KIP-650: Enhance Kafkaesque Raft semantics

Status

Current state: Under Review

Discussion thread: *TBD*

JIRA: TBD

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

Motivation

KIP-595 ships a Kafkaesque Raft algorithm to serve as the foundation of metadata quorum. We migrated and customized the Raft protocol to fit in metadata replication need, but the exploration does not end with just the basic replication and leader election. There are more standard features described in the Raft literature, which are very useful in general production system as well as in Kafka context. In this follow-up KIP, we would like to ship two major features, the pre-vote and non-leader linearizable read.

Proposed Changes

Pre-vote

In the context of Raft, it is very common for leaders to hit network partition. We implemented the progress timeout in KIP-595, so that when a majority of voters are not fetching from the leader, the current leader will start election to reach out to other followers to re-establish the leadership or learn about the new leader. However, if the zombie leader could not connect to the quorum for long enough, it will timeout and start election, while bumping its epoch. Until it eventually reconnects to the quorum, the zombie leader will repeatedly bump its epoch due to election timeout. The consequence is that the zombie leader will gain a much higher epoch than other voters inside the quorum and disrupt the quorum, introducing extra unavailability. During quorum reassignment, the removed voters could also try to start election to ask active leader to step down which affects the ongoing transition. This is called "disruptive server" the Raft literature. Furthermore, the rapid bumping of the epoch increases the chance of epoch overflow, which is very dangerous in the Kafka Raft as we have been using uint32 for epoch due to historical message formats, compared with using uint64.

Fortunately, the Raft literature documents the preventions to avoid such a corner case. The idea is called `pre-vote`, which is that the candidate will try to send out a request to all the known voters and ask whether they would vote for it. If rejected, the candidate will keep retrying to send out pre-vote requests without bumping the epoch, or disturbing the quorum. Overall this reduces the availability drop caused by a zombie node election.

Non-leader Linearizable Read

Metadata log is only written by the leader controller, but there would be three regular readers other than the leader controller:

1. Quorum follower
2. Quorum observer
3. Client: consumer, producer, admin, etc

As we know the leader controller always sees the latest change, where there is always a certain lag between its own log and the followers. If we redirect all the log read request to the leader, it will get overloaded for sure; if we allow reading from secondary copy, like observer reading from follower, or client reading from observer, there is no way to guarantee linearizable result. Consider a case where there is one event for a football game final score, the message gets written to the leader, and successfully replicated to the follower majority, which is then visible to the client. But the second read request being sent to a slow follower may not receive this message yet, so the result will show that the game hasn't stopped yet, which is inconsistent and a violation of linearizability.

blocked URL

Public Interfaces

We plan to a flag to the Vote RPC to indicate whether this request is a pre-vote:

```
{
  "apiKey": 50,
  "type": "request",
  "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": "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"},
              { "name": "PreVote", "type": "boolean", "versions": "1+", // NEW
          "about": "Suggest whether given request is a pre-vote"},
      ]
      }
    ]
    }
  ]
}
```

And the Raft node will respond with either an approval or reject to the pre-vote request in the same way as responding to the normal vote request. Each candidate node will first attempt to send pre-votes to the known quorum members. When it gets the majority of pre-votes approved, the candidate will proceed to the actual vote. If pre-vote fails, the active candidate will not continue to attempt to do the actual vote, instead it will timeout again to start another round of pre-vote candidate, which will not disrupt the live quorum. We may implement a config to constrain number of failed elections for a candidate before self shutting down, but for now human intervention should be good enough.

For linearizable read, a separate RPC called *ReadOffset* shall be implemented for the non-leader node to query the current applied offset on the active leader's state machine for a linearizable query.

```
{
  "apiKey": N,
  "type": "request",
  "name": "ReadOffsetRequest",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "ClusterId", "type": "string", "versions": "0+",
      "nullableVersions": "0+", "default": "null"},
    { "name": "ReadTimestamp", "type": "int64", "versions": "0",
      "about": "The timestamp of the read query."},
    { "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." },
      ]
      }
    ]
    }
  ]
}
```

```
{
  "apiKey": N,
  "type": "response",
  "name": "ReadOffsetResponse",
  "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": "AppliedOffset", "type": "int64", "versions": "0+",
          "about": "The last applied offset on the leader." },
      ]
      }
    ]
    }
  ]
}
```

Once getting the offset read result, follower/observers would buffer those pending read requests until its state machine has already reached the marked applied offset level, to ensure the read consistency on the follower side. If the query was on hold for too long, we rely on client side to timeout and retry another candidate based off its own timeout, instead of implementing a complex lag tracking mechanism for now.

Client side could optionally implement a flag to decide whether to perform stale read or linearizable read to avoid long holding for the result, which could be done in the follow-up work.

Compatibility, Deprecation, and Migration Plan

This is a new feature which does not affect existing component's behavior. Older client should also be compatible.

Rejected Alternatives

N/A

Citation

***Please stop calling databases CP or AP**, Published by Martin Kleppmann on 11 May 2015*