KIP-866 ZooKeeper to KRaft Migration

Status

Current state: Accepted

Discussion thread: https://lists.apache.org/thread/phnrz31dj0jz44kcjmvzrrmhhsmbx945

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

To complete the plan for KIP-500: Replace ZooKeeper with a Self-Managed Metadata Quorum, we need a way to migrate Kafka clusters from a ZooKeeper quorum to a KRaft quorum. This must be done without impact to partition availability and with minimal impact to operators and client applications.

In order to give users more confidence about undertaking the migration to KRaft, we will allow a rollback to ZooKeeper until the final step of the migration. This is accomplished by writing two copies of the metadata during the migration – one to the KRaft quorum, and one to ZooKeeper.

This KIP defines the behavior and set of new APIs for the “bridge release” as first mentioned in KIP-500.

Public Interfaces

Metrics

MBean name	Description
kafka.server:type=KafkaServer,name=MetadataType	An enumeration of: ZooKeeper (1) or KRaft (2). Each broker reports this.
kafka.controller:type=KafkaController,name=MetadataType	An enumeration of: ZooKeeper (1), KRaft (2), or Dual (3). The active controller reports this.
kafka.controller:type=KafkaController,name=Features,feature={feature},level={level}	The finalized set of features with their level as seen by the controller. Used to help operators see the cluster's current *metadata.version*
kafka.controller:type=KafkaController,name=ZkMigrationState	An enumeration of the possible migration states the cluster can be in. This is only reported by the active controller.
kafka.controller:type=KafkaController,name=MigratingZkBrokerCount	A count of ZK brokers that are registered with KRaft and ready for migration. This will only be reported by the active controller.
kafka.controller:type=KafkaController,name=ZkWriteBehindLag	The amount of lag in records that ZooKeeper is behind relative to the highest committed record in the metadata log. This metric will only be reported by the active KRaft controller.
kafka.controller:type=KafkaController,name=ZkWriteSnapshotTimeMs	The number of milliseconds the KRaft controller took reconciling a snapshot into ZK
kafka.controller:type=KafkaController,name=ZkWriteDeltaTimeMs	The number of milliseconds the KRaft controller took writing a delta into ZK

MetadataVersion (IBP)

A new MetadataVersion in the 3.4 line will be added. This version will be used for a few things in this design.

- Enable forwarding on all brokers (KIP-590: Redirect Zookeeper Mutation Protocols to The Controller)
- Usage of new BrokerRegistration RPC version
- Usage of new controller RPC versions
- Usage of new ApiVersions RPC version (by KRaft controller only)
- Usage of new ZkMigrationStateRecord
- Enable the migration components on KRaft controller and special migration behavior on ZK brokers

All brokers must be running at least this MetadataVersion before the migration can begin. ZK brokers will specify their MetadataVersion using the *inter.broker.protocol.version* as usual. The KRaft controller will bootstrap with the same MetadataVersion (which is stored in the metadata log as a feature flag – see KIP-778: KRaft to KRaft Upgrades).

Configuration

A new “*zookeeper.metadata.migration.enable*” config will be added for the ZK broker and KRaft controller. Its default will be “false”. Setting this config to “true” on each broker is a prerequisite to starting the migration. Setting this to "true" on the KRaft controllers is the trigger for starting the migration (more on that below). Setting this to "true" (or "false") on a KRaft broker has no affect.

Controller RPCs

For the three ZK controller RPCs UpdateMetadataRequest, LeaderAndIsrRequest, and StopReplicaRequest a new *IsKRaftController* field will be added. This field is used to indicate that the controller sending this RPC is a KRaft controller.

```
{
  "apiKey": 4,
  "type": "request",
  "listeners": ["zkBroker"],
  "name": "LeaderAndIsrRequest",
  "validVersions": "0-7",  // <-- New version 7
  "flexibleVersions": "4+",
  "fields": [
    { "name": "ControllerId", "type": "int32", "versions": "0+", "entityType": "brokerId",
      "about": "The controller id." },
    { "name": "isKRaftController", "type": "bool", "versions": "7+", "default": "false",
      "about": "If KRaft controller id is used during migration. See KIP-866" }, <-- New field
    { "name": "ControllerEpoch", "type": "int32", "versions": "0+",
      "about": "The controller epoch." },
    ...
   ]
}
```

```
{
  "apiKey": 5,
  "type": "request",
  "listeners": ["zkBroker"],
  "name": "StopReplicaRequest",
  "validVersions": "0-4",  // <-- New version 4
  "flexibleVersions": "2+",
  "fields": [
    { "name": "ControllerId", "type": "int32", "versions": "0+", "entityType": "brokerId",
      "about": "The controller id." },
    { "name": "isKRaftController", "type": "bool", "versions": "4+", "default": "false",
      "about": "If KRaft controller id is used during migration. See KIP-866" }, // <-- New field
    { "name": "ControllerEpoch", "type": "int32", "versions": "0+",
      "about": "The controller epoch." },
    ...
   ]
}
```

```
{
  "apiKey": 6,
  "type": "request",
  "listeners": ["zkBroker"],
  "name": "UpdateMetadataRequest",
  "validVersions": "0-8",  // <-- New version 8
  "flexibleVersions": "6+",
  "fields": [
    { "name": "ControllerId", "type": "int32", "versions": "0+", "entityType": "brokerId",
      "about": "The controller id." },
    { "name": "isKRaftController", "type": "bool", "versions": "8+", "default": "false",
      "about": "If KRaft controller id is used during migration. See KIP-866" }, // <-- New field
    { "name": "ControllerEpoch", "type": "int32", "versions": "0+",
      "about": "The controller epoch." },
    ...
   ]
}
```

Migration Metadata Record

A new metadata record is added to indicate if a ZK migration has been started or finalized.

```
{
  "apiKey": 21,
  "type": "metadata",
  "name": "ZkMigrationStateRecord",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "ZkMigrationState", "type": "int8", "versions": "0+",
      "about": "One of the possible migration states." },
  ]
}
```

The possible values for ZkMigrationState are: None (0), Pre-Migration (1), Migration (2), and Post-Migration (3). A int8 type is used to give the possibility of additional states in the future.

Broker Registration RPC

A new version of the broker registration RPC will be added to support ZK brokers registering with the KRaft quorum. A new boolean field is added to indicate that the sender of the RPC is a ZK broker that is ready for migration. The usage of this RPC by a ZK broker indicates that it has "zookeeper.metadata.migration.enable" and quorum connection configs properly set. The values of this field are the same as the equivalent field in ApiVersionsRequest.

```
{
  "apiKey":62,
  "type": "request",
  "listeners": ["controller"],
  "name": "BrokerRegistrationRequest",
  "validVersions": "0-1", // <-- New version 1
  "flexibleVersions": "0+",
  "fields": [
    // ...
    { "name": "IsMigratingZkBroker", "type": "bool", "versions": "1+", "default": "false",
      "about": "If the required configurations for ZK migration are present, this value is set to true" }
  ]
}
```

RegisterBrokerRecord

A new field is added to signify that a registered broker is a ZooKeeper broker.

```
{
  "apiKey": 0,
  "type": "metadata",
  "name": "RegisterBrokerRecord",
  "validVersions": "0-2",  // <-- New version 2
  "flexibleVersions": "0+",
  "fields": [
    { "name": "BrokerId", "type": "int32", "versions": "0+", "entityType": "brokerId",
      "about": "The broker id." },
    { "name": "IsMigratingZkBroker", "type": "bool", "versions": "2+", "default": "false",
      "about": "True if the broker is a ZK broker in migration mode. Otherwise, false" },  // <-- New field
    // ...
  ]
}
```

Migration State ZNode

As part of the propagation of KRaft metadata back to ZooKeeper while in dual-write mode, we need to keep track of what has been synchronized. A new ZNode will be introduced to keep track of which KRaft record offset has been written back to ZK. This will be used to recover the synchronization state following a KRaft controller failover.

```
ZNode /migration

{
  "version": 0,
  "kraft_controller_id": 3000,
  "kraft_controller_epoch": 1,
  "kraft_metadata_offset": 1234,
  "kraft_metadata_epoch": 10
}
```

By using conditional updates on this ZNode, will can fence old KRaft controllers from synchronizing data to ZooKeeper if there has been a new election.

Controller ZNodes

The two controller ZNodes "/controller" and "/controller_epoch" will be managed by the KRaft quorum during the migration. More details in "Controller Leadership" section below.

A new version of the JSON schema for "/controller" will be added to include a "kraftControllerEpoch" field.

```
{
  "version": 2, // <-- New version 2
  "brokerid": 3000,
  "timestamp": 1234567890,
  "kraftControllerEpoch": 42     // <-- New field
}
```

This field is intended to be informational to aid with debugging.

Operational Changes

Forwarding Enabled on Brokers

As detailed in KIP-500 and KIP-590, all brokers (ZK and KRaft) must forward administrative requests such as CreateTopics to the active KRaft controller once the migration has started. When running the new *metadata.version* defined in this KIP, all brokers will enable forwarding.

Additional ZK Broker Configs

To support connecting to a KRaft controller for requests such as AlterPartitions, the ZK brokers will need additional configs

- *controller.quorum.voters*: comma-separate list of "node@host:port" (the same as KRaft brokers would set)
- *controller.listener.names*: a comma-separated list of listeners used by the controller
- Corresponding entries in *listener.security.property.map* for the listeners given in *controller.listener.names*

Additional KRaft Broker Configs

To support connecting to ZooKeeper during the migration, the KRaft controllers will need additional configs

- *zookeeper.connect* (required)
- *zookeeper.connection.timeout.ms* (optional)
- *zookeeper.session.timeout.ms* (optional)
- *zookeeper.max.in.flight.requests* (optional)
- *zookeeper.set.acl* (optional)
- ZooKeeper SSL configs (optional)

These configs should match the ZK configs in use by the ZK controller.

Migration Trigger

The migration from ZK to KRaft will be triggered by the cluster's state. To start a migration, the cluster must meet some requirements:

1. Brokers have *inter.broker.protocol.version* set to the version added by this KIP to enable forwarding and indicate they are at the minimum software version
2. Brokers have *zookeeper.metadata.migration.enable* set to “true”. This indicates an operator has declared some intention to start the migration.
3. Brokers have the configs in "Additional ZK Broker Configs" set. This allows them to connect to the KRaft controller.
4. No brokers are offline (we will use offline replicas as a proxy for this).

5. The KRaft quorum is online and all members have *zookeeper.metadata.migration.enable* set to "true" as well as ZK configs set.

The operator can prepare the ZK brokers or KRaft controller in either order. The migration will only begin once every node is ready.

By utilizing configs and broker/controller restarts, we follow a paradigm that Kafka operators are familiar with.

Migration Overview

Here is a state machine description of the migration. There will likely be more internal states that the controller uses, but these four will be exposed as the ZkMigrationState metric.

State	Enum	Description
None	0	The cluster is in KRaft mode and was never migrated from ZooKeeper
PreMigration	1	A KRaft controller has been provisioned and has migration enabled.
Migration	2	The KRaft controller has begun the data migration, brokers are being restarted, dual-writes are in progress.
PostMigration	3	The cluster is in KRaft mode

The active ZooKeeper controller will not report this metric, only the active KRaft controller reports the state corresponding to the state of the migration.

Preparing the Cluster

The first step of the migration is to upgrade the cluster to at least the bridge release version. Upgrading the cluster to a well known starting point will reduce our compatibility matrix and ensure that the necessary logic is in place prior to the migration. The brokers must also set the configs defined above in "Migration Trigger".

To proceed with the migration, all brokers should be online to ensure they satisfy the criteria for the migration.

Controller Migration

This migration only supports dedicated KRaft controllers as the target deployment. There will be no support for migrating to a combined broker/controller KRaft deployment.

A new set of nodes will be provisioned to host the controller quorum. These controllers will be started with *zookeeper.metadata.migration.enable* set to “true”. Once the quorum is established and a leader is elected, the active controller will check that the whole quorum is ready to begin the migration. This is done by examining the new tagged field on ApiVersionsResponse that is exchanged between controllers. Following this, the controller will determine the set of extant ZK brokers and wait for incoming BrokerRegistration requests (see section on *ZK Broker Presence*). Once all known ZK brokers have registered with the KRaft controller (and they are in a valid state) the migration process will begin.

There is no ordering dependency between configuring ZK brokers for the migration and bringing up the KRaft quorum.

The first step in the migration is to copy the existing metadata from ZK and write it into the KRaft metadata log. The active controller will also establish itself as the active controller from a ZK perspective. While copying the ZK data, the controller will not handle any RPCs from brokers.

The metadata migration process will cause controller downtime proportional to the total size of metadata in ZK.

The metadata copied from ZK will be encapsulated in a single metadata transaction (KIP-868). A ZkMigrationStateRecord will also be included in this transaction.

At this point, all of the brokers are running in ZK mode and their broker-controller communication channels operate as they would with a ZK controller. The ZK brokers will learn about this new controller by receiving an UpdateMetadataRequest from the new KRaft controller. From a broker’s perspective, the controller looks and behaves like a normal ZK controller.

Metadata changes are now written to the KRaft metadata log as well as ZooKeeper.

This dual-write mode will write metadata to both the KRaft metadata log and ZooKeeper.

In order to ensure consistency of the metadata, we must stop making any writes to ZK while we are migrating the data. This is accomplished by forcing the new KRaft controller to be the active ZK controller by forcing a write to the "/controller" and "/controller_epoch" ZNodes.

Broker Migration

Following the migration of metadata and controller leadership to KRaft, the brokers are restarted one-by-one in KRaft mode. While this rolling restart is taking place, the cluster will be composed of both ZK and KRaft brokers.

The broker migration phase does not cause downtime, but it is effectively unbounded in its total duration.

There is likely no reasonable way to put a limit on how long a cluster stays in a mixed state since rolling restarts for large clusters may take several hours. It is also possible for the operator to revert back to ZK during this time.

Finalizing the Migration

Once the cluster has been fully upgraded to KRaft mode, the controller will still be running in migration mode and making dual writes to KRaft and ZK. Since the data in ZK is still consistent with that of the KRaft metadata log, it is still possible to revert back to ZK.

The time that the cluster is running all KRaft brokers/controllers, but still running in migration mode, is effectively unbounded.

Once the operator has decided to commit to KRaft mode, the final step is to restart the controller quorum and take it out of migration mode by setting *zookeeper.metadata.migration.enable* to "false" (or unsetting it). The active controller will only finalize the migration once it detects that all members of the quorum have signaled that they are finalizing the migration (again, using the tagged field in ApiVersionsResponse). Once the controller leaves migration mode, it will write a ZkMigrationStateRecord to the log and no longer perform writes to ZK. It will also disable its special handling of ZK RPCs.

At this point, the cluster is fully migrated and is running in KRaft mode. A rollback to ZK is still possible after finalizing the migration, but it must be done offline and it will cause metadata loss (which can also cause partition data loss).

Implementation and Compatibility

Dual Metadata Writes

Metadata will be written to the KRaft metadata log as well as to ZooKeeper during the migration. This gives us two important guarantees: we have a safe path back to ZK mode and compatibility with ZK broker metadata that relies on ZK watches.

At any time during the migration, it should be possible for the operator to decide to revert back to ZK mode. This process should be safe and straightforward. By writing all metadata updates to both KRaft and ZK, we can ensure that the state stored in ZK is up-to-date.

By writing metadata changes to ZK, we also maintain compatibility with a few remaining direct ZK dependencies that exist on the ZK brokers.

- ACLs
- Dynamic Configs
- Delegation Tokens

The ZK brokers still rely on the watch mechanism to learn about changes to these metadata. By performing dual writes, we cover these cases.

The controller will use a bounded write-behind approach for ZooKeeper updates. As we commit records to KRaft, we will asynchronously write data back to ZooKeeper. The number of pending ZK records will be reported as a metric so we can monitor how far behind the ZK state is from KRaft. We may also determine a bound on the number of records not yet written to ZooKeeper to avoid excessive difference between the KRaft and ZooKeeper states.

In order to ensure consistency of the data written back to ZooKeeper, we will leverage ZooKeeper multi-operation transactions. With each "multi" op sent to ZooKeeper, we will include the data being written (e.g., topics, configs, etc) along with a conditional update to the "/migration" ZNode. The contents of "/migration" will be updated with each write to include the offset of the latest record being written back to ZooKeeper. By using the conditional update, we can avoid races between KRaft controllers during a failover and ensure consistency between the metadata log and ZooKeeper.

Another benefit of using multi-operation transactions when synchronizing metadata to ZooKeeper is that we reduce the number of round-trips to ZooKeeper. This pipelining technique is also utilized by the ZK controller for performance reasons.

This dual write approach ensures that any metadata seen in ZK has also been committed to KRaft.

ZK Broker RPCs

In order to support brokers that are still running in ZK mode, the KRaft controller will need to send out additional RPCs to keep the metadata of the ZK brokers up-to-date.

LeaderAndIsr: when the KRaft controller handles AlterPartitions or performs a leader election, we will need to send LeaderAndIsr requests to ZK brokers.

UpdateMetadata: for metadata changes, the KRaft controller will need to send UpdateMetadataRequests to the ZK brokers.

StopReplicas: following reassignments and topic deletions, we will need to send StopReplicas to ZK brokers for them to stop managing certain replicas.

Each of these RPCs will include a new *IsKRaftController* field that indicates if the sending controller is a KRaft controller. Using this field, and the *zookeeper.metadata.migration.enable* config, the brokers can enable migration specific behavior.

Controller Leadership

In order to prevent further writes to ZK, the first thing the new KRaft quorum must do is take over leadership of the ZK controller. This can be achieved by unconditionally overwriting two values in ZK. The "/controller" ZNode indicates the current active controller. By overwriting it, a watch will fire on all the ZK brokers to inform them of a new controller election. The active KRaft controller will write its node ID (e.g., 3000) and epoch into this ZNode to claim controller leadership. This write will be persistent rather than the usual ephemeral write used by the ZK controller election algorithm. This will ensure that no ZK broker can claim leadership during a KRaft controller failover.

The second ZNode we will write to is "/controller_epoch". This ZNode is used for fencing writes from old controllers in ZK mode. Each write from a ZK controller is actually a conditional multi-write with a "check" operation on the "/controller_epoch" ZNode's version. By altering this node, we can ensure any in-flight writes from the previous ZK controller epoch will fail.

Every time a KRaft controller election occurs, the newly elected controller will overwrite the values in “/controller” and “/controller_epoch”. The first epoch generated by the KRaft quroum must be greater than the last ZK epoch in order to maintain the monotonic epoch invariant.

Broker Registration

While running in migration mode, the KRaft controller must know about KRaft brokers as well as ZK brokers. This will be accomplished by having the ZK brokers send the broker lifecycle RPCs to the KRaft controller.

A new version of the BrokerRegistration RPC will be used by the ZK brokers to register themselves with KRaft. The ZK brokers will set the new IsMigrationZkBroker field and populate the Features field with a "metadata.version" min and max supported equal to their IBP. The KRaft controller will only accept the registration if the given "metadata.version" is equal to the IBP/MetadataVersion of the quorum.

After successfully registering, the ZK brokers will send BrokerHeartbeat RPCs to indicate liveness. The ZK brokers will learn about other brokers in the usual way through UpdateMetadataRequest.

If a ZK broker attempts to register with an invalid node ID, cluster ID, or IBP, the KRaft controller will reject the registration and the broker will terminate.

If a KRaft broker attempts to register itself with the node ID of an existing ZK broker, the controller will reject the registration and the broker will terminate.

KRaft Controller Pre-Migration State

When the KRaft quorum is first established prior to starting a migration, it should not handle most RPCs until the initial data migration from ZooKeeper has completed. This is necessary to prevent divergence of metadata during the initial data migration. The controller will need to process RPCs related to Raft as well as BrokerRegistration and BrokerHeartbeat. Other RPCs (such as CreateTopics) will be rejected with a NOT_CONTROLLER error.

Once the metadata migration is complete, the KRaft controller will begin operating normally.

ZK Broker Presence

When the KRaft controller comes up in migration mode, it will wait for all known ZK brokers to register themselves before starting the migration. The problem with this is we cannot know precisely what ZK brokers exist. The broker registrations in ZK are ephemeral and only show the brokers that are currently alive. If an operator had the brokers offline and started a migration, this would lead the controller to think no brokers exist. To improve on this, we can add a heuristic based on the cluster metadata to better capture the full set of ZK brokers. If we look at the topic assignments and configurations, we can calculate a set of brokers which have partitions assigned to them or have a dynamic config. This approach is still imperfect since brokers could be offline and have no assignments, but it will at least prevent any partition unavailability due to a broker running old software and not being able to participate in the migration.

AdminClient, MetadataRequest, and Forwarding

When a client bootstraps metadata from the cluster, it must receive the same metadata regardless of the type of broker it is bootstrapping from. Normally, ZK brokers return the active ZK controller as the *ControllerId* and KRaft brokers return a random alive KRaft broker. In both cases, this *ControllerId* is internally read from the MetadataCache on the broker.

Since we require controller forwarding for this KIP, we can use the KRaft approach of returning a random broker (ZK or KRaft) as the *ControllerId* for clients via MetadataResponse and rely on forwarding for write operations.

For inter-broker requests such as AlterPartitions and ControlledShutdown, we do not want to add the overhead of forwarding so we'll want to include the actual controller in the UpdateMetadataRequest. However, we cannot simply include the KRaft controller as the *ControllerId*. The ZK brokers connect to a ZK controller by using the "*inter.broker.listener.name*" config and the node information from *LiveBrokers* in the UpdateMetadataRequest. For connecting to a KRaft controller, the ZK brokers will need to use the "*controller.listener.names*" and "*controller.quorum.voters*" configs. To allow this, we will use the new *IsKRaftController* field in UpdateMetadataRequest to indicate different controller types to the channel managers.

Topic Deletions

The ZK migration logic will need to deal with asynchronous topic deletions when migrating data. Normally, the ZK controller will complete these asynchronous deletions via TopicDeletionManager. If the KRaft controller takes over before a deletion has occurred, we will need to complete the deletion as part of the ZK to KRaft state migration. Once the migration is complete, we will need to finalize the deletion in ZK so that the state is consistent.

Meta.Properties

Both ZK and KRaft brokers maintain a meta.properties file in their log directories to store the ID of the node and the cluster. Each broker type uses a different version of this file.

v0 is used by ZK brokers:

```
#
#Tue Nov 29 10:15:56 EST 2022
broker.id=0
version=0
cluster.id=L05pbYc6Q4qlvxLk3rTO9A
```

v1 is used by KRaft brokers and controllers:

```
#
#Tue Nov 29 10:16:40 EST 2022
node.id=2
version=1
cluster.id=L05pbYc6Q4qlvxLk3rTO9A
```

Since these two versions contain the same data, but with different field names, we can simply support v0 and v1 in KRaft brokers and avoid modifying the file on disk. By leaving this file unchanged, we better facilitate a downgrade to ZK during the migration. Once the controller has completed the migration and written the final ZkMigrationStateRecord, the brokers can rewrite their meta.properties files as v1 in their log directories.

Rollback to ZK

As mentioned above, it should be possible for the operator to rollback to ZooKeeper at any point in the migration process prior to taking the KRaft controllers out of migration mode. The procedure for rolling back is to reverse the steps of the migration that had been completed so far.

- Brokers should be restarted one by one in ZK mode
- The KRaft controller quorum should be cleanly shutdown
- Operator can remove the persistent "/controller" and "/controller_epoch" nodes allowing for ZK controller election to take place

A clean shutdown of the KRaft quorum is important because there may be uncommitted metadata waiting to be written to ZooKeeper. A forceful shutdown could let some metadata be lost, potentially leading to data loss.

Failure Modes

There are a few failure scenarios to consider during the migration. The KRaft controller can crash while initially copying the data from ZooKeeper, the controller can crash some time after the initial migration, and the controller can fail to write new metadata back to ZK.

Initial Data Migration

For the initial migration, the controller will utilize KIP-868 Metadata Transactions to write all of the ZK metadata in a single transaction. If the controller fails before this transaction is finalized, the next active controller will abort the transaction and restart the migration process.

Controller Crashes

Once the data has been migrated and the cluster is the MigrationActive or MigrationFinished state, the KRaft controller may fail. If this happens, the Raft layer will elect a new leader which update the "/controller" and "/controller_epoch" ZNodes and take over the controller leadership as usual.

Unavailable ZooKeeper

While in the dual-write mode, it is possible for a write to ZK to fail. In this case, we will want to stop making updates to the metadata log to avoid unbounded lag between KRaft and ZooKeeper. Since ZK brokers will be reading data like ACLs and dynamic configs from ZooKeeper, we should limit the amount of divergence between ZK and KRaft brokers by setting a bound on the amount of lag between KRaft and ZooKeeper.

Incompatible Brokers

At any time during the migration, it is possible for an operator to bring up an incompatible broker. This could be a new or existing broker. In this event, the KRaft controller will see the broker registration in ZK, but it will not send it any RPCs. By refusing to send it UpdateMetadata or LeaderAndIsr RPCs, this broker will be effectively fenced from the rest of the cluster.

Misconfigurations

A few misconfiguration scenarios exist which we can guard against.

If a migration has been started, but a KRaft controller is elected that is misconfigured (does not have *zookeeper.metadata.migration.enable* or ZK configs) this controller should resign. When replaying the metadata log during its initialization phase, this controller can see that a migration is in progress by seeing the initial ZkMigrationStateRecord. Since it does not have the required configs, it can resign leadership and throw an error.

If a migration has been finalized, but the KRaft quroum comes up with *zookeeper.metadata.migration.enable*, we must not re-enter the migration mode. In this case, while replaying the log, the controller can see the second ZkMigrationStateRecord and know that the migration is finalized and should not be resumed. This should result in errors being thrown, but the quorum can continue operating as normal.

Other scenarios likely exist and will be examined as the migration feature is implemented.

Test Plan

In addition to basic "happy path" tests, we will also want to test that the migration can tolerate failures of brokers and KRaft controllers. We will also want to have tests for the correctness of the system if ZooKeeper becomes unavailable during the migration. Another class of tests for this process is metadata consistency at the broker level. Since we are supporting ZK and KRaft brokers simultaneously, we need to ensure their metadata does not stay inconsistency for very long.

Rejected Alternatives

Offline Migration

The main alternative to this design is to do an offline migration. While this would be much simpler, it would be a non-starter for many Kafka users who require minimal downtime of their cluster. By allowing for an online migration from ZK to KRaft, we can provide a path towards KRaft for all Kafka users – even ones where Kafka is critical infrastructure.

Online Broker Migration

Once KRaft has taken over leadership of the controller and migrated the ZK data, the design calls for a restart of the ZK brokers into KRaft mode. An alternative to this is to dynamically switch the brokers from using controller RPCs (UpdateMetadata and LeaderAndISR) to the metadata log. This would alleviate the need for a rolling restart of the brokers to bring them into KRaft mode. The difficulty with this approach is that there is a vast difference in the implementations between KafkaServer (ZK) and BrokerServer (KRaft). It is possible to reconcile these differences, but the effort would be very large. This option would also increase the risk of the migration since we would be modifying the "safe" state of the broker code. By leaving the ZK implementation mostly unchanged, we give ourselves a safety net for rolling back during the migration.

No Dual Writes

Another simplifying alternative would be to only write metadata into KRaft while in the migration mode. This has a few disadvantages. Primarily, it makes rolling back to ZK much more difficult, it at all possible. Secondly, we actually have a few remaining ZK *read* usages on the brokers that need the data in ZK to be up-to-date (see above section on Dual Metadata Writes).

Command/RPC based trigger

Another way to start the migration would be to have an operator issue a special command or send a special RPC. Adding human-driven manual steps like this to the migration may make it more difficult to integrate with orchestration software such as Anisble, Chef, Kubernetes, etc. By sticking with a "config and reboot" approach, the migration trigger is still simple, but easier to integrate into other control systems.

Write-ahead ZooKeeper data synchronization

An alternative to write-behind for ZooKeeper would be to write first to ZooKeeper and then write to the metadata log. The main problem with this approach is that it will make KRaft writes much slower since ZK will always be in the write path. By doing a write-behind with offset tracking, we can amortize the ZK write latency and possibly be more efficient about making bulk writes to ZK.

Combined Mode Migration Support

Since combined mode is primarily intended for developer environments, support for migrations under combined mode was not considered a priority for this design. By excluding it from this initial design, we can simply the implementation and exclude an entire system configuration from the testing matrix. The migration design is already complex, so any reduction in scope is beneficial. In the future, it is possible that we could add support for combined mode migrations based on this design.