

KIP-129: Streams Exactly-Once Semantics

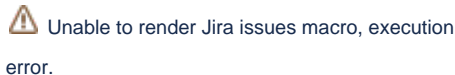
- [Status](#)
- [Motivation](#)
- [Summary of Guarantees](#)
- [Proposed Changes](#)
 - [Transactionally committing a task](#)
 - [Uncleanly shutting down a task](#)
 - [Better handling runtime errors](#)
- [Compatibility, Deprecation, and Migration Plan](#)
- [Rejected Alternatives](#)

Status

Current state: [Accepted: \[VOTE\] KIP-129: Kafka Streams Exactly-Once Semantics](#)

Discussion thread: [\[DISCUSS\] KIP-129: Kafka Streams Exactly-Once Semantics](#)

JIRA:

 Unable to render Jira issues macro, execution error.

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

Motivation

[KIP-98](#) added the following capabilities to Apache Kafka

1. An Idempotent Producer based on producer identifiers (PIDs) to eliminate duplicates.
2. Cross-partition transactions for writes and offset commits
3. Consumer support for fetching only committed messages

This proposal makes use of these capabilities to strengthen the semantics of Kafka's [streams api](#) for stream processing.

The critical question for a stream processing system is "does my stream processing application get the right answer, even if one of the instances crashes in the middle of processing?". The challenge in ensuring this is resuming the work being carried out by the failed instances in exactly the same state as before the crash.

A simple example of this in Kafka land would be a stream processor which took input from some topic, transformed it, and produced output to a new output topic. In this case "getting the right answer" means neither missing any input messages nor producing any duplicate output. This is often called "exactly once semantics" or "exactly once delivery".

In reality the failure scenarios are far more complex than this simple case of producing duplicates:

- We may be taking input from multiple sources and the ordering across these sources may be non-deterministic
- Likewise we may be producing output to multiple destination topics
- We may be aggregating or joining across multiple inputs using the managed state facilities the streams api provides
- We may be looking up side information in a data system or calling out to a service that is updated out of band

Failure and restart, especially when combined with non-determinism and changes to the persistent state computed by the application, may result not only in duplicates but in incorrect results. For example if one stage of processing is computing a count of the number of events seen then a duplicate in an upstream processing stage may lead to an incorrect count downstream. Hence the phrase "Exactly once processing" is a bit narrow, though it is correct to think of the guarantee as ensuring that the output would be the same as it would if the stream processor saw each message exactly one time (as it would in a case where no failure occurred).

Currently no stream processing layer addresses this problem when used with Kafka (or most other similar low-latency messaging layers). Some systems prevent duplication in the processing tier, but none handle duplicates in the import of streams into Kafka or the integration between the processing layer and Kafka. Even some of the systems that claim to provide strong semantics in their processing layer, discounting the integration with Kafka, have design limitations that lead to incorrect results in the presence of non-determinism. Unlike in batch processing, in stream processing non-determinism is extremely common. Simple things like the arrival order of data from different inputs or lookups against data in an external DB or service that is being updated asynchronously will introduce non-determinism to processing. Hence any semantic guarantee that doesn't handle non-determinism will be quite limited in its usefulness.

How can we provide a strong guarantee for systems to enable correct stream processing with streams coming from and going to Kafka? The features in KIP-98 provide a basis for fixing this. We can think of stream processing in Kafka as any application that performs the following activities:

1. Reading input from one or more input topics

2. Making changes to local state, which, however it is layed out on disk or in memory can be journaled to a compacted Kafka topic
3. Producing output to one or more output topics
4. Updating the consumer offset to mark the input as processed

The key problem of ensuring correct semantics is to guarantee that these actions all either happen together or don't happen together. If the application undergoes the state change but fails to produce output, or produces only part of its output, or does not record local state modifications, then upon restart it will not pick up in the same state and can't be guaranteed to produce an output it might have in a non-failure scenario.

KIP-98 provides a mechanism for wrapping all four of these actions in a single atomic "transaction" that will either happen or not happen. This combination of features in Kafka—Kafka based offset storage, transactional write capabilities, and compacted topics for journaling state changes—allow us to view the act of stream processing as a set of Kafka updates and KIP-98 allows these updates to all occur transactionally. In this sense KIP-98 brings a kind of "closure" to the Kafka protocol and makes it a basis for stream processing.

This is a very general protocol level facility and could be wrapped up in any number of possible processing or application development layers, interfaces, or languages.

However building a stream processing application using raw transactions exposed in the producer and consumer is still some work. The goal of this KIP is to take advantage of this protocol-level capability to provide strong semantics in the presence of failure in Kafka's own streams api in a way that is transparent to the end user.

In this KIP we will only focus on the user facing changes: the summary of guarantees, the public API changes, etc. The details of the design is presented in a [separate document](#).

Summary of Guarantees

With Kafka Streams, each input record fetched from the source Kafka topics will only be processed exactly once: its processing effects, both in potential associated state store updates and resulted records sent to output Kafka topics, will be reflected only one, even under failures.

Public Interfaces

The only public interface changes proposed in this KIP is adding one more config to StreamsConfig:

processing. guarantee	<p>Here are the possible values:</p> <p>exactly_once: the processing of each record will be reflected exactly once in the application's state even in the presence of failures.</p> <p>at_least_once: the processing of each record will be reflected at least once in the application's state even in the presence of failures.</p> <p>Default: at_least_once</p>
--------------------------	---

The tradeoff between these modes is that in `exactly_once` mode the output will be wrapped in a transaction and hence to consumers running in `read_committed` mode, will not be available until that task commits it's output. The frequency of this is controlled by the existing config `commit.interval.ms`. This can be made arbitrarily small, down to processing each input in its own transaction, though as that commit becomes more frequent the overhead of the transaction will be higher.

Proposed Changes

In this section, we will present only a very high level overview of the proposed changes to leverage on [KIP-98](#) features and provide the above exactly once processing guarantees.

As mentioned earlier, we have a separate design document which dives into all the implementation details, interested reader are encouraged to read it [here](#). Note that all these changes will be internal and are completely abstracted away from the users.

Transactionally committing a task

When a task needs to be committed, we need to write the offsets using the *producer.sendOffsetsToTransaction API*, so that committing the offsets is wrapped as part of the producer transaction, which will either succeed or fail along with the sent messages within the transaction atomically.

In order to do that, we need to create one producer client per each task instead of one producer per thread shared among all hosted tasks.

On the consumer side, we will set the [isolation.level](#) config to `read_committed` to make sure that any consumed messages are from committed transactions.

Uncleanly shutting down a task

For any updates applied to the local state stores, they usually cannot be rolled back once the current ongoing transaction is determined to be rolled back.

In this case, we cannot reuse the local state stores any more when resuming the task. In this case, we need to indicate that the task was uncleanly shut down and hence we need to restore the states from changelogs, which stores transactional update records and any aborted updates will not be applied.

Better handling runtime errors

Today we throw many of the runtime exceptions to users, which are potentially captured in the user-customizable handler function. With the exactly-once semantics, we should now handle some of those non-fatal errors automatically by aborting the task's ongoing transaction and then resuming from the last committed transaction.

Compatibility, Deprecation, and Migration Plan

The proposed change should be backward compatible. Users could simply swipec in the new jar as runtime dependency and restart their application (plus changing the config of `processing.guarantee`) to get the Exactly Once semantics.

Rejected Alternatives

- "producer per thread" design (including a discussion about pros/cons for both approaches)
 - https://docs.google.com/document/d/1CfOJaa6mdg5o7pLf_zXISV4oE0ZeMZwT_sG1QWg4EE/edit
 - *"Our overall proposal is, to implement KIP-129 as is as a "Stream EoS 1.0" version. The raised concerns are all valid, but hard to quantify at the moment. Implementing KIP-129, that provides a clean design, allows us to gain more insight in the performance implications. This enables us, to make an educated decision, if the "producer per task" model is performant enough (with or without any of the proposed improvements) or if a switch to a "producer per thread" model is mandatory. If we do decide to do the "producer per thread" design, the changes should be incremental and easily achievable."*