KIP-98 - Exactly Once Delivery and Transactional Messaging

[This KIP proposal is a joint work between Jason Gustafson, Flavio Junqueira, Apurva Mehta, Sriram, and guozhang Wang]

Status

Current state: *Adopted*

Discussion thread: http://search-hadoop.com/m/Kafka/uyzND1jwZrr7HRHf?subj=+DISCUSS+KIP+98+Exactly+Once+Delivery+and+Transactional+Messaging

JIRA: Unable to render Jira issues macro, execution error.

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.

In this document we present a proposal for bringing transactions to Kafka. We will only focus on the user facing changes: the client API changes, and the new configurations we will introduce, and the summary of guarantees. We also outline the basic data flow, which summarizes all the new RPCs we will introduce with transactions. The design details are presented in a separate document.

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 her to keep this use case in mind as it motivated large elements of the proposal.

Public Interfaces

Producer API changes

The producer will get five new methods (initTransactions, beginTransaction, sendOffsets, commitTransaction, abortTransaction), with the send method updated to throw a new exception. This is detailed below:

KafkaProducer.java

```
public interface Producer<K,V> extends Closeable {
```

```
  /**
   * Needs to be called before any of the other transaction methods. Assumes that
   * the transactional.id is specified in the producer configuration.
   *
   * This method does the following:
   *   1. Ensures any transactions initiated by previous instances of the producer
   *      are completed. If the previous instance had failed with a transaction in
   *      progress, it will be aborted. If the last transaction had begun completion,
   *      but not yet finished, this method awaits its completion.
   *   2. Gets the internal producer id and epoch, used in all future transactional
   *      messages issued by the producer.
   *
   * @throws IllegalStateException if the TransactionalId for the producer is not set
   *         in the configuration.
   */
  void initTransactions() throws IllegalStateException;

  /**
   * Should be called before the start of each new transaction.
   *
   * @throws ProducerFencedException if another producer is with the same
   *         transactional.id is active.
   */
  void beginTransaction() throws ProducerFencedException;

  /**
   * Sends a list of consumed offsets to the consumer group coordinator, and also marks
   * those offsets as part of the current transaction. These offsets will be considered
   * consumed only if the transaction is committed successfully.
   *
   * This method should be used when you need to batch consumed and produced messages
   * together, typically in a consume-transform-produce pattern.
   *
   * @throws ProducerFencedException if another producer is with the same
   *         transactional.id is active.
   */
  void sendOffsetsToTransaction(Map<TopicPartition, OffsetAndMetadata> offsets,
                                String consumerGroupId) throws ProducerFencedException;

  /**
   * Commits the ongoing transaction.
   *
   * @throws ProducerFencedException if another producer is with the same
   *         transactional.id is active.
   */
  void commitTransaction() throws ProducerFencedException;

  /**
   * Aborts the ongoing transaction.
   *
   * @throws ProducerFencedException if another producer is with the same
   *         transactional.id is active.


   */
  void abortTransaction() throws ProducerFencedException;


  /**
   * Send the given record asynchronously and return a future which will eventually contain the response
information.
   *
   * @param record The record to send
   * @return A future which will eventually contain the response information
   *
   */
  public Future<RecordMetadata> send(ProducerRecord<K, V> record);

  /**
   * Send a record and invoke the given callback when the record has been acknowledged by the server
   */
```

```
  public Future<RecordMetadata> send(ProducerRecord<K, V> record, Callback callback);
}
```

The OutOfOrderSequence Exception

The Producer will raise an `OutOfOrderSequenceException` if the broker detects data loss. In other words, if it receives a sequence number which is greater than the sequence it expected. This exception will be returned in the `Future` and passed to the `Callback`, if any. This is a fatal exception, and future invocations of Producer methods like `send`, `beginTransaction`, `commitTransaction`, etc. will raise an `IlegalStateException`.

An Example Application

Here is a simple application which demonstrates the use of the APIs introduced above.

KafkaTransactionsExample.java

```
public class KafkaTransactionsExample {

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


    // Note that the 'transactional.id' configuration _must_ be specified in the
    // producer config in order to use transactions.
    KafkaProducer<String, String> producer = new KafkaProducer<>(producerConfig);

    // We need to initialize transactions once per producer instance. To use transactions,
    // it is assumed that the application id is specified in the config with the key
    // transactional.id.
    //
    // This method will recover or abort transactions initiated by previous instances of a
    // producer with the same app id. Any other transactional messages will report an error
    // if initialization was not performed.
    //
    // The response indicates success or failure. Some failures are irrecoverable and will
    // require a new producer  instance. See the documentation for TransactionMetadata for a
    // list of error codes.
    producer.initTransactions();

    while(true) {
      ConsumerRecords<String, String> records = consumer.poll(CONSUMER_POLL_TIMEOUT);
      if (!records.isEmpty()) {
        // Start a new transaction. This will begin the process of batching the consumed
        // records as well
        // as an records produced as a result of processing the input records.
        //
        // We need to check the response to make sure that this producer is able to initiate
        // a new transaction.
        producer.beginTransaction();

        // Process the input records and send them to the output topic(s).
        List<ProducerRecord<String, String>> outputRecords = processRecords(records);
        for (ProducerRecord<String, String> outputRecord : outputRecords) {
          producer.send(outputRecord);
        }

        // To ensure that the consumed and produced messages are batched, we need to commit
        // the offsets through
        // the producer and not the consumer.
        //
        // If this returns an error, we should abort the transaction.

        sendOffsetsResult = producer.sendOffsetsToTransaction(getUncommittedOffsets());


        // Now that we have consumed, processed, and produced a batch of messages, let's
        // commit the results.
        // If this does not report success, then the transaction will be rolled back.
        producer.commitTransaction();
      }
    }
  }
}
```

New Configurations

Broker configs

transactional.id.timeout.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 their ids.
max.transaction.timeout.ms	The maximum allowed timeout for transactions. If a client's requested transaction time exceed this, then the broker will return a InvalidTransactionTimeout 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.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.min.isr	The minimum number of insync replicas the each partition of the transaction state topic needs to have to be considered online. Default: 2
transaction.state.log.segment.bytes	The segment size for the transaction state topic. Default: 104857600 bytes.

Producer configs

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 wait for a transaction status update from the producer before proactively aborting the ongoing transaction. This config value will be sent to the transaction coordinator along with the InitPidRequest. If this value is larger than the max.transaction.timeout.ms setting in the broker, the request will fail with a `InvalidTransactionTimeout` error. 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. The default is empty, which means transactions cannot be used.

Consumer configs

isolation.level	Here are the possible values (default is read_uncommitted): read_uncommitted: consume both committed and uncommitted messages in offset ordering. read_committed: only consume non-transactional messages or committed transactional messages in offset order. In order to maintain offset ordering, this setting means that we will have to buffer messages in the consumer until we see all messages in a given transaction.

Proposed Changes

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 by the producer on every message sent to the broker. The broker maintains in memory the sequence numbers it receives for each topic partition from every PID. The broker will reject a produce request if its sequence number is not exactly one greater than the last committed message from that PID/TopicPartition pair. Messages with a lower sequence number result in a duplicate error, which can be ignored by the producer. Messages with a higher number result in an out-of-sequence error, which indicates that some messages have been lost, and is fatal.

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 idempotent producer semantics are potentially useful for stateless applications like metrics tracking and auditing.

Transactional Guarantees

At the core, transactional guarantees enable applications to produce to multiple TopicPartitions atomically, ie. all writes to these TopicPartitions will succeed or fail as a unit.

Further, since consumer progress is recorded as a write to the offsets topic, the above capability is leveraged to enable applications to batch consumed and produced messages into a single atomic unit, ie. a set of messages may be considered consumed only if the entire 'consume-transform-produce' executed in its entirety.

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. Exactly one active producer with a given TransactionalId. 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.

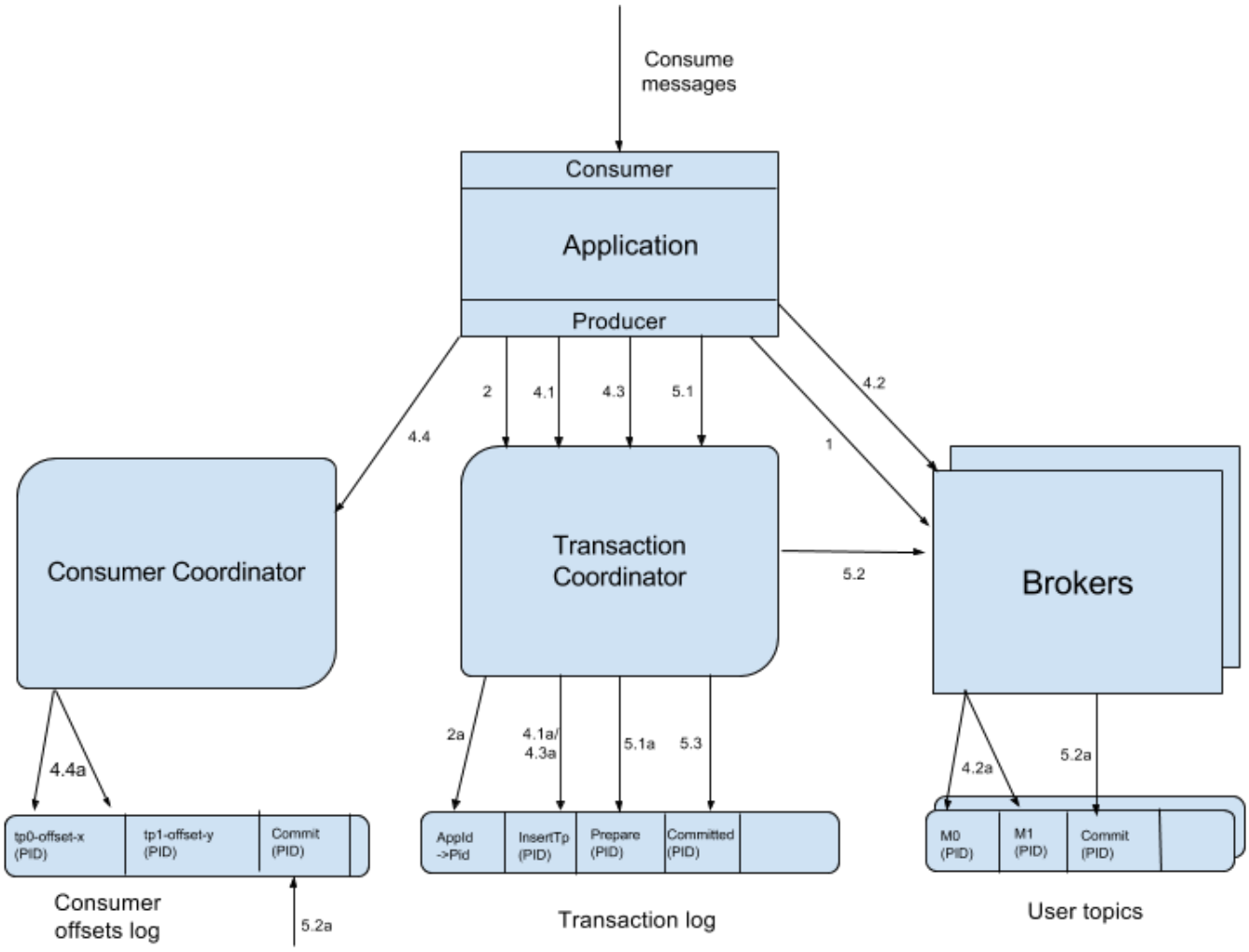
Key Concepts

To implement transactions, ie. ensuring that a group of messages are produced and consumed atomically, we introduce several new concepts:

1. We introduce a new entity called a Transaction Coordinator. Similar to the consumer group coordinator, each producer is assigned a transaction coordinator, and all the logic of assigning PIDs and managing transactions is done by the transaction coordinator.
2. We introduce a new internal kafka topic called the Transaction Log. Similar to the Consumer Offsets topic, the transaction log is a persistent and replicated record of every transaction. The transaction log is the state store for the transaction coordinator, with the snapshot of the latest version of the log encapsulating the current state of each active transaction.
3. We introduce the notion of Control Messages. These are special messages written to user topics, processed by clients, but never exposed to users. They are used, for instance, to let brokers indicate to consumers if the previously fetched messages have been committed atomically or not. Control messages have been previously proposed here.
4. We introduce a notion of TransactionalId, to enable users to uniquely identify producers in a persistent way. Different instances of a producer with the same TransactionalId will be able to resume (or abort) any transactions instantiated by the previous instance.
5. We introduce the notion of a producer epoch, which enables us to ensure that there is only one legitimate active instance of a producer with a given TransactionalId, and hence enables us to maintain transaction guarantees in the event of failures.

In additional to the new concepts above, we also introduce new request types, new versions of existing requests, and new versions of the core message format in order to support transactions. The details of all of these will be deferred to other documents.

Data Flow



In the diagram above, the sharp edged boxes represent distinct machines. The rounded boxes at the bottom represent Kafka TopicPartitions, 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 more general usage) to any broker to discover the location of its coordinator.

2. Getting a producer Id -- the InitPidRequest

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

2.1 When an TransactionalId is specified

If the transactional.id configuration is set, this TransactionalId passed along with the InitPidRequest, 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 the 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 an TransactionalId is not specified

If no TransactionalId is specified in the configuration, 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 T opicPartition 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 AddOffsetCommitsToTxnRequest

The producer has a new KafkaProducer.sendOffsetsToTransaction API method, which enables the batching of consumed and produced messages. This method takes a Map<TopicPartitions, OffsetAndMetadata> and a groupId argument.

The sendOffsetsToTransaction method sends an AddOffsetCommitsToTxnRequests 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.commitTransaction or KafkaProducer.abortTransaction must be called. The former makes the data produced in 4 available to downstream consumers. The latter effectively erases the produced data from the log: it will never be accessible to the user, 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 additional data 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 the messages with the given PID must be delivered to the user 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.

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.

We also require 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 as the producer's TransactionalId is included in the ProduceRequest schema. The WriteTxnMarker API is for inter-broker usage only, and therefore requires ClusterAction permission on the Cluster resource. Note that the writing of control messages is not permitted through the Produce API.

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.

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. Future work can explore protecting the binding between TransactionalId and PID (e.g. through the use of message authentication codes).

RPC Protocol Summary

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

FetchRequest/Response

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

```
// FetchRequest v4

FetchRequest => ReplicaId MaxWaitTime MinBytes 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

```
// 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. Note also the addition of the field for aborted transactions. This is used by the consumer in READ_COMMITTED mode to know where aborted transactions begin. This allows the consumer to discard the aborted transaction data without buffering until the associated marker is read.

ProduceRequest/Response

Sent by the producer to any brokers to produce messages. Instead of allowing the protocol to send multiple message sets for each partition, we modify the schema to allow only one message set for each partition. This allows us to remove the message set size since each message set already contains a field for the size. More importantly, since 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

```
// ProduceRequest v3

ProduceRequest => TransactionalId
                  RequiredAcks
                  Timeout
                  [TopicName [Partition MessageSetSize MessageSet]]
 TransactionalId => nullableString
 RequiredAcks => int16
 Timeout => int32
 Partition => int32
 MessageSetSize => int32
 MessageSet => bytes
```

ProduceResponse

```
// ProduceResponse v3
ProduceResponse => [TopicName [Partition ErrorCode Offset Timestamp]]
                   ThrottleTime
 TopicName => string
 Partition => int32
 ErrorCode => int16
 Offset => int64
 Timestamp => int64
 ThrottleTime => int32
```

Error codes:

- 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/Response

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.

ListOffsetRequest

```
// ListOffsetRequestV2

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

ListOffsetResponse

```
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/Response

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.

FindCoordinatorRequest

```
// v2
FindCoordinatorRequest => TransactionalId CoordinatorType
 TransactionalId => string
 CoordinatorType => byte /* 0: consumer, 1: transaction */
```

FindCoordinatorResponse

```
FindCoordinatorResponse => ErrorCode Coordinator
 ErrorCode => int16
 Coordinator => NodeId Host Port
   NodeId => int32
   Host => string
   Port => int32
```

Error codes:

- 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/Response

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

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

InitPidResponse

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

Error code:

- Ok
- NotCoordinatorForTransactionalId
- CoordinatorNotAvailable
- ConcurrentTransactions
- InvalidTransactionTimeout

AddPartitionsToTxnRequest/Response

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

AddPartitionsToTxnRequest

```
AddPartitionsToTxnRequest => TransactionalId PID Epoch [Topic [Partition]]
 TransactionalId => string
 PID => int64
 Epoch => int16
 Topic => string
 Partition => int32
```

AddPartitionsToTxnResponse

```
AddPartitionsToTxnResponse => ErrorCode
 ErrorCode: int16
```

Error code:

- Ok
- NotCoordinator
- CoordinatorNotAvailable
- CoordinatorLoadInProgress
- InvalidPidMapping
- InvalidTxnState
- ConcurrentTransactions
- GroupAuthorizationFailed

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

```
AddOffsetsToTxnRequest => TransactionalId PID Epoch ConsumerGroupID
 TransactionalId => string
 PID => int64
 Epoch => int16
 ConsumerGroupID => string
```

AddOffsetsToTxnResponse

```
AddOffsetsToTxnResponse => ErrorCode
 ErrorCode: int16
```

Error code:

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

EndTxnRequest/Response

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

EndTxnRequest

```
EndTxnRequest => TransactionalId PID Epoch Command
 TransactionalId => string
 PID => int64
 Epoch => int16
 Command => boolean (false(0) means ABORT, true(1) means COMMIT)
```

EndTxnResponse

```
EndTxnResponse => ErrorCode
 ErrorCode => int16
```

Error code:

- Ok
- InvalidProducerEpoch
- InvalidPidMapping
- CoordinatorNotAvailable

- ConcurrentTransactions
- NotCoordinatorForTransactionalId
- InvalidTxnRequest

WriteTxnMarkersRequest/Response

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

WriteTxnMarkersRequest

```
WriteTxnMarkersRequest => [CoorinadorEpoch PID Epoch Marker [Topic [Partition]]]
 CoordinatorEpoch => int32
 PID => int64
 Epoch => int16
 Marker => boolean (false(0) means ABORT, true(1) means COMMIT)
 Topic => string
 Partition => int32
```

WriteTxnMarkersResponse

```
WriteTxnMarkersResponse => [PID [TopicName [Partition ErrorCode]]]
 PID => int64
 TopicName => string
 Partition => int32
 ErrorCode => int16
```

Error code:

- Ok

TxnOffsetCommitRequest/Response

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

```
TxnOffsetCommitRequest   => ConsumerGroupID
                            PID
                            Epoch
                            RetentionTime
                            OffsetAndMetadata
 ConsumerGroupID => string
 PID => int64
 Epoch => int16
 RetentionTime => int64
 OffsetAndMetadata => [TopicName [Partition Offset Metadata]]
   TopicName => string
   Partition => int32
   Offset => int64
   Metadata => string
```

TxnOffsetCommitResponse

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

Error code:

- InvalidProducerEpoch

Note: The following is tangential to the TxnOffsetCommitRequest/Response: When an OffsetCommitRequest from a consumer failed with a retriable error, we return RetriableOffsetCommitException to the application callback. Previously, this 'RetriableOffsetCommitException' would include the underlying exception. With the changes in KIP-98, we no longer include the underlying exception in the 'RetriableOffsetCommitException'.

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:

```
MessageSet =>
  FirstOffset => int64
  Length => int32
  PartitionLeaderEpoch => int32 /* Added for KIP-101 */
  Magic => int8  /* bump up to “2” */
  CRC => int32 /* CRC32C which covers everything from Attributes on */
  Attributes => int16
  LastOffsetDelta => int32 {NEW}
  FirstTimestamp => int64 {NEW}
  MaxTimestamp => int64 {NEW}
  PID => int64 {NEW}
  ProducerEpoch => int16 {NEW}
  FirstSequence => int32 {NEW}
  Messages => [Message]

Message => {ALL FIELDS NEW}
  Length => varint
  Attributes => int8
  TimestampDelta => varint
  OffsetDelta => varint
  KeyLen => varint
  Key => data
  ValueLen => varint
  Value => data

  Headers => [Header] /* See KIP-82. Note the array uses a varint for the number of headers. */

Header => HeaderKey HeaderVal
  HeaderKeyLen => varint
  HeaderKey => string
  HeaderValueLen => varint
  HeaderValue => 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.

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 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.

The control flag indicates that the messages contained in the message set are not intended for application consumption (see below).

Compression (3)	Timestamp type (1)	Transactional (1)	Control(1)	Unused (10)

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 a signed variable-length integer. 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.

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. All of the message-level attributes are available for future use.

Unused (8)

Control Messages

We use control messages to represent transaction markers. All messages contained in a batch with the control attribute set (see above) are considered control messages and follow a specific format. Each control message must 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. The CRC32C polynomial is used for all CRC computations in the new format because optimised implementations are significantly faster (i.e. if they use the CRC32 instruction introduced in SSE4.2).
6. 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.
7. 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.
8. 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	53 + 100*7 = 753

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).

Compatibility, Deprecation, and 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. Notify clients that they can upgrade, BUT should not start using the idempotent / transactional message APIs yet.
4. [When observed that most of the clients have upgraded] Restart the brokers, with the message format version set to the latest.
5. Notify upgraded clients that they can now start using the idempotent / transactional message APIs.

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.

Note: Since the old producer has long since been deprecated and the old consumer will be deprecated in 0.11.0, these clients will not support the new format. In order to avoid the conversion hit, users will have to upgrade to the new clients. It is possible to selectively enable the message format on topics which are already using the new clients.

Test Plan

Correctness

The new features will be tested through unit, integration, and system tests.

The integration tests will focus on ensuring that the basic guarantees (outlined in the Summary of Guarantees section) are satisfied across components.

The system tests will focus on ensuring that the guarantees are satisfied even with failing components, ie. that the system works even when consumers, producers, brokers are killed in various states.

We will also add to existing compatibility system tests to ensure that old clients can still talk to the new brokers with the new message format.

Performance

This KIP introduces significant changes to the message format along with the new features.

We plan on introducing changes in a staged fashion, with the first change being to the message format. We will run our performance test suite on these message format changes and ensure that there is a minimal performance impact thanks to these changes at worst. Note that the message format changes are the only ones which can affect users who don't enable the idempotent producer and don't use transactions.

Then, we will benchmark the performance of the idempotent producer and the transactional producer separately. Finally, we will benchmark the consumer and broker performance when transactions are in use and read_committed mode is enabled. We will publish the results of all these benchmarks so that users can make informed decisions about when and how to use these features.

Rejected Alternatives

As mentioned earlier, we have a separate design document which explores the design space --including rejected alternatives-- as well as all the implementation details. The latter also includes the specifics of message format changes, new RPCs, error handling, etc.

The design document is available here.