EOS Scalability Design

The purpose of this doc is to give a walkthrough of internal changes introduced in KIP-447. It is highly recommended to read through the exactly once design doc before proceeding to read this document.

What we currently have

In KIP-98, a term called *transactional.id* was defined on transactional producer to guarantee exactly once throughout its multiple lifecycles. During initialization/restart, the producer will send an InitPidRequest to transaction coordinator hosted on one of the brokers to abort/committed any ongoing transactions it has started but not finished. Each reinitialization will bump producer epoch so that duplicate or zombie producers configured with same transactional.id could be fenced. To gain access to the transaction state, coordinator discovery process always routes the same transactional producer's initialization request to the same coordinator based on this assumed unique transactional.id.

On the stream side, we define the transactional.id based on the resource assigned to the producer, and producer's life cycle ends on every rebalance. Each producer will be in charge of dealing with exactly one task (or exactly one topic partition as input). In the rebalance, when the task assignments get shuffled, stream thread will close task producer from last generation and bounce up new producers in an one-to-one mapping with assigned tasks. The task id will be used as transactional.id for task producer initialization and could be guaranteed unique based on input topic partitions, as the assignment was inherited from consumer group protocol. Obviously this is not a well scaled solution and number of producers grow linearly with number of tasks. Too many producers come with separate memory buffers, separate threads, separate network connections which is a waste of hardware resources, and couldn't benefit from producer batching anymore.

Our goal is to allow each stream thread only initializes one thread producer and batch all the transaction states for assigned tasks together. However, this proposal will break the correctness. The problem was that transaction coordinator could no longer be leveraged for fencing since the producer state could change within one life cycle.

What we are proposing

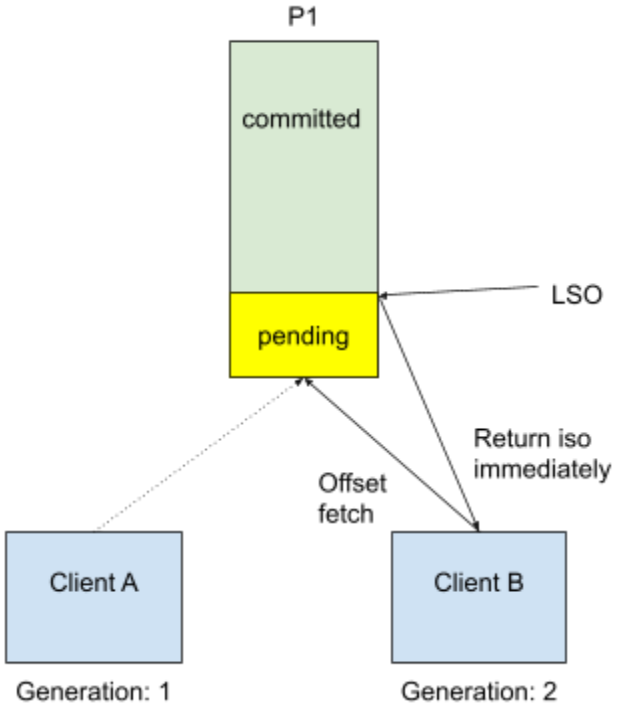
It's hard to make transaction coordinator understand the group assignment semantic, because it assumes each transactional producer as individual clients. It's also not favorable to add group assignment state to transactional coordinator neither, as group coordinator already maintains

that state. Having two copies of the same state data could easily go out of sync and cause bugs. The new proposal is to solve fencing problem on group coordinator side. Let's first take a look at a sample exactly-once use case, which is quoted from KIP-98:

```
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.commitTransactions();
      }
    }
  }
}
```

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,

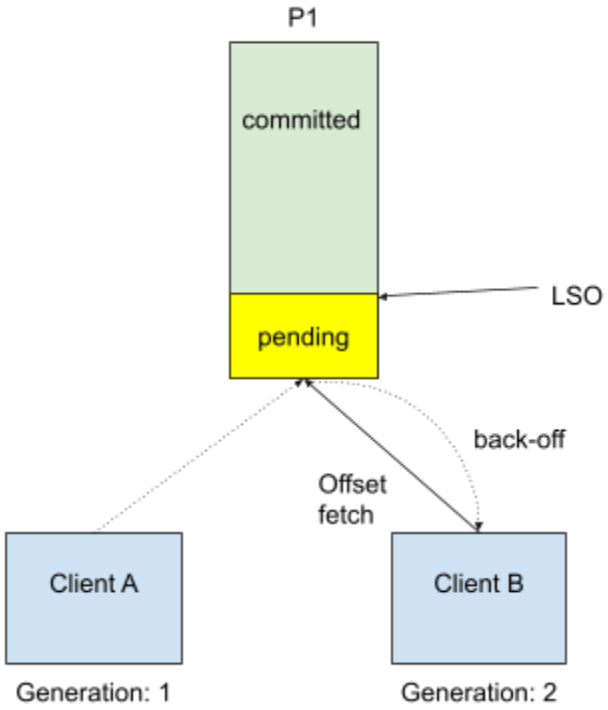
we call `consumer.poll()` to get data, and internally for the very first time we start doing so, consumer needs to know the input topic offset. This is done by a FetchOffset call to the group coordinator. With transactional processing, there could be offsets that are "pending", I.E they are part of ongoing 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 as producer calls initTransaction again. In 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. For old consumers, we will choose to send a COORDINATOR_LOAD_IN_PROGRESS exception to let it retry, too. When client receives `PendingTransactionException` or COORDINATOR_LOAD_IN_PROGRESS, 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 first 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), to reduce the potential performance loss for offset fetch delay when some instances accidentally crash.

This KIP change only takes effect in Kafka Streams EOS and customized EOS use cases following the recommended template below. Major differences from KIP-98 template are highlighted:

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

producer.initTransactions();
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,
consumer.groupMetadata());

  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. Producer needs to call sendOffsetsToTransaction(offsets, groupMetadata) to be able to fence properly,

Next we will walkthrough what will happen during each step of the code template above.

```
producer.initTransactions();
```

Nothing new will be done here, the initialization process shall be the same.

```
ConsumerRecords consumed = consumer.poll(Duration.ofMillis(5000));
Map<TopicPartition, OffsetAndMetadata> consumedOffsets =
offsets(consumed);
```

Consumer polls in a batch of data to continue processing.

Blocking on fetching offsets

As we have discussed above, the blocking point for last generation transaction is the pending consumer offsets, or LSO. On every rebalance end, consumer will do a full offset fetch call to group coordinator and potentially be asked to back off until zombie transactions get aborted through timeout.
A new error code PendingTransactionException shall be sent back to the client and consumer will keep attempting to get offsets until the transaction timeout in the worst case that someone hasn't closed its state properly on last generation. After the offset fetch is successful, the program proceeds.

```
List<ProducerRecord> processed = process(consumed);
```

Application exercises business logic on the consumed data.

```
producer.beginTransaction();
  for (ProducerRecord record : processed)
    producer.send(record);
```

Producer changes its own state to ongoing transaction, and sends data to corresponding brokers. If this is the first time it writes data to a topic partition, an AddPartitionsTxnRequest shall be sent to the transaction coordinator for it to memorize the touched topic partitions within current ongoing transaction.

```
producer.sendOffsetsToTransaction(consumedOffsets,
consumer.groupMetadata());
```

Client will attempt to send AddOffsetsToTxnRequest during `sendOffsetsToTransaction` call if this is the first time it tries to commit txn offset to a specific topic partition.

After the call to transaction coordinator succeed, client will do transaction offset commit to the group coordinator. Producer will be appending the generation.id into the request so that group coordinator could fence zombie request.

```
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
```

Producer will get fenced if both conditions are satisfied:

1. It has a non-negative generation.id field. Otherwise this suggests a standalone producer who doesn't utilize consumer state.
2. Provided generation.id doesn't equal to the current consumer group generation.

A `ProducerFencedException` shall be returned and the new producer client will crash immediately because it was assumed as a zombie.

We also opt to throw exception when the new producer calls `sendOffsetsToTransaction(offsets, consumerGroupId)` to ensure API consistency.

`producer.commitTransaction();`

Producer will try to commit the ongoing transaction by writing an EndTxnRequest to the transaction coordinator. The coordinator will write a prepare_commit marker to its transactional log and broadcast WriteTxnMarkersRequest to all the affected parties such as input/output topics, offset topics, etc.

Stream Upgrade Path

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 2.5 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.
3. Just remove/unset version config in order to make application point to actual Kafka client version 2.5. 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.

Alternative Discussions

Generic Consumer API Support

Currently we pass in *Consumer*<K, V> interface into KafkaStreams, instead of the full implementation type *KafkaConsumer*<K, V>. With the new initTransaction API we have to cast the generic *Consumer* type to *KafkaConsumer*, meaning any implementation on top of *Consumer* will be automatically failing. Whether to continue supporting generic consumer is a question we need to address during the design phase.

To make the generic consumer capable of doing transactional commit fencing, we estimate at least several public APIs should be added:

- **int generationId();**
    - Get internal generation.id access
- **map<TopicPartition, PartitionData> fetchOffsets(boolean allPartitions);**
    - A fetchOffset call to block on pending transaction offsets
- **void blockOnRebalance(long timeout);**
    - Let the member join and block until reaching rebalance

Besides public APIs, the consumer implementation must be able to handle exceptions like **CoordinatorNotAvailable** and **PendingTransactions**.