

# KIP-500: Replace ZooKeeper with a Self-Managed Metadata Quorum


- [Status](#)
- [Motivation](#)
  - [Metadata as an Event Log](#)
  - [Simpler Deployment and Configuration](#)
- [Architecture](#)
  - [Introduction](#)
  - [Overview](#)
  - [The Controller Quorum](#)
  - [Broker Metadata Management](#)
  - [The Broker State Machine](#)
  - [Broker States](#)
    - [Offline](#)
    - [Fenced](#)
    - [Online](#)
    - [Stopping](#)
  - [Transitioning some existing APIs to Controller-Only](#)
  - [New Controller APIs](#)
  - [Removing Direct ZooKeeper Access from Tools](#)
- [Compatibility, Deprecation, and Migration Plan](#)
  - [Client Compatibility](#)
  - [Bridge Release](#)
  - [Rolling Upgrade](#)
    - [Upgrade to the Bridge Release](#)
    - [Start the Controller Quorum Nodes](#)
    - [Roll the Broker Nodes](#)
    - [Roll the Controller Quorum](#)
- [Rejected Alternatives](#)
  - [Pluggable Consensus](#)
- [Follow-on Work](#)
- [References](#)

## 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).

## Motivation

Currently, Kafka uses ZooKeeper to store its metadata about partitions and brokers, and to elect a broker to be the Kafka Controller. We would like to remove this dependency on ZooKeeper. This will enable us to manage metadata in a more scalable and robust way, enabling support for more partitions. It will also simplify the deployment and configuration of Kafka.

## Metadata as an Event Log

We often talk about the benefits of managing state as a stream of events. A single number, the offset, describes a consumer's position in the stream. Multiple consumers can quickly catch up to the latest state simply by replaying all the events newer than their current offset. The log establishes a clear ordering between events, and ensures that the consumers always move along a single timeline.

However, although our users enjoy these benefits, Kafka itself has been left out. We treat changes to metadata as isolated changes with no relationship to each other. When the controller pushes out state change notifications (such as `LeaderAndIsrRequest`) to other brokers in the cluster, it is possible for brokers to get some of the changes, but not all. Although the controller retries several times, it eventually give up. This can leave brokers in a divergent state.

Worse still, although ZooKeeper is the store of record, the state in ZooKeeper often doesn't match the state that is held in memory in the controller. For example, when a partition leader changes its ISR in ZK, the controller will typically not learn about these changes for many seconds. There is no generic way for the controller to follow the ZooKeeper event log. Although the controller can set one-shot watches, the number of watches is limited for performance reasons. When a watch triggers, it doesn't tell the controller the current state-- only that the state has changed. By the time the controller re-reads the znode and sets up a new watch, the state may have changed from what it was when the watch originally fired. If there is no watch set, the controller may not learn about the change at all. In some cases, restarting the controller is the only way to resolve the discrepancy.

Rather than being stored in a separate system, metadata should be stored in Kafka itself. This will avoid all the problems associated with discrepancies between the controller state and the ZooKeeper state. Rather than pushing out notifications to brokers, brokers should simply consume metadata events from the event log. This ensures that metadata changes will always arrive in the same order. Brokers will be able to store metadata locally in a file. When they start up, they will only need to read what has changed from the controller, not the full state. This will let us support more partitions with less CPU consumption.

## Simpler Deployment and Configuration

ZooKeeper is a separate system, with its own configuration file syntax, management tools, and deployment patterns. This means that system administrators need to learn how to manage and deploy two separate distributed systems in order to deploy Kafka. This can be a daunting task for administrators, especially if they are not very familiar with deploying Java services. Unifying the system would greatly improve the "day one" experience of running Kafka, and help broaden its adoption.

Because the Kafka and ZooKeeper configurations are separate, it is easy to make mistakes. For example, administrators may set up SASL on Kafka, and incorrectly think that they have secured all of the data travelling over the network. In fact, it is also necessary to configure security in the separate, external ZooKeeper system in order to do this. Unifying the two systems would give a uniform security configuration model.

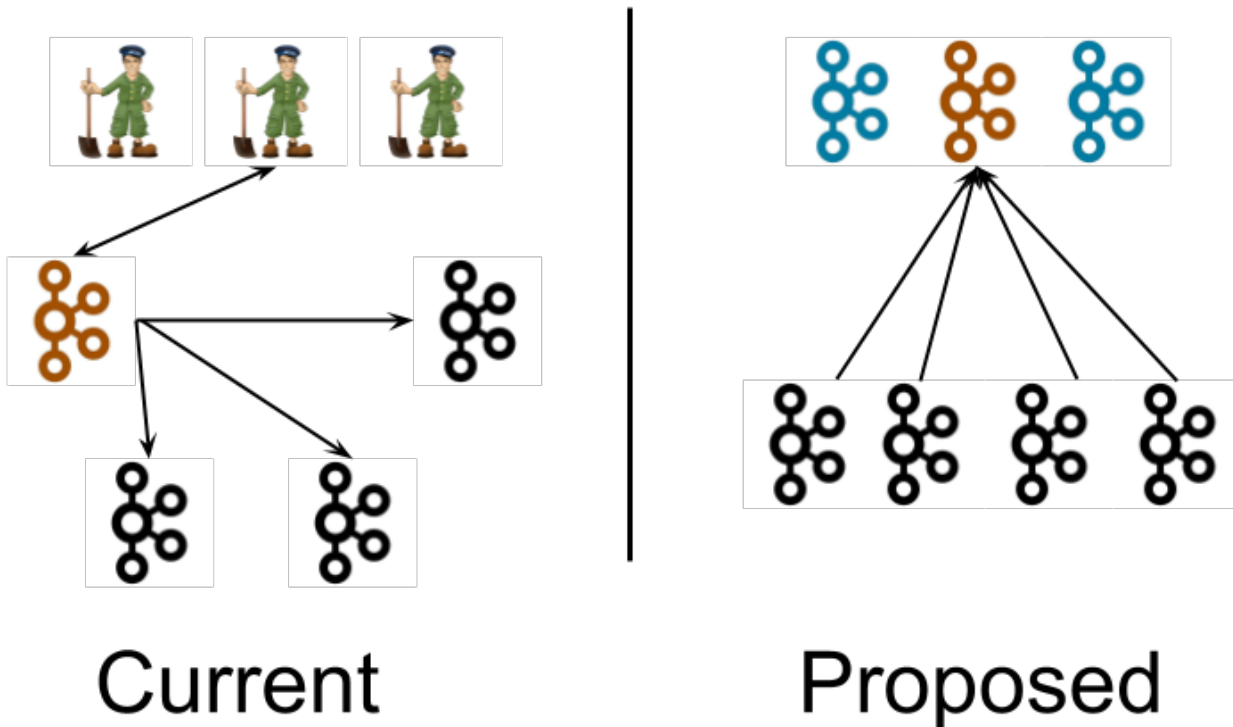
Finally, in the future we may want to support a single-node Kafka mode. This would be useful for people who wanted to quickly test out Kafka without starting multiple daemons. Removing the ZooKeeper dependency makes this possible.

## Architecture

### Introduction

This KIP presents an overall vision for a scalable post-ZooKeeper Kafka. In order to present the big picture, I have mostly left out details like RPC formats, on-disk formats, and so on. We will want to have follow-on KIPs to describe each step in greater detail. This is similar to KIP-4, which presented an overall vision which subsequent KIPs enlarged upon.

### Overview



Currently, a Kafka cluster contains several broker nodes, and an external quorum of ZooKeeper nodes. We have pictured 4 broker nodes and 3 ZooKeeper nodes in this diagram. This is a typical size for a small cluster. The controller (depicted in orange) loads its state from the ZooKeeper quorum after it is elected. The lines extending from the controller to the other nodes in the broker represent the updates which the controller pushes, such as `LeaderAndIsr` and `UpdateMetadata` messages.

Note that this diagram is slightly misleading. Other brokers besides the controller can and do communicate with ZooKeeper. So really, a line should be drawn from each broker to ZK. However, drawing that many lines would make the diagram difficult to read. Another issue which this diagram leaves out is that external command line tools and utilities can modify the state in ZooKeeper, without the involvement of the controller. As discussed earlier, these issues make it difficult to know whether the state in memory on the controller truly reflects the persistent state in ZooKeeper.

In the proposed architecture, three controller nodes substitute for the three ZooKeeper nodes. The controller nodes and the broker nodes run in separate JVMs. The controller nodes elect a single leader for the metadata partition, shown in orange. Instead of the controller pushing out updates to the brokers, the brokers pull metadata updates from this leader. That is why the arrows point towards the controller rather than away.

Note that although the controller processes are logically separate from the broker processes, they need not be physically separate. In some cases, it may make sense to deploy some or all of the controller processes on the same node as the broker processes. This is similar to how ZooKeeper processes may be deployed on the same nodes as Kafka brokers today in smaller clusters. As per usual, all sorts of deployment options are possible, including running in the same JVM.

## The Controller Quorum

The controller nodes comprise a Raft quorum which manages the metadata log. This log contains information about each change to the cluster metadata. Everything that is currently stored in ZooKeeper, such as topics, partitions, ISRs, configurations, and so on, will be stored in this log.

Using the Raft algorithm, the controller nodes will elect a leader from amongst themselves, without relying on any external system. The leader of the metadata log is called the active controller. The active controller handles all RPCs made from the brokers. The follower controllers replicate the data which is written to the active controller, and serve as hot standbys if the active controller should fail. Because the controllers will now all track the latest state, controller failover will not require a lengthy reloading period where we transfer all the state to the new controller.

Just like ZooKeeper, Raft requires a majority of nodes to be running in order to continue running. Therefore, a three-node controller cluster can survive one failure. A five-node controller cluster can survive two failures, and so on.

Periodically, the controllers will write out a snapshot of the metadata to disk. While this is conceptually similar to compaction, the code path will be a bit different because we can simply read the state from memory rather than re-reading the log from disk.

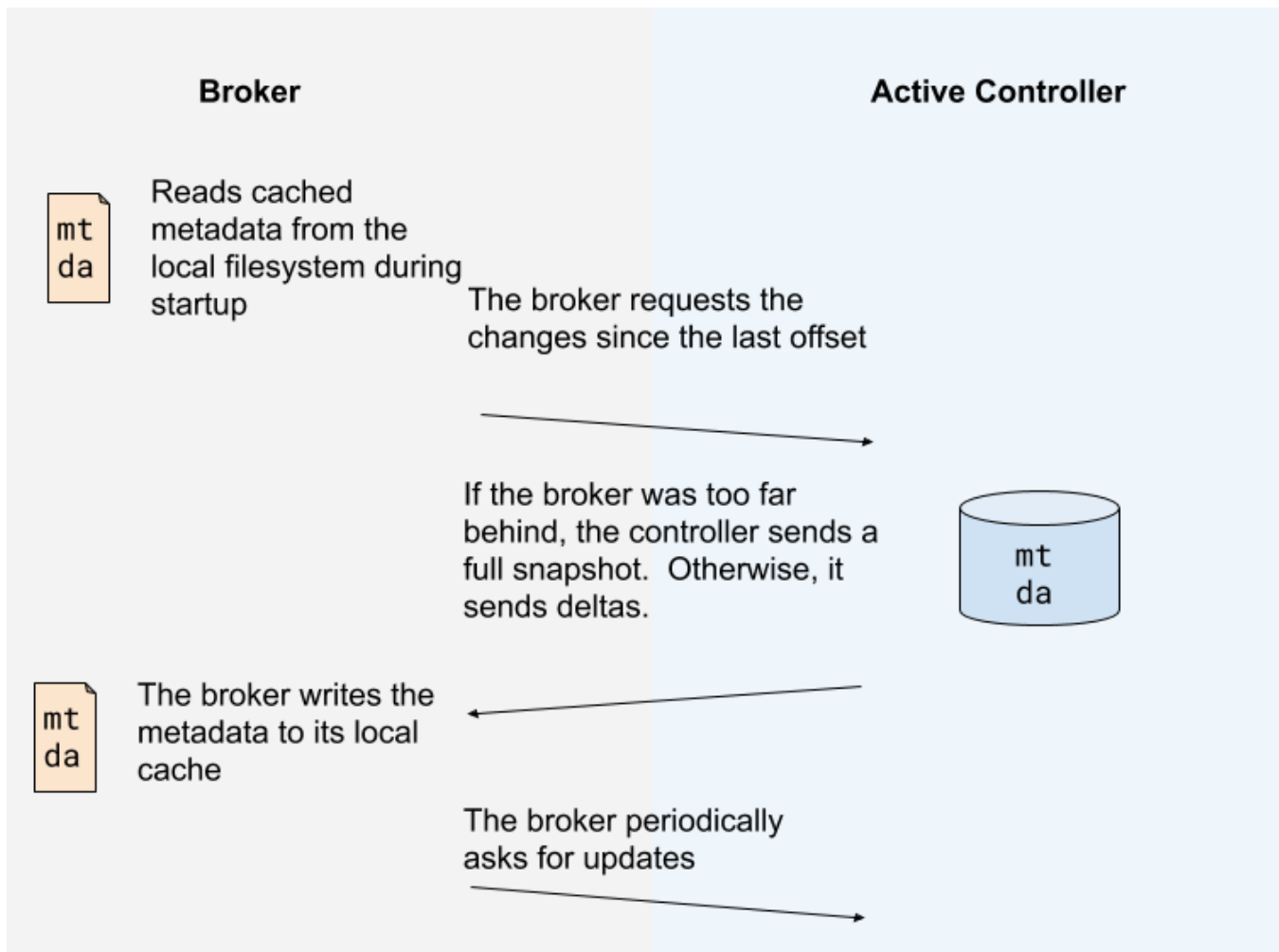
## Broker Metadata Management

Instead of the controller pushing out updates to the other brokers, those brokers will fetch updates from the active controller via the new `MetadataFetch` API.

A `MetadataFetch` is similar to a fetch request. Just like with a fetch request, the broker will track the offset of the last updates it fetched, and only request newer updates from the active controller.

The broker will persist the metadata it fetched to disk. This will allow the broker to start up very quickly, even if there are hundreds of thousands or even millions of partitions. (Note that since this persistence is an optimization, we can leave it out of the first version, if it makes development easier.)

Most of the time, the broker should only need to fetch the deltas, not the full state. However, if the broker is too far behind the active controller, or if the broker has no cached metadata at all, the controller will send a full metadata image rather than a series of deltas.



The broker will periodically ask for metadata updates from the active controller. This request will double as a heartbeat, letting the controller know that the broker is alive.

Note that while this KIP only discusses broker metadata management, client metadata management is important for scalability as well. Once the infrastructure for sending incremental metadata updates exists, we will want to use it for clients as well as for brokers. After all, there are typically a lot more clients than brokers. As the number of partitions grows, it will become more and more important to deliver metadata updates incrementally to clients that are interested in many partitions. We will discuss this further in follow-on KIPs.

## The Broker State Machine

Currently, brokers register themselves with ZooKeeper right after they start up. This registration accomplishes two things: it lets the broker know whether it has been elected as the controller, and it lets other nodes know how to contact it.

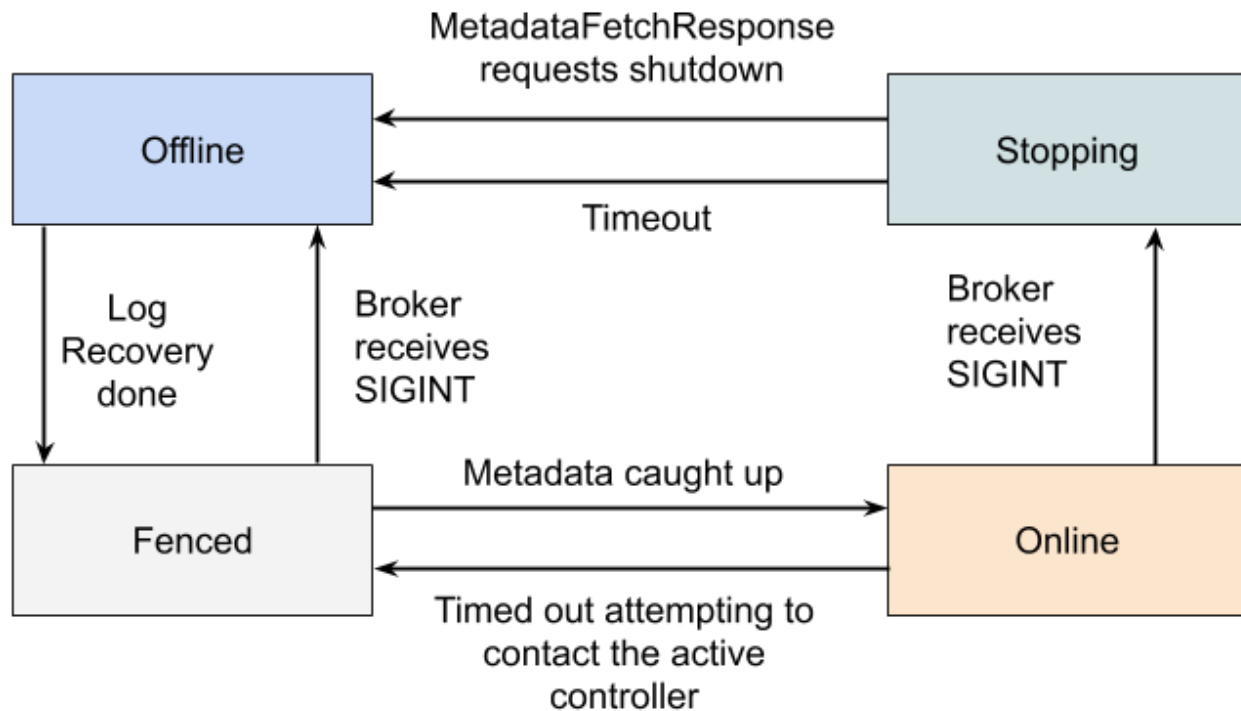
In the post-ZooKeeper world, brokers will register themselves with the controller quorum, rather than with ZooKeeper.

Currently, if a broker loses its ZooKeeper session, the controller removes it from the cluster metadata. In the post-ZooKeeper world, the active controller removes a broker from the cluster metadata if it has not sent a `MetadataFetch` heartbeat in a long enough time.

In the current world, a broker which can contact ZooKeeper but which is partitioned from the controller will continue serving user requests, but will not receive any metadata updates. This can lead to some confusing and difficult situations. For example, a producer using `acks=1` might continue to produce to a leader that actually was not the leader any more, but which failed to receive the controller's `LeaderAndIsrRequest` moving the leadership.

In the post-ZK world, cluster membership is integrated with metadata updates. Brokers cannot continue to be members of the cluster if they cannot receive metadata updates. While it is still possible for a broker to be partitioned from a particular client, the broker will be removed from the cluster if it is partitioned from the controller.

## Broker States



## Offline

When the broker process is in the Offline state, it is either not running at all, or in the process of performing single-node tasks needed to starting up such as initializing the JVM or performing log recovery.

## Fenced

When the broker is in the Fenced state, it will not respond to RPCs from clients. The broker will be in the fenced state when starting up and attempting to fetch the newest metadata. It will re-enter the fenced state if it can't contact the active controller. Fenced brokers should be omitted from the metadata sent to clients.

## Online

When a broker is online, it is ready to respond to requests from clients.

## Stopping

Brokers enter the stopping state when they receive a SIGINT. This indicates that the system administrator wants to shut down the broker.

When a broker is stopping, it is still running, but we are trying to migrate the partition leaders off of the broker.

Eventually, the active controller will ask the broker to finally go offline, by returning a special result code in the MetadataFetchResponse. Alternately, the broker will shut down if the leaders can't be moved in a predetermined amount of time.

## Transitioning some existing APIs to Controller-Only

Many operations that were formerly performed by a direct write to ZooKeeper will become controller operations instead. For example, changing configurations, altering ACLs that are stored with the default Authorizer, and so on.

New versions of the clients should send these operations directly to the active controller. This is a backwards compatible change: it will work with both old and new clusters. In order to preserve compatibility with old clients that sent these operations to a random broker, the brokers will forward these requests to the active controller.

## New Controller APIs

In some cases, we will need to create a new API to replace an operation that was formerly done via ZooKeeper. One example of this is that when the leader of a partition wants to modify the in-sync replica set, it currently modifies ZooKeeper directly. In the post-ZK world, the leader will make an RPC to the active controller instead.

## Removing Direct ZooKeeper Access from Tools

Currently, some tools and scripts directly contact ZooKeeper. In a post-ZooKeeper world, these tools must use Kafka APIs instead. Fortunately, "KIP-4: Command line and centralized administrative operations" began the task of removing direct ZooKeeper access several years ago, and it is nearly complete.

## Compatibility, Deprecation, and Migration Plan

### Client Compatibility

We will preserve compatibility with the existing Kafka clients. In some cases, the existing clients will take a less efficient code path. For example, the brokers may need to forward their requests to the active controller.

### Bridge Release

The overall plan for compatibility is to create a "bridge release" of Kafka where the ZooKeeper dependency is well-isolated.

While this release will not remove ZooKeeper, it will eliminate most of the touch points where the rest of the system communicates with it. As much as possible, we will perform all access to ZooKeeper in the controller, rather than in other brokers, clients, or tools. Therefore, although ZooKeeper will still be required for the bridge release, it will be a well-isolated dependency.

We will be able to upgrade from any version of Kafka to this bridge release, and from the bridge release to a post-ZK release. When upgrading from an earlier release to a post-ZK release, the upgrade must be done in two steps: first, you must upgrade to the bridge release, and then you must upgrade to the post-ZK release.

### Rolling Upgrade

The rolling upgrade from the bridge release will take several steps.

### Upgrade to the Bridge Release

The cluster must be upgraded to the bridge release, if it isn't already.

### Start the Controller Quorum Nodes

We will configure the controller quorum nodes with the address of the ZooKeeper quorum. Once the controller quorum is established, the active controller will enter its node information into `/brokers/ids` and overwrite the `/controller` node with its ID. This will prevent any of the un-upgraded broker nodes from becoming the controller at any future point during the rolling upgrade.

Once it has taken over the `/controller` node, the active controller will proceed to load the full state of ZooKeeper. It will write out this information to the quorum's metadata storage. After this point, the metadata quorum will be the metadata store of record, rather than the data in ZooKeeper.

We do not need to worry about the ZooKeeper state getting concurrently modified during this loading process. In the bridge release, neither the tools nor the non-controller brokers will modify ZooKeeper.

The new active controller will monitor ZooKeeper for legacy broker node registrations. It will know how to send the legacy "push" metadata requests to those nodes, during the transition period.

### Roll the Broker Nodes

We will roll the broker nodes as usual. The new broker nodes will not contact ZooKeeper. If the configuration for the zookeeper server addresses is left in the configuration, it will be ignored.

### Roll the Controller Quorum

Once the last broker node has been rolled, there will be no more need for ZooKeeper. We will remove it from the configuration of the controller quorum nodes, and then roll the controller quorum to fully remove it.

## Rejected Alternatives

### Pluggable Consensus

Rather than managing metadata ourselves, we could make the metadata storage layer pluggable so that it could work with systems other than ZooKeeper. For example, we could make it possible to store metadata in etcd, Consul, or similar systems.

Unfortunately, this strategy would not address either of the two main goals of ZooKeeper removal. Because they have ZooKeeper-like APIs and design goals, these external systems would not let us treat metadata as an event log. Because they are still external systems that are not integrated with the project, deployment and configuration would still remain more complex than they needed to be.

Supporting multiple metadata storage options would inevitably decrease the amount of testing we could give to each configuration. Our system tests would have to either run with every possible configuration storage mechanism, which would greatly increase the resources needed, or choose to leave some user under-tested. Increasing the size of test matrix in this fashion would really hurt the project.

Additionally, if we supported multiple metadata storage options, we would have to use "least common denominator" APIs. In other words, we could not use any API unless all possible metadata storage options supported it. In practice, this would make it difficult to optimize the system.

## Follow-on Work

This KIP expresses a vision of how we would like to evolve Kafka in the future. We will create follow-on KIPs to hash out the concrete details of each change.

- [KIP-455: Create an Administrative API for Replica Reassignment](#)
- [KIP-497: Add inter-broker API to alter ISR](#)
- [KIP-543: Expand ConfigCommand's non-ZK functionality](#)
- [KIP-555: Deprecate Direct Zookeeper access in Kafka Administrative Tools](#)
- [KIP-589 Add API to update Replica state in Controller](#)
- [KIP-590: Redirect Zookeeper Mutation Protocols to The Controller](#)
- [KIP-595: A Raft Protocol for the Metadata Quorum](#)
- [KIP-631: The Quorum-based Kafka Controller](#)

## References

The Raft consensus algorithm

- Ongaro, D., Ousterhout, J. [In Search of an Understandable Consensus Algorithm](#)

Handling Metadata via Write-Ahead Logging

- Shvachko, K., Kuang, H., Radia, S. Chansler, R. [The Hadoop Distributed Filesystem](#)
- Balakrishnan, M., Malkhi, D., Wobber, T. [Tango: Distributed Data Structures over a Shared Log](#)