KIP-939: Support Participation in 2PC

Status

Current state: *"Voting"*

Discussion thread: *here*

Voting thread: *here*

JIRA: *here*

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Background

2PC Refresher

Two phase commit (a.k.a. 2PC) protocol is used to implement distributed transactions across multiple participants. Each participant manages its own state that is not recovered from other participants (if some datasets can be fully recovered from another source they are considered as one participant in 2PC protocol, e.g. a database could have multiple tables and indices, but they are all recovered from one log, so it’s one participant; on the other hand a broker manages multiple partitions but they don’t recover from each other, so they are multiple participants). One of the participants (or a separate entity) is designated to be a transaction coordinator. A transaction coordinator could be different for each transaction (e.g. the first participant of a transaction becomes the transaction coordinator) or it could be the same (e.g. if we use a designated transaction coordinator it’s generally used for all transactions); in any case the role of the transaction coordinator for a transaction is that it stores the decision for the transaction outcome (commit or abort) and is able to eventually deliver it to all transaction participants.

The 2PC protocol is the following:

1. Each participant does some atomic transformations. It is generally assumed that each participant can support non-distributed transactions, in which case no additional logic is required by the 2PC: a failed transaction aborts and gets back to a state before the transaction started. If any participant fails while doing some atomic transformation, the distributed transaction is aborted.
2. When a distributed transaction is ready to commit, all participants are asked to prepare the transaction to commit. This is the PREPARE phase (the first phase of 2PC). Each participant is free to say ‘yes’ or ‘no’; if any participant says ‘no’ (or just fails or times out or etc.) the transaction is aborted. If a participant says ‘yes’ during the PREPARE phase, it must be able to commit or abort at will, based on the coordinator’s decision and must remain in-doubt until it receives the coordinator’s decision across all failures of the participant. A transaction that is prepared in a participant is said to be in an in-doubt state (or just in-doubt transaction) – in this state neither the new nor the old state can be revealed, locks are held, etc. All failure modes should be eliminated – data must be flushed and cannot be lost on restart, log needs to have space to store the commit marker, etc. – a commit cannot fail.
3. Once all the participants are prepared, we can make a transaction outcome decision. It could be either commit or abort. If any of the participants replied ‘no’ during the PREPARE phase the transaction would be aborted. If all participants reply ‘yes’ then the transaction coordinator tries to commit the transaction (e.g. by writing a record to its log). If the commit operation succeeds in the transaction coordinator, then the transaction

becomes committed. This is the COMMIT phase, the 2nd phase of the 2PC protocol. The decision is communicated to all participants eventually. The fact that some participants may receive the outcome decision later than the other doesn't affect correctness, because a participant that hasn't received the decision yet doesn't reveal the outcome (locks are held, etc.).

There are variations of implementations (e.g. how the commit decision is communicated – could be pushed by the transaction coordinator, pulled by participants, "gossiped" by the client [if the client saw a commit of a transaction on one participant other participants can infer that it's committed], etc.).

Kafka as a Participant in 2PC

Kafka is a distributed system, so transactions in Kafka are already distributed internally. The 2PC maps to Kafka protocol as the following:

1. Each partition is registered with the Kafka transaction coordinator so that it would abort the transaction in case of abort. Messages are produced to each partition as needed.
2. When a transaction is ready to commit, data is flushed to all data partitions (and commit offsets are flushed to the corresponding offset_commit partitions). If any flush fails, then it's treated as 'no' answer and the transaction is aborted. If all flushes are successful, then the commit decision is sent to the Kafka transaction coordinator to be recorded. The "prepared" transaction data is not visible to consumers reading with 'read_committed' isolation level.
3. The Kafka transaction coordinator writes a `PREPARE_COMMIT` (or `PREPARE_ABORT`) record to its log to record a transaction outcome decision. The name is somewhat misleading, because this is the COMMIT phase and once the `PREPARE_COMMIT` record is written, the transaction is committed and it's just a matter of making the transaction participants aware of decisions. The Kafka transaction coordinator pushes transaction outcome decisions until it's propagated to all participating partitions and wouldn't let another transaction (with the same **transactional.id**) started, until the transaction decision is delivered to all participating partitions.

To protect from client failures and avoid transactions to be in-doubt for a long time, Kafka has a limit on how long a transaction can be running - **transaction.max.timeout.ms** (15 minutes by default), so if a transaction is running longer than the limit it's going to be aborted.

From that perspective, Kafka already has a 2PC implementation, but it's controlled by Kafka transaction coordinator. If we could support an external transaction coordinator, Kafka would be able to become a participant in a distributed transaction with other systems.

The main requirement for a participant is that once prepared it cannot abort (or commit) on its own and must wait for a decision made by the external transaction coordinator. Currently, Kafka violates this requirement in two cases:

1. On client restart, the client must use `InitProducerId` RPC to start using transactions. `InitProducerId` aborts the on-going transaction, so even though the transaction is prepared (all data is flushed in place) it cannot survive client restart.
2. A transaction that's running longer than **transaction.max.timeout.ms** would be aborted (thus would make a decision that may be different from the external transaction coordinator's decision).

Fixing these cases would let Kafka become a participant in an externally coordinated distributed transaction.

Motivation

One challenge if you want to make a log of events a key output of a stateful application is how to update both Kafka and your derived stores together. E.g. say you want to use Kafka as a log of events or changes, but want to have a relational database and/or search index on the data. How can you keep these in sync?

Currently we have three approaches that are possible:

1. DB in front—you write to the DB and capture the changes using the database CDC mechanism (maybe using Kafka Connect) and publish them to Kafka
2. Kafka in front—you write to Kafka you subscribe to the Kafka topic and apply whatever is written there to your DB
3. Dual write—you write to both Kafka and your DB independently

Each of these has the corresponding drawbacks:

1. DB in front with change capture works well for collecting data from existing apps, but for event sourcing this is a bit of a pain since the data will all be formatted in whatever odd way the DB log captures it, and it will be highly normalized in a way that may not be very useful outside your DB schema.
2. Kafka in front lets you have nicely structured events, but you lose read-after-write consistency with your database which is a hard requirement for many (most?) uses.
3. Dual write is easy to implement but means that whenever there is a failure there is a high likelihood that the log and the database will diverge.

It would be nice to have a way to get the best characteristics of all approaches:

1. Well structured event log written directly by the app
2. Read-after-write consistency
3. Consistency between Kafka and all other data stores

Looking at the approaches and their drawbacks, dual writes approach seems to be the most promising as the drawbacks of the other approaches are pretty fundamental to them. The dual writes would work most of the time and we just need to address the atomicity of the failures cases.

To approach a solution for the atomicity of dual writes, let's consider a recipe for atomic consumption from Kafka to a database:

1. Consume data from Kafka
2. Begin database transaction
3. Write consumed data to the database
4. Write offsets of consumed data to the database
5. Commit database transaction

The overhead during normal processing is minimal. In the failure scenario, the offsets of consumed data can be read from the database and the consumption can be resumed from those offsets. Given that the offsets got committed to the database atomically with the data, the state of the database would be consistent with what's consumed from Kafka.

The key idea that makes the consumption atomic is that the database tracks the consumption state atomically with the data update.

We can extend this idea to solve dual writes atomicity problem: we can produce to Kafka in a Kafka transaction and then write the transaction state atomically with the data update. Then in the failure scenario, the transaction state in the database will help us to know whether the Kafka transaction needs to be aborted or committed.

Using this idea the dual write recipe becomes the following:

1. Begin database transaction
2. Begin Kafka transaction
3. Produce data to Kafka
4. Prepare Kafka transaction [currently implicit operation, expressed as flush]
5. Write produced data to the database
6. Write offsets of produced data to the database
7. Commit database transaction
8. Commit Kafka transaction

In the case of failure, we can read the offsets of produced data from the database, and see if the match the offsets of the prepared Kafka transaction and can make a decision whether the Kafka transaction needs to be aborted or committed -- if the offsets in the database are behind the offsets of Kafka transaction, then the transaction needs to be aborted. Currently, there is no way to inspect the offsets of prepared Kafka transactions, but offsets are just a way to uniquely identify an ongoing transaction. Once KIP-890 is implemented, we could use {producerId, epoch} to uniquely identify the transaction.

Given that neither the database nor the application using this recipe needs to interpret the actual data that identifies transaction, it would be best to encapsulate the offset information in an object that can be serialized to / deserialized from a string, so that it's easy to write to / read from the database. The requirements on the database functionality are minimal -- it just needs to support some form of strictly consistent atomic writes, e.g. a key-value store such as RocksDB could be used with dual write recipe as well as full featured SQL or NoSQL databases.

If we had

- Kafka two phase commit support (which is almost there as explained in the background section)
- KIP-890, that makes it possible to uniquely identify transactions by {producerId, epoch}
- `PreparedTransactionState` class that under the covers contains {producerId, epoch} and can be stored as a string
- `.prepareTransaction` function that returns `PreparedTransactionState`
- `.completePreparedTransaction` function that could take `PreparedTransactionState` (and commit / abort based on that)

then the dual write recipe would be more formally defined as:

Database schema prerequisite:

Some way to store a mapping of `transactionalId` (string) to `preparedTransactionState` (string). How it's going to be expressed depends on the database. In a SQL database it could be a table defined by `CREATE TABLE kafka_transaction_state (transactional_id varchar(255), prepared_transaction_state varchar(max));`. In RocksDB it could be a column family named `kafka_transaction_state` where the `transactionalId` would be the key and serialized `preparedTransactionState` would be a value. In Zookeeper it could be a path `/kafka_transaction_state` and the zknodes under this path would be transactional ids and their values would be prepared transaction states. Etc.

Normal processing recipe:

1. Begin database transaction
2. beginTransaction in Kafka
3. send messages to Kafka
4. prepareTransaction in Kafka, get the preparedTransactionState
5. Update data to reflect produced messages
6. Update transactionalId -> preparedTransactionState mapping with the new preparedTransactionState
7. Commit database transaction
8. commitTransaction in Kafka

Recovery recipe:

1. Read preparedTransactionState from the database
2. completePreparedTransaction with the preparedTransactionState

The recovery recipe is used when committing the database transaction fails or after restarts, in other cases we could just abort the transaction on failure.

Once the basic support for the dual write recipe is in place as a result of the KIP, we can implement libraries and / or template examples to facilitate integration with various databases, so that the application writers wouldn't have to implement the recipe from scratch for each application.

Here is a code example that uses the dual write recipe with JDBC and should work for most SQL databases: https://github.com/apache/kafka/pull/14231.

Another important use case that currently has no good solution is supporting Exactly Once Semantics (EOS) between Flink's KafkaSink operator and Kafka. The KafkaSink operator uses Kafka transactions to implement EOS across Flink and Kafka. In that case, Flink's job manager effectively acts as an external transaction coordinator in a 2PC protocol and Kafka is one of the participants. For more details about the Flink KafkaSink operation and how it could utilize Kafka 2PC see FLIP-319.

The KafkaSink operator manages to work around the first problem (`InitProducerId` aborting the transaction) using reflection: KafkaSink keeps track of the `producerId` and `epoch`, so if it needs to commit a transaction after the producer has crashed, it could just issue a commit without going through the `InitProducerId` (which would abort the transaction). Obviously, using reflection leads to maintenance nightmare so adding official support into Kafka so that KafkaSink could use a public API instead of reflection is a great improvement.

The second problem (transaction timeout) cannot be robustly solved without support from Kafka. The current workaround is to crank up **transaction.max.timeout.ms** and hope that Flink won't hold a transaction for a long period of time, but "hoping" is not a very robust way to provide technical guarantees.

Scope

The scope of this KIP is to enable Kafka to be a participant in a 2PC protocol and build a foundation for dual write recipe. Building libraries for dual writes for various databases is out of scope of this KIP. Any additional distributed transaction functionality (e.g. using Kafka as a transaction coordinator, supporting Open XA) is out of the scope of this KIP.

Solution Requirements and Constraints

In the essence we need two changes:

1. Officially support functionality that can be used (and is used by Flink) anyway via reflection.
2. Remove the timeout for transactions participating in 2PC.

We should also wait for KIP-890 to be implemented, so that we could use {producerId, epoch} to identify a transaction.

Adding support for 1 should have no technical implications, we just publicly support functionality that's currently can be used (and is used by Flink) via reflection anyway.

Adding support for 2 requires considering why we have timeout in the first place and how we can mitigate problems that could arise if we remove timeout.

When a partition has an ongoing transaction, there are some implications: consumers that use read_committed isolation level cannot consume past the ongoing transaction (even if there are committed transactions later in the partition, consumers with read_committed isolations won't read past messages that are part of an ongoing transaction), compaction wouldn't compact past ongoing transactions.

transaction.max.timeout.ms guarantees that an ongoing transaction is aborted within a reasonable amount of time, but to avoid violation of the 2PC protocol we need to keep the transaction open, which could put pressure on the system. To mitigate this impact, we should restrict the ability to run 2PC protocol via a privilege, so that it's easy to protect the cluster from a random rogue application.

Even with restricting 2PC functionality to well behaving applications, we cannot prevent cases of long running and / or abandoned transactions. We need some metrics to detect long running transactions. We also need a way for the admin to find and forcibly abort long running and abandoned transactions in case the application cannot properly complete them.

Proposed Changes

The `InitProducerId` API would provide way to indicate that:

- This transactional producer is a participant in 2PC protocol
- Don't abort the ongoing transaction (so that it's possible to commit a prepared transaction even after producer restarts)

The broker will check if the client is allowed to use 2PC protocol, the request will fail if 2PC protocol is not allowed for the client.

If the `InitProducerId` was asked to not abort the ongoing transaction, then the application is only allowed to call `.commitTransaction`, `.abortTransaction`, or `.completeTransaction` calling other methods (e.g. `.send`) would fail. This is because the reason to call `InitProducerId` without aborting the ongoing transaction is to complete a prepared transaction after the producer's crash.

If the `InitProducerId` was asked to not abort the ongoing transaction and the transaction is indeed ongoing, we want to increment the `epoch` (and potentially allocate new `producerId`, if the `epoch` is about to overflow) so that we can fence requests from previous incarnations of the transactional producer. On the other hand, we want to preserve the `producerId` and `epoch` of the ongoing transaction, so that we could properly match it to the prepared transaction state stored in the database. To support that, we need to persist two `(producerId, epoch)` pairs in the transaction state and return both of them in the `InitProducerId` response.

If the transaction is a participant of the 2PC protocol, we don't limit the transactional timeout by the **transaction.max.timeout.ms** any more, the transaction is never aborted automatically.

The admin client would support a new method to abort a transaction with a given transactional id. The method would just execute `InitProducerId`.

A new metric would be added to track ongoing transaction time.

Public Interfaces

RPC Changes

We will bump the `InitProducerId` API. The new schemas are going to be the following:

```
InitProducerIdRequest => TransactionalId TransactionTimeoutMs ProducerId Epoch Enable2Pc KeepPreparedTxn
 TransactionalId => NULLABLE_STRING
 TransactionTimeoutMs => INT32
 ProducerId => INT64
 Epoch => INT16
 Enable2Pc => BOOL  // NEW
 KeepPreparedTxn => BOOL  // NEW

InitProducerIdResponse => Error ProducerId Epoch OngoingTxnProducerId OngoingTxnEpoch
 Error => INT16
 ProducerId => INT64
 Epoch => INT16
 OngoingTxnProducerId => INT64  // NEW
 OngoingTxnEpoch => INT16  // NEW
```

Note that the `OngoingTxnProducerId` and `OngoingTxnEpoch` can be set to -1 if there is no ongoing transaction. In this case calling `.completeTransaction` would be a no-op.

Note that KeepPreparedTxn could be set to `true` even if Enable2Pc is `false`.

Note that TransactionTimeoutMs value is ignored if Enable2Pc is specified.

Persisted Data Format Changes

Note that this KIP is going to be implemented on KIP-890, so the `TransactionalLogValue` will already has the `NextProducerId` tagged field to store a additional producer id, we just need to add `NextEpoch` tagged field to store an additional epoch. The data format is changed in a way that makes it possible to downgrade to a version that supports KIP-915:

```
{
  "type": "data",
  "name": "TransactionLogValue",
  // Version 1 is the first flexible version.
  // KIP-915: bumping the version will no longer make this record backward compatible.
  // We suggest to add/remove only tagged fields to maintain backward compatibility.
  "validVersions": "0-1",
  "flexibleVersions": "1+",
  "fields": [
    { "name": "ProducerId", "type": "int64", "versions": "0+",
      "about": "Producer id in use by the transactional id"},
    { "name": "ProducerEpoch", "type": "int16", "versions": "0+",
      "about": "Epoch associated with the producer id"},
    { "name": "PrevProducerId", "type": "int64", "default": -1, "taggedVersions": "1+", "tag": 0,
      "about": "Producer id in use by client when committing the transaction"},
    { "name": "NextProducerId", "type": "int64", "default": -1, "taggedVersions": "1+", "tag": 1,
      "about": "Producer id returned to the client in the epoch overflow case"},
    { "name": "NextProducerEpoch", "type": "int16", "default": -1, "taggedVersions": "1+", "tag": 2,  // New
      "about": "Producer epoch associated with the producer id returned to the client in the epoch overflow
case"},
    { "name": "TransactionTimeoutMs", "type": "int32", "versions": "0+",
      "about": "Transaction timeout in milliseconds"},
    { "name": "TransactionStatus", "type": "int8", "versions": "0+",
      "about": "TransactionState the transaction is in"},
    { "name": "TransactionPartitions", "type": "[]PartitionsSchema", "versions": "0+", "nullableVersions": "0+",
      "about": "Set of partitions involved in the transaction", "fields": [
      { "name": "Topic", "type": "string", "versions": "0+"},
      { "name": "PartitionIds", "type": "[]int32", "versions": "0+"}]},
    { "name": "TransactionLastUpdateTimestampMs", "type": "int64", "versions": "0+",
      "about": "Time the transaction was last updated"},
    { "name": "TransactionStartTimestampMs", "type": "int64", "versions": "0+",
      "about": "Time the transaction was started"}
  ]
}
```

Note that for transactions with 2PC enabled the `TransactionTimeoutMs` would be set to `-1`.

Let's consider some examples of the state transitions and how the various producer ids and epochs are used.

Vanilla KIP-890 transaction case with epoch overflow:

1. InitProducerId(false); TC STATE: Empty, ProducerId=42, ProducerEpoch=MAX-1, PrevProducerId=-1, NextProducerId=-1, NextProducerEpoch=-1; RESPONSE ProducerId=42, Epoch=MAX-1, OngoingTxnProducerId=-1, OngoingTxnEpoch=-1
2. AddPartitionsToTxn; REQUEST: ProducerId=42, ProducerEpoch=MAX-1; TC STATE: Ongoing, ProducerId=42, ProducerEpoch=MAX-1, PrevProducerId=-1, NextProducerId=-1, NextProducerEpoch=-1
3. Commit; REQUEST: ProducerId=42, ProducerEpoch=MAX-1; TC STATE: PrepareCommit, ProducerId=42, ProducerEpoch=MAX, PrevProducerId=-1, NextProducerId=85, NextProducerEpoch=0; RESPONSE ProducerId=85, Epoch=0
4. (Transition in TC into CompleteCommit); TC STATE: CompleteCommit, ProducerId=85, ProducerEpoch=0, PrevProducerId=42, NextProducerId=-1, NextProducerEpoch=-1

The extra producer id info is there so that if the commit operation times out (and thus the client doesn't get the new ProducerId and ProducerEpoch) and the client retries with the previous ProducerId and ProducerEpoch we can detect the retry and return success. This logic is not new in this KIP, it's part of KIP-890. Note that with vanilla KIP-890 transactions there are no cases when both NextProducerId and PrevProducerId are set – there is at most one of the those extra fields in a given state.

2PC transaction case with two epoch overflows:

1. InitProducerId(false); TC STATE: Empty, ProducerId=42, ProducerEpoch=MAX-1, PrevProducerId=-1, NextProducerId=-1, NextProducerEpoch=-1; RESPONSE ProducerId=42, Epoch=MAX-1, OngoingTxnProducerId=-1, OngoingTxnEpoch=-1.
2. AddPartitionsToTxn; REQUEST: ProducerId=42, ProducerEpoch=MAX-1; TC STATE: Ongoing, ProducerId=42, ProducerEpoch=MAX-1, PrevProducerId=-1, NextProducerId=-1, NextProducerEpoch=-1
3. (Transaction is prepared on the client, then client crashed)
4. InitProducerId(true); TC STATE: Ongoing, ProducerId=42, ProducerEpoch=MAX-1, PrevProducerId=-1, NextProducerId=73, NextProducerEpoch=0; RESPONSE ProducerId=73, Epoch=0, OngoingTxnProducerId=42, OngoingTxnEpoch=MAX-1
5. (crash the client)
6. InitProducerId(true); TC STATE: Ongoing, ProducerId=42, ProducerEpoch=MAX-1, PrevProducerId=-1, NextProducerId=73, NextProducerEpoch=1; RESPONSE ProducerId=73, Epoch=1, OngoingTxnProducerId=42, OngoingTxnEpoch=MAX-1
7. (crash the client a few times to drive the NextProducerEpoch to MAX-1)
8. InitProducerId(true); TC STATE: Ongoing, ProducerId=42, ProducerEpoch=MAX-1, PrevProducerId=-1, NextProducerId=73, NextProducerEpoch=MAX; RESPONSE ProducerId=73, Epoch=MAX, OngoingTxnProducerId=42, OngoingTxnEpoch=MAX-1
9. Commit; REQUEST: ProducerId=73, ProducerEpoch=MAX; TC STATE: PrepareCommit, ProducerId=42, ProducerEpoch=MAX, PrevProducerId=73, NextProducerId=85, NextProducerEpoch=0; RESPONSE ProducerId=85, Epoch=0
10. (Transition in TC into CompleteCommit); TC STATE: CompleteCommit, ProducerId=85, ProducerEpoch=0, PrevProducerId=73, NextProducerId=-1, NextProducerEpoch=-1

This example highlights the following interesting details:

- InitProducerId(true) may be issued multiple times (e.g. client gets into a crash loop). The ProducerId and ProducerEpoch of the ongoing transaction always stay the same, but the NextProducerEpoch is always incremented. Eventually, NextProducerEpoch may overflow, in which case we can allocate a new NextProducerId.
- When a commit request is sent, it uses the latest ProducerId and ProducerEpoch. We send out markers using the original ongoing transaction's ProducerId and ProducerEpoch + 1, but the next transaction will use the latest ProducerId and ProducerEpoch + 1 (this is what the response is going to contain). It may happen (like in this example) that the latest ProducerEpoch is already at MAX, in which case we'd need to allocate a new ProducerId. In order to support retries we store the previous ProducerId in the PrevProducerId. Thus in such situation the PrepareCommit state can have three distinct producer ids:
  - ProducerId – this is used to send our commit markers
  - NextProducerId – this is the producer id to use for the next transaction
  - PrevProducerId – this is the producer id to avoid self-fencing on retries (i.e. if the commit request times out and the client retries with previous producer id, we can return success and new producer id)

Metric Changes

A new metric will be added

kafka.server:type=transaction-coordinator-metrics,name=active-transaction-open-time-max The max time a currently-open transaction has been open

To calculate the metrics we'll just walk through the ongoing transactions and record the max value.

Client Configuration Changes

transaction.two.phase.commit.enable The default would be 'false'. If set to 'true', then the broker is informed that the client is participating in two phase commit protocol and transactions that this client starts never expire.

transaction.timeout.ms The semantics is not changed, but it would be an error to set **transaction.timeout.ms** when **two.phase.commit.enable** is set to 'true'.

Broker Configuration Changes

transaction.two.phase.commit.enable The default would be 'false'. If it's 'false', 2PC functionality is disabled even if the ACL is set, clients that attempt to use this functionality would receive TRANSACTIONAL_ID_AUTHORIZATION_FAILED error.

KafkaProducer API Changes

New `KafkaProducer.PreparedTxnState` class is going to be defined as following:

```
static public class PreparedTxnState {
  public String toString();
  public PreparedTxnState(String serializedState);
  public PreparedTxnState();
}
```

The objects of this class can serialize to / deserialize from a string value and can be written to / read from a database. The implementation is going to store `producerId` and `epoch`.

New overloaded method will be added to `KafkaProducer`:

`public void initTransactions(boolean keepPreparedTxn)`

If the value is 'true' then the corresponding field is set in the `InitProducerIdRequest` and the `KafkaProducer` object is set into a state which only allows calling `.commitTransaction`, `.abortTransaction`, or `.completeTransaction`.

New method will be added to `KafkaProducer`:

`public PreparedTxnState prepareTransaction()`

This would flush all the pending messages and transition the producer into a mode where only `.commitTransaction`, `.abortTransaction`, or `.completeTransaction` could be called (calling other methods, e.g. `.send` , in that mode would result in `IllegalStateException` being thrown). If the call is successful (all messages successfully got flushed to all partitions) the transaction is prepared. If the 2PC is not enabled, we return the `INVALID_TXN_STATE` error.

New method would be added to `KafkaProducer`:

`public void completeTransaction(PreparedTxnState preparedTxnState)`

The method would compare the currently prepared transaction state and the state passed in the argument and either commit or abort the transaction. If the producer is not in prepared state (i.e. neither prepareTransaction was called nor initTransaction(true) was called) we return an INVALID_TXN_STATE error.

AdminClient API Changes

The `Admin` interface will support a new method:

`public TerminateTransactionResult forceTerminateTransaction(String transactionalId)`

`TerminateTransactionResult` just contains `KafkaFuture<void> result` method.

NOTE that there is an existing `abortTransaction` method that is used to abort "hanging" transactions (artifact of some gaps in the transaction protocol implementation that will be addressed in KIP-890, i.e. once part 1 of KIP-890 is implemented we won't have "hanging" transactions). "Hanging" transactions are not known to the Kafka transaction coordinator, they are just dangling messages in data partitions that cannot be aborted via the normal transaction protocol. So `abortTransaction` actually needs information about data partitions so that it could go and insert markers directly there.

On the other hand, the `forceTerminateTransaction` method would operate on a well-formed, but long running transaction for a given transactional id. Under the covers it would just use `InitProducerId` call with `keepPreparedTxn=false`.

ACL Changes

A new value will be added to the `enum AclOperation: TWO_PHASE_COMMIT ((byte) 15`. When `InitProducerId` comes with enable2Pc=true, it would have to have both `WRITE` and `TWO_PHASE_COMMIT` operation enabled on the **transactional id** resource.

Command Line Tool Changes

The `kafka-transactions.sh` tool is going to support a new command `--forceTerminateTransaction`. It has one required argument `--transactionalId` that would take the transactional id for the transaction to be terminated.

The `kafka-acls.sh` tool is going to support a new `--operation TwoPhaseCommit`.

Compatibility, Deprecation, and Migration Plan

The proposal doesn't remove or update any existing functionality, it just adds new functionality that would only be executed if the new configurations and APIs are used.

The broker can be downgraded to versions that support KIP-915. The ProducerId and ProducerEpoch fields contain the ongoing transaction information so an old broker would be able to properly commit or abort the transaction.

Test Plan

The corresponding unit an integration tests will be added.

Rejected Alternatives

Explicit “prepare” RPC

Given that the 2PC protocol is defined in terms of “prepare” and “commit” phases it seems natural to just add a “prepare” RPC to Kafka. The RPC would tell the Kafka transaction coordinator to transition the transaction into a new “prepared” state (note that our current state names are misleading – `PREPARE_COMMIT` is actually the “commit” phase).

There are some potential benefits of doing that:

1. Transactions that haven’t reached “prepared” state can be aborted via timeout.
2. New updates to “prepared” transactions can be rejected.
3. `InitProducerId` would know to not abort prepared transactions.
4. We could query for prepared transactions.

The disadvantage of an explicit “prepare” state is that we’d need to run a synchronous operation on the Kafka transaction coordinator topic and (if we want to support the benefit 2) send “prepare” markers to all leaders (so that they could bounce off new updates).

At a closer examination the benefits are not really eliminating any complexities:

1. We still need tooling and operational support to handle transactions that are stuck in the “prepared” state.
2. The external transaction coordinator would have to keep the state anyway and would know to not send any messages to prepared transactions.
3. The external transaction coordinator would have to keep the state anyway and can pass this info to `InitProducerId` (via the `keepPreparedTx` flag) instead of keeping this info in 2 places.
4. The external transaction coordinator has already the knowledge of prepared transactions.

So we decided to keep the “implicit prepare” the same way we have it in Kafka today and avoid extra synchronous operations that would just duplicate the state that is kept in the external transaction coordinator.

HeartBeat RPC

Potentially, it could be good to distinguish between abandoned transactions (i.e. a producer started a 2PC transaction and then died) and just long running transactions. We could add a heartbeat RPC between the 2 cases.

HeartBeat RPC definitely sounds like a “good thing to do”. It is not clear, though, what would be the cases when we need to handle these situations differently – a long running transaction is a concern regardless of whether it got abandoned or the application is still working on it (also the application might have a bug or etc.), the operator would still need to investigate what’s going on with the application and determine if it’s safe to abort the transaction without violating “prepared” guarantees.

So it doesn’t seem to justify extra complexity, and we just add tooling for inspection of any long running transactions.

Using TransactionTimeoutMs=MAX_INT Instead of Enable2Pc

Having a Boolean flag and then a timeout value seems redundant, we really need just either one of the other, so technically instead of adding an independent flag we could use a special timeout value to indicate that it’s a 2PC transaction. This, however, would couple intent with specific implementation; an explicit Boolean seems to reflect the intent better.

Disallowing keepPreparedTxn=true without 2PC

Without 2PC the notion of “prepared” transaction is subverted by the **transaction.max.timeout.ms** so Kafka cannot promise to keep transaction in-doubt until a decision is reached. So from a purity perspective using keepPreparedTxn=true doesn’t reflect the semantics.

In practice, however, Flink already effectively has a way to keep a “prepared” transaction by using reflection, so if we want to support all the following properties:

1. Get rid of reflection use in Flink and move to public API in new versions of Flink
2. Support new versions of Flink in Kafka clusters that don’t want to grant 2PC privileges.

Then we need to enable using keepPreparedTxn=true even if 2PC is disabled.

Using Partition Offsets as PreparedTxnState

With KIP-890, we can identify transactions as {producerId, epoch}. Without KIP-890 the only way to remember a prepared transaction state is to keep track of partitions and their offsets. The first transaction offsets are already tracked on the partition leaders, so this information is present in the cluster in some form but getting it requires calling partitions leader, which brings complexity that doesn't exist with KIP-890.

Another disadvantage is that the PrepareTxnState isn't bounded if it tracks partitions and their offsets – some producers would produce data into topics with a large number of partitions, some producers would produce data into topics with a small number of partitions. Thus, the solution could work with some configurations (e.g. it was tested with VARCHAR(255)) and break when topic-partition properties get changed. On the other hand, {producerId, epoch} has a small and fixed size.

Support Multiple Concurrent Transactions Per Producer

Currently Kafka supports one transaction per producer, which may limit concurrency. It should be in principle possible to implement support for multiple concurrent transactions, but it seems to be an independent large improvement that deserves its own KIP that should be proposed separately. If such functionality is implemented in Kafka we could amend 2PC to work with multiple transactions.

Skip Epoch Bump When KeepPreparedTxn=true

Epoch bumping for a prepared transaction has some complexity – need to keep another `(producerId, epoch)` pair, so to avoid this complexity we considered just keeping the original epoch if we need to keep the prepared transaction. This actually works for fencing zombie messages because a corresponding abort or commit would bump the epoch anyway with the KIP-890. However, if we have a split brain (two producer instances running concurrently with the same `transactionalId`) we could run into scenarios where they can stomp on each other and commit incorrect data. For example:

1. (instance1) InitProducerId(keepPreparedTxn=true), got epoch=42
2. (instance2) InitProducerId(keepPreparedTxn=true), got epoch=42
3. (instance1) CommitTxn, epoch bumped to 43
4. (instance2) CommitTxn, this is considered a retry, so it got epoch 43 as well
5. (instance1) Produce messageA w/sequence 1
6. (instance2) Produce messageB w/sequence 1, this is considered a duplicate
7. (instance2) Produce messageC w/sequence 2
8. (instance1) Produce messageD w/sequence 2, this is considered a duplicate

Now if either of those commit the transaction, it would have a mix of messages from 2 instances. With the proper epoch bump, instance1 would get fenced at step 3.

Allow Specifying Client Timeout Even When Enable2Pc=true

Technically, we could still let the client control transaction timeout that could exceed **transaction.max.timeout.ms** but it seems to be more confusing than useful.