Idempotent Producer

Introduction

Kafka provides "at least once" delivery semantics. This means that a message that is sent may delivered one or more times. What people really want is "exactly once" semantics whereby duplicate messages are not delivered.

There are two common reasons duplicate messages may occur:

1. If a client attempts to send a message to the cluster and gets a network error then retrying will potentially lead to duplicates. If network error occurred before the message was delivered, no duplication will occur. However if the network error occurs after the message is appended to the log but before the response can be delivered to the sender the sender is left not knowing what has happened. The only choices are to retry and risk duplication or to give up and declare the message lost.
2. If a consumer reads a message from a topic and then crashes then when the consumer restarts or another instances takes over consumption the new consumer will start from the last known position of the original consumer.

The second case can be handled by consumers by making use of the offset Kafka provides. They can store the offset with their output and then ensure that the new consumer always picks up from the last stored offset. Or, they can use the offset as a kind of key and use it to deduplicate their output in whatever final destination system they populate.

The first case currently has no good solution, however. The client doesn't know the offset of the message so it has no unique way to identify the message and check if the send succeeded.

For consumers that correctly handle duplicates, this proposal would strengthen the guarantees provided by Kafka to what is often called "atomic broadcast".

This proposal will introduce an optional set of ids that will provide a unique identifier for messages to avoid duplicates.

Some Nuances

Opt-in

Producer implementations that don't care about idempotency should not need to do anything special.

Transitivity: Consumers that also produce

Consider a more elaborate use case which involves copying data from a source to a Kafka topic. This would be the case with Mirror Maker, for example, or any "stream processing" use case. We want it to be the case that the process doing the population can periodically save its position in the upstream topic /database and always resume from this saved position. In the event of a crash we want the copy process to be able to resume from the last known position without producing duplicates in the destination topic. To accomplish this the copy process can save BOTH its input offset/position AND the ids we will introduce associated with its downstream topic. When it restarts after a crash it will initialize with the saved ids. This will effectively make the duplicate produce requests the same as the network error retry case described above.

Fencing

Another twist to this is that it in the mirror maker or other cases where consumer failure is automatically detected it is possible to have false positives leading to a situation where at least transiently we have two consumers reading the same input and producing the same output. It is important that we handle this "split brain" problem correctly and gracefully.

Pipelining

A related need is the ability to pipeline produce requests safely in the presence of retries. When combined with retries this can lead to messages being stored out of order. If the sender sends messages M1, M2, M3 asynchronously without waiting for responses it may then receive a success for M1 and M3 but an error for M2. If it retries M2 successfully this will lead to the topic containing the messages in the order M1, M3, M2.

Fault tolerance

A common cause of errors is actual broker failure. If a broker fails with a request outstanding and unacknowledged you don't know if the newly elected master contains the message or not and will want to retry your request. Thus the idempotency mechanism needs to work even in the presence of broker failures.

Proposed Implementation

A simple, impractical implementation for deduplication would be to have the client create a unique id for each message it sends (a UUID, say) and have the server save all such ids for all messages it retains. New messages would be checked against this database and messages that existed already would be rejected. This would satisfy at least the basic requirements, but would be hopelessly inefficient as the database would contain O(num_messages) entries. A practical implementation will have to provide a similar way of detecting duplicates but with lower space requirements and negligible lookup performance impact.

A similar but more efficient implementation would be to assign each producer a unique id (PID) and keep a sequence number that increments with each message sent. This pair effectively acts as a unique id, but the broker no longer needs to store all the ids to reason about what it has received from a given producer. This leverages the in-order property of Kafka (and TCP) to ensure that the broker need only keep a single "highwater mark" sequence number for each producer and reject any message with a lower sequence number. Specifically if $H(P)$ is the highwatermark and if the broker receives a message with PID P and sequence number S then it will accept the message iff $H(P) < S$.

The next question is whether the producer will maintain a global sequence number across all messages it sends or whether it will be per topic-partition. A global number would be simpler for the client to implement. However if the sequence number was per-partition then the broker could enforce a tighter constraint, namely that $H(P) + 1 = S$. This would allow us to handle the pipelined request case as if any request fails we will automatically fail all other in-flight requests which will allow us to thus retain retry the full set in order.

Note that what is described so far handles the transitive consumer/producer case described above. The process can periodically store both it's offset in its upstream sources as well as its PID and sequence number. When it restarts it will reinitialize with the offset, PID, and sequence number. Several of its initial requests may be rejected as they have already been sent and are below the server's highwater mark.

To complete this proposal we just need to figure out how to provide unique PIDs to producers, how to provide fault tolerance for the highwater marks, and how to provide the "fencing" described above to prevent two producers with the same PID from interfering with one another.

Implementation Details

Now I will make the proposal a bit more explicit.

The first thing to realize is that we must ensure that our deduplication works after a server failure, which means that whichever server takes over as leader for the partition must have all the same producer id information as the former leader. The easiest way to accomplish this is to add the pid fields to the messages themselves so that they are replicated in the log to the followers.

Each message will have three new integer fields: pid, sequence_number, and generation. If PID is set to 0 the server will ignore the sequence_number and generation for compatibility with clients that do not implement idempotency. The server will maintain a mapping of (pid, topic, partition) => (generation, sequence_number_highwater). The server will inspect these fields on each new incoming message and will only append messages to the log if their sequence number is exactly one greater than its highwater mark. In addition the generation must equal the generation stored by the server or be one greater. Incrementing the generation will fence off any messages from "zombie" producers as described above.

Next we need to describe how a producer discovers its PID, sequence_number and generation. To enable this we will add a new API lease_pid which will be used to allocate a unique producer id. The API will have the following format:

Request:

```
lease_pid_request => topic partition
```

Response:

```
lease_pid_response => error pid generation sequence_number expire_ms
```

This request could also have a batch version for handling multiple partitions at once, but I described the single partition case for simplicity.

There are several intended uses for this API:

- When the client first starts and has no PID it will issue a lease_pid_request with the pid field set to -1 for each partition it wishes to produce to. The server will respond with a unique pid a random starting generation and sequence number set to 0.
- The client will be responsible for leasing a new pid before the expire time is reached

Server Implementation

One detail that must be carefully thought through is the expiration of pids. The simplest solution that one might think of is tieing pids to connections so we could automatically deallocate them when the connection is broken. This doesn't really work, though as the pids must survive disconnection (indeed that is their primary point).

Instead this proposal assumes the cluster will have some fixed lifetime for pids from the point of issuance after which a pid is available for reuse. It would also be possible to allow clients to define custom expirations in their lease_pid request but that would require a more complex implementation as all replications would have to know about each expiration. The server will issue pids approximately in order so reuse will only actually occur after 4 billion pids have been issued.

Each server will allocate pids monotonically so if PID N is expired then so is PID M for M > N. This means we can just keep a simple array/list of pid-entries, new entries are added to one end and expired from the other and lookup is just based on binary search. Servers will maintain a fixed amount of memory for pids by making the pid array of fixed size and use it as a circular buffer.

Both leader and followers will maintain this structure. They will periodically snapshot it to disk along with the current offset vector for all partitions they maintain. In the event of a crash they will use this snapshot and offsets to restore from the logs.

Note that the map is updated only by produce requests, the lease_pid request does not change it! The reason for this is to ensure that all data is in the replicated logs and we don't need a second replicated structure. Nonetheless we need to ensure that if a server issues a pid and then fails before any message is produced that pid can't be issued again for that topic/partition by whichever follower takes over even though those followers won't have an entry for it in their maps. To ensure this we will use a global zookeeper sequence to issue pids. For efficiency servers can increment by 100 pids at a time and then not allocate again until they have used these up (this may waste some pids when servers crash, but that is fine).

Note that the expiration is only approximate as it is based on the time a server sees the first message for a partition. However it is only required that the server guarantee at least that much time, so retaining pids longer is okay. This means the followers can use arrival time (though arrival on followers will be slightly older than on the leader). In the event of a full data restore the circular buffer of pid entries will be full and all will have full expiration time restored.

Client Implications

The general deduplication will happen automatically in the producer. It should be cheap and easy enough to enable by default.

To integrate this in tools like mirror maker and samza that chain producers and consumers we will need to be able to save the PID and sequence number of a producer. We can do this by including this in the response returned by the producer.

The producer will need a config to set it's initial PID, sequence number, and generation at initialization time.

We may want to consider extending the OffsetCommit request to also store these fields.