KIP-730: Producer ID generation in KRaft mode

Status

Current state: *Accepted*

Discussion thread: here

Voting thread: here

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

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

Motivation

KIP-98 introduced the InitProducerId RPC (originally InitPid). The purpose of this RPC is to provide unique PIDs (Producer IDs) to producer clients which are transactional or idempotent (also referred to as EOS – Exactly Once Semantics). These unique PIDs are written into the header of a record batch along with other fields (producer epoch, sequence number, etc) which are used for transactions and EOS.

The current implementation of PID generation uses a block generation scheme utilizing ZooKeeper for persistence and concurrency control. Each time a broker needs to allocate a new block of PIDs, it will use ZooKeeper's conditional setData API to allocate the next block:

```
while not done {
  data, version = read_pid_znode()
  data = allocated_new_block()
  done = write_pid_znode_if_version_match( data, version )
}
```

This ensures that a given block of PIDs will only be allocated once across the whole cluster. The block is not persisted anywhere by the broker and only lives in the broker's memory. Once a broker has successfully generated a block of PIDs, that block is considered volatile and will be lost if the broker is restarted. This is not considered a problem since a broker will simply generate a new block upon restart and the PID space is quite large.

With KIP-500, brokers which are running the bridge release version will no longer have direct access to ZooKeeper, so we must abstract this access behind the controller.

With KIP-631, the controller no longer uses ZooKeeper for persistence and instead uses the KRaft metadata log.

This KIP aims to solve both problems by introducing a new RPC and a new method of generating Producer ID blocks using the metadata log as storage.

Public Interfaces

New AllocateProducerIds RPC to be used by brokers to request a new block of IDs from the controller. The use of this RPC in ZK mode is enabled by selecting an IBP of 3.0-IV0 or higher. This RPC is always used in KRaft mode.

```
AllocateProducerIdsRequest => BrokerId BrokerEpoch
  BrokerId => int32
  BrokerEpoch => int64
```

```
AllocateProducerIdsResponse => ErrorCode ProducerIdStart ProducerIdLen
  ErrorCode => int16
  ProducerIdStart => int64
  ProducerIdLen => int32
```

ProducerIdStart is inclusive. E.g., if the controller generates 1000 IDs starting from zero, it will return ProducerIdStart=0 and ProducerIdLen=1000 which represents IDs 0 through 999.

Possible errors could be:

- ClusterAuthorizationFailed – a client with insufficient privileges tried to make this request
- StaleBrokerEpoch – the given broker id and epoch do not match what the controller currently has
- UnknownServerError – some unexpected error occurred on the controller

An authorization error will be considered fatal and should cause the broker to terminate. This indicates the broker is incorrectly configured to communicate with the controller. All other error types should be treated as transient and the broker should retry.

The AllocateProducerIds RPC should be rate limited using the existing throttling mechanism. This will help guard against malicious or malfunctioning clients.

A new metadata record to be used by the controller in the KRaft metadata log

```
AllocateProducerIdsRecord => BrokerId BrokerEpoch ProducerIdEnd
  BrokerId => int32
  BrokerEpoch => int64
  ProducerIdEnd => int64
```

Proposed Changes

Controller

In both ZK and KRaft modes, the controller will now be responsible for generating new blocks of IDs and persisting the latest generated block. In ZK mode, the controller will use the existing ZNode and JSON format for persistence. In KRaft mode, the controller will commit a new record to the metadata log. Since the controller (in either mode) uses a single threaded event model, we can simply calculate the next block of IDs based on what is currently in memory. The controller will need to persist generated PID block so it can be “consumed” and never used again.

It will be the responsibility of the controller to determine the PID block size. We will use a block size of 1000 like the current implementation does. We include the block start and length in the RPC to accommodate for possible changes to the block size in the future.

The AllocateProducerIdsRecord will consume 20 bytes plus record overhead. Since the established upper bound on Producer ID is Long.MAX_VALUE, the required storage could theoretically be in the petabyte range (assuming 1000 IDs per block and no truncation of old records). However, in practice, we will truncate old metadata records and it is unlikely to see such excessive producer ID generation.

Since the RPC will be rate limited, we should be able to avoid cases of excessive metadata generation or PID block exhaustion.

One favorable side effect of using metadata records is that we will effectively have an audit log of producer ID blocks and which brokers requested them. This could be useful in some debugging situations and may warrant some API or tool to expose this history, but that goes beyond the scope of this KIP.

Snapshots

Metadata snapshots will need to include the the latest Producer ID block that was committed to the metadata log. Since we only need the latest block, the impact on the size of the snapshots is trivial.

Broker

The broker will now defer to the controller for allocating blocks of PIDs. This is done by sending an AllocateProducerIdsRequest to the controller using an inter-broker channel. If the request fails, the broker will retry for certain transient errors. The broker should use the ID block returned by the RPC (as opposed to waiting to replicate the AllocateProducerIdsRecord metadata record).

Since there is now a Kafka network request involved in the ID block generation, we should consider pre-fetching blocks so a client is never waiting on an InitProducerIdRequest for too long. We will likely share an existing broker-controller channel for this RPC, so we cannot guarantee low or consistent latencies.

Compatibility, Deprecation, and Migration Plan

Bridge Release

For the Kafka 3.x release that eventually becomes our KRaft “bridge release”, we will need to support the existing ZooKeeper ID-generation along with the new KRaft approach. Since the ZooKeeper approach uses the conditional update API in ZooKeeper, it is actually safe to have some brokers directly talking to ZooKeeper while others are using the new RPC to have the controller update ZooKeeper (as would happen during a rolling upgrade).

All brokers would need to be using the new RPC before upgrading to KRaft mode, which means clusters will need to upgrade to the bridge release before changing to KRaft mode.

See also: Bridge Release section of KIP-500

Upgrade from ZK

In order to upgrade from the ZK-based ID block generation, we will need to ensure that the ID blocks generated by the quorum controller do not overlap with those previously generated by ZK. This can be done by reading the latest producer ID block from ZK and generating an equivalent record in the metadata log. This will need to be incorporated into the overall KRaft upgrade plan once that is available.

Rejected Alternatives

Uncoordinated ID generation

One alternative would be to attempt to pre-allocate IDs in an uncoordinated manner using the broker ID. A scheme like this could probably be made to work, but might be hard to reconcile with previously ZK-generated ID ranges. It also means that some care would need to be taken to account for brokers that might be added in the future. This would also require that the brokers use some durable storage for their own ID ranges. Using the controller is not much more complicated than an uncoordinated approach (possibly its even simpler), and using the controller makes it easier to ensure correctness.

Idempotent RPC

Another design consideration was whether or not the AllocateProducerIdsRequest should be idempotent. If we supported this, the broker would need additional local state and the controller would need additional in-memory state and logic. This approach could help guard against bugs where a broker rapidly requests ID blocks, but it also opens up a new class of bugs where the same producer IDs are returned more than once (possibly to different brokers). Since the ID space is quite large, and this RPC will be rate limited, we can forego this additional logic and the complexity that comes with it.