KIP-595: A Raft Protocol for the Metadata Quorum

Parent KIP

KIP-500: Replace ZooKeeper with a Self-Managed Metadata Quorum (Accepted)

Status

Current state: Accepted

Discussion thread: here

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

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

Note this is a joint work proposed by Boyang Chen, guozhang Wang, and Jason Gustafson

Motivation

This proposal follows KIP-500, which lays out an architecture for Kafka based on a replicated metadata log which is maintained by a self-managed quorum. Log replication is at the heart of Kafka, and it is also at the heart of consensus protocols such as Raft. The replication protocol that we spent years improving and validating is actually not too different from Raft if you take a step back. There is a single leader at any time which is enforced by a monotonically increasing epoch; messages are unique by offset and epoch; and there is a protocol to reconcile log inconsistencies following leader changes using offset and epoch information from the log. The gaps between Kafka's replication protocol and Raft are the following:

1. Quorum commit semantics: messages in Kafka are considered committed after replication to the current ISR, which is maintained in Zookeeper.
2. Leader election: in Kafka, leader elections are done by the controller which writes the updated state to Zookeeper.

Essentially we have relied on Zookeeper's consensus protocol up now to enforce consistent replication semantics. With KIP-500, we are taking a step into the wild. This proposal is about bridging the gaps in the Kafka replication protocol in order to support the metadata quorum.

The protocol we are proposing is a sort of Raft dialect which is heavily influenced by Kafka's log replication protocol. For example, it is pull-based unlike Raft which is push-based. This adds some complication, but mostly it's just looking at Raft through a mirror. We also have favored Kafka terminology (offset /epoch) over traditional Raft terminology (index/term). Think of the protocol as beginning with Kafka log replication and adding Raft leader election.

Note that this protocol assumes something like the Kafka v2 log message format. It is not compatible with older formats because records do not include the leader epoch information that is needed for log reconciliation. We intentionally make no assumption about the physical log format and minimal assumptions about it's logical structure. This makes it usable both for internal metadata replication and (eventually) partition data replication.

A note on scope: This proposal is only concerned with the semantics and the management of the Metadata Quorum identified in the architecture in KIP-500. This includes the following:

- Specification of the replication protocol and its semantics
- Specification of the log structure and message schemas that will be used to maintain quorum state
- Tooling/metrics to view quorum health

The following aspects are specified in follow-up proposals:

- Log compaction: see KIP-630
- Controller use of the quorum and message specification: see KIP-631
- Quorum reassignment: see KIP-642

At a high level, this proposal treats the metadata log as a topic with a single partition. The benefit of this is that many of the existing APIs can be reused. For example, it allows users to use a consumer to read the contents of the metadata log for debugging purposes. We are also trying to pave the way for normal partition replication through Raft as well as eventually supporting metadata sharding with multiple partitions. In our prototype implementation, we have assumed the name `__cluster_metadata` for this topic, but this is not a formal part of this proposal.

Key Concepts

If you are familiar with Kafka replication, most of the same concepts apply to Raft replication; however we replace ISR commit semantics with quorum commit semantics and we add leader election.

Leader Epoch: Known as a "term" in Raft, this is a monotonically increasing sequence which is incremented during every leader election. This is used both to fence zombies and to reconcile logs which have diverged. Log records are uniquely identified by the offset in the log and the epoch of the leader that appended them.

High watermark: Although there is no notion of an ISR for a quorum-based replication protocol, we still have a *high watermark* to control when a record is considered "committed." This is the largest offset which is replicated to a majority of the voters. The protocol is designed to guarantee that committed records are not lost.

Voter: A voter is a replica which is eligible to cast votes during an election and potentially become a leader. Voters would also fetch from the current leader to replicate records.

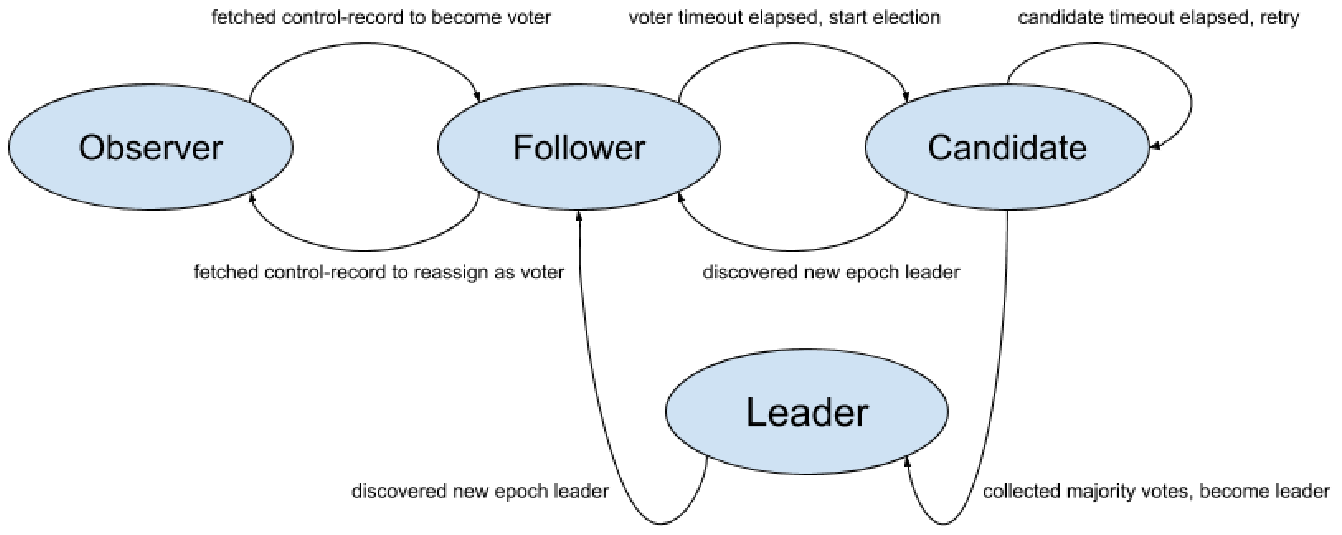
Candidate: When a voter decides to elect for the new leader, it will start an election by bumping the leader epoch and casting a vote for itself. We refer to these as *candidates.*

Leader: After a candidate gathers a majority of votes from its peers (including itself), it will become the *leader* for the current epoch. For each epoch there will be only one leader which takes client requests for new records. The single leader and all other voters form the quorum.

Follower: A voter which has either cast its vote for a current candidate or which is fetching from the current leader is known as a follower. Followers can become candidates after a configurable timeout if they have not heard from the current leader or if an active election does not conclude.

Observer: An observer is a replica which is not eligible to vote and cannot become a leader. But like other voters it can still fetch from the leader to replicate records. In other words an observer is only responsible for discovering the leader and replicating the log.

State Machine


fetched control-record to become voter
voter timeout elapsed, start election
candidate timeout elapsed, retry
Observer
Follower
Candidate
fetched control-record to reassign as voter
discovered new epoch leader
Leader
discovered new epoch leader
collected majority votes, become leader

Configurations

- `quorum.voters`: This is a connection map which contains the IDs of the voters and their respective endpoint. We use the following format for each voter in the list `{broker-id}@{broker-host):{broker-port}`. For example, ``quorum.voters=1@kafka-1:9092, 2@kafka-2:9092, 3@kafka-3:9092``.
- `quorum.fetch.timeout.ms:` Maximum time without a successful fetch from the current leader before a new election is started.
- `quorum.election.timeout.ms:` Maximum time without collected a majority of votes during the candidate state before a new election is retried.
- `quorum.election.backoff.max.ms:` Maximum exponential backoff time (based on the number if retries) after an election timeout, before a new election is triggered.
- `quorum.request.timeout.ms:` Maximum time before a pending request is considered failed and the connection is dropped.
- `quorum.retry.backoff.ms:` Initial delay between request retries. This config and the one below is used for retriable request errors or lost connectivity and are different from the election.backoff configs above.
- `quorum.retry.backoff.max.ms`: Max delay between requests. Backoff will increase exponentially beginning from `quorum.retry.backoff.ms` (the same as in KIP-580).
- `broker.id`: The existing broker id config shall be used as the voter id in the Raft quorum.

Persistent State

This proposal requires a persistent log as well as a separate file to maintain the current quorum state. The latter is needed both to support dynamic reassignment of the voters and for the persistence of vote state. Below we define the structure of this state.

Log Structure

We assume a normal log structure for the metadata log (which we may also refer to as the "metadata topic"). The protocol we define in this proposal is agnostic to the record format, but we assume the following fields at a minimum for individual records:

```
Record => Offset LeaderEpoch ControlType Key Value Timestamp
```

Records are uniquely defined by their offset in the log and the epoch of the leader that appended the record. The key and value schemas will be defined by the controller in a separate KIP; here we treat them as arbitrary byte arrays. However, we do require the ability to append "control records" to the log which are reserved only for use within the Raft quorum (e.g. this enables quorum reassignment).

The v2 message format has all the assumed/required properties already.

Quorum State

We use a separate file to store the current state of the quorum. This is both for convenience and correctness. It helps us to initialize the quorum state after a restart, but we also need it in order to know which broker we have voted for in a given election. The Raft protocol does not allow voters to change their votes, so we have to preserve this state across restarts. Below is the schema for this `quorum-state` file.

```
{
  "type": "data",
  "name": "QuorumStateMessage",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
          {"name": "LeaderId", "type": "int32", "versions": "0+"},
      {"name": "LeaderEpoch", "type": "int32", "versions": "0+"},
      {"name": "VotedId", "type": "int32", "versions": "0+"},
      {"name": "AppliedOffset", "type": "int64", "versions": "0+"},
      {"name": "CurrentVoters", "type": "[]Voter", "versions": "0+", "fields": [
        {"name": "VoterId", "type": "int32", "versions": "0+"}
      ]}
  ]
}
```

Below we define the purpose of these fields:

- `LeaderId`: this is the last known leader of the quorum. A value of -1 indicates that there is no leader.
- `LeaderEpoch`: this is the last known leader epoch. This is initialized to 0 when the quorum is bootstrapped and should never be negative.
- `VotedId`: indicates the id of the broker that this replica voted for in the current epoch. A value of -1 indicates that the replica has not (or cannot) vote.
- `AppliedOffset`: Reflects the maximum offset that has been applied to this quorum state. This is used for log recovery. The broker must scan from this point on initialization to detect updates to this file.
- `CurrentVoters`: the latest known voters for this quorum

The use of this file will be described in more detail below as we describe the protocol.

Note one key difference of this internal topic compared with other topics is that we should **always enforce fsync upon appending to local log** to guarantee Raft algorithm correctness, i.e. we are dropping group flushing for this topic. In practice, we could still optimize the fsync latency in the following way: 1) the client requests to the leader are expected to have multiple entries, 2) the fetch response from the leader can contain multiple entries, and 3) the leader can actually defer fsync until it knows "majority - 1" has get to a certain entry offset. We will leave this potential optimization as a future work out of the scope of this KIP design.

Other log-related metadata such as log start offset, recovery point, HWM, etc are still stored in the existing checkpoint file just like any other Kafka topic partitions.

Additionally, we make use of the current `meta.properties` file, which caches the value from `broker.id` in the configuration as well as the discovered clusterId. As is the case today, the configured `broker.id` does not match the cached value, then the broker will refuse to start. If we connect to a cluster with a different clusterId, then the broker will receive a fatal error and shutdown.

Leader Election and Data Replication

The key functionalities of any consensus protocol are leader election and data replication. The protocol for these two functionalities consists of four core RPCs:

- **Vote**: Sent by a voter to initiate an election.
- **BeginQuorumEpoch**: Used by a new leader to inform the voters of its status.
- **EndQuorumEpoch**: Used by a leader to gracefully step down and allow a new election.
- **Fetch**: Sent by voters and observers to the leader in order to replicate the log.

We are also adding one new API to view the health/status of quorum replication:

- **DescribeQuorum**: Administrative API to list the replication state (i.e. lag) of the voters. More of the details can also be found in KIP-642: Dynamic quorum reassignment#DescribeQuorum.

Before getting into the details of these APIs, there are a few common attributes worth mentioning upfront:

1. All requests save `Vote` have a field for the leader epoch. Voters and leaders are always expected to ensure that the request epoch is consistent with its own and return an error if it is not.
2. We piggyback current leader and epoch information on all responses. This reduces latency to discover leader changes.
3. Although this KIP only expects a single-raft quorum implementation initially, we see it is beneficial to keep the door open for multi-raft architecture in the long term. This leaves the door open for use cases such as sharded controller or general quorum based topic replication.

As mentioned above, this protocol is only concerned with leader election and log replication. It does not specify what log entries are appended to the leader's log nor how they will be received by the leader. Typically this would be through specific management APIs. For example, KIP-497 adds an AlterISR API. When the leader of the metadata quorum (i.e. the controller) receives an AlterISR request, it will append an entry to its log. Additionally, we have taken log compaction out of the scope because this is tied to the semantics of the metadata log. See KIP-630 for more details on compaction.

Common Error Codes: these APIs rely on a common set of error codes which are defined here.

- INVALID_CLUSTER_ID: The request indicates a clusterId which does not match the value cached in `meta.properties`.
- FENCED_LEADER_EPOCH: The leader epoch in the request is smaller than the latest known to the recipient of the request.
- UNKNOWN_LEADER_EPOCH: The leader epoch in the request is larger than expected. Note that this is an unexpected error. Unlike normal Kafka log replication, it cannot happen that the follower receives the newer epoch before the leader.
- OFFSET_OUT_OF_RANGE: Used in the `Fetch` API to indicate that the follower has fetched from an invalid offset and should truncate to the offset/epoch indicated in the response.
- NOT_LEADER_FOR_PARTITION: Used in `DescribeQuorum` and `AlterPartitionReassignments` to indicate that the recipient of the request is not the current leader.
- INVALID_QUORUM_STATE: This error code is reserved for cases when a request conflicts with the local known state. For example, if two separate nodes try to become leader in the same epoch, then it indicates an illegal state change.
- INCONSISTENT_VOTER_SET: Used when the request contains inconsistent membership.

Below we describe the schema and behavior of each of these APIs.

Vote

The Vote API is used by voters to hold an election. As mentioned above, the main difference from Raft is that this protocol is pull-based. Voters send fetch requests to the leaders in order to replicate from the log. These fetches also serve as a liveness check for the leader. If a voter perceives a leader as down, it will hold a new election and declare itself a candidate. A voter will begin a new election under three conditions:

1. If it fails to receive a `FetchResponse` from the current leader before expiration of `quorum.fetch.timeout.ms`
2. If it receives a `EndQuorumEpoch` request from the current leader
3. If it fails to receive a majority of votes before expiration of `quorum.election.timeout.ms` after declaring itself a candidate.

A voter triggers an election by first voting for itself and updating the `quorum-state` file. Once this state is persistent, the voter becomes a candidate and sends a `VoteRequest` to all the other voters. Note that voters, including candidates, are not allowed to change their vote once cast in a given epoch.

New elections are always delayed by a random time which is bounded by `quorum.election.backoff.max.ms`. This is part of the Raft protocol and is meant to prevent gridlocked elections. For example, with a quorum size of three, if only two voters are online, then we have to prevent repeated elections in which each voter declares itself a candidate and refuses to vote for the other.

Leader Progress Timeout

In the traditional push-based model, when a leader is disconnected from the quorum due to network partition, it will start a new election to learn the active quorum or form a new one immediately. In the pull-based model, however, say a new leader has been elected with a new epoch and everyone has learned about it except the old leader (e.g. that leader was not in the voters anymore and hence not receiving the BeginQuorumEpoch as well), then that old leader would not be notified by anyone about the new leader / epoch and become a pure "zombie leader", as there is no regular heartbeats being pushed from leader to the follower. This could lead to stale information being served to the observers and clients inside the cluster.

To resolve this issue, we will piggy-back on the "`quorum.fetch.timeout.ms`" config, such that if the leader did not receive Fetch requests from a majority of the quorum for that amount of time, it would begin a new election and start sending `VoteRequest` to voter nodes in the cluster to understand the latest quorum. If it couldn't connect to any known voter, the old leader shall keep starting new elections and bump the epoch. And if the returned response includes a newer epoch leader, this zombie leader would step down and becomes a follower. Note that the node will remain a candidate until it finds that it has been supplanted by another voter, or win the election eventually.

As we know from the Raft literature, this approach could generate disruptive voters when network partitions happen on the leader. The partitioned leader will keep increasing its epoch, and when it eventually reconnects to the quorum, it could win the election with a very large epoch number, thus reducing the quorum availability due to extra restoration time. Considering this scenario is rare, we would like to address it in a follow-up KIP.

Request Schema

```
{
  "apiKey": N,
  "type": "request",
  "name": "VoteRequest",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "ClusterId", "type": "string", "versions": "0+", "nullableVersions": "0+", "default": "null"},
    { "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": "CandidateEpoch", "type": "int32", "versions": "0+",
          "about": "The bumped epoch of the candidate sending the request"},
        { "name": "CandidateId", "type": "int32", "versions": "0+",
          "about": "The ID of the voter sending the request"},
        { "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"}
      ]
      }
    ]
    }
  ]
}
```

As noted above, the request follows the convention of other batched partition APIs, such as `Fetch` and `Produce`. Initially we expect that only a single topic partition will be included, but this leaves the door open for "multi-raft" support.

Response Schema

```
{
  "apiKey": N,
  "type": "response",
  "name": "VoteResponse",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "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+",
          "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": "VoteGranted", "type": "bool", "versions": "0+",
          "about": "True if the vote was granted and false otherwise"}
      ]
      }
    ]
    }
  ]
}
```

Vote Request Handling

Note we need to respect the **extra condition on leadership:** the candidate’s log is at least as up-to-date as any other log in the majority who vote for it. Raft determines which of two logs is more "up-to-date" by comparing the offset and epoch of the last entries in the logs. If the logs' last entry have different epochs, then the log with the later epoch is more up-to-date. If the logs end with the same epoch, then whichever log is longer is more up-to-date. The Vote request includes information about the candidate’s log, and the voter denies its vote if its own log is more up-to-date than that of the candidate.

When a voter decides to become a candidate and ask for others to vote for it, it will increment its current leader epoch as `CandidateEpoch`.

When a voter handles a Vote request:

1. Check whether the clusterId in the request matches the cached local value (if present). If not, then reject the vote and return INVALID_CLUSTER_ID.
2. First it checks whether an epoch larger than the candidate epoch from the request is known. If so, the vote is rejected.
3. Next it checks if it has voted for that candidate epoch already. If it has, then only grant the vote if the candidate id matches the id that was already voted. Otherwise, the vote is rejected.
4. If the candidate epoch is larger than the currently known epoch:
    a. Check whether `CandidateId` is one of the expected voters. If not, then reject the vote. The candidate in this case may have been part of an incomplete voter reassignment, so the voter should accept the epoch bump and itself begin a new election.
    b. Check that the candidate's log is at least as up-to-date as it (see above for the comparison rules). If yes, then grant that vote by first updating the quorum-state file, and then returning the response with `voteGranted` to yes; otherwise rejects that request with the response.

Also note that a candidate always votes for itself at the current candidate epoch. That means, it will also need to update the `quorum-state` file as "voting for myself" before sending out the vote requests. On the other hand, if it receives a `Vote` request with a larger candidate epoch, it can still grants that vote while at the same time transiting back to voter state because a newer leader may has been elected for a newer epoch.

Vote Response Handling

When receiving a `Vote` response:

1. First we need to check if the voter is still a candidate because it may already observed an election with a larger epoch; if it is no longer in "candidate" state, just ignore the response.
2. Otherwise, check if it has accumulated majority of votes for this epoch – this information does not need to be persisted, since upon failover it can just resend the vote request and the voters would just grant that request again as long as there's no newer epoch candidate – and if yes, it can transit to the leader state by updating the quorum-state file indicating itself as the leader for the new epoch. Otherwise, just record this vote in memory.
3. If a candidate has received a majority of votes, it would also start by writing the current assignment state to its log so that its `LastEpoch` and `LastEpochOffset` are updated, and then after that it can start sending out BeginQuorumEpoch requests.

Writing the dummy entry with the new epoch is not necessary for correctness, but as an optimization to reduce the client request latency. Basically it allows for faster advancement of the high watermark following a leader election (the need for this is discussed in more detail below in the handling of the `Fetch` API). The dummy record will be a control record with `Type=3`. Below we define the schema:

```
{
  "type": "data",
  "name": "LeaderChangeMessage",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
          {"name": "LeaderId", "type": "int32", "versions": "0+",
       "about": "The ID of the newly elected leader"},
      {"name": "VotedIds", "type": "[]int32", "versions": "0+",
       "about": "The IDs of the voters who voted for the current leader"},

  ]
}
```

Also note that unlike other Kafka topic partition data whose log appends are persisted asynchronously, **for this special quorum topic all log appends must be synced to FS before returning**.

Note on Gridlocked Elections: It is possible for elections to fail. For example, if each voter becomes a candidate at the same time and votes for itself. Generally if a voter fails to get a majority of votes before `quorum.election.timeout.ms`, then the vote is deemed to have failed, which will cause to step down and backoff according to `quorum.election.backoff.max.ms` before retrying. Under some situations, a candidate can immediately detect when a vote has failed. For example, if there are only two voters and a candidate fails to get a vote from the other voter (i.e. `VoteGranted` is returned as false in the `VoteResponse`), then there is no need to wait for the election timeout. The candidate in this case can immediately step down and backoff.

BeginQuorumEpoch

In traditional Raft, the leader will send an empty Append request to the other nodes in the quorum in order to assert its term. As a pull-based protocol, we need a separate `BeginQuorumEpoch` API to do the same in order to ensure election results are discovered quickly. In this protocol, once a leader has received enough votes, it will send the `BeginQuorumEpoch` request to all voters in the quorum.

Note that only voters receive the `BeginQuorumEpoch` request: observers will discover the new leader through the `Fetch` API. If a non-leader receives a `Fetch` request, it would return an error code in the response indicating that it is no longer the leader and it will also encode the current known leader id / epoch as well, then the observers can start fetching from the new leader. Upon initialization, an observer will send a `Fetch` request to any of the voters at random until the new leader is found.

Request Schema

```
{
  "apiKey": N,
  "type": "request",
  "name": "BeginQuorumEpochRequest",
  "validVersions": "0",
  "fields": [
    { "name": "ClusterId", "type": "string", "versions": "0+", "nullableVersions": "0+", "default": "null"},
    { "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": "LeaderId", "type": "int32", "versions": "0+",
          "about": "The ID of the newly elected leader"},
        { "name": "LeaderEpoch", "type": "int32", "versions": "0+",
          "about": "The epoch of the newly elected leader"}
      ]}
    ]}
  ]
}
```

Response Schema

```
{
  "apiKey": N,
  "type": "response",
  "name": "BeginQuorumEpochResponse",
  "validVersions": "0",
  "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+",
          "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"}
      ]}
    ]}
  ]
}
```

BeginQuorumEpoch Request Handling

A voter will accept a `BeginQuorumEpoch` if its leader epoch is greater than or equal to the current known epoch so long as it doesn't conflict with previous knowledge. For example, if the leaderId for epoch 5 was known to be A, then a voter will reject a `BeginQuorumEpoch` request from a separate voter B from the same epoch. If the epoch is less than the known epoch, the request is rejected.

As described in the section on Cluster Bootstrapping, a broker will fail if it receives a `BeginQuorumEpoch` with a clusterId which is inconsistent from the cached value.

As soon as a broker accepts a `BeginQuorumEpoch` request, it will transition to a *follower* state and begin sending `Fetch` requests to the new leader.

BeginQuorumEpoch Response Handling

If the response contains no errors, then the leader will record the follower in memory as having endorsed the election. The leader will continue sending `BeginQuorumEpoch` to each known voter until it has received its endorsement. This ensures that a voter that is partitioned from the network will be able to discover the leader quickly after the partition is restored. An endorsement from an existing voter may also be inferred through a received `Fetch` request with the new leader's epoch even if the `BeginQuorumEpoch` request was never received.

If the error code indicates that voter's known leader epoch is larger (i.e. if the error is FENCED_LEADER_EPOCH), then the voter will update `quorum-state` and become a follower of that leader and begin sending `Fetch` requests.

EndQuorumEpoch

The EndQuorumEpoch API is used by a leader to gracefully step down so that an election can be held immediately without waiting for the election timeout. The primary use case for this is to enable graceful shutdown. If the shutting down voter is either an active current leader or a candidate if there is an election in progress, then this request will be sent. It is also used when the leader needs to be removed from the quorum following an `AlterPartitionReassignments` request.

The EndQuorumEpochRequest will be sent to all voters in the quorum. Inside each request, leader will define the list of preferred successors sorted by each voter's current replicated offset in descending order. Based on the priority of the preferred successors, each voter will choose the corresponding delayed election time so that the most up-to-date voter has a higher chance to be elected. If the node's priority is highest, it will become candidate immediately and not wait for the election timeout. For a successor with priority **N** > 0, the next election timeout will be computed as:

```
MIN(retryBackOffMaxMs, retryBackoffMs * 2^(N - 1))
```

where retryBackoffMs and retryBackOffMaxMs are defined by the configs.

Request Schema

```
{
  "apiKey": N,
  "type": "request",
  "name": "EndQuorumEpochRequest",
  "validVersions": "0",
  "fields": [
    { "name": "ClusterId", "type": "string", "versions": "0+", "nullableVersions": "0+", "default": "null"},
    { "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": "ReplicaId", "type": "int32", "versions": "0+",
          "about": "The ID of the replica sending this request"},
        { "name": "LeaderId", "type": "int32", "versions": "0+",
          "about": "The current leader ID or -1 if there is a vote in progress"},
        { "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"}
      ]}
    ]}
  ]
}
```

Note that `LeaderId` and `ReplicaId` will be the same if the leader has been voted. If the replica is a candidate in a current election, then `LeaderId` will be -1.

Response Schema

```
{
  "apiKey": N,
  "type": "response",
  "name": "EndQuorumEpochResponse",
  "validVersions": "0",
  "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+",
          "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"}
      ]}
    ]}
  ]
}
```

EndQuorumEpoch Request Handling

Upon receiving the `EndQuorumEpoch`, the voter checks if the epoch from the request is greater than or equal to its last known epoch. If the epoch is smaller than the last known epoch, or the leader id is not known for this epoch, the request is rejected. Then the voter will check whether it is inside the given preferred successors. If not, return INCONSISTENT_VOTER_SET. If both validation pass, the voter can transit to candidate state immediately if it is first at the list. Otherwise it will wait for a computed back-off timeout to start election as stated in previous section. Before beginning to collect voters, the voter must update the `quorum-state` file.

EndQuorumEpoch Response Handling

If there's no error code, then do nothing. Otherwise if the error code indicates there's already higher epoch leader already ("hey you're old news now so I don't care if you're stepping down or what"), then updating its quorum-state file while transiting to follower state. The leader treats this as a best-effort graceful shutdown. If voters cannot be reached to send `EndQuorumEpoch`, the leader will shutdown without retrying. In the worst case, if none of the `EndQuorumEpoch` requests are received, the election timeout will eventually trigger a new election.

We shall reuse the same FENCED_LEADER_EPCOH error code to indicate that there is a larger leader epoch. Depending on whether the sender is shutting down or not, it would either ignore the response error or become a follower of the leader indicated in the response.

Fetch

The fetch request is sent by both voters and observers to the current leader in order to replicate log changes. For voters this also serves as a liveness check of the leader.

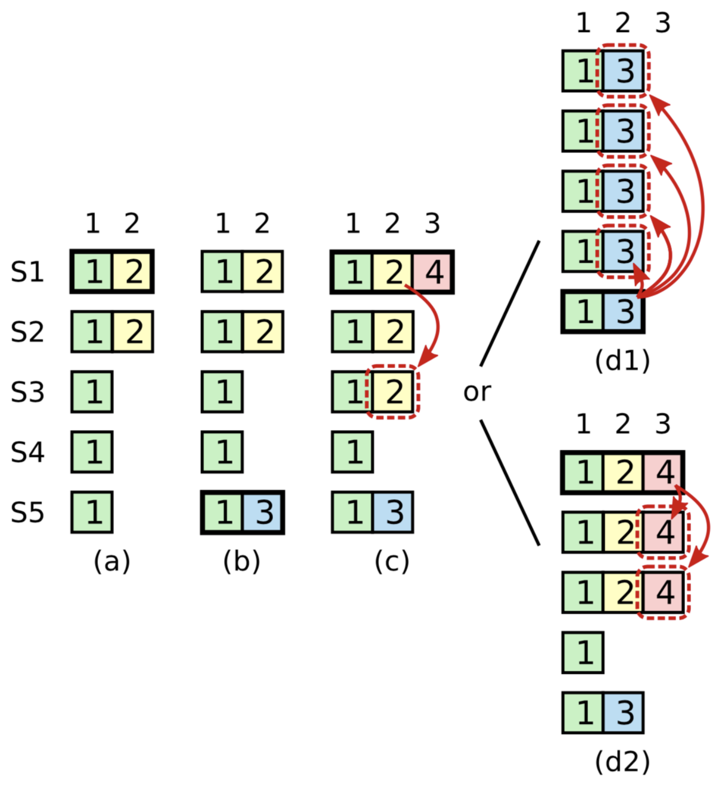
Log reconciliation: The Raft protocol does not guarantee that the replica with the largest offset is always elected. This means that following a leader election, a follower may need to truncate some of the uncommitted entries from its log. The Kafka log replication protocol has a similar problem following leader election and it is resolved with a separate truncating state which is entered after every leader change. In the truncating state, followers will use the `OffsetsForLeaderEpoch` API to find the diverging offset between its log and the leader's. Once that is found, the log is truncated and replication continue.

Rather than following the Kafka approach, this protocol piggybacks log reconciliation on the `Fetch` API, which is more akin to Raft replication. When voters or leaders send a `Fetch` request, in addition to including an offset to fetch from, they also indicate the epoch of the last offset that they have in their local log (we refer to this as the "fetch epoch"). The leader always checks whether the fetch offset and fetch epoch are consistent with its own log. If they do not match, the fetch response will indicate the largest epoch and its end offset before the requested epoch. It is the responsibility of followers to detect leader changes and use `Fetch` responses to truncate the log.

The advantage of this approach is that it simplifies the follower state machine. Basically it just needs to sends fetches to the current expected leader with the fetch offset. The response may indicate either a new leader to fetch from or a new fetch offset. It is also safer because the leader is able to validate the fetch offset on every request rather than relying on the follower to detect the epoch changes.

Since we are using the existing `Fetch` API, we believe that it would make sense to change the normal replication protocol in a similar way. We will leave this as a follow-up KIP opportunity.

Extra condition on commitment: The Raft protocol requires the leader to only commit entries from any previous epoch if the same leader has already successfully replicated an entry from the current epoch. Kafka's ISR replication protocol suffers from a similar problem and handles it by not advancing the high watermark until the leader is able to write a message with its own epoch. The diagram below taken from the Raft dissertation illustrates the scenario:



The problem concerns the conditions for commitment and leader election. In this diagram, S1 is the initial leader and writes "2," but fails to commit it to all replicas. The leadership moves to S5 which writes "3," but also fails to commit it. Leadership then returns to S1 which proceeds to attempt to commit "2." Although "2" is successfully written to a majority of the nodes, the risk is that S5 could still become leader, which would lead to the truncation of "2" even though it is present on a majority of nodes.

In the protocol we are proposing here, we address this issue by not advancing the high watermark after an election until an entry has been written by the new leader with its epoch. So we would allow the election to S5, which had written "3" even though a majority of nodes had written "2." This is is allowable because "2" was not considered committed. Therefore, as long as we do not respond to client's request for the entries it added until the high watermark is advanced beyond it then this extra condition is satisfied.

There is a similar problem in the normal partition replication protocol. When a leader is elected, it cannot know the previous high watermark until the replicas in the ISR have fetched up to the offset at the start of the new epoch. In this case, if a follower is no longer active, the leader can always shrink the ISR to ensure that the high watermark does not remain stuck. In the Raft protocol, we do not have an ISR which leaders are chosen from; we rely on the highest epoch/offset in the log to elect leaders. In order to advance the high watermark after becoming leader, we need to know that there is no other replica that could become leader unless it has replicated all of the data that we commit. This is guaranteed as soon as the leader is able to commit an entry that it itself has written since this is guaranteed to have a larger epoch than any previous entry. However, there is a risk that the client may not send any record immediately and that the high watermark could remain stuck at an offset which is lower than the log end offset. This is the purpose of the "dummy" leader change message that was mentioned in the section on `Vote` handling. After becoming leader, we immediately append this control record, which means that we the high watermark can catch up to the end offset reliably.

Current Leader Discovery: This proposal also extends the `Fetch` response to include a separate field for the receiver of the request to indicate the current leader. This is intended to improve the latency to find the current leader since otherwise a replica would need additional round trips. When a broker is restarted, it will rely on the `Fetch` protocol to find the current leader.

We believe that using the same mechanism would be helpful for normal partitions as well since consumers have to use the `Metadata` API to find the current leader after received a NOT_LEADER_FOR_PARTITION error. However, this is outside the scope of this proposal.

Request Schema

```
{
  "apiKey": 1,
  "type": "request",
  "name": "FetchRequest",
  "validVersions": "0-12",
  "flexibleVersions": "12+",
  "fields": [
    // ---------- Start new field ----------
    { "name": "ClusterId", "type": "string", "versions" "12+", "nullableVersions": "12+", "default": "null",
"taggedVersions": "12+", "tag": 0,
      "about": "The clusterId if known. This is used to validate metadata fetches prior to broker
registration." },
    // ---------- End new field ----------
    { "name": "ReplicaId", "type": "int32", "versions": "0+",
      "about": "The broker ID of the follower, of -1 if this request is from a consumer." },
    { "name": "MaxWaitTimeMs", "type": "int32", "versions": "0+",
      "about": "The maximum time in milliseconds to wait for the response." },
    { "name": "MinBytes", "type": "int32", "versions": "0+",
      "about": "The minimum bytes to accumulate in the response." },
    { "name": "MaxBytes", "type": "int32", "versions": "3+", "default": "0x7fffffff", "ignorable": true,
      "about": "The maximum bytes to fetch.  See KIP-74 for cases where this limit may not be honored." },
    { "name": "IsolationLevel", "type": "int8", "versions": "4+", "default": "0", "ignorable": false,
      "about": "This setting controls the visibility of transactional records. Using READ_UNCOMMITTED
(isolation_level = 0) makes all records visible. With READ_COMMITTED (isolation_level = 1), non-transactional
and COMMITTED transactional records are visible. To be more concrete, READ_COMMITTED returns all data from
offsets smaller than the current LSO (last stable offset), and enables the inclusion of the list of aborted
transactions in the result, which allows consumers to discard ABORTED transactional records" },
    { "name": "SessionId", "type": "int32", "versions": "7+", "default": "0", "ignorable": false,
      "about": "The fetch session ID." },
    { "name": "SessionEpoch", "type": "int32", "versions": "7+", "default": "-1", "ignorable": false,
      "about": "The fetch session epoch, which is used for ordering requests in a session" },
    { "name": "Topics", "type": "[]FetchableTopic", "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": "FetchPartitions", "type": "[]FetchPartition", "versions": "0+",
        "about": "The partitions to fetch.", "fields": [
        { "name": "PartitionIndex", "type": "int32", "versions": "0+",
          "about": "The partition index." },
        { "name": "CurrentLeaderEpoch", "type": "int32", "versions": "9+", "default": "-1", "ignorable": true,
          "about": "The current leader epoch of the partition." },
        { "name": "FetchOffset", "type": "int64", "versions": "0+",
          "about": "The message offset." },
                // ---------- Start new field ----------
        { "name": "LastFetchedEpoch", "type": "int32", "versions": "12+", "default": "-1", "taggedVersions":
"12+", "tag": 0,
          "about": "The epoch of the last replicated record"},
                // ---------- End new field ----------
        { "name": "LogStartOffset", "type": "int64", "versions": "5+", "default": "-1", "ignorable": false,
          "about": "The earliest available offset of the follower replica.  The field is only used when the
request is sent by the follower."},
        { "name": "MaxBytes", "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." }
      ]}
    ]},
    { "name": "Forgotten", "type": "[]ForgottenTopic", "versions": "7+", "ignorable": false,
      "about": "In an incremental fetch request, the partitions to remove.", "fields": [
      { "name": "Name", "type": "string", "versions": "7+", "entityType": "topicName",
        "about": "The partition name." },
      { "name": "ForgottenPartitionIndexes", "type": "[]int32", "versions": "7+",
        "about": "The partitions indexes to forget." }
    ]},
    { "name": "RackId", "type":  "string", "versions": "11+", "default": "", "ignorable": true,
      "about": "Rack ID of the consumer making this request"}
  ]
}
```

Response Schema

```
{
  "apiKey": 1,
  "type": "response",
  "name": "FetchResponse",
  "validVersions": "0-12",
  "flexibleVersions": "12+",
  "fields": [
    { "name": "ThrottleTimeMs", "type": "int32", "versions": "1+", "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": "7+", "ignorable": false,
      "about": "The top level response error code." },
    { "name": "SessionId", "type": "int32", "versions": "7+", "default": "0", "ignorable": false,
      "about": "The fetch session ID, or 0 if this is not part of a fetch session." },
    { "name": "Topics", "type": "[]FetchableTopicResponse", "versions": "0+",
      "about": "The response topics.", "fields": [
      { "name": "Name", "type": "string", "versions": "0+", "entityType": "topicName",
        "about": "The topic name." },
      { "name": "Partitions", "type": "[]FetchablePartitionResponse", "versions": "0+",
        "about": "The topic partitions.", "fields": [
        { "name": "PartitionIndex", "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": "HighWatermark", "type": "int64", "versions": "0+",
          "about": "The current high water mark." },
        { "name": "LastStableOffset", "type": "int64", "versions": "4+", "default": "-1", "ignorable": true,
          "about": "The last stable offset (or LSO) of the partition. This is the last offset such that the
state of all transactional records prior to this offset have been decided (ABORTED or COMMITTED)" },
        { "name": "LogStartOffset", "type": "int64", "versions": "5+", "default": "-1", "ignorable": true,
          "about": "The current log start offset." },
            // ---------- Start new field ----------
        { "name": "DivergingEpoch", "type": "EpochEndOffset", "versions": "12+", "taggedVersions": "12+",
"tag": 0,
          "about": "In case divergence is detected based on the `LastFetchedEpoch` and `FetchOffset` in the
request, this field indicates the largest epoch and its end offset such that subsequent records are known to
diverge",
          "fields": [
            { "name": "Epoch", "type": "int32", "versions": "12+", "default": "-1" },
            { "name": "EndOffset", "type": "int64", "versions": "12+", "default": "-1" }
        ]},
        { "name": "CurrentLeader", "type": "LeaderIdAndEpoch",
          "versions": "12+", "taggedVersions": "12+", "tag": 1, fields": [
          { "name": "LeaderId", "type": "int32", "versions": "0+",
            "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"}
        ]},
                // ---------- End new field ----------
        { "name": "Aborted", "type": "[]AbortedTransaction", "versions": "4+", "nullableVersions": "4+",
"ignorable": false,
          "about": "The aborted transactions.",  "fields": [
          { "name": "ProducerId", "type": "int64", "versions": "4+", "entityType": "producerId",
            "about": "The producer id associated with the aborted transaction." },
          { "name": "FirstOffset", "type": "int64", "versions": "4+",
            "about": "The first offset in the aborted transaction." }
        ]},
        { "name": "PreferredReadReplica", "type": "int32", "versions": "11+", "ignorable": true,
          "about": "The preferred read replica for the consumer to use on its next fetch request"},
        { "name": "Records", "type": "bytes", "versions": "0+", "nullableVersions": "0+",
          "about": "The record data." }
      ]}
    ]}
  ]
}
```

FetchRequest Handling

When a leader receives a `FetchRequest`, it must check the following:

1. Check that the clusterId if not null matches the cached value in `meta.properties`.
2. First ensure that the leader epoch from the request is the same as the locally cached value. If not, reject this request with either the FENCED_LEADER_EPOCH or UNKNOWN_LEADER_EPOCH error.
    a. If the leader epoch is smaller, then eventually this leader's BeginQuorumEpoch would reach the voter and that voter would update the epoch.
    b. If the leader epoch is larger, then eventually the receiver would learn about the new epoch anyways. Actually this case should not happen since, unlike the normal partition replication protocol, leaders are always the first to discover that they have been elected.
3. Check that `LastFetchedEpoch` is consistent with the leader's log. The leader assumes that the `LastFetchedEpoch` is the epoch of the offset prior to `FetchOffset. FetchOffset` is expected to be in the range of offsets with an epoch of `LastFetchedEpoch` or it is the start offset of the next epoch in the log. If not, return a empty response (no records included in the response) and set the `DivergingEpoch` to the largest epoch which is less than fetcher's `LastFetchedEpoch`. This is an optimization to truncation to let the follower truncate as much as possible to get to the starting divergence point with fewer Fetch round-trips. For example, If the fetcher's last epoch is X which does not match the epoch of the offset prior to the fetching offset, then it means that all the records with epoch X on the follower may have diverged and hence could be truncated.
4. If the request is from a voter not an observer, the leader can possibly advance the high-watermark. As stated above, **we only advance the high-watermark if the current leader has replicated at least one entry to majority of quorum to its current epoch**. Otherwise, the high watermark is set to the maximum offset which has been replicated to a majority of the voters.

The check in step 2 is similar to the logic that followers use today in Kafka through the `OffsetsForLeaderEpoch` API. In order to make this check efficient, Kafka maintains a `leader-epoch-checkpoint` file on disk, the contents of which is cached in memory. After every epoch change is observed, we write the epoch and its start offset to this file and update the cache. This allows us to efficiently check whether the leader's log has diverged from the follower and where the point of divergence is. Note that it may take multiple rounds of `Fetch` in order to find the first offset that diverges in the worst case.

FetchResponse Handling

When handling the response, a follower/observer will do the following:

1. If the response contains FENCED_LEADER_EPOCH error code, check the `leaderId` from the response. If it is defined, then update the `quorum-state` file and become a "follower" (may or may not have voting power) of that leader. Otherwise, retry to the `Fetch` request again against one of the remaining voters at random; in the mean time it may receive BeginQuorumEpoch request which would also update the new epoch / leader as well.
2. If the response has the field `DivergingEpoch` set, then truncate from the log all of the offsets greater than or equal to `DivergingEpoch.EndOffset` and truncate any offset with an epoch greater than `DivergingEpoch.EndOffset`.

Note that followers will have to check any records received for the presence of control records. Specifically a follower/observer must check for voter assignment messages which could change its role.

Discussion: Pull v.s. Push Model

In the original Raft paper, the push model is used for replicating states, where leaders actively send requests to followers and count quorum from returned acknowledgements to decide if an entry is committed.

In our implementation, we used a pull model for this purpose. More concretely:

- **Consistency check**: In the push model this is done at the replica's side, whereas in the pull model it has to be done at the leader. The voters / observers send the fetch request associated with its current log end offset. The leader will check its entry for offset, if matches the epoch the leader respond with a batch of entries (see below); if does not match, the broker responds to let the follower truncate all of its current epoch to the next entry of the leader’s previous epoch's end index (e.g. if the follower’s term is X, then return the next offset of term Y’s ending offset where Y is the largest term on leader that is < X). And upon receiving this response the voter / observer has to truncate its local log to that offset.
- **Extra condition on commitment:** In the push model the leader has to make sure it does not commit entries that are not from its current epoch, in the pull model the leader still has to obey this by making sure that the batch it sends in the fetch response contains at least one entry associated with its current epoch.

Based on the recursive deduction from the original paper, we can still conclude that the leader / replica log would never diverge in the middle of the log while the log end index / term still agrees.

Comparing this implementation with the original push based approach:

- On the Pro-side: with the pull model it is more natural to bootstrap a newly added member with empty logs because we do not need to let the leader retry sending request with decrementing nextIndex, instead the follower just send fetch request with zero starting offset. Also the leader could simply reject fetch requests from old-configuration’s members who are no longer part of the group in the new configuration, and upon receiving the rejection the replica knows it should shutdown now — i.e. we automatically resolve the “disruptive servers” issue.
- On the Con-side: zombie leader step-down is more cumbersome, as with the push model, the leader can step down when it cannot successfully send heartbeat requests within a follower timeout, whereas with the pull model, the zombie leader does not have that communication pattern, and one alternative approach is that after it cannot commit any entries for some time, it should try to step down. Also, the pull model could introduce extra latency (in the worst case, a single fetch interval) to determine if an entry is committed, which is a problem especially when we want to update multiple metadata entries at a given time — this is pretty common in Kafka's use case — and hence we need to consider supporting batch writes efficiently in our implementation.

On Performance: There are also tradeoffs from a performance perspective between these two models. In order to commit a record, it must be replicated to a majority of nodes. This includes both propagating the record data from the leader to the follower and propagating the successful write of that record from the follower back to the leader. The push model potentially has an edge in latency because it allows for pipelining. The leader can continue sending new records to followers while it is awaiting the committing of previous records. In the proposal here, we do not pipeline because the leader relies on getting the next offset from the `Fetch` request. However, this is not a fundamental limitation. If the leader keeps track of the last sent offset, then we could instead let the `Fetch` request be pipelined so that it indicates the last acked offset and allows the leader to choose the next offset to send. Basically rather than letting the leader keep sending append requests to the follower as new data arrives, the follower would instead keep sending fetch requests as long as the acked offset is changing. Our experience with Kafka replication suggests that this is unlikely to be necessary, so we prefer the simplicity of the current approach, which also allows us to reuse more of the existing log layer in Kafka. However, we will evaluate the performance characteristics and make improvements as necessary. Note that although we are specifying the initial versions of the protocols in this KIP, there will almost certainly be additional revisions before this reaches production.

DescribeQuorum

The DescribeQuorum API is used by the admin client to show the status of the quorum. This API must be sent to the leader, which is the only node that would have lag information for all of the voters.

Request Schema

```
{
  "apiKey": N,
  "type": "request",
  "name": "DescribeQuorumRequest",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "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." }
      ]
      }]
    }
  ]
}
```

Response Schema

```
{
  "apiKey": N,
  "type": "response",
  "name": "DescribeQuorumResponse",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "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+",
          "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": "HighWatermark", "type": "int64", "versions": "0+"},
        { "name": "CurrentVoters", "type": "[]ReplicaState", "versions": "0+" },
        { "name": "Observers", "type": "[]ReplicaState", "versions": "0+" }
      ]}
    ]}],
  "commonStructs": [
    { "name": "ReplicaState", "versions": "0+", "fields": [
      { "name": "ReplicaId", "type": "int32", "versions": "0+"},
      { "name": "LogEndOffset", "type": "int64", "versions": "0+",
        "about": "The last known log end offset of the follower or -1 if it is unknown"}
    ]}
  ]
}
```

DescribeQuorum Request Handling

This request is always sent to the leader node. We expect `AdminClient` to use the `Metadata` API in order to discover the current leader. Upon receiving the request, a node will do the following:

1. First check whether the node is the leader. If not, then return an error to let the client retry with Metadata. If the current leader is known to the receiving node, then include the `LeaderId` and `LeaderEpoch` in the response.
2. Build the response using current assignment information and cached state about replication progress.

DescribeQuorum Response Handling

On handling the response, the admin client would do the following:

1. If the response indicates that the intended node is not the current leader, then check the response to see if the `LeaderId` has been set. If so, then attempt to retry the request with the new leader.
2. If the current leader is not defined in the response (which could be the case if there is an election in progress), then backoff and retry with `Metadata`.
3. Otherwise the response can be returned to the application, or the request eventually times out.

Cluster Bootstrapping

When the cluster is initialized for the first time, the voters will find each other through the static `quorum.voters` configuration. It is the job of the first elected leader (i.e. the first controller) to generate a UUID that will serve as a unique `clusterId`. We expect this to happen within the controller state machine that defined by KIP-631. This ID will be stored in the metadata log as a message and will be propagated to all brokers in the cluster through the replication protocol defined by this proposal. (From an implementation perspective, the Raft library will provide a hook for the initialization of the clusterId.)

The clusterId replicated through the metadata log will be stored in `meta.properties`. Today, when a broker is restarted, it compares its cached clusterId from `meta.properties` with whatever ID is discovered dynamically in Zookeeper. If the IDs do not match, then the broker shuts down. The purpose of this is to limit the impact of a misconfiguration which causes the broker to connect to the wrong cluster.

We would like to have the same protection once Zookeeper is gone, but it is more challenging since replication of the metadata log itself can be destructive. For example, a broker connecting to the wrong cluster may end up truncating its metadata log incorrectly through the course of replication. A rogue broker can even cause an invalid leader to get elected, which risks the loss of committed data.

The approach we take here to address this problem is to have the core Raft requests include a field for the clusterId. This is validated upon receipt and the INVALID_CLUSTER_ID error code is returned if the values do not match. If the clusterId is not known (perhaps because the broker is being initialized for the first time), then it will specify a clusterId value of "null" and the destination broker will skip validation. This is not a 100% bulletproof solution because we have to allow this exemption for new brokers. It is still possible, for example, if a broker with empty state is started with the ID of an existing voter, which can lead to correctness violations. However, it addresses a large class of common misconfiguration cases and is no worse than what Kafka can protect against today.

Note that there is one subtlety to this proposal. If there is no active leader, which node should be trusted as the authoritative source for the clusterId? If we blindly shutdown the broker after any INVALID_CLUSTER_ID error, then a single broker could end up killing a majority of valid voters in the cluster. To address this issue, we only treat mismatched clusterId errors as fatal when the broker's value conflicts with that of an elected leader. So an INVALID_CLUSTER_ID error in a Vote request is simply treated as a vote rejection, but the same error from a `Fetch` is deemed fatal.

The exact initialization process for a voter when it starts up will be the following:

1. Send a `Fetch` request to any of the current voters including the last known epoch from `quorum-state` and the clusterId from `meta.properties` (if there is one):
    a. If the response indicates INVALID_CLUSTER_ID, then the broker will shutdown.
    b. If no clusterId is received and the broker has a non-null cached clusterId, then continue retrying the `Fetch` request against a random voter until the election timeout expires.
2. If the election timeout expires, become a candidate and send Vote requests including the current cached clusterId (or null if there is none).
    a. If the response indicates INVALID_CLUSTER_ID, the broker will not fail since the inconsistent broker might be in the minority. Instead it will continue retrying until the election completes.
3. If at any time the broker receives a `BeginQuorumEpoch` request from an elected leader with an inconsistent clusterId, then the broker will terminate.
4. If the broker receives a Vote request from one of the voters, the clusterId is not null, and the value not match the cached value, then reject the vote and return INVALID_CLUSTER_ID

For an observer, it is even simpler. It would do the same validation in step 1, but in the case that no leader can be found, it would continue retrying indefinitely. If the observer has no clusterId, then it will send `Fetch` requests with a null clusterId. Once a leader is elected and the clusterId message has become committed, then the observer will update `meta.properties` and begin including the clusterId in all future `Fetch` requests.

Tooling Support

We will add a new utility called *kafka-metadata-quorum.sh* to describe and alter quorum state. As usual, this tool will require `--bootstrap-server` to be provided. We will support the following options:

Describing Current Status

There will be two options available with `describe` :

- `describe --status`: a short summary of the quorum status and the other provides detailed information about the status of replication.
- `describe --replication`: provides detailed information about the status of replication

Here are a couple examples:

```
> bin/kafka-metadata-quorum.sh describe --status
ClusterId:              SomeClusterId
LeaderId:                                0
LeaderEpoch:                        15
HighWatermark:                       234130
MaxFollowerLag:                 34
MaxFollowerLagTimeMs:        15
CurrentVoters:                       [0, 1, 2]

> bin/kafka-metadata-quorum.sh describe --replication
ReplicaId        LogEndOffset        Lag               LagTimeMs        Status
0                        234134                      0                0                        Leader
1                        234130                      4                10                        Follower
2                        234100                      34                15                        Follower
3                        234124                      10                12                        Observer
4                        234130                      4                15                        Observer
```

Metrics

Here’s a list of proposed metrics for this new protocol:

NAME	TAGS	TYPE	NOTE
current-leader	*type=raft-manager*	dynamic gauge	-1 means UNKNOWN
current-epoch	*type=raft-manager*	dynamic gauge	0 means UNKNOWN
current-vote	*type=raft-manager*	dynamic gauge	-1 means not voted for anyone
log-end-offset	*type=raft-manager*	dynamic gauge	
log-end-epoch	*type=raft-manager*	dynamic gauge	
high-watermark	*type=raft-manager*	dynamic gauge	
current-state	*type=raft-manager*	dynamic enum	possible values: "leader", "follower", "candidate", "observer"
number-unknown-voter-connections	*type=raft-manager*	dynamic gauge	number of unknown voters whose connection information is not cached; would never be larger than quorum-size
election-latency-max/avg	*type=raft-manager*	dynamic gauge	measured on each voter as windowed sum / avg, start when becoming a candidate and end on learned or become the new leader
commit-latency-max/avg	*type=raft-manager*	dynamic gauge	measured on leader as windowed sum / avg, start when appending the record and end on hwm advanced beyond
fetch-records-rate	*type=raft-manager*	windowed rate	apply to follower and observer only
append-records-rate	*type=raft-manager*	windowed rate	apply to leader only
poll-idle-ratio-avg	*type=raft-manager*	windowed average	

Client Interactions

Since this specific quorum implementation is only to be used by Kafka internally, we do not need to add new public protocols for clients. More specifically, the leader of the quorum would act as the controller of the cluster, and any client requests that requires updating the metadata (previously stored in ZK) would be interpreted as appending new record(s) to the quorum's internal log.

For some operations that require updating multiple metadata entries such as leader migration (i.e. previously as multiple ZK writes updating more than one ZK path), they would be interpreted as a batch-record appends.

The record append (either singular or batch) would not return until the leader acknowledged that high watermark has advanced past the appended records' offsets — they have been committed. For batch appends, we would only return when all records have been committed.

Note it is possible that a request could time out before the leader has successfully committed the records, and the client or the broker itself would retry, which would result in duplicated updates to the quorum. Since in Kafka's usage, all updates are overwrites which are idempotent (as the nature of configuration is a key-value mapping). Therefore, we do not need to implement serial number or request caching to achieve "exactly-once".

Note there are several controller operations that involves ZK updates and are not part of this KIP's scope:

- Brokers can directly update ZK for shrinking / expanding ISR; this will be replaced with AlterISR request sent from leaders to the controller (KIP-497: Add inter-broker API to alter ISR). The controller would then update the metadata by appending a batch of entries to its metadata log where each topic-partition represents one entry.
- Admin requests for reassign replicas will be replaced with an AlterPartitionAssignments request to the controller (KIP-455: Create an Administrative API for Replica Reassignment). The controller would update the metadata by appending a batch of entries to its metadata log where each topic-partition represents one entry.
- Existing admin request for config changes etc will be translated to a batch of updates to its metadata log.

Log Compaction and Snapshots

The metadata log described in this KIP can grow unbounded. As discussed in the introduction, our approach to managing the size of the replicated log is described in a separate proposal: KIP-630: Kafka Raft Snapshot. Each broker in KIP-500 will be an observer of the metadata log and will materialize the entries into a cache of some kind. New and slow brokers will catch up to the leader by 1) fetching the snapshot and 2) fetching the replicated log after the snapshot. This will provide a stronger guarantee on the consistency of the metadata than is possible today.

Quorum Performance

The goal for Raft quorum is to replace Zookeeper dependency and reach higher performance for metadata operations. In the first version, we will be building necessary metrics to monitor the end-to-end latency from admin request (AlterPartitionReassignments) and client request being accepted to being committed. We shall monitor the time spent on local, primarily the time to fsync the new records and time to apply changes to the state machine, which may not be really a trivial operation. Besides we shall also monitor the time used to propagate change on the remote, I.E. latency to advance the high watermark. Benchmarks will also be built to compare the efficiency for a 3-node broker cluster using Zookeeper vs Raft, under heavy load of metadata changes. We shall also be exploring existing distributed consensus system load frameworks at the same time, but this may beyond the scope of KIP-595.

Test Plan

Raft is a well-studied consensus protocol, but there are some notable differences from the classic protocol in this proposal. As we did for the Kafka replication protocol in KIP-320, we will use model checking to validate the replication semantics of this protocol. (Note that validation of the controller state machine will be handled as part of KIP-631.)

Our primary method for testing the implementation is through Discrete Event Simulation (DES). Our prototype implementation already includes the basic framework. DES allows us to test a large number of deterministically generated random scenarios which include various kinds of faults (such as network partitions). It allows us to define system invariants programmatically which are then checked after each step in the simulation. This has already been extremely useful identifying bugs throughout development of the prototype.

Other than that, we will use the typical suite of unit/integration/system tests

Rejected Alternatives

Use an existing Raft library: Log replication is at the core of Kafka and the project should own it. Dependence on a third-party system or library would defeat one of the central motivations for KIP-500. There would be no easy way to evolve a third-party component according to the specific needs of Kafka. For example, we may eventually use this protocol for partition-level replication, but it would make compatibility much more difficult if we cannot continue to control the log layer. So if we must control both the log layer and the RPC protocol, then the benefit of a third-party library is marginal and the cost is an unnecessary constraint on future evolution. Furthermore, Raft libraries typically bring in their own RPC mechanism, serialization formats, have their own monitoring, logging, etc. All of this requires additional configuration the user needs to understand.

Start with partition replication using Raft: As mentioned in several places in this doc, we are in favor of making Raft an available replication mechanism for individual partitions. That begs the question of whether we should just start there rather than creating a separate protocol for the metadata quorum? We were indeed tempted to do so, but ultimately decided not to because of the significant amount of orthogonal overhead it adds to the KIP-500 roadmap. As an example, we would need to reconsider how metadata is propagated to clients since the controller would no longer be responsible for elections. We could continue routing everything through the controller as is currently done today, but then that weakens one of the main motivations for Raft-based partition replication. Additionally, we would need a Raft protocol which could efficiently batch elections. These problems are tractable, but solving them takes us well out of the scope of KIP-500. With that said, we wanted to make this protocol to mirror the current replication protocol in some respects to make the eventual transition easier. That is one of the reasons we opted for a pull–based protocol as mentioned below. It is also the reason we decided to reuse existing log compaction semantics.

Push vs Pull: Raft is specified as a push protocol. The leader must track the state of of all replicas in order to reconcile differences and it is responsible for pushing the changes to them. Kafka's replication protocol on the other hand is pull-based. Replicas know which offset they need and fetch it from the leader. Reconciliation is driven by the replica. This makes it suited to large numbers of observer replicas since the leader only needs to track the status of replicas in the quorum. We have opted to stick with the pull-based model in this protocol for this reason and because it allows for easier reuse of the log layer. This also simplifies the transition to raft replication for topic partitions.

Support Atomic Arbitrary Quorum Change: In the Raft literature, another quorum change approach proposed was to support arbitrary number of node changes for the existing quorum in one shot. This requires a larger scope of the code logic change as we need to maintain two sets of node quorum inside the cluster.

The idea is that while during the migration from an old quorum to a new quorum, each appended message during this period has to be routed to all the existing nodes inside both old config and new config, as well as majority votes from two quorums. The benefit of this approach is that the quorum change becomes atomic across the group, but it comes with the cost of complexing normal operations and provides little practical values, since most operation in production does not require more than one server migration. The literature explicitly does not recommend this approach and the example implementation is indeed hard to reason about correctness.

Observer Based Promotion Observer based self promotion also has merits because we do pull based model already. It could just monitor its lagging from the leader, and when the gap is consistently below a certain threshold for a couple of rounds of fetch, we may call this observer ready to be joining the group. This special logic could be embedded inside the observer when handling `BeginQuorumEpoch` request sent from the leader. It will put the request in a delayed queue and only reply the leader for a success once the observer assumed itself as `in-sync`. However at the moment, we want to centralize the information about the quorum instead of letting it scatter around and increase the interaction complexity. In that sense, this approach may not be preferable.

References

- Ongaro, Diego, and John Ousterhout. "In search of an understandable consensus algorithm." *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 2014.
- Ongaro, Diego. *Consensus: Bridging theory and practice*. Diss. Stanford University, 2014.
- Howard, Heidi, et al. "Raft refloated: Do we have consensus?." *ACM SIGOPS Operating Systems Review* 49.1 (2015): 12-21.
- R. Van Renesse. Paxos made moderately complex. http://www.cs.cornell.edu/courses/cs7412/2011sp/paxos.pdf, 2011.