iştir ve onu inbox sindirir; `resolving` state'i üzerinde bir lease, doğruluğu değil bu işi kurtarırdı. Ayrıca held kümeye bir sınır koymaz ve bir operatörün artık anlamlı olmayan held bir mesajı nasıl attığını dışarıda bırakır. Onu bitmiş bir mekanizma olarak değil, mekanizmanın şekli olarak ele alın.

Bu, kendine ait gerçek hata modları olan gerçek bir mekanizmadır; örneğin sınırsız büyüyen held-messages deposu. Bu yüzden onu yalnızca key başına ordering'in önemli olduğu ve dead-letter oranının, held kümenin küçük kalacağı kadar düşük olduğu stream'ler için kurardım. Ayrıca, başarısız olan çözüldükten sonra held mesajların hâlâ anlamlı olup olmadığını bilmeyen bir mekanizmadır. State geçişlerinin yalnızca sıralanması değil, doğrulanması gerektiğinde, Bölüm 9'daki version buffer daha güçlü bir araçtır: bir event'i, öncülü version numarasına göre gerçekten uygulanana kadar tutar ve çapalanacak bir dead-letter kaydı olmadan çalışır.

Çöplüğü önlemek

Bir DLQ, mesajlar biri onların ne anlama geldiğine karar verebileceğinden daha hızlı girdiğinde bir çöplüğe dönüşür. Belirti tanındıktır: binlerce kayıt, birçok farklı hata, hangilerinin önemli olduğundan emin olmayan insanlar ve ona dokunulmayacağına dair yazılı olmayan bir kural. O noktada DLQ artık bir güvenlik mekanizması değildir; kaybolmuş verinin nazik bir adla saklandığı bir yerdir.

Çöplüklerin nedenleri vardır ve her birinin bir karşılığı vardır.

- **Sahip yok.** Karşılığı, yukarıdaki yazılı sahiplik kuralıdır.
- **Triage ritmi yok.** Dead letter'lar yalnızca bir alarm tetiklendiğinde değil, bir takvime göre gözden geçirilmelidir. Düşük hacimli bir DLQ için haftalık gözden geçirme makul bir başlangıç noktasıdır.
- **Faydasını aşan ya da triage'dan önce dolan retention.** Geçen ay süresi dolan bir stok rezervasyonu için dead letter anlamlı biçimde redrive edilemez. Retention süresini mesajın business için ne kadar geçerli kaldığına göre belirleyin ve süre dolduğunda ne olacağına önceden karar verin: arşivle, audit kaydıyla at ya da escalate et. Tersî hata da aynı ölçüde yaygındır: triage sıklığından kısa bir retention, mesajları kimse bakmadan siler. DLQ, retention'ı inceleme aralığınızdan kısa olan bir broker kuyruğuysa, delili broker önce kaldırır. DLQ retention'ını incelemeler arasındaki en uzun aralığın üstüne ayarlayın ve platformunuzun dokümantasyonunda retention saatinin neyden başladığını okuyun. Saat dead-letter anından değil de özgün enqueue zamanından başlıyorsa, başarısız olmadan önce uzun süre bekleyen bir mesaj süresi dolmaya yakın şekilde gelir. Kaydın özgün timestamp'i saklamasının bir nedeni daha budur.
- **Her şey dead-letter ediliyor.** Akışın normal bir parçası olan bir validation reddi gibi rutin business sonuçları DLQ'ya gönderilirse gerçek hatalar boğulur. Reddedilen bir sipariş, kendi ele alınışı olan bir business event'idir, bir dead letter değil.
- **Geçici hatalar fazla erken dead-letter ediliyor.** Retry politikası kısa bir outage sırasında saniyeler içinde vazgeçerse, o outage sırasında uçuşta olan her mesaj DLQ'ya düşer. Çare Bölüm 6'dadır: deneme sayıları ve deadline'ı, hayatta kalmayı beklediğiniz outage'a göre boyutlandırılmış bir retry politikası.

Her dead letter'a bir disposition da verirdim. Triage kısa bir sonuç listesinden biriyle biter: bir düzeltmeden sonra redrive edildi, düzeltilip redrive edildi, elle karar için park edildi ya da kayıtlı bir nedenle atıldı. "Hâlâ orada duruyor" bir disposition değildir.

Trade-off'lar ve maliyetler

Bir DLQ, bir hatayı gürültülü olduğu yerden, bloke olmuş consumer'dan, sessiz olduğu bir yere taşır. Bu hem faydası hem tehlikesidir. Stream akar ve başarısız mesaj unutulabilir. Bu bölümde DLQ'nun kendisi olmayan her şey, yani alarmlar, sahip, gözden geçirme ritmi ve retention, bu sessizliğe karşı koymak için vardır.

Tartılması gereken belirli maliyetler:

- **Operasyonel yük.** Birinin triage yapması, karar vermesi ve redrive etmesi gerekir. Kimsenin zamanı olmayan bir DLQ hiç olmamasından kötüdür, çünkü problemi bir mekanizma görüntüsünün arkasına saklar.
- **Veri yönetişi.** Dead letter'lar, bazen kişisel veri içeren production payload'larını olağan yaşam döngüsünün dışında tutar. Erişim kontrollerine, retention kurallarına ve silme yollarına ihtiyaçları vardır.
- **Ordering riski.** Ordered bir stream'den bir mesajı dead-letter etmek onu atlar. Key başına bloklamayı kurmadığınız ya da partition'ı durdurmadığınız sürece sonraki event'ler onsuz uygulanır. Key başına bloklamanın kendisi de bir held-messages deposu, bir redrive bypass'ı, canlı trafikle güvenle yarışması gereken bir serbest bırakma döngüsü ve yarım kalan bir serbest bırakmayı yeniden başlatan bir sweep ekler.
- **Saklanan kimlik.** Güvenli redrive, her kayıta özgün key'i, partition'ı ve consumer input'unu, attempt sayısının yanında da bir `redrive_count` ister. Bu, saklanacak ve tutarlı tutulacak daha fazla şey demektir; o olmadan redrive yalnızca bir tahmin olabilir.
- **Araç yatırımı.** Filtreleme, rate limit'ler, kayıt başına disposition ve audit trail'i olan güvenli bir redrive aracı küçük bir projedir. Onu satın almak ya da kurmak, bir DLQ'ya sahip olmanın maliyetinin parçasıdır.

Ne zaman kullanılmaz: bir mesaj yeniden üretilebiliyorsa ya da kaybı zararsızsa, bir DLQ değerden fazla süreç ekler. Bir cache-warming event'i ya da tamamen kozmetik bir güncelleme, düşürmeleri sayan bir metric ile atılabilir. Ve hataya doğru yanıtın durmak olduğu yerde, ledger örneğindeki gibi, stream'in devam etmesine izin veren bir DLQ aleyhinize çalışır. Bu durumda consumer'ı durdurun, yüksek sesle alert verin ve nedeni düzeltin.

Pattern'in yapabileceklerinin de bir sınırı var. Bir DLQ, görünür biçimde başarısız olan mesajları tutar. Hata fırlatmadan yanlış işlenen mesajlar için hiçbir şey yapmaz; bu, Bölüm 13'ün ve onun reconciliation çalışmasının alanıdır.

Uyarı: Orijinal mesajı, dead letter dayanıklı biçimde saklanmadan önce asla ack etmeyin. DLQ publish'i ack'ten sonra başarısız olursa mesaj hiçbir yerde yoktur ve hiçbir alarm size söylemez.

Sık yapılan hatalar

- **Bağlamsız dead-lettering.** Cazip, çünkü ham body'yi taşımak en kısa koddur. Çözüm, redrive ve teşhis ikisinin de mümkün olması için tam orijinal header'lar dahil, yukarıda anlatılan metadata kayıdır.
- **Kalıcı hataları retry etmek.** Bir parse hatası zamanla düzelmez, ama toptan bir retry politikası ona bir timeout gibi davranır. Hataları sınıflandırın ve kalıcı olanları retry bütçesini harcamak yerine doğrudan DLQ'ya gönderin.
- **Nedeni düzeltmeden redrive etmek.** "Mesajları hareket ettirme" baskısı gerçektir ve sonuç, aynı mesajların DLQ'ya geri dönmesidir. Düzeltmeyi deploy edin, bir örneği redrive edin, sonra geri kalanını.
- **Her şeyi bir anda redrive etmek.** Eski mesajların bir seli canlı trafikle yarışır ve yeni bir incident başlatabilir. Rate-limit uygulayın, batch'leyin ve bir pause anahtarı tutun.
- **Yalnızca derinliğe alarm kurmak.** Bariz metric gibi hissettirir, ama geç tetiklenir. İlk varışa, hıza ve en eski mesajın yaşına alarm kurun.
- **Her şey için tek bir paylaşımlı DLQ.** Oluşturması ucuz ve sahiplenmesi imkânsızdır. Her consumer'a kendi DLQ'sunu verin ve sahibi isimle atayın.

- **Park edilmiş ve incelenmemiş mesajların karışmasına izin vermek.** DLQ asla boşalmaz, alert gevşetilir ve gerçek poison fark edilmez. Sahipleri ve gözden geçirme tarihleri olan ayrı bir parking lot tutun.
- **Paylaşımli topic'e redrive etmek.** "Mesajı geldiği yere geri koymak" gibi görünür ve event'i o topic'in diğer her consumer'ına ikinci kez teslim eder. Başarısız olan consumer'ın kendi input'una, özgün key'le redrive edin.
- **Crash-loop'a giren bir kayıta sonsuza dek beklemek.** Bir log broker'da asla vazgeçmeyen bir consumer ne dead letter üretir ne de lag dışında bir alarm; attempt'leri partition ve offset'e göre kalıcı olarak sayın ki bir restart sayacı sıfırlamasın.
- **Redrive'da ordering'i görmezden gelmek.** Redrive edilen mesaj, aynı aggregate için sonraki event'lerden sonra gelir. Bölüm 9'daki version kontrollerini kullanın ya da state'in ilerlemediğini doğrulayın.

Temel çıkarımlar

- Poison bir message her işlendiğinde başarısız olur; bu yüzden onu sonsuza dek retry etmek, tek bir kötü mesajı arkasında kuyruğa giren her şey için bir outage'a çevirir.
- Başarısız bir mesajı düşürmek ve sonsuza dek retry etmek, ikisi de bir hatayı gizlemenin yollarıdır; dead-letter queue, mesajı bağlamıyla kenara koyar ve stream'in geri kalanı devam edebilir.
- Bir dead letter, orijinal body ve header'ları değiştirilmemiş halde, kaynak konumunu, hatayı, attempt count'ı ve consumer version'ını taşımalıdır, çünkü haftalar sonra onu okuyan kişinin elinde yalnızca o kayıt vardır.
- Orijinali ack etmeden önce DLQ'ya publish edin; böylece bir çökme kurtarılabilir bir duplicate üretir, asla kaybolmuş bir mesaj üretmez.
- Her DLQ'nun adı belli bir sahip ekibe, ilk varışa, hıza ve yaşa alarma ve düzenli bir triage ritmine ihtiyacı vardır; yoksa bir çöplüğe dönüşür.
- Redrive ancak neden düzeltildiğinde, orijinal `event_id` ve key korunduğunda, mesaj başarısız olan consumer'ın kendi input'una gidip asla paylaşımli bir topic'e gitmediğinde, consumer idempotent olduğunda ve mesajlar rate-limited batch'ler halinde bir pause anahtarıyla döndüğünde güvenlidir.

- Ordered bir stream'de bir mesajı dead-letter etmek, partition'ı durdurmak ile bir boşluğun ötesinden devam etmek arasında bir seçim zorunlu kılar; consumer başına bilerek seçin, ledger'lar için durun ve key başına yalnızca mekanizma maliyetine değdiği yerde bloklayın.

Dead-lettering ve redrive, bu bölümün ertelemeye devam ettiği tek bir soruya yaslanıyor: mesajların doğru sırada gelmesi ne demektir ve gelmediklerinde ne kurabilirsiniz? Bölüm 9 bu soruyu doğrudan ele alıyor.

Bölüm 9 — Ordering

Parcelly'de bir perşembe akşamını düşünün. Bir alıcı sipariş veriyor, sonra saniyeler içinde fikrini değiştirip iptal ediyor. Orders önce `OrderPlaced`, sonra `OrderConfirmed`, sonra `OrderCancelled` yayınlıyor; hepsi `order-81724` için ve hepsi doğru sırada. Relay de onları bu sırayla publish ediyor.

Shipping'i besleyen topic varsayılan bir ayarla oluşturulmuş: mesajlar, içeriklerine bakılmaksızın altı partition'a dağıtılıyor. `OrderConfirmed` partition 2'ye, `OrderCancelled` partition 5'e düşüyor. Shipping partition başına bir consumer çalıştırıyor. Partition 5'teki consumer boşta ve hızlı; bu yüzden iptali önce okuyor ve henüz var olmayan bir sevkiyat buluyor. Omuz silkip ack ediyor. Bir an sonra, yavaş bir batch ile meşgul olan partition 2'deki consumer onayı okuyor ve artık var olmayan bir sipariş için bir sevkiyat planlıyor.

Hiçbir şey çökmedi, hiçbir mesaj kaybolmadı ve hiçbir mesaj duplicate olmadı. Her bileşen yapılandırıldığı şeyi yaptı. Pazartesi günü bir carrier, kimsenin sipariş etmediği bir paketi almaya gelecek. Böyle bir sistemin ne kadar ordering'e ihtiyacı var, o nerede kayboluyor ve ona sahip olmadığınızda ne kurabilirsiniz?

Problem ve neden zor olduğu

Order, bir sistemin sahip olduğu ya da olmadığı tek bir özellik değildir. Bir zincirdir ve her halka onu bağımsız olarak kırabilir. Relay, satırları veritabanının commit ettiğinden farklı bir sırayla publish edebilir (Bölüm 4 ve Bölüm 5 bu halkayı ele alıyor). Broker ilişkili mesajları partition'lara ya da queue'lara yayabilir. Consumer'lar paralel işleyebilir. Bir retry bir mesajı ardıllarının arkasına koyabilir (Bölüm 6) ve bir dead letter onu tamamen atlayabilir (Bölüm 8).

Halka	Order nasıl kaybolur	Nereye bakmalı
Producer ve relay	Paralel relay'ler, eşzamanlı commit'ler, commit sırasıyla eşleşmeyen id'ler	Bölüm 4 ve 5
Broker	Farklı partition'lardaki ya da queue'lardaki ilişkili mesajlar	Bu bölüm
Consumer	Bir partition'dan ya da queue'dan gelen mesajları ele alan birkaç worker	Bu bölüm
Hata yönetimi	Retry topic'leri, nack'ler, dead letter'lar, redrive	Bölüm 6 ve 8
Topoloji değişikliği	Partition sayısı değişiklikleri, consumer rebalance'ları	Bu bölüm

Bariz yanıt, global bir ordering talep etmektir: sistemdeki her event tek bir sırada, her consumer tarafından görülsün. Güvenli bir varsayılan gibi geliyor. Pahalıdır ve çoğu sistem için gereksizdir.

Global ordering neden pahalı

Tek bir toplam ordering, tek bir sıra noktası demektir. Log tabanlı bir broker'da bu tek bir partition'dır ve o, bir consumer group içinde aynı anda en fazla bir consumer tarafından okunur. Throughput tavanınız tek bir yazıcının ve tek bir okuyucunun hızı olur. Kapasite eklemek hiçbir şey yapmaz, çünkü kapasite işleri paralel çalıştırmaktan gelir ve paralellik, toplam ordering'in yasakladığı şeydir.

Birkaç servise yayılan bir sistem bunu daha da kötüleştirir. Orders, Payments ve Inventory'nin her birinin kendi veritabanı ve kendi outbox'ı var. Hepsi arasında global bir ordering, her event'i damgalayan bir koordinatör gerektirirdi ve o koordinatör her servisin write path'inde dururdu. Mimarının ortadan kaldırmak için tasarlandığı coupling'i geri satın almış olurdunuz.

Global ordering neden çoğunlukla gereksiz

İş'in gerçekte neye ihtiyacı olduğunu sorun. Parcelly, `order-81724` için `PaymentCaptured`'ın `order-90311` için `OrderPlaced`'dan önce gelip gelmediğini umursamaz. O siparişler birbirini etkilemez. Önemli olan, tek bir sipariş için `OrderPlaced`'ın `OrderConfirmed`'dan önce ve `OrderConfirmed`'ın `OrderCancelled`'dan önce görülmesidir.

Buna **per-aggregate ordering** denir. Aggregate, tanımlayıcısı partition key olarak kullanılan consistency sınırır. Bir aggregate hakkındaki her event üretildiği sırayla görülüyorsa ve farklı aggregate'ler hakkındaki event'ler serbestçe iç içe geçebiliyorsa, çoğu state machine doğru çalışır. Karşılatabileceğiniz ordering budur ve bana göre etrafında tasarım yapmanız gereken de odur.

Per-aggregate ordering'in bile geçmediği bir sınır var. Aynı sipariş için, farklı servislerce üretilen ve farklı topic'lerde taşınan event'lerin birbirine göre ordering'i yoktur. `PaymentAuthorized` ve `StockReserved`, `Orders`'a her iki sırayla da gelebilir ve bu kitap boyunca kullanılan sipariş akışının capture'ı yalnızca ikisi de geldikten sonra istemesinin nedeni tam olarak budur. Paylaşılan bir partition key'i topic'ler arası bir söz olarak görmeyin.

Aggregate başına ordering

Mekanizmayı söylemek basit, sürdürmek ise zorlayıcı: bir key seçin, aynı key'e sahip her şeyi tek bir yola gönderin ve o yolu her seferinde bir mesaj işleyin.

Partition key

Log tabanlı bir broker'da producer, bir partition seçmek için `partition_key`'i hash'ler. Aynı key'e sahip her mesaj aynı partition'a düşer ve bir partition ordered bir dizidir. Bu kitaptaki envelope bu nedenle `partition_key` taşır ve outbox onu saklar; böylece relay, üreten servise sormadan onu set edebilir (Bölüm 4).

Garantinin bir koşulu var: **aynı key, aynı partitioner, aynı partition sayısı**. Partitioner, hash fonksiyonu artı key'in hash'lenmeden önce byte'lara serialize edilme biçimidir. İki producer farklı birini kullanırsa, tek bir `order_id` iki partition'a gider ve key başına ordering, hiçbir yerde hata vermeden kaybolur. Bu, ordering'deki en somut polyglot hatadır. Farklı dillerdeki client kütüphanelerinin hepsi aynı varsayılan hash fonksiyonunu paylaşmaz; bu yüzden bir dilde yazılmış bir servis ile başka bir dilde yazılmış bir relay aynı `order_id`'yi farklı partition'lara gönderebilir. İki savunma işe yarar. Partitioner'ı her producer'da açıkça sabitleyin; aynı algoritmayı ve aynı key encoding'ini kullanın. Ya da kararı producer'ların elinden alın: relay'in partition'ı saklanan `partition_key`'den kendisi

hesaplamasını sağlayın; böylece eşlemeye tek bir kod parçası sahip olur. Her event zaten Bölüm 5'teki outbox relay'inden geçiyorsa ikincisi doğal bir yoldur.

Queue tabanlı broker'lar farklıdır. Tek bir queue üzerindeki competing consumer'lar, birden fazla olduklarında hiçbir ordering garantisi vermez, çünkü broker bir sonraki mesajı hangi consumer boştıysa ona verir. Bazı queue tabanlı broker'lar, bir grup tanımlayıcısını paylaşan mesajları serileştiren bir grupta özelliği sunar; örneğin Amazon SQS FIFO queue'larındaki message group'lar. Diğerleri bir queue'da aynı anda yalnızca tek bir consumer'ın aktif olduğu bir mod sunar. Bu özellikler çalışır ve her biri paralellığa farklı bir biçimde mal olur. Broker'ınızın o özellik altında redelivery ve visibility timeout hakkında ne vaat ettiğini kontrol edin, çünkü garanti çoğu zaman küçük puntolarla gelir. Hangi aileye güvendiğinizi ve o ailenin gerçekten ne vaat ettiğini söyleyin.

Key'i seçmek

Key, aggregate tanımlayıcısıdır ve seçim, consistency'nin nerede yaşadığı hakkında bir ifadedir. Parcelly'nin sipariş yaşam döngüsü için key `order_id`'dir. Bir siparişi state'leri boyunca taşıyan her event onu taşır ve bu, Shipping, Orders ve Ledger'a tutarlı bir sipariş başına sıra verir.

Yanıt bariz olmadığında birkaç soru yardımcı olur:

- Consumer hangi entity'nin state'ini değiştiriyor? O entity'ye göre key'leyin.
- İki event'in çakışabileceği en küçük birim nedir? Aggregate odur.
- Aynı stream, farklı aggregate'lere sahip birkaç servis tarafından mı tüketiliyor? O zaman ya producer en ince ortak olanı seçer ya da her consumer'ın kendi görünümüne ihtiyacı vardır.

Inventory gerilimi gösterir. Stok seviyeleri bir SKU'ya aittir; bu yüzden Inventory'nin kendi command'ları doğal olarak `sku`'ya göre key'lenir. Orders'tan Inventory'ye akan event'ler `order_id`'ye göre key'lenir. Bir rezervasyon event'i her iki tanımlayıcıyı da taşır ve Inventory, SKU başına ordering'e ihtiyaç duyarsa dahili olarak yeniden key'ler. Sistemdeki her topic için tek bir key seçmek cazip gelir ve aggregate'lere sahip olmanın amacını boşa harcar.

Hot partition'lar

Bir partition key, yükü ancak key'ler eşit derecede meşgulse eşit dağıtır. Değiller. Parcelly'de, tek bir satıcının tek bir ürününün bir günlük siparişlerin çoğunu bir saat içinde aldığı bir flash sale düşünün. Inventory kendi topic'ini sku'ya göre key'liyorsa, o tek key trafiğin çoğunu üretir, hepsi tek bir partition'a düşer ve tek bir consumer, diğerleri boşta beklerken bütün grubun işini yapmakla görevlendirilir.

Partition eklemek yardımcı olmaz. Tek bir key her zaman tek bir partition'a eşlenir ve partition eklemek diğer key'leri onun etrafında taşır. Dört dürüst yanıt var:

- **Aggregate'i küçültün.** İş kuralı izin veriyorsa SKU yerine bir rezervasyon id'si gibi daha ince bir şeye göre key'leyin. SKU içindeki ordering, dağılım karşılığında feda edilir ve bu yalnızca hiçbir kural SKU başına ordering gerektirmiyorsa kabul edilebilir.
- **Partition'lara değil consumer'a harcayın.** Her mesajı işlemeyi ucuz hale getirin, böylece tek bir consumer hot key'e yetişebilir.
- **İzole edin.** Bilinen hot key'leri ayrılmış bir topic'e ya da partition'a yönlendirin; böylece partition'ı paylaşan sıradan trafiği aç bırakmazlar.
- **O stream için gevşetilmiş ordering'i kabul edin.** Yükü yaymak için key'e rastgele bir suffix ekleyin ve ordering'i consumer'da onarmak için version kontrollerine (bu bölümde ilerleyen kısımda) güvenin. Bu en esnek ve en çok iş gerektiren seçenektir. Salting'in neyi sağlayıp neyi sağlamadığına dikkat edin: broker üzerindeki yükü yayar, ama version kontrolü key'i consumer'ın veritabanında aggregate'in satır lock'u üzerinde yeniden serileştirir. Throughput, ancak pahalı iş serileştirilmiş adımdan önce yapılıyorsa ya da operasyonlar commutative ise iyileşir.

Seçmeden önce key skew'ini ölçerdim. Sakin bir haftada eşit görünen bir key dağılımı yoğun bir günde kötü biçimde dengesiz olabilir ve ilk işaret genellikle tam olarak bir partition üzerindeki consumer lag'dir (Bölüm 13).

Paralellik ve ordering zıt yönler çeker

Bir consumer group, her partition'ı bir consumer'a vererek ölçeklenir; dolayısıyla partition sayısı paralelliğin tavanıdır. Shipping'in altı partition'ı varsa altı consumer paralel çalışabilir. Yedincisi boşta dururdu.

Ekipler çoğu zaman mesajları bir thread pool'a vererek tek bir consumer içinde throughput'u artırmaya çalışır. Bu zararsız görünür ve ordering'i bozar. Aynı sipariş için art arda okunan iki mesaj iki worker'a verilir ve yavaş olan worker ikinci biter. Partition mesajları sırayla teslim etti ve işleme bunu korumadı.

İkisini birden elde etmenin yolu, key'ler arasında paralelleştirmek ve bir key içinde serileştirmektir. Consumer'ın içinde key'i sabit bir lane'e hash'leyin ve her lane'e tam olarak bir worker verin.

```
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)
```

Tasarımı üç ayrıntı taşır. Birincisi offset'tir. Commit'ler partition başınadır ve contiguousDoneOffset, o partition'da kendi altındaki her mesajın, arada boşluk kalmadan, tamamlanmış olduğu en yüksek offset'tir. Lane'ler

farklı hızlarda bittiğinde en hızlı lane'in offset'ini commit etmek, bir çökmeden sonra daha yavaş bir lane'in bitirmediği mesajları atlardı. Commit, poll loop içinde yapılır; çünkü birçok log client'ının birkaç thread'den çağırılması güvenli değildir. Consumer içi paralelliğin bedeli bu bookkeeping'dir ve bir çökmeden sonraki duplicate'lerdir; bunları Bölüm 7'deki bir inbox halleder.

İkincisi sınırdır. Sınır yoksa yavaş bir lane commit noktasını yerinde tutarken poll loop belleği doldurmaya devam eder ve commit edilmiş offset ile uçustaki iş arasındaki fark, bir çökmeden sonra yeniden teslim edilen mesaj sayısının tam kendisidir. Bir lane high-water mark'ı geçtiğinde partition'ları pause etmek, aşırı yükü görünür ve güvenli olan lag'e çevirir; ikisi de olmayan bellek büyümesine değil. Mekanizma ve pause ile shed arasındaki seçim, Bölüm 6'daki "Aşırı yük altında retry'lar: backpressure" bölümünde anlatılmıştır.

Üçüncüsü hatadır. `handleMessage` başarısız olursa ve lane yerinde retry ederse, lane'i paylaşan her key onun arkasında bekler ve commit edilmiş offset ilerlemeyi bırakır. Mesajı kenara göndermek lane'in ilerlemesine izin verir ve o key'i reorder eder. Bu, Bölüm 6'nın bütün partition'lar için verdiği kararın aynısıdır, şimdi lane başına verilir ve yanıt aynı biçimde consumer'a bağlıdır.

Queue tabanlı bir broker'da aynı fikir bir seviye yukarıda ortaya çıkar. Competing consumer'lar arasında key başına ordering'e ihtiyacınız varsa ya broker'ın grupta özelliğini kullanırsınız ya da queue'nun sıra dışı teslim edeceğini kabul edip bunu consumer'da ele alırsınız. Mesajlar düzenli girdi diye birkaç consumer'ı olan bir queue'nun ordering'i koruduğunu varsaymayın. Korumaz.

Version'larla sıra dışı gelişiyi tespit etmek

Bazen broker, bir retry topic'i, bir redrive, paralel consumer'lar ya da bir partition sayısı değişikliği yüzünden ihtiyacınız olan ordering'i veremez. Bu durumlarda consumer bunu kendisi fark etmelidir. Bunun için event'lerin bir consumer'ın karşılaştırabileceği bir şey taşıması gerekir.

Karşılaştırılacak yanlış şey timestamp'lerdir. Bölüm 3, `occurred_at`'ın tek bir makinenin saatini yansıttığını ve hangi güncellemenin kazanacağına karar vermek için onu kullanmanın, hızlı saati olan bir servisin kaybetmesi

gereken her çağışmayı kazanmasına yol açtığını açıkladı. `occurred_at`'ı insanlar ve analitik için kullanın.

İşe yarayan şey bir **version numarasıdır** (sequence number da denir): aggregate'in her değışiklikte artırdığı, state değışikliğiyle aynı transaction içinde yazılan ve event ile birlikte taşınan bir sayaç. Aggregate'i yalnızca sahibi olan servis değıştirdiğı ve bunu bir transaction içinde yaptığı için version'lar aggregate başına boşluksuz bir dizi oluşturur: 1, 2, 3. Boşluksuzluk, ters yönde de bir kural ister: **her version artışı tam olarak bir yayımlanmış event üretir**. Hiçbir şey yayımlamadan version'ı artıran dahili bir güncelleme kalıcı bir boşluk bırakır ve her consumer var olmayan bir event'i bekler.

Standart envelope'ta bunun için bir alan yok ve tek bir bölüm için envelope'u değıştirmezdin. Onu payload'da, Orders event'leri için `order_version` olarak taşırdım. Bu, aggregate version'ıdır (Bölüm 4); outbox satırının sakladığı sayının aynısı. Envelope tasarımının daha derin tartışmasını Bölüm 11'e bırakıyorum. Sayaç tek bir producer'a ve tek bir aggregate'e aittir. Bir consumer `order_version` değerlerini yalnızca aynı siparişe ait Orders event'leri arasında karşılaştırır, Payments ya da Inventory'den gelen bir event ile asla karşılaştırmaz; onların kendi sayaçlarına ihtiyacı olurdu. Servisler arası bir bağımlılık paylaşılan bir sequence ile çözülmez. Capture adımının hem `PaymentAuthorized` hem `StockReserved`'ı beklerken yaptığı gibi, bir state ya da ön koşul kontrolüyle çözülür. Outbox satırı onu bedavaya alır, çünkü servis yeni version'ı, state değışikliğini ve outbox satırını tek bir transaction içinde yazar.

Consumer'ın kararı

Bir aggregate'in görünümünü tutan bir consumer, uyguladığı son version'ı da tutar. Gelen her event için version'ları karşılaştırır ve üç yoldan birini izler.

Gelen version, son uygulananla karşılaştırıldığında	Anlamı	Aksiyon
Eşit ya da düşük	Duplicate ya da stale	Yok say, say, ack et
Tam olarak bir yüksek	Sıradaki event	Uygula, yeni version'ı kaydet

Gelen version, son uygulananla karşılaştırıldığında**Anlamı****Aksiyon**

Birden fazla yüksek

Bir boşluk: bir şey eksik ya da geç

Event'i buffer'la, uygulama

```
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)
```

İki özellik bunu güvenli kılar. Version kontrolü ve state güncellemesi atomik olarak commit edilir; dolayısıyla aynı key'i ele alan iki worker version 4'ü ikisi birden uygulayamaz. Ve kontrol idempotenttir: bir duplicate, diğerleri gibi stale bir version'dır ve yok sayılır; bu yüzden Bölüm 7'deki inbox'ın yerine geçmez, onu tamamlar. `holdEvent` de idempotent olmalıdır. Zaten tutulan bir version'ın duplicate teslimi aksi halde buffer üzerindeki uniqueness constraint'e çarpar, transaction'ı başarısız kılar, mesajı ack'siz bırakır ve bir poison loop'a dönüşür.

Sıklıkla gözden kaçan iki ön koşul var. Birincisi dizinin başlangıcıdır. Stream'in ortasında başlayan bir consumer, örneğin yeni bir servis ya da daha sonraki bir offset'ten replay, önce version 5'i görür ve 1'den 4'e kadar

olan version'ları sonsuza dek bekler. Görünümünü bir snapshot'tan ya da sahibi olan servisin API'sinden, o snapshot'ın yansıttığı version ile birlikte seed edin ve karşılaştırmaya oradan başlayın.

İkincisi, boşluk tespiti yalnızca consumer her version'ı görüyorsa çalışır. Bir aggregate'in event'lerinin bir alt kümesine abone olan bir consumer, örneğin yalnızca `OrderCancelled`'a, 3 ve 7 numaralı version'ları görür ve 4 ile 6'nın eksik olduğu sonucuna, yanlış biçimde, varır. Böyle consumer'lar için kural "elimdekinden daha yeni olmayan hiçbir şeyi uygulama"ya zayıflar; bu hâlâ stale üzerine yazmalara karşı korur ama boşlukları tespit edemez.

Last-writer-wins tuzakları

En basit çakışma politikası, son yazanın kazanmasıdır. Basittir ve "son"un ne anlama geldiğine bağlı olarak üç farklı tuzak saklar.

Son gelen. Görünüm, consumer'ın en son ele aldığı mesaj tarafından ezilir. `OrderConfirmed`'dan sonra gelen yeniden teslim edilmiş bir `OrderPlaced`, status'u tekrar `PLACED`'e çeker; bu, Bölüm 7'nin işaret ettiği hatadır. Geliş sırası, broker topolojisinin vaat etmeyebileceği tek şeydir.

Timestamp'e göre son. Görünüşte daha iyi, pratikte daha kötü. Servisler arasındaki clock skew, kazanan event'in saati daha hızlı olan olabileceği anlamına gelir. Farklı makinelerde milisaniyeler içinde yazılan iki event'in timestamp sırası anlamsızdır.

Version'a göre son. Bu çalışır ve kendi sınırı vardır. Stale bir event'i düşürmek, yalnızca daha yeni event stale olanın söylediği her şeyi zaten içeriyorsa doğrudur. Bu, state taşıyan event'ler için doğrudur ("sipariş artık `CANCELLED`"). Değişiklik taşıyan event'ler için yanlıştır; örneğin "bakiyeye 4.990 ekle". Stale bir değişiklik event'ini yok saymak değişikliği kaybeder.

Bir açıklama, bunun yukarıdaki tabloyla çelişmesini önler. Boşluksuz version'lar kesin olarak sırayla uygulandığında, daha düşük bir version her zaman bir duplicate'tir, çünkü zaten uygulanmıştır; onu yok saymak delta'lar için bile doğrudur. Delta tuzağı, bir consumer en yeni version'a atlayıp bir boşluğu atlادığında ortaya çıkar; çünkü atlanan değişiklikler o zaman kaybolur.

Bu ayrım, stale ya da sıra dışı bir version'da ne yapacağınızı belirler:

- **State taşıyan event'ler** (tam state ya da öncekinin üzerine yazan bir state): stale event'i yok saymak doğrudur, çünkü sonraki onun yerine geçer.
- **Delta ya da geçiş event'leri** (increment'ler, ledger kayıtları, yerine geçmeyen her şey): onları asla düşürmeyin. Buffer'layıp reorder edin ya da operasyon commutative ise herhangi bir sırada işleyin. Özellikle Ledger, daha eski diye bir kaydı çöpe atmamalıdır.

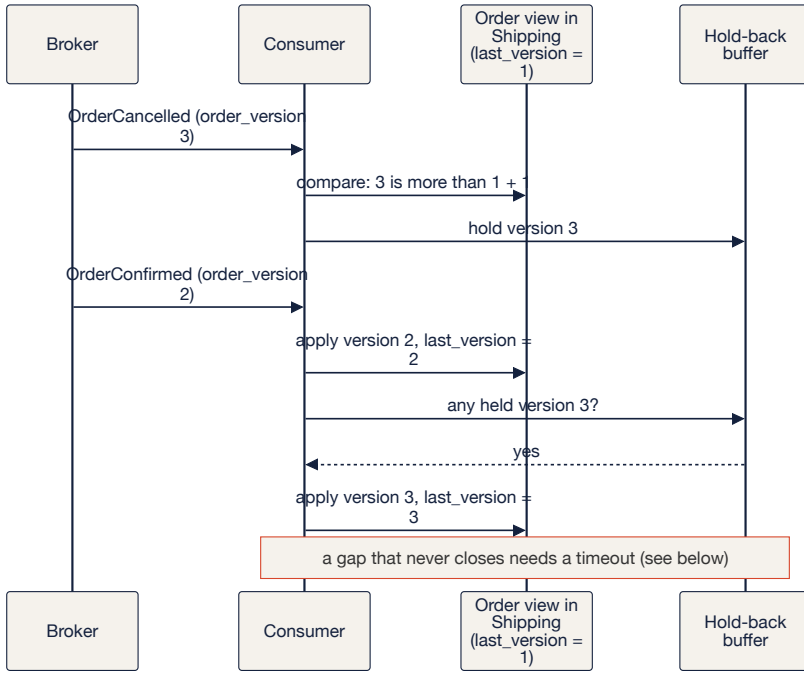
İlgili bir tuzak, silinmiş aggregate'tir. Bir consumer `OrderCancelled`'ı görünce satırını siliyorsa, geç gelen bir `OrderPlaced` satır bulamaz, karşılaştıracak hiçbir şey olmadığı için version kontrolünden geçer ve yeni bir sipariş oluşturur. Diriltme, silmeler etrafında ordering'i yok saymanın karakteristik bug'ıdır. Bir tombstone tutun: key, son version ve bir silinmiş işareti; en azından stale event'lerin hâlâ gelebileceği süre boyunca. Bölüm 8, redrive edilen bir `OrderPlaced` bir iptalden sonra geldiğinde aynı diriltmeyi gösterdi ve onu durduran, version kontrollü bir tombstone'dur.

Son olarak, yok saymak bir iz bırakmayı hak eden bir karardır. Sessizce düşürülen stale bir event, kimsenin göremediği bir bug'dır. Yok sayılan event'leri consumer başına sayın ve key ile version'ları bir log satırında tutun; böylece sayıdaki bir sıçrama, bir müşteri fark etmeden önce görünür (Bölüm 13).

Buffering ve reordering

Consumer bir boşluk gördüğünde ileri gitmek için iki yolu vardır. Event'i yine de uygulayıp yanlış bir state riskini göze alabilir ya da eksik olan gelene kadar event'i geri tutabilir. İkinci seçenek bir **hold-back buffer**'dır.

Aşağıdaki sequence diyagramı Shipping'in `order-81724` için Orders'tan iki event aldığını, version 3'ün version 2'den önce geldiğini gösteriyor; açılış senaryosundaki gibi. İkisi de tek bir producer'dan ve tek bir aggregate'ten geliyor ve version'larını karşılaştırmayı anlamlı kılan da bu. Shipping yalnızca `OrderConfirmed`'a göre hareket eder, ama bunun çalışması için sipariş için bütün Orders stream'ini okur ve başka türlü yok saydığı tipler dahil her event'in version'ını kaydeder. Görünümün `last_version = 1` değerini gerçek kılan ve boşluk tespitini baştan mümkün kılan budur. Version 3'ün nerede beklediğine ve onu neyin serbest bıraktığına bakın.



Serbest bırakma adımı bir döngüdür. Herhangi bir event'i uyguladıktan sonra consumer, buffer'ın sıradaki version'ı tutup tutmadığını kontrol eder, onu uygular ve buffer'da bir ardıl kalmayana kadar tekrarlar.

```

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)
  
```

Buffer'ın kendisi aggregate ve version ile key'lenmiş bir tablodur; dolayısıyla bir restart'tan sağ çıkar. Bellekteki bir buffer süreçle birlikte kaybolur ve broker bu mesajları zaten ack edilmiş saymıştır.

Temizlik için bir kural daha gerekir. `last_version` her sıçradığında, örneğin görünümü getirilen state ile değiştirdikten sonra, `version <= last_version` olan held satırlar geçersizdir ve consumer onları aynı

transaction içinde siler. Aksi halde tabloda sonsuza dek durur ve drain döngüsü onlara hiç ulaşmaz.

Hiç kapanmayan boşluk

Buffering iyimserdir: eksik event'in yalnızca geç kaldığını varsayar. Bazen kaybolmuştur, dead-letter edilmiştir ya da bu consumer'a hiç gönderilmemiştir. Bir sınır olmadan aggregate sonsuza dek bloke kalır ve held küme büyür.

Her buffer'ın bir timeout'a ve süre dolduğunda ne olacağına dair bir karara ihtiyacı vardır. Üç makul yanıt var ve yanılmanın maliyetine göre seçerdim:

- **Eksik state'i getirin.** Sahibi olan servisten bir API aracılığıyla güncel state'i isteyin ve görünümü değiştirin. Bu, producer böyle bir API sunduğunda çalışır ve buffer'ı bir güvenlik ağı olan bir fast path'e çevirir.
- **Alert verin ve bekleyin.** Bir insan bakar. Yanlış bir kaydın yavaş olandan daha pahalı olduğu Ledger için doğrudur.
- **Görünümü şüpheli işaretleyin ve reconciliation'ın onarmasına izin verin.** Stale kalmaya tahammül edebilen read model'ler için doğrudur (Bölüm 13).

Bölüm 8'in key başına bloklaması bu mekanizmanın version'sız biçimidir. Orada bir dead-letter işareti key'i bloke eder, sonraki event'ler bir held store'da bekler ve özel bir bypass redrive edilen event'in geçip onları serbest bırakmasını sağlar. Burada dead-letter edilen bir event bir boşluk yaratır, sonraki event'ler buffer'da bekler ve redrive edilen event onları kendi başına serbest bırakır; çünkü `last_version + 1` olarak gelir ve gerisini drain döngüsü yapar. Version'lar hem işareti hem bypass'ı ikame eder; producer onları sağlayabildiğinde bunun daha güçlü tasarım olmasının nedeni budur.

Maliyetler gerçektir: boyutlandırılacak ve temizlenecek bir buffer tablosu, ayarlanacak bir timeout politikası ve hakkında birinin page edilmesi gereken yeni bir takılı aggregate sınıfı. Event'leriniz tam state taşıyorsa ve consumer'larınız "en yeni kazanır"ı güvenle uygulayabiliyorsa buffer gereksizdir ve onu dışarıda bıraktım.

Retry'lar, dead letter'lar ve ordering

Bölüm 6 ve Bölüm 8'in her biri, buraya bıraktıkları bir ordering sorusuyla bitti. Kısa hali şudur: her hata yönetimi mekanizması bir mesajı zamanda kaydırır ve bu bir reordering'dir.

- Bir log partition'ında **yerinde retry**, ardılları bloke eder; böylece ordering korunur ve bütün partition latency olarak öder.
- **Bir retry topic'i ya da delay queue**, ardılların ilerlemesine izin verir; böylece başarısız mesaj, kendisinden önce gelen event'lerden sonra geri döner.
- Queue tabanlı bir broker'daki **visibility timeout**, timeout'tan sonra bir mesajı queue'ya geri verir; orada kendisinden sonra gönderilmiş mesajlarla yarışır.
- Bir **dead letter** mesajı atlar ve bir **redrive** onu, ardından gelen her şeyden sonra geri getirir.

İkinciden dördüncüye kadar olan seçenekleri atlatılabilir kılan şey version kontrolüdür. Bölüm 6'nın verdiği örneği alın: Inventory, bir sipariş için Orders'tan önce `OrderPlaced`'ı, sonra `OrderCancelled`'ı alıyor ve `OrderPlaced` geçici bir hatadan sonra bir retry topic'ine taşınıyor. Orders'tan gelen aggregate version'ıyla `OrderPlaced` version 1, `OrderCancelled` version 2'dir. İptal önce gelir ve Inventory'nin henüz sipariş için bir görünümü yoktur; bu yüzden `last_version` varsayılan olarak 0'dır ve 2, 0 + 1'den büyüktür. İptal, var olmayan bir rezervasyona karşı uygulanmaz, tutulur. Retry edilen `OrderPlaced` version 1 olarak geldiğinde uygulanır ve drain döngüsü iptali serbest bırakır. Bölüm 6'nın anlattığı başıboş rezervasyon ortaya çıkmaz.

Bu yalnızca version ve buffer tutan consumer'lar için çalışır. Tutmayan bir consumer, Bölüm 6'nın savunduğu gibi, partition'ı bloke etmek ile reordering'e tahammül etmek arasında seçim yapmak zorundadır. Ve Bölüm 8'in ledger'lar için tavsiyesi geçerliliğini korur: geç gelen ya da eksik bir kaydın yanlış bir finansal kayıt demek olduğu bir consumer için durun ve bekleyin. Duraklatılmış bir consumer gürültülüdür; kayıtları sessizce sıra dışı uygulayan bir Ledger değildir.

Uyarı: Version kontrolü olmayan bir consumer ile birleştirilmiş bir retry topic'i event'leri sessizce reorder eder. Hiçbir şey başarısız olmaz, hiçbir alert tetiklenmez ve state yanlıştır. Hataları kenara yönlendiriyorsanız, version kontrolünü aynı değişikliğin parçası yapın.

Partition sayısını değiştirmek

Bir key'in partition'ı, key'in ve partition sayısının bir fonksiyonudur; genellikle sayıya göre bir hash modülüdür. Sayıyı değiştirirseniz fonksiyon değişir.

Örnek olarak, bir key'in 17'ye hash'lendiğini varsayın. Dört partition ile partition 1'e, altı partition ile partition 5'e eşlenir. 20'lik bir hash partition 0'dan partition 2'ye taşınırken 14 ve 15 oldukları yerde kalır. Çoğu key taşınır. Modulo hashing altında bir key, dörtten altıya geçerken yalnızca hash'inin 12'ye göre modülü 0, 1, 2 ya da 3 ise partition'ını korur; yani key'lerin yaklaşık üçte ikisi yeni bir yere düşer. Hata nadir değildir: değişiklik sırasında uçuşta mesajı olan her key'i vurur.

Tehlike, örtüşmedir. Değişiklikten önce yazılan, taşınan bir key'in mesajları eski partition'da, hâlâ tüketilmemiş olarak durur. Değişiklikten sonra yazılan mesajlar yeni partition'a gider. İki consumer artık aynı key'i aynı anda, aralarında hiçbir ordering olmadan işler. Bir consumer ayrıca key başına local state tutuyorsa, örneğin bir cache ya da bir aggregation window'u, o state yanlış instance'ta durur.

Bununla başa çıkma yolları, kabaca tercih sırasıyla:

1. **Başta yeterli partition sağlayın.** Beklenen tepe paralelliğe karşı fazla provision edin; çünkü sayıyı tutmak ucuz, değiştirmek pahalıdır. Birçok log tabanlı broker partition eklemeye izin verir ama kaldırmaya izin vermez; dolayısıyla bir yöndeki hata kalıcıdır.
2. **Değişiklikten önce drain edin.** Producer'ları durdurun ya da stream'i pause edin, consumer'ların eski partition'ların sonuna ulaşmasını bekleyin, sonra sayıyı değiştirin. Order, bir kullanılamazlık penceresi pahasına korunur.
3. **Yeni bir topic'e migrate edin.** Doğru sayıda yeni bir topic oluşturun, consumer'ların yenisine başlamadan önce eskisini bitirdiği bir cutover

yapın ve eskisini emekliye ayırın. En güvenli ve en çok iş gerektiren yoldur.

4. **Version kontrollerine güvenin.** Her consumer sıra dışı event'leri tespit edip buffer'lıyorsa geçiş pause olmadan atlatılabilir ve geçişin kendisi kontrolün çalışıp çalışmadığının da iyi bir testidir.

Rebalance'lar ve revocation

Consumer group rebalance'ları daha küçük ölçekte benzer bir şekilde sahiptir ve her deploy'da olur. Bir partition farklı bir consumer'a taşındığında, eskisindeki uçustaki iş, yenisindeki ilk mesajlarla örtüşebilir. Örtüşmenin ne kadar büyük olacağını, consumer'ın revocation sırasında yaptığı şey belirler.

Revocation sırasında consumer, geri alınan partition'lardan yeni mesaj dispatch etmeyi bırakır, lane'lerin ellerindekini bitirmesine izin verir ya da onu bırakır ve partition'ı teslim etmeden önce contiguous offset'i commit eder. Bir rebalance'ın her partition'ı mı yoksa yalnızca taşınanları mı geri aldığı client'ınızın assignment stratejisine bağlıdır ve hepsini birden alan bir strateji bütün grubu süre boyunca durdurur; hangisini çalıştırdığınızı kontrol edin. Bir partition'ı kaybettiğini geç fark eden bir consumer, yeni sahip aynı işi yaparken zombi gibi işlemeye devam eder; bu yüzden ne yaparsanız yapın bir miktar örtüşme bekleyin.

Inbox ve version kontrolü bu örtüşmeyi yalnızca consumer'ın kendi veritabanı transaction'ı içindeki etkiler için zararsız tutar. İki consumer'ın da yaptığı bir e-posta ya da bir carrier API çağrısı için hiçbir şey yapmazlar. Bu dış yan etkilerin, Bölüm 7'nin anlattığı gibi kendi idempotency key'ine ihtiyacı vardır; çünkü inbox yerel state'i korur, başka hiçbir şeyi değil.

Her Parcellly consumer'ının neye ihtiyacı var

Her consumer'ın aynı ordering'e ihtiyacı yoktur ve onlara aynı davranmak hem bug'ların hem de israfın kaynağıdır. Parcellly için çizilmiş bir tablo, nasıl karar vereceğimi gösteriyor.

Consumer	Ordering ihtiyacı	Mekanizma	Bir mesaj eksikse
Notifications	Yok; her mesaj bağımsız bir olgu	Herhangi bir sıra, key gerekmez	Dead-letter et ve devam et
Orders status view	Sipariş başına, state taşıyan	<code>order_id</code> 'ye göre key'le, stale version'ları yok say	Güncel state'i getir
Inventory reservations	Sipariş başına, geçişler	<code>order_id</code> 'ye göre key'le, version'lar artı hold-back buffer	Key'i tut, alert ver
Shipping	Sipariş başına, <code>OrderConfirmed</code> <code>OrderCancelled</code> 'dan önce	<code>order_id</code> 'ye göre key'le, version'lar artı hold-back buffer	Key'i tut, alert ver
Ledger	Sipariş başına, boşluk yok, düşürme yok	<code>order_id</code> 'ye göre key'le, Payments'ın sahip olduğu sipariş başına sequence'ta boşlukta dur	Bir insanı bekleyerek dur

Ledger satırı sayacını bilerek adlandırır. Ledger, Payments event'lerini tüketir; bu yüzden Orders'a ait `order_version`'i değil, Payments'ın sipariş başına artırdığı bir sequence'ı karşılaştırır. Bir sayaç yalnızca ona sahip olan tek servisin event'leri arasında karşılaştırılabilir.

İlk satır en ucuzudur ve aldığından fazla itibarı hak eder. Ordering'i ele almanın en iyi yolu, consumer'ı bunun önemli olmayacağı şekilde tasarlamaktır. Bir kümeye bir öge eklemek ya da iki değer maksimumunu almak gibi commutative bir operasyon, herhangi bir sırada aynı sonucu verir. Bu tür operasyonlardan kurulu bir consumer'ın key'e, buffer'a ya da version'a ihtiyacı yoktur. Bir handler'ı bu şekilde yeniden şekillendirebildiğinizde, mekanizmaya uzanmadan önce bunu yapardım.

Trade-off'lar ve maliyetler

Paralelliği ordering ile takas ediyorsunuz ve döviz kuru aggregate'tir. Daha kaba bir key daha güçlü ordering ve daha az dağılım verir. Daha ince bir key dağılım ve daha zayıf ordering verir. İkisini birden veren bir key yoktur ve karar business kurallarını bilene aittir.

Hot key'ler bir tasarım maliyetidir, bir operasyon kazası değil. Bir business tanımlayıcısından seçilen bir partition key, business'ın skew'ini miras alır. Bir flash sale bunu ortaya çıkarır. Hafifletebilirsiniz ve o key için ordering'den vazgeçmeden kaldıramazsınız.

Version'lar producer'lardan disiplin ister. Aggregate, version'ı her değişiklik için, her kod yolundan, state değişikliğiyle aynı transaction içinde artırmalıdır. Unutan tek bir yol, consumer'ların sonsuza dek bekleyeceği bir boşluk bırakır. Terside geçerlidir: her artış tam olarak bir event yayımlamalıdır, yoksa dizide hiçbir consumer'ın kapatamayacağı bir boşluk olur. Backfill'ler, admin araçları ve veri migration'ları bunun bozulduğu yerlerdir ve stream'in ortasında başlayan bir consumer, herhangi bir şeyi karşılaştıramazdan önce seed edilmiş bir görünüme ihtiyaç duyar.

Buffer'lar state ve yeni bir hata ekler. Bir hold-back tablosunun boyutlandırılmaya, temizliğe, bir timeout'a, bir sahibe ve bir alert'e ihtiyacı vardır. Aynı zamanda takılı bir aggregate'in saklanabileceği bir yer daha. "En yeni kazanır"ı uygulayabilen consumer'lar için onu atlayın.

Version kontrolleri aggregate'ler ya da topic'ler arasında ordering sağlamaz. Tek bir aggregate'in dizisini korurlar. Bir kural, topic A'daki bir event'in topic B'deki bir event'ten önce geldiğini bilmeyi gerektiriyorsa, bir version numarası bunu sağlamaz. Bu bir workflow problemidir ve Bölüm 10 onu öyle ele alıyor.

Ne zaman uğraşmamalı. Consumer commutative ise, event'ler tam state taşıyor ve "en yeni kazanır" doğruysa ya da geç gelen bir event zararsızsa ve periyodik bir reconciliation onu onaracaksa, mekanizmayı atlayın. Daha azını yapmak maliyeti olan bir karardır ve kayıtlı bir karar olmalıdır.

Sık yapılan hatalar

- **Global ordering istemek.** Güvenli bir varsayılan gibi hissettirir ve throughput'u tek bir okuyucuya sınırlar. Gerçek gereksinimi, per-aggregate ordering'i belirtin ve onun için tasarlayın.
- **Aggregate olmayan bir şeye göre key'lemek.** Event tipine, servise ya da rastgele bir değere göre key'lemek, varsayılan bunu sizin yerinize

yaptığı için olur. Çözüm, consumer'ın state'ini değiştirdiği entity'nin tanımlayıcısına göre key'lemektir.

- **Event'leri sıralamak için timestamp kullanmak.** Doğal version numaraları gibi görünürler. Clock skew ordering'i anlamsız kılar; bunun yerine aggregate'ten bir version taşıyın.
- **Bir consumer içinde key'li lane'ler olmadan paralelleştirmek.** Bir thread pool throughput'u artırır ve partition'ın ordering'ini bozar. Key'i bir lane'e hash'leyin ve her lane'e bir worker verin.
- **Stale delta event'lerini düşürmek.** Bir version kontrolü state taşıyan event'lere uyar. Increment'lere ve ledger kayıtlarına uygulandığında gerçek değişiklikleri çöpe atar. Bir politika seçmeden önce her event'i state ya da delta olarak sınıflandırın.
- **Tombstone'ları unutmak.** İptalde bir satırı silmek, geç gelen bir event'in siparişi yeniden yaratmasına izin verir. Silmeden sonra, stale event'lerin gelebileceği süre boyunca key'i ve son version'ı tutun.
- **Event'leri bellekte tutmak.** Süreçle birlikte ölen bir buffer, broker'ın zaten teslim edilmiş saydığı event'leri kaybeder. Onu kalıcılaştırın, görünümle aynı transaction içinde.
- **Production'da partition sayısını plansız değiştirmek.** Dörtten altı partition'a geçerken key'lerin yaklaşık üçte ikisi için eşleme değişir ve ordering da onunla sessizce değişir. Drain edin, migrate edin ya da version'larla koruyun.
- **Producer'lar arasında partitioner'ları karıştırmak.** Farklı dillerdeki servisler aynı key'i farklı hash'leyebilir. Her yerde tek bir partitioner sabitleyin ya da partition'ı relay'de hesaplayın.
- **Sınırsız lane'ler.** Yavaş bir lane bellek doldururken commit noktası yerinde durur. Her lane'i sınırlayın, partition'ları pause edin ve poll loop'tan partition başına commit edin.
- **Rebalance'ta dış etkiler için inbox'a güvenmek.** Inbox yalnızca consumer'ın kendi veritabanını kapsar. Bir e-posta ya da bir carrier çağrısı kendi idempotency key'ine ihtiyaç duyar.
- **Paylaşılan key'lerin topic'ler arasında event'leri sıraladığını varsaymak.** İki topic'teki aynı `order_id`, aralarında bir sıra vermez. Consumer'ın her ikisini beklemesini sağlayın ya da bağımlılığı bir workflow'da modelleyin.

Temel çıkarımlar

- Global ordering tüm paralelliğinize mal olur ve nadiren gerekir; per-aggregate ordering, tasarlamaya değer olan ordering'dir.
- Aggregate'i tanımlayan bir partition key, log tabanlı bir broker'da entity başına ordered event'ler verir, ama yalnızca her yerde aynı partitioner ve aynı partition sayısı; queue tabanlı broker'lar bir gruplama özelliğine ya da düzensizliğe tahammül eden bir consumer'a ihtiyaç duyar.
- Bir consumer içindeki paralellik key'li olmalı, lane başına bir worker ile, sınırlı ve partition başına commit edilen; yoksa partition'ın teslim ettiği ordering'i sessizce geri alır.
- Aggregate'in kendi transaction'ı içinde artırdığı version numaraları, bir consumer'ın stale, duplicate ve eksik event'leri tespit etmesini sağlar; timestamp'ler bunu yapamaz.
- Stale bir event'i yok saymak state taşıyan event'ler için doğru, buffer'lanması ya da herhangi bir sırada uygulanması gereken delta'lar ve ledger kayıtları için yanlıştır.
- Retry topic'leri, redrive ve partition değişikliklerinin hepsi mesajları reorder eder; bu yüzden ordering'e bağlı her consumer, bunları kullanmadan önce bir version kontrolüne ihtiyaç duyar.
- Her hold-back buffer'ın bir timeout'a, bir sahibi ve bir alert'e ihtiyacı vardır, çünkü hiç kapanmayan bir boşluk takılı bir aggregate'tir; Bölüm 8'in key başına bloklaması aynı fikrin version'sız biçimidir.

Ordering her event'e tek bir aggregate içinde bir yer verir, ama bir business süreci birkaç aggregate'e ve servise yayılır ve adımlarından bazılarının, sonraki bir adım başarısız olduğunda geri alınması gerekir. Bölüm 10, adımların sırasının tasarımın kendisi olduğu ve compensation'ın bir workflow'un nasıl toparlandığı olduğu saga'lara dönüyor.

KISIM IV

İş Akışları ve Kontratlar

Bölüm 10 — Saga'lar ve Uzun Süren İş Akışları

Parcelly'de bir salı öğleden sonrasını düşünün. Bir alıcı seramik bir lamba için sipariş veriyor. Orders `OrderPlaced` yayınlıyor, Payments kartı authorize ediyor, Inventory da satıcının stokundaki son lambayı rezerve ediyor. Sonra kart reddediliyor ve Payments `PaymentFailed` yayınlıyor. Aynı anda ikinci bir alıcı "stokta yok" yazan bir ürün sayfasına bakıyor, çünkü ilk siparişin rezervasyonu lambayı hâlâ tutuyor.

Kimse yanlış bir şey yapmadı. Her servis kendi işini yaptı, her local transaction commit oldu ve her mesaj teslim edildi. Sorun şu: "bu lambayı bu alıcıya sat" diye özetlenen iş operasyonu dört servise ve birkaç dakikaya yayılmış durumda ve onu kapsayan bir transaction yok. Yarıları gerçekleşti. Diğer yarıları gerçekleşmeyecek ve birinin lambayı geri vermesi gerekiyor.

Zamanlama önemli. Henüz hiçbir şey capture edilmedi, dolayısıyla geri alma ucuz: rezervasyonu serbest bırakırsınız ve sipariş sona erer. Aynı lamba için ödeme çoktan alınmış ve hata daha geç gelmiş olsaydı, onu geri vermek farklı ve daha pahalı bir iş olurdu; bölüm bu çizgiye geri dönecek.

Şimdi aynı hatanın, satıcı ilanını iptal ederken, alıcı teslimat adresini düzenlerken ya da capture response'u dönüş yolunda kaybolurken geldiğini düşünün. Bu bölüm, böyle bir akışın temiz biçimde bitmesini ya da temiz biçimde kendini geri almasını sağlayan mekanizmayla ilgili. Üzerinde lock tutamadığınız ve herhangi bir adımın başarısız olabileceği çok servisli bir iş akışını, iş tarafının kabul ettiği bir durumda nasıl bitirirsiniz?

Problem ve neden zor olduğu

Tek bir veritabanının içinde transaction size atomicity verir: ya her değişiklik gerçekleşir ya da hiçbiri. Servisler arasında bu yok. Her servis kendi veritabanına sahip ve message broker aralarında duruyor. Bölüm 4,

tek bir servisin state deęişiklięini ve yayınladıęı event'i nasıl atomik yapacaęınızı gösterdi. Bir sonraki servisin tepkisini ilk servisin deęişiklięiyle atomik yapmadı ve yapamaz.

Eski cevap, genellikle two-phase commit olmak üzere distributed transaction'lardı. Bir coordinator karar verene kadar her katılımcıda lock tutarlar ve coordinator katılımcılar oy verdikten sonra çökerse, geri dönene kadar bu lock'ları tutarak beklerler. Protokolü destekleyen katılımcılar isterler ve en yavaş ya da en az erişilebilir katılımcıyı hepsinin sınırı hâline getirirler. Harici bir ödeme sağlayıcısı ya da bir kargo API'si transaction'ınıza hiç katılmaz. Üçüncü bir tarafa yapılan çağrıyı içeren bir akış için bu seçenek, tartmaya başlamadan önce kapanmıştır.

Saga alternatiftir. Tek bir iş operasyonunu, servis başına bir tane olmak üzere bir local transaction dizisine böler. Bir adım başarısız olduęunda saga rollback yapmaz, çünkü önceki transaction'lar zaten commit olmuştur ve başkaları sonuçlarını okumuş olabilir. Bunun yerine **compensation**'ları çalıştırır: önceki adımları anlamsal olarak geri alan yeni local transaction'lar. Compensation, işi kabul edilebilir bir duruma geri taşıyan ileriye dönük bir eylemdir; zaman makinesi deęildir.

Pratikte üç şey bunu zorlaştırır.

Birincisi, ara durumlar görünürdür. "Stok rezerve edildi" ile "ödeme capture edildi" arasında sipariş ne onaylanmıştır ne de iptal edilmiştir ve onu okuyan herkes bu ara durumu görür. Atomicity gitti; isolation da öyle.

İkincisi, compensation bir mekanizma deęil, bir tasarım kararıdır. Bazı adımların temiz bir tersi vardır, örneğin rezervasyonu serbest bırakmak. Bazılarının yalnızca yaklaşık bir tersi vardır, örneğin capture edilmiş bir ödemeyi iade etmek; bunun maliyeti komisyondur ve günler sürer. Bazılarının ise hiç yoktur: gitmiş bir e-posta gitmiş olarak kalır.

Üçüncüsü, saga'nın kendisi başarısız olabilir. İş akışını yürüten süreç adımlar arasında çökebilir, bir mesaj iki kez gelebilir, bir yanıt hiç gelmeyebilir ve bir compensation da tıpkı bir eylem gibi başarısız olabilir. Bölüm 4'ten 9'a kadar tek tek mesajlar için kurulan her şeyin artık bütün bir akış için geçerli olması gerekir.

Not: Saga benimsemeden önce işe yarar bir test: pivot'tan önceki (sonrasında saga'nın yalnızca ileri gidebildiği adım; bu bölümün ilerisinde tanımlanıyor) her adımı compensation'ıyla birlikte listeleyin ve maliyetini yazın. Pivot'tan sonraki adımların compensation'ı olmaması beklenir. Ondan önceki birkaç adımda "yok" ya da "bir destek bileti" yazıyorsa, bu servisler arasındaki sınır yanlış yerde olabilir ve adımları tek bir servisin local transaction'ına taşımak onları koordine etmekten daha ucuz olabilir.

Parcelly'nin siparişi bir saga olarak

Happy path'teki akışı alın ve adımlarını bir saga olarak adlandırın. Her adım tek bir servisteki bir local transaction'dır ve ardından, aynı transaction içinde o servisin transactional outbox'ına yazılan bir event gelir.

Adım	Servis	Local transaction	Compensation
1	Orders	Siparişi pending durumunda oluştur, <code>OrderPlaced</code> yayınla	İptal edildi olarak işaretle, <code>OrderCancelled</code> yayınla
2	Payments	Kartı authorize et, <code>PaymentAuthorized</code> yayınla	Authorization'ı void et, <code>PaymentVoided</code> yayınla
3	Inventory	Stoku rezerve et, <code>StockReserved</code> yayınla	Rezervasyonu serbest bırak, <code>StockReleased</code> yayınla
4	Payments	Ödemeyi capture et, <code>PaymentCaptured</code> yayınla	Saga içinde yok: pivot burası (iade ayrı bir iptal akışına aittir)
5	Orders	Siparişi onayla, <code>OrderConfirmed</code> yayınla	Yok: başarılı olana kadar retry edilir

İki Payments adımı, kesin anlamda veritabanı transaction'ı değildir. Her biri, sabit bir idempotency key ile harici bir sağlayıcıya yapılan bir çağrı ve ardından sonucun local olarak kaydedilmesidir; bu, Bölüm 7'nin örüntüsüdür. Saga'nın garantileri bu eşleşmeye dayanır ve sağlayıcı çağrısındaki bir timeout, Bölüm 3'ün anlattığı belirsizliği taşır.

2 ve 3. adımlar paralel çalışır ve capture, yalnızca hem `PaymentAuthorized` hem de `StockReserved` geldikten sonra istenir. Shipping `OrderConfirmed`'a tepki verir, Notifications ise Shipping'in

yayınladığına. Bu son ikisini şimdilik saga'nın dışında ele alacağız ve nedenine geri döneceğiz.

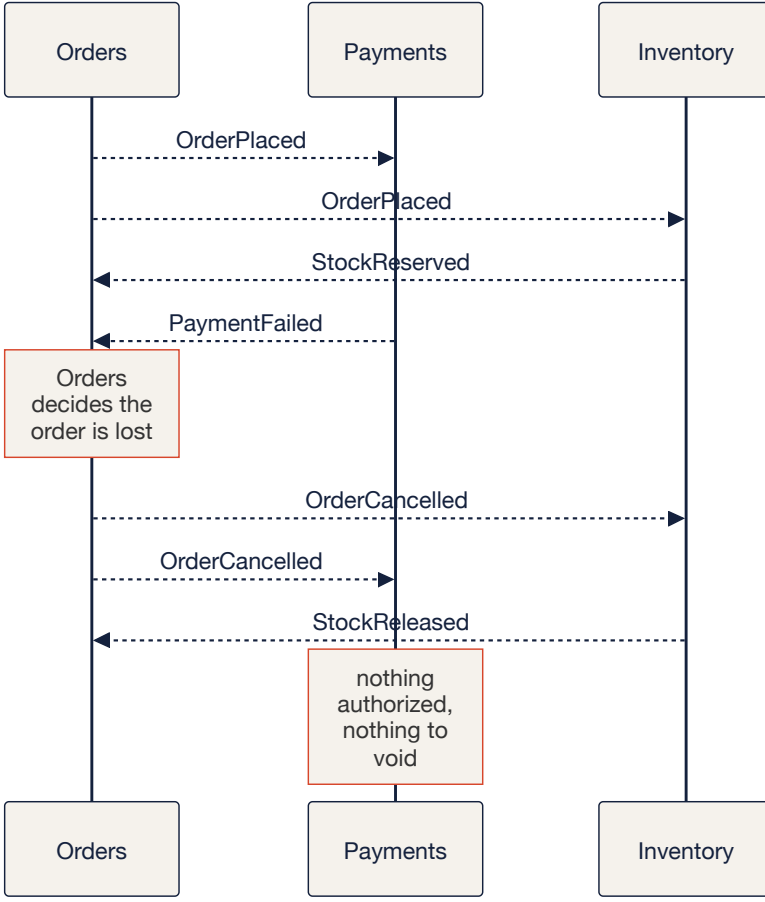
Hata dalları, event akışından gelen tabloyu izler. `PaymentFailed` gelirse hiçbir authorization yoktur; bu yüzden tek compensation, Inventory'nin yapılmış olabilecek ya da olmayabilecek bir rezervasyonu serbest bırakıp `StockReleased` yayınlamasıdır. `StockReservationFailed` gelirse bir authorization tutuluyordur ve hiçbir şey capture edilmemiştir; bu yüzden Payments onu void eder ve `PaymentVoided` yayınlar. İade yalnızca capture'dan sonra ortaya çıkar. Capture'dan önce geri alma bir release ya da void'dir ve buna iade demek ledger'da da alıcının banka ekstresinde de yanlış olur.

Her iki hata dalında da Orders tepki verir ve `OrderCancelled` yayınlar, diğer servisler de buna karşılık compensate eder. Bu cümle, choreography tasarımının tamamıdır. Tamamının istediğiniz tasarım olup olmadığı ise bir sonraki soru.

Choreography

Choreographed bir saga'da merkezi bir sahip yoktur. Her servis event'leri dinler, kendi local transaction'ını yapar ve bir sonraki event'i yayınlar. İş akışı yalnızca bu subscription'ların toplamı olarak var olur.

Aşağıdaki diyagram ödeme hatası dalını bir choreography olarak gösteriyor. Kimin neye karar verdiğine bakın: kimse talimat vermiyor ve her tepki, servisin zaten abone olduğu bir event'le tetikleniyor.



Güçlü yanları gerçek. Servisler gevşek bağlı kalır: Inventory `OrderCancelled`'in ne anlama geldiğini bilir ve Payments hakkında hiçbir şey bilmez. Başarısızlığı her iş akışını durduracak tek bir bileşen yoktur. Bir consumer eklemek, örneğin `OrderConfirmed`'a tepki veren bir analitik servisi, diğerlerinde hiçbir değişiklik gerektirmez.

Zayıf yanları akış büyüdükçe ortaya çıkar. İş akışı tanımı birkaç codebase'deki subscription'lara yayılır ve "StockReserved'dan sonra ne olur?" sorusu tek bir dosyayı okuyarak cevaplanamaz. Döngüsel bağımlılıklar sinsice araya girer: Orders, Payments'ı dinler; Payments de Orders'ı dinler. Her yeni dal event ekler ve her servisin dikkatli bir sırayla güncellenmesi gerekir. Hepsinden önemlisi, "81724 numaralı sipariş şu an nerede?" sorusunun sahibi yoktur. Cevap, birkaç servisin log'ları üzerinde yapılan bir join'dir; bu, Bölüm 1'in adını koyduğu ve Bölüm 13'ün tracing ve reconciliation ile geri döndüğü problemdir.

Choreography'nin akış kısa olduğunda, az dalı olduğunda ve hiçbir adım önceki adımların bir birleşimine bağlı olmadığında iyi çalıştığını görüyorum. İki ya da üç servisin tek bir event'e tepki vermesi doğal bir uyumdur. Parcelly'nin sipariş akışı bu çizgiyi zaten geçti, çünkü capture'ın gelmesi için iki önceki sonucun ulaşmış olması gerekiyor.

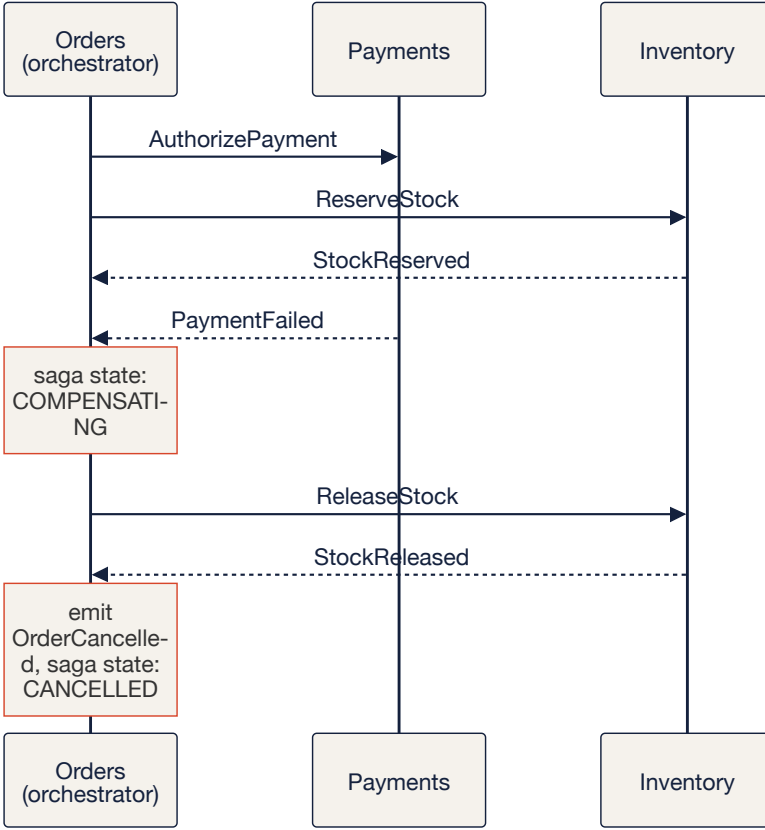
Orchestration

Orchestrated bir saga'da iş akışına tek bir bileşen sahiptir. State'i o tutar, sonraki adıma o karar verir, katılımcılara command gönderir ve onların yanıtlarına tepki verir. Katılımcılar kendi local işlerini nasıl yapacaklarını ve sonucu nasıl bildireceklerini bilir; daha büyük akışın neresinde durduklarını bilmezler.

Bu bölüm, kanonik olanların ötesinde iki command ekliyor:

`AuthorizePayment` (Orders'tan Payments'a) ve `ReserveStock` (Orders'tan Inventory'ye). Onay için command gerekmez, çünkü o orchestrator'ın kendi local transaction'ıdır. Bu kitaptaki bütün command'lar gibi bunlar da `event_id` yerine `command_id` taşır ve yanıtlar sıradan event'lerdir. Choreographed sürümde Payments ve Inventory `OrderPlaced` üzerine başlıyordu; burada onlara orchestrator söylüyor.

Diyagram aynı hatayı daha önceki gibi gösteriyor, ama artık Orders'tan çıkan her ok bir command. Saga state'inin nasıl tek bir yerde yaşadığına ve compensating command'ların geri alınacak işi nasıl tam olarak adlandırdığına bakın.



Orchestrator, "sırada ne var?" sorusunun cevaplandığı yerdir. Bir state machine'dir ve state machine okunabilir, test edilebilir ve çizilebilir. Yeni bir dal tek bir yerdeki değişikliktir. Orchestrator hangi adımların tamamlandığını bilir; bu yüzden compensation, olmuş olabilecek her şeyi değil, gerçekten olanı hedefler.

Maliyetler bunun ayna görüntüsüdür. Orchestrator her katılımcıyı tanır ve bir coupling noktası olur. Dikkatsiz davranılırsa domain mantığı servislerden sızıp onun içine girer ve katılımcılar fazladan adımları olan remote procedure call'lara dönüşür. Ayrıca doğru çalışması, her zaman açık olması ve kendisinin de kurtarılabilir olması gereken bir bileşendir; çünkü state'i kaybolmuş, çökmüş bir orchestrator, kimsenin asla bitirmeyeceği bir saga demektir.

Bence iş sürecine sahip olan servisin içinde yaşayan bir orchestrator en savunulabilir biçimdir. Parcelly'de bu Orders'tır; sipariş yaşam döngüsüne zaten o sahip. Herkesin iş akışlarına sahip olan bağımsız bir orchestration

servisi, her ekibin herhangi bir şeyi yayına almak için değiştirmesi gereken şeye dönüşme eğilimindedir.

Aralarında seçim yapmak

Hiçbir stil genel olarak daha iyi değildir. Seçim, akışın şekline ve onun hakkındaki soruları kimin cevaplaması gerektiğine bağlıdır.

Soru	Choreography	Orchestration
Akış nerede tanımlı?	Subscription'lara yayılmış	Tek bir state machine'de
Servisler arası coupling	Yalnızca event contract'ları üzerinden	Orchestrator her katılımcıyı tanır
Adım ya da dal eklemek	Birkaç servise dokunulur	Orchestrator'a ve katılımcıya dokunulur
Paralel adımlar ve join'ler	Zahmetli; her servis ne gördüğünü takip eder	Doğal; join'i orchestrator tutar
"81724 numaralı sipariş nerede?"	Log'lar üzerinde join	Tek bir satırı oku
Single point of failure	Yok, ama sahip de yok	Orchestrator; kurtarılabilir olmalı
Büyüdüğünde risk	Döngüsel, gizli bağımlılıklar	Domain mantığıyla dolu bir god service
İyi uyum	Kısa akışlar, az dal	Uzun akışlar, koşullar, timeout'lar, join'ler

İkisini karıştırmak normal ve makul. Orchestrated bir sipariş saga'sı yine de `OrderConfirmed`'ı, `Shipping` ve `Notifications`'ın choreographed biçimde tepki verdiği sade bir event olarak yayınlayabilir. Saga, koordineli geri almaya ihtiyaç duyan akış kısmını kapsamalı; bildirimler ve analitik buna ihtiyaç duymaz ve onları içeri çekmek yalnızca state ekler.

Compensation'lar ve geri alınamayan adımlar

Bir adımın compensation'ını, adımı tasarlarlarken tasarlayın; ilk incident istediğinde değil. Her adım için, saga ondan sonra başarısız olursa iş tarafının ne istediğini yazın.

Birkaç kural compensation'ları dürüst tutar.

Compensation, veritabanının geri alınması değil, yeni bir iş eylemidir.

`StockReleased`, rezervasyon satırının geçmişini silmez. Rezervasyonun sona erdiğini kaydeder. `PaymentRefunded`, `capture`'ı silmez; ledger iki kaydı da tutar. Ledger'in append-only olmasının ve compensation'ların orada gerçek event'ler olarak görünmesinin nedeni budur.

Adımları geri almanın ne kadar zor olduğuna göre sıralayın. Geri çevrilmesi kolay adımları öne, zor ya da imkânsız olanları sona koyun. `Authorization` ve rezervasyonu geri almak ucuzdur; `capture`'dan önce gelmelerinin nedeni budur. `Capture`, iadenin para ve zaman maliyeti getirmeye başladığı yerdir. Bu, **pivot step** fikrine götürür: saga'nın artık compensate etmediği ve yalnızca ileri gittiği nokta. `Parcelly`'de pivot `capture`'dır. Ondan önce hata, `release` ve `void` demektir. Ondan sonra onay aşamasındaki bir hata geri çekilme değil `retry`'dir, çünkü iş tarafının elinde alıcının parası vardır ve bir sipariş borcu vardır.

Bazı eylemlerin tersi yoktur. Gönderilmiş bir e-posta, gönderiyi teslim almış bir kargo firması, bir satıcının sisteminin üzerinde işlem yaptığı bir webhook: ardından bir mesaj daha gönderebilirsiniz ama göndermemiş olamazsınız. Bu adımlar pivot'tan sonra yer alır ve başarısızlıkları compensation ile değil, `retry` ve düzeltme ile ele alınmalıdır. `Notifications`'ın tamamen reaktif olmasının ve saga'nın hiçbir zaman parçası olmamasının nedeni de budur: adımları compensate edilemez ve hiçbir şeyi bloklayabilir olmamalıdır.

Compensation'lar başarılı olana kadar retry edilmelidir. Bir eylemin başarısızlığı bir compensation'a yol açar. Bir compensation'ın başarısızlığı, takılı kalmış bir saga ve tutarsız bir sistem dışında hiçbir şeye yol açmaz. `Compensation`'ı, Bölüm 6'nın `retry` disiplini ve Bölüm 8'in `poison message` ele alışıyla eninde sonunda tamamlanması gereken bir şey olarak ele alın. Kalıcı olarak başarısız olan bir compensation, düşürülmemeli; saga state'i kaydedilerek bir parking lot aracılığıyla bir insana ulaşmalıdır.

Compensation'lar idempotent olmalı ve erken gelmeye tolerans göstermelidir. Birincisi Bölüm 7'nin alanıdır: yinelenen bir `ReleaseStock` iki kez `release` etmemelidir. İkincisi daha ince. `Orders`'ın `ReserveStock` gönderdiğini, `timeout`'a düştüğünü ve `ReleaseStock` göndererek compensate etmeye başladığını varsayın. Rezervasyon kaybolmadı, yavaştı ve `release`, `Inventory`'ye ilk ulaşabilir. `Inventory`, bilinmeyen bir rezervasyon için gelen `release`'i no-op sayarsa, geç gelen `ReserveStock`

başarılı olur ve lambda sonsuza kadar tutulur. Güvenli şekil, Inventory'nin release'i siparişe karşı kaydetmesidir; böylece sonradan gelen rezervasyon onu görür ve reddeder. Bazı kaynaklar buna tombstone der. Adı ne olursa olsun, compensation eylemin okuyabileceği bir iz bırakmalıdır.

Saga state'i ve timeout'lar

Orchestrated bir saga'nın kalıcı state'e ihtiyacı vardır ve bu state, onu çalıştıran servisin aynı veritabanında bulunmalıdır; böylece state değişiklikleri outbox ve inbox yazımlarıyla aynı local transaction'a katılır.

```
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
```

state kolonu, adlandırılmış küçük bir konum kümesidir:

AWAITING_RESULTS, CAPTURING, COMPLETED, COMPENSATING, CANCELLED.

Saga satırı ve ilk command'lar tek bir transaction'da yazılır; dolayısıyla kimsenin gözlemleyebileceği ayrı bir "started" konumu yoktur.

authorized ve reserved boolean'ları hangi sonuçların geldiğini kaydeder; join böyle çalışır: ikisi de true olduğunda ve state

AWAITING_RESULTS olduğunda capture istenir. stock_released ve

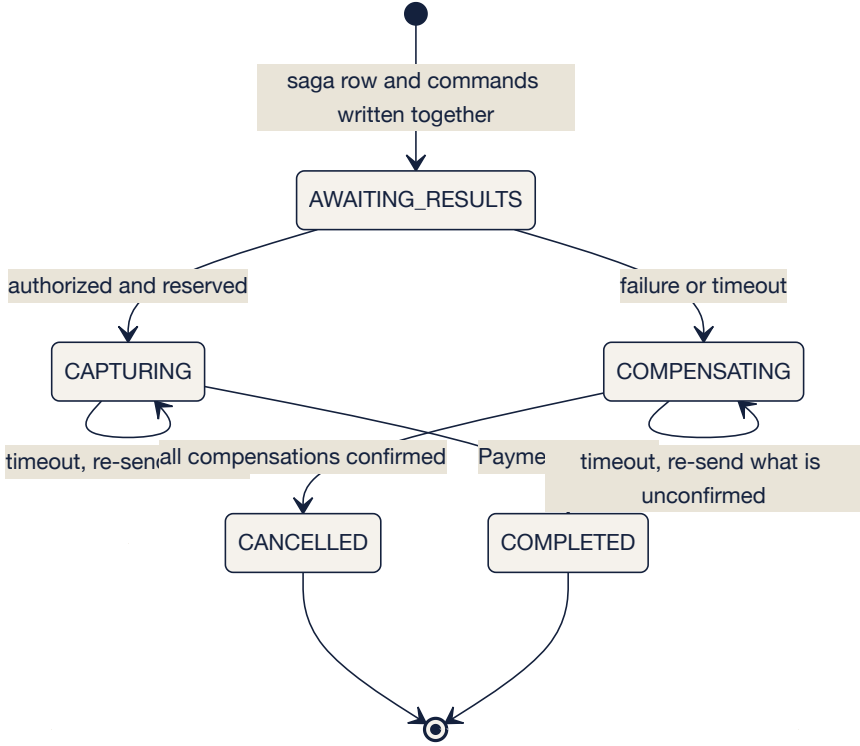
auth_voided flag'leri hangi compensation'ların onaylandığını kaydeder;

böylece "tüm compensation'lar onaylandı" sorusuna satırın kendisi cevap verebilir. Gerekmeyen bir compensation onaylanmış olarak başlar:

PaymentFailed'dan sonra void edilecek bir authorization yoktur, bu yüzden COMPENSATING'e girişte auth_voided true'dur.

order_id üzerindeki bir unique constraint, bir sipariş için iki saga'nın çalışmasını engeller. version kolonu, iki handler'ın aynı satırı aynı anda güncellemesine karşı koruma sağlar ve aşağıdaki handler'lar ona dayanır.

Bir kez çizilmiş bir state machine, herhangi bir düzyazı anlatımdan daha değerlidir. Sonraki diyagram geçerli geçişleri gösteriyor; **COMPLETED**'a çıkmayan çıkışlara ve kendi timeout'unda döngüye giren iki state'e bakın.



CAPTURING'in **COMPENSATING**'e çıkışı olmadığına dikkat edin. Pivot'un state machine'de ifade edilmiş hâli budur. Yanıtı hiç gelmeyen bir capture yeniden gönderilir ve sağlayıcı kalıcı bir ret bildirirse, içinde bir insanın bulunduğu ayrı ve açık bir iş yolu olarak ele alınır.

Döngüler, çıkışlar kadar önemlidir. **CAPTURING** ya da **COMPENSATING** içindeki bir saga bir yanıt bekler ve bir yanıt, her mesaj gibi kaybolabilir. Yalnızca **AWAITING_RESULTS**'in bir deadline'ı olsaydı, diğer ikisinden birinde takılan bir saga'yı uyandıracak hiçbir şey olmadan orada beklerdi. Bu yüzden terminal olmayan her state'in bir deadline'ı vardır ve deadline her geçişte sıfırlanır.

Timeout'lar

Bir saga'daki her bekleyişin bir deadline'ı olmalıdır, çünkü hiç gelmeyen bir yanıt da bir başarısızlıktır ve en çok unutulandır. Bölüm 3, bir timeout'un üç olasılığı nasıl gizlediğini anlattı: eylem başarısız oldu, eylem başarılı oldu ve yanıt kayboldu ya da eylem henüz bitmedi. Saga hangisi olduğunu ayırt edemez; bu yüzden her bekleyiş için yazılı bir politikaya ihtiyacı vardır.

Timeout da bir mesajdır. Bu bölüm onun için tek bir dahili event ekliyor: `SagaTimedOut`; orchestrator'ın kendi timer'ı tarafından yayınlanır. Olağan uygulama, `deadline_at`'i geçmiş saga'ları seçen ve her birine `SagaTimedOut` event'ini diğer tüm girdilerle aynı handler üzerinden besleyen bir timer sürecidir. Böylece timeout, her şey gibi aynı yolu izler: aynı inbox, aynı state kontrolleri ve aynı 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)

```

`AWAITING_RESULTS` dalı, hangi adımın tamamlandığını sormadan compensating command'ları gönderir. Bu bilinçli bir tercih. Timeout anında orchestrator geç gelen bir `StockReserved`'ın yolda olup olmadığını bilmez; bu yüzden hâlâ bekleyen geri almaları ister ve katılımcıların "geri alınacak bir şey yok"u başarı saymasına güvenir. Bunun güvenli olmasının nedeni Bölüm 7'dir.

`armNextDeadline` aynı transaction içinde iki şey yapar. `deadline_at`'i `now() + timeoutFor(saga.state)` olarak ayarlar; böylece terminal olmayan hiçbir state saatsiz kalmaz. Ve `saga.rounds` `MAX_ROUNDS`'u, diyelim beşi, aştığında saga bir zamanlama problemi olmaktan çıkıp

insana ait bir probleme dönüşür: bir alarm yükseltir ve saga state'ini Bölüm 8'in parking lot'una kaydeder; turlar da çalışmaya devam eder. Terminal saga'lara yeni deadline verilmez. Deadline, state değişikliği ve command'lar birlikte commit olduğundan, bir çökme, ilerlemiş ama artık izlenmeyen bir saga bırakamaz.

```
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))
```

CAPTURING içindeki bir saga asla geri döndürülmez. Pivot geçilmiştir; bu yüzden tek hamle, capture'ı aynı key ile yeniden göndermek ve bir insana sormaktır.

Bir timeout'un uzunluğunu sizin yerinize seçemem. En yavaş sağlıklı yolun ne kadar sürdüğüne, bir alıcının ne kadar bekleyeceğine ve stok tutmanın satıcıya neye mal olduğuna bağlıdır. Çok kısa bir değere ayarlanmış bir deadline, yavaşı başarısız hâline getirir ve sistemi compensation'larla boğar. Çok uzun biri, alıcılar stokta yok sayfasına bakarken rezervasyonları tutar.

Geç gelenler için de bir kural gerekir. **PaymentAuthorized**, saga **CANCELLED** olduktan sonra gelirse, orchestrator **VoidAuthorization**'ı yeniden göndermelidir, çünkü biri var olmayan bir sipariş için bir kartta para tutuyordur. Geç gelen event, bir authorization'ın var olduğunu kanıtlar. Timeout'ta gönderilen void hâlâ yolda olabilir, authorization tamamlandıktan hemen sonra işlenmiş olabilir ya da sağlayıcı çağrısı, Payments void'in bıraktığı işareti kontrol etmeden önce çalışmış olabilir. Void'i yeniden göndermek üçünü de çözer.

Bu yeniden gönderme ancak bir tekrar sanılmazsa işe yarar. Kimlikle ilgili iki kural burada buluşur. Tekrarının yeni iş doğurmaması gereken bir command, örneğin **CapturePayment**, saga ve adım için tek bir **command_id** taşır; böylece bir retry tanınır ve ilk sonuçla yanıtlanır. Meşru biçimde yeniden verilebilecek bir compensation ise **command_id**'sini onu

tetikleyen mesajdan alır: saga, adım ve neden olan `event_id`. Yeniden verilen void bu durumda yeni bir command'dır. Ayrıca yalnızca command üzerinde değil, hedef state üzerinde de idempotent olmalıdır; bu yüzden `Payments VoidAuthorization`'ı "81724 numaralı sipariş için canlı hiçbir authorization kalmaz" olarak okur ve ilk denemedeki cache'lenmiş "void edilecek bir şey yok" cevabını yeniden oynatmak yerine bunu kontrol eder. Yanlış state'te gelen event'leri yok sayan bir state machine bunları kaybeder. Bunları açıkça ele alın: her state ve her event için uygulamaya, yok saymaya ya da compensate etmeye karar verin.

Semantic lock'lar

Saga'lar isolation'dan vazgeçtiği için diğer akışlar bir sipariş saga'nın ortasındaiken onu okuyup değiştirebilir. Bir alıcı capture beklemedeyken teslimat adresini düzenleyebilir, bir satıcı iptal edebilir, bir destek çalışanı iade yapabilir. Bölüm 9'un version numaraları eski event'lere karşı yardımcı olur, ama iki akışın uyumsuz iş değişiklikleri yapmasını engellemez.

Semantic lock, veride "bir saga bunun üzerinde çalışıyor" diyen bir flag'dir. Bir veritabanı lock'u değildir. Diğer kodun saygı göstermek üzere yazıldığı bir status değeridir. Parcelly'de sipariş status'ü `PENDING` bunlardan biridir: o geçerliyken sipariş düzenlenemez, gönderilemez ve iade edilemez; sipariş sayfası "işleniyor" gösterir. Aynı şekilde Inventory'deki rezervasyon lamba üzerinde bir lock'tur; ikinci alıcının "stokta yok" görmesinin nedeni budur.

Semantic lock her yolda serbest bırakılmalıdır: tamamlanmada, compensation'da ve timeout'ta. Lock'unu serbest bırakmadan duran bir saga siparişi donmuş bırakır; bu, destek biletleri olarak yüzeye çıkan sessiz bir başarısızlıktır. Yukarıdaki timeout bunun koruyucusudur ve Bölüm 13'ün reconciliation'ı ikinci savunma hattıdır.

Bilmeye değer iki karşı önlem daha var. Biri, mümkün olduğunda güncellemeleri commutative yapmaktır; böylece sıra önemsiz olur. Diğer temkinli bir görünüm tasarlamaktır: örneğin satıcıya bekleyen rezervasyonları zaten düşmüş bir stok sayısı göstermek; böylece kimse tutulmak üzere olan bir şeyi vaat etmez. Hiçbiri, her status için bir saga

sürerken diğer akışların ne yapabileceğine karar verme ihtiyacını ortadan kaldırmaz. O tabloyu, yani status'e göre izin verilen işlemi, her saga tasarımının parçası olarak görüyorum.

Bir saga içinde outbox ve idempotency

Saga handler'ı, önceki bölümlerin buluştuğu yerdir. Bir mesajı tüketir, state'i değiştirir ve bir sonraki command'ı ya da event'i çıkarır. Dikkatsiz yapılırsa, iki kat dual write olur.

İşe yarayan şekil Bölüm 4 ile 7'yi birleştirir. Her handler üç şey yapan tek bir local transaction çalıştırır: mesajı inbox'a kaydeder, saga satırını günceller ve giden command'ları ya da event'leri transactional outbox'a yazar. Relay oradan yayınlar.

```
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)
```

Bu, Bölüm 7'nin yayın yapan consumer'lar için getirdiği kuraldır: inbox satırı, state değişikliği ve outbox insert'i tek bir transaction'da commit olur. Süreç commit'ten önce çökerse hiçbir şey olmamıştır ve broker yeniden teslim eder. Commit'ten sonra ack'ten önce çökerse, yeniden teslimat inbox satırını bulur, ack'lenir ve iş atlanır; eşzamanlı bir duplicate unique constraint'te kaybeder ve aynı şekilde ack'lenir. Eski ya da ilgisiz bir mesaj da kaydedilir ve ack'lenir; böylece asla geri dönmez. Saga, mesajın kaç kez teslim edildiğinden bağımsız olarak mesaj başına bir kez ilerler.

`expected_version` kontrolü, join'i doğru kılan şeydir.

`PaymentAuthorized` ve `StockReserved` aynı anda iki worker tarafından işlenebilir ve her biri iki flag'i de false olan bir satır yükler. Kontrol olmadan ikinci yazım birincisinin üzerine yazar ve saga, timeout sağlıklı bir siparişi compensate edene kadar `AWAITING_RESULTS`'ta bekler. Kontrolle bir transaction kaybeder, inbox satırıyla birlikte rollback olur ve yeniden teslimatı, diğer flag'i zaten taşıyan bir satır yükler. Satırı bir row lock ile yüklemek de aynı sonucu verir.

Saga'nın gönderdiği command'ların kendileri de at-least-once teslim edilecek mesajlardır; bu yüzden katılımcının da aynı korumaya ihtiyacı vardır ve koruma `command_id`'den gelir. Tekrar etmemesi gereken bir eylem için onu saga'dan ve adım adından türetin. Bu, Payments'ın retry edilmiş bir `CapturePayment`'ı tanımasını ve ilk sonuçla yanıt vermesini sağlar. Orchestrator her denemede rastgele bir `command_id` yeniden üretirse, her retry yeni bir iş gibi görünür ve alıcıdan iki kez ücret alınabilir. Yeniden verilebilecek compensation'lar, geç gelenler bölümündeki öbür kuralı izler: id tetikleyen mesajdan gelir ve katılımcı hedef state üzerinde idempotent olur. Envelope'ta taşınan `causation_id` ve `correlation_id` de her saga'yı izlenebilir kılar: `correlation_id` akıştır ve her `causation_id` onun içindeki bir adımdır.

Uyarı: Idempotency key olmadan gönderilen bir compensation, olmadan gönderilen bir ödeme kadar tehlikelidir. Sabit bir key'i olmayan retry edilmiş bir `RefundPayment`, alıcıya iki kez iade yapabilir ve ikinci iade geri gelmeyen paradır.

Workflow engine'lerin yeri

Yukarıdaki her şey bir tablo, bir timer ve birkaç handler ile kurulabilir ve Parcelly'nin sipariş akışı büyüklüğündeki bir akış için ben orada başlardım. Bir noktada mekanizma kodun büyük kısmı hâline gelir: kalıcı timer'lar, adım başına retry'lar, çalışan iş akışlarının versiyonlanması, her instance'ın görsel geçmişi. Workflow engine'lerin ele aldığı problem budur.

Araç sınıfı geniştir. Bazıları, iş akışının ilerleyişini kalıcı kılan ve bir çökmeden sonra sürdüren, ardışık bir program gibi görünen kodla çalışan kütüphaneler ya da servislerdir. Diğerleri, state ve geçişlerin bildirimsel bir tanımını yorumlar. Bulut sağlayıcıları yönetilen sürümler sunar ve birkaç açık kaynak proje vardır. Ürünleri karşılaştırmayacağım, çünkü bu alan bir kitaptan hızlı değişiyor ve seçim dilinize, operasyonel becerilerinize ve çalıştıracığınız bir başka kritik bileşene tahammülünüze bağlı.

Herhangi bir engine'den isteyeceğim şey kısa bir liste. İş akışı state'i nerede yaşıyor ve onu inceleyebiliyor musunuz? Bir adımın yan etkisini ve kendi ilerleyişini nasıl atomik yapıyor, yoksa dual write'ı size mi bırakıyor? Değişmiş bir tanımlı deploy ettiğinizde zaten çalışmakta olan bir iş akışına ne oluyor? Compensation'lar ve timeout'lar nasıl ifade ediliyor? İlerlemeyi saklayan ama adımınızı ayrı bir transaction'da çalıştıran bir engine yine idempotent adımlara ihtiyaç duyar; engine problemi taşır, ortadan kaldırmaz.

Dürüst özet şu: bir engine, state machine'i kurma ve işletme zahmetinden sizi kurtarır; karşılığında size bir bağımlılığa, öğrenilecek bir modele ve çökebilecek yeni bir şeye mal olur. Bir ya da iki küçük saga için bir tablo daha ucuzdur. İnsan adımları ve uzun beklemleri olan birçok uzun süren iş akışı için bir engine kendini amorti eder.

Trade-off'lar ve maliyetler

Bir saga, atomicity'yi availability ve bağımsızlık ile takas eder ve bedel birkaç para biriminde ödenir.

Karmaşıklık. Her saga state'leri çoğaltır. Her biri bir eylem ve bir compensation içeren üç adımlı bir akışın, timeout'lar ve geç gelenlerle birlikte, happy path'in düşündüğünden çok daha fazla ulaşılabilir

state'i vardır ve her biri bir karar ve bir test gerektirir. Bölüm 15, bunların nasıl test edileceğine geri dönüyor.

Görünür ara state'ler. Isolation yoktur. Alıcılar, satıcılar, destek araçları ve diğer servisler yarım kalmış bir sipariş görür. Buna göre semantic lock'lar ve net status'lerle tasarlıyorsunuz ve bazı kullanıcıların yine de görmesini kabul edersiniz.

Bilerek eventual consistency. Bir hata ile son compensation arasında sistem tutarsızdır; saniyeler ya da kötü bir timeout ile saatler boyunca. Bu ancak iş tarafı buna razı olmuşsa kabul edilebilir.

Operasyonel sahiplik. Birinin, deadline'ını aşarak bir state'te takılı kalan saga'lara, tamamlanamayan compensation'lara ve çalışan en eski saga'nın yaşına bakması gerekir. Bir sahip ve bir alarm olmadan saga'lar sessizce başarısız olur.

Compensation bedava değildir. Bir iadenin komisyonu ve gecikmesi vardır, iptal edilmiş bir gönderi ücret doğurabilir ve alıcıya "siparişiniz iptal edildi" mesajı kendi başına görünür bir event'tir. Akışı, yaygın başarısızlığın geri alınması ucuz olacak şekilde tasarlayın.

Saga ne zaman kullanılmaz: adımlar bir local transaction'ı paylaşabildiğinde; hata nadirken ve manuel düzeltme compensation kurmaktan ucuzken, ki bu düşük hacimli bir dahili süreç için meşru bir seçimdir; "iş akışı" aslında e-posta göndermek ya da bir arama index'ini güncellemek gibi koordineli geri alma gerektirmeyen bağımsız tepkilerden ibaretken. Ve compensation'ı tanımlayamadığınızda. Bir adım geri alınamıyorsa ve ertelenemiyorsa, dürüst tasarım sorusu servisler arasındaki sınırın yanlış yerde olup olmadığıdır.

Sık yapılan hatalar

- **Tek bir transaction'ın yeteceği yerde saga kurmak.** Servisler zaten var olduğu ve diyagram modern görüldüğü için cazip. İki adımın sahibi ve veritabanı aynıysa, onları tek bir local transaction'da birleştirin.
- **Happy path'i tasarlayıp compensation'ları sonraya bırakmak.** Sonra, bir incident olarak gelir. Compensation'ı adımın yanına yazın ve yayınladığı event'i adlandırın.

- **Her geri almaya rollback ya da iade demek.** Capture'dan önce compensation bir release ya da void'dir; iade yalnızca capture'dan sonra vardır. Yanlış olanı kullanmak, gerçek state'i ledger'da ve alıcının ekstresinde gizler.
- **Bir saga'nın sonsuza kadar beklemesine izin vermek.** Eksik bir deadline, bir müşteri arayana kadar görünmez. Terminal olmayan her state'teki her bekleyişe, capture yanıtı ve compensation onayları için olanlar dahil, bir timeout ve adlandırılmış bir sonuç verin.
- **Yanlış state'te gelen event'leri yok saymak.** İptal edilmiş bir sipariş için geç gelen `PaymentAuthorized`, tutulan bir karttır. Her state ve event için uygulamaya, yok saymaya ya da compensate etmeye karar verin.
- **Idempotent olmayan compensation'lar.** İki kez işlem yapan retry edilmiş bir release ya da iade, ilkin düzeltirken ikinci bir incident'a yol açar. Tekrar etmemesi gereken eylemlere sabit bir `command_id` verin, yeniden verilebilecek bir compensation'ın id'sini nedeninden türetin ve katılımcıları hedef state üzerinde idempotent yapın. Yeniden gönderilen bir void için tek bir sabit id, "void edilecek bir şey yok" cevabını yeniden oynatır ve authorization'ı tutulu bırakır.
- **Bütün işi orchestrator'a koymak.** Orchestrator sıraya ve sonuca karar vermeli; kuralların sahibi katılımcılar olmalı. Orchestrator stok seviyelerini doğruluyorsa, Inventory'nin işini yapıyordur.
- **Compensate edilemeyen adımları erkene koymak.** Carrier handoff'u capture'dan önce göndermek, bir hatanın size bir gönderi bırakıp para bırakmaması demektir. Adımları, geri alınması zor olanlar sona gelecek şekilde sıralayın.
- **Lock'un serbest bırakılmasını atlamak.** `PENDING` üzerinde konan ve timeout ya da compensation'da hiç temizlenmeyen bir semantic lock, siparişi dondurur. Onu her çıkış yolunda serbest bırakın.

Temel çıkarımlar

- Bir saga, distributed transaction'ı local transaction'lar ve compensation'lar dizisiyle değiştirir ve bunu yaparken isolation'dan vazgeçer.

- Compensation, StockReleased ya da PaymentVoided gibi yeni bir iş eylemidir, rollback değildir ve capture'dan önce asla bir iade değildir.
- Choreography az dallı kısa akışlara, orchestration join'li, timeout'lu ve çok dallı akışlara uyar; ikisi sıklıkla karıştırılır ve orchestrator, sürece sahip olan servisin içinde yer alır.
- Adımları geri almanın ne kadar zor olduğuna göre sıralayın ve compensate edilemeyen ilk adımı, capture gibi, saga'nın yalnızca ileri gittiği bir pivot olarak ele alın.
- Terminal olmayan her state'teki her bekleyişin bir deadline'ı olmalı, her state'in geç gelen ve duplicate event'ler için bir kuralı olmalı ve her compensation idempotent olmalı ve geri aldığı eylemden önce gelmeye tolerans göstermelidir.
- Bir handler'ın içinde tek bir local transaction, inbox'ı, saga state'ini ve outbox'ı birleştirmeli; eşzamanlı handler'lar birbirinin üzerine yazamasın diye bir version kontrolüyle korunmalı ve command_id, capture gibi tekrar etmemesi gereken bir eylem için sabit, yeniden verilebilecek bir compensation için ise tetikleyen mesajdan türetilmiş olmalıdır.
- Bir workflow engine, state machine'in, timer'ların ve geçmişin sahibi olabilir, ama idempotent adımlara duyulan ihtiyacı ve takılı kalan iş akışlarının net sahipliğini ortadan kaldırmaz.

Bir saga, event'lerin olanı her katılımcının okuyup güvenilebileceği bir biçimde anlattığını varsayar. Bunun doğru olup olmadığı, her event'in ne taşıdığına, ne diye adlandırıldığına ve kime ait olduğuna bağlıdır; bu da Bölüm 11'in konusudur.

Bölüm 11 — Event Tasarımı

Parcelly'nin Shipping ekibinin gönderileri daha hızlı planlamak istediğini düşünün. Bugün `OrderConfirmed`'ı alıyor; bu event bir `order_id` taşıyor ve neredeyse başka bir şey taşıyor, sonra teslimat adresini ve sipariş kalemlerini almak için HTTP üzerinden `Orders`'ı çağırıyor. `Orders` pazartesi sabahları yavaş, dolayısıyla Shipping de onunla birlikte yavaşlıyor. Bir tech lead bariz çözümü öneriyor: siparişin tamamını event'in içine koyalım, Shipping'in hiç sormasına gerek kalmasın.

Bir hafta içinde event büyüyor. `Orders` alıcının e-posta adresini, tam adresi, satıcının payout hesabını ve kendi admin ekranının kullandığı `internal_status` adlı bir alanı ekliyor. `Notifications` e-postayı kendi tablosu yerine event'ten okumaya başlıyor. `Ledger` ekibi, payout hesabının artık broker'dan geçtiğini ve uzun bir retention süresi olan bir topic'te durduğunu fark ediyor. Sonra `Orders` bir refactoring sırasında `internal_status`'ü yeniden adlandırıyor ve aynı öğleden sonra üç consumer hata veriyor.

Bu hikâyede kimse önceki bölümlerin anlamında bir güvenilirlik hatası yapmadı. `Outbox` çalıştı, consumer'lar idempotent'ti, `retry`'lar makuldü. Zarar mesajın tasarımından geldi: ne söylediği, ne taşıdığı ve onu kimin değiştirmeye yetkili olduğu. Bu bölüm, bir event'i, ilk consumer onu okuduktan yıllar sonra bile yararlı, güvenli ve değiştirilebilir kalacak şekilde nasıl tasarlayacağınızı soruyor.

Problem ve neden zor olduğu

Bölüm 1, event-driven bir sistemdeki servislerin event'in contract'ı üzerinden, yani adı, alanları ve anlamı üzerinden bağlandığına dikkat çekti. Bu coupling, event'i yazdığınız anda görünmezdir, çünkü o anda bir producer ve bir consumer vardır ve ikisi de birbirine yakın oturan kişilerce yazılmıştır. Contract dahili bir ayrıntı gibi hissettirir.

İkinci bir consumer abone olduğu anda dahili olmaktan çıkar ve producer bunu genellikle geç öğrenir ya da hiç öğrenmez. Log tabanlı bir broker'da yeni bir consumer group kimseye söylemeden topic'i okumaya başlayabilir. Queue tabanlı bir broker'da biri yeni bir queue'yu aynı exchange'e ya da topic subscription'ına bağlar. O andan itibaren payload'daki her alan, hiç tanışmayabileceğiniz insanlara verilmiş bir sözdür.

Pratikte üç şey bunu zorlaştırır.

- **Event'lere eklemek kolay, onlardan almak zordur.** Bir alan eklemek bir satır koda mal olur. Onu kaldırmak, okumadığının repository'lerdeki consumer'lar ve eski geçmiş replay edenler dahil, her consumer'ı bulmayı gerektirir.
- **Tasarım baskısı yanlış yöne işaret eder.** Üreten ekip yayına çıkmak ister, bu yüzden elinin altındaki her şeyi yayınlar. Tüketen ekipler ekstra çağrılardan kaçınmak ister, bu yüzden daha fazla alan ister. Kimseye hayır demesi için para ödenmiyor.
- **Hatalar kalıcıdır.** Bugün yazılmış bir event aylarca bir event log'unda durabilir ve henüz var olmayan bir consumer'a replay edilebilir. Kötü adlandırılmış bir event ya da sızdırılmış bir kişisel alan, onu üreten kod çoktan gittikten sonra da storage'da kalır.

Yani event tasarımı süs değildir. Birkaç saatlik düşüncenin, sonraki her değişikliğin ne kadar pahalı olacağını belirlediği yerdir.

Event'ler, command'lar ve document'ler

İlk tasarım kararı, ne tür bir mesaj gönderdiğinizdir. Üç tür yaygındır ve kuralları farklıdır.

Bir **event**, zaten gerçekleşmiş bir olguyu belirtir: `PaymentCaptured`. Producer kimin dinlediğini umursamaz ve karşılığında bir şey beklemez. Event'ler geçmiş zamanda adlandırılır, çünkü zaten olmuş bir şeyi bildirirler; sonraki bir event onu etkisiz kılabilir ama olmamış yapamaz. Bir consumer bir event'i yok sayabilir ve bu meşru bir tepkidir.

Bir **command**, belirli bir servisten bir şey yapmasını ister:

`CapturePayment`. Tek bir hedef alıcısı vardır ve alıcının harekete geçeceği beklentisi vardır. Bir command reddedilebilir ve gönderen genellikle sonucu önemser. Command'lar emir kipindedir, `event_id` yerine

`command_id` taşır ve asla event olarak adlandırılmaz. Bölüm 10'un ele aldığı orchestrated bir saga, katılımcılarını bu türden command'larla yönlendirir.

Bir **document message**, bir şeyin olduğunu ya da birinin harekete geçmesi gerektiğini söylemeden bir veri bloğu taşır; örneğin bir satıcıdan gelen gece katalog snapshot'ı. Anlamına alıcı karar verir. Document mesajlar meşrudur, ama en sık event kılığına sokulan türdür ve bu kılık karışıklığa yol açar.

Tür	Ne der	Alıcı	Adlandırma	Alıcı reddedebilir mi?
Event	Bir olgu gerçekleşti	Abone olan herkes	Geçmiş zaman: <code>OrderPlaced</code>	Evet, yok sayarak
Command	Şunu yap	Adı belirtilmiş tek bir servis	Emir kipi: <code>CapturePayment</code>	Evet, bir hata yanıtı vererek
Document	İşte biraz veri	Okuyan kim olursa	İsim: <code>CatalogSnapshot</code>	Uygulanamaz

Yaygın hata, event kılığındaki command'dır. Orders'ın emir kipinde adlandırılmış bir `SendConfirmationEmail` yayınladığını ve topic'ten geçtiği için ona event dediğini varsayın. Orders artık Notifications'ın belirli bir biçimde davranmasına bağımlıdır, ama bir reddi duymanın yolu yoktur. Notifications çökmüşse Orders bilmez. İkinci bir servis abone olursa, kendisine hitap etmeyen bir talimatla karşılaşır.

Hızlı bir test uygulardım. Hiç kimse mesajı tüketmese producer'ın hâlâ doğru olup olmayacağını sorun. `OrderPlaced` için cevap evet: sipariş ne olursa olsun verildi. `SendConfirmationEmail` için cevap hayır, çünkü mesaj yalnızca biri harekete geçerse anlam taşır. Bu ikinci tür, alıcının adlandırıldığı ve hatanın göndericiye geri döndüğü bir kanala aittir.

Bunun bir kısmı yön sorusudur da. Event'ler bir olgunun sahibinden ilgilenen herkese doğru işaret eder. Command'lar bir coordinator'dan bir yürütücüye doğru işaret eder. Bir event adı belirli bir servise verilmiş bir talimat gibi duyulmaya başladığında, yön ters çevrilmiştir.

Bir command neye benzer

İki tür farklı şekillerde de başarısız olur ve envelope bunu yansıtmak zorundadır. Kaybolan bir event tasarım gereği sessizdir: producer belirli birinin harekete geçmesini hiç beklemediği için kimse yokluğunu fark etmez. Kaybolan ya da geç kalan bir command farklıdır. Gönderen bir sonuç bekler ve kaybı ortaya çıkaran tek şey kendi deadline'ıdır; Bölüm 10'daki saga'nın terminal olmayan her state'e bir deadline koymasının nedeni budur. Bir command, gönderen vazgeçip compensate ettikten sonra da gelebilir; bu yüzden alıcının onu reddedecek bir yola ihtiyacı vardır.

Bu nedenle bir command, bir event'ten daha küçük ve farklı bir envelope taşır:

```
{
  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`, `event_id`'nin yerini alır ve Bölüm 10 nasıl türetildiğini açıklar: tekrar etmemesi gereken bir eylem için saga'dan ve adımdan; yeniden verilebilecek bir compensation için ise neden olan `event_id` ile birlikte. `reply_to` ve `expects` sonuca geri dönecek bir yol verir; böylece bir ret, göndericinin duyabileceği bir şey olur. `expires_at`, bir alıcının, gönderen beklemeyi bıraktıktan sonra gelen bir command'ı atmasını sağlar. İsteğe bağlıdır, çünkü bir bedeli vardır: alıcının saatiyle karşılaştırılır ve Bölüm 3, o saate saniyesine kadar güvenilemeyeceğinin nedenlerini verdi. Onu bir marjla ayarlayın ve saniyelere karşı değil, saatler öncesinden kalma command'lara karşı bir koruma olarak görün.

Domain event'ler ve integration event'ler

Bir servisin içinde kod çoğu zaman kendi aggregate'leri hakkında event'ler üretir: `OrderLineAdded`, `DiscountApplied`, `AddressCorrected`. Bu **domain event'ler** dahili modeli anlatır ve model değıştikçe değışir. Servisin içinde, örneğin bir read model'i güncellemek ya da aynı transaction içinde bir kuralı tetiklemek için yararlıdır.

Integration event, servis sınırını geçen sürümdür. Bilinçli olarak seçilir, diğer servislerin bilmesi gereken şeye göre adlandırılır ve kararlı tutulur. `OrderPlaced` bir integration event'tir. Order aggregate'inin o anda neye benzediğinin dökümü değildir.

Cazip olan, domain event'leri doğrudan broker'a yayınlamaktır, çünkü mekanizma zaten vardır. Bence bu, dahili yapıyı sızdırmanın en yaygın tek yoludur. Orders ekibi bir aggregate'i ikiye böldüğünde ya da iki kalem türünü birleştirdiğinde ne olacağını düşünün. Domain event'ler public contract olsaydı, iş tarafında hiçbir şey değışmediği hâlde her consumer bozulurdu.

Çözüm, üreten servisteki bir çeviri adımıdır. Domain event'ler değışmekte özgür kalır. Küçük bir mapping fonksiyonu hangilerinin hangi adlarla ve hangi alanlarla integration event'e dönüşeceğine karar verir. Public contract'ın ne olduğuna karar verdiğiniz yer o fonksiyondur ve Bölüm 4'ün transactional outbox'ı onu çağırarak için doğal bir yerdir: outbox satırına yazılan şey integration event'tir.

```

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

```

Dahili ad `OrderSubmitted` ile public ad `OrderPlaced`'in farklı olduğuna dikkat edin. Buna izin var ve sağlıklıdır. Public ad, diğer ekiplerin öğrendiği addır. Dahili ad koda uyabilir.

Thin event'ler ve fat event'ler

Payload ne kadar taşınmalı? İki uç adlandırmaya değer.

Thin event, bazen notification de denir, bir tanımlayıcı ve az bir şey daha taşır: "81724 numaralı sipariş verildi". Ayrıntılara ihtiyaç duyan bir consumer producer'a geri çağrı yapar. **Fat event**, consumer'ların geri çağrı yapmadan harekete geçmesine yetecek kadar state taşır. Bu stile çoğu zaman event-carried state transfer denir, çünkü event bir parça state'i bir servisten diğerine taşır.

Her iki tarafın da, önceki bölümlerin açıklamaya yardımcı olduğu bir hata modu vardır.

Thin event'ler, event'lerin kaldırmak için tasarlandığı senkron coupling'i geri getirir. Shipping ancak Orders o anda erişilebilirse ilerleyebilir; açılıştaki pazartesi sabahı problemi tam olarak budur. Bir de race yaratırlar: consumer geri çağrı yaptığında veri değişmiş olabilir ve

consumer, event'i tetikleyen state'ten daha yeni bir state okur. Geri çağrı başarısız olursa consumer, uzun süre yavaş kalabilecek bir bağımlılığa karşı yapılan retry'ların, yani Bölüm 6'nın dünyasına düşer.

Fat event'ler başka yerlerde bedel ödetir. Payload büyür ve sabit tutacağınıza söz verdiğiniz alan sayısı da. Consumer'lar event'i producer'ın verisinin bir kopyası gibi görmeye başlar ve kopya eskir. Pazartesi günü `OrderConfirmed`'dan adresi cache'lemiş bir consumer, sonraki event'i de okumuyorsa onu cuma günü, alıcı adresi düzelttikten sonra kullanabilir. Ve payload'daki her şey her consumer'a ve saklanan her log segment'ine kopyalanır; bu da hassas alanların riskini yükseltir.

	Thin	Fat
Payload	Tanımlayıcı ve event type	Tanımlayıcı artı consumer'ların ihtiyaç duyduğu state
Consumer, producer'ın ayakta olmasına ihtiyaç duyar	Evet, ayrıntıları almak için	Hayır
Eskime riski	Okuma anında düşük, ama sonraki değişikliklerle race var	Consumer yaşlanan bir snapshot tutar
Contract boyutu	Küçük	Büyük
Gizlilik maruziyeti	Düşük	Daha yüksek
Replay	Producer'ın o günden farklı olabilecek güncel verisine ihtiyaç duyar	Kendi içinde eksiksiz, o gün bilinenleri yeniden üretir

Ben orta bir yola yöneliyorum: **consumer'ın karar verip harekete geçmek için ihtiyaç duyduğunu taşıyın, yalnızca isteyebileceği hiçbir şeyi değil.** `OrderConfirmed` için Shipping'in sipariş kimliğine, paketlemesi gereken kalemlere ve bir teslimat adresine ihtiyacı var; bunlar event'e aittir. Alıcının ödeme yöntemine, satıcının payout ayrıntılarına ya da toplamı üreten indirim kurallarına ihtiyacı yoktur. Sınırı bulmanın yararlı bir yolu, her tüketen ekibe her alanla ne yapacağını sormaktır. Adlandırılmış bir consumer'ı ve belirtilmiş bir kullanımı olmayan bir alan, kaldırma adayıdır.

Burada ikinci bir ayrıntı önemli. Event state taşıyorsa, payload'da bu state'in hangi ana ait olduğunu söyleyin. Aggregate version'ını (Bölüm 4) kullanarak, ki Bölüm 9 bunu Parcelly siparişleri için `order_version` olarak adlandırır, bir consumer elindeki snapshot'ın ikinci bir event'ten daha yeni mi eski mi olduğunu anlayabilir; `occurred_at` yalnızca kaba bir ipucudur, nedeni aşağıda veriliyor. Bölüm 9, sıra dışı gelişi bu tür numaralarla nasıl tespit edeceğinizi ele alıyor; bu yüzden bu bölüm yalnızca version'ı olmayan bir fat event'in orada anlatılan last-writer-wins tuzağına davetiye çıkardığını not ediyor. Bu numarayı envelope'taki `schema_version` ile karıştırmayın: aggregate version'ı tek bir siparişteki değişiklikleri sayar ve her değişiklikte ilerler; `schema_version` ise payload şeklinin breaking version'ını adlandırır ve nadiren ilerler (Bölüm 12).

Seçimin global olması gerekmez. Bir ekip nadiren kullanılan event'ler için thin başlayıp gerçek bir consumer istediğinde alan ekleyebilir. Thin başlayıp büyümek, fat başlayıp küçülmekten daha ucuzdur, çünkü küçülmek breaking change demektir ve Bölüm 12 bunu uzun uzun ele alıyor.

Event'leri adlandırmak

Kitap boyunca kullanılan Parcelly event adları küçük bir kural kümesini izler ve bu bölüm onları gerekçeleriyle birlikte belirtiyor.

- **Geçmiş zaman**, çünkü event bir olgudur: `PaymentCaptured`; command olan `CapturePayment` değil.
- **Önce özne, PascalCase**: önce aggregate gelir, sonra olan şey. `ShipmentDispatched`; `DispatchedShipment` değil. Alfabetik sıralandığında, bir aggregate hakkındaki event'ler katalog ve dashboard'larda bir arada toplanır.
- **Teknik dil değil, iş dili**. `OrderConfirmed`, bir product owner'ın anladığı bir şey söyler. `OrderRowUpdated`, bir tablo hakkında bir şey söyler.
- **Adda teknoloji ve consumer olmaz**. `OrderPlacedKafkaEvent` broker'ı sızdırır. `OrderPlacedForShipping` bir consumer'ı sızdırır ve sonraki consumer event'in kendisi için olup olmadığını merak etmek zorunda kalır.

- **İki servis aynı isim hakkında olgu yayınlamadıkça servis öneki olmaz.** Orders ve Shipping'in ikisinin de bir `DeliveryDelayed` bildirebildiğini varsayın: biri satıcının paketi teslim etmediği, diğeri kargonun geciktiği anlamına gelir. Consumer'lar bunları addan ayırt edemez; bu yüzden nitelemek gerekli olur (`OrderDeliveryDelayed`, `ShipmentDeliveryDelayed`). Örnek açıklayıcıdır; bu kitaptaki Parcelly servislerinin ikisi de bunu yayınlamaz. Aksi halde producer zaten `producer` alanında görünür.

En önemli adlandırma kuralı, anlamla ilgili olandır. `OrderUpdated`, olgu testinde başarısız olan bir addır: bir şeyin değiştiğini söyler, başka bir şey söylemez. Bir consumer ne olduğunu öğrenmek için yeni state'i eskisiyle karşılaştırmak zorundadır ve değişmiş bir adresi değişmiş bir toplamdan ayıramaz. İş olgusunu belirten adları tercih edin: `OrderCancelled`, `ShipmentAddressCorrected`. Bir iş adı bulamadığınızda, bu çoğu zaman event'in bir iş değişikliğini değil bir veritabanı değişikliğini anlattığının işaretidir.

Bir başka kural kararlılıktır. Consumer'lar bir event adına bağımlı olduktan sonra onu yeniden adlandırmak, bir alanı kaldırmakla aynı maliyette bir breaking change'dir. İlk sürümden önce ad için fazladan 10 dakika harcayın.

Envelope

Bu kitaptaki her event aynı envelope'u taşır: `event_id`, `event_type`, `schema_version`, `occurred_at`, `producer`, `correlation_id`, `causation_id`, `partition_key` ve `payload`. Envelope, mesajın tooling'in iş mantığını anlamadan güvенеbileceği kısımdır. Bir relay, bir dead-letter handler'ı, bir tracing sistemi ve bir replay aracı hepsi payload'u parse etmeden onunla çalışabilir.

Her alan, bir bileşenin sorduğu bir soruyu cevaplar:

Alan	Cevapladığı soru	Kim sorar
<code>event_id</code>	Bu event'in tam kendisini daha önce gördüm mü?	Inbox, deduplication (Bölüm 7)
<code>event_type</code>	Payload'u nasıl yorumlarım?	Routing, deserialization

Alan	Cevapladığı soru	Kim sorar
schema_version	Payload şeklinin hangi breaking version'ına sahip?	Consumer'lar, schema registry (Bölüm 12)
occurred_at	Olgu, producer'ın gördüğü şekliyle ne zaman gerçekleşti?	Reconciliation, analitik, debugging
producer	Bu olgunun sahibi hangi servis?	Sahiplik, destek, soruların yönlendirilmesi
correlation_id	Bu hangi iş akışına ait?	Destek, reconciliation, log araması
causation_id	Buna doğrudan ne yol açtı?	Zincirin yeniden kurulması
partition_key	Bu event hangi aggregate ile ilgili?	Broker, ordering (Bölüm 9)
payload	Ayrıntılar neler?	İş mantığı

İki noktaya dikkat etmek gerekir. `occurred_at` producer'ın saatidir ve Bölüm 3, event'leri neden ona göre sıralamamanız gerektiğini açıkladı. Olgunun kabaca ne zaman gerçekleştiğini söyler ve bir sıra işlevi görmez. `partition_key` aggregate tanımlayıcısıdır ve seçimi Bölüm 9'un konusuydu.

`correlation_id` bir trace id değildir. Trace context (W3C Trace Context) envelope'un yanında header'larda taşınır ve Bölüm 13, bunun outbox'a nasıl yakalandığını gösterir. Trace'ler örneklenir ve süresi dolar; `correlation_id` ikisinden de sağ çıkan iş anahtarıdır ve bir destek mühendisinin arama yaptığı anahtardır.

Envelope'un diller arası hâli

Parcelly'nin servislerinin hepsi tek bir dilde yazılmamıştır. Orders ve Payments JVM üzerinde çalışır, Notifications Node'dur ve Ledger ekibinin reconciliation job'ları Python'dır. Yukarıdaki alan listesi envelope'un neleri içerdiğini söyler. Her alanın wire üzerinde nasıl yazıldığını söylemez ve ikinci bir dilin, birincide sorunsuz görünen bir contract'ı bozduğu yer burasıdır.

Alan	Wire tipi	Kural
event_id, correlation_id, causation_id, partition_key	string	Tanımlayıcılar, sayısal görünseler bile string'dir. JSON number olarak taşınan 64-bit bir id, bir JavaScript consumer tarafından sessizce yuvarlanır

Alan	Wire tipi	Kural
<code>event_type</code>	string	<code>PaymentCaptured</code> gibi kararlı, dilden bağımsız bir ad; asla bir class ya da package adı ve asla bir framework'ün varsayılan olarak doldurduğu bir type header'ı değil
<code>schema_version</code>	integer	Küçük ve her dilde bir integer
<code>occurred_at</code>	string	UTC'de, Z son ekiyle RFC 3339; milisaniye hassasiyetinde sabit, offset yok. Parser'lar geri kalan her şey konusunda anlaşmaz
payload içindeki para	integer ya da string	Belirlenmiş bir aralık içinde integer minor unit'ler (<code>total_minor</code>) ya da bir decimal string. Asla float değil. JavaScript number'ları yalnızca $2^{53} - 1$ 'e kadar kesindir; bu yüzden bunu aşabilecek bir alan string olarak taşınır
Enum benzeri alanlar	string	Kümeyi açık sayın: bir consumer, görmediği bir değeri ele alabilmelidir (Bölüm 12)

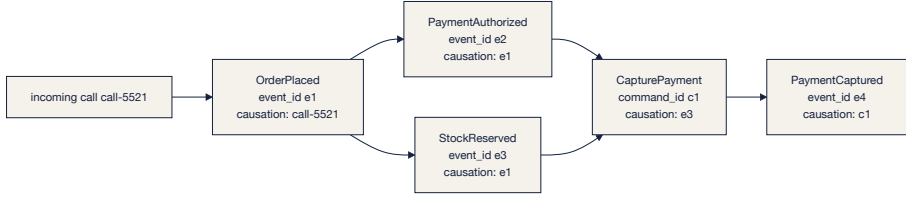
İki yerleşim kuralı contract'ı tamamlar. Envelope mesaj gövdesinde yaşar ve broker header'ları yalnızca gövdenin zaten söylediğini yansıtır; çünkü header'lar, bir mesaj kopyalandığında, dead-letter'a alındığında ya da başka bir broker'a köprülendiğinde düşen kısımdır. Ve content type mesaj başına tahmin edilmez, schema'da bir kez bildirilir. Bir consumer'ın tanımadığı alanlara ne olacağı ve formatların bunları çözümlemede nasıl farklılaştığı bir compatibility sorusudur ve Bölüm 12'ye aittir.

Correlation ve causation

Okurların en sık karıştırdığı iki tanımlayıcı `correlation_id` ve `causation_id`'dir. Farklı soruları cevaplarlar.

`correlation_id`, bir iş akışındaki her event tarafından paylaşılır. Bir alıcı 81724 numaralı siparişi verdiğinde, sonrasında gelen her şey `correlation_id: "order-81724"` taşır: authorization, rezervasyon, capture, gönderi. Bir alıcı yazıp ne olduğunu sorduğunda onunla arama yaparsınız.

`causation_id`, buna doğrudan yol açan tek mesajı işaret eder. Bir event'in `event_id`'si ya da bir command'ın `command_id`'sidir. Akış boyunca tekrarlanmaz. Her sıçrama onu kendi doğrudan tetikleyicisinin tanımlayıcısına ayarlar ve zinciri geriye doğru izlemek yolu yeniden kurar.



Son iki adıma bakın. `CapturePayment` iki event tarafından tetikleniyor ve `Orders causation_id` için birini seçmek zorunda; burada koşulu tamamlayıcı. Belirsizlik gerçek ve bütün producer'ların aynı şekilde seçmesi için kuralı yazardım. `PaymentCaptured` event'i kendinden önceki event'i değil command'ı adlandırır; envelope'un o alanda her iki tür tanımlayıcıya da izin vermesinin nedeni budur.

Akıştaki her event `correlation_id: "order-81724"` paylaşır. Bir akıştaki ilk event, correlation tanımlayıcısını ya aggregate'ten türeterek ya da gelen request'ten taşıyarak belirler. Sonraki event'ler onu değiştirmeden kopyalar.

Bu kitap boyunca aggregate kapsamlı `order-81724`'ü kullanır ve bu, sonucu olan bir seçimdir. Aggregate kapsamlı bir id basittir, ama o siparişteki her akış onun altında birleşir: üç hafta sonraki bir iade, asıl checkout ile aynı id'yi paylaşır ve ikisi birbirinden ayırt edilemez. Alternatif, her akış için yeni üretilen **flow kapsamlı** bir id'dir (`checkout-81724-1`, `refund-81724-1`) ve `order_id` payload'da aranabilir tutulur. Üretilmesi ve taşınması gereken bir şey daha demektir ve bir aggregate'in, birincisinden ayırt etmeniz gereken ikinci bir akış başlatabildiği durumlarda bu bedele değer. Bu bölümün geri kalanı ikisinde de geçerlidir. Kopyalamak için tamamıdır ve kopyalamayı unutan bir consumer, aşağı akıştaki her şey için izi koparır.

```
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
    }
```

Tetikleyicinin `partition_key`'i yalnızca iki mesaj da aynı aggregate ile ilgiliyse kopyalanabilir. Siparişin key'ini devralan, bir ödeme ya da bir gönderi hakkındaki bir follow-up, yanlış stream'e göre ordering'e tabi olurdu (Bölüm 9). Parcellly neredeyse her şeyi sipariş üzerinden key'ler; hatanın fark edilmesinin zor olmasının nedeni budur.

Bunu yanlış yapmanın bedeli en kötü zamanda ödenir. Bir incident sırasında soru "bu siparişe beş serviste ne oldu?" olur. Tutarlı bir `correlation_id` olmadan bir mühendis beş log'u elle arar. Bölüm 13, tracing ve reconciliation'ın bu alanları nasıl kullandığını gösteriyor; bu yüzden bu bölümün işi, onların ilk event'ten itibaren var ve doğru olduğundan emin olmaktır.

Bir event'te yeri olmayanlar

Bir event, producer'ının veritabanından çıkar ve farklı erişim kuralları, farklı retention ve çok sayıda okuyucusu olan bir sisteme girer. Dolayısıyla tasarım sorusu, o sisteme neyi kopyalamaya razı olduğunuz sorusudur da.

Kişisel veri. Bir e-posta adresi, telefon numarası ya da tam teslimat adresi içeren bir event, broker'ın depoladığı her yerde, belki retention penceresi boyunca saklanır ve her consumer tarafından okunur. Bir event log'unda, bir kişinin verisini sonradan silmek zahmetlidir, çünkü log append-only'dir ve consumer'lar alanları zaten kopyalamıştır. Bazı ekipler şifreleme şemalarıyla ya da ayrı topic'lerle karşılık verir. Ben daha erken başladım: event'e `buyer_id` gibi bir tanımlayıcı koyun ve meşru ihtiyacı olan bir

consumer'ın kişisel alanları o anda, sahibi olan servisin erişim kurallarına tabi olarak o servisten almasına izin verin. Bu, hassas alanlar için thin event'lere doğru eğilir ve bence bu takas değer. Bir fat event gerçekten bir adrese ihtiyaç duyuyorsa, Shipping'in duyduğu gibi, event onu bilerek, o kitleye taşınmalı ve retention süresi onun orada olduğu bilinerek seçilmelidir.

Hangi alanların kişisel veri sayıldığı ve bunun hangi yükümlülükleri getirdiği, yargı bölgenize ve hukuk danışmanlarınıza bağlıdır. Sizin kısıtlarınızı bilmiyorum. Tasarım alışkanlığı her yerde aynıdır: bir event'teki her hassas alanın adlandırılmış bir nedeni olmalıdır.

Secret'lar ve kimlik bilgileri. Token'lar, anahtarlar ve payout hesap numaraları bir payload'a ait değildir. Bir mesaj log'lanır, dead-letter hedeflerine kopyalanır, admin araçlarında gösterilir ve biletlere eklenir. Her payload'un bir insan tarafından okunacağını varsayın.

Dahili ayrıntı. Tablo adları, dahili status kodları, feature flag'ler, yalnızca bir servisin içinde anlam taşıyan veritabanı id'leri ve anlamı producer'ın koduna bağlı değerler. Açılış senaryosundaki `internal_status` örnektir. Böyle bir alan, biri onu okuduğu anda public bir sözdür.

Belirli consumer'lara talimatlar. `skip_email_if_gift` gibi bir flag Notifications'a nasıl davranacağını söyler. Event olguyu belirtmeli, consumer onunla ne yapacağına karar vermelidir.

Uyarı: Bir event log'una giren her şey kalıcı ve geniş çapta okunabilir sayılmalıdır. Bir kez yazılmış bir secret ya da kişisel alan dead-letter hedeflerine, backup'lara ve replay araçlarına kopyalanır ve onu her yerden kaldırmak, dışarıda bırakmaktan çok daha zordur.

Yeni bir event'i gözden geçirirken kısa bir kontrol listesi yardımcı olur: her alanın ona ihtiyaç duyan bir consumer'ı var mı, her alan producer'ın kararlı tutmaya razı olduğu bir şey mi ve alan, görmemesi gereken biri tarafından okunursa birine zarar verebilir mi?

Granularity

Bir event ne kadar büyük olmalı? İki zıt hata var.

Çok ince. Her kolon güncellemesi için bir `FieldChanged` event'i yayınlamak bir gürültü akışı üretir. Consumer'lar anlamı birçok mesajdan yeniden birleştirmek zorunda kalır ve o mesajların sırası, Bölüm 9'un uyardığı biçimlerde önem kazanmaya başlar. Tek bir iş olgusu olan bir değişiklik, alıcının bir adresi düzeltmesi, artık yarı uygulanmış bir durumda görülebilen dört event olarak gelir.

Çok kaba. Her değişiklikte tetiklenen ve siparişin tamamını taşıyan bir `OrderChanged` event'i, her consumer'ı yeni state'i eskisiyle diff'lemeye ve iş anlamının ne olduğunu tahmin etmeye zorlar. Ayrıca her consumer'ın, umursamadıkları değişiklikler dahil, her değişiklikte uyanması demektir.

Birim olarak iş olgusunu kullanırdım: iş tarafında birinin adlandıracağı her karar ya da olay için bir event. Bir alıcının sipariş vermesi tek bir olgudur. Bir ödemenin capture edilmesi bir başkası. Tek bir transaction birkaç olgu ürettiğinde, aynı transactional outbox yazımı üzerinden birkaç event yayınlayın; böylece birlikte commit edilir ya da rollback edilirler.

Pratik bir test, bir consumer'ın bu event'e tek başına anlamlı biçimde tepki verip veremeyeceğini sormaktır. Cevap "yalnızca sonraki üç event geldikten sonra" ise granularity çok incedir. Çoğu consumer payload'un çoğunu yok sayıyorsa ya da event adının tarif edilmesi için "ve" gerekiyorsa, çok kaba olabilir ve iki olguyu gizliyor olabilir.

Sahiplik ve public contract

Her event'in tam olarak bir sahibi vardır: event'in tanımladığı aggregate'e sahip olan servis. `PaymentCaptured`'a `Payments` sahiptir. `OrderPlaced`'a `Orders` sahiptir. Consumer'lar event'i okuyabilir, ama yalnızca sahibi onu tanımlar. Farklı bir şekil isteyen bir consumer sahibine sorar ve aynı ad altında kendi varyantını yayınlamaz.

Bir event'i **public contract** olarak ele almanın pratik sonuçları vardır ve ben onları ekibin çalışma anlaşmalarına yazardım.

- **Sahibi event'i doküman eder:** adını, anlamını, ne zaman yayınlandığını, ne zaman yayınlanmadığını, her alanı tipi ve anlamıyla ve ordering ile partition key hakkındaki garantileri.
- **Event'in bir version'ı ve bir değişiklik süreci vardır.** Bölüm 12, schema'ların consumer'ları bozmadan nasıl evrilebileceğini anlatıyor.

Temel kural buraya aittir: sahibi, yayınlanmış bir event'i okuyucuları için sonuçlarını bilmeden değiştirmez.

- **Consumer'lar kendilerini bildirir.** Kimin neyi tükettiğinin kısa bir kaydı, sessiz bir subscription'ı bilinen bir bağımlılığa dönüştürür. Gösterişli olması gerekmez. Bir pull request'in güncellemek zorunda olduğu repository'deki bir liste, başlamak için yeterlidir.
- **Consumer'lar yalnızca ihtiyaç duyduklarını okur.** Bölüm 12'deki tolerant reader kuralı buradan başlar: kullanmadığınız alanları yok sayın; böylece additive değişiklikler sizi bozmaz.
- **Yayınlama, davranışın parçasıdır.** Bir kod yolunda `OrderCancelled` yayınlamayı bırakan bir servis, bir HTTP endpoint'ini kaldıran biri kadar public arayüzünü değiştirmiştir.

"Contract" ve "sahip" sözcüklerinin ikisi de ağırlık taşır. Bir contract test edilebilir. Bir sahibe page atılabilir. Bir event'in kime ait olduğunu kimse söyleyemiyorsa, onu değiştirmeye kimin yetkili olduğunu da kimse söyleyemez ve açılış senaryosu tekrarlanır.

Trade-off'lar ve maliyetler

Disiplin hıza mal olur. Domain event'leri integration event'lere map etmek, dikkatle adlandırmak, dokümantasyon yazmak ve bir consumer listesi tutmak, ilk sürümü yavaşlatır. Karşılığı daha sonra, daha ucuz değişiklikler olarak gelir. Aynı kişi tarafından yazılmış tek producer ve tek consumer'lı bir prototip için bu tören kendini ödemeyebilir ve ikinci bir ekip ortaya çıkana kadar hafif tutardım.

Fat event'ler zamana coupling'i şekle coupling ile takas eder.

Consumer'ların artık producer'ın erişilebilir olmasına ihtiyacı yok, ama onun kopyaladığı alanlara bağımlıdır. Daha büyük bir payload, daha büyük bir contract, kararlı tutulacak daha fazla şey ve korunacak daha fazla şey demektir.

Thin event'ler yayınlamak ucuz, tüketmek pahalıdır. Her consumer bir geri çağrı öder ve producer o okuma yükünü karşılamak zorundadır; bu yük genellikle event'le aynı anda zirve yapar. Producer'ın API'si yavaşsa, thin event problemi producer'dan her consumer'a taşır.

Özenli envelope'lar bayta ve dikkate mal olur. Her mesajdaki dokuz alan yüksek hacimde birikir ve her producer bunları doğru doldurmak zorundadır. Alternatif, eksik bilgiyi bir incident sırasında yeniden kurmaktır ve bu daha pahalıya mal olur. Bunun ne kadar önemli olduğu mesaj hacminize bağlıdır. Olağan yaklaşım envelope'u küçük ve payload'u yalın tutmaktır.

Wire düzeyindeki kurallar esnekliğe mal olur. Id'ler ve para için string'ler, sabit bir timestamp biçimi ve gövdede bir envelope, polyglot bir sistemi iyi anlamda sıkıcı kılar. Aynı zamanda uçlarda dönüştürme yapmak demektir ve kendi dilinde native tipler isteyen bir ekip bu dönüşümün bedelini her mesajda öder.

Bir command'ın expiry'si bir saat bağımlılığına mal olur. `expires_at` bayat command'lara karşı korur ama alıcının saatine göre değerlendirilir. Eklediğiniz marj, gerçekten geç kalmış bir command'ın hâlâ kabul edildiği süredir.

Anonimleştirilmiş ya da yalnızca tanımlayıcı içeren event'ler, consumer'ların tek başına yapabileceklerini sınırlar. Alıcıyla iletişime geçmesi gereken bir consumer'ın sahibine bir çağrıya ya da elinde tutması için bir gerekçe verilmiş verinin yerel bir kopyasına ihtiyacı vardır. Bu bir sürtünmedir ve seçilmiş bir sürtünmedir.

Bir event'in çevresine bu kadar yapı ne zaman koyulmaz. Tek bir servisin içinde, aynı codebase tarafından tüketilen ve hiç saklanmayan tamamen dahili bir event'in envelope'a, version'a ya da sahip listesine ihtiyacı yoktur. Tam muameleyi bir sınırı geçen event'ler için ve özellikle bir deploy'dan uzun yaşayan event'ler için saklayın.

Sık yapılan hatalar

- **Domain event'leri doğrudan yayınlamak.** Mekanizma var olduğu için cazip ve sonuç, dahili refactoring'lerin breaking change'e dönüşmesidir. Integration event'lere açık bir mapping ekleyin ve o mapping'i contract olarak ele alın.
- **Bir command'a event demek.** Geçmiş zaman eki kozmetik olarak düşürülür ya da eklenir ve producer, bir reddi duymanın yolu olmadan

belirli bir alıcıya bağımlı hâle gelir. Command'ları adlandırılmış bir alıcıya gönderin ve geri bir sonuç bekleyin.

- **Updated event'leri.** OrderUpdated neyin değiştiği hakkında hiçbir şey söylemez, bu yüzden her consumer state'i diff'ler ve tahmin eder. İş olgusunu adlandırın ve ad yoksa event'in gerekli olup olmadığını sorgulayın.
- **Aggregate'in tamamını payload'a boşaltmak.** Bir konuşmadan kurtarır ve veritabanı tablosu boyutunda bir public contract yaratır. Her consumer'a her alanla ne yaptığını sorun ve kimsenin kullanmadığını kesin.
- **Kişisel veriyi ve dahili alanları elin altında oldukları için koymak.** Bunu review sırasında bir kontrol listesiyle düzeltin, çünkü yayından sonra veri zaten kontrol etmediğiniz bir storage'dadır.
- **Dil varsayılanıyla encode etmek.** event_type olarak bir class adı, number olarak id'ler, float olarak para, runtime'in seçtiği her offset ile timestamp'ler. İkinci dil onu okuyana kadar çalışır. Wire kurallarını, yukarıdaki tablonun yaptığı gibi, bir kez yazın.
- **Tetikleyicinin partition_key'ini kopyalamak.** İki mesaj tek bir aggregate ile ilgiliyse doğrudur, başka bir aggregate hakkındaki follow-up için yanlıştır. Key'i event'in kendi aggregate'inden türetin.
- **correlation_id'yi her sıçramada yeniden üretmek.** Biri varsayılan olarak yeni bir tanımlayıcı oluşturan bir helper yazar. Onu tetikleyiciden kopyalayın ve her consumer'ın bunu yaptığını test edin.
- **occurred_at'i sıra olarak kullanmak.** Bir zaman damgası bir saat okumasıdır ve saatler anlaşmaz. Partition key'i ve Bölüm 9'un anlattığı gibi bir version numarasını kullanın.
- **Sahipsizlik.** "Platform" tarafından ya da iki servis tarafından tek bir ad altında yayınlanan bir event'in, bir değişikliği onaylayabilecek kimsesi yoktur. Event type başına bir sahip servis atayın.

Temel çıkarımlar

- Bir event bir olguyu belirtir, bir command adlandırılmış bir alıcıdan bir eylem ister ve bir document veri taşır; bu üçünü karıştırmak gizli coupling'e ve kaybolan reddetmelere yol açar ve bir command'ın bir

yanıt yoluna ve isteğe bağlı bir expiry'ye ihtiyacı vardır, çünkü kaybolanı yalnızca göndericinin deadline'ı fark eder.

- Domain event'ler bir servisin dahili modelini anlatırken, integration event'ler bilinçli olarak seçilmiş public contract'tır ve aralarındaki açık bir mapping, refactoring'in breaking change'e dönüşmesini önler.
- Thin event'ler contract'ları küçük tutar ama runtime coupling'i ve geri çağrı race'lerini geri getirir; fat event'ler geri çağrıyı kaldırır ama contract'ı büyütür, bu yüzden consumer'ların harekete geçmek için ihtiyaç duyduğunu taşıyın, yalnızca isteyebileceği hiçbir şeyi değil.
- Event'leri geçmiş zamanda, önce özneyle adlandırın, iş dilini kullanın ve olguyu gizleyen Updated tarzı adlardan kaçın.
- Envelope, tooling'in payload'u okumadan deduplicate etmesi, route etmesi, trace etmesi ve sıralaması için vardır; wire encoding'i (string id'ler, RFC 3339 UTC timestamp'ler, güvenli para, dilden bağımsız type adları) onu her dilde okunabilir tutan şeydir ve correlation_id akış boyunca kopyalanırken causation_id tek doğrudan tetikleyiciyi işaret eder.
- Kişisel veriyi, secret'ları, dahili state'i ve consumer'a özgü talimatları payload'ların dışında tutun, çünkü bir event log'undaki her şey geniş çapta kopyalanır ve zor kaldırılır.
- Her event'in bir sahibi servis, dokümanite edilmiş bir anlamı ve bilinen consumer'ların bir listesi vardır; bir mesajı bilerek değiştirilebilen bir contract'a dönüştüren budur.

İyi tasarlanmış bir event'in yine de değişmesi gerekir, çünkü iş değişecek ve consumer'lar farklı hızlarda yükseltilirken eski event'ler log'da kalacak. Bölüm 12, schema evolution'a dönüyor: diğer ekiplerin zaten bağımlı olduğu bir contract'ı onları bozmadan nasıl değiştireceğiniz.

Bölüm 12 — Schema Evolution

Parcelly'nin Payments ekibinin kısmi capture'ları desteklemek istediğini düşünün. Bir satıcı iki kalemlili bir siparişin yarısını gönderiyor ve alıcıdan şimdi o yarısı, geri kalanı sonra tahsil edilmeli. `PaymentCaptured` event'i tek bir alan taşıyor, `amount_minor`; bu alan şimdiye kadar hep "siparişin tamamı" anlamına geldi. Ekip onu `captured_minor` olarak yeniden adlandırıyor ve `remaining_minor` ekliyor; bu daha temiz ve ne demek istediğini söylüyor. Payments repository'sindeki testler geçiyor ve deploy bir perşembe günü geç saatte yayına çıkıyor.

Ledger o repository'de değil. Çift kayıt (double-entry) bir kayıt işlemek için `amount_minor`'u okuyor, hiçbir şey bulamıyor ve her yeni event'i hatalı biçimlendirilmiş diye reddediyor. Notifications aynı alanı bir makbuz için okuyor ve boş bir tutar basıyor. Geçen ayın event'lerini replay eden bir raporlama işi artık aynı topic'te hem eski hem yeni adlarla karşılaşılıyor; parser'ı yalnızca birini işliyor.

Buradaki hiçbir şey Payments'taki bir bug değil. Kod, ekibin istediğini yaptı. Hata, bir event'in yazarın hiç tanışmadığı programlar tarafından okunması ve bazılarının aylar önce yazılmış event'leri okuması. Okuyucularını kontrol etmediğiniz ve eski yazımlar hâlâ log'da dururken bir contract'ı nasıl değiştirirsiniz?

Problem ve neden zor olduğu

Bir event schema'sı, bir producer ile şimdiki ve gelecekteki her consumer arasındaki bir contract'tır. Kendi codebase'inizdeki bir fonksiyon imzası da bir contract'tır, ama bir compiler iki tarafı da denetler ve ikisini de tek bir commit'te değiştirebilirsiniz. Bir event schema'sında iki taraf arasında bir compiler yoktur. Producer ve consumer'lar farklı günlerde, farklı ekiplere ait olarak deploy edilir ve broker hangi byte'lar gelirse kabul eder.

Üç özellik bunu sıradan API versiyonlamasından daha zor kılar.

Birincisi, **event'ler onları yazan koddan uzun yaşar**. Log tabanlı bir broker'da event'ler yapılandırılmış retention süresi boyunca kalır ve bazı sistemler onları süresiz tutar. Queue tabanlı bir broker'da pencere daha kısadır, ama bir backlog ya da bir dead-letter queue (DLQ) mesajları günlerce tutabilir. Bugün deploy edilmiş bir consumer'dan, bir yıl önceki bir schema ile yazılmış bir şeyi okuması istenebilir.

İkincisi, **replay geçmiş canlı bir girdiye çevirir**. Bölüm 2'de tanımlanan replay, bir projection'ı yeniden kurmanın, yeni bir servisi backfill etmenin ya da bir bug'dan sonra toparlanmanın yoludur. Bugünün koduna bütün tarihsel schema'ları bir anda verir. Yalnızca en son şekli işleyen bir consumer, biri replay edene kadar çalışır ve sonra önce en eski event'lerde hata verir. Şekil işin yalnızca yarısıdır: replay ayrıca bugünün iş kurallarını ve yan etkilerini eski olgulara uygular; örneğin bir makbuzu yeniden göndermek ya da ödeme sağlayıcısını yeniden çağırmak gibi. Replay ile yeniden kurulan bir consumer, açık bir mod ya da bir offset aralığı aracılığıyla replay yaptığını bilmeli, dış etkilerini bastırmalı ve tarihsel olgular için geçerli kurallar tutmalıdır. Deduplication tarafını Bölüm 7 ele alıyor.

Üçüncüsü, **herkesi bir anda yükseltemezsiniz**. Rolling deploy'lar, aynı servisin eski ve yeni sürümlerinin dakikalarca ya da saatlerce yan yana çalışması demektir. Her şey yükseltildikten sonra doğru olan bir değişiklik, yükseltme sırasında yine de bir incident'tir.

Bariz yaklaşım, schema'yı dahili bir ayrıntı olarak görüp consumer'ları sonradan düzeltmektir. Bu, bir consumer ve bir ekip olduğu sürece işe yarar. Producer kaç consumer olduğunu bilmediğinde işe yaramaz olur ve event-driven tasarımın yaratmak için kurulduğu durum budur. Bölüm 11, yayınlanmış bir event'in public contract olduğunu savundu. Schema evolution, bu sözü zaman içinde tutmanın yoludur.

Compatibility'nin tanımı

Aynı sözcükler birçok yerde gevşek biçimde kullanılıyor, bu yüzden kitabın kullandığı tanımlar şunlar.

- **Backward compatibility:** yeni schema'yı kullanan bir consumer, eski schema ile yazılmış event'leri okuyabilir. Yeni okuyucular, eski veri.

- **Forward compatibility:** eski schema'yı kullanan bir consumer, yeni schema ile yazılmış event'leri okuyabilir. Eski okuyucular, yeni veri.
- **Full compatibility:** ikisi birden.
- Her birinin **transitive** varyantları, yeni schema'yı yalnızca en sonuncuya değil, her önceki sürüme karşı denetler.

Yön kafanızda kolayca ters dönebilir, bu yüzden bunun yerine somut bir soru soruyorum: "Bu değişikliği deploy edersem kim bozulur?"

Compatibility	Kim neyi okur	Güvenli deploy sırası	Neye karşı korur
Backward	Yeni okuyucu, eski veri	Önce consumer'lar, sonra producer'lar	Geçmişin replay'ine, geride kalmış backlog'lara
Forward	Eski okuyucu, yeni veri	Önce producer'lar, sonra consumer'lar	İstenildiğinde yükseltilemeyen consumer'lara
Full	İki yön de	Herhangi bir sıra	Karışık filolara, bilinmeyen consumer'lara
None	İkisi de garanti edilmez	Bir migration planıyla koordineli	Hiçbir şeye; migration bölümüne bakın

İsimlerle ilgili bir uyarı. Registry'ler ve serialization kütüphaneleri bu sözcükleri küçük farklarla kullanır. Bir ayara güvenmeden önce kullandığınız araçtaki tanımları okuyun ve registry'nizin varsayılan olarak hangi varyantı uyguladığını kontrol edin: transitive olmayan bir kural, replay edilebilir bir log için yeterli değildir. Her sürümden gerçek örnek event'lerle test edin.

Replay'li bir log için, bütün geçmişe karşı backward compatibility ihtiyacınız olan özelliktir, çünkü replay eden bir consumer şimdiye kadar yayınlanmış herhangi bir sürümle karşılaşabilir. Retention'ı kısa bir queue için önceki sürümle denetlemek yeterli olabilir. Bu, bir schema kararı kadar bir retention kararıdır.

Güvenli ve breaking değişiklikler

Eski consumer'lar bir değişikliği yok sayabiliyorsa ve yeni consumer'lar onun yokluğuyla başa çıkabiliyorsa değişiklik **additive**'dir. Bir okuyucu, eski ya da yeni, bir event'i artık yorumlayamıyorsa değişiklik **breaking**'dir.

Aşağıdaki hükümler, okuyucuların tolerant reader bölümünde anlatıldığı gibi davrandığını varsayar.

PaymentCaptured üzerinde değişiklik	Olağan hüküm	Neden
Opsiyonel bir alan ekle (payment_method)	Güvenli	Eski okuyucular yok sayar; yeni okuyucular yokluğu işler
Default'u olmayan zorunlu bir alan ekle	Breaking	Yeni okuyucular eski event'leri yorumlayamaz
Opsiyonel bir alanı kaldır	Yalnızca kimse okumuyorsa güvenli	Ona bağımlı okuyucular hiçbir şey görmez
Zorunlu bir alanı kaldır	Breaking	Eski okuyucular onu bekler
Bir alanı yeniden adlandır	Breaking	Bir kaldırma artı bir ekleme olarak okunur; birkaç format bir yeniden adlandırmayı alias'larla çözebilir
Bir tipi değiştir (int'ten string'e)	Breaking	Parser'lar hata verir ya da sessizce dönüştürür
Sayısal bir tipi genişlet	Formata bağlı	Bazı formatlar yükseltir, bazıları yükseltmez
Bir alanın anlamını değiştir	Breaking ve en kötü türden	Hiçbir şey hata vermez; sonuçlar yanlıştır
Bir enumeration'a değer ekle	Riskli	Eski okuyucular onu reddedebilir ya da yanlış işleyebilir

İki satır daha uzun bir bakışı hak ediyor.

Anlamı değiştirmek tehlikeli değişikliktir

Bir alanı yeniden adlandırmak gürültüyle bozar. Ledger ilk event'te hata verir, bir alert tetiklenir ve biri stack trace'i okur. Bir alanın ad ve tipine dokunmadan ne anlama geldiğini değiştirmek sessizce bozar. Payments'ın amount_minor'un artık "toplam sipariş tutarı" yerine "bu event'te capture edilen tutar" anlamına geleceğine karar verdiğini düşünün. Her parser hâlâ başarılı olur. Ledger kısmi tutarları toplamış gibi işler ve hata haftalar sonra bir reconciliation'da (Bölüm 13) kimsenin açıklayamadığı bir fark olarak yüzeye çıkar.

Hiçbir schema aracı bunu yakalamaz, çünkü schema değişmemiştir. Yalnızca değişikliği gözden geçiren bir insan yakalar. Benim kuralım: anlam değişiyorsa, bu yeni bir alan ya da yeni bir event type'tır, eskisinde yapılan bir düzenleme asla değil.

Enumeration'lar gizli bir contract'tır

Bir enumeration değeri eklemek schema'da additive'dir ve çoğu zaman consumer'larda değildir. `payment_method`'u bir işleme dalına map eden ve "card" varsayan bir `else` dalı olan bir consumer, yeni bir `"bank_transfer"`'ı hiçbir hata vermeden yanlış işler. Producer'lar enumeration'ların açık olduğunu dokümente edebilir. Consumer'lar bilinmeyen değerleri bilinmeyen olarak ele alıp, onları bilinen bir duruma katmak yerine görünür bir hataya ya da güvenli bir default'a yönlendirebilir.

Schema registry'ler ve formatlar

Schema registry, bir sistemin kullandığı schema'ları saklayan, onlara tanımlayıcılar atayan ve önerilen bir schema'yı öncekilere karşı denetleyebilen bir servistir. Ayırmaya değer iki iş yapar.

1. **Lookup.** Bir mesaj, yazıldığı schema'nın tanımlayıcısını taşır ya da ona işaret eder. Bir consumer o schema'yı alır ve byte'ları decode eder. Kompakt binary formatları çalışır kılan budur: alan adları her mesajla birlikte taşınmaz.
2. **Gatekeeping.** Bir producer yeni bir schema register etmeden önce, registry onu seçilmiş bir compatibility kuralı altında önceki sürümlerle karşılaştırır ve kuralı bozuyorsa reddeder.

İkinci iş, en sert savunacağım olandır. Bir review kontrol listesinde yaşayan kurallar, ekibin meşgul olduğu gün atlanır. Deploy pipeline'ındaki bir kontrol meşgul olmaz.

Subject'ler, topic'ler ve event type'lar

Bir registry compatibility'yi **subject** başına denetler ve bir subject'in topic mi yoksa event type mı olduğu, birkaç event type'ın bir topic'i paylaşım paylaşamayacağını belirler. Event type'lar arasında aggregate başına ordering'in (Bölüm 9) ihtiyaç duyduğu şey paylaşımıdır: bir ödemenin

PaymentAuthorized ve PaymentCaptured event'leri ancak aynı topic'in aynı partition'ında yolculuk ederlerse sıralı kalır. Parcelly örnekleri, aggregate türü başına bir topic varsayar; örneğin `payments.payment-events`, aggregate id ile key'lenmiş birkaç event type taşır ve compatibility topic başına değil event type başına yapılandırılır. Bedeli, consumer'ların `event_type`'a göre filtrelemesi ve versiyonlu bir topic'in (Strateji 3) o aggregate'in tüm event type'larını bir anda taşımasıdır. Event type başına bir topic yapılandırması daha basittir ve type'lar arasındaki ordering'ten vazgeçer.

Üç format ailesi

Kitap formattan bağımsızdır, bu yüzden bu bir ürün karşılaştırması değil, bir oryantasyondur. Avro, Protobuf ve JSON Schema, verinin kablo üzerinde neye benzediği bakımından farklıdır ve bu da neyin değiştirilmesinin güvenli olduğunu şekillendirir.

Aile	Kablo üzerinde	Alanlar neyle tanımlanır	Pratik sonuç	Tuzak
Avro tarzı	Kompakt binary, alan adı yok	Ad; writer ve reader schema'sı arasında çözülür	Okuyucunun writer'ın schema'sına ihtiyacı var; default'lar compatibility'yi taşır	Alias'lar bir okuyucunun yeniden adlandırılmış bir alanı çözmesine izin verir; yani yeniden adlandırma her zaman breaking değildir. Eksik bir alan okuyucunun default'uyla doldurulur; dolayısıyla yokluk görünmezdir
Protobuf tarzı	Kompakt binary	Alan numarası	Yeniden adlandırmak ucuzdur; bir numarayı yeniden kullanmak tehlikelidir	Yalnızca binary kablo üzerinde ucuzdur: ProtoJSON ve üretilmiş kod adları kullanır; dolayısıyla yeniden adlandırma kaynağı ve JSON'u bozar. Açık presence'ı olmayan bir proto3 scalar, eksik bir alanı 0 olarak decode eder
JSON Schema	Metin JSON	Ad; schema runtime'da opsiyonel	Debug etmek kolay; zorlanmadıkça compatibility bir konvansiyondur	Kapalı bir modelle (<code>additionalProperties: false</code>) bir alan eklemek eski okuyucuları bozar ve registry'ler açık ve kapalı modelleri farklı değerlendirir

Üç fikir hepsine geçer.

- **Default'lar, default dürüstse, eklemeleri güvenli kılar.** Tanımlı bir default'u olan yeni bir alan, yeni bir okuyucunun eski bir event'i işlemesine izin verir. Olmadan, bir ekleme bir kırılmaya dönüşür. Okuyucunun eksik olarak tespit edebilmesi gereken bir alan için, örneğin bir tutar, default "değer yok" olmalıdır: `null` ile tipin Avro union'ı ve `null` default'u, ya da proto3 `optional` ya da bir wrapper tipi. 0 default'u, eski bir event'i sıfır tutarlı bir capture olarak decode eder ve okuyucu farkı anlayamaz. Bu durumda okuyucu değere değil, presence'a bakar.
- **Kimlik yeniden kullanılmamalıdır.** Bir alan ad ya da numarayla tanımlansın, eski bir kimliğe yeni bir anlam vermek bir kırılmadır. Tanımlayıcıları emekli edin; geri dönüştürmeyin.
- **Bilinmeyen alanlara tolerans gösterilmelidir.** Bir decoder'ın bilmediği alanları tutup tutmadığı, yok sayıp saymadığı ya da reddedip reddetmediği formata ve ayarlarına bağlıdır. Etrafında tasarım yapmadan önce öğrenin.

Bir schema dili olmadan JSON çalıştırıyorsanız, schema dosyalarını version control'de tutarak, producer'ın testlerinde doğrulayarak ve sürüm başına gerçek örnek event'ler bulundurarak yine de faydanın çoğunu alırsınız. Registry merkezi lookup ve zorlama ekler. Düşünmeyi eklemeyiz.

Version nerede yaşar

Standart envelope, `schema_version`'ı `event_type`'ın yanında bir tamsayı olarak taşır ve bir consumer hangi şekli beklemesi gerektiğini bilmek için bu çifti okur. Onu envelope'ta tutmak, bir router'ın, bir metric sayacının ya da bir DLQ aracının onu payload'u decode etmeden görebilmesi demektir. "Hâlâ kaç tane `schema_version: 1` event'i geliyor?" bir proje yerine ucuz bir sorguya dönüşür.

Burada iki numara kolayca karıştırılır; bu yüzden bu bölüm onlara ayrı adlar verir. **Breaking version**, envelope'taki `schema_version`'dır. Yalnızca eski bir okuyucunun işleyemeyeceği bir değişiklikte ilerler: bir yeniden adlandırma, bir kaldırma, bir tip değişikliği ya da bir anlam değişikliği. **Registry revision**, registry'nin register edilen her schema'ya atadığı tanımlayıcıdır ve additive olanlar dahil register edilen her değişiklikte ilerler. Bir consumer, breaking version'ını tanımadığı bir event'i reddeder. Daha yeni bir registry revision'ı ama bilinen bir breaking version'ı olan bir

event'i, bildiği en son şekil olarak okur ve tanımadığını yok sayar. İkisi de Bölüm 4'teki aggregate version'ı değildir; o, tek bir siparişteki değişiklikleri sayar ve payload'un şekliyle ilgisi yoktur.

Her additive değişiklik `schema_version`'ı artırırsaydı, henüz yükseltilmemiş her consumer her yeni event'i reddeder ve forward compatibility ortadan kalkardı. Ayrımın bedeli, doğru tutulması gereken iki numaradır. Aynı çizginin bir örneği, açık kaynaklı ve dilden bağımsız bir envelope standardı olan BabelQueue'dur (babelqueue.com): envelope'unu `schema_version` 1'de dondurur, yeni alanlara yalnızca opsiyonel ve yok sayılabilir olarak izin verir, bir yeniden adlandırma, kaldırma ya da tip değişikliği için version artışı ister, consumer'ların tanımadıkları bir version'ı reddetmesini sağlar, bilinmeyen alanların işlemeyi bozmasını önler ve mesaj yeniden yayınlandığında korunmalarını sağlar.

Aşağıdaki migration'ın dual-publish adımıyla Payments'ın yayınladığı hâliyle bir event:

```
{
  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"
  }
}
```

Diller arası compatibility

Bir schema, registry'nin kurallarına göre kusursuz biçimde compatible olabilir ve yine de iki dil arasında bozulabilir; çünkü kurallar yapıyı tarif eder, kırılma ise bir runtime'ın onu nasıl okuduğundadır. Bölüm 11, envelope'un encoding'ini sabitledi. Bunlar, payload'daki ve bir consumer'ın tanımadığı alanlarla yaptığı işlerdeki tuzaklardır.

Bilinmeyen alanlar yeniden yayınlamada kaybolur. Notifications'taki bir Node consumer'ın her `PaymentCaptured`'ı `order_id`, `captured_minor` ve `currency` alanlı bir class'a deserialize ettiğini ve event'i partner'a dönük bir topic'e kopyaladığını varsayın. Payments sonradan `payment_method` eklediğinde kopya onu sessizce düşürür. Hiçbir şey hata vermez. Alan hiçbir zaman orada değildir. Protobuf binary decoder'ları bilinmeyen alanları genellikle tutar; tipli JSON mapper'ların çoğu tutmaz. Kural: mesajı ileten bir hop, ham payload'u ya da generic bir map'i taşıyarak bilinmeyen alanları korur ya da kendi contract'ı altında kendine ait yeni bir event yayınlar.

Sayılar. `captured_minor` her dilin tamsayısına rahatça sığar. Bir int64 tanımlayıcı ya da sayaç, $2^{53} - 1$ 'in ötesinde bir JavaScript JSON number'ına sığmaz ve consumer onu hata vermeden yuvarlar. Kural: her sayısal alanın aralığını dokümente edin ve aralığı aşabilecek her şeyi ve her decimal'i string olarak gönderin.

Timestamp'ler. Avro'nun `timestamp-millis` ve `timestamp-micros` tipleri aynı türden bir değeri farklı ölçeklerde tutar. Mikrosaniye ile yazan bir writer'ın değeri milisaniye olarak okunursa, on binlerce yıl uzakta bir tarih çıkar ve yine de parse edilir. Kural: birimi schema'da sabitleyin ve asla değiştirmeyin; bir birim değişikliği breaking bir version'dır.

Enumeration'lar. Bilinmeyen bir değeri üretilmiş kod farklı ele alır: bazıları tutar, bazıları default'a eşler, bazıları exception fırlatır. Kural: her enumeration'ı açık sayın ve bilinmeyen ne demek olduğuna consumer'da karar verin.

Null ve absent. Birçok decoder ikisini ayırt etmez ve bazıları boşluğu bir default ile doldurur. Kural: yokluğu önemli olan alanlar için yukarıdaki default maddesindeki presence-aware biçimleri kullanın.

Bunların hiçbiri belirli bir kütüphanenin belirli bir sürümünü gerektirmez. Hepsi, Parcellay'nin çalıştırdığı her dildeki bir consumer ile test eden birini gerektirir; bu, Bölüm 15'in contract test'lerinin taşıyabileceği bir maliyettir.

Versiyonlama stratejileri

Bir değişikliği devreye almanın, ucuzdan pahalıya üç yolu var. Onları sırayla denerdim ve yalnızca alttaki basamak değişikliği taşıyamadığında tırmanırdım.

Strateji 1: opsiyonel bir alan ekle

Değişiklik, eski okuyucuların yok sayabileceği yeni bir şeyse, opsiyonel bir alan ekleyin. `PaymentCaptured`'a `payment_method` eklemek model örnektir. Bu, yeni bir registry revision'ı register eder ve registry'nin onu kabul edip etmeyeceğine compatibility kuralı karar verir.

`schema_version`'ı değiştirmez, çünkü hiçbir eski okuyucu bir şey kaybetmez. Additive bir değişiklik için breaking version'ı artırmak, henüz yükseltilmemiş her consumer'ın yeni event'leri reddetmesine yol açardı; bu da alanı compatible bir biçimde eklemenin amacını boşa çıkarır.

Bu strateji bir yeniden adlandırmayı, bir tip değişikliğini, zorunlu bir şeyin kaldırılmasını ya da bir anlam değişikliğini ifade edemez. Onlar için bir üst basamağa çıkın.

Strateji 2: yeni bir event type getir

Anlam değiştiğinde, bunu yeni bir adla söyleyin. `Parcellly`'nin kısmi `capture`'ı tam bir `capture`'dan farklı bir iş olgusu olarak ele aldığını varsayın. Dürüst model yeni bir event'tir; Bölüm 11'in kurallarına göre adlandırılır. Bu bölüm ek bir event olarak `PaymentPartiallyCaptured`'ı getiriyor.

Eski event anlamını korur ve consumer'ları çalışmaya devam eder. Bedeli, artık iki event'in de var olması ve "bütün `capture`'ları" isteyen bir consumer'ın ikisine de abone olması gerektiğidir. Yeni olgu kavramsal olarak farklı olduğunda bu doğru takastır; aynı olgunun daha iyi bir şekli olduğunda yanlıştır. O ikinci durum bir temsil problemidir ve bir upcaster ona daha iyi uyar.

Strateji 3: versiyonlu bir topic

Breaking bir değişiklik bütün akışa dokunduğunda, yeni şekli yeni bir topic'e koyun, örneğin `payments.payment-events.v2`. Consumer'lar kendi takvimlerinde migrate eder ve eski topic, son okuyucusu ayrıldığında

emekli edilir. Yukarıda anlatılan düzenle, yeni topic aggregate'in her event type'ını taşır.

Bu en net seçenek ve en pahalısı. İki topic'in partition key'leri, ordering'i ve offset'leri birbirinden ilişkisizdir; bu yüzden birinden diğerine geçen bir consumer nereden okumaya başlayacağına ve aynı olguyu iki kez işlemekten nasıl kaçınacağına karar vermek zorundadır. Bölüm 9'un ordering kuralları her topic için ayrı ayrı geçerlidir ve hiçbir şey event'leri ikisi arasında sıralamaz. Queue tabanlı bir broker'da karşılığı, kendi binding'leri, dead-letter ele alışı ve alert'leri olan ikinci bir queue'dur. Versiyonlu bir topic'e yalnızca bir upcaster'ın köprüleyemeyeceği gerçek bir kırılma için ve yalnızca eskisini kimin sileceğini adlandırabildiğimde başvururum.

Upcasting ve tolerant reader

İki teknik, producer'ı dondurmaya zorlamadan consumer'ların birkaç sürümle başa çıkmasına izin verir.

Upcasting

Bir **upcaster**, eski bir schema ile yazılmış bir event'i, iş mantığı onu görmeden önce güncel schema'ya dönüştürür. İş kodu tek bir şekle karşı yazılır ve her tarihsel sürümün dönüşümü tek bir yerde yaşar.


```

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

```

Version 1 dalındaki yorum asıl işi taşır. Bir upcaster geçmiş hakkında bir ifadeyi kodlar ve o ifadenin doğru olması gerekir. Bazı eski event'ler pratikte kısmiyse, upcaster kısmi bir tutarı sessizce capture'ın tamamı olarak bildirir. Upcaster'ları uydurma değil, gerçek arşivlenmiş event'lerle test edin.

`UnsupportedSchemaVersion`'ın kendine ait bir sınıflandırmaya ihtiyacı var. Mevcut kod için kalıcıdır, çünkü yeniden denemek hiçbir şeyi değiştirmez; ama bir consumer deploy'u onu iyileştirir. Bu yüzden parking lot'a gönderilmez, orada karar verecek bir şeyi olmayan bir insanı beklerdi; consumer yükseltildikten sonra redrive için tutulur (Bölüm 8).

Handler içinde, iş mantığından önce yer alan consumer tarafı, başlamak için en basit yerdir. Geçmiş yeni bir topic'e yeniden yazmak, aksi halde birçok consumer'ın aynı upcaster'a ihtiyaç duyacağı durumlarda yararlıdır. Saklanan event'leri yerinde yeniden yazmak farklı bir karardır: event-sourced sistemler çoğu zaman bunu reddeder, çünkü log kanıttır. Yasal ya da güvenlikle ilgili bir neden olmadıkça buna saygı gösterirdim ve o zaman bunu bir planla bir veri migration'ı olarak çalıştırdım.

Tolerant reader

Tolerant reader, bir event'ten yalnızca ihtiyacı olanı alır ve gerisini yok sayar. `order_id` ve `captured_minor`'u okur ve bir de `payment_method` olması umurunda değildir. Kabul ettiği konusunda liberal, bağımlı olduğu konusunda katıdır.

```
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
    }
```

Toleransın iki yüzü var. Producer'lar bir şey eklediğinde bilinmeyen alanları yok saymak güvenlidir. Eksik bir tutar yerine sessizce sıfır koymak tehlikelidir. Yukarıdaki okuyucu, bağımlı olduğu bir alan yoksa devam etmeyi reddeder ve değere değil, presence'a bakar. Bu, ancak decoder yok olan bir alanı sıfırdan ayırt edebiliyorsa dürüştür; yukarıdaki default'ların önemi de buradan gelir. Bu ret bir sınıflandırmadır: event asla başarılı olamaz, bu yüzden Bölüm 6'daki kalıcı hata yolunu izler ve Bölüm 8'in anlattığı gibi bir dead letter olarak biter.

Not: Tolerans, iki taraf arasında bir anlaşmadır. Producer eklemeye ve anlamı değiştirmemeye söz verir. Consumer anlamadığını yok saymaya ve güvenle yapamayacağını reddetmeye söz verir. Taraflardan biri kendi yarısını bozarsa, diğeri sistemi kurtaramaz.

Registry, producer'ı yayınlamadan önce denetler. Tolerant reader, bir şey sızdıktan sonra consumer'ı korur. Birini diğeri'nin yerine geçirmeye güvenmezdim.

Breaking bir değışikliğı migrate etmek

Bazen bir kırılma kaçınılmaz olur: eski şekil yanlıştır, bir düzenleme bir alanı dışarı çıkmaya zorlar ya da iki servis birlikte yeniden şekillendirilmelidir. Cevap, tek bir breaking deploy'u birkaç non-breaking deploy'a çevirmektir. Bu, event'lere uygulanan **expand and contract** örüntüsüdür: contract'ı iki şekli de destekleyecek biçimde genişletin, her okuyucuyu migrate edin, sonra eski şekli kaldırarak daraltın.

PaymentCaptured için bir migration planı

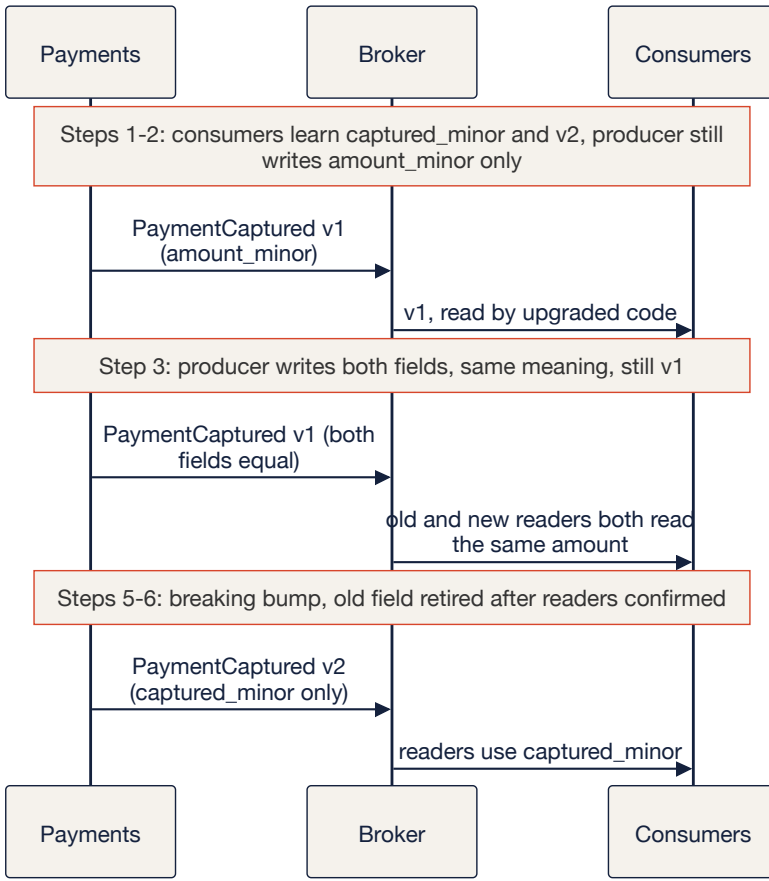
Parcelly'nin gerçekten `amount_minor`'u `captured_minor` olarak yeniden adlandırması gerektiğini ve event'le ilgili başka hiçbir şeyin değışmediğini varsayın. Değer anlamını korur: tüm sipariş için capture edilen tutar. Bu saf bir yeniden adlandırmadır ve aşağıdaki plan yalnızca bu yüzden güvenlidir. Kısmi capture'lar bir anlam değışikliğidir ve adımlardan sonra ayrıca ele alınır.

1. **Ekleme register edin.** Yeni registry revision'ı `amount_minor`'u korur ve `captured_minor`'u, default'u 0 değil "değer yok" olan opsiyonel bir alan olarak ekler. Backward-compatibility kuralı altında registry onu kabul eder. `schema_version` 1 kalır, çünkü hiçbir eski okuyucu bir şey kaybetmez.
2. **Önce consumer'ları yükseltin.** Her consumer `captured_minor`'u okumayı ve yoksa `amount_minor`'a düşmeyi öğrenir. Ayrıca henüz var olmayan breaking version 2'yi okumayı da öğrenir; böylece 5. adımdaki artış onu reddedecek kimseyle karşılaşmaz. Yeni okuyucular, yeni veriden önce var olur.
3. **Dual-publish.** Payments her event'te iki alanı da yazar, hâlâ `schema_version` 1'de; ya da değışiklik yeni bir event type ya da topic gerektiriyorsa, aynı olgunun eski ve yeni biçimini yayınlar. Bu, Bölüm 4'ün dual write'ı değildir: iki mesaj da aynı transactional outbox transaction'ından gelir, bu yüzden olağan garanti geçerlidir.
4. **Gözlemleyin.** Eski alanı ya da eski topic'i kimin hâlâ okuduğunu ölçün. `schema_version` üzerindeki producer metric'leri hâlâ neyin yazıldığını, eski topic üzerindeki consumer group etkinliği, Bölüm 11'deki consumer listesi ve Bölüm 15'in contract test'leri ise onu kimin hâlâ okuduğunu gösterir. Hiç taşınmayan bir consumer, yok sayılacak bir sayı değil, yapılacak bir konuşmadır.

5. **Eski şekli yazmayı bırakın.** Payments yalnızca `captured_minor`'u yayınlar ve `schema_version`'ı 2'ye çıkarır, çünkü eski bir okuyucunun atlatamayacağı tek adım budur. 2. adımı kaçıran bir consumer event'i reddeder ve event, o consumer yükseltildikten sonra redrive için DLQ'ya gider (Bölüm 8). 4. adım bu listeyi boş tutmak için vardır. `amount_minor`, onu taşıyan en eski event retention'dan çıkana kadar schema'da deprecated olarak kalır.
6. **Daraltın.** Eski alanı kaldırın ve varsa eski topic'i emekli edin. Fallback'i consumer'lardan ancak `captured_minor`'u olmayan en eski event retention'dan çıktıktan sonra kaldırın ya da onu kalıcı bir upcaster'a taşıyın.

Kısmi capture'lar bu planın dışında kalır. Bir capture kısmi olabildiği anda, `amount_minor`'un dürüst bir değeri kalmaz. Capture edilen tutarı taşıyorsa, anlamı yerinde değişmiştir ve 2. adımı atlamış bir okuyucu, ister bilinmediği ister geri alındığı için olsun, kısmi tutarı siparişin toplamı gibi kaydeder, hiçbir hata olmadan. Sipariş toplamını taşıyorsa, aynı okuyucu fazla kaydeder. Registry iki sürümü de geçirir, çünkü yapı değişmemiştir. Ders budur: yapısal bir compatibility kontrolü bir anlam değişikliğini göremez. Yeni anlamı yeni bir alana, yeni bir event type'a (Strateji 2, `PaymentPartiallyCaptured`) ya da yeni bir topic üzerindeki yeni bir major version'a koyun. Sonra her okuyucu yeni biçime geçene kadar kısmi capture'ları hiç yayınlamayın ve sonrasında yalnızca yeni biçimde yayınlayın.

Sıra planın kalbidir. Backward-compatible bir değişiklik için önce consumer'lar gider; forward-compatible bir değişiklik için producer'lar önce gidebilir. Hangisinin geçerli olduğunu söyleyemediğinizde, planı yazıldığı gibi çalıştırın ve bir süre iki alanı da tutun.



Topic'ler arasında dual-publishing

Plan versiyonlu bir topic kullandığında, 3. adım olguyu iki kez yayınlar, her topic'e bir kez. İkisini de okuyan bir consumer aynı iş olgusunu iki kez işlememelidir. Kopyalara farklı `event_id` değerleri verirdim, çünkü bunlar farklı mesajlardır ve orijinalin id'sini dokümente edilmiş bir alanda taşırdım. Hangisini seçerseniz seçin, consumer'ın o olgu için idempotency key'i bilinçli olarak tanımlanmalıdır. Key'leri Bölüm 7 ele alıyor; buradaki nokta, bir migration'ın idempotency varsayımlarının tam olarak sınıandığı an olduğudur.

Uyarı: Dual-publishing sırasında, bir consumer'ın hem eski hem de yeni akışı okuyup ikisini de uygulamasına asla izin vermeyin. Bir olgu için iki kaynağı olan bir consumer'ın yazılı tek bir deduplication kuralına ihtiyacı vardır, yoksa bir ledger kaydını iki kez işler.

Rollback'ler ve geçmiş

Rollback planın parçasıdır. Version 2 event'leri zaten yayınlanmışsa, producer'ı geri almak onları log'dan kaldırmaz. Version 2'yi okuyamayan bir koda geri alınmış bir consumer, artık var olan event'lerde hata verir. Bu yüzden consumer'lar, herhangi bir producer onu yazmadan önce version 2'yi okuyabilmelidir ve bir consumer rollback hedefi, retention'daki her sürümü, version 2 dahil, okuyabilmelidir. Yalnızca version 1'i okuyan kod, değişiklikten önceki son release olsa bile kaçınılması gereken bir rollback hedefidir.

Backward compatibility eski event'leri onlara dokunmadan ele alır. Fallback kodunu silmek istiyorsanız üç seçeneğiniz var: eski event'ler süresi dolana kadar upcaster'ı tutmak, saklanan event'leri yeni bir topic'e yeniden yazmak ya da state'in snapshot'ını başka bir yere alıp geçmişin geçmesine izin vermek. Her birinin bir bedeli var: kalıcı kod, güvenmeniz gereken bir migration ya da replay yeteneği. Ne sıklıkla replay ettiğinizi ve düzenleyicilerinizin kayıtları ne kadar süre tutmanızı beklediğini bilmeden sisteminizin hangisini seçmesi gerektiğini bilmiyorum.

Event'leri deprecate etmek

Event'leri eklemek, kaldırmaktan daha kolaydır, çünkü bilmediğiniz bir consumer herhangi birine bağımlı olabilir. Makul bir yaşam döngüsünün dört aşaması var.

1. **Duyurun.** Event'i registry'de ya da event kataloğunda deprecated olarak işaretleyin, yerine geçeni adlandırın, bir emeklilik tarihi ve bir iletişim kişisi verin.
2. **Ölçün.** Onu kimin hâlâ okuduğunu bulun: bir log'da consumer group üyeliği, bir queue'da binding'ler, registry lookup'ları. Listeyi kanıt olarak görün, ispat olarak değil, çünkü aylık bir batch işi bugün yok olup gelecek hafta var olabilir.
3. **Üretmeyi bırakın.** Tarihten sonra yayınlamayı bırakın.
4. **Emekli edin.** Retention son event'i eskitip okuyucular gittikten sonra schema'yı ve topic'i ya da queue'yu kaldırın.

İki uyarı. Deadline'ı olmayan bir deprecation asla bitmez, bu yüzden tarihi yazılı olarak koyun ve ona uyun. Ve emeklilikten sonra replay edilen deprecated bir event hata verir, bu yüzden emekli etmeden önce, etkilenen dönem için replay'in ne yapması gerektiğine karar verin.

Sahiplik önemlidir. Event'e ve deprecation'ına, Bölüm 11'de anlatıldığı gibi producer'ın ekibi sahiptir. Migration'lara consumer'lar sahiptir. Producer'ın tek başına karar vermesine ya da consumer'ların süresiz reddetmesine izin veren bir politika başarısız olur. Yararlı politika, producer'dan gelen bir bildirim ve consumer'ların tuttuğu bir tarihtir.

Trade-off'lar ve maliyetler

- **Bir registry, işletilecek bir servis daha demektir.** Her producer'ın deploy'unun yolunda ve kurulumu bağlı olarak consumer'ların decode işleminin yolunda durur. Availability'ye, erişim kontrolüne ve bir sahabe ihtiyaç duyar ve istemcilerin ona erişilemediği durum için bir planı olması gerekir.
- **Compatibility kuralları producer'ları bilerek yavaşlatır.** Bir alanı bir öğleden sonra yeniden adlandırabilecek bir ekip, artık altı adımlı bir migration yürütüyor. Bağımsız deploy'ların bedeli budur.
- **Upcaster'lar birikir.** Her tarihsel sürüm bir dal ve bir test ekler ve eski event'ler retention'dan çıkana kadar kod sadeleşmez. Süresiz retention, süresiz upcaster'lar demektir.
- **Tolerant reader'lar hataları gizler.** Bilinmeyen alanları yok sayan bir consumer, bir producer umursaması gereken bir şey göndermeye başladığında bunu fark etmekte de başarısız olur. İzleme olmadan tolerans, gürültülü hataları sessiz bir sürüklenmeye çevirir.
- **Dual-publishing, migration sürdüğü müddetçe trafiği ve deduplication işini çoğaltır.**
- **Bir yerine iki version numarası.** Breaking version ve registry revision, additive değişikliklerin consumer'ları bozmasını önler; bunun bedeli, bir ekibin onları dashboard'larda ve konuşmalarda ayırt etme çabasıdır.
- **Version numaraları dokümantasyon değildir.** `schema_version: 4` bir şeyin değiştiğini söyler, neyin ya da kimin etkilendiğini değil.

Schema ile birlikte tutulan bir deęişiklik log'u, numaradan daha deęerlidir.

- **Diller arası kurallar kolaylıktan ödün verdirir.** Büyük sayılar için string'ler, ham payload'un korunması ve sabit timestamp birimleri, her dilde ek dönüşüm kodu ve dil başına bir test demektir.

Bu örüntüler ne zaman kullanılmaz

Tek producer'ı, tek consumer'ı ve ikisini birlikte deploy eden tek ekibi olan bir event için tam mekanizmayı kurmayın. Koordineli bir deploy daha ucuzdur ve sigortalayacağınız risk yoktur. İkinci bir consumer ortaya çıktığında, özellikle başka bir ekibe ait olduğunda, yeniden deęerlendirin.

Bir registry'nin strict mode'unu tasarımın yerine koymayın. Bir dizi güvenli ekleme, bir event'i 40 opsiyonel alanla ve net bir amaç olmadan bırakabilir. Bir event büyümeye devam ediyorsa muhtemelen iki event'tir ve Bölüm 11'in granularity kuralları geçerlidir.

Her deęişiklik için versiyonlu bir topic oluşturmayın. Bunu yapan bir sistem, birçok yarı emekli akışla ve hangisinin güncel olduğunu bilen kimse olmadan sonlanır.

Sık yapılan hatalar

- **Yeniden adlandırmayı küçük bir deęişiklik saymak.** Yeni ad daha temiz olduğu ve testler geçtiği için cazip. Her okuyucuya göre bu bir kaldırma artı bir eklemedir. Yeni adı eskisinin yanına ekleyin, okuyucuları migrate edin, sonra eskisini kaldırın.
- **Bir alanın anlamını yerinde deęiştirmek.** Ad ve tip kalır, bu yüzden hiçbir araç itiraz etmez. Yeni anlama yeni bir alan ya da event type verin.
- **Default'u olmayan zorunlu bir alan eklemek.** Bir ekleme gibi hissettirir, ama eski event'lerde yoktur. Onu default'lu opsiyonel yapın ya da yeni bir version ve bir upcaster ile getirin.
- **Compatibility'yi yalnızca en son sürüme karşı test etmek.** Replay, retention'daki her şeyi okur. Hâlâ orada olan her sürüme karşı denetleyin ve consumer testlerinde (Bölüm 15) her birinden örnek event'ler bulundurun.

- **Producer'ları consumer'lardan önce yükseltmek.** Değişikliğin sahibi olan producer ekibi önce hazır olur. Backward-compatible bir değişiklik için önce consumer'lar gider. Deploy'dan önce sırayı doğrulayın.
- **Bilinmeyen alanlarda yüksek sesle hata vermek.** Katı bir parser her additive değişikliği bir incident'a çevirir. Kullanmadığınızı yok sayın; yalnızca bağımlı olduğunuzu reddedin.
- **Eksik bir alana sessizce default atamak.** Eksik bir tutar yerine sıfır koymak bir kusuru gizler ve 0 dolduran bir decoder bunu sizin kodunuz çalışmadan önce gizler. Bu tür alanlar için "değer yok" default'u kullanın, presence'ı kontrol edin ve bir hata verin ki hata yolu event'i bir insan için tutsun.
- **Her değişiklikte `schema_version`'ı artırmak.** O zaman her additive değişiklik, henüz yükseltilmemiş her consumer için ağır bir hataya dönüşür. Breaking version'ı yalnızca eski bir okuyucunun ele alamayacağı değişikliklerde artırın ve gerisini registry revision'ı sayın.
- **Tipli bir class üzerinden iletmek.** Kopya, class'ın tanımlamadığı her alanı düşürür ve hiçbir şey şikâyet etmez. Yeniden yayınlarken bilinmeyen alanları koruyun ya da kendi contract'ınız altında yeni bir event yayınlayın.
- **Bir alan numarasını ya da topic adını yeniden kullanmak.** Geri dönüştürülmüş bir numara eski veriyi yeni alan olarak okur ve yeniden kullanılan bir topic iki şekli karıştırır. Tanımlayıcıları emekli edin.
- **Tarihsiz ya da sahipsiz deprecate etmek.** Event sürüncemede kalır ve sonraki ekip onu kaldırmaktan korkar. Her deprecation'a bir tarih, bir yerine geçen ve soruları cevaplayan bir kişi verin.
- **Schema'yı kontrol etmeden bir consumer'ı rollback etmek.** Rollback hedefinin retention'daki her sürümü okuyabildiğini doğrulayın.

Temel çıkarımlar

- Bir event schema'sı, kontrol etmediğiniz okuyucularla ve çok önce yazılmış veriyle bir contract'tır; backward compatibility yeni okuyucuların eski veriyi okuyabilmesi, forward compatibility eski

okuyucuların yeni veriyi okuyabilmesi demektir ve replay'li bir log, retention'daki her sürüme karşı backward compatibility'ye ihtiyaç duyar.

- Opsiyonel bir alan eklemek güvenlidir; yeniden adlandırmalar, tip değişiklikleri, yeni zorunlu alanlar ve anlam değişiklikleri breaking'dir ve anlam değişikliği en kötüsüdür, çünkü hiçbir şey gürültüyle hata vermez; `schema_version`'ı yalnızca breaking bir değişiklik iletir, gerisini registry revision'ı sayar.
- Bir schema registry, compatibility'yi deploy pipeline'ında zorlayarak yerini hak eder, ama anlamı değil yapıyı denetler; bu yüzden bir alanın ne söylemesi gerektiğini yine bir insan gözden geçirir.
- Değişikliği taşıyan en ucuz versiyonlama stratejisini tercih edin: önce opsiyonel bir alan, anlam farklı bir olgu olduğunda yeni bir event type ve versiyonlu bir topic yalnızca tüm akış kırılması gerektiğinde.
- Upcaster'lar ve tolerant reader'lar, upcaster geçmiş hakkında doğru bir ifadeyi kodladığı ve okuyucu değere değil presence'a bakıp güvenle yorumlayamadığını reddettiği sürece, consumer'ların birkaç sürümle başa çıkmasına izin verir.
- Diller arasında sessiz kırılmalar şunlardır: kaybolan bilinmeyen alanlar, büyük tamsayılar, timestamp birimleri, kapalı enumeration'lar ve null ile absent farkı; bu yüzden bilmediğinizi koruyun ve temsili yazıya dökün.
- Breaking bir değişikliği bir dizi non-breaking değişiklik olarak migrate edin; artışı eski bir okuyucunun atlatamayacağı tek adıma koyun ve rollback'leri aynı dokümanda planlayın.

Güvenle evrilen bir contract, ancak sistemin içinden ne aktığını görebiliyorsanız güven verir ve Bölüm 13 oradan başlıyor.

KISIM V

İşletmek ve Kanıtlamak

Bölüm 13 — Observability ve Reconciliation

Parcelly'de bir Perşembe sabahı düşünün. Destek ekibi bir ticket iletiyor: bir alıcı iki gün önce siparişinin ödemesini yapmış, kart ekstresinde çekim görünüyor, ama sipariş sayfası hâlâ "Onay bekleniyor" diyor. Destek, Orders yönetim ekranını açıyor ve `PAYMENT_PENDING` görüyor. Payments ekibine soruyorlar; onlar veritabanlarında bir `PaymentCaptured` satırı buluyor, outbox da temiz. Orders ekibine soruyorlar; consumer log'larında bu sipariş için hiçbir hata yok. Herkesin dashboard'u yeşil.

Üç mühendis şimdi üç serviste, sipariş numarasıyla log arıyor. Bir log satırında `order-81724`, bir diğerinde `81724` yazıyor; Payments relay'i ise yalnızca kendi iç satır id'sini logluyor. Bir saat sonra event'i Payments outbox'ında, yayımlanmış olarak işaretli buluyorlar. Bir saat daha geçince öğreniyorlar ki Orders'ı besleyen broker topic'inin consumer group'u bir deploy sırasında geri kalmış. Broker'ın retention'ı, group'un henüz okumadığı bir mesaj aralığını silmiş, group'un offset-reset politikası da en son pozisyona atlayacak şekilde ayarlanmış; böylece group, hata vermeden boşluğun ardından devam etmiş. Event vardı, yayımlandı ve hiç okunmadı.

Burada bir reviewer'ın işaretleyeceği türden bir kod hatası yok. Sistem yapılması için kurulduğu şeyi yaptı ve kimse bunu yaparken göremedi. Daha kötüsü, ticket'in var olmasının sebebi bir alıcının herhangi bir alarmdan önce fark etmiş olması ve kimsenin başka kaç siparişin aynı durumda olduğunu bilmemesi. Bu bölüm iki soru soruyor. Bir ekibin "bu sipariş nerede?" sorusunu dakikalar içinde yanıtlayabilmesi için neyi görmesi gerekir ve bir ekip henüz kimsenin şikâyet etmediği siparişleri nasıl bulur?

Problem ve neden zor olduđu

Senkron bir sistemde hata genellikle bir request'e iliřtirilmiř bir stack trace ile gelir. Event-driven bir sistemde ise tek bir iř akıřının iři birka servise, birka veritabanına, bir broker'a ve birka farklı ana yayılır. Hibir tek sre hikyenin tamamını tutmaz, hibir tek log da onu iermez.

 zellik bunu zorlařtırır. Birincisi, **hop'lar zaman aısından birbirinden ayrıktır**. Producer iřini bitirir ve unuttur; consumer saniyeler ya da saatler sonra, farklı bir srete, producer'ın call stack'inden hibir iz tařımadan alıřabilir. Bir request'i i ie ağrılar boyunca izleyen aralar onu ilk mesajda kaybeder.

İkincisi, **en kt hatalar sessizdir**. Blm 3, bir consumer retention sresinden fazla geri kaldıėında broker'ın sildiėi okunmamıř mesajları anlattı. Blm 8, hata vermeden yanlıř iřlenmiř mesajları anlattı. Her iki durumda da hibir řey bařarısız olmaz, dolayısıyla hibir řey alert retmez. Bir sistem, hi gelmeyen bir event'i ancak gelmesi gerektiėini bilerek fark edebilir.

ncs, **saėlıklı grnen bileřenlerin toplamı bozuk bir akıř olabilir**. Relay alıřıyor, broker ayakta, consumer hata bildirmiyor ve bir sipariř yine de takılı kalmıř. Bileřen saėlıėı "bu sre hayatta mı?" sorusunu yanıtlar; iř ise "bu sipariř tamamlandı mı?" diye sorar. Bunlar farklı sorulardır ve bu blmn alıřtıėı yer aralarındaki bořluktur.

Blm 3, bir hatanın semantiėini  soru olarak tanımlamıřtı: ne bařarısız olabilir, bunu nasıl gzlemlersiniz ve hangi garanti ayakta kalır. Bu blm ikinci soruyla ve cevabın "gzlemleyemezsiniz" ıktıėı durumda ne yapılacaėıyla ilgilidir.

Yaklařımın iki yarıřı var. Observability, bir akıřı izleme ve birikmekte olan baskıyı grme imknı verir. **Reconciliation**, yani iki sistemin gereėe dair grřlerini karřılařtırıp farkları onarmak, hibir sinyalin bildirmediėi řeyi bulma imknı verir. Yalnızca ilk yarıya sahip bir ekip grltl hataları grr. Yalnızca ikinci yarıya sahip bir ekip sorunları gnler sonra bulur. Ben ikisini de kurardım.

Bağlamı asenkron hop'lar boyunca taşımak

Her event, Bölüm 11'den beri iki tanımlayıcı taşıyor: bir iş akışındaki her event'in paylaştığı `correlation_id` ve doğrudan tetikleyiciyi işaret eden `causation_id`. Bunlar hikâyenin omurgasıdır. Distributed tracing ise zamanlamayı ve iş birimleri arasındaki parent-child ilişkisini kaydeden ikinci bir iplik ekler.

Bir **trace**, her biri tek bir süreçteki bir iş birimi olan, zamanlanmış span'lerden oluşan bir ağaçtır. Bir süreç içinde tracing kütüphanesi geçerli span'i ortam durumunda (ambient state) tutar. Bir mesajın ötesinde ise, onu mesajın içine koymadıkça bu durum kaybolur. Standart yaklaşım, trace context'ini (bir trace id, bir span id ve bir sampling bayrağı) mesaj header'larına serileştirmektir. W3C Trace Context bunun için `traceparent` ve `tracestate` adlı iki header'dan oluşan ortak bir biçim tanımlar ve çoğu tracing kütüphanesi onu izler. Header kapasitesi ve yayılımı broker'a ve bridge'e göre değişir: bazıları header sayısını ya da boyutunu sınırlar, bazı bridge'ler de header'ları düşürür; bu yüzden mesajlarınızın gerçekte izlediği yolu kontrol edin.

Context'in nerede yakalandığı ilk bakışta görüldüğünden daha önemlidir. Bölüm 4, outbox'a tam da bunun için bir `headers` kolonu koydu. Relay daha sonra, farklı bir süreçte ve ambient span olmadan çalışır; dolayısıyla context'i yayımlama anında enjekte eden bir relay, siparişi veren trace'in yerine kendi trace'ini iletir. Context, outbox satırı yazılırken, producer transaction'ının içinde yakalanmalıdır.

```
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)
```

Consumer tarafında, yeni span'i eskisiyle ilişkilendirmenin iki yolu var. Consumer span'i producer span'inin child'ı olabilir; bu, tek bir ağaç çizer. Ya da yeni bir trace başlatıp producer'ın context'ine bir **link** tutabilir.

Doğru seçim duruma bağlıdır.

Durum	İlişki	Neden	Maliyeti
Tek mesaj, hemen tüketilen	Producer span'inin child'ı	Request'ten etkiye tek bir waterfall	Teslimat geç olursa trace tüm gecikme boyunca açık kalır ve süreler yanılır
Batch ya da fan-in	Yeni trace, her producer için bir link	Bir span'in yalnızca bir parent'ı olur; bu yüzden bir batch consumer'ı on iki producer'ın child'ı olamaz	Backend link'leri desteklemelidir ve biri onlarda gezinmek zorundadır
Fan-out	Bir producer span'i, her subscriber için bir child consumer span'i	Ağaç ağaç olarak kalır	Trace her eklenen subscriber ile büyür
Replay, redrive ya da geç teslimat	Link'li yeni trace	Saatler önceki bir parent, özgün trace'i uzatırdı	Sampling kararı yeniden verilir; dolayısıyla sample'lanmış bir producer trace'i, sample'lanmamış bir consumer trace'ine link verebilir

Yine de link'leri varsayılan yapardım, çünkü gecikme ve redelivery, bu kitabın planladığı normal durumdur. Bedeli, çoğu tracing arayüzünün parent-child için çizdiği tek uçtan uca waterfall'dır. Her zaman producer'ından birkaç an sonra çalışan bir consumer için parent-child daha iyi seçimdir ve onu seçerdim.

```
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
```

Span'in `event_id`'yi ve `delivery attempt`'i kaydettiğine dikkat edin. Log-based broker'lar teslimat sayısı tutmaz, queue-based broker'lardan da yalnızca bazıları bunu sunar; bu yüzden sayıya güvenmeden önce nereden geleceğine karar verin. Bir mesaj yeniden teslim edildiğinde aynı `event_id`'li ikinci bir span elde edersiniz. Bu çift, mesajın iki kez teslim edildiğini söyler; iki kez uygulanıp uygulanmadığını ise aşağıda anlatılan `outcome` alanı söyler.

Tracing'in sınırları var ve garanti için ona yaslanmamaya dikkat ederdim. Backend'ler sampling yapar, dolayısıyla ilgilendiğiniz o tek sipariş için trace mevcut olmayabilir. Context header'ları bir bridge, bir dead-letter kopyası ya da elle yazılmış bir producer tarafından düşürülebilir. Header'lara bağımlı kalmamanın bir yolu, trace tanımlayıcısını gövdede taşımaktır: BabelQueue envelope'u (babelqueue.com), açık kaynaklı, dilden bağımsız bir standart olarak mesaj gövdesinde bir `trace_id` taşır; bu `trace_id`'yi ilk producer üretir, sonraki producer'lar değiştirmeden kopyalar ve dead-lettering'den sağ çıkar; böylece bir hop header'ları düşürse bile iz korunur. Bir outbox satırında biten trace, o satırı henüz okumamış bir consumer'a hiç ulaşmaz. Bu yüzden envelope'daki tanımlayıcılar tracing kütüphanesinden daha önemlidir: `correlation_id`, event'i işleyen her log satırında ve her veritabanı satırında bulunmalıdır; o zaman tracing kapalıyken bile çalışır.



Diyagramı değişmeyen şey için okuyun: `correlation_id` her hop'ta aynı kalır, her servis ise kendi outbox'ında yeni bir trace context saklar.

Tek bir siparişin hikâyesini yeniden kurmak

Tanımlayıcılar yerindeyken "81724 numaralı sipariş nerede?" sorusu bir sorguya dönüşür. En ucuz uygulama bir ürün değil, bir konvansiyondur: her servis, işlenen her event için ve yayımlanan her event için, alan olarak `correlation_id`, `event_id`, `event_type`, `causation_id`, sonuç ve zaman damgasını içeren tek bir yapılandırılmış log satırı yazar. Bu log'larda `correlation_id = "order-81724"` araması, her servisteki her hop'u verir.


```
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
}
```

outcome alanı dikkati hak ediyor. Küçük, kapalı bir küme kullanın: `applied`, `duplicate_skipped`, `stale_ignored`, `failed_retry` ve `dead_lettered`. Bölüm 9, yok sayılan eski (stale) bir event'in bir iz bırakmayı hak ettiğini savundu; Bölüm 7 ise `duplicate_skipped` sonucunu üreten inbox kontrolünü ele alıyor. Bu sonuçlar birer alan olduktan sonra "bu consumer geçen saat kaç event'i stale diye düşürdü?" sorusu bir soruşturma değil, bir sayıdır.

`occurred_at`'e göre sıralamak bir zaman çizelgesi verir, `causation_id`'yi izlemek ise bir ağaç: `OrderPlaced`, `PaymentAuthorized`'a yol açtı; o da `StockReserved` ile birlikte `CapturePayment`'a yol açtı ve böyle sürer. Açılıştaki senaryoda ağaç, Payments tarafından yayımlanmış bir `PaymentCaptured` ve Orders'da hiç `consumed` satırı olmadığını gösterirdi. Eksik olan ilk hop bakılacak yerdir. Bu, herhangi bir tek log satırından daha faydalıdır: zincirdeki bir boşluk bir şeyin kaybolduğunun kanıtıdır, uzun bir zincir ise yalnızca akışın ilerlediğini gösterir.

Bölüm 10'daki bir saga işletiyorsanız, onun `saga_instance` satırı ikinci bir kaynaktır. "Orchestrator durumun ne olduğuna inanıyor?" sorusunu o yanıtlar, "gerçekte ne oldu?" sorusunu ise log'lar. İkisi çeliştiğinde, bir ticket'a değer bir şey bulmuşsunuz demektir.

En önemli akışlar için bir **event journal** de tutardım: her servisin inbox'ının yanına yazdığı, payload yerine payload'un hash'ini içeren event envelope'larından oluşan, yalnızca eklenebilir bir kayıt. Depolama ve biraz yazma yükü maliyeti vardır; karşılığında, broker event'leri unuttuktan sonra bile, yeniden kurulmuş bir event'in hâlâ özgün olanla eşleşip eşleşmediği de dahil (aşağıdaki onarım bölümüne bakın), soruları yanıtlayabilirsiniz. Retention'ı, en yavaş reconciliation'ınızın çalışabileceği kadar uzun tutun.

Önemli metric'ler

Metric'ler, tek bir mesaj bozulmadan önce baskıyı anlatır. Event-driven bir sistemde faydalı olanların listesi kısadır ve çoğuna önceki bölümlerde zaten değinilmişti. Şunlarla başladım.

Sinyal	Neyi ölçer	Neden önemli
Consumer lag (mesaj sayısı)	Consumer'ın baştan ne kadar geride olduğu	Backlog büyüklüğü; kapasite için faydalı
Mesaj yaşı, consumer tarafı	En eski işlenmemiş mesajın üretilmesinden bu yana geçen süre	İşin deneyimlediği şey
Outbox yaşı	Yayımlanmamış en eski satırın yaşı	Relay sağlığı (Bölüm 5)
End-to-end latency	Akışın ilk <code>occurred_at</code> değerinden son consumer'daki etkiye kadar geçen süre	Akışın gerçek deadline'ı
Retry oranı ve attempt sayıları	Birden fazla denemeye ihtiyaç duyan mesajların payı	Retry'ların gizlediği bozulma (Bölüm 6)
DLQ girişleri ve yaşı	Yeni dead letter'lar ve incelenmemiş en eskisi	İnsan gerektiren hatalar (Bölüm 8)
Yok sayılan ve duplicate sayıları	Düşürülen stale event'ler, atlanan duplicate'ler	Ordering ve redelivery baskısı
Drain rate ve yetiştirme süresi	İşleme hızının geliş hızına oranı; lag'in net drain rate'e bölünmesi	Bir backlog'un deadline ya da retention sınırı dolmadan eriyip erimesi (Bölüm 6)

Lag ve mesaj yaşının neden daha iyi olduğu

Mesaj cinsinden **consumer lag**, çoğu dashboard'un ilk gösterdiği sayıdır ve yanlış okunması en kolay olanıdır. 10.000 mesajlık bir lag tek başına pek bir şey ifade etmez: consumer saniyede 5.000 mesajı boşaltıyorsa iki saniye geridedir, saniyede 10 boşaltıyorsa çeyrek saat. Mesaj cinsinden lag bir kapasite sorusunu yanıtlar. Zaman cinsinden lag ise iş sorusunu yanıtlar.

Mesaj yaşı, hâlâ bekleyen en eski mesajın üretilmesinden bu yana geçen süredir. Bazı queue tabanlı broker'lar en eski mesajın yaşını doğrudan sunar; diğerlerinde bekleyen en eski mesajın bir zaman damgasından türetin. Log tabanlı bir broker'da, bunu consumer'ın commit edilmiş

offset'indeki kaydın zaman damgasından türetin. Her iki durumda da sayı saniye cinsindendir, dolayısıyla işin seçtiği bir deadline ile karşılaştırılabilir.

Hangi zaman damgasını çıkardığınız önemlidir, çünkü adaylar farklı şeyleri ölçer. Broker'ın append zamanı, bir mesajın broker'da ne kadar beklediğini ölçer; bu, consumer'ın sağlığıdır. `occurred_at`, olgunun gerçekleşmesinden bu yana geçen süreyi ölçer; bir satırın outbox'ta relay'i beklerken geçirdiği süre de buna dahildir ve bu, işin gecikme görüşüdür. Producer'ın create-time'ı kurulumla bağlıdır: yapılandırmaya bağlı olarak bir log broker ya onu ya da append zamanını saklar ve eski bir satırı yeniden yayımlayan bir relay, açıkça bir değer vermedikçe taze bir create-time elde eder. Consumer-yaş gauge'u için append zamanını, end-to-end latency için `occurred_at`'i kullanırdım. Gauge için `occurred_at` kullanmak, relay eski bir satırı yeniden yayımladığı anda sağlıklı bir consumer'ı çok geç kalmış gösterir. Bir group'un yaşı, partition'ları arasındaki maksimumdur ve alert'ler bu maksimumu kullanmalıdır.

Bir tuzağın adını koymaya değer. Yeni trafik almayan bir partition'ın lag'i sıfırdır ve naif bir hesaplama yaşı ya sıfır ya da sonsuza dek büyüyen bir değer çıkar. Boştaki bir partition'ın nasıl rapor vereceğine karar verin ve bunu test edin. Aynı fonksiyon aşağıda anlatılan retention hatasından da sağ çıkmalıdır: log'un başlangıcının gerisine düşmüş commit edilmiş bir offset silinmiş veriyi işaret eder; bu yüzden bu kontrol ilk sıradadır ve ayrıca raporlanır. Bölüm 8'deki gibi bir poison record'a takılmış bir consumer sıfır throughput ve yükselen bir yaş gösterir; birini page ettirmesi gereken şey bu kombinasyon olmalıdır.

```

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)

```

Retention, göremediğiniz deadline'dır

Mesaj yaşının ikinci, daha az bariz bir görevi var. Log tabanlı bir broker'da, retention süresinden fazla geri kalan bir consumer mesajları sessizce kaybeder: broker hiç okunmamış veriyi siler ve consumer offset-reset politikasının onu bıraktığı yerden devam eder. Queue tabanlı bir broker'da aynı şey mesajlar expire olduğunda olur. Bölüm 3 bu hatayı adlandırdı ve ona hiçbir hata eşlik etmez.

Bu reset politikası bir karardır ve her seçeneğin bir bedeli vardır. En erken mevcut offset'e reset etmek, geriye kalan her şeyi yeniden oynatır; dolayısıyla kayıp kısımdır ve consumer'ın eski mesajlardan oluşan bir seli sindirmesi gerekir. En son offset'e reset etmek ileri atlar ve kaybı sessiz kılar; açılıştaki senaryoda olan buydu. Başarısız olmak, yani consumer'ın durup bir insanı beklemesi, sessiz bir kaybı bir incident'e dönüştürür. Bir boşluğun paraya mal olduğu her akış için başarısız olmayı seçerdim ve biri karar verene kadar consumer'ın kapalı kalmasını kabul ederdim. Bölüm 14, boşluğu fark ettikten sonra ondan nasıl toparlanılacağını anlatır: bir zaman damgasına reset edip yan etkileri bastırarak replay yapmak.

Savunma aritmetiktir. Mesaj yaşını retention ya da expiry süresiyle karşılaştırın ve yaş bunun bir kesrine, diyelim yarısına ulaştığında alert verin. O noktada deadline hâlâ ileridedir ve bir insanın consumer eklemek, bir producer'ı durdurmak ya da retention'ı uzatmak için vakti vardır. Deadline geçtikten sonra tek kanıt event'lerin yokluğudur ve reconciliation tam da bunu tespit etmek için vardır. Oranı yaşın kendisiyle aynı dashboard'a koyardım, çünkü iş için kabul edilebilir ama retention için ölümcül bir lag gerçek bir durumdur.

Bir CDC connector'ü için eşdeğer sinyal, connector'ün veritabanının log'unun ne kadarını henüz tüketmediği ve veritabanının onun için ne kadarını tutmaya razı olduğudur. Veritabanının log'unu tuttuğundan daha uzun süre kapalı kalan bir connector'ün toparlanmak için yeni bir snapshot'a ihtiyacı olabilir ve bu başlı başına bir kesintidir.

Latency, retry'lar ve gerisi

End-to-end latency'nin bir grafikten önce bir tanıma ihtiyacı var. Ben bunu, bir akıştaki ilk event'in `occurred_at` değerinden son etkinin commit edildiği ana kadar ölçerdim; örneğin `OrderPlaced`'ten `OrderConfirmed`'a. Ortalama değil, percentile'lar raporlayın: dağılımın yavaş ucu, destek ekibine yazan alıcılardır. Son consumer yalnızca tükettiği envelope'u görür ve o envelope, akışın başlangıcını değil kendi olgusunun zamanını taşır. Ya akışın başlangıç zamanını bir header'da taşıyın ya da onu `correlation_id` ile event journal'da veya sahibi olan serviste arayın. Sonra ölçümü son `event_type` ile etiketleyin. Her iki zaman damgası da farklı servislerin saatlerinden gelir, bu yüzden sonuçta biraz skew bekleyin; bu, bir iş deadline'ının yanında küçüktür ve saniyenin altındaki farklara alert kurmamak için bir sebeptir.

Drain rate, backpressure'ın gözlemlenebilir biçimidir. Lag'i, consumer'ın mesajları işleme hızı ile mesajların geliş hızı arasındaki farka bölerseniz yetiştirme süresini elde edersiniz. Gelenler işlenenleri aşıyorsa lag hiç erimez ve alert kurulacak sayı budur; yaş retention sınırına ulaşmadan çok önce. Bölüm 3 backpressure'ı tanımladı, Bölüm 6 ise bir consumer'ın üstlendiği işin nasıl sınırlanacağını ele alıyor.

Retry oranı ve attempt sayıları, Bölüm 6'nın gizli hatalar dediği sorunu görünür kılar: retry'lar onu sessizce emerken haftalardır bozulmuş olan bir bağımlılık. Attempt sayısı histogramı hata sayısından daha bilgilendiricidir,

çünkü her seferinde üçüncü denemede başarılı olan bir mesaj hiç hata üretmez.

DLQ girişleri ve incelenmemiş en eski dead letter'ın yaşı Bölüm 8'den gelir. Bölüm 7 ve 9'dan **duplicate ve stale sayılarını** ekleyin. Bir deploy'un hemen ardından yükselen duplicate sayısı normaldir, çünkü işin sahipliği el değiştirir; sakın bir saatte yükselen sayı ise yavaş handler'lara işaret eder. Queue tabanlı bir broker'da olağan neden, handler'ın çalışma süresinden kısa bir visibility timeout'tur. Log tabanlı bir broker'da bu bir rebalance'tır ya da poll timeout'unu aşıp partition'larını kaybeden bir consumer'dır.

Anlamlı alert'ler

Çoğu alerting kurulumunun başarısızlığı çok az veri olması değildir. Verinin insanları etkiler için değil sebepler için page etmesi ve birkaç haftalık gürültüden sonra kimsenin pager'a güvenmemesidir. Alert'leri tek bir ölçüte tabi tutardım: bu alert sabahın üçünde ateşlenirse, adı belli bir kişi ne yapılacağını söyleyebilir.

Bir iş deadline'ına bağlı semptomlar üzerine alert kurun.

"Yayımlanmamış en eski outbox satırı iki dakikadan eski" ifadesi, bir durum değişikliği ile duyurusu arasında bir şeyin takıldığıнын semptomudur. "Consumer CPU'su yüzde 80'in üzerinde" bir sebeptir ve önemli olabilir de olmayabilir de. Sebep metric'lerini birinin araştırma yaptığı dashboard'larda kullanın, pager'ı semptomlara saklayın.

Her alert'e bir **sahip ve gerekçesi olan bir eşik** verin. Bölüm 8, DLQ'lar için aynı noktayı vurguladı: kaybolan bir e-posta ile kaybolan bir capture farklı eşikleri hak eder ve fark alarının yanına yazılmalıdır. Gerekçe önemlidir, çünkü eşikler kayar ve gerekçesi belirtilmemiş bir sayı, ilk rahatsız ettiğinde yükseltilir.

Varlığın yanı sıra yokluk için de alert kurun. Çöken bir relay hata üretmez; hiçbir şey üretmez. Tüm üyelerini kaybetmiş bir consumer group lag gösterir, ama event almayı tamamen bırakmış bir topic ne lag gösterir ne hata. Mesai saatlerindeki siparişler gibi düzenli bir ritmi olan akışlar için, "10 dakikadır hiç OrderPlaced event'i gözlemlenmedi" gibi bir alert,

diğer tüm sinyallerin kaçırdığı producer tarafı sessizliğini yakalar. Eşiğin trafik örüntüsünü izlemesi gerekir ve ritmi olmayan akışlar için yanlış araçtır.

Son olarak, çoğu ekip için iki kademe yeterlidir: bir deadline'ı tehdit eden ya da veri kaybettiren her şey için bir page ve düşük değerli mesajlardan oluşan büyüyen bir DLQ gibi yavaş baskı için bir ticket. Üçüncü bir kademe göz ardı edilme eğilimindedir.

Uyarı: Gürültüyü durdurmak için bir alert eşiğini yükseltmek, bir ekibin bir rahatsızlığı bir kör noktayla takas etme yoludur. Bir alert çok sık ateşleniyorsa, sebebini ya da tanımını düzeltin. Ateşlemeyi bırakana kadar sayıyı yükseltmeyin.

Reconciliation

Metric'ler ve trace'ler sistemin ne yaptığını gözlemler. Sisteme hiç girmemiş bir event'i, hata vermeden yanlış durumu commit etmiş bir handler'ı ya da birinin elle uyguladığı bir onarımı size söyleyemezler. Bunlar için sistemi, neyin doğru olması gerektiğine dair bağımsız bir kayıtlarla karşılaştırırsınız. Reconciliation budur.

Bir reconciliation job'ı, aynı olguların iki görünümünü alır ve farkları raporlar. Çiftler, aramaya başladığınızda genellikle bellidir:

- **Orders'a karşı Payments:** PAID durumundaki her siparişin capture edilmiş bir ödemesi vardır ve capture edilmiş her ödeme, onu bilen bir siparişe aittir.
- **Payments'a karşı Ledger:** her PaymentCaptured'ın aynı para biriminde aynı tutar için eşleşen bir LedgerEntryPosted'ı vardır.
- **Kayıtlarınıza karşı bir sağlayıcının kayıtları:** ödeme sağlayıcısının settlement raporu ile kendi capture kayıtlarınız. Bölüm 7, idempotent hâle getirilemeyen etkiler için buraya işaret etti; orada reconciliation son savunma hattıdır.
- **Bir read model'e karşı kaynağı:** Bölüm 9'un şüpheli olarak işaretlenip daha sonra onarılabileceğini söylediği bir arama indeksi ya da bir view.

Üç fark türü çoğu durumu kapsar: **missing** (kaynakta var, türetilmiş tarafta yok), **stale** (ikisinde de var ama türetilmiş taraf aynı fikirde değil, genellikle geride olduğu için) ve **extra** (türetilmiş tarafta kaynağın bilmediği bir şey var). Her biri farklı bir sebebe işaret eder. Missing, kaybolmuş ya da hiç tüketilmemiş event'leri düşündürür; stale, ordering ya da lag'i, ara sıra da yanlış değer yazmış bir handler'ı düşündürür; extra ise bir bug'ı ya da elle yapılmış bir değişikliği.

Özen gerektiren kısım şudur: uçuştaki event'ler tıpkı uyumsuzluklar gibi görünür. Dört saniye önce capture edilmiş ve henüz Ledger'da olmayan bir ödeme normaldir. Bunu raporlayan bir job gürültü üretir ve insanlara onu görmezden gelmeyi öğretir. Bu yüzden job yalnızca normal end-to-end latency'nizden seçilmiş bir **grace period**'dan daha eski kayıtları karşılaştırır ve bir uyumsuzluğu raporlamadan önce ikinci kez doğrular.


```

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)

```

Bu bloğun birkaç ayrıntısı doğruluğu taşır. Pencere sınırlıdır ve kesme noktası uygulanır, dolayısıyla job oturmaya vakit bulmuş veriyi okur. Her batch, veritabanları arasında bir join yerine anahtarlı bir sorgudur; çünkü Parcelly servisleri birbirlerinin tablolarını okumaz, bu yüzden job diğer tarafı o servisin kendi arayüzü ya da yayımlanmış bir extract aracılığıyla getirir. Job yalnızca uyumsuzlukları kaydeder ve hiçbir şeyi değiştirmez. Ve öge ile tür üzerinden anahtarlanmış bir discrepancy tablosuna upsert yaptığı için, iki kez çalıştırmak ya da bir pencerenin yarısında çöktükten sonra çalıştırmak aynı tabloyu üretir; büyüyen bir duplicate satır yığını oluşmaz. Aynı fonksiyon, iki taraf aynı fikre vardığında discrepancy'leri kapatır; aşağıdaki onarım kodunun dayandığı adım budur. Ledger'a iki kez

işlenmiş bir ödeme, eşleşmeleri sayan ileri geçiş tarafından yakalanır. Ters geçiş ise arkasında hiçbir capture olmayan bir "extra" kaydı bulan şeydir; hiç capture edilmemiş bir ödeme için yapılmış bir Ledger kaydı gibi; tek yönlü bir kontrol bunu asla bulamaz.

Kadansı, bir uyumsuzluğun vereceği zarara göre seçin. Payments'a karşı Ledger finansaldır ve sık bir çalıştırmayı hak eder; bir backstop olarak da günlük tam bir karşılaştırma yapılır. Bir arama indeksi için gecelik bir çalıştırma gerekebilir. Job'ın kendi sağlığı da önemlidir: beklenen penceresinde tamamlanmadıysa alert verin, çünkü sessizce durmuş bir reconciliation hiç olmamasından kötüdür; insanlar onun izlediğine inanır.

Not: Bir reconciliation job'ı kontrol ettiği verinin bir consumer'ıdır ve yanılabilir. Bir uyumsuzluğa önce bir soruşturma ipucu olarak yaklaşın. Yeni bir job'ın ilk haftalarının çoğu, zamanlamadan, yuvarlamadan ya da işe uymayan bir karşılaştırma kuralından kaynaklanan false positive'leri bulmakla geçer.

Repair ve backfill

Bir uyumsuzluğu bulmak işin yarısıdır. Diğer yarısı, ikinci bir uyumsuzluk yaratmadan durumu eşleştirecek şekilde değiştirmektir.

Bağlı kalacağım kural, **onarımların normal değişikliklerle aynı yoldan geçmesidir**. Ledger'da capture edilmiş bir ödeme için bir kayıt eksiğe, satırı bir veritabanı konsolu ile eklemeyin. Bölüm 3, operatör eylemlerini hata modelinin bir parçası olarak ele aldı; elle düzenlenmiş bir satır event yaymaz, dolayısıyla o olgunun diğer tüm consumer'ları yanlış kalır ve bir sonraki reconciliation kendi yarattığınız bir tutarsızlığı raporlar. Bunun yerine, olguyu ya da onu doğuran komutu producer'ın outbox'u üzerinden yeniden yayımlayın ki consumer'lar onu her şeyi ele aldıkları gibi ele alsın.

Bir onarım yeniden denenebilir, kesintiye uğrayabilir ve iki mühendis tarafından iki kez çalıştırılabilir; her durumda güvenli kalmalıdır. Bölüm 7'nin idempotency'si buna ancak onarım event'i **özgün event_id**'yi taşıyorsa yanıt verir. Yeni bir id'li onarım event'i, her consumer için farklı bir event'tir. İlk event yalnızca gecikmişse onarımdan sonra gelir ve Ledger

ödemeyi iki kez işler. Bu yüzden onarım, zaten var olan olguyu yeniden yayımlar: aynı `event_id` ve `causation_id` ile aynı envelope; ikinci gelen kopya hangisiyse inbox onu duplicate olarak ele alır.

Bu, producer'ın özgün id'yi saklamasını gerektirir. En basit yol, outbox satırı yazılırken `event_id`'yi iş kaydında (burada capture'da) payload'un hash'i ile birlikte saklamaktır. Outbox satırı hâlâ varsa yayımlanmamış olarak işaretlenir ve relay onu tekrar gönderir. Temizlenmişse envelope, saklanan id ile kayıttan yeniden kurulur.

Yeniden kurmanın bir sınırı var. Kayıt güncel durumdur, event ise geçmişten bir olgu. Kayıt o zamandan beri değiştiyse, diyelim kısmi bir iade tutarı düzeltti, ondan yeniden kurulan payload aynı `event_id`'yi taşıırken özgün olandan farklı olur. Hash'in saklanmasının sebebi budur: onarım payload'u yeniden kurar, hash'ini saklanan hash ile karşılaştırır ve farklıysa bir insan için durur. Saklanmış id'si ya da hash'i olmayan bir kayıt, ki bunlar genellikle bu alışkanlık yerleşmeden yazılmış olanlardır, iş anahtarından türetilmiş bir id'ye ve değişemeyecek alanlara başvurmak zorundadır ve o zaman consumer ayrıca Bölüm 7'nin anlattığı gibi bu anahtar üzerindeki bir unique constraint ile korunmalıdır. Ben bu constraint'i Ledger'a zaten ikinci bir savunma hattı olarak ekledim.

Bir yeniden yayımlama ayrıca **fan-out** yapar. Topic, envelope'u yalnızca Ledger'a değil, her subscriber'a teslim eder. Inbox'ında hâlâ `event_id` bulunan bir subscriber onu atlar. Inbox satırı temizlenmiş olan, çünkü event dedup penceresinden (Bölüm 7) eski olan bir subscriber ise onu ikinci kez uygulamalar ve Notifications aynı e-postayı yeniden gönderebilir. Dolayısıyla iki güvenli yol var. Event, tüm subscriber'lar arasındaki en kısa dedup penceresinden gençse, yukarıdaki gibi producer'ın outbox'ı üzerinden yeniden yayımlayın. Daha eskiyse, onarımı yalnızca onu kaçırın consumer'a teslim edin: o consumer'ın kendi onarım queue'su ya da topic'i üzerinden veya yalnızca o group'un offset'lerini reset ederek, ve koruma olarak bir iş anahtarı constraint'ine güvenin. Bölüm 8, aynı ilkeyi redrive için uygular.

```

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

```

İki ayrıntı güvenliği taşır. Yayımlanmamış bir satırı tekrar yayımlanmamış olarak işaretlemek tekrarlandığında zararsızdır ve eşzamanlı ikinci bir insert'ten gelen unique violation bir hata olarak değil, "zaten ele alındı" olarak değerlendirilir. Yeniden yayımlanan envelope ayrıca özgün `causation_id`'sini korur, böylece causation zinciri doğru kalır ve discrepancy'ye olan bağlantı onarım kaydında yaşar. Son yorum önemli olandır. Onarım fonksiyonu discrepancy'yi çözülmüş olarak işaretlemez. Bunu iki taraf aynı fikre vardığında bir sonraki reconciliation çalıştırması yapar. Bu, yayımlanıp sonra kaybolmuş bir onarımın düzeltilmiş olarak raporlanmasını önler.

Genellikle **backfill** denen toplu onarım, ölçek ekler. Canlı bir consumer'a 30.000 event'lik bir backfill, normal trafikle rekabet eder ve akışın latency'sini deadline'ının ötesine itebilir. Bölüm 8'in redrive için önerdiği

gibi, onu küçük batch'ler hâlinde, bir rate limit ve bir duraklatma anahtarıyla çalıştırın ve çalışırken mesaj yaşını ve consumer lag'i izleyin. Log tabanlı bir broker'da, türetilmiş bir view'ı yeniden kuran ayrı bir consumer group'a bilinçli bir replay yapıp ardından geçiş yapmayı tercih edebilirsiniz. Consumer bir projection olduğunda bu daha temizdir; consumer e-posta göndermek gibi yan etkilere sahip olduğunda ise yanlıştır.

Producer üzerinden onarım bir avuç kayıt için uygundur. Hasar daha geniş olduğunda, örneğin bir consumer group'un koca bir retention boşluğunu atlaması, geri yüklenmiş bir veritabanı ya da kaybedilmiş bir broker cluster'ı durumunda, kayıt kayıtlarını yeniden yayımlamak doğru araç olmaktan çıkar. Bölüm 14, replay ile toparlanmayı, felaketleri ve broker geçişini ele alıyor.

Her onarımın bir kaydı olmalı: kim çalıştırdı, hangi discrepancy'leri kapsadı, ne yayımladı ve ne zaman. Denetim izi, bir sonraki mühendisin 14:10'daki duplicate sıçramasının bir backfill olduğunu öğrenme yoludur ve bir düzenleyicinin ya da finans ekibinin bir Ledger kaydının bir düzeltme olduğunu sonradan öğrenme yoludur.

İş invariant'ları

Reconciliation iki sistemi karşılaştırır. Bir **invariant kontrolü** ise bir sistemi işin doğru kabul ettiği bir kuralla karşılaştırır. Invariant'ları değerlendirmek çoğu zaman daha ucuzdur ve hiçbir ikinci sistemin yakalayamayacağı sorunları yakalar; mesela iki sistemin yanlış cevap üzerinde anlaşmasına yol açan bir bug gibi.

Parcelly için birkaç örnek biçimi gösterir:

- Capture edilmiş bir ödemesi olmayan hiçbir sipariş **CONFIRMED** değildir.
- Hiçbir rezervasyon deadline'ından uzun süre tutulmaz.
- Hiçbir saga instance'ı **deadline_at**'ini geçecek şekilde **AWAITING_RESULTS**'ta durmaz.
- Capture edilmiş her ödemenin tam olarak bir Ledger kaydı vardır; sıfır değil, iki de değil.
- Bir siparişin Ledger kayıtlarının toplamı sıfıra denkleşir.

Her biri hiç satır döndürmemesi gereken bir sorgudur. Onu bir zamanlamaya göre çalıştırın ve dönen her satırı bir sahibi olan bir discrepancy olarak ele alın. Saga örneği, Bölüm 10'un açık bıraktığıdır: timeout handler'ı donmuş bir siparişe karşı ilk korumadır ve takılı instance'lar üzerindeki bir invariant sorgusu ikincisidir, çünkü ateşlenmesi gereken handler'ın kendisi de başarısız olmuş olabilir.

```
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)
```

Invariant ve öge ile anahtarlama, süren bir ihlalin çoğaltılmak yerine güncellendiği anlamına gelir ve `first_seen_at` bir sorunun ne kadar süredir açık olduğunu söyler. Bu yaş, başlı başına alert kurmaya değer bir metric'tir.

Invariant'ları ürün ve finans insanları için değil, onlarla birlikte yazdım. Mühendisler sistemin neler yapabildiğini bilir ve işin neyi kusur saydığını her zaman bilmez. "Bir iade asla capture'ı aşmaz" gibi bir kural finans için barizdir ve kodun hiçbir yerinde yazılı olmayabilir.

Trade-off'lar ve maliyetler

Observability ve reconciliation bedava değildir ve kendi kendilerine çalışmazlar.

Telemetry'nin bir bedeli var. Her işlenen event için yapılandırılmış log'lar log hacmini katlar ve metric'ler üzerindeki `order_id` gibi yüksek kardinaliteli label'lar bir metric backend'ini boğabilir. Tanımlayıcıları log'larda ve trace'lerde tutun, metric label'larını servis, topic ve event type gibi küçük kümelerle sınırlayın. Trace'lere sampling uygulayın ve ihtiyacınız olan trace'in mevcut olmayabileceğini kabul edin.

Bir event journal veriyi çoğaltır. "Ne oldu?" sorusunun yasal ya da finansal ağırlığı olan akışlar için buna değer. Kozmetik bir bildirim gibi düşük değerli bir akış için log'lar yeterlidir ve journal, okuyucusu olmayan bir maliyettir.

Reconciliation production verisini okur. Bir karşılaştırma job'ı kaynak veritabanını yükleyebilir ve batch'lemeye özen göstermeyen bir job, koruduğu servisi yavaşlatabilir. Elinizde varsa bir replica üzerinde çalıştırın ve batch boyutunu ve hızını sınırlayın. Ayrıca bir bağımlılık ekler: diğer tarafın arayüzü kapalıysa job çalışamaz.

Alert'ler dikkat harcar. Her page, bir sonrakinin itibarını harcar. İyi sahiplenilmiş küçük bir alert kümesi, kimsenin okumadığı büyük bir kümeden daha fazlasını yapar.

Onarımın kendi riski var. Bir onarım mekanizması, production durumunu normal request akışının dışında değiştirmenin bir yoludur. Denetim, tehlikeli türler için onaylar ve idempotent bir tasarım olmadan, bir problem sınıfını daha kötüsüyle değiştirir.

Tüm bunları ne zaman kurmamalı: tek servisli ve tek consumer'lı, riski düşük ve akışları kısa bir sistemin bir event journal'a, invariant tablolarına ya da gecelik bir reconciliation'a ihtiyacı yoktur. `correlation_id`li yapılandırılmış log'lar ve mesaj yaşı üzerine bir alert, her ölçekte bağlı kalacağım asgaridir. Gerisi, sessiz bir hatanın maliyetiyle birlikte büyür. Kaybolan bir event'in maliyeti bir ledger uyumsuzluğuyorsa önce reconciliation'ı kurun. Maliyeti eski bir arama sonucuysa onu en sona bırakın.

Sık yapılan hatalar

- **Akış üzerine değil, sağlık üzerine alert kurmak.** Yeşil CPU ve bellek grafikleri süreçlerin hayatta olduğunu söyler ve siparişlerin tamamlanıp tamamlanmadığı hakkında hiçbir şey söylemez. Cazip gelir, çünkü altyapı metric'leri zaten vardır. Semptom alert'lerini ekleyin: mesaj yaşı, outbox yaşı ve invariant ihlalleri.
- **Lag'i yalnızca mesaj cinsinden izlemek.** Bir mesaj sayısına bağlı bir deadline yoktur. Yaşa çevirin ve yaşı iş deadline'ı ve retention süresiyle karşılaştırın.
- **Trace context'ini relay'de enjekte etmek.** Relay'in ambient span'i yoktur, bu yüzden yanlış olanı ilişitir ya da hiçbirini. Context'i outbox satırı yazılırken yakalayın ve değiştirmeden iletin.
- **Tutarsız tanımlayıcılar.** Bir servis `order-81724` loglar, bir diğeri `81724`. Correlation ile arama, tam da gerektiği anda başarısız olur.

Biçimi event tasarımında bir kez tanımlayın ve paylaşılan bir logging helper'ı ile zorunlu kılın. Bu tek biçimi sentetik trafik için de koruyun: Bölüm 15, probe'ları ayrı bir header ile işaretler, asla özel bir `correlation_id` ile değil.

- **Her yaş için tek bir zaman damgası kullanmak.** Producer create-time, broker append zamanı ve `occurred_at` farklı gecikmeleri ölçer. Consumer-yaş gauge'u için append zamanını, end-to-end latency için `occurred_at`'i kullanın.
- **Bir onarımı her subscriber'a yeniden yayımlamak.** Inbox satırı temizlenmiş bir consumer event'i yeniden uygular. Event'i en kısa dedup penceresiyle karşılaştırın ya da onarımı yalnızca onu kaçıran consumer'a teslim edin.
- **Grace period olmadan reconcile etmek.** Job, uçustaki her event'i bir uyumsuzluk olarak raporlar, ekip onu görmezden gelmeyi öğrenir ve tek gerçek bulgu gürültüde kaybolur. Yalnızca oturmuş kayıtları karşılaştırın ve raporlamadan önce yeniden kontrol edin.
- **Satırları elle düzenleyerek onarmak.** Bir tabloyu düzeltir, event yamaz ve her downstream consumer'ı yanlış bırakır. Onarımı producer ve outbox üzerinden yapın ki değişiklik diğerleri gibi yol alsın.
- **Idempotent olmayan bir onarım.** İkinci bir çalıştırma düzeltmeyi çoğaltır. Özgün `event_id` ile yeniden yayımlayın ki inbox ikinci bir çalıştırmayı da geç gelen özgün event'i de tanınsın.
- **Reconciliation'ın çürütmesine izin vermek.** Job bir schema değişikliğinden sonra bozulur, kimse fark etmez ve ekip kapsandığına inanır. Bir çalıştırma geç kaldığında ya da başarısız olduğunda alert verin ve discrepancy backlog'unu bir zamanlamaya göre gözden geçirin.

Temel çıkarımlar

- Servislere yayılmış bir akış, her hop'tan sağ çıkan tanımlayıcılara ihtiyaç duyar: her log satırında ve satırda `correlation_id` ve relay'in doğru olanı iletmesi için outbox satırında saklanan trace context'i.
- Tracing yardımcı olur ama sampling'e tabidir ve düşürülebilir, link'lere karşı parent-child ise duruma göre bir ödünleşimdir; bu yüzden

envelope tanımlayıcılarının tek başına bir siparişin hikâyesini yeniden kurmaya yetmesi gerekir.

- Bir iş deadline'ına ve broker retention'ına karşı ölçülen mesaj yaşı, mesaj cinsinden lag'den daha faydalıdır ve geride kalan bir consumer'ın event'leri sessizce kaybetmesinden önce sizi uyaran sinyaldir; offset-reset politikası da böyle bir kaybın sessiz mi yoksa gürültülü mü olacağını belirler.
- Deadline'lara bağlı semptomlara alert kurun, her alert'e bir sahip ve eşiği için belirtilmiş bir gerekçe verin ve sessizlik için de alert ekleyin; çünkü veri kaybeden hatalar çoğu zaman hiç sinyal üretmez.
- Reconciliation, hiçbir metric'in raporlamadığını bulur: gerçeğin iki görünümü arasındaki oturmuş kayıtları karşılaştırır, farkları missing, stale ya da extra olarak sınıflandırır ve hiçbir şeyi değiştirmeden kaydeder.
- Onarımlar normal yoldan geçmeli, geç gelen bir özgün event ile bir onarımın ikisinin birden etki edememesi için özgün `event_id`'yi yeniden yayımlamalı ve bir denetim izi taşınmalıdır; bir discrepancy da onarımın kendisiyle değil, bir sonraki başarılı karşılaştırmayla kapanır. Bir yeniden yayımlama her subscriber'a ulaşır; bu yüzden eski event'ler yalnızca onları kaçıran consumer'a gider.
- Capture edilmiş ödemesi olmayan onaylı sipariş olmaması gibi iş invariant kontrolleri, aynı fikirde olan iki sistemin gizleyeceği kusurları yakalar.

Gözlem ve reconciliation, sistemin yanlış davrandığını size söyler, onarım ise bir avuç kaydı düzeltir. Bölüm 14, hasar bir onarımın erişebileceğinden daha geniş olduğunda ne yapılacağını anlatıyor: replay ile toparlanmak, bir felaketten sağ çıkmak ve farklı bir broker'a geçmek.

Bölüm 14 — Recovery, Felaketler ve Broker Migration

Parcelly'de bir Salı gününü düşünün. Marketplace'teki her event'i taşıyan managed broker cluster'ına ulaşamıyor. Sağlayıcının durum sayfası, cluster'ı barındıran zone'un çöktüğünü ve bir saat içinde yedek bir cluster'ın hazır olacağını söylüyor. O cluster geldiğinde topic'ler ya var ama boş ya da son birkaç dakikanın içinde bir yere kadar veri tutuyor. O noktanın tam olarak nerede olduğunu kimse söyleyemiyor.

Servisler sağlam. Orders, Payments, Inventory, Shipping ve Ledger'ın her birinin kendi veritabanı var ve her veritabanı sapasağlam. Her servisin publish'i denemeyi sürdüren bir relay'i var ve outbox'lar doluyor. Consumer'lar olmayan bir broker'ı bekleyerek boşta duruyor. Bu arada ekibin, broker geri geldiğinde ne yapacağına karar vermesi gerekiyor ve seçeneklerin hepsi makul görünüyor. Consumer'ları yeni cluster'a yönlendirip çalışmaya bırakmak. Her şeyi outbox'lardan yeniden yayımlamak. Bir şeyin son backup'ını geri yüklemek. Her seçenek bir store için doğru, bir diğeri için yanlış.

Bu bölüm, bir store'un bozulmasından, kaybolmasından ya da bilerek değiştirilmesinden sonraki günle ilgili. Tek bir soru soruyor. Broker, veritabanı ya da region eskisi gibi olmadığında, kurduğunuz garantilerden hangileri hâlâ ayakta ve onlardan güvenebileceğiniz bir duruma dönmek için nasıl yararlanırsınız?

Problem ve neden zor olduğu

Önceki bölümler, store'ları sağlam bırakan hatalara karşı savunma kurdu: iki adım arasındaki bir çökme, yavaş bir consumer, bir duplicate, bir poison message. Recovery farklıdır, çünkü ters giden şey hafızanın kendisidir. Bir store zamanda geriye gitti, geçmişinin bir kısmını kaybetti

ya da başka bir store'la değiştiriliyor. Kitaptaki her savunma, bir store'un bir şeyi hatırlamasına yaslanır; dolayısıyla soru, hangi hafızanın ayakta kaldığı ve hangisinin kalmadığı olur.

Bu iki sebeple zordur. Birincisi, store'lar birbirinden bağımsız ve farklı yönlerde bozulur. Altı saat öncesine geri yüklenmiş bir veritabanı ile son iki dakikasını kaybetmiş bir broker birbirleriyle, ikisinin de fark edemeyeceği bir biçimde çelişir. İkisi de kendi içinde tutarlıdır. Bölüm 3 buna bir store'un tamamını kaybetmek dedi ve başlangıçta hiçbir şeyin hata vermediğini not etti.

İkincisi, recovery işi nadirdir ve nadir iş baskı altında kötü yapılır. Bir consumer group'u geriye taşıyan komutlar, bir replay'ın ikinci bir e-posta turu göndermesini engelleyen ayarlar, servislerin yeniden başlatılma sırası: bunların hepsi önceden vermesi ucuz, anında doğaçlama yapması pahalı kararlardır.

Bu bölümün tezi, kitabın en yalın hâldeki tezidir. Hayatta kalan şey broker değildir. Kaybedilmiş bir cluster'dan, geri yüklenmiş bir veritabanından ya da ürün değişikliğinden sonra ayakta kalan, onun etrafında tasarladığınız failure semantics'tir: hiç onaylanmamış olanı hatırlayan transactional outbox (Bölüm 4), zaten yapılmış olanı tanıyan inbox, kararlı tanımlayıcı, contract ve gerçeğin iki görüşünü karşılaştırma alışkanlığı. Bunlar yerindeyse broker değiştirilebilir. Yerinde değilse, sistemin doğruluğunu broker taşıyordu ve bunu kimse bilmiyordu.

Replay ile recovery

Replay, Bölüm 2'de tanımlandığı gibi, geçmişin bir event log'dan bilerek yeniden okunmasıdır. Yalnızca log ailesi için geçerlidir, çünkü bir queue teslim ettiğini siler. Bölüm 15 replay'i bir test tekniği olarak kullanıyor. Burada bir recovery aracıdır ve bir prosedürü vardır.

Uyduğu durum, Bölüm 13'ü açan durumdur: bir consumer group geri kaldı, retention okumadığı mesajları sildi ve group bir boşluğun ötesine geçti. Veri hâlâ retention içindeyse, düşünülecek ilk recovery, event'leri producer'dan tek tek yeniden üretmek değildir. Group'un konumunu boşluktan önceye geri taşımak ve yeniden okumaktır.

Bir group'u belirli bir zamana reset etmek

Bir consumer group'un konumu, partition başına bir offset olarak saklanır. Onu geriye taşımak broker üzerinde yapılan bir yönetim işlemidir ve log tabanlı broker'ların çoğu ya açık bir offset'i ya da bir zaman damgasını hedef alabilir; zaman damgasını broker, o andan sonraki ilk offset'e çözümler. Ben zaman damgasını tercih ederim. Offset, tek bir cluster'daki tek bir topic'in tek bir partition'ı için anlam taşıyan bir sayıdır; zaman damgası ise bildiğiniz şeydir: deploy 09:40'ta başladı, o hâlde 09:30'a geri dön. Broker'ınızın zaman damgasını nasıl çözümlediğini dokümantasyonundan kontrol edin, çünkü kullandığı zaman her zaman producer'ın zamanı ya da broker'ın ekleme zamanı değildir.

Prosedür kısadır ve sıra önemlidir.

1. Group'u durdurun. Çalışmakta olan bir group'un offset'lerini reset etmek, broker'a bağlı olarak ya başarısız olur ya da bir sonraki commit tarafından ezilir.
2. Herhangi bir şeyi değiştirmeden önce mevcut offset'leri kaydedin. Reset yanlışsa geri dönmenin tek yolu budur.
3. Hedefi bir marjla seçin. İhtiyacınız olduğunu düşündüğünüzden daha eskiye gidin, çünkü duplicate'ler ucuzdur, boşluk ise değildir.
4. Hedefe reset edin, group'u yeniden başlatın ve boşalırken en eski işlenmemiş mesajın yaşını (Bölüm 13) izleyin.
5. Group başa dönene kadar izlemeyi sürdürün, sonra reconcile edin; çünkü reset yalnızca log'un hâlâ tuttuğu şeyi onarır.

Marj seçmek yalnızca inbox sayesinde güvenlidir (Bölüm 7); inbox, group'un zaten işlediği mesajların yeniden okunmasını no-op'a çevirir. Consumer'ları idempotent olmayan bir group, prosedür ne derse desin, reset edilmemelidir.

Replay'i tek bir consumer group ile sınırlamak

Bir log, 10 group'un aynı topic'i okumasına izin verir ve her birinin kendi offset'leri vardır. Reset'i hedeflemeyi güvenli kılan özellik budur. Event'leri kaçıran group'u reset edin, başka hiçbir group fark etmez. Bir şey kaçırmış en küçük group kümesini reset edin ve çalıştırmadan önce onları plan içinde adlarıyla yazın. Payments'ın `OrderPlaced` consumer'ı konumunu korurken Inventory'nin group'u son günü yeniden okur.

Aynı disiplinin ikinci bir biçimi, replay için ayrı bir group'tur. Bir projection yeniden kurulacaksa, örneğin Orders'ın arama view'ı, başka bir adla yeni bir group başlatın; bu group baştan okuyup taze bir tabloya yazsın ve yetiştğinde okuyucuyu ona geçirin. Eski group hiç kımıldamaz, dolayısıyla yeniden kurulum ters giderse geri alınacak bir şey yoktur. Bir süre için depolama ve ikinci bir consumer maliyeti vardır; bedava bir rollback için küçük bir bedel.

Yan etkileri bastırmak

Replay, bugünün kodunu dünün olgularına uygular. Bir handler e-posta gönderiyor, ödeme sağlayıcısını çağırıyor ya da bir kargo firmasını bilgilendiriyorsa, replay bunu yeniden yapmasına yol açar. Bölüm 12 bunu gündeme getirdi ve verdiği kural başlangıç noktasıdır: replay ile yeniden kurulan bir consumer, replay yaptığını bilmelidir.

İnsanların akla gelen ilk fikri mesajda bir flag, "bu bir replay" diyen bir header'dır. Bu işe yaramaz, çünkü yeniden oynatılan mesaj, orijinal mesajdır. Broker, producer'ın yazdığı byte'ları tutar ve hiçbir şey onların ikinci kez okunduğunu işaretlemez. Mod, mesaja değil consumer'a aittir. Replay yapan group ayrı bir deployment olarak, `mode: replay` gibi bir yapılandırma değeriyle çalışır ya da etki katmanı group adını kontrol eder.

Mod bilindiğinde, etkilerin kendisi için dört seçenek vardır ve her birinin bir bedeli vardır.

- **Etkileri kenarda kapıyla denetleyin.** Her dış çağrı, modu kontrol eden ve replay'de hiçbir şey yapmayan tek bir fonksiyondan geçer. En açık tasarımdır ve denetlemesi en ucuz olanıdır, ama yalnızca hiçbir handler kapıyı atlamıyorsa işe yarar. Bir sağlayıcıyı doğrudan çağırarak yeni bir handler onu boşa çıkarır; bu yüzden kural bir test gerektirir (Bölüm 15).
- **Dış çağrılara idempotency key koyun.** Capture çağrısı `event_id`'den türetilmiş bir key taşıyorsa, sağlayıcı ikinci denemeyi tanır. Bölüm 7 bunu ele aldı. Yalnızca sağlayıcı key'i hatırladığı sürece korur; bu yüzden ikinci bir savunma hattıdır ve geçen ayın replay'i için güvenilecek olan değildir.
- **Türetilmiş event'lerin outbox'ını duraklatın.** Yeniden oynatılan bir handler, aynı transaction içinde kendi outbox'ına yeni satırlar yazar ve

bu satırlar bir sonraki servise gider. Yeni event'ler taze tanımlayıcılar taşıyorsa, downstream consumer'lar bunları yeni olgular olarak ele alır. Ya giden her event'in tanımlayıcısını girdideki `event_id`'den ve handler'ın adından deterministik olarak türetin, böylece ikinci çalıştırma aynı tanımlayıcıları üretir ve downstream inbox'lar bunları emer; ya da ne ürettiğini kontrol edene kadar replay yapan servisin relay'ini bekletin.

- **Farklı bir ortama replay yapın.** Etkisi olmayan bir projection için yeni bir tablo, yukarıdaki gibi en temiz cevaptır. Etkisi olan bir handler için, etki katmanı bir stub'a bağlı ayrı bir deployment, replay'i gerçekten çalıştırmadan önce test etmenin bir yoludur.

Uyarı: E-posta gönderen, kart çeken ya da kargo firmalarını çağıran bir group'u kapı ve key olmadan reset etmek, her tarihsel event'i yeni bir eyleme çevirir. Inbox'ı da reset ettiyseniz inbox yardımcı olmaz; dış etkiler için hiçbir zaman olamazdı, çünkü sağlayıcı sizin veritabanınızla birlikte geri sarılmaz. Bastırma yöntemine offset'leri taşımadan önce karar verin, ilk şikâyetten sonra değil.

Dedup window ve inbox

Inbox bir replay'in ne yaptığını değiştirir ve bu, ne istediğinize bağlı olarak sizin lehinize ya da aleyhinize çalışır.

Consumer'ın veritabanı sağlamsa ve yeniden okuduğunuz aralık onun dedup window'unun (Bölüm 7) içindeyse, inbox daha önce işlediği her `event_id`'yi tanır ve atlar. Bir boşluktan sonra istediğiniz budur: group yalnızca eksik olanı işler, gerisi ucuzdur. Bir replay'in, bir event'i hiç görmemiş consumer'ı, görmüş olanları rahatsız etmeden onarabilmesinin sebebi de budur.

Aralık pencereden eskiyse, inbox satırları gitmiştir. Consumer eski event'leri ikinci kez uygular. Capture başına tek ledger kaydı gibi bir business-key constraint ile korunan bir consumer için ikinci uygulama zararsızca başarısız olur. Böyle bir koruması olmayan için etki iki kez gerçekleşir. Dolayısıyla dedup window, recovery planının bir parçasıdır: replay'den önce aralıktaki en eski event'i, onu görecektir group'lar arasındaki

en kısa pencereyi ve ikisinin örtüşmediği yerde consumer'ı neyin koruduğunu belirtin. Bölüm 7, pencerenin planladığınız her replay'den uzun yaşaması gerektiği kuralını verdi. Bu, o plandır.

Inbox bir engel de olabilir. Bug'lı bir consumer yanlış state yazdı. Bug'ı düzeltip replay yapıyorsunuz ve her mesaj atlanıyor, çünkü her `event_id` inbox'ta. Bu durum birazdan geliyor.

Replay, onarım ve redrive

Üç araç mesajları yeniden hareket ettirir ve neye etki ettikleri bakımından farklıdır.

Araç	Neye etki eder	Bölüm	Ne zaman doğru
Onarım	Orijinal <code>event_id</code> ile producer'ın outbox'ı üzerinden yeniden yayımlanan bir avuç kayıt	13	Birkaç öge eksik ya da yanlış ve producer hangileri olduğunu söyleyebiliyor
Redrive	Dead letter'lar, onları reddeden consumer'a geri taşınır	8	Mesajlar görünür biçimde başarısız oldu ve sebep düzeltildi
Replay	Log'un bir aralığı, tek bir group tarafından yeniden okunur	14	Birçok key etkilendi, veri hâlâ retention içinde ve consumer idempotent

Ayırt edici soru, neyin yanlış olduğunu kimin bildiğidir. Onarımda, producer ya da bir reconciliation job'ı belirli öğeleri adlandırır. Redrive'da, mesajları sistemin kendisi kenara ayırmıştır. Replay'de kimsenin elinde liste yoktur: hasar bir zaman aralığıdır ve bir aralığı kapsamanın en ucuz yolu hepsini yeniden okuyup neyin yapıldığını inbox'ın ayıklamasına izin vermektir. Replay, kesinlikten kapsama doğru takas yapar.

Retention bir recovery sınırıdır

Replay yalnızca log'un veri tuttuğu kadar geriye ulaşır. Uzun zaman önce disk maliyeti için seçilmiş bir ayar olan retention, bu yüzden elinizdeki en ucuz recovery'nin de sınırıdır. Retention'ı aşan bir kesinti ya da retention'dan sonra keşfedilen bir bug, sizi ucuz yoldan producer'lardan yeniden yayımlamaya ve reconcile etmeye taşır.

Retention'ı yalnızca diske göre değil, ekibin bir sorunu fark edip düzeltmesi için gereken süreye göre ayarladım. Bir reconciliation job'ı günlük çalışıyorsa ve bir düzeltmenin yayına çıkması bir gün sürüyorsa, bir günlük retention geriye hiçbir şey bırakmaz. Retention maliyet tarafıyla da etkileşir: daha uzun retention, bir replay'ın karşılaşılabileceği eski şemaların sayısını artırır (Bölüm 12) ve kişisel verinin log'da yaşadığı süreyi uzatır. Sisteminiz için doğru sayıyı bilmiyorum. Ne kadar hızlı fark ettiğinize, onarımın ne kadar sürdüğüne ve log'un neyi tutmasına izin verildiğine bağlı.

Başarısız olmadan yanlış çalışan bir consumer

Bölüm 3'te, hatasız bir consumer bug'ı için bir satır var: handler başarılı oldu ve yanlış sonucu yazdı. Replay'e en çok ihtiyaç duyan recovery budur ve inbox'ın yolunuza çıktığı recovery de budur.

Ledger'ın PaymentCaptured consumer'ının, bir hafta boyunca bir para birimi için tutarları yanlış üsle çevirdiğini düşünün. Her mesaj acknowledge edildi. Hiçbir şey başarısız olmadı ve inbox her event_id'yi tutuyor. Reconciliation (Bölüm 13) sonunda, Ledger'ın o para birimi için toplamalarının Payments ile uyuşmadığını bildiriyor. Düzeltme yayına çıkıyor ve şimdi yanlış satırlar mevcut ve düzeltilmeleri gerekiyor.

Topic'i aynı consumer'a replay etmek hiçbir şey yapmaz, çünkü inbox her event'in tamamlandığını söyler. İki dürüst yol vardır ve seçim Ledger'ın ne olduğuna bağlıdır.

Consumer'ın state'i türetilmişse ve atılabiliyorsa, örneğin bir projection, yeniden kurun. Boş bir inbox ile yeni bir store'a yeni bir group başlatın, etkilenen ilk event'ten replay yapın, reconciliation ile kaynağa karşı doğrulayın ve geçiş yapın. Yanlış state düzeltilmez, atılır.

Consumer'ın state'i yeniden yazılmaması gereken bir kayıtsa, örneğin yalnızca eklenebilen bir ledger, replay yine yardımcı olur ama üzerine yazmak için değil. Etkilenen aralığı düzeltilmiş kodla, bir düzeltme modunda okuyun. Her event için kaydın ne olması gerektiğini hesaplayın, önceki düzeltmeler düşüldükten sonra post edilmiş olanla karşılaştırın ve fark için, tanımlayıcısı orijinal event_id'den ve düzeltmenin kendi adından türetilmiş bir düzeltme kaydı post edin; böylece düzeltmeyi iki kez çalıştırmak bir no-op olur. Bu işi ledger kaydının kendi tanımlayıcısı

üzerindeki unique constraint yapar, dolayısıyla düzeltme için inbox satırı gerekmez ve döngü yinelenen bir key'i "zaten düzeltildi" sayıp sonraki event'e geçer.

Yanlış kayıtlar tarih olarak kalır. Düzeltmeler, nedenini söyleyen yeni olgulardır. Bu, yeniden kurmaktan daha yavaş ve daha açıktır ve bir denetçinin kabul ettiği tek biçimdir.

```
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
```

Ortak nokta şudur: replay etkilenen aralığı ucuza bulur, her event'le ne yapılacağı kararı ise consumer'ın kendi semantics'ine aittir.

Felaket senaryoları

Felaket, kaybın bir mesaj değil bir store olduğu bir hatadır. Olasılıkları düzenlemenin yararlı yolu, hangi store'un zamanda geriye gittiğine göre düzenlemektir. Her senaryoyu Bölüm 3'ün kullandığı kalıpla yazın: ne bozuldu, nasıl görünür ve hangi garanti hâlâ geçerli. Üçünün sonuncusu işe yarayandır, çünkü neye güvendiğinizi adlandırır ve oraya koyacak bir şeyi olmayan bir akış, o senaryoda açıkta demektir. Senaryolardan önce, neyi hedeflediğiniz hakkında bir söz.

Her store için hedefler

İki sayı bir recovery'yi tanımlar. **Recovery point objective (RPO)**, ne kadar veri kaybetmeyi göze alabileceğinizdir ve zaman olarak ölçülür: altı saatte bir backup, altı saate kadar RPO demektir. **Recovery time objective (RTO)**, servissiz ne kadar kalabileceğinizdir. İkisi de birer karardır ve olağan hata, bunlardan birini bütün sistem için vermektir.

Farklıdır, çünkü store'lar farklıdır. Bu bölümdeki store'lar için bir taslak şöyle görünür.

Store	Kaybın maliyeti	Tipik dönüş yolu	Neyi belirler
Bir servis veritabanı	Bir event taşııyorsa başka hiçbir yerde olmayan state	Backup artı log shipping	Veritabanı başına RPO, verinin iş değeriyle belirlenir
Outbox	Henüz yayımlanmamış event'ler	Aynı veritabanı, yani aynı RPO	Kaybolan her satır, hiç var olmamış bir event'tir
Inbox	Neyin işlendiğinin hafızası	Etkilerle aynı veritabanı	Onlarla birlikte geri gider (Bölüm 3)
Broker	Uçuştaki event'ler ve tutulan geçmiş	Replication ya da producer'lardan yeniden yayımlama	RPO'yu replication, RTO'yu consumer'ların ne kadar hızlı boşalttığı belirler
Türetilmiş bir projection	Yeniden kurulabiliyorsa hiçbir şey	Replay	RTO replay süresidir, log her şeyi tutuyorsa RPO sıfırdır
Bir dış sağlayıcı	Kontrolünüz dışındaki etkiler	Onların kayıtları ve reconciliation	İki hedefinizin de dışında

Tablonun vurguladığı nokta, replay edilen projection için RTO'nun bir boşaltma hesabı olduğudur. Log bir ay tutuyorsa ve group saatte bir günlük veri işleyebiliyorsa, yeniden kurulum yaklaşık 30 saat sürer ve RTO budur (bu rakamları varsayın; sizinkiler Bölüm 13'teki gibi ölçümden gelir). Boşaltma hızını hiç ölçmemiş bir ekip RTO'sunu bilmez.

Hedefleri ayrıca Bölüm 3'teki failure model'e, akış akış bağlardım. Payment capture akışının toleransı, notification akışınınkinden farklıdır ve Ledger'ın bir boşluğa toleransı sıfıra daha yakındır. Tek tip bir hedef, notification'lar için fazla sıkı, ledger için fazla gevşek çıkar.

Bir veritabanını backup'tan geri yüklemek

Orders'ın veritabanının altı saat öncesine geri yüklendiğini varsayın. Bu, kitabın mekanizmasına ne yapar?

Outbox, state ile birlikte geri gider. O altı saatte yazılan satırlar, duyurdukları siparişlerle birlikte gitmiştir; Payments ve Inventory ise bu event'ler üzerinde zaten harekete geçmiş olabilir. Geri yükleme noktasında yayımlanmamış olup eski zaman çizelgesinde sonradan yayımlanmış satırlar yeniden yayımlanmamış olur, dolayısıyla relay bunları ikinci kez yayımlar. `event_id` aynı olduğu için bu zararsız bir duplicate'tir.

Daha keskin bir tehlike var. Sipariş tanımlayıcıları bir sequence'ten geliyorsa, geri yüklenen sequence geri sarılmıştır ve sıradaki sipariş, kaybolan zaman çizelgesinde başka bir siparişe verilmiş bir tanımlayıcı alacaktır. Payments şimdi, Orders'ın unuttuğu bir alıcıya ait `order-81724` için bir capture tutuyor ve yeni `order-81724` başka birine ait. Bu sebeple aggregate'ler için opak rastgele tanımlayıcıları tercih ederdim. Bir sequence tutmak zorundaysanız, trafiği kabul etmeden önce onu şimdiye kadar verilmiş en yüksek değerin ötesine ilerletin; bu sayıyı downstream servislerden okuyabilirsiniz.

Veritabanı geri yüklenen bir consumer'ın **inbox**'ı, etkilerle birlikte geri gider. Bölüm 3 bunun doğru olduğuna dikkat çekti: geri yükleme noktasından önce işlenen mesajlar hâlâ inbox'ta ve etkileri hayatta kaldı; ondan sonrakiler ise iki yerde de unutuldu. Bu özellik yalnızca inbox ve etkiler aynı veritabanında yaşıyorsa geçerlidir.

Geri gitmeyen şey, broker'da yaşayan consumer'ın **offset**'idir. Group, geri yüklenen veritabanının unuttuğu altı saatlik event'in ötesinde, başta konumlanmış durumda. Biri group'u geri taşımadıkça consumer onları bir daha asla görmez; bu yüzden geri yükleme planı, o group'un bir reset'ini içerir.

Hedef özen ister, çünkü geri yükleme noktası veritabanının saatindeki bir zamandır, zaman damgası reset'i ise log'daki event zamanları üzerinde çalışır. Geri yükleme anında group başta değildi. Bir konumdaydı ve o konumdaki event, group'un mesaj yaşı kadar (Bölüm 13) öncesinde üretilmişti. Dolayısıyla hedef, group'un o andaki konumundaki event

zamanı eksi bir marjdır. Hedef yalnızca geri yükleme noktasından belirlenirse, bir saat geride olan bir group, veritabanı geri giderken henüz işlemediği o saati atlar.

Konumu mesaj yaşı geçmişinden ya da kaydedilmiş offset'lerden okuyun ve bu geçmiş, geri yüklemek üzere olduğunuz veritabanının dışında tutun. Bazı ekipler offset'i etkilerle aynı veritabanında, aynı transaction içinde saklar; böylece ikisi birlikte geri sarılır. Bu, elle yapılan adımı ortadan kaldırır ve offset muhasebesini broker'dan devralır; tartılması gereken bir maliyettir.

Dış etkiler olduğu yerde kalır. Payments'ın kaybolan zaman çizelgesinde sağlayıcıya gönderdiği capture, sağlayıcının kayıtlarında vardır, Payments'ın veritabanında yoktur. Reconciliation'ın (Bölüm 13) gerekli olduğu yer burasıdır. Kayıtlarınızı sağlayıcının kilerle karşılaştırın ve farkları onarın.

Sistemin yeniden başlatılma sırası önemlidir ve bir prosedür olarak yazılmaya değer.

```
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")
```

Adımların kendisi önemlidir. Önce iki tarafı da duraklatmak, geri yüklenen servisin tutarsız state'ini daha da kötüleştirmesini engeller ve devam etmeden önce reconcile etmek Bölüm 3'ü izler. Reset, geri yüklemeyi bir eklemeyle birlikte bir replay problemine çevirir.

Kaybolan saatleri yeniden okuyan geri yüklenmiş bir Orders, bir `CapturePayment` komutu dahil yeni outbox satırları üretir; taze bir `command_id` Payments'ın inbox'ına yeni görünür ve iki kez capture yapabilir. Bu yüzden türetilmiş event'lerin id'leri, yan etki bölümündeki gibi girdideki `event_id`'den gelir. `CapturePayment` gibi iki event'in tetiklediği bir komut ise id'sini bunun yerine business key'den alır, örneğin `order_id` artı komut tipi (Bölüm 7); çünkü ikisinden hangisinin en son okunduğu replay'de değişebilir. Relay, biri ne üretildiğini kontrol edene kadar bekletilir.

Broker verisini kaybetmek

Bir broker cluster'ı kaybetmeden veri kaybedebilir. Log ailesinde, geride kalmış bir replica leader'a terfi ettirilebilir ve sahip olmadığı mesajlar gider. Acknowledgment seviyesi ve in-sync replica'ların minimum sayısı, producer'a böyle bir mesajın güvende olduğunun söylenip söylenmediğini belirler (Bölüm 2). Bazı broker'lar güncelliğini yitirmiş bir replica'nın terfisine yapılandırmayla izin verir ya da yasaklar; sizinkinin varsayılanını bulup yazıya dökerdim. Retention de aynısını sessizce yapabilir, Bölüm 13'ün gösterdiği gibi.

Gördükleriniz: sıçrayan ya da aralık dışı olarak raporlanan consumer konumları, end offset'i eskisinden düşük olan bir topic ve olaydan sonra "eksik" türünde reconciliation bulguları. Broker'ın acknowledgment'ı sınırı olan bir kanıttır ve sınır, o anda var olan kopya sayısıdır.

Hayatta kalan: outbox, satırları tuttuysanız. Yayımladıktan hemen sonra satırları temizleyen bir relay, ne gönderdiğine dair hafızaya sahip değildir. Yayımlanmış satırları bir süre, örnek olarak birkaç gün tutan bir relay ise etkilenen penceredeki satırları yayımlanmamış olarak işaretleyip yeniden gönderebilir. Yeniden yayımlama orijinal `event_id`'yi korur, dolayısıyla bir event'i gerçekten almış consumer'lar ikinci kopyayı duplicate olarak ele alır. Bu, Bölüm 13'ün onarım adımının bir aralığa uygulanmış hâlidir ve aynı fan-out uyarısını taşır: bir subscriber'ın dedup window'undan eski event'ler, herkese değil, onları kaçıran tek consumer'a teslim edilmelidir.

Outbox satırları gitmişse, sıradaki kaynak producer'ın kendi kayıtlarıdır: envelope'u iş kaydından, saklanmış `event_id` ve payload hash'ile yeniden kurun ve hash'in farklı olduğu yerde bir insan için durun. Ve ikisi de yoksa, event kurtarılamaz ve reconciliation onu raporlamayı sürdürür. Bu noktada doğru soru bir iş sorusudur: eksik olgu başka bir sistemden yeniden mi kurulacak, yoksa kayıp olarak kabul edilip açıklanacak mı? Bu sorunun nadiren sorulacağı biçimde tasarlamak, outbox satırlarını bir süre tutmanın satın aldığı şeydir.

Bir region'ı kaybetmek

Bir region kaybı önceki ikisini birleştirir ve kendine ait bir problem ekler. Parcellly'nin ikinci bir region'da, topic'leri sürekli kopyalayan bir replication aracıyla beslenen bir standby cluster işlettiğini ve veritabanlarının asenkron olarak replicate edildiğini varsayın.

Belirtilecek ilk özellik, **asenكرون replication'ın bir RPO'su olduğudur** ve bu, hata anındaki replication lag'dir. Primary region'da yazılıp henüz kopyalanmamış her şey standby'ın bakış açısından gitmiştir.

Kaybettiğinizin ölçüsü budur ve dakika dakika değişen bir sayıdır. Bölüm 13'ün mesaj yaşını ele alışı gibi, belirtilmiş bir limite karşı alert kurmaya da değer: RPO'nuzu izliyorsunuz.

İkinci özellik, **offset'lerin yolculuk etmemesidir**. Standby mesajların bir kopyasını tutar, ama kopyanın kendi offset'leri vardır; bunlar replica yazdığında atanır. Bir consumer'ın primary'deki commit edilmiş konumu standby hakkında hiçbir şey söylemez. Bazı replication araçları offset çevirisi sunar; sizinkinin neyi garanti ettiğini kontrol ederdim ve bunun tam doğru olmasına bağlı bir recovery tasarlamazdım.

Taşınabilir plan, standby üzerinde her group'u, primary çöktüğünde ulaştığı konumdaki event zamanından eksi bir marjla başlatmak ve örtüşmeyi inbox'ın emmesine izin vermektir. Bu, veritabanı geri yüklemesindeki kuraldır. 40 dakika geride olan bir group, 40 dakika geriden başlar. Hiç replicate edilmemiş event'ler standby'da hiç yoktur ve aşağıdaki outbox penceresi onları kapsar.

Primary'nin kendi commit bilgisi region ile birlikte kaybolduğundan, mesaj yaşı geçmişinin onun dışında tutulması gerekir. Gitmişse, consumer'ların dedup window'ları içindeki en eski noktaya geri dönün ve fazladan

duplicate'leri kabul edin. Bir alert eşiği yeterli değildir, çünkü alert o sırada ateşleniyor olabilir.

Üçüncü özellik, **failover'daki duplicate'lerin olası değil kesin olmasıdır**. Gerçek konumundan daha erken yeniden başlayan bir consumer event'leri iki kez işler. Region çöktüğünde yayımlamanın ortasında olan bir relay, hayatta kalan veritabanından ikinci kez yayımlayabilir.

Boşluklar madalyonun öbür yüzüdür: primary'nin acknowledge ettiği ve standby'in hiç almadığı event'ler. Producer'ın veritabanı hayattaysa ve standby'in broker'ından daha yeniyse, standby'in hiç görmediği, yayımlanmış olarak işaretli satırlar tutar. Bunlar için outbox'ın yeniden yayımlama penceresini broker kaybı durumundaki gibi genişletin: son replication lag aralığındaki satırları, artı bir marjı, yayımlanmamış olarak işaretleyin ve yeniden gönderin. Bu pencereyi replication lag boyutlandırır ve yalnızca bu pencereyi. Duplicate'ler consumer'da hiçbir şeye mal olmaz. Boşluklar, korumaya çalıştığınız şeye mal olur.

Producer'ın veritabanının kendisi region ile birlikte kaybolduysa ve daha eski bir replica'dan geri yüklendiyse, durum backup'tan geri yüklemeye indirgenir; beraberinde geri sarılmış sequence'ler ve unutulmuş siparişlerle.



Diyagramı soldan okuyun. Bilinmeyenle ilgilenen iki adım, yeniden yayımlama penceresi ve zaman damgası marjı, ikisi de duplicate'e doğru hata yapar. Sonraki her adım, duplicate'i zararsız kılmak için vardır.

Broker migration

Migration, planladığınız bir felakettir. Bir ekip maliyet, destek sonu, managed bir teklif ya da erişim kalıbındaki bir değişiklik yüzünden bir broker'dan diğerine geçer ve bu geçiş yukarıdaki felaketlerle aynı bileşenlere sahiptir: bir tarafta veri, öbür tarafta yeni boş bir yer, taşınmayan konumlar ve iki sistemin aynı anda canlı olduğu bir dönem. Fark, dikkatli olmak için vaktiniz olması ve sizi doğaçlamaya zorlayan bir acil durumun olmamasıdır.

Kitabın araçları migration'ı kontrollü bir değişiklik yapar. Sebep, bunların birkaçının bilerek uygulama ile broker arasına yerleştirilmiş olmasıdır.

Tek geçiş noktası olarak relay

Bir transactional outbox ile (Bölüm 4), üreten bir servis broker ile hiç konuşmaz. Kendi veritabanına yazar ve relay (Bölüm 5) satırları broker'a taşır. Yani broker seçimi tek bir süreçte kodlanmıştır. Event'lerin nereye gittiğini değiştirmek, relay'in nereye yayımladığını değiştirmek demektir ve producer'ın kodu değişmez. Bu, her servise yayılmış bir değişiklikten küçük bir değişikliktir; Bölüm 4'ün başka bir sebeple önerdiği bir örüntünün yan faydası.

Relay envelope'a da sahiptir. `event_id`'yi outbox satırından alır ve yayımladığı mesaja koyar. Yeni broker'da aynı event aynı `event_id`'ye sahiptir ve migration'ın geri kalanını güvenli kılan, bu eşitliktir.

Dual-publish

En güvenli ilk adım, relay'in her satırı iki broker'a da yayımlamasıdır. Bir süre boyunca her event, bugünün consumer'larının onu okuduğu eski broker'da ve henüz kimsenin okumadığı yenisinde var olur.

Dual-publish'in kendisi bir dual write'tır ve Bölüm 4 bunların neden tehlikeli olduğunu açıklıyor. Burada risk daha küçüktür, çünkü doğruluk kaynağı outbox satırındır ve bir çökme, satırın yeniden gönderilmesini bırakır.

Tasarım sorusu, relay'in neyi beklediğidir. Bir satırı yalnızca iki publish de başarılı olduktan sonra yayımlanmış olarak işaretlerse, ulaşamayan bir gölge broker, satırları yayımlanmamış bırakır ve kimsenin okumadığı bir broker uğruna consumer'ların bağlı olduğu canlı akışı durdurur. Bu yüzden iki hedefi ayrı ayrı izleyin. Satır, Bölüm 5 anlamında, eski broker ona sahip olduğunda yayımlanmıştır ve ikinci bir işaretleyici yeni broker'ı kaydeder. `id` sırasıyla tek bir geçiş, ikinci işaretleyicisi olmayan satırları yeni broker'a besler; bu, key başına sırayı korur ve temizleme ikisini de bekler. Maliyeti bir kolon ve bir geçiş daha; karşılığında yeni broker'ın kesintisi, Parcellly'nin kesintisi olarak değil, karşılaştırmada büyüyen bir boşluk olarak görünür.

Bu aşamada eski broker, consumer'ların güvendiği broker olarak kalır. Yeni broker bir gölgedir ve tek işi, orijinal ile karşılaştırılmaktır.

Mirroring ve bridge'ler

Relay'de dual-publish'e bir alternatif, mesajları broker'lar arasında bir araçla kopyalamaktır: log'dan log'a için bir mirroring aracı, log'dan queue'ya ve queue'dan log'a için bir bridge ya da connector. Bu, relay değişikliği gerektirmez ve dual-publish'in taşımadığı, migration başlamadan önce var olan geçmişi de taşır.

İki uyarı. Bir mirror ya da bridge hem bir consumer hem bir producer'dır; dolayısıyla at-least-once'tır ve her consumer-producer çiftinin yaptığı gibi duplicate üretir. Ve `event_id`'yi, mesaj gövdesinde ya da bir header'da, değiştirmeden geçirmelidir. Kendi tanımlayıcısını üreten bir bridge, kopyalanan her event'i yeni bir event'e çevirir ve inbox hiçbirini tanıyamaz. Buna güvenmeden önce bir örnek üzerinde test edin. Bir mirror'ın partitioning'i, key'leri ve header'ları koruyup korumadığı ve partition'lar arasında hangi sırayla koruduğu araca göre değişir ve cevap, bu kitapta değil, aracın dokümantasyonundadır.

Offset'ler çevrilmez

Bir consumer group'un eski broker'daki offset'i yenisinde anlamsızdır. Aynı türden iki broker arasında bile kopyanın kendi numaralandırması vardır. Bir queue ile bir log arasında ise bir tarafta hiç offset yoktur.

Cevap, region kaybı bölümünün verdiği cevaptır: güvenli bir başlangıç noktası seçin ve idempotency'ye yaslanın. Her group için, group'un eski broker'da henüz işlemeyi bitirmediği her event'in ondan sonra yayımlanmış olacağı bir zaman damgası seçin. Group'un geçiş anındaki lag'ini alın, zamana çevirin ve bir marj ekleyin. Yeni group'u yeni broker'da o zaman damgasından başlatın. Eski group'un zaten işlediği event'leri yeniden okur ve inbox onları atlar.

Dedup window burada bir replay'e uyguladığı gibi uygulanır: yeni group'un yeniden okuyacağı en eski event, her consumer'ın penceresinin içinde olmalıdır.

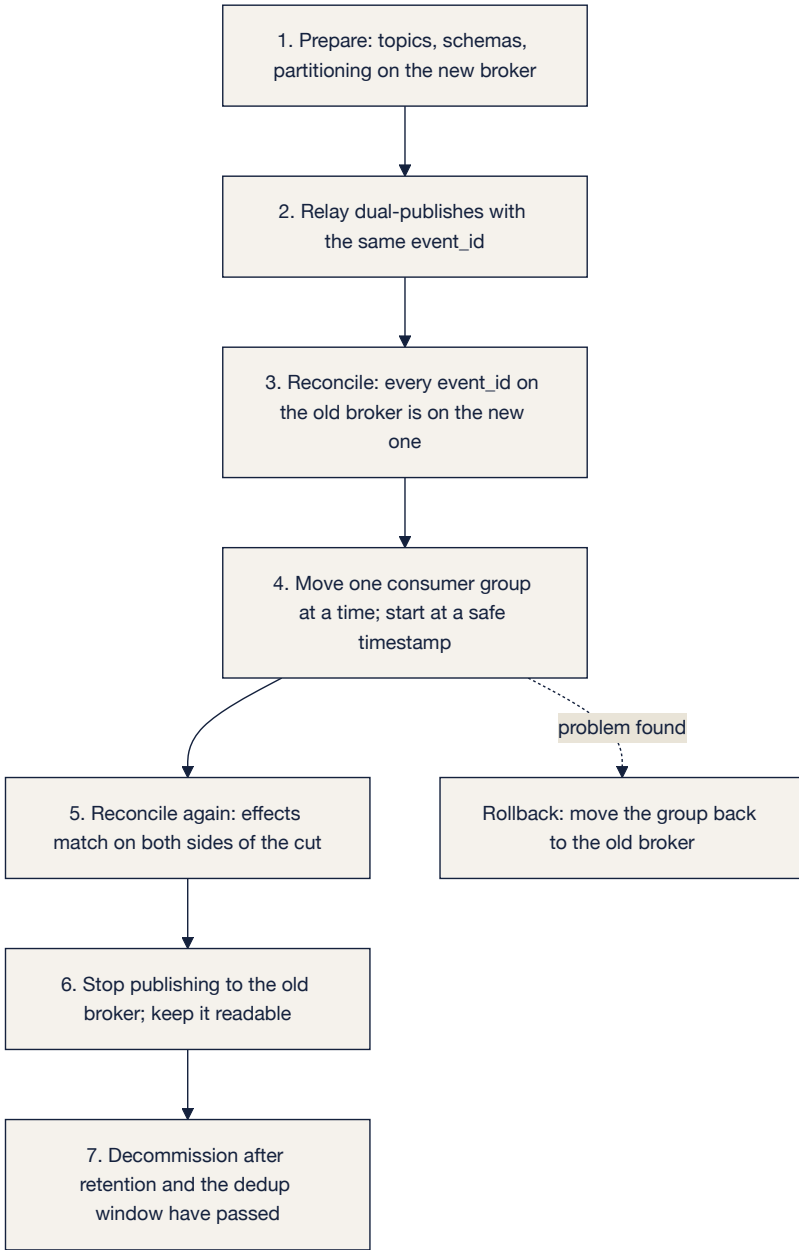
Ordering ve partition'lar

Bölüm 9, key başına sıra için koşulu verdi: aynı key, aynı partitioner, aynı partition sayısı. Bir migration üçünden en az birini değiştirir. Yeni broker farklı bir partitioner kullanabilir, böylece bir `order_id` hiç gitmediği bir partition'a düşer ve yeni topic farklı sayıda partition ile oluşturulabilir, çünkü insanların sayıyı yeniden düşündüğü an budur.

Bundan iki şey çıkar. Partition'ı, Bölüm 9'un önerdiği gibi, relay'de saklanmış `partition_key`'den hesaplayın; böylece eşlemeye tek bir kod parçası sahip olur ve bir key'in event'lerinin yeni broker'da tek bir partition'a düştüğünü test edin. Ve cutover'ı, hiçbir key'in iki yerde birden uçuşta event'i olmayacak biçimde planlayın. Bir consumer yeni topic'i okumaya başlarken eski topic aynı `order_id` için hâlâ işlenmemiş event'ler tutuyorsa, yeni event'ler eskilerden önce işlenebilir: yine `OrderConfirmed`'dan önce `OrderCancelled`, Bölüm 9'daki hata. Bundan kaçınmanın iki yolu var. Yeni group'u başlatmadan önce eski backlog'u sıfıra kadar boşaltın; bu, boşaltmanın süresi kadar işlemeyi duraklatır. Ya da Bölüm 9'un anlattığı gibi, state taşıyan event'ler ve daha eskisini reddeden bir aggregate version ile consumer'ları tasarım gereği toleranslı yapın. Sıranın önemli olduğu ve duraklamanın karşılanabilir olduğu her akış için önce boşaltırdım.

Cutover

İşlemlerin sırası, insanların sorduğu kısımdır: önce consumer'lar mı, önce producer'lar mı? Dual-publish ile soru çoğunlukla ortadan kalkar, çünkü tek bir consumer bile geçmeden önce producer'lar zaten ikisine de yazıyordur. Sıra şu olur.



Reconciliation geçişi çerçeveler. Öncesinde, bir pencere boyunca iki broker'daki event_id değerlerinin kümesini karşılaştırın: eski broker'da olup yenisinde eksik olan her şey, kimse yeni broker'a bağlı değilken bulunmuş bir dual-publish kusurudur. Sonrasında etkileri karşılaştırın: 14:00'te taşınan bir group, eski broker'da üreteceği state'in aynısını

üretmiş olmalıdır ve bunu doğrulayan, Bölüm 13'ün servisler arası reconciliation'ıdır. Consumer'lar sağlıklı olduğu için bitti ilan edilen bir migration, yalnızca hiçbir şeyin çökmediğini doğrulamıştır.

Consumer group'ları teker teker taşıyın; en az kritik olanla, diyelim Notifications, başlayıp Ledger ile bitirin. Her group küçük, ayrı ayrı geri alınabilir bir adımdır.

Rollback

Rollback planı olmayan bir migration tek yönlü bir kapıdır. Bu tasarımla plan neredeyse bedavadır, tek bir koşul sürdüğü süreç: eski broker her event'i almaya devam eder. Relay dual-publish yaparken eski broker eksiksizdir; dolayısıyla rollback, group'u eski broker'a bir zaman damgasında bir marjla geri yöneltmek demektir. Inbox, group'un yeni yerde işlediği event'leri emer.

Altıncı adımın geç gelmesinin sebebi bu koşuldur. Relay eski broker'a yazmayı bıraktığında, rollback akışın tersine çevrilmesini gerektirir: relay yeniden eskisine yazar ve eski broker'ın consumer'ları, durduğu noktadan başlar. Outbox satırları tuttuğu için bu hâlâ mümkündür ve daha yavaştır. Rollback ayrıca eski broker'daki retention ile ve dedup window ile sınırlıdır; bu yüzden onu yalnızca aldığı son event'ten beri ikisi de geçtiğinde devreden çıkarırdım. Eski broker'ı son consumer'ın taşındığı gün emekliye ayırmak, yeni broker'ın sorunlarının ortaya çıkma olasılığının en yüksek olduğu anda rollback'i ortadan kaldırır.

Queue ile log arasında

Migration aileler arasında geçtiğinde semantics değişir ve onu adres değişikliği sayan bir plan yanlıştır.

Veçhe	Queue'dan log'a	Log'dan queue'ya
Geçmiş	Kazanılır: replay mümkün olur (ve retention bir karar olur)	Kaybolur: replay yok, dolayısıyla recovery outbox'a ve producer'lara yaslanır
Acknowledgment	Mesaj başına ack bir offset commit'ine dönüşür; tek bir yavaş mesaj artık partition'ını bloke edebilir (Bölüm 9)	Bir offset commit'i mesaj başına ack'e dönüşür; visibility timeout'ları ve redelivery ile

Veçhe	Queue'dan log'a	Log'dan queue'ya
Ordering	Queue'nun sunduğu yerde group sırası, partition başına sıraya dönüşür; key'i seçin	Partition başına sıra yeniden kurulmalıdır, örneğin queue'nun desteklediği yerde message group'larla
Dead letter'lar	Redrive, sizin kurduğunuz ya da yapılandırığınız bir şeydir; log'un yerleşik bir dead-letter queue'su (DLQ) yoktur, yalnızca sizin oluşturduğunuz bir dead-letter topic'i vardır (Bölüm 8)	Queue'nun DLQ özellikleri ortaya çıkar ve limitleri ile retention'ı farklıdır
Gecikme ve süre aşımı	Geciktirilmiş teslimat ve mesaj başına expiry bulunmayabilir ve bir ikame gerektirir	Gecikme ve expiry yerleşik olabilir; retention'a dayalı recovery ortadan kalkar

Her satır, consumer'larınızın güvendiği bir davranışın sessizce değişebileceği bir yerdir. Hedefi seçmeden önce her consumer'ın bağlı olduğu davranışların listesini yazardım ve önemli olanları, örneğin partition'ını bloke etmemesi gereken bir poison message'ı, bir provada test ederdim (Bölüm 15).

Recovery'yi prova etmek

İlk çalıştırışınız baskı altındaysa, bunların hiçbirisi pek değerli değildir. Hiç yapılmamış bir restore bir inançtır. Gerçekçi hacme karşı hiç çalıştırılmamış bir replay prosedürünün bir süresi yoktur.

İki prova maliyetini çıkarır. Bir **restore test**, bir backup alır, onu bir scratch ortamına geri yükler ve geri yüklemenin outbox'a, inbox'a ve offset'lere ne yaptığını kontrol eder, sonra resume prosedürünü çalıştırır ve süresini ölçer. Bir **game day**, güvenli bir yerde bir şeyi, bir broker'ı, bir zone'u, bir consumer veritabanını bilerek kaldırır ve ekibin yalnızca runbook'u kullanarak toparlanmasını sağlar. Bölüm 15, bu tatbikatların nasıl tasarlanacağını ele alıyor. Gerekçe buraya aittir: hangi adımları unuttuğunuzu yalnızca recovery'yi çalıştırmak gösterir.

Bu bölümün senaryoları için bir runbook'un kısa bir soru listesini yanıtlaması gerekir; bu liste yazılı olmalı, birinin kafasında taşınmamalı. Bölüm 16 bunları runbook rehberliğine koyuyor.

- Hangi store etkilendi ve hangisi zamanda geriye gitti?
- Recovery point nedir ve nasıl belirlendi?
- Hangi consumer group'lar, hangi zaman damgasına ve hangi marjla reset edilmeli?
- Replay sırasında hangi etkiler, hangi mekanizmayla bastırılmalı?
- Yeniden okunacak her aralık, etkilenen her consumer'ın dedup window'unun içinde mi?
- Ne, hangi sırayla duraklatıldı ve kim devam ettirecek?
- Recovery'nin işe yaradığını hangi reconciliation'lar doğrular ve onları kim okuyor?
- Her şey ters gidiyorsa hepsi nasıl geri alınır?

Trade-off'lar ve maliyetler

Retention bir recovery bütçesidir. Daha uzun retention, ucuz recovery'yi daha uzun süre erişilebilir tutar; daha önce sıralanan maliyetlerle. Daha kısa retention sizi producer'lardan yeniden yayımlamaya iter.

Outbox satırlarını tutmak depolamaya ve bir temizlik job'ına mal olur.

Yeniden yayımlama penceresini mümkün kılan satırlar, daha yalın bir tasarımın hemen sildiği satırlardır. Change data capture (CDC) ve anında silme ile pencere yoktur (Bölüm 5). Güvenliği istiyorsanız, yayımlanmış satırları belirtilmiş bir süre tutun ve tabloyu izleyin.

Replay kör bir araçtır. Aralıktaki her şeyi, sağlam olanlar dahil, yeniden okur ve her consumer'ın idempotent, her etkinin kapıyla denetlenmiş olmasına yaslanır. Bilinen birkaç kayıt için onarım daha ucuz ve daha güvenlidir. Replay yerini, hasar geniş olduğunda ve kimsenin elinde liste olmadığıda kazanır.

Yan etki bastırma, test edilmesi gereken koddur. Var olan ama yeni bir handler tarafından atlanan bir kapı, hiç olmamasından kötüdür, çünkü güven yaratır. Maliyeti kalıcı bir test ve code review'da bir kuraldır.

Dual-publish ile migration iki kat maliyetlidir. İki broker aynı anda çalışır, relay daha fazla iş yapar ve iki hedefi izler, eski taraftaki duplicate'ler artar. Karşılığında, kimse yeni broker'a bağlanmadan önce doğrulama ve bir rollback elde edersiniz. Sistem bir bakım penceresine

tahammül ediyorsa, doğrudan cutover daha ucuzdur: relay'i durdurun, eski consumer'ları boşaltın, relay'i yeni broker'a yöneltin ve consumer'ları orada başlatın. Bir outbox ile bu, kayıpsız ama kesintili bir geçiş gerektirir, çünkü satırlar bekler. Doğrulamadan ve rollback'ten vazgeçer ve ledger için bunu seçmezdim.

Hepsini ne zaman kurmamalı. Bir boşluğun ucuz olduğu bir akış, örneğin dahili bir metrics feed'i, yeniden yayımlama penceresine, runbook'a ya da game day'e ihtiyaç duymaz. Recovery'yi hatanın maliyetine göre ayarlayın ve hangi akışları kapsam dışı bırakmaya karar verdiğinizizi yazın. Planda olmamaları bir karar olmalıdır.

Sık yapılan hatalar

- **Bir group'u yan etkilerini kontrol etmeden reset etmek.** Cazip gelir, çünkü reset tek bir komuttur ve veri oradadır. Sonuç ikinci bir e-posta turu ve ikinci bir charge denemesidir. Herhangi bir offset'i taşımadan önce etkilerin nasıl bastırılacağına karar verin.
- **Çok fazla group'u reset etmek.** Herkesi baştan başlatmak titiz hissettirir. Reset ettiğiniz her group geçmişi yeniden okur ve kendi etkilerinin riskini taşır. Yalnızca bir şey kaçırmış group'ları reset edin.
- **Bir replay header'ına güvenmek.** Cazip gelir, çünkü mesajdaki bir flag en ucuz sinyal gibi görünür. Yeniden oynatılan mesaj orijinal mesajdır ve yeniden okunduğuna dair hiçbir işaret taşımaz. Mod, consumer deployment'ına ya da etki katmanına aittir.
- **Bug'ın inbox'ı olan bir consumer'a replay yapmak.** Bir consumer bug'ından sonra inbox her mesajı tamamlanmış diye atlar. Türetilmiş state'i taze bir store'a yeniden kurun ya da yalnızca eklenebilen bir state için açık düzeltmeler post edin.
- **Bütün sistem için tek bir recovery hedefi.** Cazip gelir, çünkü üzerinde anlaşılacak tek bir sayıdır. Notification'lar için fazla sıkı, Ledger için fazla gevşektir. Hedefleri akış başına ve store başına belirleyin.
- **Bir veritabanını geri yükleyip hemen devam etmek.** Servis sağlıklı ve dashboard'lar yeşil, dolayısıyla trafik yeniden açılıyor. Geri yüklenen servis o zaman geri sarılmış state'i yayımlar, tanımlayıcıları

yeniden kullanır ve offset'lerinin atladığı event'leri kaçıır. Duraklatın, reconcile edin, offset'leri reset edin, sonra devam edin.

- **Geri yüklemekten sonra sequence tabanlı tanımlayıcıları yeniden kullanmak.** Sequence'ler varsayılandır ve iki sipariş bir tanımlayıcıyı paylaşıp kadar hiçbir şey yanlış görünmez. Geri sarılmış bir sequence, zaten başka bir siparişe ait olan bir tanımlayıcı verir. Opak tanımlayıcılar kullanın ya da sequence'i verilmiş en yüksek değerin ötesine ilerletin.
- **Offset çevirisine güvenmek.** Cazip gelir, çünkü araç bunu duyuruyor ve zaman damgası aritmetikinden kurtarırdı. Mesajları kopyalayan bir replication aracı offset'leri tam olarak eşleyebilir de eşlemeyebilir de. Zaman damgalarını ve inbox'ı kullanarak, onsuz çalışan bir recovery planlayın.
- **Yeni tanımlayıcılar üreten bir bridge.** Bir bridge sıklıkla her mesajı kendi envelope'una sarmak üzere yapılandırılır ve bu, tek consumer'lı bir testi geçer. Kopyalanan event'ler yeni görünür, dolayısıyla inbox hiçbir şeyi tanımaz ve her duplicate bir etkiye dönüşür. `event_id`'yi değiştirmeden geçirin ve test edin.
- **Eski broker'ı son consumer taşınırken emekliye ayırmak.** Migration bitmiş gibi hissettirir. Rollback'i, en çok gerekeceği anda ortadan kaldırır. Retention ve dedup window'un geçmesini bekleyin.
- **Hiç backup geri yüklememek.** Backup job'ı yeşil. O zaman ilk restore, gerçek olandır. Bir scratch ortamına bir takvimle geri yükleyin ve süresini ölçün.

Temel çıkarımlar

- Bir store kaybedildiğinde, geri yüklendiğinde ya da değiştirildiğinde hayatta kalan şey, etrafındaki failure semantics'tir: outbox, inbox, kararlı `event_id`, contract ve reconciliation; broker'ın kendisi ise değiştirilebilirdir.
- Replay, log ailesi için bir recovery aracıdır: tek bir consumer group'u durdurun, offset'lerini kaydedin, group'un gerçekte bulunduğu yerden ölçülmüş bir zaman damgasına, bir marjla reset edin ve veri hâlâ retention içinde olduğu sürece duplicate'leri inbox'ın emmesine izin verin.

- Yeniden oynatılan mesaj orijinaline benzer; bu yüzden replay yapan consumer modunu bilmeli ve etkilerini kapıyla denetlemeli, downstream tanımlayıcıları deterministik olarak türetmeli ya da outbox'ını bekletmeli ve asla bir header'a güvenmemelidir.
- Dedup window her recovery planının parçasıdır, çünkü pencereden eski bir aralık duplicate'leri yeniden içeri alır ve bir consumer bug'ı inbox'ı bir korumadan, yeniden kurulum ya da açık düzeltmeler gerektiren bir engelle çevirir.
- Recovery point ve recovery time hedefleri her store'a ve akışa aittir; geri yüklenen bir veritabanı outbox'ını, inbox'ını ve sequence'lerini geri sarar, offset'ler ise broker'ın bıraktığı yerde kalır; bu yüzden resume prosedürü duraklatır, reconcile eder, offset'leri reset eder ve ancak sonra devam eder.
- Region'lar arası asenkron replication'ın RPO'su lag'ine eşittir ve veriyi taşır ama consumer offset'lerini taşımaz; bu yüzden bir failover her group'u kendi çöküş anındaki konumunun event zamanından başlatır, outbox'ın bir yeniden yayımlama penceresini replication lag'e göre boyutlandırır ve duplicate'leri kesin, boşlukları asıl risk sayar.
- Bir broker migration'ı relay'i tek geçiş noktası olarak kullanır, aynı event_id ile dual-publish yapar, her seferinde tek bir consumer group'u güvenli bir zaman damgasından taşır, öncesinde ve sonrasında reconcile eder ve rollback artık gerekmeğe kadar eski broker'ı okunabilir tutar.

Hiç çalıştırılmamış bir recovery prosedürü bir hipotezdir ve bir sistemin hatadan sağ çıkmasını sağlaması beklenen kod da öyle. Bölüm 15, bu bölümün anlattığı restore'ları, failover'ları ve replay'leri nasıl prova edeceğinizi dahil, event-driven sistemlerin nasıl test edileceğini ele alıyor.

Bölüm 15 — Event-Driven Sistemleri Test Etmek

Parcelly'de Inventory ekibini, uzun bir hafta sonundan önceki Cuma günü düşünün. Test suite'leri yeşil: 400 test, birkaç dakikalık çalışma süresi, kimsenin tartışmadığı bir coverage. `OrderPlaced` handler'ının bir event kuran, handler'ı çağırان ve stoğun iki azaldığını kontrol eden bir testi var. Her seferinde geçiyor. Testler, broker yerine her mesajı bir kez, sırayla ve anında teslim eden bellek içi bir taklit kullanıyor.

Cumartesi günü bir deploy consumer'ları yeniden başlatıyor. Handler'lar ısınırken olağandan yavaş çalışıyor ve biri visibility timeout'unu aşıyor; böylece broker, birincisi hâlâ işlerken aynı `OrderPlaced` mesajını ikinci bir consumer'a veriyor. İkisi de inbox'ı okuyor, ikisi de boş buluyor ve ikisi de stoğu rezerve etmeye devam ediyor. Birinin başarısız olmasını sağlaması gereken unique constraint gerçek schema'da yoktu ve hiçbir test bunu fark etmedi, çünkü testler hiç constraint uygulamayan ve hiçbir zaman iki handler'ı aynı anda çalıştırmayan, elle yazılmış sahte bir veritabanı üzerinde koşuyordu. Suite kontrol ettiği hiçbir şeyde yanılmamıştı. Önemli olan tek şeyi hiç kontrol etmemişti.

Bu arada, Orders suite'indeki bir test 20 çalıştırmadan birinde başarısız oluyor. Ekip ona bakmayı bırakmış. Bir event'in gelmesi için iki saniye bekliyor ve bazen event üç saniye alıyor.

İki hikâyenin kökü aynı: testler var olmayan bir sistemi anlatıyor. Bu bölüm pratik bir soruyu yanıtlıyor. En kötü bug'ları yalnızca duplication, delay ve reordering altında ortaya çıkan kodu, yavaş, flaky ya da göz ardı edilen bir suite kurmadan nasıl test edersiniz?

Problem ve neden zor olduđu

Bölüm 1, happy path'in çaba harcamadan test edebildiğiniz ve inanmaya meyilli olduğunuz yol olduğunu savundu. Bölüm 2'den 14'e kadar olanlar ise happy path'in hiç göstermediği hatalara karşı savunmalar kurdu. Savunmalar koddur ve ele aldığı hata altında hiç çalıştırılmamış kod bir umuttur. Bölüm 3, her girdisi için bir "verified by" alanı olan bir hata modeli tanıttı. Bu alanın doldurulduğu yer bu bölümdür.

Üç özellik bu testi zorlaştırır.

Hatalar girdi değil, interleaving'dir. Bir fonksiyon testi bir girdi verir ve bir çıktıyı kontrol eder. Duplicate-reservation bug'ı belirli bir zamanlama gerektirir: zaman içinde çakışan iki teslimat ya da bir commit, ardından bir crash ve bir redelivery. Hiçbir girdi değeri onu tetiklemez. Testin, işlerin ne zaman olduğunu kontrol etmesi gerekir; bu da test harness'ının broker'ın davranışına, saate ve crash noktalarına sahip olması demektir.

Asenkronluk hakikat anını gizler. Bir handler çalıştıktan sonra etki başka bir servise saniyeler sonra düşebilir. Bir test hemen assert edemez ve ne kadar bekleyeceğini bilemez. Sabit bir süre uyumak her iki yönde de yanlıştır: çok kısaysa test flaky olur, çok uzunsa suite yavaş.

Sahteler tam da önemli yerlerde yalan söyler. Idempotency bir unique constraint'e dayanır. Atomicity bir transaction'a dayanır. Ordering partition atamasına dayanır. Bunları atlayan bir sahte size yeşil bir suite verir ve hiçbir bilgi vermez. Gerçekçiliğin bedeli hızdır, dolayısıyla zanaat, bunu nerede ödeyeceğinize karar vermektir.

Test piramidini uyarlamak

Klasik test piramidi hâlâ geçerli: altta çok sayıda hızlı test, üstte az sayıda yavaş test. Değişen şey, her katmanın ne işe yaradığı ve her katmanın hangi hatayı yakalamaktan sorumlu olduğudur.

Katman	Ne çalışır	Gerçek mi sahte mi	Neyi yakalar
Handler testleri	Gerçek bir veritabanına karşı tek bir handler	Gerçek veritabanı, broker yok	Yanlış etki, eksik inbox, bozuk atomicity, kötü duplicate ve ordering işleme

Katman	Ne çalışır	Gerçek mi sahte mi	Neyi yakalar
Contract testleri	Producer ve consumer arasındaki event şekilleri	Hiç servis yok	Bir consumer'ı bozan schema değişiklikleri
Component testleri	Gerçek bir broker, outbox ve relay ile tek bir servis	Gerçek broker, diğer servisler stub'lanmış	Wiring: outbox'tan relay'e, topic'e, ack'ler, retry'lar, dead-lettering
Akış testleri	Birlikte birkaç servis, bir hata enjekte edilmiş	Her şey gerçek, küçük ölçekte	Compensation'lar ve timeout'lar gibi servisler arası davranış
Production probe'ları	Canlı sistemde sentetik event'ler	Gerçek	Hiçbir pre-production ortamının göstermediği deploy ve yapılandırma sorunları

Tablodaki iki seçim birer cümleyi hak ediyor. Handler katmanı gerçek bir veritabanı motoru kullanır, çünkü inbox, outbox yazımı ve version kontrolü, bir sahtenin yeniden uygulaması gereken constraint'lere ve transaction'lara bağlıdır; bunları yeniden uygulamak Inventory suite'inin kandırılma şeklidir. Buna karşılık broker o katmanda dışarıda bırakılır. Bir handler, bir mesaj ve bir durumdan yeni bir duruma ve bazı giden mesajlara giden bir fonksiyondur. Onu doğrudan çağırabiliyorsanız hızlıca test edebilirsiniz.

Piramidin şekli, hataların yaşadığı yerden de çıkar. Bu kitaptaki savunmaların çoğu consumer tarafı ya da producer tarafı mantığıdır ve handler katmanı ona ucuza erişir. Yalnızca wiring bir broker'a ihtiyaç duyar ve orada test sayısı azdır, çünkü her biri iş mantığının değil, yapılandırmanın bir özelliğini kanıtlar.

Handler'ları test etmek

Bir handler testi, kontratı kitabın dilinde ifade etmelidir: bu mesaj ve bu durum verildiğinde, etki bir kez gerçekleşir ve giden event outbox'a yazılır. Sonra aynı test, mesajın tekrar geldiği ikinci bir perde alır.

```
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
```

Son assertion, ekiplerin unuttuğu olandır. İki kez rezerve etmeyen ama `StockReserved`'ı iki kez yayan bir duplicate yine de downstream'de zarar verir ve outbox'u kontrol etmek bunu yakalar.

Bu test, inbox'taki hızlı yol kontrolünün ele aldığı sıralı duplicate'leri kapsar. Tehlikeli durum, iki worker'ın da kontrolü geçtiği eşzamanlı duplicate'tir. Bunu test etmek için harness iki handler'ı aynı anda çalıştırır ve ikisini de transaction'dan hemen öncesinde tutmak için bir barrier kullanır; böylece interleaving umulmak yerine zorlanır.

```
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
```

Bu test, biri unique constraint'i kaldırıp oku-sonra-ekle kontrolüne güvenirse başarısız olur; bu, review'da fark edilmesi en zor regresyondur. Bölüm 7, constraint'in neden güvenlik ağı olduğunu açıkladı. Böyle bir test o açıklamayı çalıştırılabilir kılar.

Crash noktaları da aynı şekilde çalışır. Handler, ack'in gideceği yerde `crashPoint("before-ack")` çağırırsa, bir test o noktayı kurabilir ve orada simüle edilmiş bir crash fırlatabilir.

```
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
```

`on` bloğunun içindeki assertion'ı okuyun. Commit'in crash'ten sağ çıktığını kontrol eder; bu hikâyenin yarısıdır. Diğer yarısı, ardından gelen redelivery'dir. Yalnızca ikinci yarıyı kontrol eden bir crash testi, hiçbir şey commit etmemiş bir handler için de geçerdi.

Duplicate, reordering ve delay enjekte etmek

Tek tek testler, düşündüğünüz durumları kodlar. Hata modeli, düşünmediklerinizi de içerir; bu yüzden bütün bir event akışını bozan ve sonucun değişmediğini kontrol eden bir test katmanı ekledim.

Fikir bir yakınsama (convergence) özelliğidir. Bir aggregate için kaydedilmiş, doğru sıralı bir akış alın. Beklenen son durumu, event'leri doğrudan katlayarak (fold) hesaplayın; hiç broker yok. Sonra aynı event'leri handler üzerinden bozulmuş bir zamanlamayla teslim edin; bazıları bir pencere içinde karıştırılmış, bazıları çoğaltılmış olsun ve son durumun beklenene eşit olduğunu assert edin.

```

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

```

Bunu CI'da birkaç yüz seed için çalıştırın. İki ayrıntı onu dürüst tutar. Seed loglanır; böylece 4172 numaralı seed'deki bir hata, bir hikâye değil, dizüstü bilgisayarda yeniden çalıştırabileceğiniz bir hatadır. Ve referans sonuç handler'dan farklı bir kod yolundan gelir; böylece handler'daki bir bug beklentide de ortaya çıkamaz.

Delay üçüncü bozulmadır. Gecikmiş bir mesaj, ardıllarından sonra ya da bir timeout ateşlendikten sonra gelen mesajdır. Burada kontrol edilecek şey zamandır: sistem saatini doğrudan okuyan handler'lar ve saga'lar, beklemeden timeout'lar için test edilemez. Saati bir bağımlılık olarak geçirirdim ve testin onu ilerletmesine izin verirdim. Bölüm 10, bir saga'nın içinde olabileceği durumları listeledi. Bu durumda bir saga testi bir tabloya dönüşür: her durum için gelebilecek her event'i, geç olanlar dahil, teslim edin ve ortaya çıkan durumu ve yayılan compensation'ı kontrol edin.

Saga durumu	Gelen event	Beklenen sonuç
AWAITING_RESULTS	PaymentAuthorized	AWAITING_RESULTS'ta kalır, stoğu bekler
AWAITING_RESULTS, ödeme tutuluyor	Timeout ateşlenir	COMPENSATING; ReleaseStock ve VoidAuthorization komutları outbox'a yazılır

Saga durumu	Gelen event	Beklenen sonuç
COMPENSATING ya da CANCELLED	Geç gelen StockReserved	Saga geçişi yok. Geç gelen bir StockReserved, Inventory'nin ReleaseStock ona ulaşmadan önce rezerve ettiği anlamına gelir; bu yüzden o komut stoğu serbest bırakır. Release önce gelmiş olsaydı, tombstone'u rezervasyonu reddederdi (Bölüm 10)
COMPLETED	Duplicate PaymentCaptured	Yok sayılır

Önemli olan üçüncü satırdır. Timeout'tan sonra geç varış, saga'ların yanlış yaptığı durumdur; çünkü happy path ve timeout yolu ayrı ayrı test edilmiş, çarpışma ise hiç edilmemiştir. Tablo, akış diyagramını izleyen biri tarafından değil, "bu durumda başka ne gelebilir?" diye soran biri tarafından yazılmalıdır.

Not: Testleriniz rastgele bir zamanlamaya dayanıyorsa, bulunan birkaç başarısız seed'i sabit regresyon durumları olarak saklayın. Bir kez yeniden üretilmiş bir bug, yeniden üretilmiş kalmalıdır.

Contract testleri ve uyumluluk kontrolleri

Handler testleri kendi event'lerini kurar; bu, consumer'ı consumer'ın producer hakkındaki fikrine karşı test ettikleri anlamına gelir. Producer bir alanı değiştirdiğinde consumer'ın suite'inde hiçbir şey bunu fark etmez. Bölüm 12 uyumluluk kurallarını ve onları uygulayan bir schema registry'yi anlattı. Bu yapıyı kapsar. Contract testleri ise registry'nin göremediği kısmı kapsar: her consumer'ın gerçekte neye bağımlı olduğunu.

Bir **consumer-driven contract**'ta her consumer bir event'ten okuduğu alanları ve tipleri yazar ve producer'ın pipeline'ı schema'sının bunları hâlâ sağlayıp sağlamadığını kontrol eder. Yön önemlidir. Producer hangi alanların yük taşıdığını bilemez, consumer bilebilir. Dolayısıyla bir contract, consumer'a ait olan ve producer'ın CI'nın okuyabileceği bir yerde saklanan küçük bir dosyadır.


```

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)

```

İki yarıya ihtiyaç var. Producer'ın pipeline'ı, kayıtlı her contract'a karşı `verifyContracts` çalıştırır. Consumer'ın pipeline'ı ise kendi contract'larındaki her örnek event'i gerçek deserializer'ıyla decode eder ve handler'ı onun üzerinde çalıştırır. Birlikte, bir alanı sağlamayı bırakan bir producer'ı ve hiç beyan etmediği bir alanı okumaya başlayan bir consumer'ı yakalarlar. İkinci yakalama yalnızca örnek contract'tan kuruluysa ve yalnızca beyan edilmiş alanları içeriyorsa geçerlidir; producer'ın tam event'inden kopyalanmış bir örnek, consumer'ın beyan edilmemiş bir alanı okumasına ve yine de geçmesine izin verir.

Örnekler, Bölüm 12'nin önerdiği gibi, retention içinde kalan her sürümü kapsamalıdır. Replay orada ne varsa onu okur ve yalnızca en yeni şekli işleyen bir consumer ilk eski event'te başarısız olur. Registry'nin uyumluluk kontrolünün CI'da ait olduğu yer de burasıdır: bir producer'ın schema değişikliği, topic'e ulaştıktan sonra değil, kaydedilebilmeden önce build'i başarısız kılmalıdır.

Sınır konusunda dürüst olun. Bir contract, `amount_minor`'ın var olduğunu ve bir integer olduğunu kanıtlar. Hâlâ "minor birimlerle tutar" anlamına geldiğini kanıtlamaz. Yapıyı değiştirmeden anlamı değiştirmek her contract testini geçer ve yalnızca bir insan reviewer'ın ya da consumer'daki bir davranış testinin gözüne çarpar. Yeşil contract'ları anlam hakkında bir beyan olarak ele almazdım.

Eventually-consistent assertion'lar

Bir end-to-end test bir şey yayımlar ve sonra başka yerdeki bir etki üzerine assert eder. Etki eninde sonunda düşer, dolayısıyla assertion bunu tolere eden bir biçime ihtiyaç duyar. Standart biçim, bir deadline'lı poll'dur.

```
function eventually(check, timeout, interval):
    deadline = now() + timeout
    while true:
        observation = check()
        if observation.satisfied:
            return
        if now() >= deadline:
            // report what was last seen, not only that time ran out
            raise AssertionError(observation.description)
        sleep(interval)
```

Yapı görüldüğünden daha önemlidir. Kontrol, deadline'a ulaşıldıktan sonra bir kez daha çalışır; böylece son uyku sırasında sağlanan bir koşul yine de geçer, interval timeout'a göre kısadır ve hata mesajı son gözlemi taşır. "Timed out" diyen bir mesaj hiçbir şey söylemez. "order-1 AWAITING_RESULTS'ta takılı, 1 held event" diyen mesaj nereye bakacağınızı söyler.

Flaky asenkron testlerin kısa bir neden listesi vardır ve çoğu nedeninde düzeltilebilir.

- **Sabit sleep'ler.** `eventually` ile değiştirin. Timeout bir tavana dönüşür ve test koşul sağlanır sağlanmaz biter.
- **Hızlı bir dizüstü bilgisayara göre ayarlanmış timeout'lar.** Tavanı cömert tutun, çünkü koşul hızlı sağlandığında hiçbir şeye mal olmaz ve onu performans beklentilerinden ayrı tutun. Bir latency bütçesi, farklı bir ortamda farklı bir testtir.
- **Testler arasında paylaşılan durum.** Paylaşılan bir topic'te kalan bir mesaj, sonraki testin yayımlamadığı bir event görmesine yol açar. Her teste kendi correlation id'sini verin ve ona göre filtreleyin, ya da kendi topic'ini veya queue'sunu verin.
- **Söz verilmeyen yerde varsayılan ordering.** Bir test farklı partition'lardan gelen iki event'in sırası üzerine assert ediyorsa, broker'ın hiç söz vermediği bir şey üzerine assert ediyordur (Bölüm 9).

- **Hiçbir şeyin olmadığını assert etmek.** "İkinci bir e-posta gönderilmedi" poll ile başarıya dönüştürülemez, çünkü daha uzun beklemek yine de bir tane gösterebilir. Sistemi quiescence'a sürmeyi, sonra assert etmeyi tercih edin. Quiescence, tek bir consumer'ın değil tüm yolun özelliğidir: outbox boştur, relay boşta, ilgili her consumer'ın lag'i sıfırdır, hiçbir retry ya da timer zamanlanmamıştır ve retry ile delay queue'ları boştur. Ancak o zaman, mesajı hâlâ uçuşta olan bir downstream hop dahil, etkiyi üretebilecek hiçbir şey kalmamış olur.

Son durum için harness'ın "hiçbir şey beklemede kalmayana kadar çalıştır" diyebilmesi gerekir. Bir handler ya da component testinde harness, queue boşalana ve enjekte edilmiş saatin ateşlenecek timer'ı kalmayana kadar consumer'ı pompalayabilir. Bu, duvar saati zamanını beklemekten hem daha hızlı hem de daha deterministiktir ve davranışsal testlerin çoğunu handler katmanında tutmamın ve gerçek zamanı yalnızca wiring'in test edildiği yerde kullanmamın ana sebebidir.

Fault injection ve chaos

Şimdiye kadarki her şey kodu izolasyon içinde test eder. Sıradaki soru, bir araya getirilmiş sistemin hata modelinin iddia ettiği gibi davranıp davranmadığıdır. Fault injection, çalışan bir ortama bilerek bir hata koyar ve sistemin tepkisini kontrol eder.

Bölüm 3'teki hata modelinden başlayın ve araçları değil, girdileri seçin. Her biri için hipotezi girdinin kendi terimleriyle yazın: "bir consumer commit ile ack arasında öldürülürse, redelivery hiçbir şeyi değiştirmez". Sonra onu enjekte edin: süreci o anda öldürün, relay'i duraklatın, broker'ın response'larını geciktirin, bir servis ile veritabanı arasındaki bağlantıyı kesin, bir broker node'unu yayımlama sırasında yeniden başlatın. Sonrasında log'ları değil, invariant'ları kontrol edin: stok rezervasyonlarla eşleşiyor, ledger'da her capture için bir kayıt var, hiçbir outbox satırı takılı değil.

O listeye recovery drill'leri de ekleyin, çünkü hiç denenmemiş bir savunma bir umuttur. Hiç geri yüklenmemiş bir backup klasik örnektir. Birini scratch bir ortama geri yükleyin ve bir restore'un outbox'a, inbox'a ve consumer offset'lerine ne yaptığını kontrol edin; bunu Bölüm 14 ele alıyor: yeniden

dirilen outbox satırları, geri sarılan ve bu yüzden duplicate'lerin artık tanınmadığı inbox satırları, artık eşleşmeyen offset'ler. Bir standby cluster'a failover yapın ya da prova edin ve toparlanma süresini ve kaybedilen veriyi, o akış için Bölüm 14'te seçilen recovery hedefleriyle karşılaştırın. Ve yetiştirmeyi test edin: bir backlog oluşturacak kadar uzun süre bir consumer'ı duraklatın, trafik gelmeye devam ederken yeniden başlatın ve backlog'un akışın deadline'ından ve retention dolmadan önce erimesini kontrol edin. Bu, Bölüm 13'teki drain rate'i ve Bölüm 6'daki uçustaki iş sınırlarını kontrol eder.

Bunu artan blast radius katmanlarında çalıştırırdım. Ayrılmış bir ortamdaki akış testleri CI'da bir zamanlamayla, küçük ve tekrarlanabilir şekilde hata enjekte eder. Daha büyük bir ortamdaki periyodik bir game day, bunları bir ekip egzersizi olarak enjekte eder; kısmen bug bulmak, kısmen de birinin fark edip etmediğini öğrenmek için. Bölüm 13'teki alert'ler de kod kadar test altındadır. Bir consumer'ı öldürdüyseniz ve hiçbir alert ateşlenmediyse, monitoring'de bir bug buldunuz demektir.

Production'da chaos olası bir sonraki adımdır, bir başlangıç noktası değil. Yalnızca sistemin aynı hataya pre-production'da dayandığı gösterildiğinde, blast radius sınırlanabildiğinde ve biri deneyi anında durdurabildiğinde anlamlıdır. Bunlara sahip olmayan bir ekip orada başlamamalıdır ve game day'lerde uzun süre kalmaktan utanmazdım. Bir fault-injection pratiğinin değeri hipotez, enjeksiyon ve kontrol döngüsündedir; enjeksiyonun ne kadar dramatik olduğunda değil.

Uyarı: Bir broker'ı, bir veritabanını ya da bir ödeme sağlayıcı hesabını gerçek trafikle paylaşan bir fault injection çalıştırması, planı olan bir kesintidir. Herhangi bir şeyi kapatmadan önce ortamı izole edin ya da enjeksiyonu sentetik trafikle sınırlayın.

Regresyon testi olarak replay

Bir event log, gerçekte olanın bir kayıdır ve replay, yani geçmişin bilinçli olarak yeniden okunması, onu bir regresyon suite'ine dönüştürür. Gerçek event'lerden bir dilim alın, onları bir consumer'ın yeni sürümünden bir scratch ortamında geçirin ve türetilmiş durumu eski sürümün ürettiğiyle karşılaştırın. Farklar ya bir bug düzeltilmesinde olduğu gibi istenendir ya da

regresyondur. Replay'in ayrıca bir boşluktan ya da bir restore'dan, geçmişini yeniden okuyarak toparlanmak gibi operasyonel bir kullanımı da vardır ve bunu Bölüm 14 ele alıyor. Bu bölüm yalnızca onu test etmek için kullanmakla ilgilidir.

Bu, elle yazılmış fixture'ların asla yakalayamayacağı şeyleri yakalar: üç schema sürümü önceki event şekilleri, kimsenin hayal etmediği sıralar, testlerde hep var olan ve geçmişin küçük bir kesirinde null olan bir alan. Aynı zamanda Bölüm 12'nin eski event'lerin hâlâ decode edildiği iddiasının testine en yakın şeydir.

Uygulanabilir bir prosedür beş adımdır.

1. **Bir dilim seçin.** Bir zaman penceresi ya da bir aggregate örnekleminin tüm geçmişi. Her schema sürümünü ve bilinen incident'ları içeren bir örnek, rastgele olandan iyidir.
2. **Maskelleyin.** Replay edilen event'ler alıcı verisi içerir. Özgün sınırlarını terk etmeden önce anonimleştirin ya da scratch ortamını production ile aynı kontroller altında tutun.
3. **Etkileri ve yayılan event'leri devre dışı bırakın.** Replay eden consumer kart çekmemeli, e-posta göndermemeli ya da kargo firmalarını çağırmamalıdır. Etkiler, kendilerinden ne istendiğini kaydeden stub'lara gider ve stub'ların kayıtları çıktının parçasıdır. Outbox'ı ve relay'i bir scratch broker'a yayımlamalı ya da yakalanmalı, asla production topic'lerine bağlanmamalıdır; yoksa replay geçmişini canlı consumer'lara yeniden yayımlar.
4. **Eski ve yeni sürümleri aynı dilim üzerinde çalıştırın.** Türetilmiş durumu ve yayılan event'leri aggregate aggregate karşılaştırın.
5. **Her farkı sınıflandırın.** İstenen, zararsız ya da bir regresyon. Kimsenin açıklayamadığı bir fark, aksi kanıtlanana kadar bir regresyondur.

Replay, geçmişin hâlâ orada olduğu log ailesi için geçerlidir. Klasik bir queue'da tüketilen mesajlar gitmiştir. Eşdeğeri yakalanmış bir örnektir: mesajların bir kesirini anonimleştirilmiş olarak kontrol ettiğiniz bir depoya kopyalayan bir tap. Yakalama yalnızca sampling kadar iyi olduğu için daha zayıf bir araçtır, ama aynı amaca hizmet eder.

Replay'in bir maliyet sınırı da var. Büyük geçmişler gerçek zaman ve depolama alır ve karşılaştırma araçları ekibinizin kurup çalışır tutması gereken bir şeydir. Tam geçmiş replay'ini event'lerin nasıl yorumlandığına dokunan değişiklikler için saklardım ve rutin sürümler için daha küçük bir dilim kullanırdım.

Production'da doğrulamak

Hiçbir pre-production ortamı production'ın trafik şeklini, verisini, yapılandırmasını ve tuhaflıklarını yeniden üretmez. Bir deploy'dan sonra bir topic üzerinde eksik bir izin, eski kalmış bir secret ya da yanlış yönlendirilmiş bir queue gibi bazı hatalar yalnızca gerçek sistemde ortaya çıkar. Cevap pre-production testlerinden vazgeçmek değil, hataların bulunduğu yerde çalışan son bir katman eklemektir.

Bir **sentetik event**, gerçek pipeline boyunca yol alan ve sonunda kontrol edilen, bilinçli olarak enjekte edilmiş, açıkça işaretlenmiş bir trafik parçasıdır. Parcelly için bu, bir probe tarafından birkaç dakikada bir verilen ve OrderPlaced'ten ShipmentScheduled'a kadar akan bir sipariştir.

```
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)
```

Etiket güvenlik mekanizmasıdır ve bu, `correlation_id` üzerindeki bir örnek değil, özel bir `synthetic` header'dır. Aynı zamanda davranışı da kontrol eden bir tanımlayıcı bir tuzaktır ve Bölüm 13, correlation ile arama her yerde çalışsın diye `order-81724` gibi tek tip bir id biçimi ister.

Bedeli, header'ı kopyalamayı unutan bir hop'un, hiçbir hata olmadan bir probe'u gerçek trafiğe çevirmesidir. Her servis onu `correlation_id`'yi kopyalayan aynı helper ile kopyalar, probe ayrıca Payments ve Shipping'in birbirinden bağımsız olarak tanıdığı özel bir sentetik alıcı hesabı kullanır ve bir test header'ı bilerek düşürüp hiçbir gerçek çekim ya da gönderi oluşmadığını kontrol eder.

Her servis bayrağı tanır ve çekirdek mantığını değil, kenarlarını değiştirir: Payments sağlayıcının sandbox'ıyla konuşur, Shipping kargo firmasına devri atlar, Notifications teslimatı bastırır ve Ledger, finansın raporlardan hariç tuttuğu bir test hesabına kayıt düşer. Bu dallar yalnızca probe'lar için çalışan koddur, dolayısıyla içlerindeki bir hata yalnızca production'ın açığa çıkardığı bir bug'dır. Inventory de aynı muameleye ihtiyaç duyar, yoksa her probe gerçek stok rezerve eder: ayrılmış sentetik bir SKU'ya karşı rezerve edin ya da etiketli siparişler için rezervasyonu atlayın. Dalların kendilerini handler katmanında test ederdim ve probe'u bilerek bir şeyi bozup başarısız olduğunu kontrol ederek test ederdim.

Temizliğin bir tuzağı daha var. Timeout olan bir probe'un bitirememiş olması şart değildir; event'leri hâlâ uçuşta olabilir ve geç gelen bir `StockReserved` ya da `PaymentAuthorized`, iptalden sonra gelebilir. Probe'un en baştan kaydedilmesinin ve her servisin `synthetic` bayrağına göre anahtarlanmasının sebebi budur: periyodik bir süpürme bütçeden eski probe'ları bulup geride bıraktıkları her şeyi iptal edebilir ve iptalin kendisi idempotent'tir, dolayısıyla iki kez çalıştırmak zararsızdır.

Bir probe'un bir bütçeye de ihtiyacı var ve bütçenin bir sahibi olmalı. Başarısız bir probe birini page eder, dolayısıyla bütçe yanlış alarmlardan kaçınacak kadar cömert ve anlamlı olacak kadar sıkı olmalıdır. Bölüm 13, end-to-end latency'yi ve ona nasıl alert kurulacağını ele alır. Probe bu ölçümün en ucuz kaynağıdır, çünkü son saatte hiçbir gerçek siparişin dokunmadığı kısımlar dahil, tüm yolu çalıştırır.

Production doğrulaması, Bölüm 13'teki reconciliation job'larını da içerir. Bir reconciliation çalıştırması, gerçek veri üzerinde sürekli yürütülen bir invariant testidir ve kimsenin simüle etmeyi düşünmediği hataları yakalar. Onu piramidin son katmanı sayıyorum.

Trade-off'lar ve maliyetler

Gerçekçilik zamana mal olur. Gerçek veritabanları, gerçek broker'lar ve enjekte edilmiş hatalar testleri yavaşlatır. Hızlı olan suite, önemli olanı atladığı için hızlıydı. Yavaş katmanları açıkça bütçelerdim: handler testleri saniyeler içinde, component testleri birkaç dakikada, akış testleri ve fault injection her commit'te değil bir zamanlamayla.

Contract testleri koordinasyon ekler. Her consumer, producer'ın pipeline'ında yaşayan dosyalara sahiptir ve birinin onları güncel tutması gerekir. İki servisli ve tek ekipli bir sistem için schema'nın paylaşılan bir testi yeterli olabilir. Contract testi, producer ve consumer ekipleri bağımsız deploy ettiğinde karşılığını vermeye başlar ve ondan önce az kazandırır.

Rastgele zamanlamalar araştırma işi ekler. Başarısız bir seed hem bir bulmaca hem de gerçek bir kusurdur. Bundan hoşlanmayan ekipler, test artık hiçbir şey bulmayana kadar seed sayısını düşürmeye varır. Çare, seed'leri loglamak ve sabit regresyonları tutmaktır; tekniği terk etmek değil.

Chaos ve probe'ların blast radius'u vardır. İkisi de canlı ya da paylaşılan sistemlere dokunur. Ne zaman kullanılmaz: sistem hataya henüz izolasyonda dayanmadığında, hiç kimse çalıştırmayı anında durduramadığında ya da etiketli trafik gerçek paradan ve gerçek alıcılardan ayrı tutulmadığında.

Replay'i kurmak pahalıdır. Maskeleye, etki stub'ları ve karşılaştırma altyapısı başlı başına bir projedir. Kısa geçmişli küçük bir sistem, handler testlerinden ve crash noktalarından daha fazlasını kazanır. Başa baş noktanızın nerede olduğunu bilmiyorum, çünkü bu consumer'ların ne sıklıkla değiştiğine ve geçmişin yanlış yorumlanmasının ne kadar maliyetli olacağına bağlıdır; ama beklenmedik bir eski event yüzünden bir incident yaşamış bir ekip bunu genellikle tahmin edebilir.

Sık yapılan hatalar

- **Hiç yanlış davranmayan bir broker sahtesine karşı test etmek.** Hızlı olduğu ve ilk testler geçtiği için cazip gelir. Sahtenin duplicate etmesini, geciktirmesini ve karıştırmasını sağlayın ya da hatayı bu bölümdeki gibi açık çağrılarla handler testlerine koyun.

- **Veritabanının constraint'lerini taklit etmek.** Elle yazılmış bir sahte uniqueness'i, isolation'ı ya da atomicity'yi uygulamaz; bu yüzden savunma var olsa da olmasa da testler geçer. Bir constraint'e bağlı her test için gerçek motoru, bir konteynerde ya da scratch bir instance'ta kullanın.
- **Sabit bir süre uyumak.** Beklemenin en basit yolu gibi görünür. Koşulu bir tavanla poll edin ve başarısızlıkta son gözlemi raporlayın.
- **Commit ile ack arasındaki crash'i hiç test etmemek.** Egzotik hissettirir, ama her deploy'da olur. Her handler'a bir crash noktası ve onu kuran bir test ekleyin.
- **Yalnızca son durumu test etmek, outbox'u değil.** Durumu değiştirmeyen bir duplicate yine de ikinci bir event yayabilir. Outbox'a yazılanlar üzerine assert edin.
- **Gerekliliği değil, varlığı kontrol eden contract testleri.** Producer'ın schema'sında optional olan bir alan yok olabilir. Contract'a "required" dedirtin ve bunu kontrol edin.
- **Seed'siz rastgele testler.** Yeniden üretilmeyen bir hata silinir. Seed'i loglayın ve başarısız seed'leri sabit durumlar olarak saklayın.
- **Hiç geri yüklenmemiş bir backup'a güvenmek.** Backup job'ı başarı raporlar, böylece savunma test edilmiş gibi hissettirir. Onu bir zamanlamayla scratch bir ortama geri yükleyin ve sonrasında outbox'ı, inbox'ı ve offset'leri kontrol edin.
- **Probe'ları gerçek paraya karşı çalıştırmak.** Canlı ödeme sağlayıcısına ya da gerçek bir kargo firmasına ulaşan sentetik trafik bir probe değildir. Etiketli trafiği kenarda sandbox'lara yönlendirin ve bu yönlendirmeyi test edin.

Temel çıkarımlar

- Her mesajı bir kez ve sırayla teslim eden bir broker varsayan testler var olmayan bir sistemi test eder; bu yüzden bu kitaptaki her savunmanın, savunduğu hatayı uygulayan bir teste ihtiyacı vardır.
- Handler testleri gerçek bir veritabanı motoru üzerinde olmalıdır, çünkü inbox, outbox ve version kontrolleri, sahtelerin yeniden üretmediği constraint'lere ve transaction'lara bağlıdır.

- Duplicate, reordering ve delay en iyi, bozulmuş bir akış üzerinde bir yakınsama özelliği olarak test edilir; seed loglanır ki her hata yeniden üretilebilsin.
- Consumer-driven contract'lar yapısal kırılmaları deploy'dan önce yakalar, ama anlam değişikliğini göremez; bu yüzden bir alanın neyi söylemesi gerektiğine dair bir insan review'ı sürecin parçası olarak kalır.
- Sabit sleep'leri bir koşula ve bir tavana karşı poll etmekle değiştirin, sessizlik yerine quiescence üzerine assert edin ve timeout'ların beklemeden test edilebilmesi için saati enjekte edin.
- Fault injection, hata modelinin girdilerinden başlamalı ve sonrasında invariant'ları kontrol etmeli; blast radius'u ancak sistem her adıma dayandığını kanıtladıkça artırmalıdır; ayrıca restore'lar, failover'lar ve yetiştirme için recovery drill'lerini de içermelidir.
- Gerçek geçmişin replay'i ve production'daki sentetik probe'lar, elle yazılmış fixture'ların ve pre-production ortamlarının yakalayamadığını yakalar; güvenli olmaları için maskeleye, etki stub'larına ve etiketli trafiğe ihtiyaç duyarlar.

Bu şekilde test edilmiş bir sisteme biraz daha güvenilebilir, ama asla tamamen değil; ve bu, bir ekibin bir tasarıma hiç güvenmeden önce neyi kontrol etmesi gerektiği sorusunu bırakıyor. Bölüm 16 kitabı bir güvenilirlik kontrol listesinde topluyor ve kontrol listesinin yapamadıklarıyla kapanıyor.

Bölüm 16 — Bir Güvenilirlik Kontrol Listesi ve Kapanış Düşünceleri

Parcelly'de sizin bir perşembe günü yapılan bir design review'ı düşünün. Ekip yeni bir akış eklemek üzere: satıcılar kısmi iade teklif edebilecek ve Payments bunun için yeni bir event yayacak. Diyagram temiz. Biri okları çizmiş, event'i adlandırmış ve consumer'ları sıralamış. Toplantı 30 dakikaya ayrılmış ve 20 dakikada bitecek gibi görünüyor.

Sonra bir tech lead düz bir soru soruyor: "İade yapılırsa ve event hiç publish edilmezse ne olur?" Oda sessizleşiyor; örüntüyü kimse bilmediği için değil, kimse sormadığı için. Outbox codebase'in başka bir yerinde var. Idempotent consumer da başka bir yerde var. Bu yeni akışın onları kullanıp kullanmadığı ayrı bir mesele ve diyagram bunu söylemiyor.

Güvenilirlik genellikle böyle aşınır. Ekip örüntüleri biliyor. Silikleşen, her yeni akış için her örüntünün gerçekten yerinde olduğunu kontrol etme alışkanlığı. Dolayısıyla bu son bölümün sorusu pratik: sistemin kötü bir günde sözlerini tutup tutamayacağını öğrenmek için akış akış neler sorardınız ve cevapların sahibi kim olmalı?

Problem ve neden zor olduğu

Güvenilirlik, birleşim yerlerinin bir özelliğidir ve birleşim yerleri hiçbir tek dosyada görünmez. Bir producer doğru olabilir, bir consumer doğru olabilir ve aralarındaki akış yine de event kaybedebilir, çünkü devir hiç incelenmemiştir. Bölüm 1 bu argümanı kitabın başında yaptı ve o zamandan beri bölümler birleşim yerlerini tek tek ele aldı.

Şimdiki zorluk, bilginin 15 bölüme yayılmış olması. Hiç kimse bir review sırasında bu kadarını aklında tutmaz. Deadline baskısı altında atlanan sorular, cevabı "o gün gelene kadar çalışır" olanlardır: dedup window, dead-letter owner'ı, uyumluluk kontrolü, sayıya değil yaşa göre tetiklenen alert.

Kontrol listesi olağan cevaptır ve bilinen sınırları vardır. Düşünemez. Bir maddeyi anlamayan biri yine de onu işaretleyebilir ve tamamlanmış bir liste sahte güven verir. Aşağıdaki listeyi bir formun geçebileceği bir kapı olarak değil, bir konuşma için istem olarak kullandım. Her madde, orada olma nedenini açıklayan bölüme işaret eder; böylece katılmayan bir okuyucu gerekçeyi bulabilir ve öncülle tartışabilir.

Her maddeyi genel olarak sistem hakkında değil, Parcelly sipariş akışı gibi belirli bir akış hakkında bir soru olarak ele alın. "Bir outbox kullanıyoruz" bir cevap değildir. "Orders, OrderPlaced'i sipariş satırıyla aynı transaction'da outbox'ına yazar" bir cevaptır.

Listenin arkasındaki yöntem

Liste yeni bir fikir değil. Kitabın tezinin bir review'a uygulanmış hâli: güvenilir event-driven sistemler message broker'ların değil, **failure semantics**'in etrafında kurulur: belirli bir birleşim yeri için neyin başarısız olabileceği, bunu nasıl gözlemleyeceğiniz ve hangi garantinin ayakta kalacağı.

Bu üç soru eder ve aşağıdaki her madde, bunlardan birinin bir birleşim yeri hakkında sorulmuş hâlidir. Bir hatayı adlandıran madde neyin başarısız olabileceğini sorar. Bir metriği, bir alert'i ya da bir testi adlandıran madde bunu nasıl gözlemleyeceğinizi sorar. Bir ayarı, bir constraint'i ya da bir örüntüyü adlandıran madde hangi garantinin ayakta kalacağını sorar. Maddelerinin üç sorudan yalnızca birine cevap verdiği bir birleşim yeri yarım kalmıştır ve ikinci soruya cevabı olmayan bir birleşim yeri kimseye haber vermeden başarısız olur.

Kontrol listesi

Liste, hatanın yaşadığı yere göre gruplanmıştır. Sağdaki sütun, cevap "hayır" ya da "emin değilim" olduğunda yeniden okunacak bölümü adlandırır.

Producer

Kontrol	Bölüm
Durum değişikliği ve event tek bir yerel transaction'da yazılır, iş yolunda doğrudan publish yoktur.	4

Kontrol	Bölüm
Outbox satırı tam envelope'u taşır ve <code>event_id</code> yazma anında oluşturulur, asla yeniden üretilmez.	4
Ekip, publication'ın at-least-once olduğunu ve duplicate'lerin bedel olduğunu yazıya dökmüştür.	4
Relay önce publish eder, sonra işaretler ve artan bir id cursor'ına göre değil, duruma göre seçer.	5
Varsa birden fazla relay, satırları güvenle claim eder ve bir aggregate'in satırlarını tek bir worker'da tutar.	5
Yayınlanmamış en eski satırın yaşı ölçülür ve yalnızca yayınlanmış satırlar purge edilir.	5

Bir ekip bir review'da yalnızca tek bir soruyu cevaplayabilecekse, o ilki olmasını isterdim, çünkü sonraki her garanti event'in var olduğunu varsayar. Bir akışın outbox'a ihtiyacı yoksa, çünkü adımları yerel bir transaction'ı paylaşabiliyorsa, ekip bunu ve nedenini söylemelidir (Bölüm 1).

Broker

Kontrol	Bölüm
Her önemli event için ekip, "published"ın ne anlama geldiğini kaydetmiştir: bir producer acknowledgment'ının arkasındaki replication ve flush ayarları.	2, 3
Acknowledgment ya da offset commit etkiden sonra gerçekleşir; böylece semantik at-least-once'tır, kazara bir at-most-once değil.	2
Visibility timeout'lar, session timeout'lar ve handler içi retry süresi birlikte ayarlanır; böylece yavaş iş, hâlâ çalışırken yeniden teslim edilmez.	2, 6
Partition key aggregate'i tanımlar ve ekip, broker ailesinin hangi ordering'i sunduğunu bilir.	9
Her dil ve her producer aynı partitioner'ı ve key encoding'ini kullanır ya da partition'ı relay hesaplar.	9
Retention, gerçekçi en uzun kesintiyi artı onu fark etme süresini aşar.	3, 13
Consumer'lar deploy'da zarıfçe durur; böylece rebalancing rutin bir duplicate kaynağı olmaz.	3

Bu maddelerin çoğu kod değil, ayardır. Bir incident varsayılanların ne olduğunu gösterene kadar varsayılanlarda kalırlar ve o noktada izlerini sürmek zordur.

Consumer

Kontrol	Bölüm
Inbox satırı ve iş değişikliği, key üzerinde bir unique constraint ile tek bir yerel transaction'da commit edilir.	7
Dedup window, retention'ı, redrive'ı ve planlı replay'i aşar ve bir business-key constraint'i, ötesinde domain tablosunu kapsar.	7
Harici yan etkiler, denemeden değil işten türetilmiş deterministik bir idempotency key taşır.	7
Hatalar transient, permanent ya da unknown olarak sınıflandırılır ve retry'lar jitter'lı exponential backoff, bir deadline ve paylaşılan bir budget kullanır.	6
Yürütülen iş sınırlandırılır, dependency yavaşladığında concurrency düşer ve consumer başarısız olmak yerine duraklar; böylece aşırı yük lag olarak görünür. Ingress'teki bir limit, yeni işi çağırana net bir sinyalle reddeder ya da erteler.	3, 6
Bir poison message, gövde, header'lar, kaynak pozisyon, hata, deneme sayısı ve consumer versiyonuyla birlikte bir DLQ'da biter ve orijinal ack'lenmeden önce publish edilir.	8
Tek bir kayıttaki takılıp kalan consumer, throughput'suz lag ve pozisyon başına crash sayacıyla tespit edilir ve biri fark edene kadar döngüde kalmaz.	8, 13
Redrive orijinal event_id'yi korur, rate-limit'li batch'ler hâlinde çalışır ve bir pause switch'i vardır.	8
Aggregate başına sıra, keyed lane'ler, versiyon kontrolleri ve timeout'ları olan hold-back buffer'larla korunur.	9
Pivot'tan önceki her workflow adımının idempotent bir compensation'ı vardır ve her bekleyişin bir deadline'ı vardır.	10

Bu en uzun gruptur, çünkü kitabın örüntülerinin çoğu buraya iner. Ağır geliyorsa bu doğru bir his: at-least-once delivery yükü consumer'a devreder.

Contract

Kontrol	Bölüm
Her event bir olguyu belirtir, geçmiş zamanda adlandırılır ve tek bir sahip servisi vardır.	11
Integration event, domain event'ten eşlenmiş, bilinçli bir public sözleşmedir ve payload'da kişisel veri ya da secret yoktur.	11
Command imperative kipte adlandırılır, tek bir hedef alıcısı vardır, bir command_id taşır, reddin göndericiye ulaşması için bir reply path'i vardır ve asla event diye adlandırılmaz; gönderen kendi deadline'ıyla bekler.	10, 11

Kontrol	Bölüm
<code>correlation_id</code> akış boyunca kopyalanır ve <code>causation_id</code> doğrudan tetikleyiciyi gösterir.	11
Envelope'un timestamp, identifier ve para gibi wire type'ları, onu okuyan ya da yazan her dil için sabitlenmiştir.	11
Her değişiklik compatible ya da breaking olarak sınıflandırılır ve uyumluluk deploy pipeline'ında kontrol edilir.	12
Anlam değişikliği bir insan tarafından gözden geçirilir, çünkü hiçbir araç onu tespit etmez.	12
Consumer'lar kullanmadıkları alanları yok sayar ve görmedikleri değerlere tolerans gösterir, bir mesajı ileten hop bilinmeyen alanları korur ve <code>schema_version</code> 'ını bilmedikleri bir event reddedilir ya da park edilir.	12
Breaking değişiklikler, aynı dokümanda planlanmış rollback'lerle birlikte, bir dizi non-breaking adım olarak migrate edilir ve eski versiyonu hâlâ okuyan consumer'lar bilinir.	11, 12

Operasyon

Kontrol	Bölüm
Trace context ve <code>correlation_id</code> her asenkron hop'tan sağ çıkar; böylece bir siparişin hikâyesi tanımlayıcısından yeniden kurulabilir.	13
Consumer lag, mesaj yaşı, outbox yaşı, retry oranı ve DLQ'ya gelenler ölçülür ve alert'ler yaşa ve orana göre tetiklenir.	13
Reconciliation job'ları, source of truth'u türetilmiş durumla bir takvime göre karşılaştırır ve onarımlar çalışmadan önce gözden geçirilir.	13
"Her capture edilmiş ödemenin bir ledger entry'si vardır" gibi iş invariant'ları doğrudan kontrol edilir.	13
Handler'lar duplicate'ler, reordering ve gecikmeyle test edilir ve contract test'ler CI'da çalışır.	15
En önemli akışlar için fault injection ve replay tabanlı regresyon testleri vardır.	15
Her store'un belirtilmiş bir recovery point objective'i (RPO) ve recovery time objective'i (RTO) vardır ve ekip, bir restore'da hangi store'un zamanda geri gittiğini bilir.	14
Bir replay prosedürü; consumer group'ları, timestamp'i ve marjını, bastırdığı yan etkileri ve yeniden oynatılan her aralığın sığması gereken dedup window'u adlandırır.	7, 14
Bir broker migration planı relay'i tek geçiş noktası olarak kullanır, consumer deduplication ile dual-publish eder ve cutover ile rollback adımları içerir.	14

Kontrol	Bölüm
Restore'lar ve game day'ler prova edilmiştir ve aşağıdaki runbook'lar provanın gösterdiklerinden yazılmıştır.	14, 15
Yazılı bir hata modeli owner'ları, testleri ve review tarihlerini adlandırır.	3

Bir design review için sorular

Kontrol listesi var olan bir tasarımı doğrular. Bu sorular, henüz var olmayan bir tasarımı şekillendirmeye yardım eder. Onları bu sırayla sorardım, çünkü her cevap bir sonrakinin sınırlar.

- 1. Olgu nedir ve sahibi kim?** Event'i geçmiş zamanda adlandıramıyor ve tek bir sahip servis gösteremiyorsanız, tasarım bir broker için hazır değildir (Bölüm 11).
- 2. Bunun için bir event gerekiyor mu?** Adımlar yerel bir transaction'ı paylaşabiliyorsa ya da çağırının anında cevaba ihtiyacı varsa, senkron bir çağrı daha iyi araç olabilir (Bölüm 1).
- 3. Commit ile publish arasında ne olur?** Örüntüyü değil, çökme penceresini sorun. "Commit'ten hemen sonra publish ediyoruz", bir deploy'u bekleyen bir kayıptır (Bölüm 4).
- 4. Bu mesaj iki kez geldiğinde ne olur, geç geldiğinde ne olur?** Bunlar, farklı çözümleri olan farklı hatalardır (Bölüm 7 ve 9).
- 5. "Eventually"nin sınırı nedir?** Bekleyiş için bir süre, bir alert ve kullanıcıya dönük bir durum adlandırın (Bölüm 1).
- 6. Bu mesaj asla başarılı olamıyorsa ne olur?** Nerede bittiğini, kimin gördüğünü ve nasıl geri döndüğünü sorun (Bölüm 8).
- 7. Consumer ya da arkasındaki dependency producer'dan yavaşsa ne olur?** Neyin sınırlandığını, neyin duraklayacağını, hangi işin bekleyeceğini ve bu seçimin kime ait olduğunu sorun (Bölüm 3 ve 6).
- 8. Sonraki bir adım başarısız olursa, öncekileri ne geri alır?** Compensation'ı ve ardından hiçbir şeyin geri alınmadığı pivot'u adlandırın (Bölüm 10).
- 9. Bu contract'ı bir yıl sonra nasıl değiştireceğiz?** Kimse bunu düşünmediyse cevap bir incident sırasında doğaçlama üretilecektir (Bölüm 12).

10. **Bozuk olduğunu nasıl anlayacağız ve onarıldığını nasıl anlayacağız?** Tespit ve doğrulama ayrı sorulardır (Bölüm 13 ve 15).
11. **Bir store, broker ya da bir region kaybedilirse ne kadarını kaybederiz ve ne kadar sürede geri döneriz?** Her store için bir RPO ve bir RTO isteyin, bir de ardından gelen replay'i ya da restore'u (Bölüm 14).
12. **Kim page edilir ve ilk olarak ne yapar?** Cevap bir isimse bir owner'ınız var demektir. "Ekip" ise, olmayabilir.

Bunlardan bazılarının belirli bir günde temiz bir cevabı yoktur ve bu sorun değil. "Henüz bilmiyoruz ve nasıl öğreneceğimiz şu" demek, kimsenin kontrol etmediği kendinden emin bir cevaptan daha iyi bir sonuçtur.

Organizasyonel taraf

Bu kitaptaki her örüntünün bir insan yarısı vardır. Bir outbox, en eski satırının yaşını izleyen birine ihtiyaç duyar. Bir DLQ, onu okuyan birine ihtiyaç duyar. Bir schema registry, "compatible"ın ne demek olduğuna karar veren birine ihtiyaç duyar. İnsan yarısı eksik olduğunda örüntü diyagramda hâlâ vardır ve artık işini yapmaz.

Sahiplik

Üç tür sahipliği yazardım.

- **Event sahipliği.** Her event'in tek bir sahip servisi ve ekibi vardır; bu ekip event'in ne anlama geldiğini ve onu kimin tükettiğini dokümante eder (Bölüm 11).
- **Operasyonel sahiplik.** Her DLQ'nun, parking lot'un, reconciliation raporunun ve alert'in adı belli bir owner'ı vardır. Kimsenin sahip olmadığı bir alert bir bildirimdir ve bildirimler görmezden gelinir.
- **Servisler arası akışlar.** Parcelly sipariş akışı altı servisi geçer. Hiçbir tek servis sahip olmasa bile, bir ekibin akışa bir bütün olarak, timeout'ları ve takılan workflow'ları dahil, sahip olması gerekir (Bölüm 10).

Runbook'lar

Runbook, okuyucunun yorgun olduğunu ve sistemi altı aydır görmediğini varsayarak yazdığınız şeydir. Her alert için ne anlama geldiğini, önce nereye bakılacağını, hangi eylemlerin güvenli olduğunu, hangilerinin olmadığını ve ilk adımlar yardımcı olmazsa kimin aranacağını söyleyin.

Bazı runbook'lar bir alert'e bağlıdır, bazıları ise ekipte kimsenin yakın zamanda yapmadığı kadar nadir olan bir olaya. Bölüm 14 ikinci türü ekler. Recovery'yi prova etmeye dair bölümü, bir recovery runbook'unun cevaplama gereken sekiz soruyu sıralar; bunlar birinin kafasında taşınmaz, yazıya dökülür:

1. Hangi store etkilendi ve hangisi zamanda geri gitti?
2. Recovery point nedir ve nasıl belirlendi?
3. Hangi consumer group'lar, hangi timestamp'e ve hangi marjla reset edilmeli?
4. Replay sırasında hangi etkiler bastırılmalı ve hangi mekanizmayla?
5. Yeniden okunacak her aralık, etkilenen her consumer'ın dedup window'unun içinde mi?
6. Ne, hangi sırayla duraklatıldı ve kim devam ettirecek?
7. Hangi reconciliation'lar recovery'nin işe yaradığını doğrular ve onları kim okur?
8. İşler kötü gidiyorsa her şey nasıl geri alınır?

Tablo, runbook yazacağım olayları sıralar. Son sütun, her birinin sekiz sorudan hangilerini cevaplama beklediğimi işaretler. Bu bir başlangıç noktasıdır ve bir ekibin kendi sistemi kendi sorularını ekleyecektir.

Olay	Tehlikeli eylem	Sorular
Bir poison message'a takılan consumer (Bölüm 8)	Pozisyonu kaydı tutulmadan atlamak ya da mesaj geçene kadar retry limitini yükseltmek	6, 7, 8
Lag'i retention'a yaklaşan bir consumer (Bölüm 13 ve 14)	Mesajların okunmadan expire olmasına izin vermek ya da kimse karar vermeden ileri reset etmek	2, 3, 5, 7
Offset reset ve replay (Bölüm 14)	Yan etkiler canlıyken ya da dedup window'un ötesinde replay yapmak	3, 4, 5, 6, 8

Olay	Tehlikeli eylem	Sorular
Redrive (Bölüm 8)	Bütün bir DLQ'yu, hâlâ zorlanan bir dependency'ye bir anda geri göndermek	5, 6, 8
Onarım ve backfill (Bölüm 13)	Gözden geçirilmemiş bir betiği production durumuna karşı çalıştırmak	4, 7, 8
Bir veritabanını backup'tan restore etmek (Bölüm 14)	Offset'lerini reset etmeden consumer'ları devam ettirmek ya da geri sarılmış sequence'in yeniden verdiği identifier'ları yeniden kullanmak	1, 2, 3, 4, 5, 6, 7, 8
Broker veri kaybı ya da failover (Bölüm 14)	Grup'ları artık eskiden ifade ettikleri anlamı taşımayan pozisyonlardan devam ettirmek	1, 2, 3, 5, 6, 7
Region tahliyesi (Bölüm 14)	Bir standby'ı ne kadar geride olduğunu bilmeden promote etmek	1, 2, 3, 6, 7, 8
Broker migration cutover'ı (Bölüm 14)	Consumer'ları dual-publish edilmiş kopyalar deduplicate edilmeden ya da geri dönüş yolu olmadan taşımak	3, 5, 6, 8

Bunlardan ikisi özel özen hak eder. Redrive ilk sırada gelir, çünkü dikkatsiz bir tanesi sınırlanmış bir hatayı bir sele çevirir (Bölüm 8). Onarım ve backfill ikinci sırada gelir, çünkü türetilmiş durumu düzelten bir betik bir production değişikliğidir ve bir review'u, bir dry run'ı ve ne yaptığının kaydını hak eder (Bölüm 13).

On-call

Bir event-driven sistem için on-call, bir request-response servisi için on-call'dan önemli bir noktada farklıdır: hata çoğu zaman sessizdir. Senkron bir servis hata döndürür ve bir grafik kırmızıya döner. Bir event-driven sistem hiçbir şey döndürmeyebilir, çünkü mesaj bir queue'da bekliyordur. Yaş, lag ve işlenmemiş iş alert'leri birincil sinyaldir ve nöbetçi mühendisin, izin beklemeden queue'nun, DLQ'nun ve outbox'ın içine bakabilecek kadar erişime ihtiyacı vardır.

Not: Onarımı, ihtiyaç duymadan önce pratik edin. Bir test DLQ'sunu bir kez, gündüz, bir meslektaşın izlemesiyle redrive etmiş bir ekip, gerçeğini yalnızca runbook'u okumuş bir ekipten farklı ele alacaktır. Bölüm 14 aynı noktayı restore'lar ve game day'ler için yapar, Bölüm 15 ise tatbikatların nasıl tasarlanacağını ele alır.

Trade-off'lar ve maliyetler

Bir kontrol listesinin maliyetleri vardır ve ilki dikkattir. Her akışa uygulanan 43 maddelik bir liste çok uzundur ve hepsini düşük riskli bir bildirim akışına uygulayan bir ekip, sorunun kontrol listesi olduğu sonucuna varacaktır. Listeyi hatanın maliyetine göre ağırlıklandırırđım. Para hareket ettiren ya da stoęu deęiřtiren bir akış listenin tamamını hak eder. Bir hatırlatma e-postası idempotency'yi ve bir DLQ owner'ını, belki de başka pek bir şeyi hak etmez.

İkinci maliyet, sahte güvencedir. Tamamlanmış bir liste, birinin her maddeyi düşündüğünü söyler, sistemin doğru olduğunu deęil. Bölüm 13, 14 ve 15, düşünölmüş tasarımların yine de yazarlarının hayal etmedięi şekillerde başarısız olması nedeniyle var. Kontrol listesini son deęil, en ucuz savunma katmanı olarak ele alın.

Üçüncü maliyet, sürüklenmedir. Bir kez yazılan liste, ekibin bir anda neye inandığının kaydına dönüşür. Her incident'ten sonra yeniden gözden geçirin ve yerini hak etmeyi bırakmış maddeleri silin. İnsanların kullandığı kısa bir liste, insanların imzaladığı uzun bir listeden daha deęerlidir.

Ne zaman kullanılmayacağı: kontrol listesini muhakemenin yerine koymak için kullanmayın ve atılacak prototipler için talep etmeyin. Bir akış gelecek ay silinecekse, listenin tamamının maliyeti hatanın maliyetini aşar. Bunu açıkça söyleyin ve prototipin ya öleceęi ya da düzgün bir review alacağı tarihi yazın.

Sık yapılan hatalar

- **Kontrol listesini bir kapı olarak ele almak.** Bir kapı, işaretler sayılarak denetlenebilir ve cazibe de budur. Sonuç, doldurulmuş bir form ve tartışılmamış bir tasarımdır. Maddeleri soru olarak kullanın ve birini şaşırtan cevapları kaydedin.
- **Akışlar yerine örüntüleri kontrol etmek.** "Bir outbox'ımız var" ilerleme gibi hissettirir. Bu akışın onu kullanıp kullanmadığını sorun ve tabloyu ve transaction'ı adlandırın.
- **Her şeyi her yere uygulamak.** Kapsamlı hissettirir, sonra yorgunluęa ve terk etmeye yol açar. Listeyi hatanın maliyetine göre ağırlıklandırın ve hangi maddeleri neden atladığınızı söyleyin.

- **İnsan maddelerini sonraya bırakmak.** Owner'lar, runbook'lar ve review tarihleri koddan daha az acil görünür ve gece bir alert tetiklendiğinde ilk eksik olan şeydir. Onları design review'ının kendisinde atayın.
- **Cevapları asla yeniden gözden geçirmemek.** Bir yıl önce review'dan geçen bir akış, o zamandan beri consumer'lar, versiyonlar ve insanlar kazanmıştır. Cevaplara, Bölüm 3'ün hata modeli için önerdiği gibi, bir review tarihi koyun.
- **Listenin tam olduğuna inanmak.** Sisteminizin, bu kitabın adlandırmadığı hata modları vardır. Karşılaştığınızda onları kendi listenize ekleyin.

Temel çıkarımlar

- Güvenilirlik, bilinmeyen örüntülerden çok atlanan sorularla aşınır; bu yüzden her yeni akışta sorulan sabit bir soru kümesi, odadaki uzmanlıktan daha iyi korur.
- Kontrol listesindeki her madde kitabın üç sorusundan biridir: neyin başarısız olabileceği, bunu nasıl gözlemleyeceğiniz ve hangi garantinin ayakta kalacağı; bir birleşim yeri hakkında sorulur. Bu yüzden her madde genel olarak sistem için değil, belirli bir akış için cevaplanmalıdır.
- Listeyi hatanın maliyetine göre ağırlıklandırın: para ya da stok hareket ettiren akışlar her maddeyi hak eder, düşük riskli akışlar ise duplicate etkileri ve sessiz kaybı önleyen birkaç maddeyi.
- Bu kitaptaki her örüntünün bir insan yarısı vardır; bu yüzden owner'lar, runbook'lar, alert alıcıları ve review tarihleri kodla aynı design review'ına aittir.
- Event-driven hatalar sessizdir; bu yüzden on-call, yaş, lag ve işlenmemiş iş alert'lerine bağlıdır ve nadir olaylar, yani bir restore, bir replay, bir region kaybı ve bir broker migration, gerçekten çalıştırılmış runbook'lara ihtiyaç duyar.
- Tamamlanmış bir kontrol listesi birinin her maddeyi düşündüğünü gösterir, sistemin doğru olduğunu değil; reconciliation ve testin ek katmanlar olarak var olmasının nedeni budur.

- Güvenilirlik bir kez yayınladığınız bir özellik değil, küçük sorularla, yazılı cevaplarla ve ihtiyaç duyulmadan önce prova edilmiş onarımlarla sürdürdüğünüz bir disiplindir.

Parcelly, bir özelliğin bitmesi gibi asla güvenilir olmaz. Biri mesaj iki kez geldiğinde ne olduğunu her sorduğunda, cevabı yazdığında ve ona bir owner verdiğinde daha güvenilir olur. Hiçbir broker bunu sunmaz ve hiçbir kütüphane bunu birlikte göndermez. Bu, kurdukları her sistemin bir gün bir ağ, bir deploy ya da yorgun bir meslektaş tarafından yanlış kullanılacağını bilen ve o güne dramsız tasarım yapmaya karar vermiş insanların sürdürdüğü bir dikkat alışkanlığıdır. Bu kitaptan almanızı isteyeceğim şey bu: bir örüntü kümesi değil, bir sonraki soruyu sorma alışkanlığı.

Ek: Sözlük ve Örüntülere Hızlı Başvuru

Bu ek iki bölümden oluşuyor. Kısım A, kitabın kullandığı terimlerin alfabetik sırayla verildiği, her terimin en kapsamlı tanımlandığı ya da kullanıldığı bölümün belirtildiği bir sözlüktür. Kısım B, bir tasarım review'u sırasında göz gezdirebileceğiniz bir tablo olarak yazılmış, örüntülere dair bir hızlı başvuru.

İkisi de bölümlerin yerini tutmaz. Buradaki bir tanım bir hatırlatmadır ve bir tablo satırı bir işaretçidir. Gerekçe, maliyetler ve bir örüntünün kullanılmaması gereken durumlar son sütunda adı geçen bölümde yer alır.

A. Sözlük

acknowledgment (ack). Consumer'ın broker'a, bir mesajın işlendiğini ve queue'dan kaldırılabilceğini ya da consumer'ın konumunun ilerleyebileceğini bildirdiği sinyal. Olumsuz biçimi **nack**'tir. Bölüm 2.

aggregate. Bir partition key'in sahibi olan tutarlılık sınırı. Bir aggregate'in event'leri birbirine göre sıralıdır; farklı aggregate'lerin event'leri değildir. Bölüm 9.

aggregate version. Tek bir aggregate'teki değişikliklerin sayacı; outbox satırında saklanır ve payload'da taşınır, örneğin Orders için `order_version`. Bir aggregate'in event'lerini sıralar ve payload'ın şekli hakkında hiçbir şey söylemez. **Schema version** ve **registry revision**'dan farklı bir sayıdır. Bölüm 4 tanımlıyor; Bölüm 9 kullanıyor.

at-least-once. Bir mesajın bir ya da daha fazla kez teslim edildiği, dolayısıyla duplicate'lerin mümkün olduğu bir delivery semantiği. Mesaj hâlâ retain edilirken ve consumer yalnızca etkiden sonra ack'lerken geçerlidir. Retention'ı aşacak kadar lag'e düşen bir consumer yine de mesajları sessizce kaybeder. Kitabın varsayılanıdır. Bölüm 2 tanımlıyor; Bölüm 3 kaybı ele alıyor.

at-most-once. Bir mesajın sıfır ya da bir kez teslim edildiği, dolayısıyla bir mesajın kaybolabileceği ama asla iki kez teslim edilmediği bir delivery semantiği. Bölüm 2.

backfill. Çok sayıda event'i canlı bir consumer'a besleyen, küçük ve hız sınırlı batch'ler hâlinde çalıştırılan toplu onarım ya da yeniden inşa. Bölüm 13.

backpressure. Yükün sonlu kalması için her aşamadaki yürütülen işi sınırlamak. Broker aracılı bir sistemde, bir consumer'ın alımı üzerindeki prefetch ya da poll başına kayıt sayısı gibi limitler olarak ve broker'ın emdiği lag olarak görünür. Baskı bir producer'a yalnızca broker flow control'ü ya da açık throttling yoluyla ulaşır. Bölüm 3 bunu hata modeline yerleştiriyor; Bölüm 6 mekanizmaları veriyor.

backward compatibility. Yeni schema'yı kullanan bir consumer'ın eski schema ile yazılmış event'leri okuyabilmesi: yeni okuyucular, eski veri. Bölüm 12.

business key. Envelope'tan değil, işin anlamından türetilen bir tanımlayıcı; örneğin bir ödeme id'si artı bir işlem. Aynı işin iki farklı mesajla gelmesine karşı korur. Bölüm 7.

causation_id. Bu event'e doğrudan neden olan event'i ya da command'i işaret eden envelope alanı; `correlation_id` ise tüm akışı adlandırır. Bölüm 11.

change data capture (CDC). Bir veritabanının transaction log'unu okuyarak commit edilmiş değişiklikleri mesajlara dönüştürmek. Outbox bağlamında, CDC üzerine kurulu bir relay, outbox satırlarını tabloyu poll etmek yerine log'dan okur. Bölüm 5.

circuit breaker. Hatalar bir eşiği aştıktan sonra çağrıları durduran, ardından bir beklemenin sonunda deneme çağrılarının geçmesine izin veren, bir dependency'nin etrafındaki koruma. Bölüm 6.

command. `CapturePayment` gibi emir kipiyle adlandırılmış, bir şeyi yapma isteği. Bir command tek bir servise yöneliktir ve reddedilebilir. Bir event değildir. Bölüm 11.

compensation. stoğu serbest bırakmak ya da bir authorization'ı void etmek gibi, daha önceki bir saga adımını anlamsal olarak geri alan adım. Kitap buna rollback demez. Bölüm 10.

competing consumers. Her teslimatın bunlardan birine gitmesi için tek bir queue'yu paylaşan birkaç consumer. Yeniden teslim edilen bir mesaj başka bir consumer'a ulaşabilir. Consumer group'un queue ailesindeki karşılığıdır. Bölüm 2.

consumer. Bir topic ya da queue'dan mesaj okuyan ve bunlara göre hareket eden process. Bölüm 2.

consumer-driven contract. Bir consumer'a ait, bir event'ten okuduğu alanları ve türleri listeleyen ve producer'ın pipeline'ında kontrol edilen dosya. Bölüm 15.

consumer group. Bir topic'in partition'larını paylaşan bir consumer kümesi; her partition aynı anda bir üye tarafından okunur. Bölüm 2.

consumer lag. Bir consumer'ın bir partition ya da queue'nun başının ne kadar gerisinde olduğu; mesaj ya da süre olarak ölçülür. Bölüm 2 tanımlıyor; Bölüm 3 ve 13 kullanıyor.

consumer outbox. Takip eden bir mesaj publish eden bir consumer'ın, inbox satırı ve state değişikliğiyle aynı transaction'da yazdığı outbox. Commit'ten önceki bir çökme geride hiçbir şey bırakmaz, commit'ten sonraki bir çökmeyi ise redelivery'de inbox emer. Kuralı Bölüm 7 getiriyor; Bölüm 10 bunu saga adımlarına uyguluyor.

correlation_id. Tek bir iş akışındaki her event'in paylaştığı envelope alanı. Eşlikçisi `causation_id`, bunu doğrudan neden olan event ya da command'i işaret eder. Bölüm 11.

cutover. Bir broker migration'ında, bir consumer group'un yeni broker'dan okumaya başladığı adım; iki broker reconcile edildikten sonra, bir rollback planlanarak ve her seferinde bir group olmak üzere atılır. Bölüm 14.

dead letter. İşlenmesi başarısız olduğu için kenara alınmış mesaj. Niteleyici olarak terim tire ile yazılır: `dead-letter queue`. Bölüm 8.

dead-letter queue (DLQ). Bir consumer ya da broker'ın vazgeçtiği mesajları koyduğu ayrı bir hedef. Log-based bir broker'da aynı fikir bir **dead-letter topic**'tir. Bölüm 8.

deadline. Sonrasında bir sonucun artık faydalı olmadığı an. Retry'lar ondan önce durmalıdır. Bölüm 6.

dedup window. Bir consumer'ın işlenmiş id'leri hatırladığı dönem. Inbox satırı silinmiş bir mesaj yeniden işlenebilir. Bölüm 7.

deferred. Mesaj hakkında hiçbir şey söylemeyen bir hata için retry sınıfı; çünkü dependency'nin çöktüğü bilinmektedir, örneğin bir circuit breaker açıkken. Deferred bir mesaj hiçbir deneme tüketmez ve bu yüzden asla dead-letter'a alınmaz. Consumer duraklar ve mesaj bekler. Bölüm 6.

dual write. Bir veritabanı ve bir broker gibi iki sistemi ortak bir transaction olmadan güncellemek. Bölüm 4.

dual-publish. Bir migration sırasında aynı olguyu iki biçimde ya da iki hedefe publish etmek: bir schema değişikliğinde iki alan ya da iki topic (Bölüm 12), bir broker migration'ında iki broker (Bölüm 14). Bir broker migration'ında relay, yeni hedef için ikinci bir marker tutar; böylece eski broker eksiksiz kalır ve yenisi onunla reconcile edilebilir.

duplicate. Birden fazla kez işlenen ya da teslim edilen mesaj. Bölüm 2.

envelope. Payload'dan ayrı olarak, `event_id`, `event_type`, `schema_version`, `occurred_at`, `correlation_id` ve `causation_id` gibi, her event'le birlikte seyahat eden standart alan kümesi. Bölüm 11.

event. İçeriği halihazırda olmuş bir olgu olan, geçmiş zamanla adlandırılmış mesaj. Bölüm 11.

event_id. Tek bir event'i tanımlayan envelope alanı. Outbox satırı yazıldığında oluşturulur ve asla yeniden üretilmez; tüketim için olağan idempotency key'dir. Bölüm 4 ve 7.

event journal. Bir servisin, broker event'leri unuttuktan sonra da soruların cevaplanabilmesi için inbox'ının yanında tuttuğu, payload içermeyen, yalnızca eklemeli event envelope kaydı. Bölüm 13.

event log. Event'lerin durable, yalnızca eklemeli dizisi. Bölüm 2.

eventual consistency. Güncellemeler durursa replica'ların ya da servislerin, ne kadar süreceğine dair bir sınır olmadan, aynı state'e yakınsaması özelliği. Bölüm 1.

exactly-once. Kitapta yalnızca nitelenmiş bir etki olarak kullanılır: at-least-once delivery artı idempotent bir consumer; kitap buna effectively-once da der. Bir delivery garantisi değildir. Bir broker'ın exactly-once processing'i o broker'ın sınırı içinde gerçektir ve orada biter. Bölüm 2.

expand and contract. Tek bir kırıncı değişikliği birkaç uyumlu adıma çeviren bir migration örüntüsü: iki biçimi de destekle, her okuyucuyu migrate et, sonra eski biçimi kaldır. Bölüm 12.

failover. Aktif olan başarısız olduğunda bir broker partition'ının, bir veritabanının ya da bir cluster'ın başka bir kopyasına tasarlanmış geçiş. Henüz replicate edilmemiş olanı kaybedebilir. Bölüm 3 ve 14.

failure model. Bir sistemin birleşim yerlerinin ve her biri için neyin başarısız olabileceğinin, bunun nasıl gözlemleneceğinin ve hangi garantinin ayakta kalacağını; owner'lar, testler ve review tarihleriyle birlikte yazılı listesi. Bölüm 3.

failure semantics. Belirli bir birleşim yeri için neyin başarısız olabileceği, bunu nasıl gözlemleyeceğiniz ve hangi garantinin ayakta kalacağı. Kitabın tezidir. Önsöz bunu söylüyor, Bölüm 3 hata modeli olarak uyguluyor ve Bölüm 16 bunu review kontrol listesine dönüştürüyor.

fat event. Consumer'ların producer'a geri çağrı yapmadan hareket edebilmesi için yeterli state taşıyan event. Karşısı **thin event**'tir. Bölüm 11.

forward compatibility. Eski schema'yı kullanan bir consumer'ın yeni schema ile yazılmış event'leri okuyabilmesi: eski okuyucular, yeni veri. Bölüm 12.

grace period. Bir reconciliation işinin bir kaydı karşılaştırmadan önce kaydın ulaşması gereken yaş; böylece hâlâ yolda olan event'ler tutarsızlık olarak raporlanmaz. Bölüm 13.

head-of-line blocking. Bir consumer'ın atlamayacağı bir kaydın, aynı partition ya da stream'deki arkasındaki her kaydı bloke etmesi. Bölüm 8.

hold-back buffer. Bir consumer'ın, öncülü henüz gelmemiş bir event'i tuttuğu ve boşluk kapandığında ya da bir timeout dolduğunda serbest bıraktığı durable tablo. Bölüm 9.

idempotency key. Bir consumer ya da API'nin tekrarlanan işi tanımak için kullandığı key. Event tüketiminde normalde `event_id`'dir. Bölüm 7.

idempotent consumer. Aynı mesajı tekrar tekrar işlemesi, bir kez işlemesiyle aynı etkiyi yaratan consumer. Bölüm 7.

inbox. Consumer tarafında, handler'ın etkileriyle aynı transaction'da yazılan, işlenmiş mesaj id'lerinin tablosu. Bölüm 7.

integration event. Bilinçli bir public kontrat olarak publish edilen, bir domain event'ten eşlenmiş, kişisel veri ve secret içermeyen event. Bölüm 11.

invariant check. İş tarafının doğru kabul ettiği bir kural için zamanlanmış sorgu; örneğin her capture edilmiş ödemenin tam olarak bir Ledger kaydına sahip olması. Dönen her satır bir tutarsızlıktır. Bölüm 13.

jitter. Çok sayıda client'ın adım adım retry etmemesi için bir retry gecikmesine eklenen rastgelelik. "Jitter'lı exponential backoff" standart ifadedir. Bölüm 6.

link (trace). Teslimat geç, tekrarlı ya da batch'li olduğunda parent-child ilişkisi yerine kullanılan, bir consumer span'inden producer'ın trace context'ine verilen referans. Bölüm 13.

message. Bir broker üzerinden gönderilen veri birimi: bir event, bir command ya da bir doküman. Bölüm 11.

message age. Hâlâ bekleyen en eski mesajın üretilmesinden bu yana geçen süre; bir iş deadline'ı ve retention süresiyle karşılaştırılır. Bölüm 13.

message broker. Mesajları servisler arasında taşıyan altyapı. Kitapta iki aile görünür: log-based ve queue-based. Bölüm 2.

mirroring. Mesajları broker'lar arasında bir araçla kopyalamak: iki log arasında bir mirroring aracı ya da bir log ile bir queue arasında bir bridge veya connector. Bir migration öncesinden gelen geçmişi de taşır ve bir standby cluster'ı beslemeye devam edebilir; kopyadaki offset'ler kopyanın kendisine aittir. Bölüm 14.

offset. Bir consumer'ın bir event log'un bir partition'ındaki konumu. Bölüm 2.

offset commit. Log-based bir broker'da consumer'ın, bir pozisyondan önceki her şeyi işlediğine dair kaydı. Mesaj başına ack'in yerini alır. Bölüm 2.

offset reset. Bir consumer group'un commit edilmiş pozisyonunu, geçmişin yeniden okunması için, genellikle bir marjla bir timestamp'e doğru geriye taşımak. Offset'ler tek bir cluster'a aittir ve bir başkasına çevrilmez. Bölüm 14.

outbox. Durum değişikliğiyle aynı yerel transaction'da yazılan, bir relay'in event'leri publish ettiği tablo. Tam terim **transactional outbox**'tır. Bölüm 4.

parking lot. İnsan kararı gerektiren dead letter'lar için, DLQ'nun kendisinden ayrı bir bekletme yeri. Bölüm 8.

partition. Sıralı olan ve bir consumer group'un bir consumer'ı tarafından aynı anda okunan, log-based bir topic'in birimi. Bölüm 2 ve 9.

partition key. Event'leri bir partition'a yönlendirmek ve böylece aggregate başına sırayı korumak için kullanılan aggregate tanımlayıcısı. Bölüm 9.

pivot step. Bir saga'da, sonrasında iş akışının artık compensation yapmadığı ve yalnızca ileri gittiği nokta. Bölüm 10.

poison message. İşlendiği her seferde başarısız olan mesaj. Log-based bir broker'da process'i çöktüren bir mesaj, consumer'ı tek bir pozisyonda takılı bırakır; bu yüzden throughput'suz lag ve pozisyon başına bir sayaçla tespit edilir. Bölüm 8.

producer. Mesaj publish eden bir process ya da servis ve ürettiği event'lerin sahibi. Bölüm 11.

queue. Her mesajın tek tek tüketilip ack'lendiği bir broker yapısı. Bölüm 2.

rebalance. Üyeler bir consumer group'a katıldığında ya da ayrıldığında, ya da bir üyenin yanıt vermediğine hükmedildiğinde partition'ların yeniden atanması. İki consumer kısa bir süre aynı partition üzerinde çalışabilir. Bölüm 3.

reconciliation. İki sistemin gerçeklik görünümünü karşılaştırmak ve farkları onarmak. Bölüm 13.

recovery point objective (RPO). Ne kadar veri kaybetmeyi göze alabileceğiniz; süre olarak ölçülür. Altı saatte bir alınan backup, altı saate kadar bir RPO demektir ve her store'un kendi RPO'su vardır. Bölüm 14.

recovery time objective (RTO). Servissiz ne kadar kalabileceğiniz. Replay edilen bir projection için bir drain hesabıdır. Bölüm 14.

redelivery. Broker'ın bir mesajı, örneğin kaçırılmış bir ack ya da süresi dolmuş bir visibility timeout'tan sonra, yeniden teslim etmesi. Replay farklı bir şeydir. Bölüm 2.

redrive. Dead letter'ları, kendilerine takılan consumer'a, geldikleri paylaşılan topic'e asla değil, o consumer'ın kendi girdisine geri vermek. Key ve event_id korunur, deneme sayısı yeniden başlar ve redrive_count turu kaydeder. Bölüm 8.

registry revision. Bir schema registry'nin kayıtlı her schema'ya atadığı tanımlayıcı. Additive olanlar da dahil, kaydedilen her değişiklikte ilerler ve schema version ile aggregate version'dan farklı bir sayıdır. Bölüm 12.

relay. Outbox satırlarını broker'a taşıyan process. Bölüm 5.

replay. Geçmiş bir event log'dan bilerek yeniden okumak. Bölüm 2 tanımlıyor; Bölüm 7, 12, 13, 14 ve 15 kullanıyor.

retention. Bir broker'ın bir mesajı okunup okunmadığına bakmaksızın ne kadar süre tuttuğu: bir log'da zaman ya da boyut limiti, bir queue'da expiry. Replay'i, redrive'ı ve lag'e düşmüş bir consumer'ın ne kadar süre kapalı kalabileceğini sınırlar. Bölüm 2 tanımlıyor; Bölüm 3, 13 ve 14 kullanıyor.

retry budget. Retry'lara, çağrı başına değil, toplam trafiğin bir payı olarak konan sınır. Bölüm 6.

retry storm. Zorlanan bir dependency'ye yük ekleyen ve onu ayakta kalamaz hâlde tutan retry dalgası. Bölüm 6.

saga. Bir koordinatör tarafından orchestrate edilen ya da event'ler üzerinden choreograph edilen, compensation'larla birlikte yerel transaction'lardan oluşan uzun süren iş akışı. Bölüm 10.

schema evolution. Event'lerin biçimini zaman içinde, var olan okuyucuları ve yazıcıları bozmadan değiştirmek. Bölüm 12.

schema registry. Schema'ları saklayan, tanımlayıcı atayan ve önerilen bir schema'yı öncekilere karşı kontrol eden servis. Bölüm 12.

schema version. Envelope'taki integer `schema_version`; Bölüm 12 buna breaking version da diyor. Yalnızca eski bir okuyucunun kaldıramayacağı bir değişiklikte ilerler ve **registry revision** ile **aggregate version**'dan farklı bir sayıdır. Bölüm 12.

semantic lock. Verideki, bir saga'nın bir kayıt üzerinde çalıştığını söyleyen ve diğer kodun buna saygı gösterecek şekilde yazıldığı bir durum bayrağı. Bir veritabanı lock'ı değildir. Bölüm 10.

synthetic event. Bir probe tarafından canlı pipeline'a enjekte edilen ve akışın sonunda kontrol edilen etiketli trafik. Bölüm 15.

tolerant reader. Bir event'ten yalnızca ihtiyaç duyduğu alanları alan ve geri kalanı yok sayan consumer. Bölüm 12.

tombstone. Bir consumer'ın, bir aggregate'in satırını sildikten sonra tuttuğu key, son version ve deleted marker'ı; böylece geç gelen bir event satırı yeniden yaratamaz. Bölüm 9.

topic. Her iki broker ailesinde de adı olan bir mesaj akışı. Bölüm 2.

trace context. Bir iş birimini dağıttık bir trace içine yerleştiren trace id ve span id; outbox satırına yakalanır ve mesaj header'larında gönderilir, böylece consumer'ın span'i producer'ınkine bağlanabilir. Bölüm 13.

upcaster. Eski bir schema altında yazılmış bir event'i, iş mantığı onu görmeden önce güncel olana dönüştüren kod. Bölüm 12.

version check. Bir consumer'ın, eski event'leri yok saymak ve boşlukları tespit etmek için, bir event'teki version'ı o aggregate için uyguladığı son version ile karşılaştırması. Bölüm 9.

visibility timeout. Queue-based bir broker'da, teslim edilmiş bir mesajın ack'lenmemişse yeniden teslim edilmeden önce gizli kaldığı süre. Bölüm 2.

B. Örüntülere hızlı başvuru

Her satır, örüntünün ele aldığı hatayı kitabın terimleriyle ve size neye mal olduğunu söyler. Bir örüntü bir hatayı ele alabileceğiniz bir yere taşır; hatayı ortadan kaldırmaz. Bölüm 16 bu satırları bir review kontrol listesine dönüştürüyor.

Örüntü	Ele alınan hata	Maliyeti	Bölüm
Transactional outbox	Bir veritabanı ile bir broker arasındaki dual write	Ek tablo, çalıştırılacak bir relay, consumer'lara itilen duplicate'ler	4
Polling relay	Outbox satırlarını az altyapıyla broker'a taşımak	Polling gecikmesi, veritabanı üzerinde yük, özenli cursor tasarımı	5
Change data capture	Outbox satırlarını veritabanı log'undan düşük gecikmeyle publish etmek	Operasyonel ağırlık, connector failover'ı, yönetilecek log retention'ı	5
Jitter'lı exponential backoff	Toparlanan bir dependency'yi aşırı yükleyen retry'lar	Eklenen latency, gecikmelerin ve deneme sınırlarının ayarlanması	6
Retry budget	Bir kesintiye büyüten retry'lar	Başarılı olabilecek bazı çağrılar hızlıca başarısız olur	6
Backpressure ve flow control	Emebileceğinden fazla iş beslenen bir dependency ya da consumer	Lag, duraklatılan consumer'lar ve reddedilen iş; neyin bekleyeceğine dair bir iş kararı	6
Circuit breaker	Başarısız bir dependency'ye tekrarlanan çağrılar	Ayarlanacak state; breaker açıkken reddedilen çağrılar	6
Retry topic ya da delay queue	Handler içinde beklenemeyecek kadar uzun retry gecikmeleri	Daha fazla topic, daha fazla hareketli parça, daha zor ordering	6
Inbox'lı idempotent consumer	At-least-once delivery'den gelen duplicate'ler	Her mesajda tablo yazımları, yönetilecek retention	7
Business key	Aynı işin farklı mesajlarla gelmesi	İşlem başına key tasarımı ve unique constraint'ler	7
Downstream'e geçirilen idempotency key	Tekrarlanan harici yan etkiler	Downstream servisten destek gerektirir	7
Dead-letter queue	Redelivery döngüsünde sıkışan poison message'lar	Sahibi, alarmları ve onarım yolu olması gereken bir yığın	8
Parking lot	İnsan kararı gerektiren dead letter'lar	Manuel iş ve net bir sahiplik kuralı	8

Örüntü	Ele alınan hata	Maliyeti	Bölüm
Redrive	Neden düzeltildikten sonra dead letter'ları kurtarmak	Duplicate'ler, reordering ve ana akış üzerinde yük	8
Consumer outbox	Durumu değiştirip takip eden bir mesaj publish eden ve ikisinden birini kaybeden bir consumer	Consumer tarafında bir outbox ve çalıştırılacak bir relay	7, 10
Aggregate başına partition key	Tek bir aggregate içinde sırası bozuk event'ler	Hot partition'lar, paralellik üzerinde sınırlar	9
Consumer'da version check	Eski ya da sırası bozuk event'lerin daha yeni state'in üzerine yazması	Bir version alanı ve her boşluk için bir karar	9
Hold-back buffer	Sırayla uygulanması gereken bir dizideki boşluklar	Bellek, timeout'lar ve hiç kapanmayan boşluk	9
Saga (orchestrated ya da choreographed)	Tek bir transaction olmadan servislere yayılan bir iş akışı	Tasarlanıp işletilecek state'ler, timeout'lar ve compensation'lar	10
Compensation	Sonraki bir hatadan sonra tamamlanmış saga adımlarını geri almak	Bazı adımlar geri alınamaz; pivot step sınırı işaretler	10
Semantic lock	Bir saga'nın üzerinde çalıştığı bir kayıta eşzamanlı değişiklikler	Saga çalışırken bloke edilen işlemler	10
Diller arasında alan başına sabitlenen wire type'lar	Aynı envelope alanının iki runtime tarafından farklı okunması	Her dilde disiplin ve diller arası testler	11
Standart event envelope	Araçların ve consumer'ların izleyemediği ya da yönlendiremediği event'ler	Her producer genelinde disiplin	11
Thin ya da fat event	Geri çağrılar ile taşınan state arasında seçim yapmak	Thin: okuma anında coupling. Fat: daha büyük kontrat ve kopyalanan veri	11
Compatibility kontrollü schema registry	Producer'ların consumer'ları sessizce bozması	Çalıştırılacak bir servis ve izlenecek bir süreç	12
Tolerant reader ve upcaster	Okuyucuların yeni alanlarda ya da eski sürümlerde bozulması	Her geçmiş sürüm için tutulan dönüşüm kodu	12

Örüntü	Ele alınan hata	Maliyeti	Bölüm
Expand and contract	Kaçınılamayan kırıcı bir değişiklik	Birkaç deploy, dual-publish, daha uzun bir migration	12
Trace context ve correlation id	Asenkron hop'lar boyunca bir akışın hikâyesini kaybetmek	Her producer ve consumer'da yayılım	13
Reconciliation ve onarım	Bir kaynak sistem ile türetilmiş state arasında sessiz ıraksama	Karşılaştırma işleri, onarım araçları ve sonuçların review'u	13
Event journal	Broker'ın unuttuğu event'lere dair sorular	Her mesajda yazılan bir tablo ve yönetilecek retention	13
Backfill	Mevcut veriden yeniden inşa edilmesi ya da doldurulması gereken türetilmiş state	Hız sınırlı batch'ler ve bir pause switch'i, ve canlı bir consumer üzerindeki yük	13
Invariant check'ler	Hiçbir teknik metric'in göstermediği iş düzeyinde bozulma	Invariant'ları tanımlamak ve alert'lerin sahibi olmak	13
Offset-reset replay	Yanlış çalışmış bir consumer ya da zamanda geri gitmiş bir store; geçmişe yeniden ihtiyaç var	Aralığı kapsayan bir dedup window, bastırılmış yan etkiler ve sınır olarak retention	14
Store başına recovery point ve time objective'leri	Üzerinde anlaşılmış kayıp ya da kesinti olmadan, doğaçlamayla ele alınan felaketler	Her store için ölçülecek ve prova edilecek bir sayı	14
Recovery için cross-cluster replication	Bir broker cluster'ının ya da bir region'ın kaybı	Çalıştırılacak bir standby, RPO olan bir replication lag'i ve çevrilmeyen offset'ler	14
Dual-publish ve cutover	Event kaybetmeden ya da etkileri duplicate etmeden başka bir broker'a geçmek	Outbox'ta ikinci bir marker, reconciliation ve daha yavaş bir migration	14
Consumer-driven contract test'ler	Ekipler arasında kontrat kayması	Test bakımı ve bunları kimin çalıştıracağı üzerinde anlaşma	15
Handler ve perturbation test'leri	Duplicate, gecikme ya da reordering altında hiç sınanmamış handler'lar	Gerçek veritabanlı test altyapısı ve daha yavaş test paketleri	15

Örüntü	Ele alınan hata	Maliyeti	Bölüm
Fault injection	Çalışan bir sistemde hiç sınanmamış hata modeli girdileri	Ayrılmış bir ortam, sınırlanacak bir etki alanı	15
Replay tabanlı regression test	Gerçek geçmiş üzerinde davranışı bozan değişiklikler	Bakımı yapılacak, seçilmiş, anonimleştirilmiş örnek veri	15
Production'da synthetic event'ler	Canlı yoldaki sessiz hatalar	Gerçek iş verisinden yalıtılması gereken test verisi	15

Tabloyu okumak

Birçok satır çiftler hâlinde gelir: bir hatayı düzelten bir örüntü ve ilkinin getirdiği hatayı düzelten ikinci bir örüntü. Outbox duplicate'ler yaratır, bu yüzden consumer'ın bir inbox'a ihtiyacı vardır. Retry'lar yük yaratır, bu yüzden bir budget'a ihtiyaç duyarlar. Bir dead-letter queue bir yığın yaratır, bu yüzden bir sahibe ve bir redrive prosedürüne ihtiyacı vardır.

Bu zincir kitabın biçimidir. Bir satırı benimseyip eşini atlarsanız hata yer değiştirmiş olur, yok olmaz. Bence tablo en çok sağdan sola okunduğunda faydalıdır: sisteminizin bugün sahip olduğu hatadan başlayın, onu adlandıran satırı bulun ve bölümü okumadan önce maliyet sütununu okuyun.

KİTAP HAKKINDA

Güvenilir event-driven sistemler message broker'ların değil, failure semantics'in etrafında kurulur.

Kitap, bir bileşenin işi başka bir bileşene devrettiği her birleşim yeri için aynı üç soruyu sorar: ne başarısız olabilir, bunu nasıl gözlemlersiniz ve hangi garanti ayakta kalır. At-least-once delivery bir kusur değil, bir gerçek olarak ele alınır; bu yüzden handler'lar idempotent olur, duplicate'ler, kaybolan mesajlar, retry'lar ve sırasız gelişler umutla değil tasarımla karşılanır. Kitap Parcelly üzerinden transactional outbox, relay'ler ve change data capture, retry, idempotency, dead letter, ordering, saga, schema evolution, observability ve recovery konularını, her birinin bedeliyle birlikte işler.

YAZAR HAKKINDA

Muhammet Şafak 2008'den bu yana yazılım geliştiriyor. Uzun yıllar boyunca sağlık, finans ve telekomünikasyon domainlerinde aktif görev aldı. Teknolojiye değil, probleme odaklanarak geliştirme yapılması gerektiğini savunuyor. 2024'ten bu yana yüksek trafikli sistemleri uçtan uca ölçeklenebilir biçimde tasarlayıp geliştiriyor.

Küçük ve orta ölçekli yazılım geliştirme ekiplerine mentorluk yapıyor. Açık kaynak projeler yürütüyor. Kararlarını gerekçesi ve bedeliyle yazıyor.

Güvenilir sistemler güvenilir parçalardan kurulmaz. Bozulan parçaların etrafına kurulur.

I

ERİŞİM

II

EVENT'LER

III

AGENT'LAR

