Transactional Messaging in Kafka

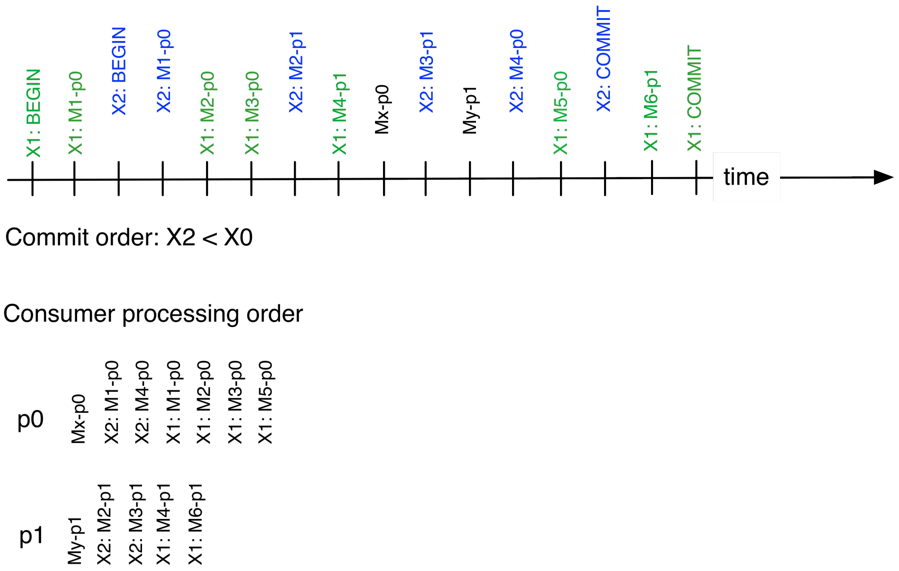
Kafka provides at-least-once messaging guarantees. Duplicates can arise due to either producer retries or consumer restarts after failure. One way to provide exactly-once messaging semantics is to implement an idempotent producer. This has been covered at length in the proposal for an Idempotent Producer. An alternative and more general approach is to support transactional messaging. This can enable use-cases such as replicated logging for transactional data services in addition to the classic idempotent producer use-cases.

What is transactional messaging?

Producers can explicitly initiate transactional sessions, send (transactional) messages within those sessions and either commit or abort the transaction. The guarantees that we aim to achieve for transactions are perhaps best understood by enumerating the functional requirements.

1. Atomicity: A consumer's *application* should not be exposed to messages from uncommitted transactions.
2. Durability: The broker cannot lose any committed transactions.
3. Ordering: A transaction-aware consumer should see transactions in the original transaction-order within each partition.
4. Interleaving: Each partition should be able to accept messages from both transactional and non-transactional producers
5. There should be no duplicate messages within transactions.

If interleaving of transactional and non-transactional messages is allowed, then the relative ordering of non-transactional and transactional messages will be based on the relative order of append (for non-transactional messages) and final commit (for the transactional messages).



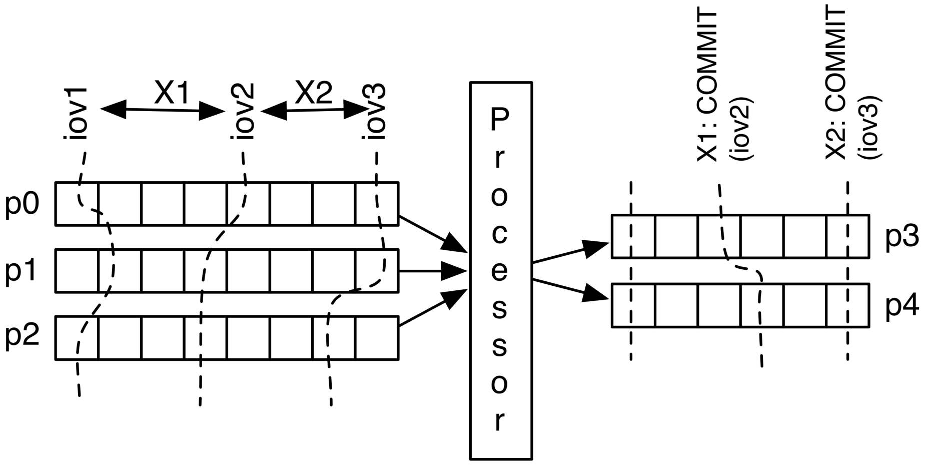
So in the above diagram, partitions p0 and p1 receive messages for transactions X1 and X2, and non-transactional messages as well. The time-line is the time of arrival of the messages at the broker. Since X2 is committed first, each partition will expose messages from X2 before X1. Since the non-transactional messages arrived before the commits for X1 and X2, those messages will be exposed before messages from either transaction.

Furthermore, we have the following requirements pertaining to performance, usability and implementation complexity:

1. The implementation should be scalable. E.g., a dedicated log per transaction is unacceptable.
2. Performance:
    a. The throughput of a transactional producer should be comparable to that of a non-transactional producer.
    b. Acceptable latency. E.g., avoid copying the transactional data as much as possible.
    c. Any implementation should not make the partition unavailable (say, due to locking) for an unreasonable period of time.

3. Client simplicity: Favor a scheme that lend to a simpler client-side implementation (even if it adds more complexity to the broker). For example, it is acceptable (but not ideal) for a consumer implementation to (internally) buffer and subsequently discard messages from uncommitted transactions. i.e., if the chosen implementation allows the broker to materialize messages from uncommitted transactions in the data logs.

Finally, it is worth adding that any implementation should also provide the ability to associate each transaction's input state with the transaction itself. This is necessary to facilitate retries for transactions - i.e., if a transaction needs to be aborted and retried, then the entire input for that transaction needs to be replayed.



Each transaction is associated with a block of input that is processed and results in the output (transaction). When we commit the transaction we would need to also associate the next block of input with that transaction. In the event of a failure the processor would need to query (the downstream Kafka cluster) to determine the next block that needs to be processed. In our case, this would simply be an input offset vector (IOV) for the input partitions that are being processed for each transaction.

Implementation overview

In this implementation proposal, the producer sends transactional control messages that signal the begin/end/abort state of transactions to a highly-available transaction coordinator which manages transactions using a multi-phase protocol. The producer sends transaction control records (begin/end /abort) to the transaction coordinator, and sends the payload of the transaction directly to the destination data partitions. Consumers need to be transaction-aware and buffer each pending transaction until they reach its corresponding end (commit/abort) record.

- Transaction group
- Producer(s) in that transaction group
- Transaction coordinator for that transaction group
- Leader brokers (of the transaction payload's data partitions)
- Consumers of transactions

Transaction group

The transaction group is used to map to a specific transaction coordinator (say, based on a hash against the number of journal partitions). The producers in the group would need to be configured to use this group. Since all transactions from these producers go through this coordinator, we can achieve strict ordering across these transactional producers.

Producer IDs and state groups

In this section, I will go over the need to introduce two new parameters for transactional producers: producer ID and producer group. These don't necessarily need to be part of the producer configuration but may be specified as a parameter in the producer's transactional API.

The preceding overview describes the need to associate the input state of a producer (or in general a processor of some input) along with the last committed transaction. This enables the processor to redo a transaction (by recreating the input state for that transaction - which in our use cases is typically a vector of offsets).

We can utilize the consumer offset management feature to maintain this state. The consumer offset manager associates each key (*consumergroup-topic-partition*) to the last checkpointed offset and metadata for that partition. In the case of a transactional processor, we would want to save the offsets of its consumer that are associated with the commit point of the transaction. This offset commit record (in the *__consumer_offsets* topic) should be written as part of the transaction. i.e., the *__consumer_offsets* topic's partition that stores offsets for the consumer group will need to participate in the transaction. So (for example) suppose a producer fails in the middle of a transaction (which the transaction coordinator subsequently expires); when the producer recovers, it can issue an offset fetch request to recover the input offsets associated with the last committed transaction and resume transactional processing from that point.

There are a few enhancements that we need to make to the offset manager and the compacted *__consumer_offsets* topic in order to support this. First, the compacted topic will now contain transactional control records as well. We will need to come up with an eviction strategy for these control records. Second, the offset manager needs to become transaction-aware - specifically, an offset fetch request should return an error if the group is associated with a pending transaction.

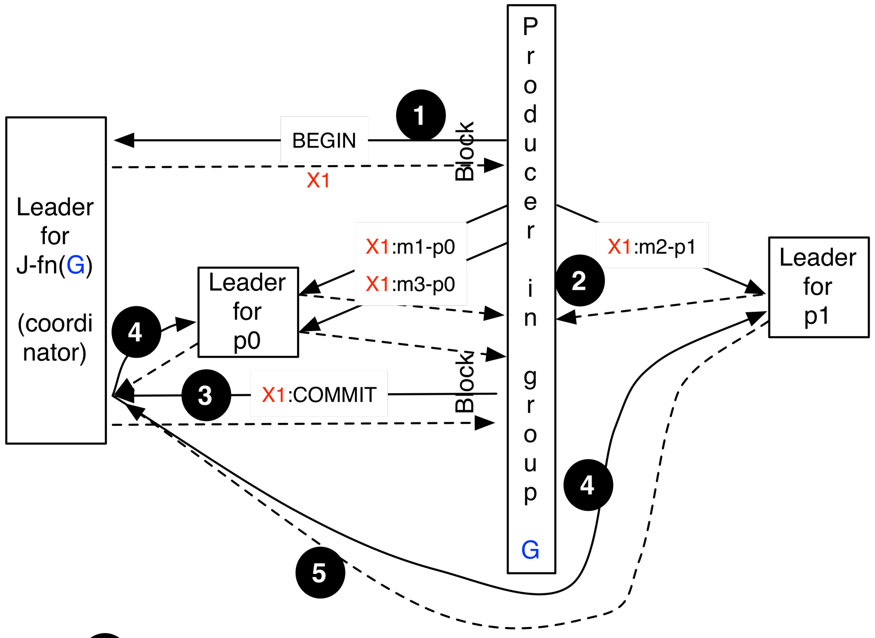
Transaction coordinator

- The transaction coordinator is the leader of a specific partition of the *__transaction_control* topic (i.e., the journal log). It is the point of entry for initiating, committing and aborting transactions. It maintains the following (in-memory) state:
- A high-watermark (of the journal log) corresponding to the BEGIN record of the first in-flight transaction. The transaction coordinator can periodically checkpoint this high-watermark to ZooKeeper.
- For all in-flight transactions that follow the HW in the journal log:
    - The list of topic-partitions (of the payload) of the transaction.
    - Timeout of the transaction.
    - The producer ID associated with the transaction and its generation.

We need to ensure that whatever retention policy is in place does not delete a log segment if the transactional HW is within that log segment. This could just be an additional parameter to the retention policy. i.e., segments including and following that offset should not be deleted.

Transaction flow

(Assuming no failures at each step.)



1 J-fn(G)

...	X1:BEGIN	...

2

p-0	...	X1:m1-p0	...	X1:m3-p0	...
p-1	...	X1:m2-p1	...		

3 J-fn(G)

...	X1:BEGIN	...	X1:PREPARE-COMMIT

4

p-0	...	X1:m1-p0	...	X1:m3-p0	...	X1:COMMITTED
p-1	...	X1:m2-p1	...	X1:COMMITTED		

5 J-fn(G)

...	X1:BEGIN	...	X1:PREPARE-COMMIT	...	X1:COMMITTED

InitPhase

(Step 1 in the figure.)

1. Producer: Determine which broker is the coordinator for its group.
2. Producer: Send a BeginTransaction(producerId, generation, partitions...) request to the coordinator and await response. We could provide a variation of this API that also includes a time-out. If the producer needs to commit its consumer state as part of the transaction it will need to include the relevant partition of the *__consumer_offsets* topic in the BeginTransaction request.
3. Broker: Generate a TxId.
4. Coordinator: Append a BEGIN(TxId, producerId, generation, partitions...) record to the journal log and send response.

5. Producer: Read back response (which will include the TxId).
6. Coordinator (and followers): Update in-memory state of in-flight transactions and data partitions of the transaction.

SendPhase

(Step 2 in the figure.)

Producer: Send transaction payloads (i.e., records) to the leader brokers of the data partitions. Each record will contain the TxId and TxCtl fields. The TxCtl is really only required to mark the final commit (or abort). The producer request envelope will include the producer ID and generation as well, but these will not be appended to the data logs.

EndPhase (When the producer is ready to commit a transaction,)

(Steps 3, 4, 5 in the figure.)

1. Producer: Send an OffsetCommitRequest to commit the input state associated with the end of that transaction (i.e., input for start of next transaction).
2. Producer: Send a CommitTransaction(TxId, producerId, generation) request to the coordinator and await response. (A non-error response indicates to the producer that the transaction will be committed.)
3. Coordinator: Append the PREPARE_COMMIT(TxId) request to the journal log and then send a response to the producer.
4. Coordinator: Send a CommitTransaction(TxId, partitions...) request to every leader broker (for records in the transaction payload).
5. Leader brokers:
    a. If this is a leader for a partition of a topic other than __consumer_offsets: Upon receiving a CommitTransaction(TxId, partition1, partition2, ...) request it will append an empty (null) record (i.e., no key/value) to the respective partition and set the TxId and TxCtl (to COMMITTED) fields of the record. The leader broker will then respond to the coordinator.
    b. If this is a leader for a partition of __consumer_offsets: append a record to the partition with key set to G-LAST-COMMIT, value set to TxId. It should also set the TxId and TxCtl fields of the record. The broker will then respond to the coordinator.
7. Coordinator: Append a COMMITTED(TxId) request to the journal log.
8. Coordinator (and followers): Advance HW if possible (see above for details of the HW).

Aborting a transaction

A transaction may be aborted by a producer due to failures while sending the transaction's payload records. A transaction may also be aborted by the coordinator when it has not been committed within a configurable timeout period.

- Producer: Send an AbortTransaction(TxId) request to the coordinator and await response. (A non-error response indicates to the producer that the transaction will be aborted.)
- Coordinator: Append a PREPARE_ABORT(TxId) record to the journal log and then send a response to the producer.
- Coordinator: Send a AbortTransaction(TxId, partitions...) request to every leader broker (for records in the transaction payload). (The abort-action on the receiving brokers is similar to the above for commits.)

Failure cases (in basic transaction flow)

- Timeout or error response when producer sends BeginTransaction(TxId): producer simply retries (with same TxId).
- Broker-side error while the producer is sending the data: The producer should abort (and subsequently redo) the transaction (with a new TxId). If the producer does not abort the transaction, the coordinator will abort the transaction after the transaction's timeout period. Redoing the transaction is required only in the case of an error for which the request data may have been appended and replicated to the follower. For example, a producer request timeout would require a redo while a NotLeaderForPartitionException does not require a redo.
- Timeout or error response when producer sends CommitTransaction(TxId): producer simply retries the transaction (with same TxId). (However, see section on idempotence further down.)
- Producer failure while a transaction is pending: if the coordinator is in a position to detect a closed socket (when it sends the response to the BeginTransactionRequest) then it can proactively abort the transaction. Otherwise the transaction will be aborted after its timeout period.
- Coordinator failure: i.e., when the coordinator moves to another broker (i.e., leadership of a journal partition moves). The coordinator will scan the log from the last checkpointed HW. If there are any transactions that were in PREPARE_COMMIT or PREPARE_ABORT, the new coordinator will redo the COMMIT and ABORT. Note that transactions that are in-flight when a coordinator goes down don't necessarily need to be aborted - i.e., the producer can just send its CommitTransactionRequest to the new coordinator.

Do transactions make sense for compacted topics?

Compacted topics discard earlier records with the same key during the compaction process. Is this legal if those records are part of a transaction? It is perhaps a bit weird but may not be too harmful since the rationale for using the compaction policy within a topic is to retain the latest update for keyed data.

Still, some messages from within a transaction can get compacted out while others remain. i.e., it is possible that a consumer (that is lagging behind) at an offset within a compacted segment will only see some messages from a transaction. So if that application was (say) updating some table and the messages in the transaction correspond to different keys then this scenario could result in an inconsistent view of the database. i.e., this is a caveat to keep in mind when using transactions within compacted topics.

Compacting topics with transactional messages

The transactional HW is the offset of the earliest pending transaction. The issue is, do we allow messages from committed transactions after the transactional HW (or non- transactional messages) to participate in a round of compaction. I think the answer is yes. There are two cases to consider:

- The transactional HW is in the active segment: In this case, there is no issue since only segments that have rolled over participate in the compaction process.
- The transactional HW is in a dirty portion that has rolled over from the active segment and is thus eligible for compaction: In this case we have two options:

- Do compaction as usual except that messages that are part of a pending transaction need to be copied over to the "cleaned" log. The cleaner checkpoint should not be moved beyond the transactional HW. i.e., even if compaction can be done past the transactional HW that portion should still be considered "dirty" for the next round.
- Don't compact past the transactional HW.
- Both of these approaches require that the transactional HW be maintained on the followers as well. This is not as bad as it sounds since this information can be made available from the transaction coordinator module. The transactional HW only needs to be approximate. i.e., it could be slightly stale at the expense of achieving slightly less compaction than possible.

Pipelining

The above transaction flow contains a number of points where the producer may need to block:

- Response for BeginTransaction request.
- Response for CommitTransaction request.

So in the absence of pipelining, long transactions are good for producer throughput, but require more buffering in consumers. Short transactions require less buffering in consumers, but adversely impact producer throughput (which may be an acceptable penalty to pay to achieve transactional messaging).

It is possible to support pipelining to some degree (courtesy Raul). Earlier versions of this write-up describe a concept of transaction batches. However, that does not actually benefit consumers since transactions in the batch are not exposed by the transaction coordinator until the producer commits the entire batch. It will reduce some load on the transaction coordinator and improve producer throughput but it is probably not worth the effort.

Fencing (a.k.a single-writer requirement)

Some use-cases have the problem of a processor soft-failing in the middle of the send-phase of the transaction (say, due to a long GC) and having a new processor with the same group and producer ID taking over and retrying the transaction. While the retry is in-flight the failed processor may recover and resume its send-phase of the earlier attempt.

This can be addressed by including the producer ID and a generation along with each control record and payload record of the transaction. The producer should store its generation along with the producer's state. When a producer starts up to perform a series of transactions it should increment the generation. The transaction coordinator and leader brokers of the payload partitions can keep track of the current producer generation of pending transactions and reject any requests that come from a producer with an older generation. The producer ID needs to be included in the control records that the transaction coordinator appends to the *__transaction_control* journal log but it does not need to be included in the records of the actual data logs.

One nuance to this is when a producer is starting up for the first time and obtains its group state which will be empty and therefore sets its generation to zero. If it soft-fails at that point and a fail-over producer repeats the same process, we could end up with two producers with the same ID and generation. I think this can be addressed simply by having the transaction coordinator ensure that for a given producerId-generation combination, there can be only one producer connection. If it detects this condition, it can close both connections and abort any transaction that may have been initiated. (The leader brokers should also keep track of in-flight transactions, their associated producerIDs-generations and do the same. Since the abort from the coordinator can arrive before the producer actually stops sending data, the broker needs to reject those producer requests since it does not correspond to any valid pending transaction.)

Transactional consumer

- Consumer configuration will specify whether or not it will consume in READ_COMMITTED or READ_UNCOMMITTED mode.
- READ_UNCOMMITTED: The only change is that the consumer should ignore the transactional control messages.
- In order to support READ_COMMITTED, the consumer will need to maintain a PendingTransactionMap [TxId-Partition] -> messages. This map buffers messages for each transaction/partition combination. It should ideally have the capability of (seamlessly) spilling to disk if it hits a (configurable) memory limit.
- The consumer will maintain an internal iterator to iterate over messages as they arrive. It will buffer messages as long as they are part of a pending transaction but will expose them to the application iterator as soon as it reads a COMMIT for the transaction (or discard them if it reads an ABORT).
- READ_COMMMITTED:
    - When the consumer iterator implementation encounters a transactional control message for a partition, it does not return the message to the application. Instead, it begins to buffer those messages in the PendingTransactionMap.
    - Offset management: The consumer's offset state for each partition will now be an offset tuple. The first offset is the start offset of the earliest pending transaction in that partition. The second offset is the offset of the internal iterator.
    - We may want to maintain one more offset - which is the internal offset of the transaction that we are currently processing. This may be required with long transactions - i.e., a consumer may be in the middle of processing a transaction and fail to call poll in time.)
- Consumer failure
    - If a consumer fails it can delete any state that spilt over to disk (if applicable) and start with a new map.
    - If the previous check-pointed state was (o1, o2) it will resume fetching from o1 and ignore any committed transactions between o1 and o2.

Idempotence

Transaction support should be sufficient to achieve idempotence. However, a producer will need to buffer messages of the transaction until it receives the commit response for that transaction. We can avoid this need to buffer by incorporating sequence numbers in the message header and incorporating pieces of the original idempotent producer proposal. In fact, we would probably want to have some form of the idempotent producer to be available (stand-alone) to avoid aborting long-running transactions that run into (say) a temporary network glitch during the data send phase.