KIP-642: Dynamic quorum reassignment

Status

Current state: Draft

Discussion thread:

JIRA:

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

Motivation

KIP-595 proposed a raft-like replication protocol for the metadata quorum that is required by KIP-500. This proposal adds support for dynamic quorum reassignment.

- To increase or reduce the size of the voter set
- To change quorum membership in the case of failures (e.g. disaster recovery)
- To switch between dedicated deployment and mixed deployment easily, with making controller nodes standalone or integrate with data nodes

Zookeeper itself began supporting dynamic reconfiguration in 3.5, so this is also about reaching parity with the capabilities Kafka has today by leveraging Zookeeper. Finally, dynamic reassignment is also a prerequisite to making Raft-based replication available to local topic partitions.

Quorum Reassignment

The protocol for changing the active voters is well-described in the Raft literature. The high-level idea is to use a new control record type to write quorum changes to the log. Once a quorum change has been written and committed to the log, then the quorum change can take effect.

This will be a new control record type with Type=2 in the key schema. Below we describe the message value schema used for the quorum change messages written to the log.

```
{
  "type": "message",
  "name": "VoterAssignmentMessage",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
      {"name": "ClusterId", "type": "string", "versions": "0+"}
      {"name": "CurrentVoters", "type": "[]Voter", "versions": "0+", "fields": [
        {"name": "VoterId", "type": "int32", "versions": "0+"}
      ]},
      {"name": "TargetVoters", "type": "[]Voter", "versions": "0+", "nullableVersions": "0+", "fields": [
        {"name": "VoterId", "type": "int32", "versions": "0+"}
      ]}
  ]
}
```

DescribeQuorum

The DescribeQuorum API is used by the admin client to show the status of the quorum. This includes showing the progress of a quorum reassignment and viewing the lag of followers and observers.

Unlike the `DiscoverBrokers` request, 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": 56,
  "type": "request",
  "name": "DescribeQuorumRequest",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "Topics", "type": "[]DescribeQuorumTopicRequest",
      "versions": "0+", "fields": [
      { "name": "TopicName", "type": "string", "versions": "0+", "entityType": "topicName",
        "about": "The topic name." },
      { "name": "Partitions", "type": "[]DescribeQuorumPartitionRequest",
        "versions": "0+", "fields": [
        { "name": "PartitionIndex", "type": "int32", "versions": "0+",
          "about": "The partition index." }
      ]
      }]
    }
  ]
}
```

Response Schema

```
{
  "apiKey": 56,
  "type": "response",
  "name": "DescribeQuorumResponse",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "Topics", "type": "[]DescribeQuorumTopicResponse",
      "versions": "0+", "fields": [
      { "name": "TopicName", "type": "string", "versions": "0+", "entityType": "topicName",
        "about": "The topic name." },
      { "name": "Partitions", "type": "[]DescribeQuorumPartitionResponse",
        "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": "TargetVoters", "type": "[]ReplicaState", "versions": "0+" },  // NEW FIELD
        { "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 `DiscoverBrokers and Metadata` 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 `DiscoverBrokers`. 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 `DiscoverBrokers and Metadata`.
3. Otherwise the response can be returned to the application, or the request eventually times out.

AlterPartitionReassignments

The `AlterPartitionReassignments` API was introduced in KIP-455 and is used by the admin client to change the replicas of a topic partition. We propose to reuse it for Raft voter reassignment as well.

The effect of an `AlterPartitionReassignments` is to change the `TargetVoters` field in the `VoterAssignmentMessage` defined above. Once this is done, the leader will begin the process of bringing the new nodes into the quorum and kicking out the nodes which are no longer needed.

Cancellation: If `Replicas` is set to null in the request, then effectively we will cancel an ongoing reassignment and leave the quorum with the current voters. The more preferable option is to always set the intended `TargetVoters`. Note that it is always possible to send a new `AlterPartitionReassignments` request even if the pending reassignment has not finished. So if we are in the middle of a reassignment from (1, 2, 3) to (4, 5, 6), then the user can cancel the reassignment by resubmitting (1, 2, 3) as the `TargetVoters`.

AlterPartitionReassignments Request Handling

This request must be sent to the leader (which is the controller in KIP-500). Upon receiving the `AlterPartitionReassignments` request, the node will verify a couple of things:

1. If the target node is not the leader of the quorum, return NOT_LEADER_FOR_PARTITION code to the admin client.
2. Check whether `TargetVoters` in the request matches that from the latest assignment. If so, check whether it has been committed. If it has, then return immediately. Otherwise go to step 4.
3. If the `TargetVoters` in the request does not match the current `TargetVoters`, then append a new `VoterAssignmentMessage` to the log.
4. Return successfully only when the desired `TargetVoters` has been safely committed to the log.

After returning the result of the request, the leader will begin the process to modify the quorum. This is described in more detail below, but basically the leader will begin tracking the fetch state of the target voters. It will then make a sequence of single-movement alterations by appending new `VoterAssignmentMessage` records to the log.

AlterPartitionReassignments Response Handling

The current response handling of this API in the `AdminClient` is sufficient for our purpose:

1. If the response error indicates that the intended node is not the current leader, use the `Metadata` API to find the latest leader and retry.
2. Otherwise return the successful response to the application.

Note that the `AdminClient` should retry the `AlterPartitionReassignments` request if it times out before the reassignment had been committed to the log. If the caller does not continue retrying the operation, then there is no guarantee about whether or not the reassignment had been successfully received by the cluster.

Once the reassignment has been accepted by the leader, then a user can monitor the status of the reassignment through the `DescribeQuorum` API.

Quorum Change Protocol

This protocol allows arbitrary quorum changes through the `AlterPartitionReassignments` API. Internally, we arrive at the target quorum by making a sequence of single-member changes.

Before getting into the details, we consider an example which summarizes the basic mechanics. Suppose that we start with a quorum consisting of the replicas (1, 2, 3) and we want to change the quorum to (4, 5, 6). When the leader receives the request to alter the quorum, it will write a message to the current quorum indicating the desired change:

```
currentVoters: [1, 2, 3]
targetVoters: [4, 5, 6]
```

Once this message has been committed to the quorum, the leader will add 4 to the current quorum by writing a new log entry:

```
currentVoters: [1, 2, 3, 4]
targetVoters: [4, 5, 6]
```

A common optimization for this step is to wait until 4 has caught up near the end of the log before adding it. Similarly, we may choose to add another replica first if it has already caught up.

Once the check passes, the leader will finally send out the control record to modify the existing voter list. Once that entry has been committed to the log, the new membership takes effect.

Then we could remove one of the existing replicas (say 3) by writing a new message:

```
currentVoters: [1, 2, 4]
targetVoters: [4, 5, 6]
```

Continuing in this way, eventually arrive at the following state:

```
currentVoters: [1, 4, 5, 6]
targetVoters: [4, 5, 6]
```

Suppose that replica 1 is the current leader. Before finalizing the reassignment, it will resign its leadership by sending `EndQuorumEpoch` to the other voters. Once the new leader takes over, it will complete the reassignment by removing replica 1 from the current voters. At this point, the quorum change has completed and we can clear the `targetVoters` field:

```
currentVoters: [4, 5, 6]
targetVoters: null
```

The previous leader will resume as a voter until it knows that it is no longer one of them. The leader will always choose to remove itself last as long as progress can still be made.

To summarize the protocol details:

1. Quorum changes are executed through a series of single-member changes.
2. The `BeginQuorumEpoch` API is used to notify new voters of their new status.
3. The `EndQuorumEpoch` API is used by leaders when they need to remove themselves from the quorum.

Reassignment Algorithm

Upon receiving an `AlterPartitionReassignments` request, the leader will do the following:

1. Append an `VoterAssignmentMessage` to the log with the current voters as `CurrentVoters` and the `Replicas` from the `AlterPartitionReassignments` request as the `TargetVoters`.
2. Leader will compute 3 sets based on `CurrentVoters` and `TargetVoters`:
    a. `RemovingVoters`: voters to be removed
    b. `RetainedVoters`: voters shared between current and target
    c. `NewVoters`: voters to be added
3. Based on comparison between `size(NewVoters)` and `size(RemovingVoters)`,
    a. If `size(NewVoters) >= size(RemovingVoters)`, pick one of `NewVoters` as `NV` by writing a record with `CurrentVoters=CurrentVoters + NV`, and `TargetVoters=TargetVoters`.
    b. else pick one of `RemovingVoters` as `RV`, preferably a non-leader voter, by writing a record with `CurrentVoters=CurrentVoters - RV`, and `TargetVoters=TargetVoters`.
4. Once the record is committed, the membership change is safe to be applied. Note that followers will begin acting with the new voter information as soon as the log entry has been appended. They do not wait for it to be committed.
5. As there is a potential delay for propagating the removal message to the removing voter, we piggy-back on the `Fetch` to inform the voter to downgrade immediately after the new membership gets committed. See the error code NOT_FOLLOWER.
6. The leader will continue this process until one of the following scenarios happens:
    a. If `TargetVoters = CurrentVoters`, then the reassignment is done. The leader will append a new entry with `TargetVoters=null` to the log.
    b. If the leader is the last remaining node in `RemovingVoters`, then it will step down by sending `EndQuorumEpoch` to the current voters. It will continue as a voter until the next leader removes it from the quorum.

Note that there is one main subtlety with this process. When a follower receives the new quorum state, it immediately begins acting with the new state in mind. Specifically, if the follower becomes a candidate, it will expect votes from a majority of the new voters specified by the reassignment. However, it is possible that the `VoterAssignmentMessage` gets truncated from the follower's log because a newly elected leader did not have it in its log. In this case, the follower needs to be able to revert to the previous quorum state. To make this simple, voters will only persist quorum state changes in `quorum-state` after they have been committed. Upon initialization, any uncommitted state changes will be found by scanning forward from the `LastOffset` indicated in the `quorum-state`.

In a similar vein, if the `VoterAssignmentMessage` fails to be copied to all voters before a leader failure, then there could be temporary disagreement about voter membership. Each voter must act on the information they have in their own log when deciding whether to grant votes. It is possible in this case for a voter to receive a request from a non-voter (according to its own information). Voters must reject votes from non-voters, but that does not mean that the non-voter cannot ultimately win the election. Hence when a voter receives a `VoteRequest` from a non-voter, it must then become a candidate.

Observer Promotion

To ensure no downtime of the cluster switch, newly added nodes should already be acting as an up-to-date observer to avoid unnecessary harm to the cluster availability. There are two approaches to achieve this goal, either from leader side or from observer side:

- **Leader based approach**: leader is responsible for tracking the observer progress by monitoring the new entry replication speed, and promote the observer when one replication round is less than election timeout, which suggests the gap is sufficiently small. This idea adds burden to leader logic complexity and overhead for leadership transfer, but is more centralized progress management.
- **Observer based approach**: a self-nomination approach through observer initiates readIndex call to make sure it is sufficiently catching up with the leader. If we see consecutive rounds of readIndex success within election timeout, the observer will trigger a config change to add itself as a follower on next round. This is a de-centralized design which saves the dependency on elected leader to decide the role, thus easier to reason about.

To be effectively collecting information from one place and have the membership change logic centralized, leader based approach is more favorable. However, tracking the progress of an observer should only happen during reassignment, which is also the reason why we may see incomplete log status from DescribeQuorum API when the cluster is stable.

Quorum Bootstrapping

This section discusses the process for establishing the quorum when the cluster is first initialized and afterwards when nodes are restarted. There are essentially two problems when a broker starts up:

1. How to find the connection information of the other brokers and to advertise our own connection information
2. How to find the current status of the quorum (i.e. which node is the leader and who are the other voters)

As with Kafka today, we assume that each broker has an advertised listener for inter-broker communications. A key question in this design concerns the *pe rmanence* of the advertised listener. When a broker is restarted, is it safe to assume that it will re-register with the same advertised listener or not? Kafka today sets no requirement on this; a broker can change its advertised listener every time it restarts. This can be easily supported because the Zookeeper connection information does not change and the broker can use Zookeeper for broker discovery. Without Zookeeper to lean on, we must either have stricter constraints on the mutability of the advertised listener, or we must have a dynamic mechanism to discover changes.

To illustrate the problem, consider the case when all brokers are shutdown and then restarted. When the first brokers are starting up, there will not be a sufficient number of them to form a majority and elect a leader for the metadata log. We cannot rely on the metadata log as we can with Zookeeper in this case to be able to register. At a minimum, the current voters need some way to find each other in order to elect a leader. If we are allowed to assume that the advertised listeners do not change, then each voter's advertised listener could be written into the `quorum-state` file for example. However, if we cannot assume a static advertised listener, then we need a different approach.

A second case occurs when new brokers are added to the cluster. The original voter configuration specified by `bootstrap.quorum.voters` may no longer be relevant since the voters may have been changed by a dynamic reassignment. So how should the new brokers discover where the current voters are and how to connect to them?

The approach we take here is intended to give the most flexibility to different environments. We have added a `bootstrap.servers` configuration, which is used in a similar way to the same configuration in clients. When a broker first starts up, it will attempt to connect to one of the endpoints specified by this configuration in order to discover other brokers and the quorum. The endpoints defined by this config could point to a load balancer or a VIP; they could also point to an explicit set of brokers. Once the broker has connected, it will use a new `AdvertiseBroker` API in order to install its endpoint information and make itself reachable to other nodes. The same API is used to discover other nodes in the cluster.

Once connected to the cluster, the broker will send a `Fetch` to any one of the existing brokers in order to find the current leader.

1. The broker cannot find connection information for any of the voters.

When first initialized, we rely on the `bootstrap.quorum.voters` configuration to define the expected voters. Brokers will only rely on this when starting up if there is no quorum state written to the log and if the broker fails to discover an existing quorum through the `DiscoverBrokers` API. We suggest that a `--bootstrap` flag could be passed through `kafka-server-start.sh` when the cluster is first started to add some additional safety. In that case, we would only use the config when the flag is specified and otherwise expect to discover the quorum dynamically.

Assuming the broker fails to discover an existing quorum, it will then check its `broker.id` to see if it is expected to be one of the initial voters. If so, then it will immediately include itself in `DiscoverBrokers` responses. Until the quorum is established, brokers will send `DiscoverBrokers` to the nodes in `boo tstrap.servers` and other discovered voters in order to find the other expected members. Once enough brokers are known, the brokers will begin a vote to elect the first leader.

Note that the `DiscoverBrokers` API is intended to be a gossip API. Quorum members use this to advertise their connection information on bootstrapping and to find new quorum members. All brokers use the latest quorum state in response to `DiscoverBrokers` requests that they receive. The latest voter state is determined by looking at 1) the leader epoch, and 2) boot timestamps. Brokers only accept monotonically increasing updates to be cached locally. This approach is going to replace the current ZooKeeper ephemeral nodes based approach for broker (de-)registration procedure, where broker registration will de done by a new broker gossiping its host information via DiscoverBrokersRequest. As for broker de-registration, it will be done separately for voters and observers: for voters, we would first change the quorum to degrade the shutting down voters to observers of the quorum; and for observers, they can directly shuts down themselves after they've let the controller to move its hosted partitions to other hosts.

Bootstrapping Procedure

To summarize, this is the procedure that brokers will follow on initialization:

1. Upon starting up, brokers always try to bootstrap its knowledge of the quorum by first reading the `quorum-state` file and then scanning forward from `AppliedOffset` to the end of the log to see if there are any changes to the quorum state. For newly started brokers, the log / file would all be empty so no previous knowledge can be restored.
2. If after step 1), there's some known quorum state along with a leader / epoch already, the broker would:
    a. Promote itself from observer to voter if it finds out that it's a voter for the epoch.
    b. Start sending `Fetch` request to the current leader it knows (it may not be the latest epoch's leader actually).
3. Otherwise, it will try to learn the quorum state by sending `DiscoverBrokers` to any other brokers inside the cluster via `boostrap.servers` as the second option of quorum state discovery.
    a. As long as a broker does not know all the current quorum voter's connections, it should continue periodically ask other brokers via `Disco verBrokers.`
4. Send out MetadataRequest to the discovered brokers to find the current metadata partition leader.
    a. As long as a broker does not know the current quorum (including the leader and the voters), it should continue periodically ask other brokers via `Metadata`.
5. If even step 3) 4) cannot find any quorum information – e.g. when there's no other brokers in the cluster, or there's a network partition preventing this broker to talk to others in the cluster – fallback to the third option of quorum state discover by checking if it is among the brokers listed in `quor um.voters`.
    a. If so, then it will promote to voter state and add its own connection information to the cached quorum state and return that in the `Discov erBrokers` responses it answers to other brokers; otherwise stays in observer state.
    b. In either case, it continues to try to send `DiscoverBrokers` to all other brokers in the cluster via `boostrap.servers.`
6. For any voter, after it has learned a majority number of voters in the expected quorum from `DiscoverBrokers` responses, it will begin a vote.

When a leader has been elected for the first time in a cluster (i.e. if the leader's log is empty), the first thing it will do is append a `VoterAssignmentMessa ge` (described in more detail below) which will contain `quorum.voters` as the initial `CurrentVoters`. Once this message has been persisted in the log, then we no longer need `quorum.voters` and users can safely roll the cluster without this config.

ClusterId generation: Note that the first leader is also responsible for providing the `ClusterId` field which is part of the `VoterAssignmentMessage`. If the cluster is being migrated from Zookeeper, then we expect to reuse the existing clusterId. If the cluster is starting for the first time, then a new one will be generated -- in practice it is the user's responsibility to guarantee the newly generated clusterId is unique. Once this message has been committed to the log, the leader and all future leaders will strictly validate that this value matches the `ClusterId` provided in requests when receiving `Vote`, `BeginEpoch`, `EndEpoch`, and `Fetch` requests.

The leader will also append a `LeaderChangeMessage` as described in the `VoteResponse` handling above. This is not needed for correctness. It is just useful for debugging and to ensure that the high watermark always has an opportunity to advance after an election.

Bootstrapping Example

With this procedure in mind, a convenient way to initialize a cluster might be the following.

1. Start with a single node configured with `broker.id=0` and `quorum.voters=0`.
2. Start the node and verify quorum status. This would also be a good opportunity to make dynamic config changes or initialize security configurations.
3. Add additional nodes to the cluster and verify that they can connect to the quorum.
4. Finally use `AlterPartitionReassignments` to grow the quorum to the intended size.

Changes to Quorum State

```
{
  "type": "data",
  "name": "QuorumStateMessage",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
      {"name": "ClusterId", "type": "string", "versions": "0+"},
          {"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+"}
      ]},
      {"name": "TargetVoters", "type": "[]Voter", "versions": "0+", "nullableVersions": "0+", "fields": [
        {"name": "VoterId", "type": "int32", "versions": "0+"}
      ]}
  ]
}
```

Discussion: Replication Progress Timeout for Zombie Leader

There's one caveat of the pull-based model: 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".

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 start sending `Metadata` request to random nodes in the cluster to understand the latest quorum. If it couldn't connect to any known voter, the old leader shall reset the connection information and send out `DiscoverBrokers`. And if the returned response includes a newer epoch leader, this zombie leader would step down and becomes an observer; and if it realized that it is still within the current quorum's voter list, it would start fetching from that leader. Note that the node will remain a leader until it finds that it has been supplanted by another voter.

Rejected Alternatives

If there are alternative ways of accomplishing the same thing, what were they? The purpose of this section is to motivate why the design is the way it is and not some other way.