KIP-853: KRaft Controller Membership Changes

Status

Current state: *Under Discussion*

Discussion threads: https://lists.apache.org/thread/zb5l1fsqw9vj25zkmtnrk6xm7q3dkm1v, https://lists.apache.org/thread/6o3sjvcb8dx1ozqfpltb7p0w76b4nd46

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-595, KIP-630 and KIP631, introduced the KRaft cluster metadata topic partition. This is a partition with replicas that can achieve consensus on the Kafka log without relying on the Controller or ZK. The KRaft Controllers use this partition to order operations on the cluster, commit them to disk and replicate them to other controllers and brokers.

Consensus on the cluster metadata partition was achieved by the voters (controllers). If the operator of a KRaft cluster wanted to make changes to the set of voters, they would have to shutdown all of the controllers nodes and manually make changes to the on-disk state of the old controllers and new controllers. If the operator wanted to replace an existing voter because of a disk failure or general hardware failure, they would have to make sure that the new voter node has a superset of the previous voter's on-disk state. Both of these solutions are manual and error prone.

This KIP describes a protocol for extending KIP-595 so that the operators can programmatically update the voters set in a way that is consistent and available. There are two important use cases that this KIP supports. One use case is that the operator wants to change the number of controllers by adding or removing a controller. The other use case is that the operation wants to replace a controller because of a disk or hardware failure.

Key terms

These are the definition of some important terms that are used through the document.

KRaft: The consensus protocol inspired by Raft for Kafka. This protocol is defined by KIP-595: A Raft Protocol for the Metadata Quorum, KIP-630: Kafka Raft Snapshot. The implementation is in the Java `raft` module.

Controller: The controller or the metadata state machine is the application built using the KRaft consensus layer. The implementation is in the Java `metadata` module.

Voters: A voter is any replica that can transition to the candidate state and to the leader state. Voters for the KRaft cluster metadata partition are also called controllers. Voters are required to have an ID and UUID.

Voters set: A voters set is the group of voters for a partition. Each replica keeps their own set of voters that are part of the topic partition. For a replica, the voters are the replica ID and UUID is in its own voters set. A candidate needs to get votes from the majority of its own voters set before it can become the leader of an epoch. When a voter becomes a leader it will use its voters set to determine when an offset has been committed.

Observers: An observer is any replica that is not in the voters set. This is because they have an ID and UUID which is not in the voters set or they don't have an ID or UUID.

Proposed changes

This KIP is inspired by Chapter 4 of Consensus: Bridging Theory and Practice [2]. The description of this KIP makes the assumption that the reader is familiar with the references enumerated at the bottom of this page. The *user explanation* section explains the feature as the user would interact with it. The *reference explanation* section goes into the implementation details of this feature.

User explanation

There will be two mechanisms for bootstrapping the KRaft cluster metadata partition. For context, the set of voters in the KRaft cluster metadata partition is currently bootstrapped and configured using the `controller.quorum.voters` server property. This property is also used to configure the brokers with the set of endpoints that know the location of the leader if it exists.

This feature will instead persist the set of voters in the cluster metadata partition log, snapshots and quorum state. The new `controller.quorum.bootstrap.servers` property will be added to configure replicas with a set of endpoints that could be used to discover the latest leader.

Bootstrapping with one voter

The recommended method for creating a new KRaft cluster metadata partition is to bootstrap it with one voter. This can be done with the following CLI command:

```
kafka-storage format --cluster-id <cluster-id> --release-version 3.8 --standalone --config controller.properties
```

This command will 1) create a meta.properties file in metadata.log.dir with a randomly generated directory.id, 2) create a snapshot at `00000000000000000000-0000000000.checkpoint` with the necessary control records (`KRaftVersionRecord` and `VotersRecord`) to make this Kafka node the only voter for the quorum.

Bootstrapping with multiple voters

In some deployment scenarios and to support a similar configuration to Apache ZooKeeper, the KRaft cluster metadata partition can also be bootstrap with more than one voter. This can be done with the following CLI command:

```
kafka-storage format --cluster-id <cluster-id> --release-version 3.8 --controller-quorum-voters <replica-id>-
<replica-uuid>@<host>:<port>,... --config controller.properties
```

This command is similar to the standalone version but the snapshot at `00000000000000000000-0000000000.checkpoint` will instead contain a `VotersRecord` that includes information for all of the voters specified in --controller-quorum-voters. Just like the `controller.quorum.voters` properties, it is important that the set of voters is equal in all of the controllers with the same cluster id.

Upgrade KRaft protocol

The changes to the protocol and replicated state for this feature are not compatible with the existing KRaft implementation. This feature cannot be enabled unless all of the replicas for the KRaft cluster metadata partition support this feature. The user can upgrade the KRaft version of an existing KRaft cluster with the following command:

```
kafka-feature upgrade --release-version 3.8 --bootstrap-server <endpoints>
```

This command will parse the release-version to the matching `metadata.version` and `kraft.version` features. It will send a `UpdateFeatures` request to a node with both features set to the matching version. KRaft will write the `KRaftVersionRecord` control record, if all of the controllers and brokers support the new version. KRaft will use the information in the controller registration, broker registration and add voter records to determine if the new version is compatible.

Add controller

To increase the number of controller the user needs to format a controller node, start the controller node and add that node to the KRaft cluster metadata partition. Formatting a controller node can be done with the following CLI command:

```
kafka-storage format --cluster-id <cluster-id> --release-version 3.8 --config controller.properties
```

Notice that neither the `--standalone` or `--controller-quorum-voters` flags is used. After the new controller node has been started, the node can be added to the KRaft cluster metadata partition with the following CLI command:

```
kafka-metadata-quorum --bootstrap-server <endpoints> add-controller --config controller.properties
```

Remove controller

To decrease the number of controllers the user needs to execute the following CLI command:

```
kafka-metadata-quorum --bootstrap-server <endpoints> remove-controller --controller-id <replica-id> --
controller-uuid <replica-uuid>
```

Common scenarios

To better illustrate this feature this section describes few common scenarios that the Kafka administrator may perform.

Automatic joining controllers

Imagine that the user is trying to create KRaft controller quorum with 3 voters (1, 2, 3).

For all of the controller properties files assume that the `controller.quorum.auto.join.enable` is set to `true` and the `controller.quorum.bootstrap.servers` contains the endpoint of controller 1.

On controller 1 the user runs:

```
kafka-storage format --cluster-id <cluster-id> --release-version 3.8 --standalone --config controller.properties
```

and starts the controller. When controller 1 starts it sees that is already in the voters set so it eventually becomes leader.

On controller 2 and 3 the user runs:

```
kafka-storage format --cluster-id <cluster-id> --release-version 3.8 --config controller.properties
```

and starts the controllers. Notice that neither `--standalone` or `--controller-quorum-voters` is used for controller 2 and 3 so the controllers will start as observers. These controllers will discover the leader using `controller.quorum.bootstrap.servers` and will use the `AddVoter` RPC to join the voters set.

Disk failure recovery

If one of the replicas encounter a disk failure the operator can replace this disk with a new disk and start the replica.

Let's assume that the cluster has the following voters for the KRaft cluster metadata partition *(1, UUID1)*, *(2, UUID2)* and *(3, UUID3)*. The first element of the tuples is the replica id. The second element of the tuples is the replica uuid. The replica uuid, or directory id, is generated using the storage tool when the node is formatted.

At this point the disk for replica 3 is replaced and formatted. This means that when replica 3 starts it will have a new replica uuid (*UUID3'*) and an empty set of voters. Replica 3 will discover the partition leader either using `controller.quorum.bootstrap.servers` or by the leader sending a `BeginQuorumEpoch` request to replica 3. Once the new replica, (3, UUID3'), discovers the leader it send the `Fetch` and `FetchSnapshot` requests to the leader. At this point the leader's set of voters will be (1, UUID1), (2, UUID2) and (3, UUID3) and the set of observers will include (3, UUID3').

This state can be discovered by the operator by using the `DescribeQuorum` RPC, the Admin client or the `kafka-metadata-quorum` CLI. The operator can now decide to add replica *(3, UUID3')* to the set of voters using the Admin client or the `kafka-metadata-quorum` CLI.

When the `AddVoter` RPC succeeds the voters set will be *(1, UUID1)*, *(2, UUID2)*, *(3, UUID3)* and *(3, UUID3')*. The operator can remove the failed disk from the voters set by using the Admin client or the `kafka-metadata-quorum` CLI.

When the `RemoveVoter` RPC succeeds the voters will be *(1, UUID1)*, *(2, UUID2)*, and *(3, UUID3')*. At this point the Kafka cluster has successfully recover from a disk failure in a controller node in a consistent way.

Node failure recovery

Let's assume that the voters set is *(1, UUID1)*, *(2, UUID2)* and *(3, UUID3)*.

At this point replica 3 has failed and the Kafka operator would like to replace it. The operator would start a new controller with replica id 4. The node 4's metadata log dir would get formatted, generate and persist the directory id *UUID4*. Replica 4 will discover the leader by sending `Fetch` and `FetchSnapshot` request to the servers enumerated in `controller.quorum.bootstrap.servers`. After a successful `Fetch` RPC, the leader's set of voters will be *(1, UUID1)*, *(2, UUID2)*, *(3, UUID3)* and set of observers will include *(4, UUID4)*.

The operator can now decide to add replica *(4, UUID4)* to the set of voters. When this operation succeeds the set of voters will be *(1, UUID1)*, *(2, UUID2)*, *(3, UUID3)* and *(4, UUID4)*.

The operator can now decided to remove replica *(3, UUID3)* from the set of voters.

Reference explanation

The general goal of this KIP is to allow the user to dynamically change the set of voters (also known as controllers) for the KRaft cluster metadata partition. This is achieved by storing the set of voters and their known endpoints in the log instead of the `controller.quorum.voters` properties. Because the set of voters is stored in the log it allows the leader to replicate this information to all of the fetching replicas (voters and observers). Since old log segments (records) can be deleted once a snapshot has been created, the KRaft snapshots will also contain the set of voters up to the included offset.

The fetching (following) voters will discover the set of voters and their endpoints by fetching the latest log from the leader. How do new voters discovery the leader's endpoint? The leader will push this information to new voters (or voters that were offline for a long time) using the `BeginQuorumEpoch` request. The active leader sends `BeginQuorumEpoch` to all of the voters in the voters set when it becomes leader for an epoch.

The leader doesn't send `BeginQuorumEpoch` to observers since this are dynamic and are not included in the KRaft partition log. Observer will instead discover the leader using the `controller.quorum.bootstrap.servers`. It is important that this property includes at least one of the available voters, else brokers (observers) will not be able to discover the leader of the KRaft cluster metadata partition.

To make changes to the voters set consistent it is required that the majority of the competing voters sets commit the voter changes. In this design the competing voters sets are the current voters set and new voters set. Since this design only allows one voter change at a time the majority of the new configuration always overlaps (intercepts) the majority of the old configuration. This is done by the leader committing the current epoch when it becomes leader and committing single voter changes with the new voters set before accepting another voter change.

The rest of these section and subsection goes into the detail changes required to implement this new functionality.

Directory id or replica UUID

In addition to a replica ID each replica assigned to a KRaft topic partition will have a replica UUID. This UUID will be generated once and persisted in the `meta.properties` file for the `metadata.log.dir`. Replica UUID and directory id was first introduced in KIP-858: Handle JBOD broker disk failure in KRaft.

There are two cases when a directory id will be generated and persisted:

1. `kafka-storage` format command will generate a directory id for all of the log directories including the metadata log dir.
2. `meta.properties` exists but it doesn't include a `directory.id` property. This case will generate and persist a directory id to support upgrade from Kafka versions that don't support this feature to versions that support this feature.

Supported features

This feature can only be enabled if all of the voters (controllers) and observers (brokers) support this feature. This is required mainly because this feature needs to write a new control record (`VotersRecord`) to the KRaft cluster metadata partition. All replicas need to be able to read and decode these new control records.

There will be a new SupportedFeature added to the ApiVersions response of the Kafka nodes. The name of this new supported feature will be `kraft.version`. The default value be 0 and represents that only KIP-595: A Raft Protocol for the Metadata Quorum, KIP-630: Kafka Raft Snapshot and KIP-996: Pre-Vote are supported. Version 1 means that this KIP is supported.

When the clients sends a `UpdateFeatures` RPC to the active controller, if the `FeatureUpdates.Feature` property is `kraft.version`, the associated information will be passed to KRaft client. The KRaft client will implement two different algorithms to check if the upgrade is supported by voters and observers. For voters, the KRaft leader will compare the upgraded version against all of the voters' supported `kraft.version`. The KRaft leader will learn about the supported versions through the UpdateVoter RPC.

The KRaft leader doesn't know the `kraft.version` for the observers (brokers) because observers don't send the UpdateVoter request to the leader. The controller state machine will instead push the brokers' `kraft.version` information to the KRaft client (`RaftClient` in the implementation). The KRaft will use the controller provided `kraft.version` to check if an upgrade is supported.

Any `UpdateFeature` RPC that attempts to downgrade the kraft.version from 1 to 0 will be rejected.

Voter changes

Adding voters

Voters are added to the cluster metadata partition by sending an `AddVoter` RPC to the leader. For safety Kafka will only allow one voter change operation at a time. If there are any pending voter change operations the leader will wait for them to finish.

If there are no pending voter change operations the leader send an ApiVersions request to the new voter's endpoint to discover it's `kraft.version` support features. If the new leader supports the current `kraft.version`, it will write a `VotersRecord` to the log, with the current voters set plus the voter getting added, and immediately update its in-memory quorum state to include this voter as part of the quorum. Any replica that replicates and reads this `VotersRecord` will update their in-memory voters set to include this new voter. Voters will not wait for these records to get committed before updating their voters set.

Once the `VotersRecord` operation has been committed by the majority of the new voters set, the leader can respond to the `AddVoter` RPC and process new voter change operations.

Removing voters

Voter are removed from the cluster metadata partition by sending a `RemoveVoter` RPC to the leader. This works similar to adding a voter. If there are no pending voter change operations the leader will append the `VotersRecord` to the log, the the current voters set minus the voter getting removed, and immediately update its voters set to the new configuration.

Once the `VotersRecord` operation has been committed by the majority of the new voters set, the leader can respond to the RPC. If the removed voters is the leader, the leader will resign from the quorum when the `VotersRecord` has been committed. To allow this operation to be committed and for the leader to resign the followers will continue to fetch from the leader even if the leader is not part of the new voters set. In KRaft, leader election is triggered when the voter hasn't received a successful response in the fetch timeout.

Endpoints information

Replicas need to be able to connect and send RPCs to each others to implement the KRaft protocol. There are two important sets of endpoints that are needed to allow the replicas to connect to each other.

Voters set

The voters set is the collection of replicas and their endpoints that are part that are voters and can become leaders. The voters set is stored in the snapshot and log using the `VotersRecord` control record. If the partition doesn't contain a `VotersRecord` control record then the value stored in the `controller.quorum.voters` property will be used.

This set of endpoints and replicas will be use by the `Vote`, `BeginQuorumEpoch` and `EndQuorumEpoch` RPCs. In other words, replicas will use the voters set to establish leadership and to propagate leadership information to all of the voters.

Bootstrap servers

When the leader is known `Fetch` requests will be sent to the leader's endpoint. The leader's endpoint can be discovered from many of the KRaft RPCs but typically it will be discover from the `BeginQuorumEpoch` request for voters (controllers) or the `Fetch` response for observers (brokers).

In the case that the leader is not known the replica (observer) will send the `Fetch` RPC to the endpoints in the bootstrap servers configuration. The set of endpoints in the bootstrap servers configuration will come from the `controller.quorum.bootstrap.servers` configuration property. If that property is not specified the endpoints in `controller.quorum.voters` will be used.

Leader election

It is possible for the leader to add a new voter to the voters set, write the `VotersRecord` to the log and only replicate it to some of the voters in the new configuration. If the leader fails before this record has been replicated to the new voter it is possible that a new leader cannot be elected. This is because voters reject vote request from replicas that are not in the voters set. This check will be removed and replicas will reply to `Vote` request when the candidate is not in the voters set or the voting replica is not in the voters set. The candidate must still have a longer log (offset and epoch) before the voter will grant a vote to the candidate.e

Once a leader is elected, leader will propagate this information to all of the voters using the `BeginQuorumEpoch` RPC. The leader will continue to send the `BeginQuorumEpoch` requests to a voter if the voter doesn't send a `Fetch` or `FetchSnapshot` request within the "check quorum" timeout.

First leader

When a KRaft voter becomes leader it will write a `KRaftVersionRecord` and `VotersRecord` to the log if the log or the latest snapshot doesn't contain any VotersRecord. This is done to make sure that the voters set in the bootstrap snapshot gets replicated to all of the voters and to not rely on all of the voters being configured with the same bootstrapped voters set.

Automatic endpoint and directory id discovery

To improve the usability of this feature it would beneficial for the leader of the KRaft cluster metadata partition to automatically rediscover the voters' endpoints. This makes it possible for the operator to update the endpoint of a voter without having to use the `kafka-metadata-quorum` tool. When a voter becomes a follower and discovers a new leader, it will always send an UpdateVoter RPC to the leader. This request instructs the leader to update the endpoints of the matching replica id and replica uuid. When a voter becomes a leader it will also write a VotersRecord control record with the updated endpoints and `kraft.version` feature.

The directory id, or replica uuid, will behave differently. The quorum shouldn't automatically update the directory id, since different values means that the disk was replaced. For directory id, the leader will only override it if it was not previously set. This behavior is useful for when a cluster gets upgraded to a `kraft.version` greater than 0.

High watermark

As described in KIP-595, the high-watermark will be calculated using the fetch offset of the majority of the voters. When a replica is removed or added it is possible for the high-watermark to decrease. The leader will not allow the high-watermark to decrease and will guarantee that is is monotonically increasing for both the state machines and the remote replicas.

With this KIP, it is possible for the leader to not be part of the voters set when the replica removed is the leader. In this case the leader will continue to handle Fetch and FetchSnapshot request as normal but it will not count itself when computing the high watermark.

Snapshots

The snapshot generation code needs to be extended to include these new KRaft specific control record for VotersRecord. Before this KIP the snapshot didn't include any KRaft generated control records. When the state machine (controller or broker) calls RaftClient.createSnapshot, the KRaft client will write the SnapshotHeaderRecord, KRaftVersionRecord and VotersRecord controller records at the beginning of the snapshot.

Note that the current KRaft implementation already writes the SnapshotHeaderRecord. The new KRaft implementation will write `KRraftVersionRecord` and `VotersRecord`, if the `kraft.version` is greater than 0.

To efficiently implement this feature the KRaft implementation will keep track of all voters sets between the latest snapshot and the LEO. The replicas will append the offset and voters set as it reads VotersRecord control records from the snapshot or log. Whenever the replica generates a snapshot for a given offset it can remove any additional entry that has a smaller offset.

Internal listener

This KIP requires that the KRaft implementation now read and decode uncommitted data from log and snapshot to discover the voters set. This also means that the KRaft implementation needs to handle this uncommitted voter sets getting truncated and removed from the log. As the internal listener reads new voters sets and endpoints it needs to update the network client with this new information.

Controller auto joinning

To make it easier for users to operate a KRaft controller cluster, they can have KRaft controllers automatically join the cluster's voters set by setting the `controller.quorum.auto.join.enable` to `true`. In this case, the controller will remove any voter in the voters set that matches its replica id but doesn't match its directory id (replica uuid) by sending the `RemoveVoter` RPC. Once the controller has remove all duplicate replica uuid it will add itself to the voters set by sending a `AddVoter` RPC to the leader, if its replica id and replica uuid is not in the voters set. The `TimoutMs` for the `AddVoter` RPC will be set to 30 seconds.

For an example, imagine that the user is trying to create KRaft controller quorum with 3 voters (1, 2, 3).

On controller 1 the user runs:

```
kafka-storage format --cluster-id ABC --release-version "3.8" --standalone --config ...
```

and starts the controller. When controller 1 starts it sees that is already in the voters set so it doesn't perform any AddVoter and RemoveVoter RPC. Controller 1 eventually becomes leader.

On controller 2 and 3 the user runs:

```
kafka-storage format --cluster-id ABC --release-version "3.8" --config ...
```

and starts the controllers. Notice that neither `--standalone` or `--controller-quorum-voters` is used for controller 2 and 3 so the controllers start as observers. These controllers will discover the leader using `controller.quorum.bootstrap.servers` and will use the `RemoveVoter` and `AddVoter` RPC as described in the beginning of this section.

Public Interfaces

Configuration

There only two configurations for this feature.

controller.quorum.voters

This is an existing configuration. This configuration describes the state of the quorum and will only be used if the kraft.version feature is 0.

controller.quorum.bootstrap.servers

This is a list of nodes that brokers and new controllers can use to discover the quorum leader. Brokers and new controllers (observers) will send Fetch requests to all of the nodes in this configuration until they discover the quorum leader and the Fetch request succeeds. The quorum voters and their configuration will be learned by fetching and reading the records from the log and snapshot. This includes committed and uncommitted records.

If this configuration is specified, observers will not use the `controller.quorum.voters` endpoints to discover the leader.

controller.quorum.auto.join.enable

Controls whether a KRaft controller should automatically join the cluster metadata partition for its cluster id. If the configuration is set to true the controller must be stopped before removing the controller with `kafka-metadata-quorum remove-controller`. The default value is false.

Log and snapshot control records

A few new control records will be added to the log and snapshot of a KRaft partition.

KRaftVersionRecord

Control record for recording the latest KRaft protocol version. The KRaft protocol version is used to determine which version of LeaderChangeMessage to write. It is also used to determine if the VotersRecord control record can be written to the topic partition.

The `ControlRecordType` is **TBD** and will be updated when the code is committed to Kafka.

```
{
  "type": "data",
  "name": "KRaftVersionRecord",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "Version", "type": "int16", "versions": "0+",
      "about": "The version of the kraft version record" },
    { "name": "KRaftVersion", "type": "int16", "versions": "0+",
      "about": "The kraft protocol version" }
  ]
}
```

Handling

When KRaft replicas read this record from the log they will update their finalized `kraft.version` once the HWM shows this record as committed.

LeaderChangeMessage

Add an optional VoterUuid to Voter. This change is not needed for correctness but it is nice to have for tracing and debugging. The leader will write version 0 if the `kraft.version` is 0. The leader will write version 1 if the `kraft.version` is 1.

```
git diff upstream/trunk clients/src/main/resources/common/message/LeaderChangeMessage.json
diff --git a/clients/src/main/resources/common/message/LeaderChangeMessage.json b/clients/src/main/resources
/common/message/LeaderChangeMessage.json
index fdd7733388..2b019a2a80 100644
--- a/clients/src/main/resources/common/message/LeaderChangeMessage.json
+++ b/clients/src/main/resources/common/message/LeaderChangeMessage.json
@@ -16,7 +16,7 @@
 {
   "type": "data",
   "name": "LeaderChangeMessage",
-  "validVersions": "0",
+  "validVersions": "0-1",
   "flexibleVersions": "0+",
   "fields": [
     {"name": "Version", "type": "int16", "versions": "0+",
@@ -30,7 +30,8 @@
   ],
   "commonStructs": [
     { "name": "Voter", "versions": "0+", "fields": [
-      {"name": "VoterId", "type": "int32", "versions": "0+"}
+      { "name": "VoterId", "type": "int32", "versions": "0+" },
+      { "name": "VoterUuid", "type": "uuid", "versions": "1+" }
     ]}
   ]
 }
```

VotersRecord

A control record for specifying the latest and entire voters set. This record will be written to the generated snapshot and will specify the voters set at the snapshot id's end offset.

This record will also be written to the log by the leader when handling the `AddVoter`, `RemoveVoter` and `UpdateVoter` RPCs. When handling these RPCs the leader will make sure that the difference in voters is at most one addition or one removal. This is needed to satisfy the invariant that voter changes are committed by the old voters set and new voters set.

This record will also be written by the leader to the log if it has never been written to the log in the past. This semantic is nice to have to consistently replicate the bootstrapping snapshot, at `00000000000000000000-0000000000.checkpoint`, at the leader to all of the voters.

The `ControlRecordType` is **TBD** and will be updated when the code is committed to Kafka.

```
{
  "type": "data",
  "name": "VotersRecord",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "Version", "type": "int16", "versions": "0+",
      "about": "The version of the voters record" },
    { "name": "Voters", "type": "[]Voter", "versions": "0+", "fields": [
      { "name": "VoterId", "type": "int32", "versions": "0+", "entityType": "brokerId",
        "about": "The replica id of the voter in the topic partition" },
      { "name": "VoterUuid", "type": "uuid", "versions": "0+",
        "about": "The directory id of the voter in the topic partition" },
      { "name": "EndPoints", "type": "[]Endpoint", "versions": "0+",
        "about": "The endpoint that can be used to communicate with the voter", "fields": [
        { "name": "Name", "type": "string", "versions": "0+", "mapKey": true,
          "about": "The name of the endpoint" },
        { "name": "Host", "type": "string", "versions": "0+",
          "about": "The hostname" },
        { "name": "Port", "type": "uint16", "versions": "0+",
          "about": "The port" }
      ]},
      { "name": "KRaftVersionFeature", "type": "KRaftVersionFeature", "versions": "0+",
        "about": "The range of versions of the protocol that the replica supports", "fields": [
        { "name": "MinSupportedVersion", "type": "int16", "versions": "0+",
          "about": "The minimum supported KRaft protocol version" },
        { "name": "MaxSupportedVersion", "type": "int16", "versions": "0+",
          "about": "The maximum supported KRaft protocol version" }
      ]}
    ]}
  ]
}
```

Handling

KRaft replicas will read all of the control records in the snapshot and the log irrespective of the commit state and HWM. When a replica encounters a `VotersRecord` it will replace the current voters set.

If the local replica is the leader and it is getting removed, the replica will stay leader until the `VotersRecord` gets committed or the epoch advances (which forces it to lose leadership).

If the local replica is getting added to the voters set, it will allow the transition to prospective candidate when the fetch timer expires. The fetch timer is reset whenever it receives a successful `Fetch` or `FetchSnapshot` response.

Quorum state

Each KRaft topic partition has a quorum state (`QuorumStateData`) that gets persisted in the `quorum-state` file in the directory for the topic partition. The following changes will be made to this state:

1. ClusterId will get removed in version 1. This field is not used because the cluster id is persisted in `<metadata.log.dir>/meta.properties`.
2. AppliedOffset will get removed in version 1. This field is not used in version 0.
3. VotedUuid will get added in version 1. The voting replica will persist both the voter ID and UUID of the candidate for which it voted.
4. CurrentVoters will get removed in version 1. This field was only used to validate that the controller.quorum.voters value didn't change. In kraft. version 1 the set of voters will be stored in the snapshot and log instead.

Version 0 of this data will get written if the `kraft.version` is 0. Version 1 of this data will get written if the `kraft.version` is 1.

If a candidate sends a vote request with a replica UUID (version 2 of the RPC) but the voter's kraft.version is at 0, the voter will not persist the voted UUID and use version 0 of quorum data. The voter will continue to track the voted UUID in memory. This mean that after a restart the voter can vote for a different UUID but this should be rare and can only happing while the protocol is getting upgraded.

```
% git diff upstream/trunk raft/src/main/resources/common/message/QuorumStateData.json
diff --git a/raft/src/main/resources/common/message/QuorumStateData.json b/raft/src/main/resources/common
/message/QuorumStateData.json
index d71a32c75d..1ae14ec843 100644
--- a/raft/src/main/resources/common/message/QuorumStateData.json
+++ b/raft/src/main/resources/common/message/QuorumStateData.json
@@ -16,19 +16,17 @@
 {
   "type": "data",
   "name": "QuorumStateData",
-  "validVersions": "0",
+  "validVersions": "0-1",
   "flexibleVersions": "0+",
   "fields": [
-    {"name": "ClusterId", "type": "string", "versions": "0+"},
-    {"name": "LeaderId", "type": "int32", "versions": "0+", "default": "-1"},
-    {"name": "LeaderEpoch", "type": "int32", "versions": "0+", "default": "-1"},
-    {"name": "VotedId", "type": "int32", "versions": "0+", "default": "-1"},
-    {"name": "AppliedOffset", "type": "int64", "versions": "0+"},
-    {"name": "CurrentVoters", "type": "[]Voter", "versions": "0+", "nullableVersions": "0+"}
-  ],
-  "commonStructs": [
-    { "name": "Voter", "versions": "0+", "fields": [
-      {"name": "VoterId", "type": "int32", "versions": "0+"}
+    { "name": "ClusterId", "type": "string", "versions": "0" },
+    { "name": "LeaderId", "type": "int32", "versions": "0+", "default": "-1" },
+    { "name": "LeaderEpoch", "type": "int32", "versions": "0+", "default": "-1" },
+    { "name": "VotedId", "type": "int32", "versions": "0+", "default": "-1" },
+    { "name": "VotedUuid", "type": "uuid", "versions": "1+" },
+    { "name": "AppliedOffset", "type": "int64", "versions": "0" },
+    { "name": "CurrentVoters", "type": "[]Voter", "versions": "0", "nullableVersions": "0", "fields": [
+      { "name": "VoterId", "type": "int32", "versions": "0" }
     ]}
   ]
 }
```

RPCs

UpdateFeatures

The schema and version for this API will not change but the handling of the RPC will be updated to handle the `kraft.version` feature.

Handling

The controller node will forward any changes to the `kraft.version` to the RaftClient.

When upgrading the version, the RaftClient will:

1. Verify that the current set of voters support the new version. The leader will learn the range of supported version from the UpdateVoter RPC. See that section for more details.
2. Verify that all of the observers (brokers) support the new version. The leader will learn range of supported version from the QuorumController. The quorum controller will update the RaftClient whenever the broker registration changes.
3. If supported, the RaftClient will atomically write a control record batch to the log with all of the KRaftVersion and the `VotersRecord` control records. The VotersRecord controller records only need to get written when updating from version 0 to version 1.
4. Reply to the RPC when the control record batch is committed.

Downgrading to the kraft.version from 1 to 0 will be rejected with INVALID_UPDATE_VERSION.

AddVoter

This RPC can be sent by an administrative client to add a voter to the set of voters. This RPC can be sent to a broker or controller, when sent to a broker, the broker will forward the request to the active controller.

Handling

When the leader receives an AddVoter request it will do the following:

1. Wait for the fetch offset of the replica (ID, UUID) to catch up to the log end offset of the leader.

2. Wait until there are no uncommitted VotersRecord. Note that the implementation may just return a REQUEST_TIMED_OUT error if there are pending operations.
3. Wait for the LeaderChangeMessage control record from the current epoch to get committed. Note that the implementation may just return a REQUEST_TIMED_OUT error if there are pending operations.
4. Send an ApiVersions RPC to the first listener to discover the supported `kraft.version` of the new voter.
5. Check that the new voter supports the current `kraft.version`.
6. Append the updated VotersRecord to the log.
7. The KRaft internal listener will read this record from the log and add the voter to the voters set.
8. Wait for the VotersRecord to commit using the majority of new voters set. Return a REQUEST_TIMED_OUT error if it doesn't succeed in time.
9. Send the AddVoter response to the client.

In 1., the leader needs to wait for the replica to catch up because when the VotersRecord is appended to the log, the set of voter changes. If the new voter is too far behind then it can take some time for it to reach the HWM. During this time the leader cannot commit data and the quorum will be unavailable from the perspective of the state machine. We mitigate this by waiting for the new replica to catch up before adding it to the set of voters.

Here is an example, where waiting for the new voter to catch up to the LEO is helpful. Let's assume that there are 3 voters with only two of them at the LEO or HWM.

```
Voter 1: LEO = 200
Voter 2: LEO = 200
Voter 3: LEO = 100
```

In this example the HWM is 200. To advance the HWM past 200 only voter 1 and 2 need to Fetch the new records. For voter 3 to be counted towards advancing the HWM it needs to Fetch to or past offset 200 (100 records in this example).

At this point if the user decides to add a new voter (4) to the voters set and the leader doesn't wait for voter (4) to catch up to the LEO, the voters set may become:

```
Voter 1: LEO = 200
Voter 2: LEO = 200
Voter 3: LEO = 100
Voter 4: LEO = 0
```

In this state the majority must include either voter 3, voter 4 or both voters 3 and 4. This means that leader cannot advance the HWM until at least either voter 3 or voter 4 have caught up to the HWM. In other words, the KRaft topic partition won't be able to commit new records until at least one of those two voters (3 or 4) are caught up to the HWM/LEO.

In 3., the leader will wait for its current epoch to commit by waiting for the LeaderChangeMessage to commit. This is required to guarantee that the two competing voters sets, the one from a previous leader and the one from the current leader, only differ by at most one voter. Waiting for the current epoch to commit means that there cannot be some other competing voters set from another leader that can later override this leader's new voters set. See bug in single-server membership changes for more details on this.

In 7., the new replica will be part of the quorum so the leader will start sending BeginQuorumEpoch requests to this replica. It is possible that the new replica has not yet replicated and applied this VotersRecord so it doesn't know that it is a voter for this topic partition. The new replica will accept the BeginQuorumEpoch RPC even if it is doesn't believe it is a member of the voters set.

The replica will return the following errors:

1. NOT_LEADER_FOR_PARTITION - when the request is sent to a replica that is not the leader.
2. DUPLICATE_VOTER - when the request contains a replica id is already in the committed voters set. Note that this means that duplicate replica ids are not allowed. This is useful to make automatic voter addition safer.
3. UNSUPPORTED_VERSION - when the `kraft.version` is not greater than 1.
4. INVALID_REQUEST - when the new voter doesn't support the current `kraft.version`.
5. REQUEST_TIMED_OUT - when the new voter didn't catch-up to the LEO in the time specified in the request.

Request

```
{
  "apiKey": 76,
  "type": "request",
  "listeners": ["controller", "broker"],
  "name": "AddVoterRequest",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "ClusterId", "type": "string", "versions": "0+" },
    { "name": "TimeoutMs", "type": "int32", "versions": "0+" },
    { "name": "TopicName", "type": "string", "versions": "0+", "entityType": "topicName",
      "about": "The name of the topic" },
    { "name": "TopicId", "type": "uuid", "versions": "0+",
      "about": "The unique topic ID" },
    { "name": "Partition", "type": "int32", "versions": "0+",
      "about": "The partition index" },
    { "name": "VoterId", "type": "int32", "versions": "0+",
      "about": "The replica id of the voter getting added to the topic partition" },
    { "name": "VoterUuid", "type": "uuid", "versions": "0+",
      "about": "The directory id of the voter getting added to the topic partition" },
    { "name": "Listeners", "type": "[]Listener", "versions": "0+",
      "about": "The endpoints that can be used to communicate with the voter", "fields": [
      { "name": "Name", "type": "string", "versions": "0+", "mapKey": true,
        "about": "The name of the endpoint" },
      { "name": "Host", "type": "string", "versions": "0+",
        "about": "The hostname" },
      { "name": "Port", "type": "uint16", "versions": "0+",
        "about": "The port" }
    ]}
  ]
}
```

Response

```
{
  "apiKey": 76,
  "type": "response",
  "name": "AddVoterResponse",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "ErrorCode", "type": "int16", "versions": "0+",
      "about": "The error code, or 0 if there was no error" },
    { "name": "ErrorMessage", "type": "string", "versions": "0+", "nullableVersions": "0+", "ignorable": true,
      "about": "The error message, or null if there was no error." },
    { "name": "CurrentLeader", "type": "LeaderIdAndEpoch", "versions": "0+", "taggedVersions": "0+", "tag": 0,
"fields": [
      { "name": "LeaderId", "type": "int32", "versions": "0+", "default": "-1", "entityType" : "brokerId",
        "about": "The replica id of the current leader or -1 if the leader is unknown" },
      { "name": "LeaderEpoch", "type": "int32", "versions": "0+", "default": "-1",
        "about": "The latest known leader epoch" }
    ]},
    { "name": "NodeEndpoint", "type": "NodeEndpoint", "versions": "0+", "taggedVersions": "0+", "tag": 1,
      "about": "Endpoint for current leader of the topic partition", "fields": [
      { "name": "Host", "type": "string", "versions": "0+", "about": "The node's hostname" },
      { "name": "Port", "type": "int32", "versions": "0+", "about": "The node's port" }
    ]}
  ]
}
```

RemoveVoter

This RPC can be sent by an administrative client to remove a voter from the set of voters. This RPC can be sent to a broker or controller. The broker will forward the request to the controller.

Handling

When the leader receives a RemoveVoter request it will do the following:

1. Wait until there are no uncommitted VotersRecord. Note that the implementation may just return a REQUEST_TIMED_OUT error if there are pending operations.
2. Wait for the LeaderChangeMessage control record from the current epoch to get committed. Note that the implementation may just return a REQUEST_TIMED_OUT error if there are pending operations.
3. Append the VotersRecord to the log with the updated voters set.
4. The KRaft internal listener will read this record from the log and remove the voter from the voters set.
5. Wait for the VotersRecord to commit using the majority of new configuration. Return a REQUEST_TIMED_OUT error if it doesn't succeed in time.
6. Send the RemoveVoter response to the client.
7. Resign by sending EndQuorumEpoch RPCs if the removed replica is the leader.

In 3. and 4. it is possible for the VotersRecord would remove the current leader from the voters set. In this case the leader needs to allow Fetch and FetchSnapshot requests from replicas. The leader should not count itself when determining the majority and determining if records have been committed.

The replica will return the following errors:

1. NOT_LEADER_FOR_PARTITION - when the request is sent to a replica that is not the leader.
2. VOTER_NOT_FOUND - when the request contains a replica ID and UUID that is already not in the committed voters set.
3. UNSUPPORTED_VERSION - when the `kraft.version` is not greater than 1.
4. REQUEST_TIMED_OUT

Request

```
{
  "apiKey": 77,
  "type": "request",
  "listeners": ["controller", "broker"],
  "name": "RemoveVoterRequest",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "ClusterId", "type": "string", "versions": "0+" },
    { "name": "TopicName", "type": "string", "versions": "0+", "entityType": "topicName",
      "about": "The name of the topic" },
    { "name": "TopicId", "type": "uuid", "versions": "0+",
      "about": "The unique topic ID" },
    { "name": "Partition", "type": "int32", "versions": "0+",
      "about": "The partition index" },
    { "name": "VoterId", "type": "int32", "versions": "0+",
      "about": "The replica id of the voter getting removed from the topic partition" },
    { "name": "VoterUuid", "type": "uuid", "versions": "0+",
      "about": "The directory id of the voter getting removed from the topic partition" }
  ]
}
```

Response

```
{
  "apiKey": 77,
  "type": "response",
  "name": "RemoveVoterResponse",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "ErrorCode", "type": "int16", "versions": "0+",
      "about": "The error code, or 0 if there was no error" },
    { "name": "ErrorMessage", "type": "string", "versions": "0+", "nullableVersions": "0+", "ignorable": true,
      "about": "The error message, or null if there was no error." },
    { "name": "CurrentLeader", "type": "LeaderIdAndEpoch", "versions": "0+", "taggedVersions": "0+", "tag": 0,
"fields": [
      { "name": "LeaderId", "type": "int32", "versions": "0+", "default": "-1", "entityType" : "brokerId",
        "about": "The replica id of the current leader or -1 if the leader is unknown" },
      { "name": "LeaderEpoch", "type": "int32", "versions": "0+", "default": "-1",
        "about": "The latest known leader epoch" }
    ]},
    { "name": "NodeEndpoint", "type": "NodeEndpoint", "versions": "0+", "taggedVersions": "0+", "tag": 1,
      "about": "Endpoint for current leader of the topic partition", "fields": [
      { "name": "Host", "type": "string", "versions": "0+", "about": "The node's hostname" },
      { "name": "Port", "type": "int32", "versions": "0+", "about": "The node's port" }
    ]}
  ]
}
```

UpdateVoter

This RPC is different from AddVoter in two ways: 1. it will be sent only by voters to the latest known leader and 2. it includes both listener endpoints and `kraft.version` information. This RPC is useful to automatically update a voter's endpoints and kraft.version information without additional operator intervention.

The voter will always send the UpdateVoter RPC whenever it starts and whenever the leader changes. The voter will continue to send the UpdateVoter RPC until the call has been acknowledge by the current leader.

Handling

When the leader receives an UpdateVoter request it will do the following:

1. Wait until there are no uncommitted add or remove voter records. Note that the implementation may just return a REQUEST_TIMED_OUT error if there are pending operations.
2. Wait for the LeaderChangeMessage control record from the current epoch to get committed. Note that the implementation may just return a REQUEST_TIMED_OUT error if there are pending operations.
3. Check that the updated voter supports the current `kraft.version`.

4. Append the updated VotersRecord to the log.
5. The KRaft internal listener will read this record from the log and update the voter's information. This include updating the endpoint used by the KRaft NetworkClient.
6. Wait for the VotersRecord to commit using the majority of new voters set.
7. Send the UpdateVoter response to the client.

The replica will return the following errors:

1. NOT_LEADER_FOR_PARTITION - when the request is sent to a replica that is not the leader.
2. VOTER_NOT_FOUND - when the request contains a replica ID and UUID that is not in the committed voters set.
3. INVALID_UPDATE - when the minimum and maximum supported `kraft.version` in the request doesn't include the finalized kraft.version.
4. REQUEST_TIMED_OUT

Request

```
{
  "apiKey": 78,
  "type": "request",
  "listeners": ["controller"],
  "name": "UpdateVoterRequest",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "ClusterId", "type": "string", "versions": "0+" },
    { "name": "TopicName", "type": "string", "versions": "0+", "entityType": "topicName",
      "about": "The name of the topic" },
    { "name": "TopicId", "type": "uuid", "versions": "0+",
      "about": "The unique topic ID" },
    { "name": "Partition", "type": "int32", "versions": "0+",
      "about": "The partition index" },
    { "name": "VoterId", "type": "int32", "versions": "0+",
      "about": "The replica id of the voter getting updated in the topic partition" },
    { "name": "VoterUuid", "type": "uuid", "versions": "0+",
      "about": "The directory id of the voter getting updated in the topic partition" },
    { "name": "Listeners", "type": "[]Listener", "versions": "0+",
      "about": "The endpoint that can be used to communicate with the leader", "fields": [
      { "name": "Name", "type": "string", "versions": "0+", "mapKey": true,
        "about": "The name of the endpoint" },
      { "name": "Host", "type": "string", "versions": "0+",
        "about": "The hostname" },
      { "name": "Port", "type": "uint16", "versions": "0+",
        "about": "The port" }
    ]},
    { "name": "KRaftVersionFeature", "type": "KRaftVersionFeature", "versions": "0+",
      "about": "The range of versions of the protocol that the replica supports", "fields": [
      { "name": "MinSupportedVersion", "type": "int16", "versions": "0+",
        "about": "The minimum supported KRaft protocol version" },
      { "name": "MaxSupportedVersion", "type": "int16", "versions": "0+",
        "about": "The maximum supported KRaft protocol version" }
    ]}
  ]
}
```

Response

```
{
  "apiKey": 78,
  "type": "response",
  "name": "UpdateVoterResponse",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "ErrorCode", "type": "int16", "versions": "0+",
      "about": "The error code, or 0 if there was no error" },
    { "name": "CurrentLeader", "type": "LeaderIdAndEpoch", "versions": "0+", "taggedVersions": "0+", "tag": 0,
"fields": [
      { "name": "LeaderId", "type": "int32", "versions": "0+", "default": "-1", "entityType" : "brokerId",
        "about": "The replica id of the current leader or -1 if the leader is unknown" },
      { "name": "LeaderEpoch", "type": "int32", "versions": "0+", "default": "-1",
        "about": "The latest known leader epoch" }
    ]},
    { "name": "NodeEndpoint", "type": "NodeEndpoint", "versions": "0+", "taggedVersions": "0+", "tag": 1,
      "about": "Endpoint for current leader of the topic partition", "fields": [
      { "name": "Host", "type": "string", "versions": "0+", "about": "The node's hostname" },
      { "name": "Port", "type": "int32", "versions": "0+", "about": "The node's port" }
    ]}
  ]
}
```

Vote

Handling

Since the set of voters can change and not all replicas know the latest voters set, handling of Vote request needs to be relaxed from what was defined and implemented for KIP-595. KRaft replicas will accept Vote requests from all replicas. Candidate replicas don't need to be in the voters' voters set to receive a vote. This is needed to be able to elect a leader from the new voters set even though the new voters set hasn't been replicated to all of its voters.

When the leader removes a voter from the voters set it is possible for the removed voter's Fetch timeout to expire before the replica has replicated the latest VotersRecord. If this happens removed replica will become a candidate, increase its epoch and eventually force the leader to change. To avoid this scenario this KIP is going to rely on KIP-996: Pre-Vote to fenced the removed replica from increasing its epoch:

> *When servers receive VoteRequests with the `PreVote` field set to `true`, they will respond with `VoteGranted` set to*
>
> - *`true` **if they are not a Follower*** and *the epoch and offsets in the Pre-Vote request satisfy the same requirements as a standard vote*
> - *`false` if otherwise*

The voter will persist both the candidate ID and UUID in the quorum state so that it only votes for at most one candidate for a given epoch.

The replica will return the following new errors:

1. INVALID_REQUEST - when the voter ID and UUID doesn't match the local ID and UUID.
2. UNSUPPORTED_VERSION - when a non-empty candidate UUID is specified but the voter doesn't support kraft.version 1.

The replica will not return the following errors anymore:

1. INCONSISTENT_VOTER_SET - because the voting replica will not validate that either the candidate or the voter are voters.

Request

```
git diff upstream/trunk clients/src/main/resources/common/message/VoteRequest.json
diff --git a/clients/src/main/resources/common/message/VoteRequest.json b/clients/src/main/resources/common
/message/VoteRequest.json
index 35583a790b..ff808cbafe 100644
--- a/clients/src/main/resources/common/message/VoteRequest.json
+++ b/clients/src/main/resources/common/message/VoteRequest.json
@@ -18,11 +18,13 @@
   "type": "request",
   "listeners": ["controller"],
   "name": "VoteRequest",
-  "validVersions": "0",
+  "validVersions": "0-1",
   "flexibleVersions": "0+",
   "fields": [
     { "name": "ClusterId", "type": "string", "versions": "0+",
       "nullableVersions": "0+", "default": "null"},
+    { "name": "VoterId", "type": "int32", "versions": "1+", "ignorable": true, "default": "-1", "entityType":
"brokerId",
+      "about": "The replica id of the voter receiving the request" },
     { "name": "Topics", "type": "[]TopicData",
       "versions": "0+", "fields": [
       { "name": "TopicName", "type": "string", "versions": "0+", "entityType": "topicName",
@@ -34,14 +36,16 @@
         { "name": "CandidateEpoch", "type": "int32", "versions": "0+",
           "about": "The bumped epoch of the candidate sending the request"},
         { "name": "CandidateId", "type": "int32", "versions": "0+", "entityType": "brokerId",
-          "about": "The ID of the voter sending the request"},
+          "about": "The replica id of the voter sending the request"},
+        { "name": "CandidateUuid", "type": "uuid", "versions": "1+",
+          "about": "The directory id of the voter sending the request" },
+        { "name": "VoterUuid", "type": "uuid", "versions": "1+",
+          "about": "The directory id of the voter receiving the request to vote, empty uuid if unknown" },
         { "name": "LastOffsetEpoch", "type": "int32", "versions": "0+",
           "about": "The epoch of the last record written to the metadata log"},
         { "name": "LastOffset", "type": "int64", "versions": "0+",
           "about": "The offset of the last record written to the metadata log"}
       ]}
     ]}
   ]
 }
```

Response

```
git diff upstream/trunk clients/src/main/resources/common/message/VoteResponse.json
diff --git a/clients/src/main/resources/common/message/VoteResponse.json b/clients/src/main/resources/common
/message/VoteResponse.json
index b92d0070c1..21d0aa5312 100644
--- a/clients/src/main/resources/common/message/VoteResponse.json
+++ b/clients/src/main/resources/common/message/VoteResponse.json
@@ -17,7 +17,7 @@
   "apiKey": 52,
   "type": "response",
   "name": "VoteResponse",
-  "validVersions": "0",
+  "validVersions": "0-1",
   "flexibleVersions": "0+",
   "fields": [
     { "name": "ErrorCode", "type": "int16", "versions": "0+",
@@ -37,9 +37,14 @@
           "about": "The latest known leader epoch"},
         { "name": "VoteGranted", "type": "bool", "versions": "0+",
           "about": "True if the vote was granted and false otherwise"}
       ]}
     ]},
+    { "name": "NodeEndpoints", "type": "[]NodeEndpoint", "versions": "1+", "taggedVersions": "1+", "tag": 0,
+      "about": "Endpoints for all current-leaders enumerated in PartitionData", "fields": [
+      { "name": "NodeId", "type": "int32", "versions": "1+",
+        "mapKey": true, "entityType": "brokerId", "about": "The ID of the associated node"},
+      { "name": "Host", "type": "string", "versions": "1+", "about": "The node's hostname" },
+      { "name": "Port", "type": "int32", "versions": "1+", "about": "The node's port" }
+    ]}
   ]
 }
```

BeginQuorumEpoch

Handling

When handling the BeginQuorumEpoch request the replica will accept the request if the LeaderEpoch is equal or greater than their epoch. The receiving replica will not check if the new leader or itself is in the voters set. This change is required because the receiving replica may not have fetched the latest voters set.

The BeginQuorumEpoch RPC will be used by the leader to propagate the leader endpoint to all of the voters. This is needed so that all voters have the necessary information to send Fetch and FetchSnaphsot requests.

The **check quorum** algorithm implemented by the leader needs to be extended so that BeginQuorumEpoch request are sent to any voter that is not actively fetching (Fetch and FetchSnapshot) from the leader.

The replica will return the following new errors:

1. INVALID_REQUEST - when the voter ID and UUID doesn't match the local ID and UUID.

The replica will not return the following errors anymore:

1. INCONSISTENT_VOTER_SET - because the voting replica will not validate that either the candidate or the voter are voters.

Request

```
git diff upstream/trunk clients/src/main/resources/common/message/BeginQuorumEpochRequest.json
diff --git a/clients/src/main/resources/common/message/BeginQuorumEpochRequest.json b/clients/src/main/resources
/common/message/BeginQuorumEpochRequest.json
index d9d6d92c88..edd128fb8c 100644
--- a/clients/src/main/resources/common/message/BeginQuorumEpochRequest.json
+++ b/clients/src/main/resources/common/message/BeginQuorumEpochRequest.json
@@ -18,24 +18,35 @@
   "type": "request",
   "listeners": ["controller"],
   "name": "BeginQuorumEpochRequest",
-  "validVersions": "0",
-  "flexibleVersions": "none",
+  "validVersions": "0-1",
+  "flexibleVersions": "1+",
   "fields": [
     { "name": "ClusterId", "type": "string", "versions": "0+",
       "nullableVersions": "0+", "default": "null"},
+    { "name": "VoterId", "type": "int32", "versions": "1+", "entityType": "brokerId",
+      "about": "The voter ID of the receiving replica" },
     { "name": "Topics", "type": "[]TopicData",
       "versions": "0+", "fields": [
       { "name": "TopicName", "type": "string", "versions": "0+", "entityType": "topicName",
         "about": "The topic name" },
       { "name": "Partitions", "type": "[]PartitionData",
         "versions": "0+", "fields": [
         { "name": "PartitionIndex", "type": "int32", "versions": "0+",
           "about": "The partition index" },
+        { "name": "VoterUuid", "type": "uuid", "versions": "1+",
+          "about": "The directory id of the receiving replica" },
         { "name": "LeaderId", "type": "int32", "versions": "0+", "entityType": "brokerId",
           "about": "The ID of the newly elected leader"},
         { "name": "LeaderEpoch", "type": "int32", "versions": "0+",
           "about": "The epoch of the newly elected leader"}
       ]}
+    ]},
+    { "name": "NodeEndpoint", "type": "NodeEndpoint", "versions": "1+", "taggedVersions": "1+", "tag": 0,
+      "about": "Endpoint for the leader", "fields": [
+      { "name": "NodeId", "type": "int32", "versions": "1+",
+        "mapKey": true, "entityType": "brokerId", "about": "The ID of the associated node" },
+      { "name": "Host", "type": "string", "versions": "1+", "about": "The node's hostname" },
+      { "name": "Port", "type": "int32", "versions": "1+", "about": "The node's port" }
     ]}
   ]
 }
```

Response

```
git diff upstream/trunk clients/src/main/resources/common/message/BeginQuorumEpochResponse.json
diff --git a/clients/src/main/resources/common/message/BeginQuorumEpochResponse.json b/clients/src/main
/resources/common/message/BeginQuorumEpochResponse.json
index 4b7d7f5a95..12639bba2f 100644
--- a/clients/src/main/resources/common/message/BeginQuorumEpochResponse.json
+++ b/clients/src/main/resources/common/message/BeginQuorumEpochResponse.json
@@ -17,25 +17,32 @@
   "apiKey": 53,
   "type": "response",
   "name": "BeginQuorumEpochResponse",
-  "validVersions": "0",
-  "flexibleVersions": "none",
+  "validVersions": "0-1",
+  "flexibleVersions": "1+",
   "fields": [
     { "name": "ErrorCode", "type": "int16", "versions": "0+",
       "about": "The top level error code"},
     { "name": "Topics", "type": "[]TopicData",
       "versions": "0+", "fields": [
       { "name": "TopicName", "type": "string", "versions": "0+", "entityType": "topicName",
         "about": "The topic name" },
       { "name": "Partitions", "type": "[]PartitionData",
         "versions": "0+", "fields": [
         { "name": "PartitionIndex", "type": "int32", "versions": "0+",
           "about": "The partition index" },
         { "name": "ErrorCode", "type": "int16", "versions": "0+"},
         { "name": "LeaderId", "type": "int32", "versions": "0+", "entityType": "brokerId",
           "about": "The ID of the current leader or -1 if the leader is unknown"},
         { "name": "LeaderEpoch", "type": "int32", "versions": "0+",
           "about": "The latest known leader epoch"}
       ]}
+    ]},
+    { "name": "NodeEndpoints", "type": "[]NodeEndpoint", "versions": "1+", "taggedVersions": "1+", "tag": 0,
+      "about": "Endpoints for all leaders enumerated in PartitionData", "fields": [
+      { "name": "NodeId", "type": "int32", "versions": "1+",
+        "mapKey": true, "entityType": "brokerId", "about": "The ID of the associated node" },
+      { "name": "Host", "type": "string", "versions": "1+", "about": "The node's hostname" },
+      { "name": "Port", "type": "int32", "versions": "1+", "about": "The node's port" }
     ]}
   ]
 }
```

EndQuorumEpoch

In addition to the current cases where the leader resigns and sends an EndQuorumEpoch RPC, the leader will also resign and send this RPC once a VotersRecord has committed and the leader is not part of the latest voters set.

Handling

This request will be handle similar to how it is described in KIP-595. The receiving replica will take the directory id (candidate uuid) into account when computing the fetch timeout exponential backoff.

Request

PreferredSuccessor which is an array is replica ids, will be replaced by PreferredCandidates which is an array of the tuple candidate id and candidate uuid.

```
git diff upstream/trunk clients/src/main/resources/common/message/EndQuorumEpochRequest.json
diff --git a/clients/src/main/resources/common/message/EndQuorumEpochRequest.json b/clients/src/main/resources
/common/message/EndQuorumEpochRequest.json
index a6e4076412..d9122fa930 100644
--- a/clients/src/main/resources/common/message/EndQuorumEpochRequest.json
+++ b/clients/src/main/resources/common/message/EndQuorumEpochRequest.json
@@ -18,8 +18,8 @@
   "type": "request",
   "listeners": ["controller"],
   "name": "EndQuorumEpochRequest",
-  "validVersions": "0",
-  "flexibleVersions": "none",
+  "validVersions": "0-1",
+  "flexibleVersions": "1+",
   "fields": [
     { "name": "ClusterId", "type": "string", "versions": "0+",
       "nullableVersions": "0+", "default": "null"},
@@ -35,8 +35,13 @@
           "about": "The current leader ID that is resigning"},
         { "name": "LeaderEpoch", "type": "int32", "versions": "0+",
           "about": "The current epoch"},
-        { "name": "PreferredSuccessors", "type": "[]int32", "versions": "0+",
-          "about": "A sorted list of preferred successors to start the election"}
+        { "name": "PreferredSuccessors", "type": "[]int32", "versions": "0",
+          "about": "A sorted list of preferred successors to start the election" },
+        { "name": "PreferredCandidates", "type": "[]ReplicaInfo", "versions": "1+",
+          "about": "A sorted list of preferred successors to start the election", "fields": [
+          { "name": "CandidateId", "type": "int32", "versions": "1+", "entityType": "brokerId" },
+          { "name": "CandidateUuid", "type": "uuid", "versions": "1+" }
+        ]}
       ]}
     ]}
   ]
```

Response

```
git diff upstream/trunk clients/src/main/resources/common/message/EndQuorumEpochResponse.json
diff --git a/clients/src/main/resources/common/message/EndQuorumEpochResponse.json b/clients/src/main/resources
/common/message/EndQuorumEpochResponse.json
index cd23247045..0d5d61b7e7 100644
--- a/clients/src/main/resources/common/message/EndQuorumEpochResponse.json
+++ b/clients/src/main/resources/common/message/EndQuorumEpochResponse.json
@@ -17,8 +17,8 @@
   "apiKey": 54,
   "type": "response",
   "name": "EndQuorumEpochResponse",
-  "validVersions": "0",
-  "flexibleVersions": "none",
+  "validVersions": "0-1",
+  "flexibleVersions": "1+",
   "fields": [
     { "name": "ErrorCode", "type": "int16", "versions": "0+",
       "about": "The top level error code."},
@@ -36,6 +36,13 @@
         { "name": "LeaderEpoch", "type": "int32", "versions": "0+",
           "about": "The latest known leader epoch"}
       ]}
+    ]},
+    { "name": "NodeEndpoints", "type": "[]NodeEndpoint", "versions": "1+", "taggedVersions": "1+", "tag": 0,
+      "about": "Endpoints for all leaders enumerated in PartitionData", "fields": [
+      { "name": "NodeId", "type": "int32", "versions": "1+",
+        "mapKey": true, "entityType": "brokerId", "about": "The ID of the associated node" },
+      { "name": "Host", "type": "string", "versions": "1+", "about": "The node's hostname" },
+      { "name": "Port", "type": "int32", "versions": "1+", "about": "The node's port" }
     ]}
   ]
 }
```

Fetch

Handling

The leader will track the fetched offset for the replica tuple (ID and UUID). Replicas are uniquely identified by their ID and UUID so their state will be tracked using their ID and UUID.

When removing the leader from the voters set, it will remain the leader for that epoch until the VotersRecord gets committed. This means that the leader needs to allow replicas (voters and observers) to fetch from the leader even if it is not part of the voters set. This also means that if the leader is not part of the voters set it should not include itself when computing the committed offset (also known as the high-watermark) and when checking that the quorum is alive.

Request

Add the replica UUID (directory id) to the Fetch request. This is needed so that the leader can correctly track persisted offsets all of the voters.

```
git diff upstream/trunk clients/src/main/resources/common/message/FetchRequest.json
diff --git a/clients/src/main/resources/common/message/FetchRequest.json b/clients/src/main/resources/common
/message/FetchRequest.json
index 235357d004..ff86469831 100644
--- a/clients/src/main/resources/common/message/FetchRequest.json
+++ b/clients/src/main/resources/common/message/FetchRequest.json
@@ -55,7 +55,9 @@
   // deprecate the old ReplicaId field and set its default value to -1. (KIP-903)
   //
   // Version 16 is the same as version 15 (KIP-951).
-  "validVersions": "0-16",
+  //
+  // Version 17 adds directory id support from KIP-853
+  "validVersions": "0-17",
   "deprecatedVersions": "0-3",
   "flexibleVersions": "12+",
   "fields": [
@@ -100,7 +102,9 @@
         { "name": "LogStartOffset", "type": "int64", "versions": "5+", "default": "-1", "ignorable": true,
          "about": "The earliest available offset of the follower replica.  The field is only used when the
request is sent by the follower."},
         { "name": "PartitionMaxBytes", "type": "int32", "versions": "0+",
-          "about": "The maximum bytes to fetch from this partition.  See KIP-74 for cases where this limit may
not be honored." }
+          "about": "The maximum bytes to fetch from this partition.  See KIP-74 for cases where this limit may
not be honored." },
+        { "name": "ReplicaUuid", "type": "uuid", "versions": "17+", "taggedVersions": "17+", "tag": 0,
+          "about": "The directory id of the follower fetching" }
       ]}
     ]},
     { "name": "ForgottenTopicsData", "type": "[]ForgottenTopic", "versions": "7+", "ignorable": false,
```

Response

```
git diff upstream/trunk clients/src/main/resources/common/message/FetchResponse.json
diff --git a/clients/src/main/resources/common/message/FetchResponse.json b/clients/src/main/resources/common
/message/FetchResponse.json
index e5f49ba6fd..b432a79719 100644
--- a/clients/src/main/resources/common/message/FetchResponse.json
+++ b/clients/src/main/resources/common/message/FetchResponse.json
@@ -47,7 +47,7 @@
   // Version 15 is the same as version 14 (KIP-903).
   //
   // Version 16 adds the 'NodeEndpoints' field (KIP-951).
-  "validVersions": "0-16",
+  "validVersions": "0-17",
   "flexibleVersions": "12+",
   "fields": [
     { "name": "ThrottleTimeMs", "type": "int32", "versions": "1+", "ignorable": true,
```

FetchSnapshot

Handling

No changes are needed to the handling of this RPC. The only thing to keep in mind is that replicas will accept FetchSnapshot request as long as they are the latest leader even if they are not in the current voters set.

Request

Add the replica UUID (directory id) to the Fetch request. This is not needed for correctness because leaders only track fetched offsets using the Fetch RPC but it is good add for consistency and debugability.

```
git diff upstream/trunk clients/src/main/resources/common/message/FetchSnapshotRequest.json
diff --git a/clients/src/main/resources/common/message/FetchSnapshotRequest.json b/clients/src/main/resources
/common/message/FetchSnapshotRequest.json
index 358ef2e322..9a577d6289 100644
--- a/clients/src/main/resources/common/message/FetchSnapshotRequest.json
+++ b/clients/src/main/resources/common/message/FetchSnapshotRequest.json
@@ -18,11 +18,11 @@
   "type": "request",
   "listeners": ["controller"],
   "name": "FetchSnapshotRequest",
-  "validVersions": "0",
+  "validVersions": "0-1",
   "flexibleVersions": "0+",
   "fields": [
     { "name": "ClusterId", "type": "string", "versions": "0+", "nullableVersions": "0+", "default": "null",
"taggedVersions": "0+", "tag": 0,
       "about": "The cluster ID if known" },
     { "name": "ReplicaId", "type": "int32", "versions": "0+", "default": "-1", "entityType": "brokerId",
       "about": "The broker ID of the follower" },
     { "name": "MaxBytes", "type": "int32", "versions": "0+", "default": "0x7fffffff",
@@ -44,7 +44,9 @@
           { "name": "Epoch", "type": "int32", "versions": "0+" }
         ]},
         { "name": "Position", "type": "int64", "versions": "0+",
-           "about": "The byte position within the snapshot to start fetching from" }
+           "about": "The byte position within the snapshot to start fetching from" },
+         { "name": "ReplicaUuid", "type": "uuid", "versions": "1+", "taggedVersions": "1+", "tag": 0,
+           "about": "The directory id of the follower fetching" }
        ]}
      ]}
    ]
```

Response

Version 1 renames LeaderIdAndEpoch to CurrentLeader and adds Endpoint to CurrentLeader.

```
git diff upstream/trunk clients/src/main/resources/common/message/FetchSnapshotResponse.json
diff --git a/clients/src/main/resources/common/message/FetchSnapshotResponse.json b/clients/src/main/resources
/common/message/FetchSnapshotResponse.json
index 887a5e4401..2d9d269930 100644
--- a/clients/src/main/resources/common/message/FetchSnapshotResponse.json
+++ b/clients/src/main/resources/common/message/FetchSnapshotResponse.json
@@ -17,23 +17,23 @@
   "apiKey": 59,
   "type": "response",
   "name": "FetchSnapshotResponse",
-  "validVersions": "0",
+  "validVersions": "0-1",
   "flexibleVersions": "0+",
   "fields": [
     { "name": "ThrottleTimeMs", "type": "int32", "versions": "0+", "ignorable": true,
       "about": "The duration in milliseconds for which the request was throttled due to a quota violation, or
zero if the request did not violate any quota" },
     { "name": "ErrorCode", "type": "int16", "versions": "0+", "ignorable": false,
       "about": "The top level response error code" },
     { "name": "Topics", "type": "[]TopicSnapshot", "versions": "0+",
       "about": "The topics to fetch", "fields": [
       { "name": "Name", "type": "string", "versions": "0+", "entityType": "topicName",
         "about": "The name of the topic to fetch" },
       { "name": "Partitions", "type": "[]PartitionSnapshot", "versions": "0+",
         "about": "The partitions to fetch", "fields": [
         { "name": "Index", "type": "int32", "versions": "0+",
           "about": "The partition index" },
         { "name": "ErrorCode", "type": "int16", "versions": "0+",
           "about": "The error code, or 0 if there was no fetch error" },
         { "name": "SnapshotId", "type": "SnapshotId", "versions": "0+",
           "about": "The snapshot endOffset and epoch fetched",
           "fields": [
@@ -43,17 +43,24 @@
         { "name": "CurrentLeader", "type": "LeaderIdAndEpoch",
           "versions": "0+", "taggedVersions": "0+", "tag": 0, "fields": [
           { "name": "LeaderId", "type": "int32", "versions": "0+", "entityType": "brokerId",
             "about": "The ID of the current leader or -1 if the leader is unknown" },
           { "name": "LeaderEpoch", "type": "int32", "versions": "0+",
             "about": "The latest known leader epoch" }
         ]},
         { "name": "Size", "type": "int64", "versions": "0+",
           "about": "The total size of the snapshot" },
         { "name": "Position", "type": "int64", "versions": "0+",
           "about": "The starting byte position within the snapshot included in the Bytes field" },
         { "name": "UnalignedRecords", "type": "records", "versions": "0+",
           "about": "Snapshot data in records format which may not be aligned on an offset boundary" }
       ]}
+    ]},
+    { "name": "NodeEndpoints", "type": "[]NodeEndpoint", "versions": "1+", "taggedVersions": "1+", "tag": 0,
+      "about": "Endpoints for all current-leaders enumerated in PartitionSnapshot", "fields": [
+      { "name": "NodeId", "type": "int32", "versions": "1+",
+        "mapKey": true, "entityType": "brokerId", "about": "The ID of the associated node" },
+      { "name": "Host", "type": "string", "versions": "1+", "about": "The node's hostname" },
+      { "name": "Port", "type": "int32", "versions": "1+", "about": "The node's port" }
     ]}
   ]
 }
```

DescribeQuorum

Handling

The DescribeQuorum RPC will get forwarded to the KRaft cluster metadata leader. The response for this RPC will be extended to include the UUID of all of the voters and observers, and the endpoint information for all of the voters.

Since the node and listener information is index by the node id, if there are multiple voters with the same replica id only the latest entry will be returned and used.

Request

```
git diff upstream/trunk clients/src/main/resources/common/message/DescribeQuorumRequest.json
diff --git a/clients/src/main/resources/common/message/DescribeQuorumRequest.json b/clients/src/main/resources
/common/message/DescribeQuorumRequest.json
index cee8fe6982..93ab303aaf 100644
--- a/clients/src/main/resources/common/message/DescribeQuorumRequest.json
+++ b/clients/src/main/resources/common/message/DescribeQuorumRequest.json
@@ -19,7 +19,7 @@
   "listeners": ["broker", "controller"],
   "name": "DescribeQuorumRequest",
   // Version 1 adds additional fields in the response. The request is unchanged (KIP-836).
-  "validVersions": "0-1",
+  "validVersions": "0-2",
   "flexibleVersions": "0+",
   "fields": [
     { "name": "Topics", "type": "[]TopicData",
```

Response

```
git diff upstream/trunk clients/src/main/resources/common/message/DescribeQuorumResponse.json
diff --git a/clients/src/main/resources/common/message/DescribeQuorumResponse.json b/clients/src/main/resources
/common/message/DescribeQuorumResponse.json
index 0ea6271238..b54cd6bd50 100644
--- a/clients/src/main/resources/common/message/DescribeQuorumResponse.json
+++ b/clients/src/main/resources/common/message/DescribeQuorumResponse.json
@@ -18,11 +18,13 @@
   "type": "response",
   "name": "DescribeQuorumResponse",
   // Version 1 adds LastFetchTimeStamp and LastCaughtUpTimestamp in ReplicaState (KIP-836).
-  "validVersions": "0-1",
+  "validVersions": "0-2",
   "flexibleVersions": "0+",
   "fields": [
     { "name": "ErrorCode", "type": "int16", "versions": "0+",
       "about": "The top level error code."},
+    { "name": "ErrorMessage", "type": "string", "versions": "2+", "nullableVersions": "2+", "ignorable": true,
+      "about": "The error message, or null if there was no error." },
     { "name": "Topics", "type": "[]TopicData",
       "versions": "0+", "fields": [
       { "name": "TopicName", "type": "string", "versions": "0+", "entityType": "topicName",
@@ -32,6 +34,8 @@
         { "name": "PartitionIndex", "type": "int32", "versions": "0+",
           "about": "The partition index." },
         { "name": "ErrorCode", "type": "int16", "versions": "0+"},
+        { "name": "ErrorMessage", "type": "string", "versions": "2+", "nullableVersions": "2+", "ignorable":
true,
+          "about": "The error message, or null if there was no error." },
         { "name": "LeaderId", "type": "int32", "versions": "0+", "entityType": "brokerId",
           "about": "The ID of the current leader or -1 if the leader is unknown."},
         { "name": "LeaderEpoch", "type": "int32", "versions": "0+",
@@ -40,10 +44,25 @@
         { "name": "CurrentVoters", "type": "[]ReplicaState", "versions": "0+" },
         { "name": "Observers", "type": "[]ReplicaState", "versions": "0+" }
       ]}
-    ]}],
+    ]},
+    { "name": "Nodes", "type": "[]Node", "versions": "2+", "fields": [
+      { "name": "NodeId", "type": "int32", "versions": "2+",
+        "mapKey": true, "entityType": "brokerId", "about": "The ID of the associated node" },
+      { "name": "Listeners", "type": "[]Listener",
+        "about": "The listeners of this controller", "versions": "0+", "fields": [
+        { "name": "Name", "type": "string", "versions": "0+", "mapKey": true,
+          "about": "The name of the endpoint" },
+        { "name": "Host", "type": "string", "versions": "0+",
+          "about": "The hostname" },
+        { "name": "Port", "type": "uint16", "versions": "0+",
+          "about": "The port" }
+      ]}
+    ]}
+  ],
   "commonStructs": [
     { "name": "ReplicaState", "versions": "0+", "fields": [
       { "name": "ReplicaId", "type": "int32", "versions": "0+", "entityType": "brokerId" },
+      { "name": "ReplicaUuid", "type": "uuid", "versions": "2+" },
       { "name": "LogEndOffset", "type": "int64", "versions": "0+",
         "about": "The last known log end offset of the follower or -1 if it is unknown"},
       { "name": "LastFetchTimestamp", "type": "int64", "versions": "1+", "ignorable": true, "default": -1,
```

Admin client

The Java Admin client will be extended to support the new field in the DescribeQuorum response and the new AddVoter and RemoveVoter RPCs. When the broker bootstrap servers is used the admin client will send the request to the least loaded broker. When the controller bootstrap servers is used the admin client will send the request to the KRaft leader (active controller).

Monitoring

NAME	TAGS	TYPE	NOTE
number-of-voters	type=raft-metrics	gauge	number of voters for the cluster metadata topic partition.
number-of-observers	type=raft-metrics	guage	number of observer that could be promoted to voters.
pending-add-voter	type=raft-metrics	guage	1 if there is a pending add voter operation, 0 otherwise.
pending-remove-voter	type=raft-metrics	guage	1 if there is a pending remove voter operation, 0 otherwise.
TBD	TBD	guage	1 if a controller node is not a voter for the KRaft cluster metadata partition, 0 otherwise.
duplicate-voter-ids	type=raft-metrics	gauge	Counts the number of duplicate replica id in the set of voters.
number-of-offline-voters	type=raft-metrics	gauge	Number of voters with a last Fetch timestamp greater than the Fetch timeout.
ignored-static-voters	TBD	gauge	1 if controller.quorum.voter is set and the kraft.version is greater than 0, 0 otherwise.

Command line interface

kafka-metadata-shell

A future KIP will describe how the kafka-metadata-shell tool will be extended to be able to read and display KRaft control records from the quorum, snapshot and log.

kafka-storage

The format command will get extended as follow.

--standalone

This command will 1) create a meta.properties file in metadata.log.dir with a randomly generated directory.id, 2) create a snapshot at `00000000000000000000-0000000000.checkpoint` with the necessary control records (`KRaftVersionRecord` and `VotersRecord`) to make this Kafka node the only voter for the quorum.

This option is unsafe because it doesn’t use the quorum to establish the new quorum. This only should only be executed once on one of the voters. The rest of the voters should join the quorum by using the `kafka-metadata-quorum add-controller` command.

--controller-quorum-voters

The value to this option will have the follow schema <replica-id>[-<replica-uuid>]@<host>:<port>. Logically, this function is very similar to the controller. quorum.voters. The important difference is that it will optionally support the user directly specifying the replica's directory id.

When the format command is executed with this option it will read the node.id configured in the properties file specified by the --config option and compare it against the <replica-id> specified in --controller.quorum.voters. If there is a match, it will write the <replica-uuid> specified to the directory.id property in the meta.properties for the metadata.log.dir directory.

Similar to --standalone this command will create a snapshot at `00000000000000000000-0000000000.checkpoint` with the necessary control records (`KRaftVersionRecord` and `VotersRecord`).

This option is unsafe. It is important that the operator uses the same value across all of the voters specified and only executes this command once per voter.

kafka-features

The upgrade and downgrade command will support a new configuration flag. A downgrade that results in the decrease of the `kraft.version` will be rejected by the KRaft leader.

--release-software

The value specified in this flag will be used to find the corresponding `metadata.version` and `kraft.version`. The `--metadata` version flag will get deprecated and will be a synonym for `--release-software`.

kafka-metadata-quorum

This tool as described in KIP-595 and KIP-836 will be improved to support these additional commands and options:

describe --status

This command be extended to print the new information added to the DescribeQuorum RPC. The includes the directory id for all of the replicas (voters and observers). The known endpoints for all of the voters. Any uncommitted voter changes.

```
kafka-metadata-quorum describe --status
ClusterId:                SomeClusterId
LeaderId:                 0
LeaderEpoch:              15
HighWatermark:            234130
MaxFollowerLag:           34
MaxFollowerLagTimeMs:     15
CurrentVoters:            [{"id": 0, "uuid": "UUID1", "endpoints": ["host:port"]}, {"id": 1, "uuid": "UUID2",
"endpoints": ["host:port"]}, {"id": 2, "uuid": "UUID2", "endpoints": ["host:port"]}]
Observers:                [{"id": 3, "uuid": "UUID3"}]
UncommittedAddedVoter:    {"id": 2, "uuid": "UUID2", "endpoints": ["host:port"]}
UncommittedRemovedVoter:  {"id": 2, "uuid": "UUID2", "endpoints": ["host:port"]}
```

describe --replication

This command will print on additional column for the replica uuid after the replica id. E.g.

```
kafka-metadata-quorum --describe replication
ReplicaId   ReplicaUuid   LogEndOffset   ...
0           uuid1         234134         ...
...
```

add-controller --config <server.properties>

This command is use to add controllers to the KRaft cluster metadata partition. This command must be executed using the server configuration of the new controller. The command will read the server properties file to read the replica id, the endpoints, and the meta.properties for the directory id. The tool will set the `TimoutMs` for the `AddVoterRequest` to 30 seconds.

remove-controller --controller-id <controller-id> --controller-uuid <controller-uuid>

This command is used to remove voters from the KRaft cluster metadata partition. The flags --controller-id and --controller-uuid must be specified.

Compatibility, deprecation, and migration plan

RPC versions will be negotiated using the `ApiVersions` RPC. KRaft will use the `kraft.version` to determine which version of `KRaftVersionRecord` and `VotersRecord` to write to the log and which version of `QuorumStateData` to write to the `quorum-state` file.

Downgrading the `kraft.version` from 1 to 0 is not possible. This is mainly due to the fact that `kraft.version` 1 writes data to the log and snapshot that gets replicated to all of the replicas.

When to remove controller.quorum.voters

It is safe for the operator to remove the configuration for `controller.quorum.voters` when the `kraft.version` has been upgrade to version 1. All of the Kafka nodes will expose the `ignored-static-voter` metrics. If all of the Kafka nodes expose a 1 for this metrics, it is safe to remove `controller.quorum.voters` from the node configuration.

Test plan

This KIP will be tested using unittest, integration tests, system test, simulation tests and TLA+ specification.

Rejected alternatives

KIP-642: Dynamic quorum reassignment

KIP-642: Dynamic quorum reassignment. KIP-642 describe how to perform dynamic reassignment will multiple voters added and removed. KIP-642 doesn't support disk failures and it would be more difficult to implement compared to this KIP-853. In a future KIP we can describe how we can add administrative operations that support the addition and removal of multiple voters.

Leverage metadata controller mechanisms

There are some common problems between the Kafka Controller and KRaft like bootstrapping cluster information, versioning and propagating node information. The Kafka controller has solved both of these problems in the metadata layer. The metadata layer is an application built using the KRaft layer. KRaft and KIP-853 should not dependent on the metadata layer for its solution to these problems. One advantage is to allow other use cases for KRaft that are not the cluster metadata partition but the main advantage is achieving a clear separation for both reliability, testability and debuggability.

In the future we can extend Kafka so that the Kafka controller relies on KRaft's solution to these problems.

References

1. Ongaro, Diego, and John Ousterhout. "In search of an understandable consensus algorithm." *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 2014.
2. Ongaro, Diego. *Consensus: Bridging theory and practice*. Diss. Stanford University, 2014.
3. Bug in single-server membership changes
4. KIP-595: A Raft Protocol for the Metadata Quorum
5. KIP-630: Kafka Raft Snapshot
6. KIP-631: The Quorum-based Kafka Controller