O'REILLY®

ZooKeeper

DISTRIBUTED PROCESS COORDINATION

Flavio Junqueira & Benjamin Reed

O'REILLY®

ZooKeeper

Building distributed applications is difficult enough without having to coordinate the actions that make them work. This practical guide shows how Apache ZooKeeper helps you manage distributed systems, so you can focus mainly on application logic. Even *with* ZooKeeper, implementing coordination tasks is not trivial, but this book provides good practices to give you a head start, and points out caveats that developers and administrators alike need to watch for along the way.

In three separate sections, ZooKeeper contributors Flavio Junqueira and Benjamin Reed introduce the principles of distributed systems, provide ZooKeeper programming techniques, and include the information you need to administer this service.

- Learn how ZooKeeper solves common coordination tasks
- Explore the ZooKeeper API's Java and C implementations and how they differ
- Use methods to track and react to ZooKeeper state changes
- Handle failures of the network, application processes, and ZooKeeper itself
- Learn about ZooKeeper's trickier aspects dealing with concurrency, ordering, and configuration
- Use the Curator high-level interface for connection management
- Become familiar with ZooKeeper internals and administration tools

Flavio Junqueira is a member of the research staff of Microsoft Research in Cambridge, UK. He is an active contributor to Apache projects such as ZooKeeper and BookKeeper.

Benjamin Reed is a software engineer at Facebook. He helped start the Pig, ZooKeeper, and BookKeeper projects hosted by the Apache Software Foundation.

DATABASES/WEB

US $29.99 CAN $31.99
ISBN: 978-1-449-36130-3

Leader Elections

The leader is a server that has been chosen by an ensemble of servers and that continues to have support from that ensemble. The purpose of the leader is to order client requests that change the ZooKeeper state: `create`, `setData`, and `delete`. The leader transforms each request into a transaction, as explained in the previous section, and proposes to the followers that the ensemble accepts and applies them in the order issued by the leader.

To exercise leadership, a server must have support from a quorum of servers. As we discussed in Chapter 2, quorums must intersect to avoid the problem that we call *split brain*: two subsets of servers making progress independently. This situation leads to inconsistent system state, and clients end up getting different results depending on which server they happen to contact. We gave a concrete example of this situation in "ZooKeeper Quorums" on page 24.

The groups that elect and support a leader must intersect on at least one server process. We use the term *quorum* to denote such subsets of processes. Quorums pairwise intersect.

Progress

Because a quorum of servers is necessary for progress, ZooKeeper cannot make progress in the case that enough servers have permanently failed that no quorum can be formed. It is OK if servers are brought down and eventually boot up again, but for progress to be made, a quorum must eventually boot up. We relax this constraint when we discuss the possibility of reconfiguring ensembles in the next chapter. Reconfiguration can change quorums over time.

Each server starts in the `LOOKING` state, where it must either elect a new leader or find the existing one. If a leader already exists, other servers inform the new one which server is the leader. At this point, the new server connects to the leader and makes sure that its own state is consistent with the state of the leader.

If an ensemble of servers, however, are all in the `LOOKING` state, they must communicate to elect a leader. They exchange messages to converge on a common choice for the leader. The server that wins this election enters the `LEADING` state, while the other servers in the ensemble enter the `FOLLOWING` state.

The leader election messages are called *leader election notifications*, or simply *notifications*. The protocol is extremely simple. When a server enters the `LOOKING` state, it sends a batch of notification messages, one to each of the other servers in the ensemble. The message contains its current *vote*, which consists of the server's identifier (*sid*) and the zxid (*zxid*) of the most recent transaction it executed. Thus, `(1,5)` is a vote sent by the server with a sid of 1 and a most recent zxid of 5. (For the purposes of leader election, a zxid is a single number, but in some other protocols it is represented as an epoch and a counter.)

Upon receiving a vote, a server changes its vote according to the following rules:

1. Let `voteId` and `voteZxid` be the identifier and the zxid in the current vote of the receiver, whereas `myZxid` and `mySid` are the values of the receiver itself.
2. If (`voteZxid` > `myZxid`) or (`voteZxid` = `myZxid` and `voteId` > `mySid`), keep the current vote.
3. Otherwise, change my vote by assigning `myZxid` to `voteZxid` and `mySid` to `voteZxid`.

In short, the server that is most up to date wins, because it has the most recent zxid. We'll see later that this simplifies the process of restarting a quorum when a leader dies. If multiple servers have the most recent zxid, the one with the highest sid wins.

Once a server receives the same vote from a quorum of servers, the server declares the leader elected. If the elected leader is the server itself, it starts executing the leader role. Otherwise, it becomes a follower and tries to connect to the elected leader. Note that it is not guaranteed that the follower will be able to connect to the elected leader. The elected leader might have crashed, for example. Once it connects, the follower and the leader sync their state, and only after syncing can the follower start processing new requests.

Looking for a Leader

The Java class in ZooKeeper that implements an election is `QuorumPeer`. Its `run` method implements the main loop of the server. When in the `LOOKING` state, it executes `lookForLeader` to elect a leader. This method basically executes the protocol we have just discussed. Before returning, the method sets the state of the server to either `LEADING` or `FOLLOWING`. `OBSERVING` is also an option that will be discussed later. If the server is leading, it creates a new `Leader` and runs it. If it is following, it creates a new `Follower` and runs it.

Let's go over an example of an execution of this protocol. Figure 9-1 shows three servers, each starting with a different initial vote corresponding to the server identifier and the last zxid of the server. Each server receives the votes of the other two, and after the first round, servers s_2 and s_3 change their votes to `(1,6)`. Servers s_2 and s_3 send a new batch of notifications after changing their votes, and after receiving these new notifications, each server has notifications from a quorum with the same vote. They consequently elect server s_1 to be the leader.

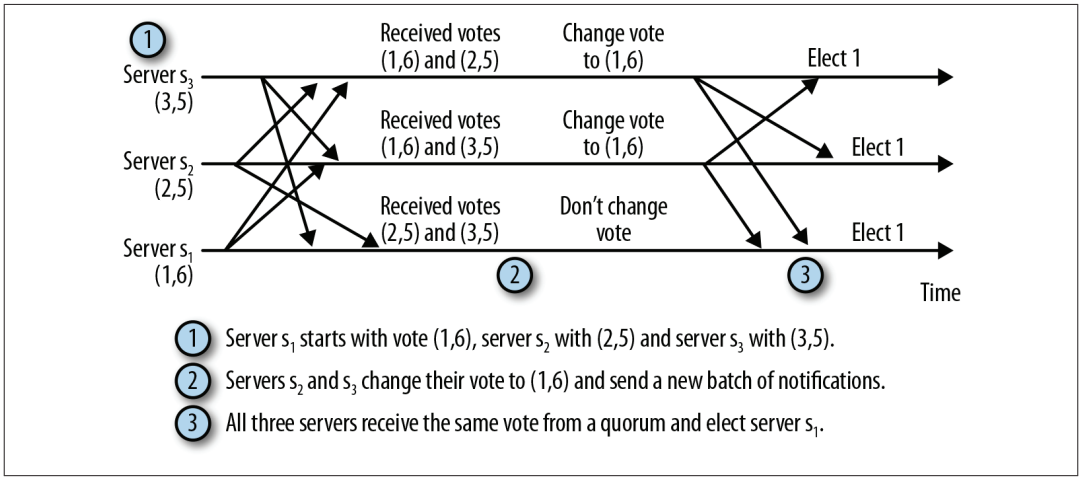

Figure 9-1. Example of a leader election execution

Not all executions are as well behaved as the one in Figure 9-1. In Figure 9-2, we show an example in which s_2 makes an early decision and elects a different leader from servers s_1 and s_3. This happens because the network happens to introduce a long delay in delivering the message from s_1 to s_2 that shows that s_1 has the higher zxid. In the meantime, s_2 elects s_3. In consequence, s_1 and s_3 will form a quorum, leaving out s_2.

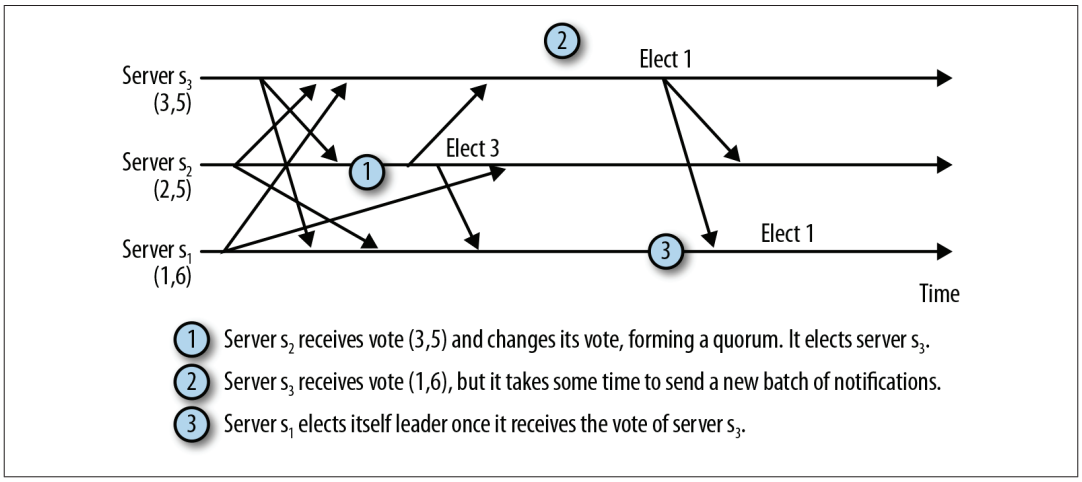

Figure 9-2. Interleaving of messages causes a server to elect a different leader

Having s_2 elect a different leader does not cause the service to behave incorrectly, because s_3 will not respond to s_2 as leader. Eventually s_2 will time out trying to get a response from its elected leader, s_3, and try again. Trying again, however, means that during this time s_2 will not be available to process client requests, which is undesirable.

One simple observation from this example is that if s_2 had waited a bit longer to elect a leader, it would have made the right choice. We show this situation in Figure 9-3. It is hard to know how much time a server should wait, though. The current implementation of `FastLeaderElection`, the default leader election implementation, uses a fixed value of 200 ms (see the constant `finalizeWait`). This value is longer than the expected message delay in modern data centers (less than a millisecond to a few milliseconds), but not long enough to make a substantial difference to recovery time. In case this delay (or any other chosen delay) is not sufficiently long, one or more servers will end up falsely electing a leader that does not have enough followers, so the servers will have to go back to leader election. Falsely electing a leader might make the overall recovery time longer because servers will connect and sync unnecessarily, and still need to send more messages to elect another leader.

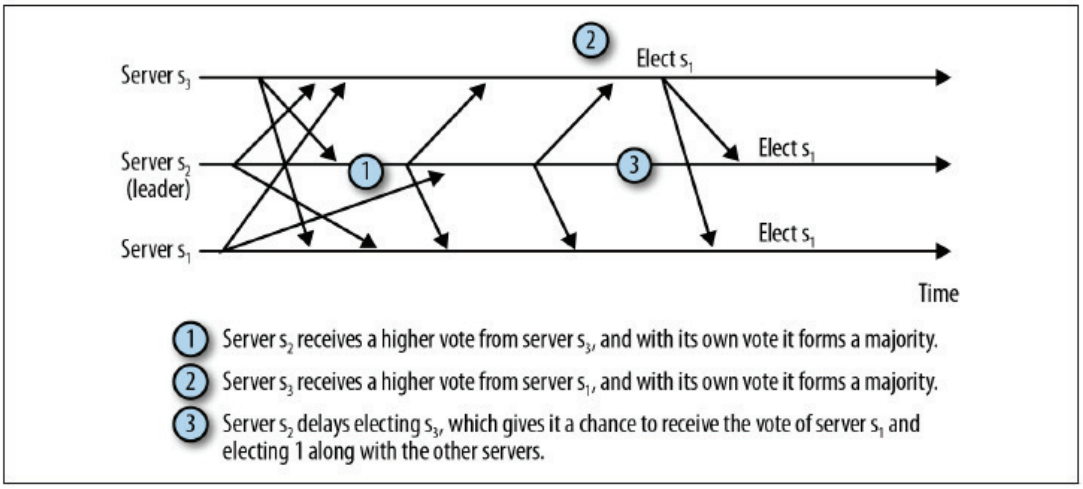

Figure 9-3. Longer delay in electing a leader

What's Fast about Fast Leader Election?

If you are wondering about it, we call the current default leader election algorithm *fast* for historical reasons. The initial leader election algorithm implemented a pull-based model, and the interval for a server to pull votes was about 1 second. This approach added some delay to recovery. With the current implementation, we are able to elect a leader faster.

To implement a new leader election algorithm, we need to implement the `Election` interface in the *quorum* package. To enable users to choose among the leader election implementations available, the code uses simple integer identifiers (see `QuorumPeer.createElectionAlgorithm()`). The other two implementations available currently are `LeaderElection` and `AuthFastLeaderElection`, but they have been deprecated as of release 3.4.0, so in some future releases you may not even find them.