Building Systems Using Transactions in Apache Kafka®

How Kafka's transactions provide you with accurate, repeatable results from chains of many stream processors or microservices, connected via event streams.

Apache Kafka® ships with built-in transactions, in much the same way that most relational databases do. The implementation is quite different, as we will see, but the goal is similar: to ensure that our programs create predictable and repeatable results, even when things fail.

Transactions do three important things:

for example, `Order Confirmed` and `Decrease`
one of the two succeeded.

correct results (even something as simple as

ate stores are backed by Kafka topics. For
writing to the state store, writing to the
backing topic, writing the output, and committing offsets. Transactions enable this atomicity.

Here, we'll dive into transactions, looking at the problems they solve, how we should make use of them, and how they actually work under the covers.

Using the Transactions API for Multiple Topic Writing

As a simple example, imagine we have an account validation service. It takes deposits in, validates them, and then sends a new message back to Kafka marking the deposit as validated.

Kafka records the progress that each consumer makes by storing an offset in a special topic, called `__consumer_offsets`. To ensure that each deposit is committed exactly once, the final two actions (Figure 1)—(2a) send the "Deposit Validated" message back to Kafka, and (2b) commit the appropriate offset to the consumer_offsets topic—must occur as a single atomic unit. The code for this looks something like the following:

Copy

```
//Read and validate deposits validated
Deposits = validate(consumer.poll(0))

//Send validated deposits & commit offsets atomically
producer.beginTransaction()
producer.send(validatedDeposits)
producer.sendOffsetsToTransaction(offsets(consumer))
producer.endTransaction()
```

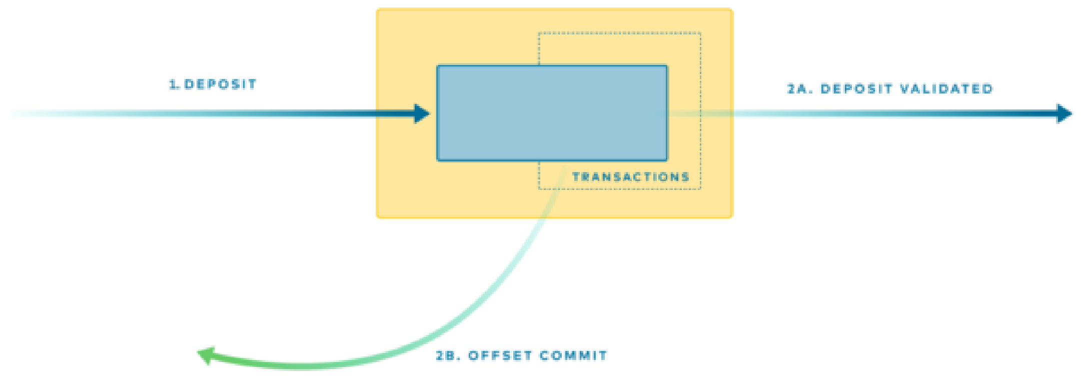

Figure 1. A single message operation is in fact two operations: a send and an acknowledge, which must be performed atomically to avoid duplication

If you are using the Kafka Streams API, no extra code is required. You simply enable the feature using a simple configuration: `processing.guarantee=exactly_once`

Two Opportunities for Duplicates

Duplicate events are commonly introduced to topics due to failures when interacting with the Brokers. For instance, a producer may fail to receive an acknowledgement of a successful write (say due to network errors), and so introduce a duplicate upon retrying the write. Similarly, a consumer may fail to increment its consumer offset due to similar failure modes, resulting in duplicate consumption of the same events (Figure 2).

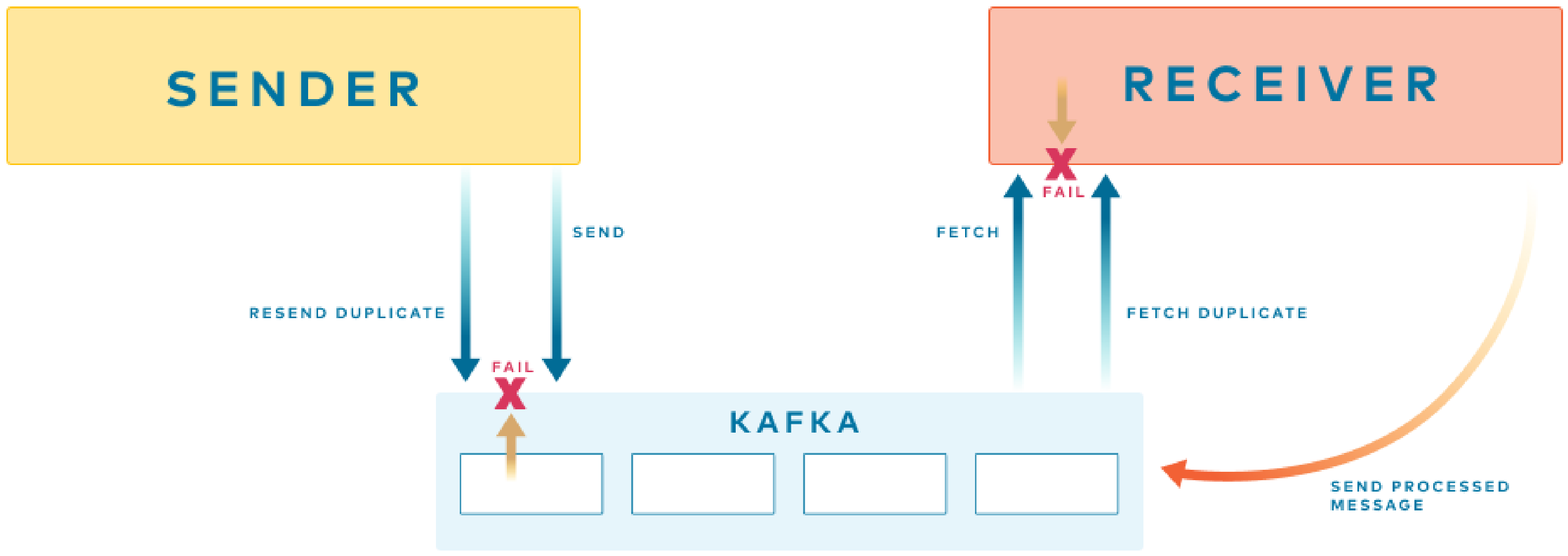

Figure 2. Message brokers provide two opportunities for failure—one when sending to the broker, and one when reading from it

Transactions address both of these issues. Each producer is given a unique identifier, and each message is given a sequence number. The combination of the two uniquely defines each batch of messages sent. The broker uses this unique sequence number to work out if a message is already in the log and discards it if it is. This allows the Kafka Broker to fence out duplicate writes without having to store a unique ID for every event it has previously seen.

On the read side, we might simply deduplicate (e.g., in a database). But Kafka's transactions actually provide a broader guarantee, more akin to transactions in a database, tying all messages sent together in a single atomic commit. So idempotence is built into the broker, and then an atomic commit is layered on top.

How Kafka's Transactions Work Under the Covers

Looking at the code example in the previous section, you might notice that Kafka's transactions implementation looks a lot like transactions in a database. You start a transaction, write messages to Kafka, then commit or abort. But the whole model is actually pretty different because of course it's designed for streaming. One key difference is the use of marker messages that make their way through the various streams. Marker messages are an idea first introduced by Chandy and Lamport almost 30 years ago in a method called the Snapshot Marker Model. Kafka's transactions are an adaptation of this idea, albeit with a subtly different goal.

While this approach to transactional messaging is complex to implement, conceptually it's quite easy to understand (Figure 3). Take our previous example, where two messages were atomically written to two different topics atomically. One message goes to the Deposits topic, the other to the `__consumer_offsets` topic.

First, begin markers are sent to each topic. Next, the messages are sent. Finally, once all the messages of the transaction have been written, we flush each topic with a Commit (or Abort) marker, concluding the transaction.

The purpose of a transaction is to ensure that only "committed" data is seen by downstream programs. To make this work, when a consumer sees a Begin marker it starts buffering internally, preventing processing until a Commit marker arrives. Then, and only then, are the messages released to the consuming program.

Note, in practice two clever optimizations are made to improve the efficiency of this process:

1. Buffering is moved from the consumer to the broker, reducing memory pressure.
2. Begin markers are also optimized out i.e. the Commit/Abort marker implies a Begin marker for the next transaction.

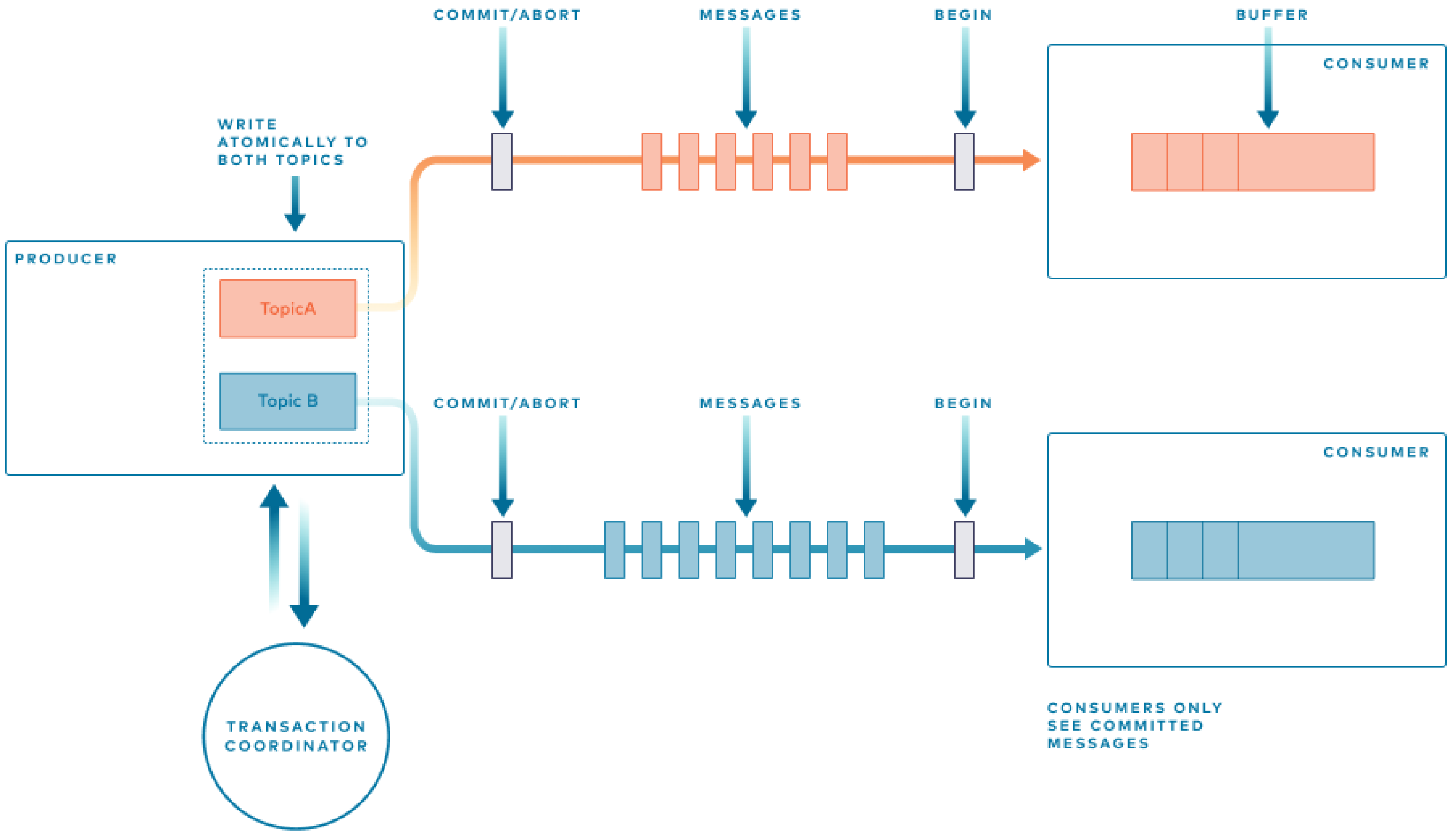

Figure 3. Conceptual model of transactions in Kafka

Sending the Commit markers involves the use of a transaction coordinator, which is a dedicated service running on each Kafka Broker. This is necessary to ensure that Commit markers are atomically appended to for each participating message batch, regardless of topic, partition, and broker locality.

The transaction coordinator is the ultimate arbiter that finalizes a transaction, and it maintains a durable transaction log, also stored in Kafka, to back this up (this step implements two-phase commit).

There is of course an overhead that comes with this feature, and if you were required to commit after every message, the degradation of performance would be noticeable. In practice, however, the overhead is minimal as it is averaged across all the messages in a batch, allowing us to balance transactional overhead with worst-case latency. For example, batches that commit every 100 ms, with a 1 KB message size, have a 3% overhead when compared to in-order, at-least-once delivery. You can test this out yourself with the performance test scripts that ship with Kafka.

There are many other subtle details to this implementation, particularly around recovering from failure, fencing zombie processes, and correctly allocating IDs, but what we have covered here is enough to provide a high-level understanding of how this feature works. For a comprehensive explanation of how transactions work, see the post "Transactions in Apache Kafka" by Apurva Mehta and Jason Gustafson.

The Killer Feature: Multi-Operation-Commits

Kafka can be used to store data in the log, with the most common means being a state store (a disk-resident hash table, held inside the API, and backed by a Kafka topic) in Kafka Streams. As a state store gets its durability from a Kafka topic, we can use transactions to tie writes to the state store and writes to other output topics together. This turns out to be an extremely powerful pattern because it mimics the tying of messaging and databases together atomically, something that traditionally required painfully slow protocols like XA.

The database used by Kafka Streams is a state store. Because state stores are backed by Kafka topics, transactions let us tie messages we send and state we save in state stores together, atomically.

Imagine we extend the previous deposits example so that our validation service keeps track of the balance as money is deposited. Given that the initial balance is $50, and we deposit $5, then the balance should go to $55. Our service records that $5 was deposited, but it also stores the updated balance ($55) by writing it to the log-backed state-store. See Figure 4.

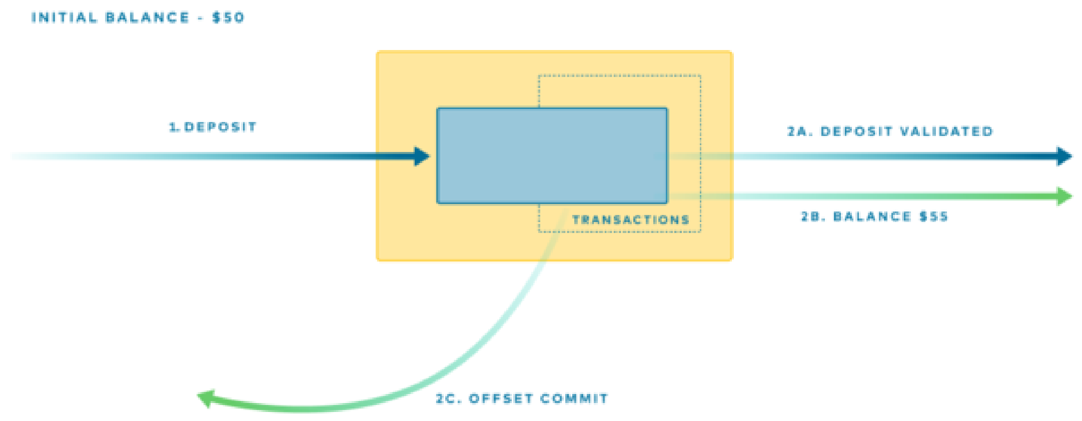

Figure 4. Three messages are sent atomically: a deposit, a balance update, and the acknowledgment

If transactions are enabled in Kafka Streams, all these operations will be wrapped in a transaction automatically, ensuring the balance will always be atomically in sync with deposits. You can achieve the same process with the product and consumer by wrapping the calls manually in your code, and the current account balance can be reread on startup.

What's powerful about this example is that it blends concepts of both messaging and state management. We listen to events, act, and create new events, but we also manage state, the current balance, in Kafka—all wrapped in the same transaction.

What Can't Transactions Do?

The main restriction with Transactions is they only work in situations where both the input comes from Kafka and the output is written to a Kafka topic. If you are calling an external service (e.g., via HTTP), updating a database, writing to stdout, or anything other than writing to and from the Kafka broker, transactional guarantees won't apply and calls can be duplicated. This is exactly how a transactional database works: the transaction works only within the confines of the database, but because Kafka is often used to link systems it can be a cause of confusion. Put another way, Kafka's transactions are not inter-system transactions such as those provided by technologies that implement XA.

Summary

Transactions affect the way we build applications in a number of specific ways:

- You no longer need to worry about duplicates or making your applications idempotent. This means long-chained data pipelines or stream processing applications can be built without duplicates being a concern.
- You can publish multiple messages and state updates in the same transaction, provided they all go to their own Kafka topics.

Kafka's transactions free you from the worries of failure and retries in a distributed world—worries that really should be a concern of the event broker, not of your code. This raises the level of abstraction, making it easier to get accurate, repeatable results from large estates of fine-grained services.