Exactly Once Delivery and Transactional Messaging in Kafka

Motivation

This document outlines a proposal for strengthening the message delivery semantics of Kafka. This builds on significant work which has been done previously, specifically, here and here.

Kafka currently provides at least once semantics, viz. When tuned for reliability, users are guaranteed that every message write will be persisted at least once, without data loss. Duplicates may occur in the stream due to producer retries. For instance, the broker may crash between committing a message and sending an acknowledgment to the producer, causing the producer to retry and thus resulting in a duplicate message in the stream.

Users of messaging systems greatly benefit from the more stringent idempotent producer semantics, viz. Every message write will be persisted exactly once, without duplicates and without data loss -- even in the event of client retries or broker failures. These stronger semantics not only make writing applications easier, they expand the space of applications which can use a given messaging system.

However, idempotent producers don't provide guarantees for writes across multiple TopicPartitions. For this, one needs stronger transactional guarantees, ie. the ability to write to several TopicPartitions atomically. By atomically, we mean the ability to commit a set of messages across TopicPartitions as a unit: either all messages are committed, or none of them are.

Stream processing applications, which are a pipelines of ‘consume-transform-produce’ tasks, absolutely require transactional guarantees when duplicate processing of the stream is unacceptable. As such, adding transactional guarantees to Kafka --a streaming platform-- makes it much more useful not just for stream processing, but a variety of other applications.

A little bit about Transactions and Streams.

In the previous section, we mentioned the main motivation for transactions is to enable exactly once processing in Kafka Streams. It is worth digging into this use case a little more, as it motivates many of the tradeoffs in our design.

Recall that data transformation using Kafka Streams typically happens through multiple stream processors, each of which is connected by Kafka topics. This setup is known as a stream topology and is basically a DAG where the stream processors are nodes and the connecting Kafka topics are vertices. This pattern is typical of all streaming architectures. You can read more about the Kafka streams architecture here.

As such, a transaction for Kafka streams would essentially encompass the input messages, the updates to the local state store, and the output messages. Including input offsets in a transaction motivates adding the `sendOffsets` API to the Producer interface, described below. Further details will be presented in a separate KIP.

Further, stream topologies can get pretty deep --10 stages is not uncommon. If output messages are only materialized on transaction commits, then a topology which is N stages deep will take N x T to process its input, where T is the average time of a single transaction. So Kafka Streams requires speculative execution, where output messages can be read by downstream processors even before they are committed. Otherwise transactions would not be an option for serious streaming applications. This motivates the ‘read uncommitted’ consumer mode described later.

These are two specific instances where we chose to optimize for the streams use case. As the reader works through this document we encourage him/her to keep this use case in mind as it motivated large elements of the proposal.

Summary of Guarantees

Idempotent Producer Guarantees

To implement idempotent producer semantics, we introduce the concepts of a *producer id,* henceforth called the *PID*, and *sequence numbers* for Kafka messages. Every new producer will be assigned a unique *PID* during initialization. The PID assignment is completely transparent to users and is never exposed by clients.

For a given PID, sequence numbers will start from zero and be monotonically increasing, with one sequence number per topic partition produced to. The sequence number will be incremented for every message sent by the producer. Similarly, the broker will increment the sequence number associated with the PID -> topic partition pair for every message it commits for that topic partition. Finally, the broker will reject a message from a producer unless its sequence number is exactly one greater than the last committed message from that PID -> topic partition pair.

This ensures that, even though a producer must retry requests upon failures, every message will be persisted in the log exactly once. Further, since each new instance of a producer is assigned a new, unique, PID, we can only guarantee idempotent production within a single producer session. These semantics have been discussed previously in this document.

Transactional Guarantees

As mentioned in the Motivation section, transactional guarantees enable applications to batch consumed and produced messages into a single atomic unit.

In particular, a 'batch' of messages in a transaction can be consumed from and written to multiple partitions, and are 'atomic' in the sense that writes will fail or succeed as a single unit. Consumers may or may not consume these messages atomically, depending on their configuration. This has been previously discussed here.

Additionally, stateful applications will also be able to ensure continuity across multiple sessions of the application. In other words, Kafka can guarantee idempotent production and transaction recovery across application bounces.

To achieve this, we require that the application provides a unique id which is stable across all sessions of the application. For the rest of this document, we refer to such an id as the *TransactionalId*. While there may be a 1-1 mapping between an TransactionalId and the internal PID, the main difference is the the TransactionalId is provided by users, and is what enables idempotent guarantees across producers sessions described below.

When provided with such an *TransactionalId*, Kafka will guarantee:

1. Idempotent production across application sessions. This is achieved by fencing off old generations when a new instance with the same TransactionalId comes online.

2. Transaction recovery across application sessions. If an application instance dies, the next instance can be guaranteed that any unfinished transactions have been completed (whether aborted or committed), leaving the new instance in a clean state prior to resuming work.

Note that the transactional guarantees mentioned here are from the point of view of the producer. On the consumer side, the guarantees are a bit weaker. In particular, we cannot guarantee that all the messages of a committed transaction will be consumed all together. This is for several reasons:

1. For compacted topics, some messages of a transaction maybe overwritten by newer versions.

2. Transactions may straddle log segments. Hence when old segments are deleted, we may lose some messages in the first part of a transaction.
3. Consumers may seek to arbitrary points within a transaction, hence missing some of the initial messages.
4. Consumer may not consume from all the partitions which participated in a transaction. Hence they will never be able to read all the messages that comprised the transaction.

Design Overview

In this section, we will present only a very high level overview of the key concepts and data flow of transactions. Further sections flesh these concepts out in detail.

Key Concepts

Transactional Messaging

The first part of the design is to enable producers to send a group of messages as a single transaction that either succeeds or fails atomically. In order to achieve this, we introduce a new server-side module called **transaction coordinator**, to manage transactions of messages sent by producers, and commit / abort the appends of these messages as a whole. The transaction coordinator maintains a **transaction log**, which is stored as an internal topic (we call it the **transaction topic**) to persist transaction status for recovery. Similar to the “offsets log” which maintains consumer offsets and group state in the internal `__consumer_offsets` topic, producers do not read or write directly to the transaction topic. Instead they talk to their transaction coordinator who is the leader broker of the hosted partition of the topic. The coordinator can then append the new state of the indicated transactions to its owned transaction topic partition.

We will talk about how the transaction coordinator manages the transaction status from producer requests and persist it in the transaction log in the Transaction Coordinator section.

Offset Commits in Transactions

Many applications talking to Kafka include both consumers and producers, where the applications consume messages from input Kafka topics and produce new messages to output Kafka topics. To achieve “*exactly once*” messaging, we need to make the committing of the consumer offsets part of the producer transactions in order to achieve atomicity. Otherwise, if there is a failure between committing the producer transaction and committing the consumer offsets, data duplicates or data loss will incur upon failover depending on the ordering of these two operations: if committing producer transaction executes first, then upon recovery the input messages will be re-consumed since offsets were not committed, hence *data duplicates*; if committing consumer offsets executes first, then upon recover the output messages that are failed to commit will not be re-send again, hence *data loss*.

Therefore, we want to guarantee that for each message consumed from the input topics, the resulting message(s) from processing this message will be reflected in the output topics exactly once, even under failures. In order to support this guarantee, we need to include the consumer's offset commits in the producer's transaction.

We will talk about how to enhance the consumer coordinator that takes care of the offset commits to be transaction-aware in the Consumer Coordinator section.

Control Messages for Transactions

For messages appended to Kafka log partitions, in order to indicate whether they are committed or aborted, a special type of message called control message will be used (some of the motivations are already discussed in KAFKA-1639). Control messages do not contain application data in the value payload and should not be exposed to applications. It is only used for internal communication between brokers and clients. For producer transactions, we will introduce a set of **transaction markers** implemented as control messages, such that the consumer client can interpret them to determine whether any given message has been committed or aborted. And based on the transaction status, the consumer client can then determine whether and when to return these messages to the application.

We will talk about how transaction markers are managed in the Broker and Consumer sections.

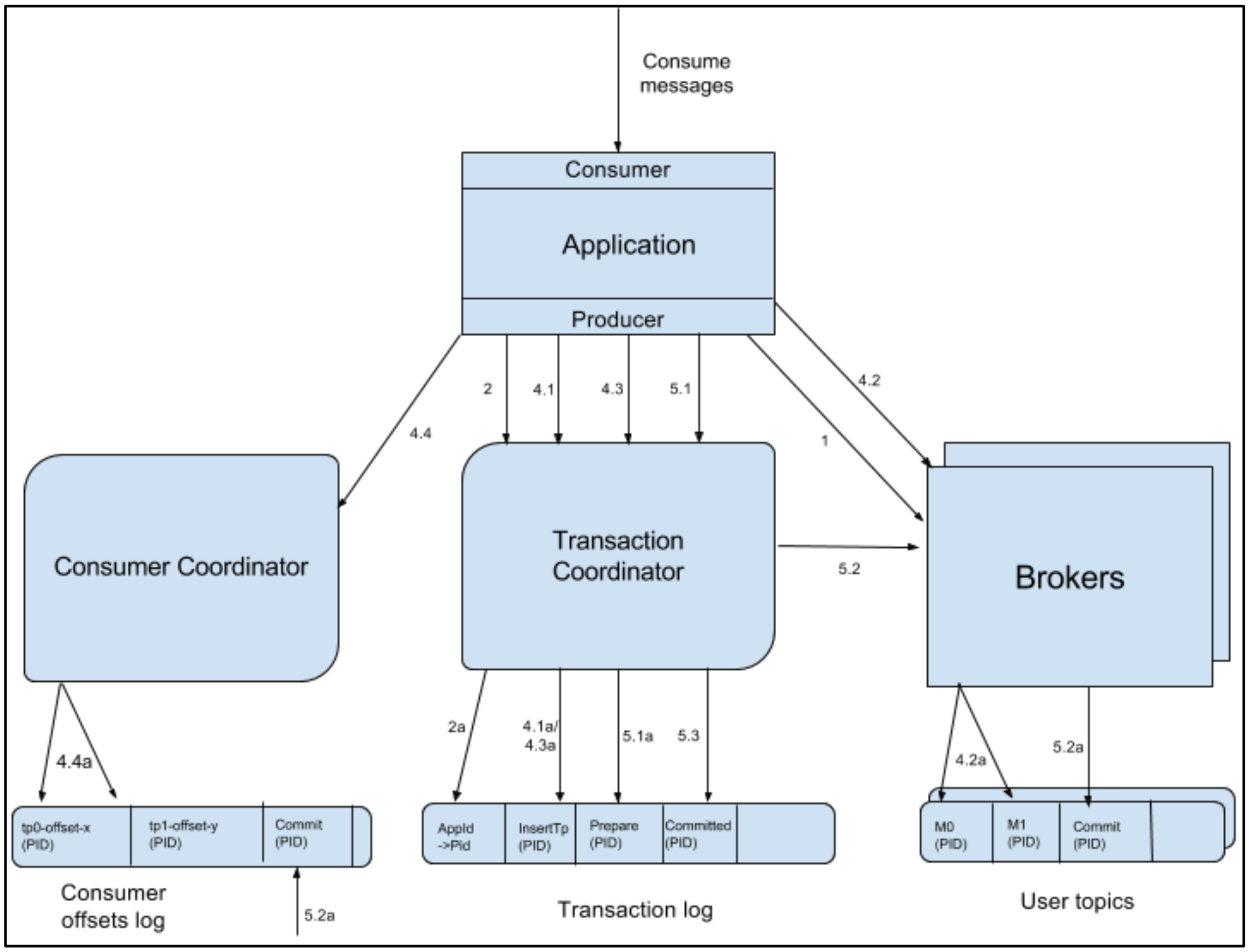
Producer Identifiers and Idempotency

Within a transaction, we also need to make sure that there is no duplicate messages generated by the producer. To achieve this, we are going to add **sequence numbers** to messages to allow the brokers to de-duplicate messages per producer and topic partition. For each topic partition that is written to, the producer maintains a sequence number counter and assigns the next number in the sequence for each new message. The broker verifies that the next message produced has been assigned the next number and otherwise returns an error. In addition, since the sequence number is per producer and topic partition, we also need to uniquely identify a producer across multiple sessions (i.e. when the producer fails and recreates, etc). Hence we introduce a new **TransactionalId** to distinguish producers, along with an **epoch number** so that zombie writers with the same TransactionalId can be fenced.

At any given point in time, a producer can only have one ongoing transaction, so we can distinguish messages that belong to different transactions by their respective TransactionalId. Producers with the same TransactionalId will talk to the same transaction coordinator which also keeps track of their TransactionalIds in addition to managing their transaction status.

We will talk about how transactional producers can interact with the transaction coordinators in the [Transactional Producer]() section.

Data Flow



In the diagram above, the sharp edged boxes represent distinct machines. The rounded boxes at the bottom represent Kafka topic partitions, and the diagonally rounded boxes represent logical entities which run inside brokers.

Each arrow represents either an RPC, or a write to a Kafka topic. These operations occur in the sequence indicated by the numbers next to each arrow. The sections below are numbered to match the operations in the diagram above, and describe the operation in question.

1. Finding a transaction coordinator -- the [FindCoordinatorRequest]()

Since the transaction coordinator is at the center assigning PIDs and managing transactions,the first thing a producer has to do is issue a *FindCoordinatorRequest* (previously known as *GroupCoordinatorRequest*, but renamed for general usage) to any broker to discover the location of its coordinator. Note that if no TransactionalId is specified in the configuration, this step can be skipped.

2. Getting a producer Id -- the InitPidRequest

The producer must send an InitPidRequest to get idempotent delivery or to use transactions. Which semantics are allowed depends on whether or not the transactional.id configuration is provided or not.

2.1 When a TransactionalId is specified

After discovering the location of its coordinator, the next step is to retrieve the producer's *PID*. This is achieved by sending an *InitPidRequest* to the transaction coordinator.

The TransactionalId is passed in the *InitPidRequest* along with the transaction timeout, and the mapping to the corresponding PID is logged in the transaction log in step 2a. This enables us to return the same PID for the TransactionalId to future instances of the producer, and hence enables recovering or aborting previously incomplete transactions.

In addition to returning the PID, the *InitPidRequest* performs the following tasks:

1. Bumps up the epoch of the PID, so that any previous zombie instance of the producer is fenced off and cannot move forward with its transaction.
2. Recovers (rolls forward or rolls back) any transaction left incomplete by the previous instance of the producer.

The handling of the *InitPidRequest* is synchronous. Once it returns, the producer can send data and start new transactions.

2.2 When a TransactionalId is not specified

If no TransactionalId is specified in the configuration, the *InitPidRequest* can be sent to any broker. A fresh PID is assigned, and the producer only enjoys idempotent semantics and transactional semantics within a single session.

3. Starting a Transaction -- the beginTransaction API

The new KafkaProducer will have a `beginTransaction()` method which has to be called to signal the start of a new transaction. The producer records local state indicating that the transaction has begun, but the transaction won't begin from the coordinator's perspective until the first record is sent.

4. The consume-transform-produce loop

In this stage, the producer begins to consume-transform-produce the messages that comprise the transaction. This is a long phase and is potentially comprised of multiple requests.

4.1 AddPartitionsToTxnRequest

The producer sends this request to the transaction coordinator the first time a new TopicPartition is written to as part of a transaction. The addition of this *TopicPartition* to the transaction is logged by the coordinator in step 4.1a. We need this information so that we can write the commit or abort markers to each TopicPartition (see section 5.2 for details). If this is the first partition added to the transaction, the coordinator will also start the transaction timer.

4.2 ProduceRequest

The producer writes a bunch of messages to the user's TopicPartitions through one or more *ProduceRequests* (fired from the *send* method of the producer). These requests include the PID, epoch, and sequence number as denoted in 4.2a.

4.3 AddOffsetsToTxnRequest

The producer has a new `sendOffsets` API method, which enables the batching of consumed and produced messages. This method takes a map of the offsets to commit and a *groupId* argument, which corresponds to the name of the associated consumer group.

The `sendOffsets` method sends an *AddOffsetsToTxnRequests* with the groupId to the transaction coordinator, from which it can deduce the TopicPartition for this consumer group in the internal `__consumer_offsets` topic. The transaction coordinator logs the addition of this topic partition to the transaction log in step 4.3a.

4.4 [TxnOffsetCommitRequest]

Also as part of `sendOffsets`, the producer will send a *TxnOffsetCommitRequest* to the consumer coordinator to persist the offsets in the `__consumer_offsets` topic (step 4.4a). The consumer coordinator validates that the producer is allowed to make this request (and is not a zombie) by using the PID and producer epoch which are sent as part of this request.

The consumed offsets are not visible externally until the transaction is committed, the process for which we will discuss now.

5. Committing or Aborting a Transaction

Once the data has been written, the user must call the new `commitTransaction` or `abortTransaction` methods of the `KafkaProducer`. These methods will begin the process of committing or aborting the transaction respectively.

5.1 [EndTxnRequest]

When a producer is finished with a transaction, the newly introduced `KafkaProducer.commitTranaction` or `KafkaProducer.abortTransaction` must be called. The former makes the data produced in step 4 above available to downstream consumers. The latter effectively erases the produced data from the log: it will never be accessible to the user (at the READ_COMMITTED isolation level), ie. downstream consumers will read and discard the aborted messages.

Regardless of which producer method is called, the producer issues an *EndTxnRequest* to the transaction coordinator, with a field indicating whether the transaction is to be committed or aborted. Upon receiving this request, the coordinator:

1. Writes a *PREPARE_COMMIT* or *PREPARE_ABORT* message to the transaction log. (step 5.1a)
2. Begins the process of writing the command messages known as COMMIT (or ABORT) markers to the user logs through the *WriteTxnMarkerRequest.* (see section 5.2 below).
3. Finally writes the *COMMITTED* (or *ABORTED)* message to transaction log. (see 5.3 below).

5.2 WriteTxnMarkerRequest

This request is issued by the transaction coordinator to the leader of each TopicPartition which is part of the transaction. Upon receiving this request, each broker will write a *COMMIT(PID)* or *ABORT(PID)* control message to the log. (step 5.2a)

This message indicates to consumers whether messages with the given PID should be delivered or dropped. As such, the broker will not return messages which have a PID (meaning these messages are part of a transaction) until it reads a corresponding *COMMIT* or *ABORT* message of that PID, at which point it will deliver or skip the messages respectively. In addition, in order to maintain offset ordering in message delivery, brokers would maintain an offset called last stable offset (LSO) below which all transactional messages have either been committed or aborted.

Note that, if the `__consumer_offsets` topic is one of the TopicPartitions in the transaction, the commit (or abort) marker is also written to the log, and the consumer coordinator is notified that it needs to materialize these offsets in the case of a commit or ignore them in the case of an abort (step 5.2a on the left).

5.3 Writing the final Commit or Abort Message

After all the commit or abort markers are written the data logs, the transaction coordinator writes the final *COMMITTED* or *ABORTED* message to the transaction log, indicating that the transaction is complete (step 5.3 in the diagram). At this point, most of the messages pertaining to the transaction in the transaction log can be removed.

We only need to retain the PID of the completed transaction along with a timestamp, so we can eventually remove the TransactionalId->PID mapping for the producer. See the Expiring PIDs section below.

In the rest of this design doc we will provide a detailed description of the above data flow along with the proposed changes on different modules.

Transactional Producer

Transactional Producer requires a user-provided **TransactionalId** during initialization in order to generate transactions. This guarantees atomicity within the transaction and at the same time fences duplicate messages from zombie writers as long as they are sending transactions.

Public APIs

We first introduce a set of new public APIs to the `KafkaProducer` class, and describe how these APIs will be implemented.

```
/* initialize the producer as a transactional producer */

initTransactions()
```

The following steps will be taken when `initTransactions()` is called:

1. If no TransactionalId has been provided in configuration, skip to step 3.
2. Send a `FindCoordinatorRequest` with the configured TransactionalId and with `CoordinatorType` encoded as "transaction" to a random broker. Block for the corresponding response, which will return the assigned transaction coordinator for this producer.
3. Send an `InitPidRequest` to the transaction coordinator or to a random broker if no TransactionalId was provided in configuration. Block for the corresponding response to get the returned PID.

```
/* start a transaction to produce messages */

beginTransaction()
```

The following steps are executed on the producer when `beginTransaction` is called:

1. Check if the producer is transactional (i.e. `init` has been called), if not throw an exception (we omit this step in the rest of the APIs, but they all need to execute it).
2. Check whether a transaction has already been started. If so, raise an exception.

```
/* send offsets for a given consumer group within this
transaction */

sendOffsetsToTransaction(

            Map<TopicPartition, OffsetAndMetadata> offsets,

            String consumerGroupId)
```

The following steps are executed on the producer when `sendOffsetsToTransaction` is called:

1. Check if it is currently within a transaction, if not throw an exception; otherwise proceed to the next step.
2. Check if this function has ever been called for the given `groupId` within this transaction. If not then send an `AddOffsetsToTxnRequest` to the transaction coordinator, block until the corresponding response is received; otherwise proceed to the next step.
3. Send a `TxnOffsetCommitRequest` to the coordinator return from the response in the previous step, block until the corresponding response is received.

```
/* commit the transaction with its produced messages */

commitTransaction()
```

The following steps are executed on the producer when `commitTransaction` is called:

1. Check if there is an active transaction, if not throw an exception; otherwise proceed to the next step.
2. Call `flush` to make sure all sent messages in this transactions are acknowledged.
3. Send an `EndTxnRequest` with `COMMIT` command to the transaction coordinator, block until the corresponding response is received.

```
/* abort the transaction with its produced messages */

abortTransaction()
```

The following steps are executed on the producer when `abortTransaction` is called:

1. Check if there is an active transaction, if not throw an exception; otherwise proceed to the next step.
2. Immediately fail and drop any buffered messages that are transactional. Await any in-flight messages which haven't been acknowledged.
3. Send an `EndTxnRequest` with `ABORT` command to the transaction coordinator, block until the corresponding response is received.

```
/* send a record within the transaction */
send(ProducerRecord<K, V> record)
```

With an ongoing transaction (i.e. after `beginTransaction` is called but before `commitTransaction` or `abortTransaction` is called), the producer will maintain the set of partitions it has produced to. When `send` is called, the following steps will be added:

1. Check if the producer has a PID. If not, send an `InitPidRequest` following the procedure above.
2. Check whether a transaction is ongoing. If so, check if the destination topic partition is in the list of produced partitions. If not, then send an `AddPartitionToTxnRequest` to the transaction coordinator. Block until the corresponding response is received, and update the set. This ensures that the coordinator knows which partitions have been included in the transaction before any data has been written.

Discussion on Thread Safety. The transactional producer can only have one outstanding transaction at any given time. A call to `beginTransaction()` with another ongoing transaction is treated as an error. Once a transaction begins, it is possible to use the `send()` API from multiple threads, but there must be one and only one subsequent call to `commitTransaction()` or `abortTransaction()`.

Note that with a non-transactional producer, the first `send` call will be blocking for two round trips (`GroupCoordinatorRequest` and `InitPidRequest`).

Error Handling

Transactional producer handles error codes returned from the transaction responses above differently:

`InvalidProducerEpoch`: this is a fatal error, meaning the producer itself is a zombie since another instance of the producer has been up and running, stop this producer and throw an exception.

`InvalidPidMapping`: the coordinator has no current PID mapping for this TransactionalId. Establish a new one via the `InitPidRequest` with the TransactionalId.

`NotCoordinatorForTransactionalId`: the coordinator is not assigned with the TransactionalId, try to re-discover the transaction coordinator from brokers via the `FindCoordinatorRequest` with the TransactionalId.

`InvalidTxnRequest`: the transaction protocol is violated, this should not happen with the correct client implementation; so if it ever happens it means your client implementation is wrong.

`CoordinatorNotAvailable`: the transaction coordinator is still initializing, just retry after backing off.

`DuplicateSequenceNumber`: the sequence number from `ProduceRequest` is lower than the expected sequence number. In this case, the messages are duplicates and hence the producer can ignore this error and proceed to the next messages queued to be sent.

`InvalidSequenceNumber`: this is a fatal error indicating the sequence number from `ProduceRequest` is larger than expected sequence number. Assuming a correct client, this should only happen if the broker loses data for the respective partition (i.e. log may have been truncated). Hence we should stop this producer and raise to the user as a fatal exception.

`InvalidTransactionTimeout`: fatal error sent from an InitPidRequest indicating that the timeout value passed by the producer is invalid (not within the allowable timeout range).

Discussion on Invalid Sequence. To reduce the likelihood of the `InvalidSequenceNumber` error code, users should have `acks=all` enabled on the producer and unclean leader election should be disabled. It is still possible in some disaster scenarios to lose data in the log. To continue producing in this case, applications must catch the exception and initialize a new producer instance.

Added Configurations

The following configs will be added to the producer client:

`enable.idempotence`	Whether or not idempotence is enabled (`false` by default). If disabled, the producer will not set the PID field in produce requests and the current producer delivery semantics will be in effect. Note that idempotence must be enabled in order to use transactions. When idempotence is enabled, we enforce that acks=all, retries > 1, and max.inflight.requests.per.connection=1. Without these values for these configurations, we cannot guarantee idempotence. If these settings are not explicitly overridden by the application, the producer will set acks=all, retries=Integer.MAX_VALUE, and max.inflight.requests.per.connection=1 when idempotence is

	enabled.
`transaction.timeout.ms`	The maximum amount of time in ms that the transaction coordinator will for a transaction to be completed by the client before proactively aborting the ongoing transaction. This config value will be sent to the transaction coordinator along with the `InitPidRequest`. Default is `60000`. This makes a transaction to not block downstream consumption more than a minute, which is generally allowable in real-time apps.
`transactional.id`	The TransactionalId to use for transactional delivery. This enables reliability semantics which span multiple producer sessions since it allows the client to guarantee that transactions using the same TransactionalId have been completed prior to starting any new transactions. If no TransactionalId is provided, then the producer is limited to idempotent delivery. Note that `enable.idempotence` must be enabled if a TransactionalId is configured. Default is “”.

Transaction Coordinator

Each broker will construct a transaction coordinator module during the initialization process. The transaction coordinator handles requests from the transactional producer to keep track of their **transaction status**, and at the same time maintain their **PIDs** across multiple sessions via client-provided **TransactionalIds**. The transaction coordinator maintains the following information in memory:

1. A map from TransactionalId to assigned PID**,** plus current epoch number, and 2) the transaction timeout value.

2. A map from PID to the current ongoing transaction status of the producer indicated by the PID, plus the participant topic-partitions, and the last time when this status was updated.

In addition, the transaction coordinator also persists both mappings to the transaction topic partitions it owns, so that they can be used for recovery.

Transaction Log

As mentioned in the summary, the transaction log is stored as an internal transaction topic partitioned among all the brokers. Log compaction is turned on by default on the transaction topic. Messages stored in this topic have versions for both the key and value fields:

```
/* Producer TransactionalId mapping message */

Key => Version TransactionalId

  Version => 0 (int16)

  TransactionalId => String

Value  =>  Version  ProducerId  ProducerEpoch  TxnTimeoutDuration
TxnStatus [TxnPartitions] TxnEntryLastUpdateTime TxnStartTime

  Version => 0 (int16)

  ProducerId => int64

  ProducerEpoch => int16

  TxnTimeoutDuration => int32

  TxnStatus => int8

  TxnPartitions => [Topic [Partition]]

     Topic => String

     Partition => int32

  TxnLastUpdateTime => int64

  TxnStartTime => int64
```

The status field above has the following possible values:

`BEGIN`	The transaction has started.
`PREPARE_COMMIT`	The transaction will be committed.
`PREPARE_ABORT`	The transaction will be aborted.
`COMPLETE_COMMIT`	The transaction was committed.
`COMPLETE_ABORT`	The transaction was aborted.

Writing of the `PREPARE_XX` transaction message can be treated as the synchronization point: once it is appended (and replicated) to the log, the transaction is guaranteed to be committed or aborted. And even when the coordinator fails, upon recovery, this transaction will be rolled forward or rolled back as well.

Writing of the TransactionalId message can be treated as persisting the creation or update of the `TransactionalId -> PID` entry. Note that if there are more than one transaction topic partitions owned by the transaction coordinator, the transaction messages are written only to the partition that the TransactionalId entry belongs to.

We will use the timestamp of the transaction status message in order to determine when the transaction has timed out using the transaction timeout from the `InitPidRequest` (which is stored in the TransactionalId mapping message). Once the difference between the current time and the timestamp from the status message exceeds the timeout, the transaction will be aborted.

This works similarly for expiration of the TransactionalId, but note 1) that the transactionalId will not be expired if there is an on-going transaction, and 2) if the client corresponding to a transactionalId has not begun any transactions, we use the timestamp from the mapping message for expiration.

When a transaction is completed (whether aborted or committed), the transaction state of the producer is changed to `Completed` and we clear the set of topic partitions associated with the completed transaction.

Transaction Coordinator Startup

Upon assignment of one of the transaction log partitions by the controller (i.e., upon getting elected as the leader of the partition), the coordinator will execute the following steps:

1. Read its currently assigned transaction topic partitions and bootstrap the Transaction status cache. The coordinator will scan the transaction log from the beginning, verify basic consistency, and materialize the entries. It performs the following actions as it reads the entries from the transaction log:

    a. Check whether there is a previous entry with the same TransactionalId and a higher epoch. If so, throw an exception. In particular, this indicates the log is corrupt. All future transactional RPCs to this coordintaor will result in a \`NotCoordinatorForTransactionalId\` error code, and this partition of the log will be effectively disabled.

    b. Update the transaction status cache for the transactionalId in question with the contents of the current log entry, including the last update time, and partitions in the transaction, and status. If there are multiple log entries with the same transactionalId, the last copy will be the one which remains materialized in the cache. The log cleaner will eventually compact out the older copies.

When committing a transaction, the following steps will be executed by the coordinator:

1. Send an `WriteTxnMarkerRequest` with the `COMMIT marker` to all the leaders of the transaction's added partitions.
2. When all the responses have been received, append a `COMPLETE_COMMIT` transaction message to the transaction topic. We do not need to wait for this record to be fully replicated since otherwise we will just redo this protocol again.

When aborting a transaction, the following steps will be executed by the coordinator:

1. Send an `WriteTxnMarkerRequest` with the `ABORT marker` to all the host brokers of the transaction partitions.
2. When all the responses have been received, append a `COMPLETE_ABORT` transaction message to the transaction topic. Do not need to wait for this record to be fully replicated since otherwise we will just redo this protocol again.

Discussion on Unavailable Partitions. When committing or aborting a transaction, if one of the partitions involved in the commit is unavailable, then the transaction will be unable to be completed. Concretely, say that we have appended a `PREPARE_COMMIT` message to the transaction log, and we are about to send the `WriteTxnMarkerRequest`, but one of the partitions is unavailable. We cannot complete the commit until the partition comes back online, at which point the "roll forward" logic will be executed again. This may cause a transaction to be delayed longer than the transaction timeout, but there is no alternative since consumers may be blocking awaiting the transaction's completion. **It is important to keep in mind that we strongly rely on partition availability for progress.** Note, however, that consumers in `READ_COMMITTED` mode will only be blocked from consumption on the unavailable partition; other partitions included in the transaction can be consumed before the transaction has finished rolling forward.

Transaction Coordinator Request Handling

When receiving the `InitPidRequest` from a producer *with a non-empty TransactionalId* (see here for handling the empty case), the following steps will be executed in order to send back the response:

1. Check if it is the assigned transaction coordinator for the TransactionalId, if not reply with the `NotCoordinatorForTransactionalId` error code.

2. If there is already an entry with the TransactionalId in the mapping, check whether there is an ongoing transaction for the PID. If there is and it has not been completed, then follow the abort logic. If the transaction has been prepared, but not completed, await its completion. We will only move to the next step after there is no incomplete transaction for the PID.

3. Increment the epoch number, append the updated `TransactionalId message`. If there is no entry with the TransactionalId in the mapping, construct a PID with the initialized epoch number; append an `TransactionalId message` into the transaction topic, insert into the mapping and reply with the PID / epoch / timestamp.

4. Respond with the latest PID and Epoch for the TransactionalId.

Note that coordinator's PID construction logic does NOT guarantee that it will always result in the same PID for a given TransactionalId (more details discussed here). In fact, in this design we make minimal assumptions about the PID returned from this API, other than that it is unique (across the Kafka cluster) and will never be assigned twice. One potential way to do this is to use Zookeeper to reserve blocks of the PID space on each coordinator. For example, when broker 0 is first initialized, it can reserve PIDs 0-100, while broker 1 can reserve 101-200. In this way, the broker can ensure that it provides unique PIDs without incurring too much additional overhead.

When receiving the `AddPartitionsToTxnRequest` from a producer, the following steps will be executed in order to send back the response.

1. If the TransactionalId does not exist in the TransactionalId mapping or if the mapped PID is different from that in the request, reply with `InvalidPidMapping`; otherwise proceed to next step.

2. If the PID's epoch number is different from the current TransactionalId PID mapping, reply with the `InvalidProducerEpoch` error code; otherwise proceed to next step.

3. Check if there is already an entry in the transaction status mapping.

    a. If there is already an entry in the transaction status mapping, check if its status is `BEGIN` and the epoch number is correct, if yes append an transaction status message into the transaction topic with the updated partition list, wait for this message to be replicated, update the transaction status entry and timestamp in the TransactionalId map and reply OK; otherwise reply with `InvalidTxnRequest` error code.

    b. Otherwise append a `BEGIN` transaction message into the transaction topic, wait for this message to be replicated and then insert it into the transaction status mapping and update the timestamp in the TransactionalId map and reply OK.

When receiving the `AddOffsetsToTxnRequest` from a producer, the following steps will be executed in order to send back the response.

1. If the TransactionalId does not exist in the TransactionalId mapping or if the mapped PID is different from that in the request, reply with `InvalidPidMapping`; otherwise proceed to next step.
2. If the PID's epoch number is different from the current TransactionalId mapping, reply with the `InvalidProducerEpoch` error code; otherwise proceed to next step.
3. If there is already an entry in the transaction status mapping, check if its status is `BEGIN` and the epoch number is correct, if yes calculate the internal offset topic partition based on the `ConsumerGroupID` field, append a `BEGIN transaction message` into the transaction topic with updated partition list, wait for this message to be replicated, update the transaction status entry and the timestamp in the TransactionalId map and reply OK with the calculated partition's lead broker as the consumer coordinator; otherwise reply with `InvalidTxnRequest` error code.
4. If there is no entry in the transaction status mapping reply with `InvalidTxnRequest` error code.

When receiving the `EndTxnRequest` from a producer, the following steps will be executed in order to send back the response.

1. If the TransactionalId does not exist in the TransactionalId mapping or if the mapped PID is different from that in the request, reply with `InvalidPidMapping`; otherwise proceed to next step.
2. Check if the PID's epoch number is correct against the TransactionalId mapping. If not, reply with the `InvalidProducerEpoch` error code; otherwise proceed to the next step.
3. If there is already an entry in the transaction status mapping, check its status
    a. If the status is `BEGIN`, go on to step 4.
    b. If the status is `COMPLETE_COMMIT` and the command from the `EndTxnRequest` is `COMMIT`, return OK.
    c. If the status is `COMPLETE_ABORT` and the command from the `EndTxnRequest` is `ABORT`, return OK.
    d. Otherwise, reply with `InvalidTxnRequest` error code.
4. Update the timestamp in the TransactionalId map.

5. Depending on the command field of the request, append a `PREPARE_XX transaction message` to the transaction topic with all the transaction partitions kept in the transaction status map, wait until the message is replicated.
6. Commit or abort the transaction following the procedure depending on the command field.
7. Reply OK.

Discussion on Coordinator Committing Transactions. The main motivation for having the transaction coordinator complete the commit / abort protocol after the `PREPARE_XXX` transaction message is appended to the transaction log is to keep the producer client thin (i.e. not letting producers to send the request to brokers to write transaction markers), and to ensure that transactions will always eventually be completed. However, it comes with an overhead of increased inter-broker communication traffic: suppose there are `N` producers sending messages in transactions, and each producer's transaction rate is `M/sec`, and each transaction touches `P` topic partitions on average, inter-broker communications will be increased by `M * N * P` round trips per sec. We need to conduct some system performance test to make sure this additional inter-broker traffic would not largely impact the broker cluster.

Discussion on Coordinator Failure During Transaction Completion: It is possible for the coordinator to fail at any time during the completion of a transaction. In general, the client responds by finding the new coordinator and retrying the `EndTxnRequest`. If the coordinator had already written the `PREPARE_COMMIT` or `PREPARE_ABORT` status to the transaction log, and had begun writing the corresponding markers to the data partitions, then the new coordinator may repeat some of this work (i.e. there may be duplicate COMMIT or ABORT markers in the log), but this is not a problem as long as no new transactions have been started by the same producer. It is also possible for the coordinator to fail after writing the `COMPLETE_COMMIT` or `COMPLETE_ABORT` status, but before the `EndTxnRequest` had returned to the user. In this case, the client will retry the `EndTxnRequest` after finding the new coordinator. As long as the command matches the completed state of the transaction after coordinator recovery, the coordinator will return a successful response. If not for this, there would be no way for the client to determine what happened to the transaction.

Coordinator-side Transaction Expiration

When a producer fails, its transaction coordinator should be able to pro-actively expire its ongoing transaction. In order to do so, the transaction coordinator will periodically trigger the following procedure:

1. Scan the transaction status map in memory. For each transaction:

a. If its status is `BEGIN`, and its corresponding expire timestamp is smaller than the current timestamp, pro-actively expire the transaction by doing the following:

   i. First void the PID by bumping up the epoch number in the TransactionalId map and writing a new TransactionalId message into the transaction log. Wait for it to be fully replicated.

   ii. Then rollback the transaction following the procedure **with the bumped up epoch number**, so that brokers can update their cached PID as well in order to fence Zombie writers (see more discussions below).

b. If its status is `PREPARE_COMMIT`, then complete the committing process of the transaction.

c. If its status is `PREPARE_ABORT`, then complete the aborting process of the transaction.

Coordinator TransactionalId Expiration

Ideally, we would like to keep TransactionalId entries in the mapping forever, but for practical purposes we want to evict the ones that are not used any longer to avoid having the mapping growing without bounds. Consequently, we need a mechanism to detect inactivity and evict the corresponding identifiers. In order to do so, the transaction coordinator will periodically trigger the following procedure:

1. Scan the TransactionalId map in memory. For each `TransactionalId -> PID` entry, if it does NOT have a current ongoing transaction in the transaction status map, AND the age of the last completed transaction is greater than the TransactionalId expiration config, remove the entry from the map. We will write the tombstone for the TransactionalId, but do not care if it fails, since in the worst case the TransactionalId will persist for a little longer (ie. the transactional.id.expration.ms duration).

Discussion on PID Expiration: It is possible for a producer to continue using the PID that its TransactionalId was mapped to in a non-transactional way even after the TransactionalId has been expired. If the producer continues writing to partitions without starting a new transaction, its PID will remain in the broker's sequence table as long as the messages are still present in the log. It is possible for another producer using the same TransactionalId to then acquire a new PID from the transaction coordinator and either begin using transactions or "idempotent mode." This does not violate any of the guarantees of either the idempotent or transactional producers.

1. For the transactional producer, we guarantee that there can be only one active producer at any time. Since we ensure that active transactions are completed before expiring an TransactionalId, we can guarantee that a zombie producer will be fenced when it tries to

start another one (whether or not a new producer with the same TransactionalId has generated a new PID mapping).

2. For the idempotent producer (i.e., producer that do not use transactional APIs), currently we do not make any cross-session guarantees in any case. In the future, we can extend this guarantee by having the producer to periodically send `InitPidRequest` to the transaction coordinator to keep the `TransactionalId` from expiring, which preserves the producer's zombie defence.

See below for more detail on how PID expiration works.

Added Broker Configurations

The following configs will be added to the broker:

`transactional.id.expiration.ms`	The maximum amount of time in ms that the transaction coordinator will wait before proactively expire a producer TransactionalId without receiving any transaction status updates from it. Default is `604800000` (7 days). This allows periodic weekly producer jobs to maintain its id.
`max.transaction.timeout.ms`	The maximum allowed timeout for transactions. If a client's requested transaction time exceeds this, then the broker will return an error in `InitPidRequest`. This prevents a client from too large of a timeout, which can stall consumers reading from topics included in the transaction. Default is `900000` (15 min). This is a conservative upper bound on the period of time a transaction of messages will need to be sent.
`transaction.state.log.min.isr`	The minimum number of insync replicas for the transaction state topic. Default: 2