KIP-27 - Conditional Publish

- Status
- Motivation
- Public Interfaces
- Proposed Changes
- Worked Example
- Compatibility, Deprecation, and Migration Plan
- Rejected Alternatives
    - Idempotent Producer or Transactional Messaging
    - Comparing Offsets by Key

Status

Current state: *"Under Discussion"*

Discussion thread: *here*

JIRA: *here*

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

When using the current Producer APIs, the resulting state of the log is hard to predict or control: it's possible for a producer to end up writing duplicate messages to Kafka, or have multiple competing tasks write to the same commit log and create an inconsistent state.

This document proposes adding a simple concurrency primitive to the Kafka producer. The producer is allowed to attach an 'expected offset' to each published message — and if that offset does not match the upcoming offset at the time the message is appended to the log, the broker refuses the publish request. Like the common check-and-set operation, this 'conditional publish' lets the client confirm the current state before going ahead with the write.

Again like check-and-set, this only makes good progress when contention for a single partition is low. Fortunately, when Kafka is used as a commit log or as a stream-processing transport, it's common to want a **single** producer for a given partition — and in many of these cases, the 'conditional publish' turns out to be quite useful.

- A producer can re-send a message indefinitely and be sure that at most one of those attempts will succeed; and if two producers accidentally write to the end of the partition at once, we can be certain that at least one of them will fail.
- It's possible to 'bulk load' Kafka this way, by appending messages consecutively to a partition. Even if the list is much larger than the buffer size or the producer has to be restarted, each message will be appended to the log just once.
- If a process is using Kafka as a commit log -- reading from a partition to bootstrap, then writing any updates to that same partition -- it can be sure that it's seen all of the messages in that partition at the moment it does its first (successful) write.

In all of these cases, the 'conditional publish' operation can support much stronger guarantees than the existing publish semantics.

Public Interfaces

The proposal requires a few additions to the public interface.

- Add a new `offset` field to the `ProduceRecord` class, with a default value of -1.
- Add a new `check.expected.offsets` property to the log config.
- Add a new error code and exception type for an offset mismatch.

Proposed Changes

The current produce request already includes an offset for every message, which the server currently ignores. This proposal repurposes that field as the expected offset of the message. This requires two main changes:

- Clients need an API to specify the expected offset of a message. This requires adding a new `offset` field to the existing produce record, and using the value of that field as the offset for that message in the outgoing message set. (Or the sigil value -1, if unspecified.)
- The server needs to be modified to compare the expected offset to the upcoming offset for the matching partition, and refuse the entire produce request if any of the offsets don't match. This involves a couple additional checks at offset-assignment time, and a new error code and exception type to signal the error to the client.

This feature also requires a new config parameter, which can be enabled globally or per-topic.

The JIRA ticket includes a rough-draft patch, which should give a sense of the scope of the change.

Worked Example

To see how this low-level coordination mechanism can be used by high-level code, we'll look at the design of a simple distributed key-value store.

Our store distributes data by partitioning the keyspace into multiple shards, such that every key belongs to a single shard. The database also maintains a 'leader' for each shard which is responsible for maintaining all of that shard's data. (This behaves like the Kafka consumer's rebalancer — normally we expect a single leader for each shard, but we might transiently have multiple 'leaders' during a partition or leadership transition.) The database uses a Kafka topic as a write-ahead log for all operations, with one partition per shard; the logged operations are used for both crash recovery and replication. We'll assume that the result of an operation (or even whether or not that operation is permitted at all) might depend on all the previous operations in that shard. As we append to the log, then, it's important that the operations are not duplicated, skipped, or reordered; if two 'leaders' try to append at once, it's also important that we don't interleave their operations in the log.

Followers and 'bootstrapping' leaders simply consume the partition, applying each operation to their local state and tracking the upcoming offset. Once a leader believes it has reached the end of the partition, it will start accepting new operations from clients and appending them to the partition. It batches multiple operations to Kafka in a single produce request, with a maximum of one in-flight request at a time. Each record is assigned an expected offset, starting with the leader's upcoming offset. If the produce request is successful: the leader can apply the operations to its persistent state, return successful responses to all the clients, set the new upcoming offset, and send the next batch of records to Kafka. If the produce request fails, maybe after a few retries: the leader should fail the corresponding requests and resume bootstrapping from what it had thought was the upcoming offset, perhaps checking to see if it is still the leader.

This append logic preserves all the correctness properties we required above. Suppose our leader is bootstrapped up to an upcoming offset of **k**, and publishes a new batch of operations to the log. If **k** doesn't match the actual upcoming offset in the partition (perhaps a new leader has been elected and appended first, or the broker returned stale metadata as in KAFKA-2334) the publish will have no effect. If the offsets do match, the server will try and append the messages to the log. Let's imagine the log contains messages [A,B,C], and the leader published messages [D,E,F]. The append might succeed, resulting in a log of [A,B,C,D,E,F]; it might fail, leaving the original log of [A,B,C]; or it might fail while replicating and keep only a prefix of the operations, leaving a log of [A,B,C,D]. In no case will we ever skip over messages in the batch, or change the order, so the result will always be a legal series of operations. Finally: since we wait to acknowledge updates until after they've been appended to the log, acknowledged operations should not be lost.

Compatibility, Deprecation, and Migration Plan

This proposal 'hides' the offset check behind a per-log configuration flag. Without any config changes, new clients will work with the existing version of the server, and existing clients will work with the new server.

If the new configuration parameter is enabled, a user will also need to use the updated client — existing clients put arbitrary values in the offset field, so those produce requests might fail. Since the only reason to enable the config is to manually specify the offsets, which requires an up-to-date client anyways, this does not seem to be a serious limitation.

In a future major version of Kafka, it may make sense to have all clients default the offset to -1 and leave the offset check on for all partitions. However, this would be a breaking change.

Rejected Alternatives

Idempotent Producer or Transactional Messaging

A couple of designs or proposals exist for adding transactional or idempotency features to Kafka: Idempotent Producer and Transactional Messaging in Kafka. These proposals, if implemented, would likely overlap with the use-cases for this feature — for example, one can imagine using transactional messaging to implement a more reliable mirror maker.

However, this proposal is a much more conservative extension to the existing Kafka API; this makes the implementation very simple, and it imposes little to no runtime cost in memory or time. Even if these more elaborate coordination proposals are implemented, this feature would still be useful in cases where simplicity and performance are important.

Comparing Offsets by Key

A similar feature was suggested on the mailing list — instead of checking that the given offset was equal to the upcoming offset for the partition, it would check that the given offset was greater than the last offset for the message's *key*. This feature would have similar advantages to the current proposal, and would reduce contention in some situations: many producers could write to the same partition concurrently, as long as none of them sent messages with the same key.

Compared to the current proposal, this has a couple of downsides:

- It requires an additional datastructure, of size proportional to the number of unique keys present in the log, which would need to be maintained and checkpointed.
- There are other use cases where checking offsets at the partition level is important: for example, a KV store in Samza might use a partition as a commit log for a KV store... and it would want to ensure that no other task is writing to that partition, not just some particular key. (Of course, per-key or per-partition checking could be selected by a config flag.)

While this may make a good future extension, it seems simpler to exclude it from the first iteration.