


Streaming transaction optimization in distributed environments

The demand for stream processing applications with stronger semantics, lower latency, and more reliability has grown in recent years. Many big data entities, such as financial institutions and cryptocurrency companies, use distributed platforms for streaming processing applications to process debits and credits on user accounts. There is no room for errors in these situations — it's essential that every transaction is completed exactly once, with no exceptions.

Unlike centralized systems, distributed environments can scale their reach and reduce the risks of having a single point of failure. However, while seemingly fault-tolerant, some distributed environments have limitations that cause them to be slower and more problematic.

Recent advancements such as  are changing the streaming transaction landscape, paving the way for a faster, more reliable, efficient, and optimized path for streaming data systems.

Distributed environments

Open-source distributed event streaming platforms are used for high-performance data pipelines, streaming analytics, and mission-critical applications. Kafka has emerged as the go-to application for streaming workloads within the manufacturing, banking, streaming, and telecom industries around the world, where speed, accuracy, and security are critical. Many distributed systems, like Kafka, operate on a **two-phase commit protocol (2PC)**.

Kafka is well-known for its transactional capabilities. Within this distributed framework, new data enters from an external source, which is an **input topic**. The **application** (the API platform) consumes data from that topic, transforms the data, and then pushes it to an **output topic** (the database).

The **offset topic** oversees where the transaction is at from an input topic standpoint. The offset topic tells all the applications that are consuming from this topic where they're at with the consumption of data from the input topic.

There are multiple possible transaction delivery guarantees, including at-most-once, at-least-once, and **exactly-once delivery**. Exactly-once means each message is delivered once and only once. When working with transactions, it's important that the workload is carried out in an exactly-once, all-or-nothing semantic.

Common issues and limitations in a distributed framework

Two key issues persist in a distributed framework:

Slow data replication

Distributed systems don't share a unified memory, so it's possible for messages and data to get lost in the network while moving from one machine to another. Also, topics are always replicated by a factor of three to prevent data loss, but these typical distributed systems yield delays.

Application crashes

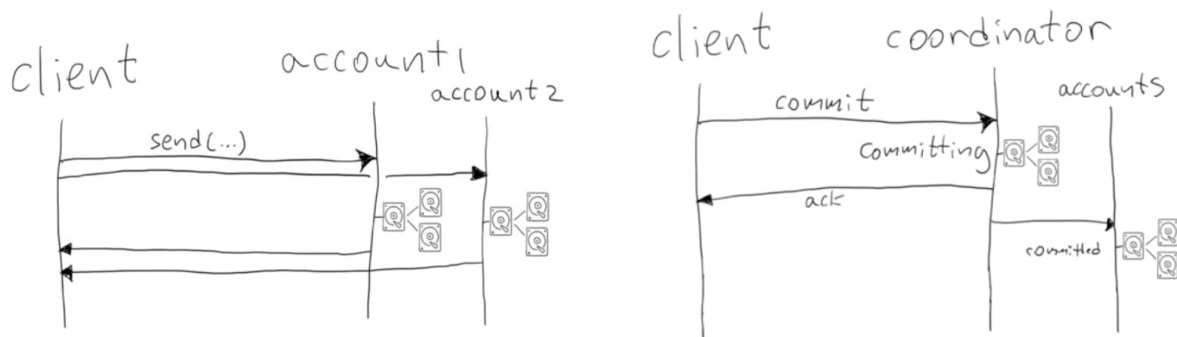
In distributed environments, applications may crash or temporarily lose connectivity to the rest of the system due to network disconnectivity or overloading (e.g., if all the nodes of the distributed system try to send data at once). When the application comes back up after crashing, a review of the offset will take place to identify the most recent updated offset. If there was no update, the application will reprocess the most recent offset that was identified, creating a duplicate of the output topic — therefore violating the exactly-once processing semantics. When occurring in a banking system, the duplication would result in an error.

It's essential to have a system that can consistently, securely, and reliably perform an exactly-once transaction across the offset topic in the output topic, especially within high-stakes account and order transactions. These transactional commands need to either completely succeed or, in the case of an error, fail, roll back the data, and abort the transaction.

The distributed framework for transactions

Let's look at a distributed framework that uses a standard **2PC** through the example of transferring money from bank account 1 to bank account 2.

- A client (a machine that runs a front-end application) will register with a transactional coordinator to interact with accounts 1 and 2, which are considered two topics. The coordinator is responsible for managing account members and marks across those topics to indicate that a transaction has begun. Next, the client will be informed that we're ready to start the transaction(s).
- The system sends a command to update account 1, checking the balance to ensure there are funds to cover the transfer. Next, the send command moves money from account 1 to account 2. Both must be persisted across topics, replicating the data three times across the cluster.
- After this, a commit is sent to the coordinator, returning back to the client indicating the transaction is in a committing state. The accounts are updated and finally fully committed.



Optimizing data streaming transactions with modern engines

This type of distributed framework requires a lot of back and forth with the client, and a lot of times we'll need to touch the disk to actually complete the transaction. It takes a lot of work and latency throughout the entire architecture to achieve exactly one transaction — but it doesn't have to.

The following approach uses **Redpanda**, an alternative Kafka-compatible tool that can control the information flow of how, when, and where things are stored, transferred, accessed, mutated, and eventually delivered.

You can think of a platform like **Redpanda** as an engine ledger for transactions that uses a thread-per-core architecture for delivering stable tail-latencies. **Redpanda** conducts a [parallel commit protocol](#) (versus 2PC), allowing the send command to validate that everything has been fully replicated during the *final* commit phase. **Redpanda** also ensures that its send command is **eventually consistent**, allowing data to be highly available.

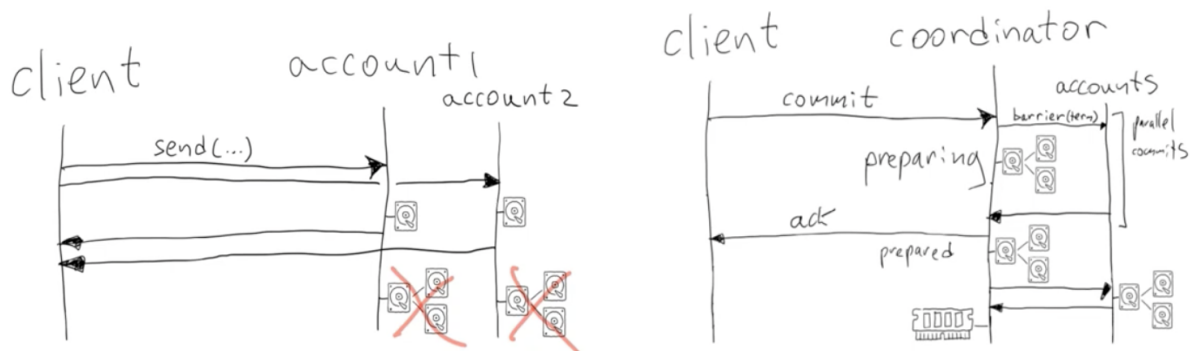
An optimized framework

Let's look at this updated framework using our previous example. With an optimized framework, the system would:

- Inform the coordinator about the two different topics (accounts 1 and 2) and check the **barrier term** (i.e., what the latest offset is within those topics so we can understand the state of those accounts). We store that data into the coordinator's memory (RAM versus disk) and return back to the client to communicate we're ready for the transaction.
- Send the update to the two accounts in a parallel fashion, meaning there is no need to wait for acknowledgment from the client. We're only persisting this on the ledger — we've reached a temporary state of the transaction that hasn't been fully replicated or

committed yet. This is the **eventually consistent** portion of this framework — we're verifying that the ledger has written this data to RAM.

- Note that eventual consistency is still prone to anomalies. To prevent these anomalies from getting through, we validate that all the send command data was fully replicated during the commit phase.
- Prepare the transaction for the commit phase. We'll start to fully write everything to RAM on the coordinator's side, while also ensuring that the term hasn't changed during the send portion.
 - If it has, we return the standard error to the client to let them know they need to redo the transaction because the state changed under them behind the scenes. This is only an issue if there are conflicts or multiple overlapping transactions.
- Check the barrier term exists and is within the barrier we've defined. We write that to RAM and inform the client that the commit has been completed. We then finalize the transaction from the coordinator's standpoint.




The benefits of adopting a modern engine


Kafka API is good — why not make it better? Implementing new advancements addresses gaps in typical distributed frameworks, allowing for far less disk interaction, less replication that has to be waited on and across the entire process, and less risk of duplications. There's room to optimize transactions, so you can do more with less.

- $O(N)$:
 - Replication protocol
 - Disk IO
 - Client-Cluster RTT
 - Intra-cluster RTT
- Add Topic to Txn
- Commit

- $O(N)$:
 - ~~Replication protocol~~
 - ~~Disk IO~~
 - ~~Client-Cluster RTT~~
 - ~~Intra-cluster RTT~~
- Add Topic to Txn
- Commit

A core element of these rising advancements is that they aim to address gaps of typical distributed environments:

- Idempotency: Idempotency is a crucial aspect of any distributed system. Kafka offers idempotent producer enablement, but it can guarantee the exactly-once delivery only for the messages sent within a single session. New advancements are offering **idempotent producer** features that can reduce and even eliminate the impact of duplicate messages received as a result of failed acknowledgments — resulting in more stable and reliable communication.
- Storing data as RAM: Direct memory access (DMA) is a method that allows an IO device to send or receive data directly to or from the main memory. This allows data to bypass the CPU and speed up memory operations.
- Security isn't compromised: Optimizing transaction protocols doesn't mean you have to compromise data security.  for example, leverages the Raft consensus algorithm for managing the replicated log, giving users a sound primitive for configuration and replication.

To learn more, watch the  where we explore how these new advancements can act as overarching reliability and optimization tools in a transactional framework.