KIP-447: Producer scalability for exactly once semantics

Status

Current state: Adopted (2.6.0)

Discussion thread: here

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

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.

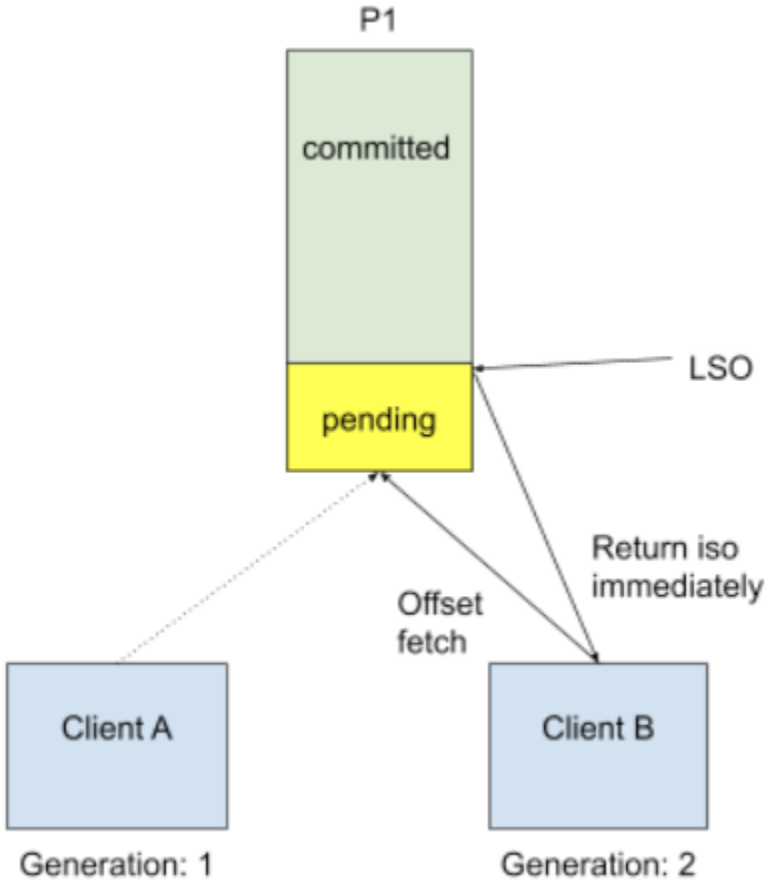
Proposed Changes

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

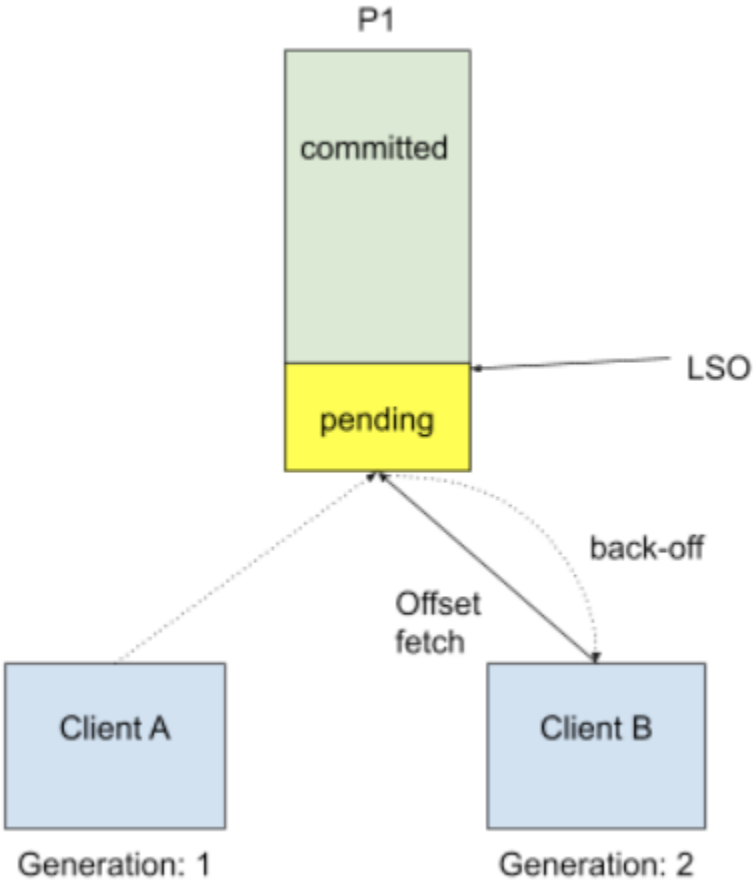
```
public class KafkaTransactionsExample {

  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 sendOffsetsToTransaction API which gives us access to the consumer group states, such as generation.id, member.id and group.instance.id, etc.

```
interface Producer {
   /**
     * Sends a list of specified offsets to the consumer group coordinator, and also marks
     * those offsets as part of the current transaction. These offsets will be considered
     * committed only if the transaction is committed successfully. The committed offset should
     * be the next message your application will consume, i.e. lastProcessedMessageOffset + 1.
     * <p>
     * This method should be used when you need to batch consumed and produced messages
     * together, typically in a consume-transform-produce pattern. Thus, the specified
     * {@code consumerGroupId} should be the same as config parameter {@code group.id} of the used
     * {@link KafkaConsumer consumer}. Note, that the consumer should have {@code enable.auto.commit=false}
     * and should also not commit offsets manually (via {@link KafkaConsumer#commitSync(Map) sync} or
     * {@link KafkaConsumer#commitAsync(Map, OffsetCommitCallback) async} commits).
     *
     * This API won't deprecate the existing {@link KafkaProducer#sendOffsetsToTransaction(Map, String)
sendOffsets} API as standalone
     * mode EOS applications are still relying on it. If the broker doesn't support the new underlying
transactional API, the caller will crash.
         *
     * @throws IllegalStateException if no transactional.id has been configured or no transaction has been
started
     * @throws ProducerFencedException fatal error indicating another producer with the same transactional.id
is active
     * @throws org.apache.kafka.common.errors.UnsupportedVersionException fatal error indicating the broker
     *         does not support transactions (i.e. if its version is lower than 0.11.0.0)
     * @throws org.apache.kafka.common.errors.UnsupportedForMessageFormatException  fatal error indicating the
message
     *         format used for the offsets topic on the broker does not support transactions
     * @throws org.apache.kafka.common.errors.AuthorizationException fatal error indicating that the configured
     *         transactional.id is not authorized. See the exception for more details
     * @throws org.apache.kafka.common.errors.IllegalGenerationException if the passed in consumer metadata has
illegal generation
     * @throws org.apache.kafka.common.errors.FencedInstanceIdException if the passed in consumer metadata has
a fenced group.instance.id
     * @throws KafkaException if the producer has encountered a previous fatal or abortable error, or for any
     *         other unexpected error
     */
   void sendOffsetsToTransaction(Map<TopicPartition, OffsetAndMetadata> offsets, ConsumerGroupMetadata
consumerGroupMetadata) throws ProducerFencedException, IllegalGenerationException, FencedInstanceIdException;
// NEW
}
```

Shrink transactional.timeout

We shall set `transaction.timout.ms` default to 10000 ms (10 seconds) on Kafka Streams. For non-stream users, we highly recommend you to do the same if you want to use the new semantics.

Shrink abort timeout transaction scheduler interval

We shall set `transaction.abort.timed.out.transaction.cleanup.interval.ms` default to 10000 ms (10 seconds) on broker side (previously one minute). This config controls the scheduled interval for purging expired transactions, which we need to tune more frequently to timeout zombie transactions.

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 involved in EOS, I.E on isolation level read_committed.

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

The PendingTransactionException could also be surfaced to all the public APIs in Consumer:

KafkaConsumer.java

```
    /**
     * Get the offset of the <i>next record</i> that will be fetched (if a record with that offset exists).
     * This method may issue a remote call to the server if there is no current position for the given
partition.
     * <p>
     * This call will block until either the position could be determined or an unrecoverable error is
     * encountered (in which case it is thrown to the caller).
     *
     * @param partition The partition to get the position for
     * @param timeout   The maximum duration of the method
     *
     * @return The current position of the consumer (that is, the offset of the next record to be fetched)
     * @throws IllegalArgumentException if the provided TopicPartition is not assigned to this consumer
     * @throws org.apache.kafka.clients.consumer.InvalidOffsetException if no offset is currently defined for
     *             the partition
     * @throws org.apache.kafka.common.errors.WakeupException if {@link #wakeup()} is called before or while
this
     *             function is called
     * @throws org.apache.kafka.common.errors.InterruptException if the calling thread is interrupted before or
while
     *             this function is called
     * @throws org.apache.kafka.common.errors.TimeoutException if time spent blocking exceeds the {@code
timeout}
     * @throws org.apache.kafka.common.errors.AuthenticationException if authentication fails. See the
exception for more details
     * @throws org.apache.kafka.common.errors.AuthorizationException if not authorized to the topic or to the
     *             configured groupId. See the exception for more details
+    * @throws org.apache.kafka.common.errors.PendingTransactionException if there are other pending
transactional commits
     * @throws org.apache.kafka.common.KafkaException for any other unrecoverable errors
     */
    long position(TopicPartition partition, Duration timeout);

    /**
     * Get the last committed offset for the given partition (whether the commit happened by this process or
     * another). This offset will be used as the position for the consumer in the event of a failure.
     * <p>
     * This call will block to do a remote call to get the latest committed offsets from the server.
     *
     * @param partition The partition to check
     * @param timeout   The maximum duration of the method
     *
     * @return The last committed offset and metadata or null if there was no prior commit
     * @throws org.apache.kafka.common.errors.WakeupException if {@link #wakeup()} is called before or while
this
     *             function is called
     * @throws org.apache.kafka.common.errors.InterruptException if the calling thread is interrupted before or
while
     *             this function is called
     * @throws org.apache.kafka.common.errors.AuthenticationException if authentication fails. See the
exception for more details
     * @throws org.apache.kafka.common.errors.AuthorizationException if not authorized to the topic or to the
     *             configured groupId. See the exception for more details
+    * @throws org.apache.kafka.common.errors.PendingTransactionException if there are other pending
transactional commits
     * @throws org.apache.kafka.common.KafkaException for any other unrecoverable errors
     * @throws org.apache.kafka.common.errors.TimeoutException if time spent blocking exceeds the {@code
timeout}
     */
```

```
    OffsetAndMetadata committed(TopicPartition partition, final Duration timeout);

    /**
     * Fetch data for the topics or partitions specified using one of the subscribe/assign APIs. It is an error
to not have
     * subscribed to any topics or partitions before polling for data.
     * <p>
     * On each poll, consumer will try to use the last consumed offset as the starting offset and fetch
sequentially. The last
     * consumed offset can be manually set through {@link #seek(TopicPartition, long)} or automatically set as
the last committed
     * offset for the subscribed list of partitions
     *
     * <p>
     * This method returns immediately if there are records available. Otherwise, it will await the passed
timeout.
     * If the timeout expires, an empty record set will be returned. Note that this method may block beyond the
     * timeout in order to execute custom {@link ConsumerRebalanceListener} callbacks.
     *
     *
     * @param timeout The maximum time to block (must not be greater than {@link Long#MAX_VALUE} milliseconds)
     *
     * @return map of topic to records since the last fetch for the subscribed list of topics and partitions
     *
     * @throws org.apache.kafka.clients.consumer.InvalidOffsetException if the offset for a partition or set of
     *             partitions is undefined or out of range and no offset reset policy has been configured
     * @throws org.apache.kafka.common.errors.WakeupException if {@link #wakeup()} is called before or while
this
     *             function is called
     * @throws org.apache.kafka.common.errors.InterruptException if the calling thread is interrupted before or
while
     *             this function is called
     * @throws org.apache.kafka.common.errors.AuthenticationException if authentication fails. See the
exception for more details
     * @throws org.apache.kafka.common.errors.AuthorizationException if caller lacks Read access to any of the
subscribed
     *             topics or to the configured groupId. See the exception for more details
     * @throws org.apache.kafka.common.KafkaException for any other unrecoverable errors (e.g. invalid groupId
or
     *             session timeout, errors deserializing key/value pairs, your rebalance callback thrown
exceptions,
     *             or any new error cases in future versions)
     * @throws java.lang.IllegalArgumentException if the timeout value is negative
     * @throws java.lang.IllegalStateException if the consumer is not subscribed to any topics or manually
assigned any
     *             partitions to consume from
     * @throws java.lang.ArithmeticException if the timeout is greater than {@link Long#MAX_VALUE} milliseconds.
     * @throws org.apache.kafka.common.errors.InvalidTopicException if the current subscription contains any
invalid
     *             topic (per {@link org.apache.kafka.common.internals.Topic#validate(String)})
     * @throws org.apache.kafka.common.errors.FencedInstanceIdException if this consumer instance gets fenced
by broker.
+    * @throws org.apache.kafka.common.errors.PendingTransactionException if there are other pending
transactional commits
     */
    @Override
    public ConsumerRecords<K, V> poll(final Duration timeout) {
        return poll(time.timer(timeout), true);
    }
```

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. This is already done by KIP-429, and we just showcase the some high level structure here for convenience.

```
ConsumerGroupMetadata {
  public String groupId();

  public int generationId();

  public String memberId();

  public Optional<String> groupInstanceId();
}
```

As we see, the metadata exposed contains member.id, group.instance.id and generation.id, which are essentially the identifiers we use in the normal offset commit protocol. To be able to keep track of the latest metadata information, we will add a top-level API to consumer:

Consumer.java

```
ConsumerGroupMetadata groupMetadata();
```

So that EOS users could get refreshed group metadata as needed.

To pass the information to broker, member.id, group.instance.id and generation.id field shall be added to `TxnOffsetCommitRequest`, which makes 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, default -1 // NEW
  MemberId                        => nullable String // NEW
  GroupInstanceId           => nullable String // NEW
```

If one of the field is not matching correctly on server side, 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 non-EOS usage example:

```
  KafkaConsumer consumer = new KafkaConsumer<>(consumerConfig);
  // Recommend a smaller txn timeout, for example 10 seconds.
  producerConfig.put(ProducerConfig.TRANSACTION_TIMEOUT_CONFIG, 10000);
  KafkaProducer producer = new KafkaProducer(producerConfig);

  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);

        // Pass the entire consumer group metadata
    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.

For Kafka Streams that needs to be backward compatible to older broker versions, we add a new config value for config parameter PROCESSING_GUARANTEE that we call "exaclty_once_`beta`". Hence, users with new broker versions can opt-in the new feature (ie, using a single producer per thread for EOS instead of a producer per task).

```
public class StreamsConfig {

    public static final String EXACTLY_ONCE_BETA = "exactly_once_beta";

}
```

Additionally, we *remove* the task-level metrics "commit-latency-max" and "commit-latency-avg" because the current committing of tasks that consists of multiple steps (like flushing, committing offsets/transactions, writing checkpoint files) is not done on a per-task level any longer, but the different steps are split out into individual phased over all tasks and committing offsets and transactions is unified into one step for all tasks at once. Because those metics cannot be collected in a useful way any longer, we cannot deprecate the metrics and remove in a future release, but need to remove them directly without a deprecation period.

Compatibility, Deprecation, and Migration Plan

Given the new fetch offset fencing mechanism, to opt-into the new producer per thread model, first this new fencing mechanism must be enabled, before the existing producer side fencing can be disabled.

Following the above principle, two rounds of rolling bounced for all Kafka Streams instances are required as follows:

1. Broker must be upgraded to 2.5 first
2. Upgrade the stream application binary and keep the PROCESSING_GUARATNEE setting at "exactly_once". Do the first rolling bounce, and make sure the group is stable with every instance on 2.6 binary.
3. Upgrade the PROCESSING_GUARANTEE setting to "exaclty_once_beta" and do a second rolling bounce to starts using new thread producer for EOS.

The reason for doing two rolling bounces is because the old (ie, 2.5) 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 opt-in accessing the consumer state and send TxnOffsetCommitRequest with generation. With this foundational change, it is save to execute step 3.

***Note that the above recommended upgrade path is for users who need consistency guarantee. For users who don't worry about consistency, step 2 & 3 could be combined into a single rolling bounce with Kafka client library upgrade. The application should resume work without problem.

Of course, enabling producer per thread is optional, and thus, uses can also just stay with the "exctly_once" configuration (or with "at_least_once" if exactly-once semantics are not used to begin with). For both cases, it is not required to upgrade the brokers to 2.5, and a single rolling bounce upgrade of the Kafka Streams applications is sufficient.

Note: It is also possible to switch back from "exactly_once_beta" to "exactly_once" with a single round of rolling bounces.

A downgrade from 2.6 "exaclty_once_beta" to 2.5 (that only supports "exaclty_once") requires two rolling bounced (i.e., follow the upgrade path in reverse order).

Non-stream EOS Upgrade

As for non-streams users, they would require following steps:

1. Same as stream, upgrade the broker version to 2.5
2. Change `sendOffsetsToTransaction` to the new version. Note the service would crash if the detected broker version through txn offset commit protocol is lower than 2.5.
3. Rolling bounce the application

Since we couldn't predict all the implementation cases, this upgrade guide is not guaranteed to be safe for online operation, and there would be a risk of state inconsistency/data loss.

Working with Older Brokers

As stated in the upgrade path, if the broker version is too old, we shall not enable thread producer even running with Kafka Streams 2.6. If you enable "exaclty_once_beta" against pre 2.5 brokers, Kafka Streams will raise an error.

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.