KIP-996: Pre-Vote

Status

Current state: Implementing

Discussion thread: https://lists.apache.org/thread/pqj9f1r3rk83oqtxxtg6y5h7m7cf56r2

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

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

Motivation

This KIP will go over scenarios where we might expect disruptive servers and discuss how **Pre-Vote** (as originally detailed in the extended Raft paper and in KIP-650) along with **Followers rejecting Pre-Vote Requests** can ensure correctness when it comes to network partitions.

Pre-Vote is the idea of “canvassing” the cluster to check if it would receive a majority of votes - if yes it increases its epoch and sends a disruptive vote request. If not, it does not increase its epoch and does not send a vote request.

Followers rejecting Pre-Vote Requests entails servers rejecting any Pre-Vote requests received prior to their own fetch timeout expiring. In other words, Followers should reject Pre-Votes. The idea here is if we've recently heard from a leader, we should not attempt to elect a new one just yet.

Throughout this KIP, we will differentiate between Pre-Vote and the original Vote request behavior with **"Pre-Vote"** and **"standard Vote"**.

Disruptive server scenarios

When a follower becomes partitioned from the rest of the quorum, it will continuously increase its epoch to start elections until it is able to regain connection to the leader/rest of the quorum. When the server regains connectivity, it will disturb the rest of the quorum as they will be forced to participate in an unnecessary election. While this situation only results in one leader stepping down, as we start supporting larger quorums these events may occur more frequently per quorum.

For instance, here's a great example from https://decentralizedthoughts.github.io/2020-12-12-raft-liveness-full-omission/ which demonstrates a scenario where we could have flip-flopping leadership changes.

blocked URL

Let's say server 3 is the current leader. Server 4 will eventually start an election because it is unable to find the leader, causing Server 2 to transition to Unattached. Server 4 will not be able to receive enough votes to become leader, but Server 2 will be able to once its election timer expires. As Server 5 is also unable to communicate to Server 2, this kicks off a back and forth of leadership transition between Servers 2 and 3.

While network partitions may be the main issue we expect to encounter/mitigate impact for, it’s possible that bugs and user error create similar effects to network partitions that we need to guard against. For instance, a bug which causes fetch requests to periodically timeout or setting `controller.quorum.fetch.timeout.ms` and other related configs too low.

Public Interfaces

We will add a new field `PreVote` to VoteRequests and VoteResponses to signal whether the requests and responses are for Pre-Votes. The server does *not* increase its epoch prior to sending a Pre-Vote request.

```
{
  "apiKey": 52,
  "type": "request",
  "listeners": ["controller"],
  "name": "VoteRequest",
  "validVersions": "0-1",
  "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": "ReplicaEpoch", "type": "int32", "versions": "0+",
          "about": "The epoch of the prospective or candidate sending the request"},
        { "name": "ReplicaId", "type": "int32", "versions": "0+", "entityType": "brokerId",
          "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"},
        { "name": "PreVote", "type": "bool", "versions": "1+",
          "about": "Whether the request is a PreVote request (no epoch increase) or not."}
...
}
```

```
{
  "apiKey": 52,
  "type": "response",
  "name": "VoteResponse",
  "validVersions": "0-1",
  "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+", "entityType": "brokerId",
          "about": "The ID of the current leader or -1 if the leader is unknown."},
        { "name": "LeaderEpoch", "type": "int32", "versions": "0+",
          "about": "The latest known leader epoch"},
        { "name": "VoteGranted", "type": "bool", "versions": "0+",
          "about": "True if the vote was granted and false otherwise"},
        { "name": "PreVote", "type": "bool", "versions": "1+",
          "about": "Whether the response is a PreVote response or not."}

...
}
```

Proposed Changes

We add a new quorum state `Prospective` for servers which are sending Pre-Vote requests as well as new state transitions. The original (left) and new states (right) are below for comparison.

```
 * Unattached|Resigned transitions to:
 *    Unattached: After learning of a new
election with a higher epoch
 *    Voted: After granting a vote to a
candidate
 *    Candidate: After expiration of the
election timeout
 *    Follower: After discovering a leader
with an equal or larger epoch
 *
 * Voted transitions to:
 *    Unattached: After learning of a new
election with a higher epoch
 *    Candidate: After expiration of the
election timeout
 *
 * Candidate transitions to:
 *    Unattached: After learning of a new
election with a higher epoch
 *    Candidate: After expiration of the
election timeout
 *    Leader: After receiving a majority of
votes
 *
 * Leader transitions to:
 *    Unattached: After learning of a new
election with a higher epoch
 *    Resigned: When shutting down gracefully
 *
 * Follower transitions to:
 *    Unattached: After learning of a new
election with a higher epoch
 *    Candidate: After expiration of the
fetch timeout
 *    Follower: After discovering a leader
with a larger epoch
```

```
 * Unattached|Resigned transitions to:
 *    Unattached: After learning of a candidate with a higher epoch
(clarifying language)
 *    Voted: After granting a standard vote to a candidate
(clarifying language)
 *    Prospective: After expiration of the election timeout
 *    Follower: After discovering a leader with an equal or larger
epoch
 *
 * Voted transitions to:
 *    Unattached: After learning of a candidate with a higher epoch
 *    Prospective: After expiration of the election timeout
 *    Follower: After discovering a leader with an equal or larger
epoch (missed in original docs)

 *
 * Prospective transitions to:
 *    Unattached: After learning of a candidate with a higher
epoch
 *    Prospective: After expiration of the election timeout
 *    Candidate: After receiving a majority of pre-votes
 *    Follower: After discovering a leader with an equal or larger
epoch
 *
 * Candidate transitions to:
 *    Unattached: After learning of a candidate with a higher
epoch
 *    Prospective: After expiration of the election timeout
 *    Leader: After receiving a majority of standard votes
 *    Follower: After discovering a leader with an equal or larger
epoch (missed in original docs)
 *
 * Leader transitions to:
 *    Unattached: After learning of a candidate with a higher epoch
 *    Resigned: When shutting down gracefully
 *
 * Follower transitions to:
 *    Unattached: After learning of a candidate with a higher
epoch
 *    Prospective: After expiration of the fetch timeout
 *    Follower: After discovering a leader with a larger epoch
```

A follower will now transition to `Prospective` instead of `Candidate` when its fetch timeout expires. Servers will only be able to transition to `Candidate` state from the `Prospective` state.

A `Prospective` server will send a `VoteRequest` with the `PreVote` field set to `true` and `ReplicaEpoch` set to its current, unbumped epoch. If [majority - 1] of `VoteResponse` grant the vote, the server will transition to `Candidate` and will then bump its epoch up and send a `VoteRequest` with `PreVote` set to `false` (which is the original behavior).

When servers receive VoteRequests with the `PreVote` field set to `true`, they will respond with `VoteGranted` set to

- `true` **if they are not a Follower** *and* the epoch and offsets in the Pre-Vote request satisfy the same requirements as a standard vote
- `false` if otherwise

When a server receives VoteResponses, it will follow it up with another VoteRequest with `PreVote` set to either `true` (send another Pre-Vote) or `false` (send a standard vote)

- `false` (standard vote) if the server has received [majority - 1] VoteResponses with `VoteGranted` set to `true` within [election.timeout.ms + a little randomness]
- `true` (another Pre-Vote) if the server receives [majority] VoteResponse with `VoteGranted` set to `false` within [election.timeout.ms + a little randomness]
- `true` if the server does not receive enough votes (granted or rejected) within [election.timeout.ms + a little randomness]

If a server happens to receive multiple `VoteResponses` from another server for a particular `VoteRequest`, it can take the first and ignore the rest. We could also choose to take the last, but taking the first is simpler. A server does not need to worry about persisting its election state for a Pre-Vote response like we currently do for `VoteResponses` because the direct result of the Pre-Vote phase does not elect leaders.

Also, if a `Candidate` is unable to be elected (transition to `Leader`) before its election timeout expires, it will transition back to `Prospective`. This will handle the case if a network partition occurs while the server is in `Candidate` state and prevent unnecessary loss of leadership.

How does this prevent unnecessary leadership loss?

We prevent servers from increasing their epoch prior to establishing they can win an election.

Can Pre-Vote prevent a quorum from electing a leader?

Yes, Pre-Vote needs an additional safeguard to prevent scenarios where eligible leaders cannot be elected.

If a leader is unable to send FETCH responses to [majority - 1] of servers, no new metadata can be committed and we will need a new leader to make progress. We may need the minority of servers which are able to communicate with the leader to grant their vote to prospectives which *can* communicate with a majority of the cluster. Without Pre-Vote, the epoch bump would have forced servers to participate in the election. With Pre-Vote, the minority of servers which are connected to the leader will *not* grant Pre-Vote requests. This is the reason why an additional "Check Quorum" safeguard is needed which is what KAFKA-15489 implements. Check Quorum ensures a leader steps down if it is unable to send FETCH responses to a majority of servers. This will free up all servers to grant their votes to eligible prospectives.

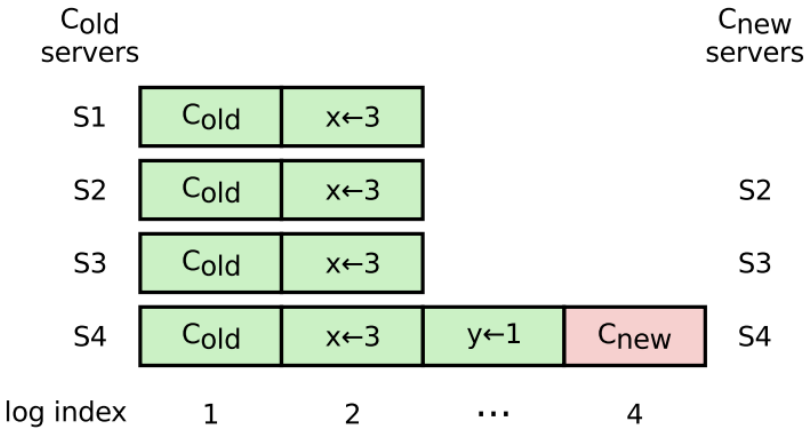
Why do we need Followers to reject Pre-Vote requests? Shouldn't the Pre-Vote and Check Quorum mechanism be enough to prevent disruptive servers?

If servers are still within their fetch timeout, this means they have recently heard from a leader. It can be less disruptive if they refuse to vote for a new leader while still following an existing one.

The following scenarios show why just Pre-Vote and Check Quorum (without Followers rejecting Pre-Votes) are not enough.

- **Scenario A**: We can image a scenario where two servers (S1 & S2) are both up-to-date on the log but unable to maintain a stable connection with each other. Let's say S1 is the leader. When S2 loses connectivity with S1 and is unable to find the leader, it will start a Pre-Vote. Since its log may be up-to-date, a majority of the quorum may grant the Pre-Vote and S2 may then start and win a standard vote to become the new leader. This is already a disruption since S1 was serving the majority of the quorum well as a leader. When S1 loses connectivity with S2 again it will start an election, and this bouncing of leadership could continue.

- **Scenario B**: A server in an old configuration (e.g. S1 in the below diagram, pg 41 of Raft paper) starts a “pre-vote” when the leader is temporarily unavailable, and is elected because it is as up-to-date as the majority of the quorum. We can not technically rely on the original leader replicating fast enough to remove S1 from the quorum - we can imagine some bug/limitation with quorum reconfiguration causes S1 to continuously try to start elections when the leader is trying to remove it from the quorum. This scenario will be covered by KIP-853: KRaft Controller Membership Changes or future work if not covered here.



Compatibility

We currently use ApiVersions to gate new/newer versions of Raft APIs from being used before all servers can support it. This is useful in the upgrade scenario for Pre-Vote - if a server attempts to send out a Pre-Vote request while any other server in the quorum does not understand it, it will get back an UnsupportedVersionException from the network client and knows to default back to the old behavior. Specifically, the server will transition from `Prospecti ve` immediately to `Candidate` state, and will send standard votes instead which can be understood by servers on older software versions.

Let's take a look at an edge case. As the network client will only check the supported version of the peer that we are intending to send a request to, we can imagine a scenario where a server first sends PreVotes to peers which understand PreVote, and then attempts to send PreVote to a peer which does not. If the server receives and processes a majority of granted PreVote responses prior to hitting the UnsupportedVersionException, it can transition to `Candid ate` phase. Otherwise, it will also transition to `Candidate` phase once it hits the exception, and send standard vote requests to all servers. Any PreVote responses received while in `Candidate` phase would be ignored.

Test Plan

This will be tested with unit tests, integration tests, system tests, and TLA+.

Rejected Alternatives

Not adding a new quorum state for Pre-Vote

Adding a new state should keep the logic for existing states closer to their original behavior and prevent overcomplicating them. This could aid in debugging as well since we know definitively that servers in `Prospective` state are sending Pre-Votes, while servers in `Candidate` state are sending standard votes.

Rejecting VoteRequests received within fetch timeout (w/o Pre-Vote)

This was originally proposed in the Raft paper as a necessary safeguard to prevent **Scenario A** from occurring, but we can see how this could extend to cover all the other disruptive scenarios mentioned.

- When a partitioned server rejoins and forces the cluster to participate in an election …
  - if a majority of the cluster is not receiving fetch responses from the current leader, they consider the vote request and make the appropriate state transitions. An election would be needed in this case anyways.
  - if the rest of the cluster is still receiving fetch responses from the current leader, they reject the vote request from the disruptive follower. No one transitions to a new state (e.g. Unattached) as a result of the vote request, current leader is not disrupted.
- For a server in an old configuration (that's not in the new configuration) …
  - if the current leader is still responding to fetch requests in a reasonable amount of time, the server is prevented from starting and winning elections, which would delay reconfiguration.
  - if the current leader is not responding to fetch requests, then the server could still win an election (this scenario calls for an election anyways). KIP-853 should cover preventing this case if necessary.
- For a server w/ new disk/data loss …
  - if the current leader is still responding to fetch requests in a reasonable amount of time, the server is prevented from starting and winning elections, which could lead to loss of committed data.

- if the current leader is not responding to fetch requests, we can reject VoteRequests from the server w/ new disk/data loss if it isn't sufficiently caught up on replication. KIP-853 should cover this case w/ storage ids if necessary.

However, this would not be a good standalone alternative to Pre-Vote because once a server starts a disruptive election (disruptive in the sense that the current leader still has majority), its epoch may increase while none of the other servers' epochs do. The most likely way for the server to rejoin the quorum now with its inflated epoch would be to win an election.

Separate RPC for Pre-Vote

This would be added toil with no real added benefits. Since a Pre-Vote and a standard Vote are similar in concept, it makes sense to cover both with the same RPC. We can add clear logging and metrics to easily differentiate between Pre-Vote and standard Vote requests.

Covering disk loss scenario in scope

This scenario shares similarities with adding new servers to the quorum, which KIP-853: KRaft Controller Membership Changes would handle. If a server loses its disk and fails to fully catch up to the leader prior to another server starting an election, it may vote for any server which is at least as caught up as itself (which might be less than the last leader). One way to handle this is to add logic preventing servers with new disks (determined via a unique storage id) from voting prior to sufficiently catching up on the log. Another way is to reject pre-vote requests from these servers. We leave this scenario to be covered by KIP-853 or future work because of the similarities with adding new servers.

Time	Server 1	Server 2	Server 3
T0	Leader with majority of quorum (Server 1, Server 3) caught up with its committed data	Lagging follower	Follower
T1			Disk failure
T2	Leader Unattached state	Follower Unattached state	Comes back up w/ new disk, triggers an election before catching up on replication
			Will not be elected
T4		Election ms times out and starts an election	
T5		Votes for Server 2	Votes for Server 2
T6		Elected as leader leading to data loss	