KIP-447: Producer scalability for exactly once semantics

Status

Current state: Under discussion

Discussion thread: here

JIRA: KAFKA-8587 - Getting issue details... STATUS

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

Motivation

Exactly once semantics (EOS) provides transactional message processing guarantees. Producers can write to multiple partitions atomically so that either all writes succeed or all writes fail. This can be used in the context of stream processing frameworks, such as Kafka Streams, to ensure exactly once processing between topics.

In Kafka EOS, we use the concept of a "transactional Id" in order to preserve exactly once processing guarantees across process failures and restarts. Essentially this allows us to guarantee that for a given transactional Id, there can only be one producer instance that is active and permitted to make progress at any time. Zombie producers are fenced by an epoch which is associated with each transactional Id. We can also guarantee that upon initialization, any transactions which were still in progress are completed before we begin processing. This is the point of the `initTransactions()` API.

The problem we are trying to solve in this proposal is a semantic mismatch between consumers in a group and transactional producers. In a consumer group, ownership of partitions can transfer between group members through the rebalance protocol. For transactional producers, assignments are assumed to be static. Every transactional id must map to a consistent set of input partitions. To preserve the static partition mapping in a consumer group where assignments are frequently changing, the simplest solution is to create a separate producer for every input partition. This is what Kafka Streams does today.

This architecture does not scale well as the number of input partitions increases. Every producer come with separate memory buffers, a separate thread, separate network connections. This limits the performance of the producer since we cannot effectively use the output of multiple tasks to improve batching. It also causes unneeded load on brokers since there are more concurrent transactions and more redundant metadata management.

It's strongly recommended to read the detailed design doc for better understanding the internal changes. This KIP only presents high level ideas.

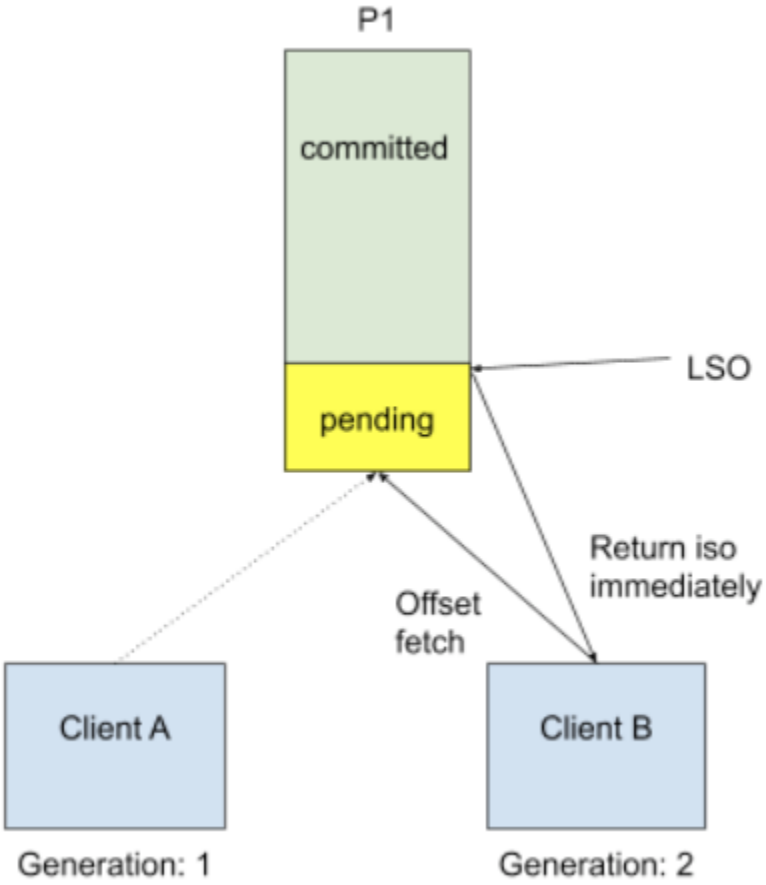
Proposed Changes

The root of the problem is that transaction coordinators have no knowledge of consumer group semantics. They simply do not understand that partitions can be moved between processes. Let's take a look at a sample exactly-once use case, which is quoted from KIP-98:

KafkaTransactionsExample.java

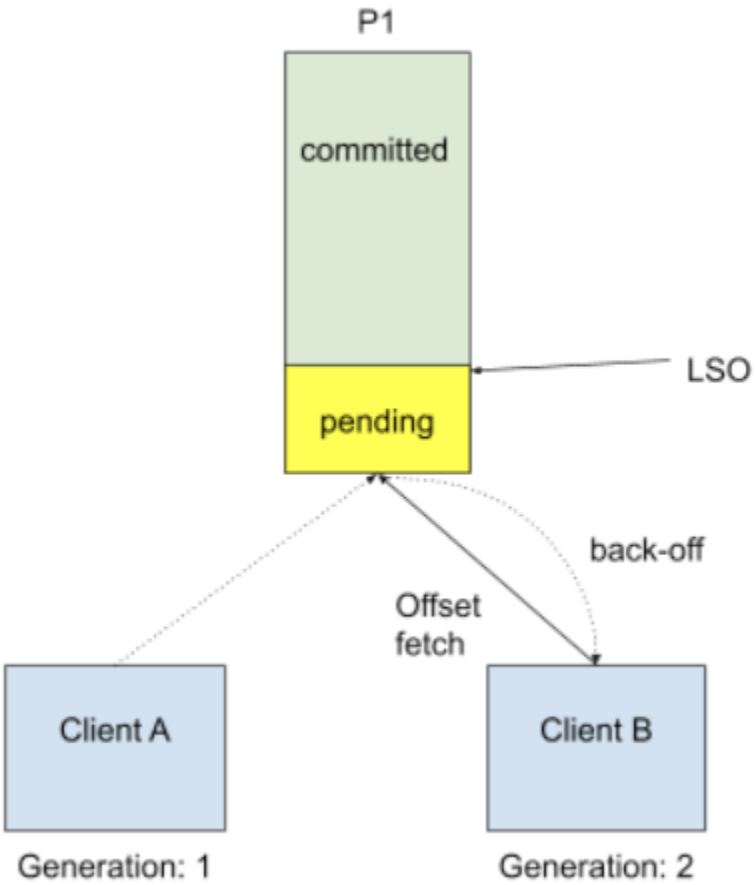
```
public class KafkaTransactionsExample {

  public static void main(String args[]) {
    KafkaConsumer<String, String> consumer = new KafkaConsumer<>(consumerConfig);

    KafkaProducer<String, String> producer = new KafkaProducer<>(producerConfig);
    producer.initTransactions();

    while(true) {
      ConsumerRecords<String, String> records = consumer.poll(CONSUMER_POLL_TIMEOUT);
      if (!records.isEmpty()) {
        producer.beginTransaction();

        List<ProducerRecord<String, String>> outputRecords = processRecords(records);
        for (ProducerRecord<String, String> outputRecord : outputRecords) {
          producer.send(outputRecord);
        }

        sendOffsetsResult = producer.sendOffsetsToTransaction(getUncommittedOffsets());

        producer.endTransaction();
      }
    }
  }
}
```

As one could see, the first thing when a producer starts up is to register its identity through `initTransactions` API. Transaction coordinator leverages this step in order to fence producers using the same transactional.id and to ensure that previous transactions must complete. In the above template, we call consumer.poll() to get data, and internally for the very first time we start doing so, consumer needs to know the input topic offset. This is done by a FetchOffset call to group coordinator. With transactional processing, there could be offsets that are "pending", I.E they are part of some ongoing transactions. Upon receiving FetchOffset request, broker will export offset position to the "latest stable offset" (LSO), which is the largest offset that has already been committed when consumer isolation.level is `read_committed`. Since we rely on unique transactional.id to revoke stale transaction, we believe any pending transaction will be aborted when producer calls initTransaction again. During normal use case such as Kafka Streams, we will also explicitly close producer to send out a EndTransaction request to make sure we start from clean state.



This approach is no longer safe when we allow topic partitions to move around transactional producers, since transactional coordinator doesn't know about partition assignment and producer won't call initTransaction again during its life cycle. Omitting pending offsets and proceed could introduce duplicate processing. The proposed solution is to reject FetchOffset request by sending out a new exception called PendingTransactionException to new client when there is pending transactional offset commits, so that old transaction will eventually expire due to transaction timeout. After expiration, transaction coordinator will take care of writing abort transaction markers and bump the producer epoch. When client receives PendingTransactionException, it will back-off and retry getting input offset until all the pending transaction offsets are cleared. This is a trade-off between availability and correctness. The worst case for availability loss is just waiting for transaction timeout when the last generation producer wasn't shut down gracefully, which should be rare.

Below is the new approach we discussed:



Note that the current default transaction.timeout is set to one minute, which is too long for Kafka Streams EOS use cases. Considering the default commit interval was set to only 100 milliseconds, we would doom to hit session timeout if we don't actively commit offsets during that tight window. So we suggest to shrink the transaction timeout to be the same default value as session timeout (10 seconds) on Kafka Streams, to reduce the potential performance loss for offset fetch delay when some instances accidentally crash.

Public Interfaces

The main addition of this KIP is a new variant of the current `initTransactions` API which gives us access to the consumer group states, such as member state and generation.id.

```
interface Producer {
  /**
   * This API shall be called for consumer group aware transactional producers.
   */
  void initTransactions(Consumer<byte[], byte[]> consumer); // NEW

  /**
   * No longer need to pass in the consumer group id in a case where we already get access to the consumer
state.
   */
  void sendOffsetsToTransaction(Map<TopicPartition, OffsetAndMetadata> offsets) throws ProducerFencedException,
IllegalGenerationException; // NEW
}
```

Shrink transactional.timeout

We shall set `transaction.timout.ms` default to 10000 ms (10 seconds) on Kafka Streams.

Offset Fetch Request

We will add a new error code for consumer to wait for pending transaction clearance. In order to be able to return corresponding exceptions for old/new clients, we shall also bump the OffsetFetch protocol version.

```
PENDING_TRANSACTION(85, "There are pending transactions for the offset topic that need to be cleared",
PendingTransactionException::new),
```

In the meantime, this offset fetch back-off should be only applied to EOS use cases, not general offset fetch use case such as admin client access. we shall also define a flag within offset fetch request so that we only trigger back-off logic when the request is on isolation level read_committed.

```
OffsetFetchRequest => Partitions GroupId IsolationLevel
  Partitions          => List<TopicPartition>
  GroupId             => String
  WaitTransaction     => Boolean // NEW
```

Fence Zombie

A zombie process may invoke InitProducerId after falling out of the consumer group. In order to distinguish zombie requests, we need to leverage group coordinator to fence out of sync client.

To help get access to consumer state for txn producer, consumer will expose a new API for some of its internal states as an opaque struct:

```
// public
interface GroupMetadata {
}

// private
interface ConsumerGroupMetadata extends GroupMetadata {
  final String groupId;
  final int generationId;
  final String memberId;
  final Optional<String> groupInstanceId;
}

// Consumer API
public GroupMetadata groupMetadata();
```

Thus producer could poll the metadata as it needs during normal processing.

A new generation.id field shall be added to the `TxnOffsetCommitRequest` request. In the meantime we also suggest to add member.id and group. instance.id to the request to make the txn offset commit fencing consistent with normal offset fencing.

```
TxnOffsetCommitRequest => TransactionalId GroupId ProducerId ProducerEpoch Offsets GenerationId
  TransactionalId     => String
  GroupId             => String
  ProducerId                    => int64
  ProducerEpoch       => int16
  Offsets                  => Map<TopicPartition, CommittedOffset>
  GenerationId        => int32 // NEW
  MemberId                        => String // NEW
  GroupInstanceId           => String // NEW
```

If the generation.id is not matching group generation, the client will be fenced immediately. An edge case is defined as:

```
1. Client A tries to commit offsets for topic partition P1, but haven't got the chance to do txn offset commit
before a long GC.
2. Client A gets out of sync and becomes a zombie due to session timeout, group rebalanced.
3. Another client B was assigned with P1.
4. Client B doesn't see pending offsets because A hasn't committed anything, so it will proceed with
potentially `pending` input data
5. Client A was back online, and continue trying to do txn commit. Here if we have generation.id, we will catch
it!
```

And here is a recommended new transactional API usage example:

```
  KafkaConsumer consumer = new KafkaConsumer<>(consumerConfig);
  KafkaProducer producer = new KafkaProducer();

  // Will access consumer internal state. Only called once in the app's life cycle.
  // Note that this is a blocking call until consumer successfully joins the group.
  producer.initTransactions(consumer);
  while (true) {
    // Read some records from the consumer and collect the offsets to commit
    ConsumerRecords consumed = consumer.poll(Duration.ofMillis(5000)); // This will be the fencing point if
there are pending offsets for the first time.
    Map<TopicPartition, OffsetAndMetadata> consumedOffsets = offsets(consumed);

    // Do some processing and build the records we want to produce
    List<ProducerRecord> processed = process(consumed);

    // Write the records and commit offsets under a single transaction
    producer.beginTransaction();
    for (ProducerRecord record : processed)
      producer.send(record);

    producer.sendOffsetsToTransaction(consumedOffsets);

    producer.commitTransaction();
  }
```

Some key observations are:

1. User must be utilizing both consumer and producer as a complete EOS application,
2. User needs to store transactional offsets inside Kafka group coordinator, not in any other external system for the sake of fencing,
3. Need to call the new producer.initTransactions(consumer); which passes in a consumer struct for state access during initialization,
4. Producer no longer needs to call sendOffsetsToTransaction(offsets, consumerGroupId) because we will be able to access consumer group id internally. Instead just pass offsets as single parameter.

Compatibility, Deprecation, and Migration Plan

It's extremely hard to preserve two types of stream clients within the same application due to the difficulty of state machine reasoning and fencing. It would be the same ideology for the design of upgrade path: **one should never allow task producer and thread producer under the same application group.**

Following the above principle, Kafka Streams uses version probing to solve the upgrade problem. Step by step guides are:

1. Broker must be upgraded to 2.4 first. This means the `inter.broker.protocol.version` (IBP) has to be set to the latest. Any produce request with higher version will automatically get fenced because of no support.
2. Upgrade the stream application binary and choose to set UPGRADE_FROM_CONFIG config to 2.3 or lower. Do the first rolling bounce, and make sure the group is stable with every instance on 2.4 binary.
3. Just remove/unset that config, to make application point to actual Kafka client version 2.4. Do second rolling bounce and now the application officially starts using new thread producer for EOS.

The reason for doing two rolling bounces is because the old transactional producer doesn't have access to consumer generation, so group coordinator doesn't have an effective way to fence old zombies. By doing first rolling bounce, the task producer will also opt in accessing the consumer state and send TxnOffsetCommitRequest with generation. With this foundational change, it is much safer to execute step 3.

Rejected Alternatives

- We could use admin client to fetch the inter.broker.protocol on start to choose which type of producer they want to use. This approach however is harder than we expected, because brokers maybe on the different versions and if we need user to handle the tricky behavior during upgrade, it would actually be unfavorable. So a hard-coded config is a better option we have at hand.
- We have considered to leverage transaction coordinator to remember the assignment info for each transactional producer, however this means we are copying the state data into 2 separate locations and could go out of sync easily. We choose to still use group coordinator to do the generation and partition fencing in this case.
- We once decided to use a transactional.group.id config to replace the transactional.id, and consolidate all the transaction states for an application under one transactional coordinator. This use case is no longer needed once we rely on group coordinator to do the fencing, and we could re-implement it any time in the future with new upgrade procedure safely.