Engineering Exactly-Once Semantics

Eliminating SEV-1 Inconsistencies via Transactional Kafka and Fencing

CORE STACK: Java / Spring Boot | Kafka Transactions | Redis | PostgreSQL | Micrometer

Resume Bullet Anchored:
Stabilized Java Spring Boot services to prevent 12 SEV-1 outages via transactional Kafka and idempotent sink processing.

OVERVIEW

In large scale supply chain systems, a single duplicate event is not just a logging error; it is a financial risk. At Centiro, our core shipment processing engine suffered from intermittent double processing during consumer group rebalances and network partitions. These incidents were classified as SEV-1 because they triggered duplicate carrier bookings and incorrect billing. I drove the technical implementation of the transition from at-least-once delivery to Exactly-Once Semantics (EOS) using Kafka Transactions and idempotent sinks, effectively eliminating the primary failure mode associated with these outages.

CONTEXT AND TECHNICAL CONSTRAINTS

We operated a fleet of Java Spring Boot microservices handling high-throughput event streams (p95 latency under 200ms). The legacy system relied on standard Kafka retry logic. If a consumer processed a message but crashed before committing the offset, the re-assigned consumer would process the same message again.

Technical hurdles included:

1. **Non-Idempotent Downstream APIs:** Many carrier webhooks did not support idempotency keys, making duplicate prevention at the source mandatory.
2. **Zombie Producers:** Ghost writes from older producer instances during network blips caused split-brain scenarios in our data store.
3. **Throughput Requirements:** The solution had to maintain a steady-state throughput of approx. 18k events per second while absorbing the transactional commit overhead.

SYSTEM ARCHITECTURE

I implemented a robust Check-then-Act pattern powered by Kafka Transaction Coordinator to enforce atomicity across multiple topics and state stores.

```
[ Upstream Service ]
        |
        | 1. Init Transaction (Initializes TransactionalID mapping)
        V
[ Kafka Transaction Coordinator ] <--- Fences older Epochs
        |
        | 2. Atomic Write (Offsets + Data)
        V
[ Consumer / Sink Service ] (Isolation Level: read_committed)
        |
        | 3. Distributed Deduplication (Redis SETNX)
        | 4. Database Transaction (PostgreSQL)
        V
[ External Carrier Gateway ]
```

THE PROBLEM: DISTRIBUTED INCONSISTENCY

The primary failure mode was the Zombie Producer. During long Garbage Collection (GC) pauses, a producer would be marked dead by the cluster, and its partitions would be reassigned. When the original producer resumed, it would attempt to complete its stale write. Without transactional fencing, this resulted in out-of-order events and duplicate shipments.

Symptoms included:

1. Duplicate shipment records in the audit log (approx. 0.5 percent of total volume).
2. Race conditions between the event consumer and the billing aggregator.
3. Frequent manual database patches by SREs to reconcile carrier requests.

THE SOLUTION: KAFKA EOS AND FENCING

I evaluated Two-Phase Commit (2PC) but rejected it due to its inability to handle coordinator failure without blocking. Instead, I opted for Kafka native EOS.

1. **Epoch-Based Fencing:** By assigning a unique transactional.id to each producer, the Kafka coordinator automatically increments the producer epoch. Any older producer attempting a write is rejected with a ProducerFencedException, effectively fencing the stale producer.
2. **Isolation Levels:** I updated all downstream consumers to read_committed. This ensured that uncommitted or aborted messages due to producer failure were never visible to the business logic.
3. **Idempotent Sink Filter:** I authored the technical design for a middleware layer that computed a SHA-256 hash of the payload and stored it in Redis with a 24-hour TTL. This provided a secondary defense for rare edge cases where Kafka might deliver a duplicate within a single transaction window.

VALIDATION AND RESILIENCE TESTING

We simulated production failures using partition-kill scripts and network delay injection. Using Micrometer, we tracked the kafka.producer.transaction.error.rate and verified that no duplicates were persisted to the database during simulated failure injections. Load testing showed a 4 to 6 percent increase in end-to-end latency, which was an acceptable tradeoff for total data consistency.

RESULTS AND SLIS

1. **Outage Prevention:** Achieved a 100 percent reduction in consistency-related SEV-1 incidents over a 12-month period compared to historical trends.
2. **Data Accuracy:** Achieved 100 percent parity in message-to-sink validation tests across the primary event stream.
3. **Engineering Efficiency:** Reduced manual data reconciliation effort by approx. 15 hours per week during peak shipping cycles.

MISTAKES AND COURSE CORRECTIONS

1. **Transaction Timeout Tuning:** Initially, default timeouts (60s) were too short for our largest batch re-indexes. I increased this to 120s after observing transaction timeouts and fencing errors during peak load.
2. **Redis Key Collision:** Early versions of the hash filter did not include the event-type in the key. This caused false positives for different events with similar payloads. I corrected this by using a structured key: idmp:event_type:payload_hash.
3. **Rebalance Storms:** We found that frequent rebalances were aborting transactions prematurely. We tuned max.poll.interval.ms to give our logic more headroom during heavy processing.

KEY LEARNINGS

1. **Explicit over Implicit:** Never assume at-least-once is good enough for financial or logistics data.
2. **Fencing is Crucial:** Distributed systems must have a mechanism to fence stale participants to prevent state corruption.
3. **Monitoring the Internal State:** Observing Kafka internal metrics like producer-retry-rate is as important as observing business KPIs.

CONFIDENTIALITY NOTE

Specific carrier names and internal service identifiers have been omitted. Metrics and volumes are based on sanitized production data to protect Centiro Solutions proprietary interests.