Exactly Once Delivery and Transactional Messaging in Kafka

The definitive design.

Update History

2017.02.22

- Message key and value are no longer optional fields as indicated through attributes.

2017.02.15

- Removed `BeginTxnRequest` in favor of allowing the transaction to be started implicitly upon receiving the first `AddPartitionToTxnRequest`.
- Moved the transaction timeout that was previously passed in `BeginTxnRequest` into `InitPidRequest`.
- Re-added MessageSetSize field to `ProduceRequest`. Although it is not strictly necessary, it makes parsing the request much easier.
- Clarified the semantics of transactionalId expiration. It is based on the age of the most recent transaction, not on the time the InitPIDRequest was received.
- Updated the doc to indicate that there will be only one PID snapshot file. There was no good reason to keep two.
- Added a section on metrics.

2017.02.13

- Changed MaxTimestampDelta to MaxTimestamp in the MessageSet.
- Added TransactionTimeoutMs in BeginTxnRequest. Also added a new error : InvalidTransactionTimeout
- Added configurations for the transaction state log topic.

2017.02.07

- Remove message-level CRC and deprecate client checksum APIs.
- Individual message timestamps are variable length and relative to an initial timestamp stored at the message set.

2017.02.03

- Added TransactionalId to the `ProduceRequest` and added section to describe authorization of produce requests using the `ProducerTransactionalId` resource.
- The `AddPartitionsToTxn` API requires Topic Write permission for all topics included in the request.
- The `WriteTxnMarker` API requires ClusterAction permission on the Cluster.
- `GroupCoordinatorRequest` renamed to `FindCoordinatorRequest` since there is no group for transactional producers.

2017.02.01

- Changed AppID to TransactionalId.
- Added notes on client compatibility and third-party authorizers to migration notes.
- `InitPIDRequest` renamed to `InitPidRequest`.

2017.01.27

- Provided default values for added client / broker configs.

2017.01.26

- Added section on transaction coordinator authorization.
- `AddOffsetCommitsToTxn` RPC renamed to `AddOffsetsToTxn`.
- `UpdateTxn` RPC renamed to `WriteTxnMarker`.

2017.01.25

- Added a Future work section for feedback which should be incorporated in version 2.

2017.01.24

- Reworked the schema of transaction control messages so that we could include the coordinator epoch, which is needed to fence zombie transaction coordinators.
- Included the schema for the PID snapshot file.
- Added producer config to enable idempotence.

2017.01.20

- The `GroupType` field in `GroupCoordinatorRequest` was renamed to `CoordinatorType`.

- Added version 2 of `ListOffsetRequest` to give the client a way to find a partition's LSO in order to support `seekToEnd()` in `READ_COMMITTED`.
- LSO has been added to the `FetchResponse` to be able to compute consumer lag in `READ_COMMITTED`.

2017.01.19

- Incorporated abort index proposal.
- The `NoPidMapping` error code was changed to `InvalidPidMapping` to also express the case where the PID passed in a request is different from that stored in the transaction coordinator.
- Idempotent producers that do not need transactions can send `InitPidRequest` to any broker with a null TransactionalId in order to be assigned a PID.
- Slight modification to handling of `EndTxnRequest` for the edge case in which the coordinator fails after completing the transaction successfully, but before returning the response of the request to the client.

2017.01.17

- Added a discussion on TransactionalIds and PIDs in the rejected alternatives.

2017.01.05

- Added a motivation subsection about how this proposal is guided by the streams usecase for transactions.
- Updated the 'Transactional Guarantees' to include the guarantees for the consumer. Previously we only discussed guarantees for the producer.
- Changed the consumer config from 'fetch.mode', to 'isolation.level', to be more inline with database terminology. The values were correspondingly changed as well.
- Miscellaneous other changes based on KIP feedback.

2016.11.29

- Update transaction maker messages to use attribute fields for all controller flags.
- Incorporated motivation, guarantees, and data flow sections from the unified KIP doc.

2016.11.25

- Merge `TxnCoordinatorRequest` into `GroupCoordinatorRequest` with bumped up version and a newly added field.
- Update the `send` function procedure with non-transactional producer.

2016.11.23

- `GetPIDRequest` changed to `InitPidRequest` and handling is updated to ensure that pending transactions are completed before returning.
- `TransactionalId` was added to the transaction coordinator APIs. This makes it easier to check whether the client is contacting the right transaction coordinator.
- Changed `ProducerNotRecognized` error to `NoPIDMapping`.
- New client configuration `transactional.id` and the TransactionalId parameter was removed from the `initialize` API, which was renamed to `initTransactions`, to emphasize the fact that it is only needed when using transactions.

2016.11.22

- *Incorporated Idempotent Producer KIP doc, including the message format changes.*
- *Add `init` API to the producer.*

2016.11.21

- *Added `SendOffsets` API to the producer.*
- *Added `TxnOffsetCommitRequest` into the PRC protocol to not pollute the original `OffsetCommitRequest`.*
- *Augment `AddOffsetsToTxnResponse` with the consumer coordinator information.*

2016.11.19

- *Collapsed `CommitTxnRequest` and `AbortTxnRequest` with `UpdateTxnRequest`, rename `PrepareTxnRequest` to `EndTxnRequest`.*
- *Added a discussion section about fencing zombie writer WITHOUT requiring that same TransactionalIds always map to same PIDs.*
- *Updated transaction status message format on the transaction topic.*
- *Added producer-side exception handling description.*
- *Updated logic in TransactionalId expiration, and also updated `GetPIDRequest` with TransactionalId.*
- *Added a discussion section on committing offsets within a transaction.*
- *Added a discussion section on updating log compaction for transactional messages.*

This document serves to describe the detailed implementation design of KIP-98: Exactly Once Delivery and Transactional Messaging. Readers are highly recommended to read the KIP proposal itself before continue reading on this doc.

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.

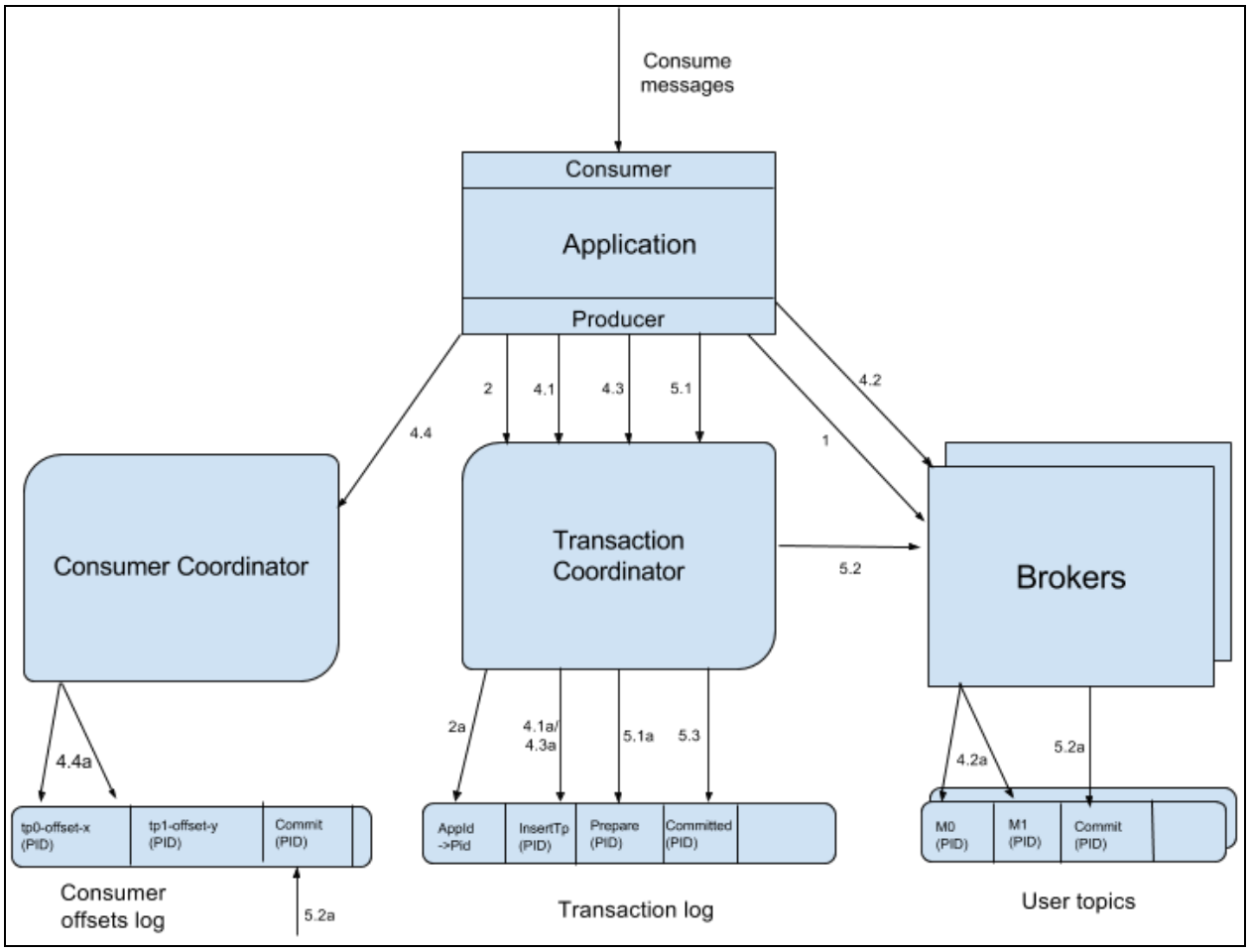
Producer Identifiers and Idempotency

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 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 consumer will buffer messages which have a PID until it reads a corresponding *COMMIT* or *ABORT* message, at which point it will deliver or drop the messages respectively.

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 likelihood of `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 overidden 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 of 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 an `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. 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 exceed 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
`transaction.state.log.replication.factor`	The number of replicas for the transaction state topic. Default: 3
`transaction.state.log.num.partitions`	The number of partitions for the transaction state topic. Default: 50

`transaction.state.log.segment.bytes`	The segment size for the transaction state topic. Default: 104857600 bytes.
`transaction.state.log.load.buffer.size`	The loading buffer size for the transaction stat topic. Default: 5242880 bytes.

Authorization

It is desirable to control access to the transaction log to ensure that clients cannot intentionally or unintentionally interfere with each other's transactions. In this work, we introduce a new resource type to represent the TransactionalId tied to transactional producers, and an associated error code for authorization failures.

```
case object ProducerTransactionalId extends ResourceType {
  val name = "ProducerTransactionalId"
  val errorCode = Errors.TRANSACTIONAL_ID_AUTHORIZATION_FAILED.code
}
```

The transaction coordinator handles each of the following requests: `InitPID`, `AddPartitionsToTxn`, `AddOffsetsToTxn`, and `EndTxn`. Each request to the transaction coordinator includes the producer's TransactionalId and can be used for authorization. Each of these requests mutates the transaction state of the producer, so they all require `Write` access to the corresponding `ProducerTransactionalId` resource. Additionally, the `AddPartitionsToTxn` API requires Write access to the topics corresponding to the included partitions, and the `AddOffsetsToTxn` API requires Read access to the group included in the request.

Discussion on limitations of coordinator authorization: Although we can control access to the transaction log using the TransactionalId, we cannot prevent a malicious producer from hijacking the PID of another producer and writing data to the log. This would allow the attacker to either insert bad data into an active transaction or to fence the authorized producer by forcing an epoch bump. It is not possible for the malicious producer to finish a transaction, however, because the brokers do not allow clients to write control messages. Note also that the malicious producer would have to have `Write` permission to the same set of topics used by the legitimate producer, so it is still possible to use topic ACLs combined with TransactionalId ACLs to protect sensitive topics. The brokers can verify the TransactionalId in produce requests, which ensures that the client has been authorized for transactions, but does not authorize the PID directly.

Future work can explore protecting the binding between TransactionalId and PID (e.g. through the use of message authentication codes such as in KIP-48).

Broker

Besides fencing duplicate messages and Zombie writers based on the PID, epoch and sequence number in the produce request as described in the Transactional Producer section, each broker must also handle requests sent from the transaction coordinators for writing the commit and abort markers into the log.

At the same time, brokers also need to handle requests from clients asking for their assigned coordinator, which will be the leader broker of the transaction topic's partition calculated from the producer's TransactionalId.

Transaction Markers

Transaction markers are written by the brokers themselves as control messages. As described in the Message Format section, we will refactor the on-disk message format by separating the message level and message-set level schema, and one of the changes is to use 2 bytes for the message set level attributes (previously it was only one byte). We use one bit from these attributes to indicate that the message set is transactional (i.e. it contains messages which are part of a transaction).

Compression (3)	Timestamp type (1)	**Transactional (1)**	Unused (11)

Transaction control messages are normal Kafka messages, but we use one bit in the message-level attributes to indicate that the message is a control message.

Control Flag (1)	Unused (7)

The type of the control message is packed into the message key. This proposal adds two control message types:

1. COMMIT (`ControlMessageType` = 0)
2. ABORT (`ControlMessageType` = 1)

The schema of the control message value field is specific to the control message type. Generally we assume this schema is opaque to clients. For the transaction markers, we use the following schema:

```
TransactionControlMessageValue => Version CoordinatorEpoch

  Version => int16

  CoordinatorEpoch => int32
```

The timestamp in these control messages will always be the log append time. However, this timestamp will not be indexed, and hence seeking by timestamp will ignore control messages.

Discussion on Coordinator Fencing: The transaction coordinator uses the `WriteTxnMarker` API to write the `COMMIT` and `ABORT` control messages to the leaders of partitions included in the transaction. It is possible for an unexpected pause on the coordinator to cause that request to be delivered after another transaction coordinator has been promoted (through partition leader election), and even after the pending transaction was completed and a new transaction begun. The partition leader must be able to detect this situation and reject the `WriteTxnMarker` request from the zombie coordinator or the current transaction could be inadvertently committed or aborted. This is the purpose of the `CoordinatorEpoch` field that we have included in the control messages. The value of this field corresponds to the leader epoch of the partition in the transaction log topic that the given producer was assigned to. Producers which were assigned different transaction coordinators can (and will) write to the same data partition, so coordinator epoch tracking is per-producer, as is shown in the section below on PID sequence management.

Last Stable Offset Tracking

We require the broker to track the *last stable offset* (LSO) for each partition. The LSO is defined as the latest offset such that the status of all transactional messages at lower offsets have been determined (i.e. committed or aborted). To do this, the broker must maintain in memory the set of active transactions along with their initial offsets. The LSO is always equal to the minimum of the initial offsets across all active transactions. This serves two purposes:

1. In the `READ_COMMITTED` isolation level, only offsets lower than the LSO are exposed to consumers.
2. The LSO and the initial offset of each transaction is needed when writing entries to the aborted transaction index (discussed in the next section).

Aborted Transaction Index

In addition to writing the `ABORT` and `COMMIT` control messages when receiving `WriteTxnMarker` requests from the transaction coordinator, each broker maintains a separate index to keep track of the aborted transactions. This facilitates the `READ_COMMITTED` isolation level for consumers.

Each log segment for a partition is given a separate append-only file to serve as an index for all transactions which were aborted in the corresponding log segment. This file is created lazily upon the first aborted transaction written to each log segment. We assume generally that aborted transactions are rare, so this file should stay small.

The schema for the entries in this index is the following:

```
TransactionEntry =>
  Version => int16
  PID => int64
  FirstOffset => int64
  LastOffset => int64
  LastStableOffset => int64
```

The LSO written in each entry is relative to the time that the entry was inserted. This allows the broker to efficiently scan through the aborted transactions corresponding to a fetch range in order to find the list of transactions from that range which were aborted. The first and last offset are used to filter the transactions which are actually needed for a given fetch response.

When log segments are deleted, so is the corresponding aborted transaction index.

Compacting Transactional Data

The presence of transaction markers creates two challenges for the log cleaner:

1. The log cleaner must be transaction-aware. Messages from aborted transactions should not cause the removal of any messages from committed transactions.

2. Care must be taken when removing transaction markers. If the messages from an aborted transaction is removed at or around the same time as the ABORT marker itself, it is possible for a consumer to see the aborted data, yet miss the marker.

Making the log cleaner transaction-aware is straightforward. Since we have the aborted transaction index, we can remove the messages from aborted transactions by following the entries from the index (in much the same way that a consumer would). The only restriction is that we cannot clean beyond the LSO. This is unlikely to be much of a restriction in practice since we usually expect the LSO to be contained in the active segment, which is not currently not cleaned.

The second issue is similar to a known bug in the log cleaner, and we propose to piggyback on top of the solution to address it.

PID-Sequence Management

For each topic partition, the broker will maintain a mapping in memory from the PID to the epoch, sequence number, the last offset successfully written to the log, and the coordinator epoch from each producer (for transactional producers). The source of truth for this information is always the log itself.

PID Snapshots

When a broker is restarted, it is possible to recreate the entire map of PIDs and corresponding sequence numbers by scanning the full log. However, it is more efficient to write periodic checkpoints to disk containing the state of the map at a given offset. We call this a *PID Snapshot*. When the broker resumes, it only needs to read the latest snapshot and scan the remainder of the log from the corresponding snapshot offset. This is similar to how the log cleaner works.

If for whatever reason, the snapshot file is deleted or the checkpointed offset is out of range, we can rebuild the map by scanning the full log. This can happen, for example, in a power failure scenario if the log has not been flushed to disk. To make it more likely that we can recover without requiring a full scan, we can keep several checkpoint files: if the last checkpointed offset is out of range, we can try one of the earlier ones.

The schema for the PID snapshot file is provided below:

```
PidSnapshot => Version CRC [PidEntry]

  Version => int16

  CRC => int32

  PidEntry => PID Epoch Sequence LastOffset
```

```
PID => int64

Epoch => int16

Sequence => int32

LastOffset => int64

CoordinatorEpoch => int32
```

The name of the snapshot files indicates the corresponding offset from the log that the snapshot was taken from. For example, “000000000050.pidsnapshot” indicates the snapshot was taken from offset 50 (in general, we will include 20 characters for the offset as is currently done with log segments).

In the interests of not scanning too far back in the log on startup or when log truncation happens during leader failover, we will create snapshots on a periodic basis. A reasonable period seems to be 30 to 60 seconds. We will maintain the two most recent snapshots so that in the worst case, we would have to scan at most 60 to 120 seconds of the log to rebuild the new Pid-Sequence mapping.

PID Expiration

It would be undesirable to let the PID-sequence map grew indefinitely, so we need a mechanism for **PID expiration**. We expire producerId’s when the age of the last message with that producerId exceeds the transactionalId expiration time or the topic’s retention time, whichever happens sooner. This rule applies even for non-transactional producers.

If the transactionalId expiration time is less than the topic’s retention time, then the producerId will be ‘logically’ expired. In particular, its information will not be materialized in the producerId->sequence mapping, but the messages with that producerId would remain in the log until they are eventually removed.

Discussion on PID expiration and Request Rejection. Suppose that a producer sends no writes to a topic partition for a long enough duration that its PID is expired by the leader of that partition. If it then attempts to send another write using the next expected sequence number, the broker must reject it with an error indicating the sequence number is out of range. If not, we risk violating deduplication guarantees since we cannot know if the new message is a duplicate of a message which was already deleted. This may cause “false negatives” since the broker could be unnecessarily rejecting the request, but users can control the risk of this through the topic’s retention settings.

Discussion on PID Expiration for Compacted Topics. PID expiration on compacted topics is a bit trickier since messages may be deleted in a different order than they were written by the producer. Hence we take a slightly different approach: before beginning a round of cleaning, we build a set of candidate PIDs for expiration. For example, we can populate the candidates with the PIDs which have not written to the latest log segment. As we iterate through the messages of the log, we remove entries from the candidate set as soon as we find any entry which will be retained. Once we reach the end, we know that any remaining PIDs can be removed as long as the last written offset is smaller than the next dirty offset.

The difficulty with this approach is that the amount of elapsed time before PID expiration is then nondeterministic. If a producer is "unlucky," its last message may be overwritten by another producer shortly after its own write. If this was its only entry in the log and we expire the PID, then the producer will receive an `InvalidSequenceNumber` error on its next produce attempt. We therefore need to ensure that the last sequence number for each producer is retained in the log for a deterministic amount of time.

To address this problem, we propose to preserve the last epoch and sequence number written by each producer. This is allowed by the new message format we are proposing in this document. As before, we retain this message until the transactionalId expiration timeout or the topic's retention time elapses, whichever happens sooner.

Coordinator-Broker request handling

When receiving the `WriteTxnMarkerRequest` from a transaction coordinator, the following steps will be executed to send back the response.

1. If the broker has a corresponding PID, verify that the received producer epoch is greater than or equal to the current epoch. If not, return `InvalidProducerEpoch`. Otherwise, insert the PID to its maintained list and update the epoch if necessary.

2. Append a `COMMIT` or `ABORT transaction marker` as a control message to each of the indicated partitions in the request, depending on the `Marker` field of the request. Wait for the message to be fully replicated and then reply OK.

3. If the `Marker` field is `COMMIT` and partition lists include the internal offset topic, then proceed to reflect the committed offsets into its consumer coordinator's cache.

4. If the `Marker` field is `ABORT`, add an entry to the aborted transaction index using the current LSO and the starting and ending offsets from the transaction.

5. Update the LSO for the partition if necessary (i.e. if the current LSO is one less than the initial offset of the completed transaction).

Discussion on Transaction-based Fencing. Upon failure recovery, a producer instance needs to fence off earlier instances before it resumes producing new data. It does it through the `InitPidRequest` as the epoch number returned in the response is strictly greater than any previous epochs that has been ever returned in other `InitPIDResponse` for this TransactionalId.

Now the question is, with TransactionalId expiration based on its producer's transaction activity, i.e., if the producer does not send any of the transaction requests to the coordinator for some period of time, then its `TransactionalId -> PID` will expire, so is zombie writer fencing still effective? The short answer is "yes" and the long answer goes in two folds:

1. After the producer has already lost its TransactionalId entry (and hence becomes a zombie), if the producer ever tries to start a new transaction, it must talk to the coordinator and hence will be notified that its PID is no longer valid. *If the producer does not ever start a transaction again, then it can continue sending messages with the old PID and this zombie writer is hence not fenced, which is OK according to our semantics.* We only guarantee that two producers sharing the same TransactionalId are not allowed to execute transactions concurrently.

2. If the zombie has an ongoing transaction at the time its TransactionalId mapping is expired, we know that the transaction would have be aborted first since we ensure that the TransactionalId expiration time is strictly larger than the transaction timeout. Hence, any future transactional requests will receive the fatal `ProducerFencedException`, and the producer will abort.

Client-Broker request handling

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

1. If the TransactionalId is provided, forward the request to the transaction coordinator module and follow the procedure here.

2. Assign a new PID to the producer and send a response.

When receiving the `GroupCoordinatorRequest` from a producer with the `CoordinatorType` as "transaction," the following steps will be executed to send back the response.

1. Compute the partition id based on the provided TransactionalId.

2. Check its metadata for the topic-partition's leader, if exist then return the broker information; otherwise reply `CoordinatorNotAvailable` error.

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

1. Check whether the produce request has a PID.
    a. Clients which have upgraded, but which are not yet using idempotent or transactional features will have an invalid PID and will go through the normal produce logic.
    b. If the request has a PID, check whether the topic supports the new message format. If not, respond with the `UNSUPPORTED_FOR_MESSAGE_FORMAT` error.
2. Check the sequence number map to determine if the PID is already present.
    a. If it is not, check sequence number.
        i. If it is 0, insert the PID, epoch, and sequence number into the PID mapping table and proceed to append the entries.
        ii. If the sequence is not zero, return the `InvalidSequenceNumber` error code.
    b. If the PID is present in the mapping, check the epoch.
        i. If the epoch is older than the current one, return `InvalidProducerEpoch` error code.
        ii. If the epoch matches the current epoch, check the sequence number:
            1. If it matches the current sequence number, allow the append.
            2. If the sequence number is older than the current sequence number, return `DuplicateSequenceNumber`.
            3. If it is newer, return `InvalidSequenceNumber`.
        iii. If the epoch is newer, check the sequence number. If it is 0, update the mapping and allow the append. If it is not 0, return `InvalidSequenceNumber`.

When receiving a `FetchRequest` from a consumer, the following steps will be executed to send back the response:

1. If the isolation level of the request is `READ_UNCOMMITTED`, follow the normal fetch flow. In the remaining steps, we assume `READ_COMMITTED`.

2. Fetch the data from the log according to existing logic. The initial offset is taken from the fetch request and we use the max fetch size to find the range of the log file to return in the fetch response.

3. Determine the range of offsets from the fetched data. We know the initial offset from the request, but we need to do some work to find the final offset. We propose to use the offset index to lookup an approximate upper bound for the last offset in the fetch range using the last byte position in the log file included in the fetched data.

4. Scan the aborted transaction indices of all log segments greater than or equal to the segment including the first fetched offset. Build a list of all aborted transactions which intersect the fetch range. We can stop when we have either reached an entry with an LSO greater than or equal to the last offset in the fetch, or when there are no more segment indices to scan.

5. Return the fetched data and the list of aborted transactions to the client.

Authorization

From a security perspective, there are three main points we need to address:

- Authorize clients to produce to a topic (including the writing of control messages)
- Authorize clients to consume from a topic
- Authorize clients to access the transaction log (via coordinator only)

We can address the first two points primarily by relying on ACLs that grant read/write access to topics. There is no change required to the security framework in the case we use read/write operations with the topic resource.

However, we will introduce additional authorization to produce transactional data. This can be used to minimize the risk of an “endless transaction attack,” in which a malicious producer writes transactional data without corresponding `COMMIT` or `ABORT` markers in order to prevent the LSO from advancing and consumers from making progress. We can use the `ProducerTransactionalId` resource introduced above to ensure that the producer is authorized to write transactional data. The producer’s `TransactionalId` is also included in the `ProduceRequest` schema.

Additionally, since the `WriteTxnMarker` API should only be allowed if the request came from another broker, the client must have `ClusterAction` authorization on the `Cluster` resource.

This is the same way that other inter-broker APIs are protected. Also note that this is the only API through which a control message can be added to the log. An attempt to do so through the Produce API will be rejected.

For the transaction log, we can also use ACLs that grant read/write access to the transaction log, although in this case access is through the coordinator and the coordinator is responsible for authorizing access through authorization of the `TransactionalId`. Clients will not be allowed to write directly to the transaction log using the Produce API, though it is useful to make it accessible to consumers with Read permission for the purpose of debugging.

In addition, this design protocol does not prevent a malicious producer client from "hijacking" another producer's PID and hence its transactions (since the PID itself is a random 64-bit, in practice it should be very rare), as long as it is authorized to use transaction and / or write to one of the corresponding data partitions. We leave this for future work.

Consumer Coordinator

As mentioned in the summary, many Kafka streaming applications need to both consume from input topics and produce to output topics at the same time. When consumer offsets are committed for the input topics, they need to be done along with the produced transactions as well, such that *for each message consumed from the input Kafka topics, the result message(s) of processing this message will be reflected in the output Kafka topics exactly once, even under failures*.

In order to support this scenario, we need to make the consumer coordinator transaction-aware. More specifically, we need a new API which allows the producer to send offset commits as part of a transaction. For this we introduce the `TxnOffsetCommitRequest` API.

Client-Coordinator request handling

When receiving the `TxnOffsetCommitRequest` from a producer, the following steps will be added before proceed to the checking consumer group metadata.

1. Skip the check for whether or not the consumer is valid for the current generation id and the member list (the request does actually not contain this information), i.e. blindly accept the request and proceed to the next step directly.

2. Check if the specified `PID` is recognized and the `Epoch` is correct, if not reply with `InvalidProducerEpoch`; otherwise proceed to the next step.

3. Append to the offset topic, but skip updating the offset cache in the delayed produce callback, until a `WriteTxnMarkerRequest` from the transaction coordinator is received including the offset topic partitions.

Discussion on NOT Checking Consumer Status for Transactional Offset Commits. Note that in the `TxnOffsetCommitRequest` we do not include the consumer group memberID and the generationID fields, and hence the consumer coordinator cannot validate this information upon receiving the request as we do in the `OffsetCommitRequest`. This is not necessary because if a rebalance has happened and this consumer instance becomes a zombie, even if this offset message is appended in the offset topic, the transaction will be rejected later on when it tries to commit the transaction via the `EndTxnRequest`.

Consumer Coordinator Startup

In addition, when initializing the consumer coordinator by bootstrapping from the offset topic, we also need to change the procedure as follows:

1. For each consumer offset message read from the offset topic, check if `PID` and `Epoch` fields are specified, if yes hold it from putting into the cache.

2. For each control message read from the offset topic, if it is a `COMMIT` transaction marker then put the previously kept offset entry into the cache; if it is an `ABORT` transaction maker then forget the previously kept offset entry.

Discussion on Reusing Offset Topics within Transactions. An alternative approach to enable applications to commit input topic offsets as part of an output transaction is to use a separate topic. More specifically, suppose an application task that consumes from a topic T_C, processes the input messages, and produces to a topic T_P. The application task can use a separate topic, e.g., T_{io} , to store the input offsets. The application persists the consumed offsets in this way so that the output messages and messages containing the input offsets are produced as a transaction of messages atomically: the output messages are successfully produced to T_p iff the input offset messages are successfully produced (committed) to T_{io}. There are pros and cons for this alternative approaches regarding implementation complexity, operations / tooling, and security, etc, which is summarized in this doc. We decided to reuse the offset topic primarily for its tooling and operation simplicity.

Consumer

Consumers need to recognize the transaction marker control messages when fetching from the brokers, and based on its configuration parameter to determine whether it wants to return committed messages only or all messages (no matter if they are already committed or not). In either case, messages are always delivered in offset order.

Added Configurations and Consume Protocol

The following configs will be added to the `KafkaConsumer` client:

`isolation.level`	Here are the possible values: `read_uncommitted`: consume all available messages in offset ordering. This is the default value. `read_committed`: only consume non-transactional messages or transactional messages that are already committed, in offset ordering. Default: `read_uncommitted`

The isolation level is passed to the brokers in the `FetchRequest`. For `READ_UNCOMMITTED`, the consumer uses the same fetch logic as previous versions of Kafka. For `READ_COMMITTED`, the consumer must do some extra work to filter aborted transactions.

Recall that the fetch response contains a list of the aborted transactions which intersect with the fetch range. Any transactions included in a fetch which are not among the aborted transactions are assumed to be committed. The broker will not return records from any offsets lower than the current LSO in `READ_COMMITTED`, so we are guaranteed that all transactions received in the fetch have been completed.

Initially, the aborted transactions from the fetch response are arranged in a minheap, which is ordered by their initial offsets. Separately, we maintain a set of the PIDs which have aborted transaction ranges which contain the consumer's current position. The logic to follow when processing the fetched data is as follows:

1. If the message is a transaction control message, and the status is `ABORT`, then remove the corresponding PID from the set of PIDs with active aborted transactions. If the status is `COMMIT`, ignore the message.

2. If the message is a normal message, compare the offset and PID with the head of the aborted transaction minheap. If the PID matches and the offset is greater than or equal to the corresponding initial offset from the aborted transaction entry, remove the head from the minheap and insert the PID into the set of PIDs with aborted transactions.

3. Check whether the PID is contained in the aborted transaction set. If so, discard the record set; otherwise, add it to the records to be returned to the user.

When seeking to a new offset, the consumer will clear the transaction state for that partition. It can always rebuild it after the next fetch response is received.

Discussion on Seeks. The isolation level will be used to determine the behavior of seeks. In `READ_UNCOMMITTED`, the seek API behaves exactly as it does currently. In `READ_COMMITTED`, the only difference is that the consumer is not allowed to seek beyond the log's LSO. In other words, a call to `seekToEnd()` will result in the offset being set to the LSO instead of the log end offset. For this, we have updated the `ListOffsets` API to support retrieval of the LSO.

Discussion on Pro-active Transaction Timeout. One motivation to let transaction coordinator to pro-actively timeout transactions is that upon producer failure, we do not want to rely on the producer eventually recovering and completing the transaction: for example, if a producer fails within a transaction and the coordinator does not pro-actively abort it, this transaction will become a "dangling" transaction that will not be completed until the producer resumes with the same TransactionalId, and any consumers fetching on the partitions included in this transaction in `READ_COMMITTED` will be effectively blocked waiting for the LSO to advance. This issue will be more severe if one topic partition has multiple transactional producers writing to it in an interleaving manner, since one dangling transaction will cause all other transactions to not be able to be consumed due to the offset ordering.

One question though, is whether a pro-active timeout of a transaction will still fence a zombie writer. The short answer is yes. Upon timing out and aborting the transaction, the coordinator will bump the epoch associated with the PID and write `ABORT` markers to all partitions which had been included in the transaction. If the zombie is still attempting to write to any of these partitions, it will be fenced as soon as the `ABORT` marker is written. Alternatively, if it attempts to commit or abort the transaction, it will also be fenced by the coordinator.

Discussion on Transaction Ordering. In this design, we are assuming that the consumer delivers messages in offset order to preserve the behavior that Kafka users currently expect. A different way is to deliver messages in "transaction order": as the consumer fetches commit markers, it enables the corresponding messages to be consumed. For example, say that we have two TransactionalIds P_1 and P_2 both producing to the same given topic partition and the messages of these two TransactionalIds are interleaved in the log. A consumer of this partition allocates a buffer for the messages of P_1 and another for the messages of P_2. Upon receiving a commit marker for P_2, the consumer delivers the buffered messages of P_2 immediately. If it receives a commit marker for P_1 while delivering the messages of P_2, then it queues the messages of P_1 to be delivered after the ones of P_2.

One main advantage of this approach is that it delivers messages as soon as the commit marker is processed, and hence improves latency. The main reason for stepping away from this option is offset management: because consumed messages do not follow offset order, an application that manually manages offsets needs to store more information than just the last offset consumed. The application needs to at least keep the first missing message and all other messages it has already processed; when automating offset commits for commit order delivery, we must track the offset of the last commit marker processed. Upon resuming a consumer for the partition, we need to start processing commit markers from the last persisted, and for every new commit marker the consumer processes, it needs to scan the logs backwards to find the messages to deliver, assuming they are not already buffered. If they are already buffered, then it is just a matter of delivering. To illustrated the worst case scenario, say we have the following content in the log:

C_2, M_3, C_1, M_2, M_1

- M_1 is the head (offset 0)
- M_1 is associated to commit C_1
- M_2 and M_3 are associated to commit C_2

Say that a consumer reads C_1, M_2, M_1 and delivers M_1 after processing C_1. The consumer commits the offset for C_1, which is 2. The consumer crashes and some other consumer in the group picks up the partition. The new consumer resumes from offset 2, it reads C_2 and M_3, but once it processes the commit marker, it does not know what other messages from that transaction it is missing. In particular, it is missing M_2 and it needs to scan backwards from C1 to find M_2. Once it finds M_2, it continues to scan backwards depending on the commit sequence number of the message: if it is greater than zero, then it needs to keep scanning.

It is important to note that this behavior can be implemented at a later time if we determine that it is useful and critical for the success of the feature. For the moment, given that we have no strong evidence that applications can benefit from the commit order option, we opted for not implementing it.

Message Format

In order to add new fields such as PID and epoch into the produced messages for transactional messaging and de-duplication, we need to change Kafka's message format and bump up its version (i.e. the "magic byte"). More specifically, we need to add the following fields into each message:

- `PID => int64`
- `Epoch => int16`
- `Sequence number => int32`

Adding these fields on the message-level format schema potentially adds a considerable amount of overhead; on the other hand, at least the PID and epoch will never change within a set of messages from a given producer. We therefore propose to enhance the current concept of a **message set** by giving it a separate schema from an individual message. In this way, we can locate these fields only at the message set level which allows the additional overhead to be amortized across batches of messages rather than paying the cost for each message separately.

Both the epoch and sequence number will wrap around once int16_max and int32_max are reached. Since there is a single point of allocation and validation for both the epoch and sequence number, wrapping these values will not break either the idempotent or transactional semantics.

For reference, the current message format (v1) is the following:

```
MessageSet => [Offset MessageSize Message]

  Offset => int64

  MessageSize => int32


Message => Crc Magic Attributes Timestamp Key Value

  Crc => int32

  Magic => int8
```

```
Attributes => int8

Timestamp => int64

Key => bytes

Value => bytes
```

A message set is a sequence of messages. To support compression, we currently play a trick with this format and allow the compressed output of a message set to be embedded in the value field of another message (a.k.a., the “wrapper message”). In this design, we propose to extend this concept to non-compressed messages and to decouple the schema for the message wrapper (which contains the compressed message set). This allows us to maintain a separate set of fields at the message set level and avoid some costly redundancy (old fields in blue, changes in red):

```
MessageSet =>
```

- `FirstOffset => int64`
- `Length => int32`
- `CRC => int32`
- `Magic => int8  /* bump up to “2” */`
- `Attributes => int16`
- `LastOffsetDelta => int32`
- `FirstTimestamp => int64`
- `MaxTimestamp => int64`
- `PID => int64`
- `Epoch => int16`
- `FirstSequence => int32`
- `Messages => Message1, Message2, … , MessageN`

```
Message =>
```

- `Length => varint`
- `Attributes => int8`
- `TimestampDelta => varint`
- `OffsetDelta => varint`
- `KeyLen => varint`
- `Key => data`
- `ValueLen => varint`
- `Value => data`

The ability to store some fields only at the message set level allows us to conserve space considerably when batching messages into a message set. For example, there is no need to write the PID within each message since it will always be the same for all messages within each message set. In addition, by separating the message level format and message set format, now we can also use variable-length types for the inner (relative) offsets and save considerably over a fixed 8-byte field size. Note also that there is only a single CRC computed for the full message set.

Message Set Fields

The first four fields of a message set in this format *must to* be the same as the existing format because any fields before the magic byte cannot be changed in order to provide a path for upgrades following a similar approach as was used in KIP-32. Clients which request an older version of the format will require conversion on the broker.

The offset provided in the message set header represents the offset **of the first message in the set**. Similarly, we the sequence number field represents the sequence number of the first message. We also include an “offset delta” at the message set level to provide an easy way to compute the last offset / sequence number in the set: i.e. the starting offset of the next message set should be “offset + offset delta”. This also allows us to search for the message set corresponding to a particular offset without scanning the individual messages, which may or may not be compressed. Similarly, we can use this to easily compute the next expected sequence number.

The message set also includes the timestamp of the first message included in the message set. All message timestamps are computed relative to this timestamp (note that the timestamp delta

in each message is a signed integer since timestamps are not assumed to be monotonic). The max timestamp, which is needed for indexing, can be derived by summing the initial timestamp and the `MaxTimestampDelta` field, which is exposed in the message set.

The offset, sequence number, and offset delta values of the message set *never* change after the creation of the message set. The log cleaner may remove individual messages from the message set, and it may remove the message set itself once all messages have been removed, but we must preserve the range of sequence numbers that were ever used in a message set since we depend on this to determine the next sequence number expected for each PID.

Message Set Attributes: The message set attributes are essentially the same as in the existing format, though we have added an additional byte for future use. In addition to the existing 3 bits used to indicate the compression codec and 1 bit for timestamp type, we will use another bit to indicate that the message set is transactional (see Transaction Markers section). This lets consumers in READ_COMMITTED know whether a transaction marker is expected for a given message set.

Compression (3)	Timestamp type (1)	Transactional (1)	Unused (11)

Discussion on Maximum Message Size. The broker's configuration `max.message.size` previously controlled the maximum size of a single uncompressed message or a compressed set of messages. With this design, it now controls the maximum message set size, compressed or not. In practice, the difference is minor because a single message can be written as a singleton message set, with the small increase in overhead mentioned above.

Message Fields

The length field of the message format is encoded as an unsigned variable-length int, abbr. "uintVar". Similarly the offset delta and key length fields are encoded as unitVar as well. The message's offset can then be calculated as the offset of the message set + offset delta. At the end we still maintains a message-level CRC (reason discussed below).

Message Attributes: In this format, we have also added a single byte for individual message attributes. Only message sets can be compressed, so there is no need to reserve some of these attributes for the compression type. The control flag indicates that the message is a control

message, which means it is not intended for application consumption. The remaining bits are currently unused, though one could be used for KIP-87 (message tombstone marker).

Control Flag (1)	Unused (7)

Control messages will always have a non-null key, which is used to indicate the type of **control message type** with the following schema:

```
ControlMessageKey => Version ControlMessageType

  Version => int16

  ControlMessageType => int16
```

In this proposal, a control message type of 0 indicates a `COMMIT` marker, and a control message type of 1 indicates an `ABORT` marker. The schema for control values is generally specific to the control message type.

Discussion on Message-level Schema. A few additional notes about this schema:

1. Having easy access to the offset of the first message allows us to stream messages to the user on demand. In the existing format, we only know the last offset in each message set, so we have to read the messages fully into memory in order to compute the offset of the first message to be returned to the user.

2. As before, the message set header has a fixed size. This is important because it allows us to do in-place offset/timestamp assignment on the broker before writing to disk.

3. We have removed the per-message CRC in this format. We hesitated initially to do so because of its use in some auditing applications for end-to-end validation. The problem is that it is not safe, even currently, to assume that the CRC seen by the producer will match that seen by the consumer. One case where it is not preserved is when the topic is configured to use the log append time. Another is when messages need to be up-converted prior to appending to the log. For these reasons, and to conserve space and save computation, we have removed the CRC and deprecated client usage of these fields.

4. The message set CRC covers the header and message data. Alternatively, we could let it cover only the header, but if compressed data is corrupted, then decompression may

fail with obscure errors. Additionally, that would require us to add the message-level CRC back to the message.

5. Individual messages within a message set have their full size (including header, key, and value) as the first field. This is designed to make deserialization efficient. As we do for the message set itself, we can read the size from the input stream, allocate memory accordingly, and do a single read up to the end of the message. This also makes it easier to skip over the messages if we are looking for a particular one, which potentially saves us from copying the key and value.

6. We have not included a field for the size of the value in the message schema since it can be computed directly using the message size and the length of the header and key.

7. We have used a variable length integer to represent timestamps. Our approach is to let the first message

Space Comparison

As the batch size increases, the overhead of the new format grows smaller compared to the old format because of the eliminated redundancy. The overhead per message in the old format is fixed at 34 bytes. For the new format, the message set overhead is 53 bytes, while per-message overhead ranges from 6 to 25 bytes. This makes it more costly to send individual messages, but space is quickly recovered with even modest batching. For example, assuming a fixed message size of 1K with 100 byte keys and reasonably close timestamps, the overhead increases by only 7 bytes for each additional batched message (2 bytes for the message size, 1 byte for attributes, 2 bytes for timestamp delta, 1 byte for offset delta, and 1 byte for key size) :

Batch Size	Old Format Overhead	New Format Overhead
1	34*1 = 34	53 + 1*7 = 60
3	34*3 = 102	53 + 3*7 = 74
10	34*10 = 340	53 + 10*7 = 123
50	34*50 = 1700	53 + 50*7 = 403

100	34*100 = 3400	45 + 100*7 = 745

Client API Changes

As noted above, we have removed the individual message CRC computation from the message format. This is currently exposed in the client API in `RecordMetadata` on the producer and in `ConsumerRecord` on the consumer. We intend to deprecate these APIs:

```
class RecordMetadata {
  @Deprecated
  long checksum();
}

class ConsumerRecord {
  @Deprecated
  long checksum();
}
```

Until these methods are removed, we should continue to support these APIs. We propose the following:

1. For old magic values (version 0 and 1), the checksum will continue to return the CRC from the message itself.
2. For the new version, we will compute a checksum on-demand using the message key, value and timestamp.

Migration Plan

We follow the same approach used in KIP-32. To upgrade from a previous message format version, users should:

1. Upgrade the brokers once with the inter-broker protocol set to the previous deployed version.
2. Upgrade the brokers again with an updated inter-broker protocol, but leaving the message format unchanged.
3. Upgrade all or most clients, but continue with idempotence/transactions disabled. Clients which attempt to use idempotent or transactional features will be rejected by the broker.

4. Restart the brokers, with the message format version set to the latest. It is also possible to enable the new message format on topics individually.

5. Once the message format is upgraded, clients can enable idempotence/transactions production.

The reason for step 3 is to avoid the performance cost for down-converting messages to an older format, which effectively loses the “zero-copy” optimization. Ideally, all consumers are upgraded before the producers even begin writing to the new message format.

Client Compatibility

In KIP-97, the Java client enabled support for compatibility across different version of the broker. Below we describe the behavior for several important cases:

1. Since we were only supporting versions later than 0.10.0.0, we have so far only needed to deal with one message format version. After this KIP, the producer will need to support magic version 1 (the current version), and magic version 2 (proposed here). The version to use will depend on the version of the `ProduceRequest` that the broker supports. A challenge here is that we need to know the supported version at the time record accumulation begins. Our initial plan to address this is to connect to the partition leader prior to beginning accumulation for that partition.

2. Old versions of the broker will obviously not support idempotent producing or transactions. If either of these facilities are enabled and we encounter a broker which does not support them, an exception will be raised to the user.

3. When fetching from an older broker (with an older message format), `READ_COMMITED` behaves exactly the same as `READ_UNCOMMITTED`. The behavior only differs for messages which have the transactional bit set.

4. When in `READ_COMMITTED` mode, the behavior of `seekToEnd` is modified to seek to the LSO. For older brokers which do not support the new `ListOffsetRequest` version (which allows fetching the LSO), we will retain the current behavior of seeking to the log end offset.

Third-party Authorizers

The addition of the ProducerTransactionalId resource means that third-party authorizers will need to be updated before requests to the transaction coordinator can be protected. In general

we should emphasize that authorizers should reject authorization requests for unknown resource types.

RPC Protocol Summary

We summarize all the new request / response pairs as well as modified requests in this section.

FetchRequest

Sent by the consumer to any partition leaders to fetch messages. We bump the API version to allow the consumer to specify the required isolation level. We also modify the response schema to include the list of aborted transactions included in the range of fetched messages.

```
// FetchRequest v4
FetchRequest => ReplicaId MaxWaitTime MinBytes MaxBytes IsolationLevel
[TopicName [Partition FetchOffset MaxBytes]]
  ReplicaId => int32
  MaxWaitTime => int32
  MinBytes => int32
  IsolationLevel => int8 (READ_COMMITTED | READ_UNCOMMITTED)
  TopicName => string
  Partition => int32
  FetchOffset => int64
  MaxBytes => int32



// FetchResponse v4
FetchResponse => ThrottleTime [TopicName [Partition ErrorCode
HighwaterMarkOffset LastStableOffset AbortedTransactions MessageSetSize
MessageSet]]
  ThrottleTime => int32
  TopicName => string
  Partition => int32
  ErrorCode => int16
  HighwaterMarkOffset => int64
  LastStableOffset => int64
  AbortedTransactions => [PID FirstOffset]
    PID => int64
    FirstOffset => int64
  MessageSetSize => int32
```

When the consumer sends a request for an older version, the broker assumes the READ_UNCOMMITTED isolation level and converts the message set to the appropriate format before sending back the response. Hence zero-copy cannot be used. This conversion can be costly when compression is enabled, so it is important to update the client as soon as possible.

We have also added the LSO to the fetch response. In READ_COMMMITED, the consumer will use this to compute lag instead of the high watermark.

ProduceRequest

Sent by the producer to any brokers to produce messages. Instead of allowing the protocol to send multiple message sets for each partition, we change the handling of this request to only allow one message set for each partition. As long as there is only one message set to be written to the log, partial produce failures are no longer possible. The full message set is either successfully written to the log (and replicated) or it is not.

We include the TransactionalId in order to ensure that producers using transactional messages (i.e. those with the transaction bit set in the attributes) are authorized to do so. If the client is not using transactions, this field should be null.

```
// ProduceRequest v3

ProduceRequest => TransactionalId

                  RequiredAcks

                  Timeout

                  [TopicName [Partition MessageSetSize MessageSet]]

  TransactionalId => nullableString

  RequiredAcks => int16

  Timeout => int32

  Partition => int32

  MessageSetSize => int32

  MessageSet => bytes
```

```
// ProduceResponse v3

ProduceResponse => [TopicName [Partition ErrorCode Offset Timestamp]]

                   ThrottleTime

  TopicName => string

  Partition => int32

  ErrorCode => int16

  Offset => int64

  Timestamp => int64

  ThrottleTime => int32
```

Error code:

- DuplicateSequenceNumber [NEW]
- InvalidSequenceNumber [NEW]
- InvalidProducerEpoch [NEW]
- UNSUPPORTED_FOR_MESSAGE_FORMAT

Note that clients sending version 3 of the produce request MUST use the new message set format. The broker may still down-convert the message to an older format when writing to the log, depending on the internal message format specified.

ListOffsetRequest

Sent by the client to search offsets by timestamp and to find the first and last offsets for a partition. In this proposal, we modify this request to also support retrieval of the last stable offset, which is needed by the consumer to implement `seekToEnd()` in `READ_COMMITTED` mode.

```
// v2
ListOffsetRequest => ReplicaId [TopicName [Partition Time]]
  ReplicaId => int32
  TopicName => string
  Partition => int32
  Time => int64

ListOffsetResponse => [TopicName [PartitionOffsets]]
  PartitionOffsets => Partition ErrorCode Timestamp [Offset]
  Partition => int32
  ErrorCode => int16
  Timestamp => int64
  Offset => int64
```

The schema is exactly the same as version 1, but we now support a new sentinel timestamp in the request (-3) to retrieve the LSO.

FindCoordinatorRequest

Sent by client to any broker to find the corresponding coordinator. This is the same API that was previously used to find the group coordinator, but we have changed the name to reflect the more general usage (there is no group for transactional producers). We bump up the version of the request and add a new field indicating the group type, which can be either Consumer or Txn. Request handling details can be found here.

```
// v2

FindCoordinatorRequest => CoordinatorKey CoordinatorType

  CoordinatorKey => string

  CoordinatorType => byte   /* 0: consumer, 1: transaction */


FindCoordinatorResponse => ErrorCode Coordinator

  ErrorCode => int16

  Coordinator => NodeId Host Port

    NodeId => int32
```

```
    Host => string
    Port => int32
```

```
Error code:
```

- Ok
- CoordinatorNotAvailable

The node id is the identifier of the broker. We use the coordinator id to identify the connection to the corresponding broker.

InitPidRequest

Sent by producer to its transaction coordinator to to get the assigned PID, increment its epoch, and fence any previous producers sharing the same TransactionalId. Request handling details can be found here.

```
InitPidRequest => TransactionalId TransactionTimeoutMs
  TransactionalId => String
  TransactionTimeoutMs => int32
```

```
InitPidResponse => Error PID Epoch
  Error => Int16
  PID => Int64
  Epoch => Int16
```

```
Error code:
```

- Ok

- NotCoordinatorForTransactionalId
- CoordinatorNotAvailable
- ConcurrentTransactions
- InvalidTransactionTimeout

AddPartitionsToTxnRequest

Sent by producer to its transaction coordinator to add a partition to the current ongoing transaction. Request handling details can be found here.

```
AddPartitionsToTxnRequest => TransactionalId PID Epoch [Topic [Partition]]
  TransactionalId => string
  PID => int64
  Epoch => int32
  Topic => string
  Partition => int32

AddPartitionsToTxnResponse => ErrorCode
  ErrorCode: int16
```

Error code:

- Ok
- InvalidProducerEpoch
- InvalidPidMapping

- NotCoordinatorForTransactionalId
- CoordinatorNotAvailable
- ConcurrentTransactions
- InvalidTxnRequest

AddOffsetsToTxnRequest

Sent by the producer to its transaction coordinator to indicate a consumer offset commit operation is called as part of the current ongoing transaction. Request handling details can be found here.

```
AddOffsetsToTxnRequest => TransactionalId PID Epoch ConsumerGroupID
  TransactionalId => string
  PID => int64
  Epoch => int32
  ConsumerGroupID => string

AddOffsetsToTxnResponse => ErrorCode
  ErrorCode: int16
```

Error code:

- Ok
- InvalidProducerEpoch
- InvalidPidMapping
- ConcurrentTransactions
- NotCoordinatorForTransactionalId

- CoordinatorNotAvailable
- InvalidTxnRequest

EndTxnRequest

Sent by producer to its transaction coordinator to prepare committing or aborting the current ongoing transaction. Request handling details can be found here.

```
EndTxnRequest => TransactionalId PID Epoch Command

  TransactionalId => string

  PID => int64

  Epoch => int32

  Command => boolean (0 means ABORT, 1 means COMMIT)


EndTxnResponse => ErrorCode

  ErrorCode => int16
```

Error code:

- Ok
- InvalidProducerEpoch
- InvalidPidMapping
- CoordinatorNotAvailable
- ConcurrentTransactions
- NotCoordinatorForTransactionalId
- InvalidTxnRequest

WriteTxnMarkerRequest

Sent by transaction coordinator to broker to commit the transaction. Request handling details can be found here.

```
WriteTxnMarkerRequest => [PID Epoch Marker CoordinatorEpoch [Topic [Partition]]]

  PID => int64

  Epoch => int32

  CoordinatorEpoch => int32

  Marker => int8 (0 = COMMIT, 1 = ABORT)

  Topic => string

  Partition => int32


WriteTxnMarkerResponse => [Pid [Topic [Partition ErrorCode]]]

  Topic => string

  Partition => int32

  ErrorCode => int16
```

Error code:

- Ok

TxnOffsetCommitRequest

Sent by transactional producers to consumer group coordinator to commit offsets within a single transaction. Request handling details can be found here.

Note that just like consumers, users will not be exposed to set the retention time explicitly, and the default value (`-1`) will always be used which lets broker to determine its retention time.

```
TxnOffsetCommitRequest   => ConsumerGroupID
                            PID
                            Epoch
                            RetentionTime
                            OffsetAndMetadata
  ConsumerGroupID => string
  PID => int64
  Epoch => int32
  RetentionTime => int64
  OffsetAndMetadata => [TopicName [Partition Offset Metadata]]
    TopicName => string
    Partition => int32
    Offset => int64
    Metadata => string

TxnOffsetCommitResponse => [TopicName [Partition ErrorCode]]]
  TopicName => string
  Partition => int32
  ErrorCode => int16
```

Error code:

- InvalidProducerEpoch

Metrics

As part of this work, we would need to expose new metrics to make the system operable. These would include:

1. Number of live PIDs (a proxy for the size of the PID->Sequence map)
2. Current LSO per partition (useful to detect stuck consumers and lost commit/abort markers).
3. Number of active transactionalIds (proxy for the memory consumed by the transaction coordinator).

Future Work

Recovering from correlated hard failures.

When there is a correlated hard failure (e.g., power outage), it's possible that an existing commit/abort marker is lost in all replicas. This may not be fixed by the transaction coordinator automatically and the consumer may get stuck on that incomplete transaction forever.

In particular, if a correlated hard failure causes us to lose the markers everywhere, the LSO on the broker would not advance and consumers would block (but not buffer). This would be a noticeable situation.

A simple tool may make use of internal functions to effectively do a 'beginTransaction', 'AddTopicPartitiontoTransaction', 'commitTransaction'. This would ensure that the markers are rewritten to all partitions by the transaction coordinator.

To make this workable, we need to make sure that the transaction coordinator adds a sufficient logging so that we know the TransactionalId -> PID mapping as well as the partitions participating in each transaction. The broker should also probably log information so that we know which unfinished transaction (ie. which PID) is preventing the LSO from moving forward. Both these things will make it fairly easy to configure the tool.

Of course, it is possible for the producer to continue onto another transaction before the tool is run, in which case the data will be corrupt since the second transaction will include messages from the first. But this is no worse than Kafka's existing durability semantics which this proposal relies on.

Producer HeartBeat

Currently, there is no check for producer liveness. This means that if a producer has not been sending transactional requests for a long time, its TransactionalId will be expired by the

coordinator, making it a zombie who can no longer initiate transactions. Adding a heartbeat thread would solve this problem and avoid surprises for applications.

Another use case of producer heartbeat is that we can then do not let the “newcomers” (i.e. a new client with the same TransactionalId) to always kick out the old ones, which will be useful in some single-writer scenario.

Update Producer Interceptor

It would make sense to add new APIs to the producer interceptor to expose the transaction lifecycle (i.e. beginning, committing, and aborting transactions). A challenge here is the fact that the client will not necessarily receive a commit or abort event in the case of a failure.

Rejected Alternatives

On TransactionalIds and PIDs

There have been multiple questions about the notions of TransactionalId and PID: about whether we need both, and if we need both, then can we generate the PID from the TransactionalId by some hash function on the client (vs. a broker side zookeeper based mapping).

The reason we went with a user-provided TransactionalId that maps to a broker-generated PID is the following:

- A persistent identifier for a producer is essential for transactions to be recoverable across producer sessions. For a generated identifier, the application must be responsible for its persistence. In some cases, this is quite difficult. For example, in Kafka Streams, we would like for each task to have its own transactional identifier. If it is generated, then we must create a separate topic to map the taskId to the generated identifier so that we can always lookup the transactional id when a task is restarted. This leads to a “bootstrapping problem” in which two instances of the same task (say one of them is a zombie) may both try to generate the identifier at the same time. We cannot let them both succeed since we can only have one writer for each task. One way to resolve this is to use a datastore with “first-write-wins” semantics, but this is awkward at best if implemented using a Kafka topic. In any case, it seemed much simpler for the task to simply derive its transactional identifier from the taskId itself.
- We would like to be able to restrict access to the transaction log. Doing so requires that we define the “resource” we are trying to protect. For consumers, we have the consumer groupID, but producers would not have anything similar to use if the transactional ids were generated. We could create a producer group, but it is quite awkward to need to

introduce this concept only for the sake of authorization. On the other hand, the TransactionalId gives us exactly what we need. Authorizing a principal for a given TransactionalId gives them access to write to the transaction log using that TransactionalId.

- We chose not to use the externally created TransactionalId directly in the message set because size is a premium. The TransactionalId is a variable length string, and could be very long, so we were concerned about the increased overhead. Hence we chose to store an internally generated PID in the message set instead.
- We chose to generate the PID on the broker using zookeeper versus doing a static hash of TransactionalId to PID on the producer because we did not want to risk collisions. When PIDs collide, one producer will fence the other off, and the only solution would be to restart one of the producers with a new TransactionalId with the hope that it doesn't clash either. This failure mode is non intuitive and potentially hard to fix. Hence we decided not to use any sort of static hashing to map the TransactionalId to PID.