ENGINEERING BLOG

ZOOKEEPER

admin July 23, 2019 Leave a comment Edit

Apache ZooKeeper is an effort to develop and maintain an open-source server which enables highly reliable distributed coordination.

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications. Each time they are implemented there is a lot of work that goes into fixing the bugs and race conditions that are inevitable. Because of the difficulty of implementing these kinds of services, applications initially usually skimp on them, which make them brittle in the presence of change and difficult to manage. Even when done correctly, different implementations of these services lead to management complexity when the applications are deployed.

OVERVIEW:

ZooKeeper is a **distributed, open-source coordination service** for distributed applications. It exposes a simple set of primitives that distributed applications can build upon to implement higher level services for synchronization, configuration, maintenance, and groups and naming. It is designed to be easy to program, and uses a data model styled after the familiar directory tree structure of file systems. It runs in Java and has bindings for both Java and C.

Coordination services are notoriously hard to get right. They are especially prone to errors such as race conditions and deadlock. The motivation behind ZooKeeper is to relieve distributed applications the responsibility of implementing coordination services from scratch.

DESIGN GOALS :

1. **Zookeeper is simple** : Allows distributed processes to coordinate with each other through a shared namespace which is organized similarly to a standard file system.

The name space consists of data registers – called znodes, in ZooKeeper parlance – and these are similar to files and directories. Unlike a typical file system, which is designed for storage, ZooKeeper data is kept in-memory, which means ZooKeeper can achieve high throughput and low latency numbers. The ZooKeeper implementation puts a premium on high performance, highly available, strictly ordered access.

The performance aspects means it can be used in large, distributed systems.

The reliability aspects keep it from being a single point of failure.

The strict ordering means that sophisticated synchronization primitives can be implemented at the client.

2. **ZooKeeper is replicated** : Like the distributed processes it coordinates, ZooKeeper itself is intended to be replicated over a set of hosts called an ensemble. The servers that make up the ZooKeeper service must all know about each other. They maintain an in-memory image of state, along with a transaction logs and snapshots in a persistent store. As long as a majority of the servers are available, the ZooKeeper service will be available. Clients connect to a single ZooKeeper server. The client maintains a TCP connection through which it sends requests, gets responses, gets watch events, and sends heart beats. If the TCP connection to the server breaks, the client will connect to a different server.
3. **ZooKeeper is ordered** : ZooKeeper stamps each update with a number that reflects the order of all ZooKeeper transactions. Subsequent operations can use the order to implement higher-level abstractions, such as synchronization primitives.
4. **ZooKeeper is fast** : It is especially fast in “read-dominant” workloads. ZooKeeper applications run on thousands of machines, and it performs best where reads are more common than writes, at ratios of around 10:1.

DATA MODEL AND HIERARCHICAL NAMESPACE :

Nodes and ephemeral nodes : Unlike is standard file systems, each node in a ZooKeeper namespace can have data associated with it as well as children. It is like having a file-system that allows a file to also be a directory. We use the term znode to make it clear that we are talking about ZooKeeper data nodes.

Znodes maintain a stat structure that includes version numbers for data changes, ACL changes, and timestamps, to allow cache validations and coordinated updates. Each time a znode's data changes, the version number increases. For instance, whenever a client retrieves data it also receives the version of the data. The data stored at each znode in a namespace is read and written atomically. Reads get all the data bytes associated with a znode and a write replaces all the data. Each node has an Access Control List (ACL) that restricts who can do what.

ZooKeeper also has the notion of ephemeral nodes. These znodes exists as long as the session that created the znode is active. When the session ends the znode is deleted.

Conditional updates and watches : ZooKeeper supports the concept of watches. Clients can set a watch on a znodes. A watch will be triggered and removed when the znode changes. When a watch is triggered the client receives a packet saying that the znode has changed. And if the connection between the client and one of the ZooKeeper servers is broken, the client will receive a local notification.

Guarantees : [Sequential consistency, Atomicity, Single system image, Reliability, Timeliness]

IMPLEMENTATION :

1. The replicated database is an in-memory database containing the entire data tree. Updates are logged to disk for recoverability, and writes are serialized to disk before they are applied to the in-memory database.
2. Every ZooKeeper server services clients. Clients connect to exactly one server to submit requests. Read requests are serviced from the local replica of each server database. Requests that change the state of the service, write requests, are processed by an agreement protocol.
3. As part of the agreement protocol all write requests from clients are forwarded to a single server, called the leader. The rest of the ZooKeeper servers, called followers, receive message proposals from the leader and agree upon message delivery. The messaging layer takes care of replacing leaders on failures and syncing followers with leaders.
4. ZooKeeper uses a custom atomic messaging protocol. Since the messaging layer is atomic, ZooKeeper can guarantee that the local replicas never diverge. When the

leader receives a write request, it calculates what the state of the system is when the write is to be applied and transforms this into a transaction that captures this new state.

It is especially high performance in applications where reads outnumber writes, since writes involve synchronizing the state of all servers.

REPLICATED ZOOKEEPER :

But in production, you should run ZooKeeper in replicated mode. A replicated group of servers in the same application is called a quorum, and in replicated mode, all servers in the quorum have copies of the same configuration file.

[For replicated mode, a minimum of three servers are required, and it is strongly recommended that you have an odd number of servers. If you only have two servers, then you are in a situation where if one of them fails, there are not enough machines to form a **majority quorum**. Two servers is inherently less stable than a single server, because there are two single points of failure.]

```
tickTime=2000
dataDir=/var/lib/zookeeper
clientPort=2181
initLimit=5
syncLimit=2
server.1=zoo1:2888:3888
server.2=zoo2:2888:3888
server.3=zoo3:2888:3888
```

initLimit is timeouts ZooKeeper uses to limit the length of time the ZooKeeper servers in quorum have to connect to a leader. **syncLimit** limits how far out of date a server can be from a leader. [**tickTime** It is used to do heartbeats and the minimum session timeout will be twice the value.]

Peers use the former port to connect to other peers. Such a connection is necessary so that peers can communicate, for example, to agree upon the order of updates. More specifically, a ZooKeeper server uses this port to connect followers to the leader. When a new leader arises, a follower opens a TCP connection to the leader using this port. Because the default leader election also uses TCP, we currently require another port for leader election. This is the second port in the server entry.

ZOOKEEPER RECIPES AND SOLUTIONS :

One of the most interesting things about ZooKeeper is that even though ZooKeeper uses asynchronous notifications, you can use it to build synchronous consistency primitives, such as queues and locks. As you will see, this is possible because ZooKeeper imposes an overall order on updates, and has mechanisms to expose this ordering.

Note that the recipes below attempt to employ best practices. In particular, they avoid polling, timers or anything else that would result in a “herd effect”, causing bursts of traffic and limiting scalability.

There are many useful functions that can be imagined that aren’t included here – revocable read-write priority locks, as just one example. And some of the constructs mentioned here – locks, in particular – illustrate certain points, even though you may find other constructs, such as event handlers or queues, a more practical means of performing the same function. In general, the examples in this section are designed to stimulate thought.

Services directly provided by ZooKeeper : [Name service, Configuration, Group membership]

Group Membership : The group is represented by a node. Members of the group create ephemeral nodes under the group node. Nodes of the members that fail abnormally will be removed automatically when ZooKeeper detects the failure.

BARRIERS :

Distributed systems use barriers to block processing of a set of nodes until a condition is met at which all the nodes are allowed to proceed.

—> [True] Client waits —> [Watch triggered] Client proceeds

CLIENT ——> ZOOKEEPER (exists)

—> [False] barrier is gone and client proceeds

Double barriers : enables clients to **synchronize** the **beginning** and the **end** of a computation.

TWO PHASE COMMIT :

A two-phase commit protocol is an algorithm that lets all clients in a distributed system agree either to commit a transaction or abort.

INTERNALS :

ZAB : ZOOKEEPER ATOMIC BROADCAST PROTOCOL : Zab is the ZooKeeper Atomic Broadcast protocol. We use it to propagate state changes produced by the ZooKeeper leader.

Guarantees, Properties and Definitions :

Reliable Delivery : If a message, m, is delivered by one server, it will be eventually delivered by all servers.

Total Order : If a message is delivered before message b by one server, a will be delivered before b by all servers. If a and b are delivered messages, either a will be delivered before b or b will be delivered before a.

Causal Order : If a message b is sent after a message a has been delivered by the sender of b, a must be ordered before b. If a sender sends c after sending b, c must be ordered after b.

FROM BOOK :

Distributed system : a system comprised of multiple software components running independently and concurrently across multiple physical machines.

Processes in a distributed system have two broad options for communication: they can exchange messages directly through a network, or read and write to some shared storage. ZooKeeper uses the shared storage model to let applications implement coordination and synchronization primitives. But shared storage itself requires network communication between the processes and the storage.

Issues to be looked upon :

1. Message delays
2. Processor speed
3. Clock drift

Master – Worker Architecture :

In general, in such an architecture a master process is responsible for keeping track of the workers and tasks available, and for assigning tasks to workers.

To implement three key problems to be solved :

- Master crashes
    - Fetch the state of crashed master
    - Avoid split-brain
- Worker crashes
    - Ability of master to detect worker crashes
    - Recovery procedure for side effects of partial processing
- Communication failures
    - Impact on synchronisation primitives

Zookeeper takes care of this by

1. Enables clients to say some data is ephemeral.
2. Clients notify ensemble periodically that they are alive.

Requirements for master-worker architecture :

1. Master election: It is critical for progress to have a master available to assign tasks to workers.
2. Crash detection: The master must be able to detect when workers crash or disconnect.

3. Group membership management: The master must be able to figure out which workers are available to execute tasks.
4. Metadata management: The master and the workers must be able to store assignments and execution statuses in a reliable manner.

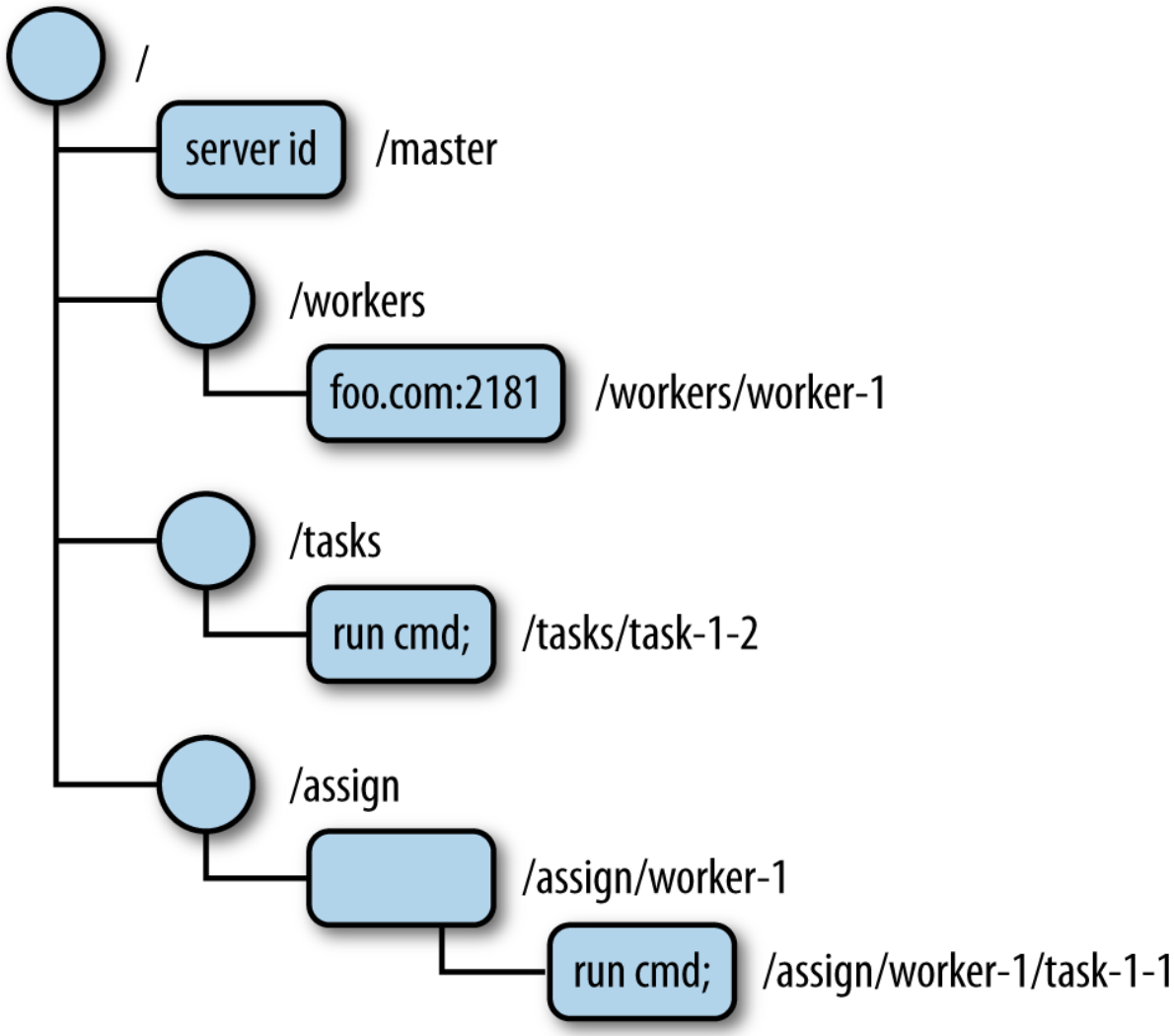
The truly difficult problems you will encounter as you develop distributed applications have to do with faults—specifically, crashes and communication faults. To handle faulty processes, the processes that are still running must be able to detect the failure; messages may be lost, and there may even be clock drift.

Ideally, we design our systems under the assumption that communication is asynchronous: the machines we use may experience clock drift and may experience communication failures. A similar result known as CAP, which stands for Consistency, Availability, and Partition-tolerance, says that when designing a distributed system we may want all three of those properties, but that no system can handle all three.

ZooKeeper has been designed with mostly **consistency and availability** in mind, although it also provides read-only capability in the presence of network partitions. Paxos and virtual synchrony have been particularly influential in the design of ZooKeeper.

Several primitives used for coordination are commonly shared across many applications. Consequently, one way of designing a service used for coordination is to come up with a list of primitives, expose calls to create instances of each primitive, and manipulate these instances directly. ZooKeeper does not expose primitives directly. Instead, it exposes a file system-like API comprised of a small set of calls that enables applications to implement their own primitives.

We typically use recipes to denote these implementations of primitives. Recipes include ZooKeeper operations that manipulate small data nodes, called znodes, that are organized hierarchically as a tree, just like in a file system.



The absence of data often conveys important information about a znode. In a master–worker example, for instance, the absence of a master znode means that no master is currently elected.

1. The /workers znode is the parent znode to all znodes representing a worker available in the system. shows that one worker (foo.com:2181) is available. If a worker becomes unavailable, its znode should be removed from /workers.
2. The /tasks znode is the parent of all tasks created and waiting for workers to execute them. Clients of the master–worker application add new znodes as children of /tasks to represent new tasks and wait for znodes representing the status of the task.
3. The /assign znode is the parent of all znodes representing an assignment of a task to a worker. When a master assigns a task to a worker, it adds a child znode to

/assign.

One important note is that ZooKeeper does not allow partial writes or reads of the znode data. When setting the data of a znode or reading it, the content of the znode is replaced or read entirely.ZooKeeper clients connect to a ZooKeeper service and establish a session through which they make API calls.

API calls provided by Zookeeper :

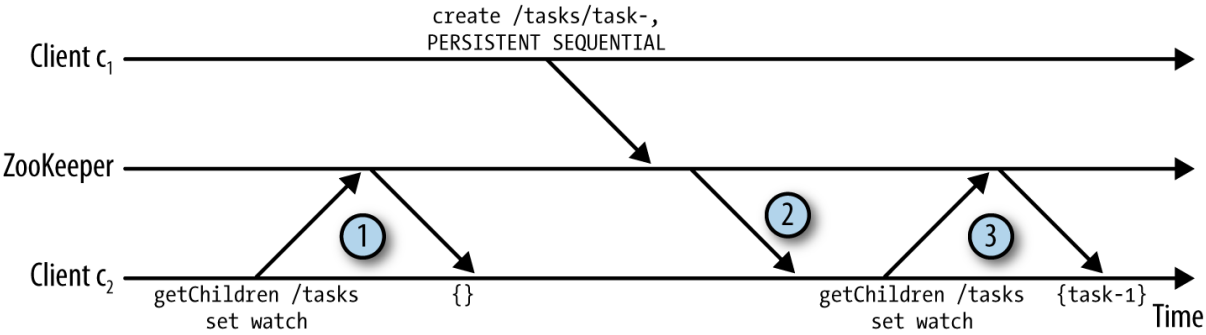
1. create /path
2. delete /path
3. exists /path
4. setData /path
5. getData /path
6. getChildren /path

Different modes of znodes : [Persistent | Ephemeral]

1. Persistent : can only be deleted by the delete call [Task assignment]
2. Ephemeral : If the client that created it closes connection or crashes. [Master, worker presence]
3. Persistent sequential
4. Ephemeral sequential

For sequential nodes, zookeeper assigns a number at the end. /tasks/task-1

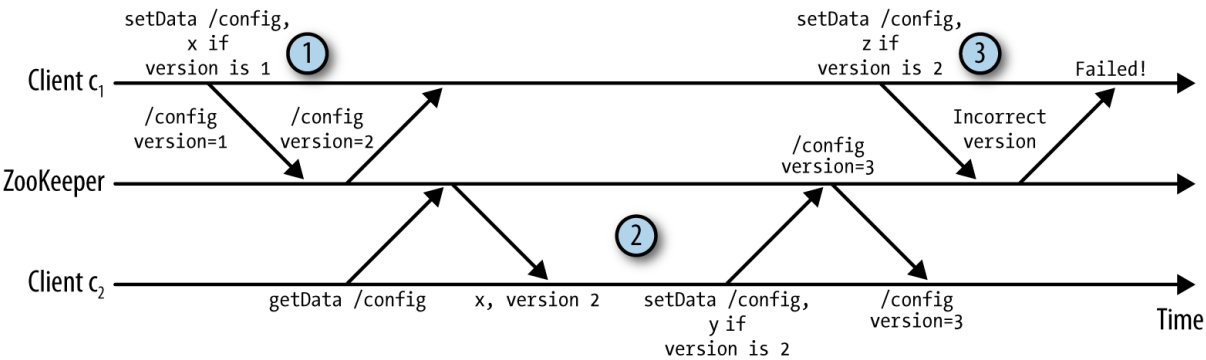
Because ZooKeeper is typically accessed as a remote service, accessing a znode every time a client needs to know its content would be very expensive. This is a common problem with polling. To replace the client polling,we have opted for a mechanism based on notifications: clients register with ZooKeeper to receive notifications of changes to znodes. Registering to receive a notification for a given znode consists of setting a watch. A watch is a one-shot operation, which means that it triggers one notification. To receive multiple notifications over time, the client must set a new watch upon receiving each notification.



1. Client c_2 reads the list of tasks, initially empty. It sets a watch for changes.
2. When there is a change, the client is notified.
3. Client c_2 reads the children of /tasks and observes the new task.

Before setting a new watch, the state of the znode is observed and notification is sent accordingly.

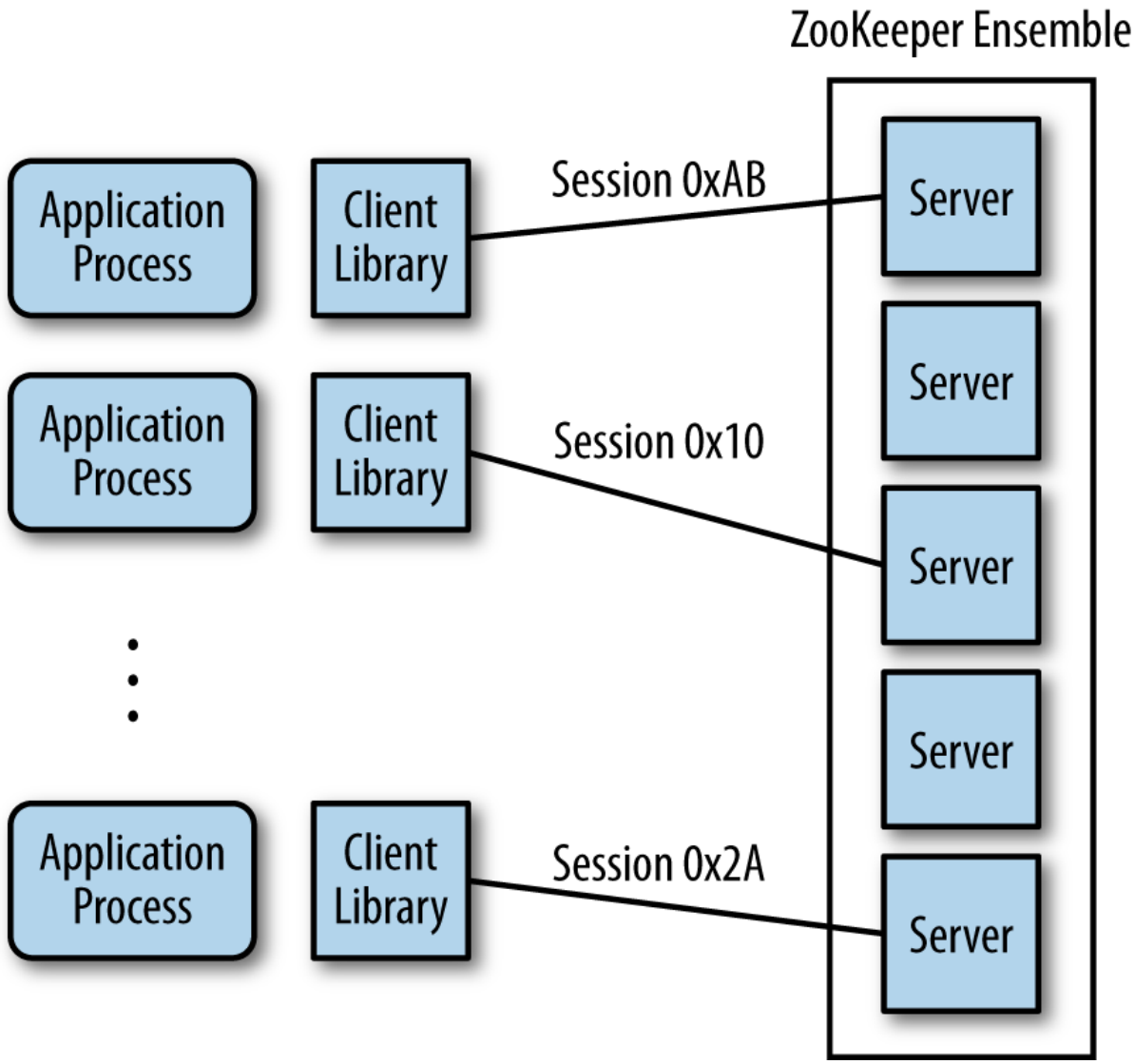
One important guarantee of notifications is that they are delivered to a client before any other change is made to the same znode. If a client sets a watch to a znode and there are two consecutive updates to the znode, the client receives the notification after the first update and before it has a chance to observe the second update by, say, reading the znode data. The key property we satisfy is the one that notifications preserve the order of updates the client observes. Although changes to the state of ZooKeeper may end up propagating more slowly to any given client, we guarantee that clients observe changes to the ZooKeeper state according to a global order.



1. Client c_1 writes the first version of /config.
2. Client c_2 reads /config and writes the second version.
3. Client c_1 tries to write a change to /config, but the request fails because the version does not match.

A couple of operations in the API can be executed conditionally: setData and delete. Both calls take a version as an input parameter, and the operation succeeds only if the version passed by the client matches the current version on the server.

ARCHITECTURE :



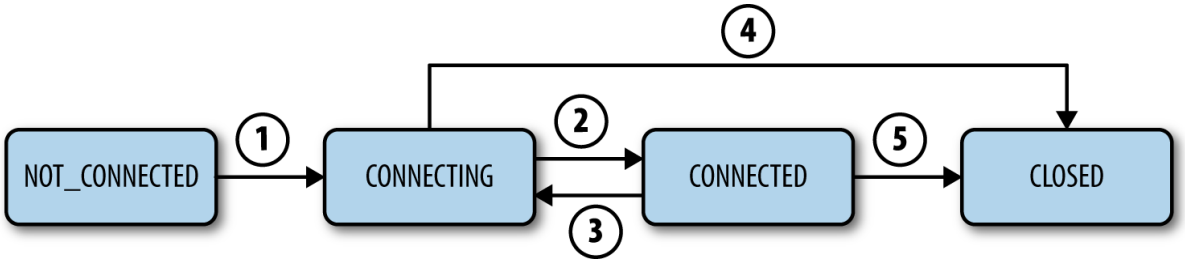
Applications communicate with ZooKeeper ensemble using the client library. ZooKeeper servers run in two modes: standalone and quorum. In quorum mode, a group of ZooKeeper servers, which we call a ZooKeeper ensemble, replicates the state, and together they serve client requests. It is the minimum number of servers that have to be running and available in order for ZooKeeper to work. This number is also the minimum number of servers that have to store a client's data before telling the client it is safely stored. Using such a majority scheme, we are able to tolerate the crash of f servers, where f is less than half of the servers in the ensemble.

Before executing any request against a ZooKeeper ensemble, a client must establish a session with the service. All operations a client submits to ZooKeeper are associated to a session. When a session ends for any reason, the ephemeral nodes created during that session disappear. Moving a session to a different server is handled transparently by the ZooKeeper client library in case if it does not hear from the connected server.

Sessions offer order guarantees, which means that requests in a session are executed in FIFO (first in, first out) order, argument completely changes with the concurrent sessions or multiple sessions.

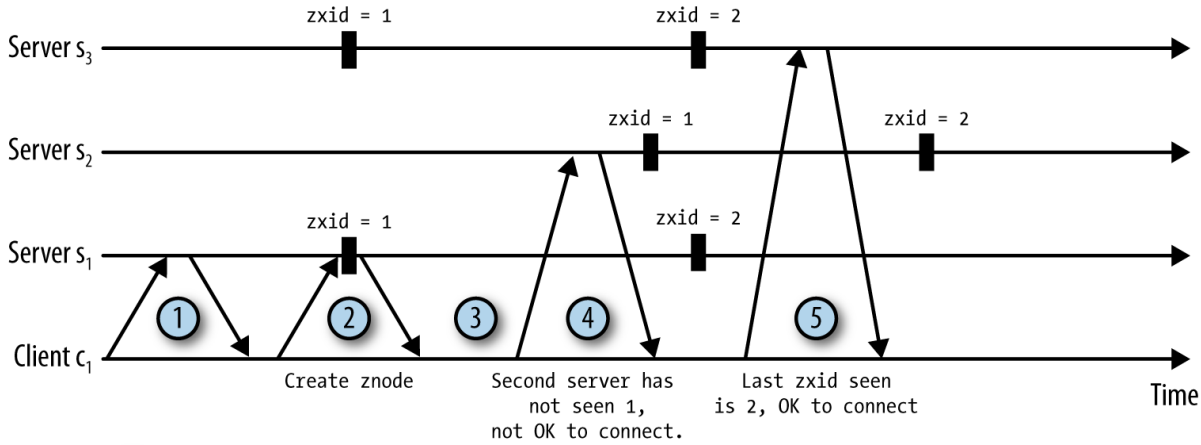
The lifetime of a session corresponds to the period between its creation and its end, whether it is closed gracefully or expires because of a timeout

.



One important parameter you should set when creating a session is the session timeout, which is the amount of time the ZooKeeper service allows a session before declaring it expired. In quorum mode, the application is supposed to pass a list of servers the client can connect to and choose from.

When trying to connect to a different server, it is important for the ZooKeeper state of this server to be at least as fresh as the last ZooKeeper state the client has observed. ZooKeeper determines freshness by ordering updates in the service. Every change to the state of a ZooKeeper deployment is totally ordered with respect to all other executed updates. In the ZooKeeper implementation, the transaction identifiers that the system assigns to each update establish the order.



1. Client connects to Server s_1.
2. Client executes a create operation. The operation succeeds and has zxid 1, assigned by the service.
3. Client disconnects from s_1.
4. Client tries to connect to s_2, but the server has a lower zxid.
5. Client attempt to connect to s_3 succeeds.

Running ZooKeeper in quorum :

In configuration file,

server.1=127.0.0.1:2222:2223

server.2=127.0.0.1:3333:3334

server.3=127.0.0.1:4444:4445

The second and third fields are TCP port numbers used for quorum communication and leader election.

Set up some data directories. When we start up a server, it needs to know which server it is.A server figures out its ID by reading a file named myid in the data directory.

echo 1 > z1/data/myid

echo 2 > z2/data/myid

echo 3 > z3/data/myid

Start server 1 :

$ {PATH_TO_ZK}/bin/zkServer.sh start ./z1.cfg

Log :

[myid:1] – INFO [QuorumPeer[myid=1]/…:2181:QuorumPeer@670] – LOOKING… [myid:1] – INFO [QuorumPeer[myid=1]/ …:2181:FastLeaderElection@740] -New election. My id = 1, proposed zxid=0x0… [myid:1] – INFO [WorkerReceiver[myid=1]:FastLeaderElection@542] -Notification: 1 …, LOOKING (my state)… [myid:1] – WARN [WorkerSender[myid=1]:QuorumCnxManager@368] –

Cannot open channel to 2 at election address /127.0.0.1:3334

Java.net.ConnectException: Connection refused

at java.net.PlainSocketImpl.socketConnect(Native Method)

at java.net.PlainSocketImpl.doConnect(PlainSocketImpl.java:351)

Start server 2 :

$ {PATH_TO_ZK}/bin/zkServer.sh start ./z2.cfg

Log :

… [myid:2] – INFO [QuorumPeer[myid=2]/…:2182:Leader@345] –

LEADING- LEADER ELECTION TOOK – 279

… [myid:2] – INFO [QuorumPeer[myid=2]/…:2182:FileTxnSnapLog@240] –

Snapshotting: 0x0 to ./data/version-2/snapshot.0

Clients connect in a random order to servers in the connect string.This allows ZooKeeper to achieve simple load balancing. However, it doesn’t allow clients to

specify a preference for a server to connect to. One of the nice things about ZooKeeper is that, apart from the connect string, it doesn't matter to the clients how many servers make up the ZooKeeper service.

To implement locks we just have to make the /lock znode ephemeral when we create it. Other processes that try to create /lock fail so long as the znode exists. So, they watch for changes to /lock and try to acquire the lock again once they detect that /lock has been deleted. Upon receiving a notification that /lock has been deleted, if a process p' is still interested in acquiring the lock, it repeats the steps of attempting to create /lock, and if another process has created the znode already, watching it.

Master-worker model involves three roles :

1. Master : watches for new workers and tasks, assigning tasks to available workers.
2. Worker : register themselves with the system, to make sure that the master sees they are available to execute tasks, and then watch for new tasks.
3. Client : create new tasks and wait for responses from the system.

In a real application, these znodes need to be created either by a primary process before it starts assigning tasks or by some bootstrap procedure. Regardless of how they are created, once they exist, the master needs to watch for changes in the children of /workers and /tasks.

ZOOKEEPER INTERNALS :

But what exactly are these servers doing with the operations the clients send?

The leader is the central point for handling all requests that change the ZooKeeper system. It acts as a sequencer and establishes the order of updates to the ZooKeeper state. Followers receive and vote on the updates proposed by the leader to guarantee that updates to the state survive crashes. There is a third kind of server, however, called an observer. Observers do not participate in the decision process of what requests get applied; they only learn what has been decided upon. Observers are there for scalability reasons.

ZooKeeper servers process read requests (exists, getData, and getChildren) locally. ZooKeeper is pretty fast at serving read-dominated workloads. Client requests that change the state of ZooKeeper (create, delete, and setData) are forwarded to the

leader. The leader executes the request, producing a state update that we call a transaction. Whereas the request expresses the operation the way the client originates it, the transaction comprises the steps taken to modify the ZooKeeper state to reflect the execution of the request.

A transaction is treated as a unit, in the sense that all changes it contains must be applied atomically. In the setData example, changing the data without an accompanying change to the version accordingly leads to trouble. For a long time, the design used a single thread in each server to apply transactions. Having a single thread guarantees that the transactions are applied sequentially without interference. A transaction is also idempotent. That is, we can apply the same transaction twice and we will get the same result. We can even apply multiple transactions multiple times and get the same result, as long as we apply them in the same order every time.We take advantage of this idempotent property during recovery.

Leader Election

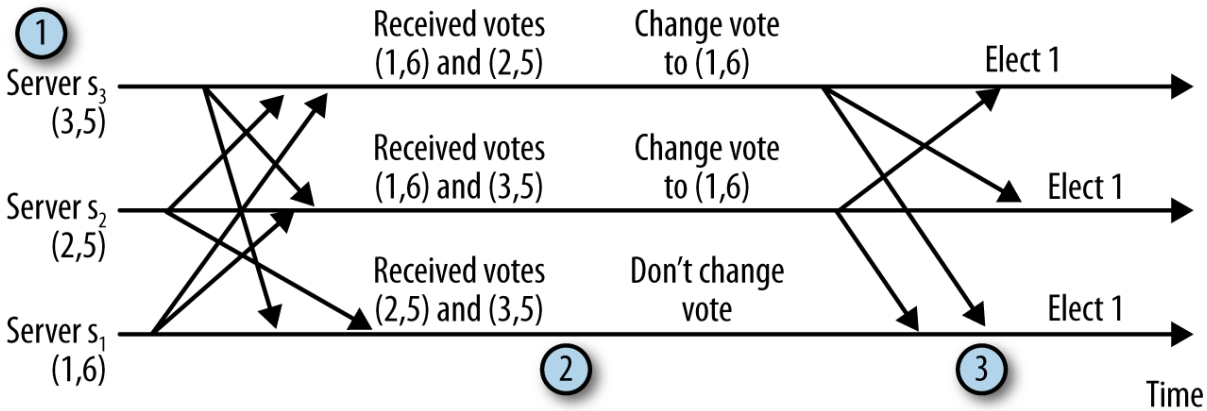
The purpose of the leader is to order client requests that change the ZooKeeper state: create, setData, and delete. The leader transforms each request into a transaction. Each server starts in the LOOKING state, where it must either elect a new leader or find the existing one. If a leader already exists, 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.

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.

A zxid is a long (64-bit) integer split into two parts: the epoch and the counter.

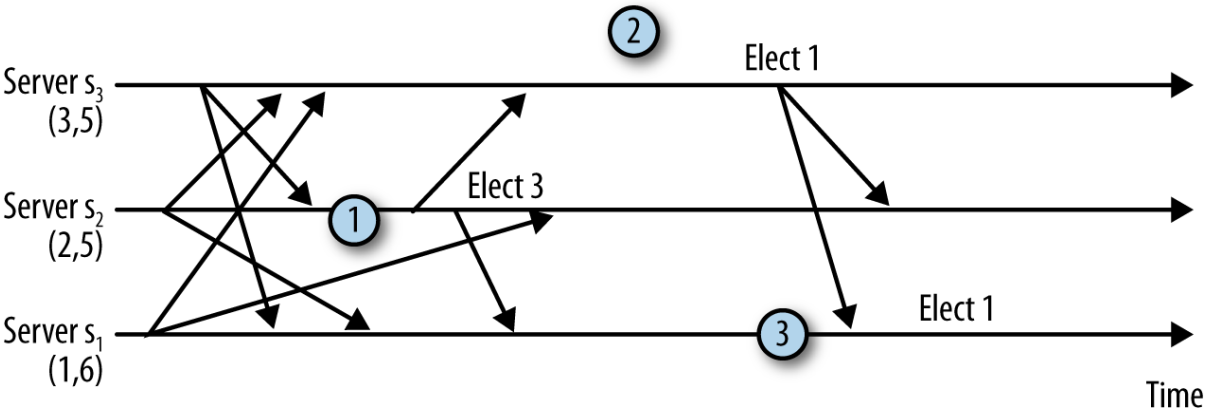
The Java class in ZooKeeper that implements an election is QuorumPeer.

Ideal leader election :



1. Server s_1 starts with vote (1,6), server s_2 with (2,5) and server s_3 with (3,5).
2. Servers s_2 and s_3 change their vote to (1,6) and send a new batch of notifications.
3. All three servers receive the same vote from a quorum and elect server s_1.

Leader election gone wrong :



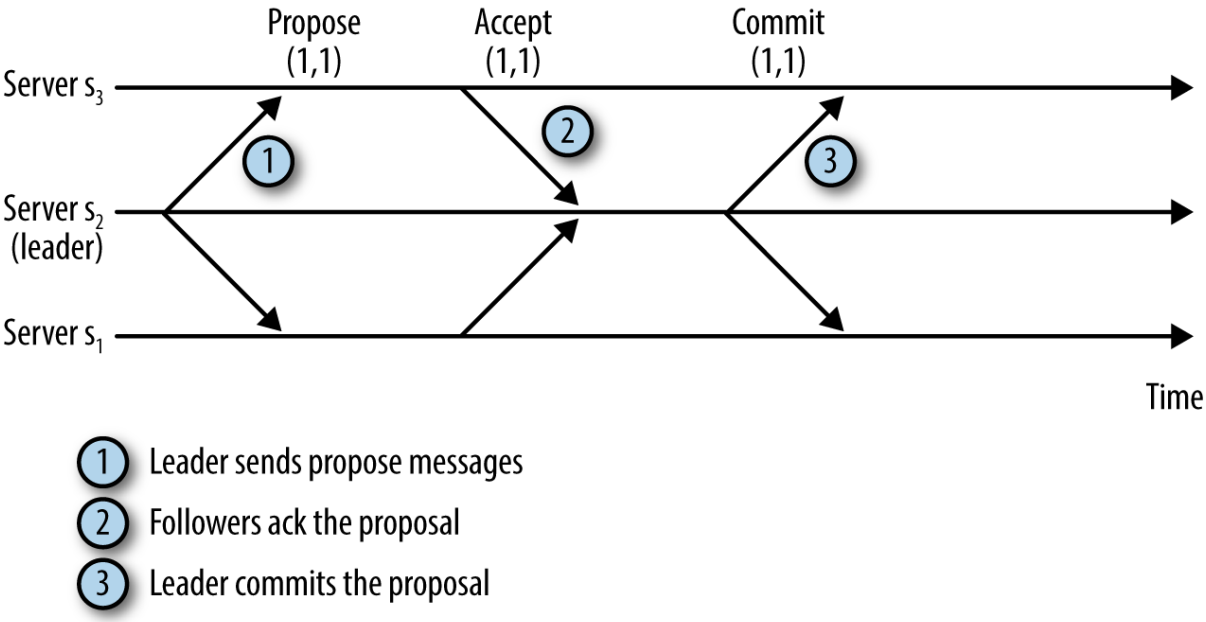
1. Server s_2 receives vote (3,5) and changes its vote, forming a quorum. It elects server s_3.
2. Server s_3 receives vote (1,6), but it takes some time to send a new batch of notifications.
3. Server s_1 elects itself leader once it receives the vote of server s_3.

Having s2 elect a different leader does not cause the service to behave incorrectly, because s3 will not respond to s2 as leader. Eventually s2 will time out trying to get a response from its elected leader, s3, and try again. Trying again, however, means that during this time s2 will not be available to process client requests, which is undesirable. The initial leader election algorithm implemented a pull-based model, and the interval for a server to pull votes was about 1 second.

ZAB : Broadcasting state updates :

How a server determines that a transaction has been committed. This follows a protocol called Zab: the ZooKeeper Atomic Broadcast protocol. Protocol to commit a transaction is very simple, resembling a two phase commit :

1. The leader sends a PROPOSAL message, p, to all followers.
2. Upon receiving p, a follower responds to the leader with an ACK, informing the leader that it has accepted the proposal.
3. Upon receiving acknowledgments from a quorum (the quorum includes the leader itself), the leader sends a message informing the followers to COMMIT it.



Before acknowledging a proposal, the follower needs to perform a couple of additional checks. The follower needs to check that the proposal is from the leader it is currently following, and that it is acknowledging proposals and committing transactions in the same order that the leader broadcasts them in. Zab records a transaction in a quorum of servers. A quorum must acknowledge a transaction before the leader commits it, and a follower records on disk the fact that it has acknowledged the transaction.

Transactions can still end up on some servers and not on others, because servers can fail while trying to write a transaction to storage. ZooKeeper can bring all servers up to date whenever a new quorum is created and a new leader chosen. The notion of epochs represents the changes in leadership over time. An epoch refers to the period during which a given server exercised leadership. Remember that each zxid includes the epoch as its first element, so each zxid can easily be associated to the epoch in

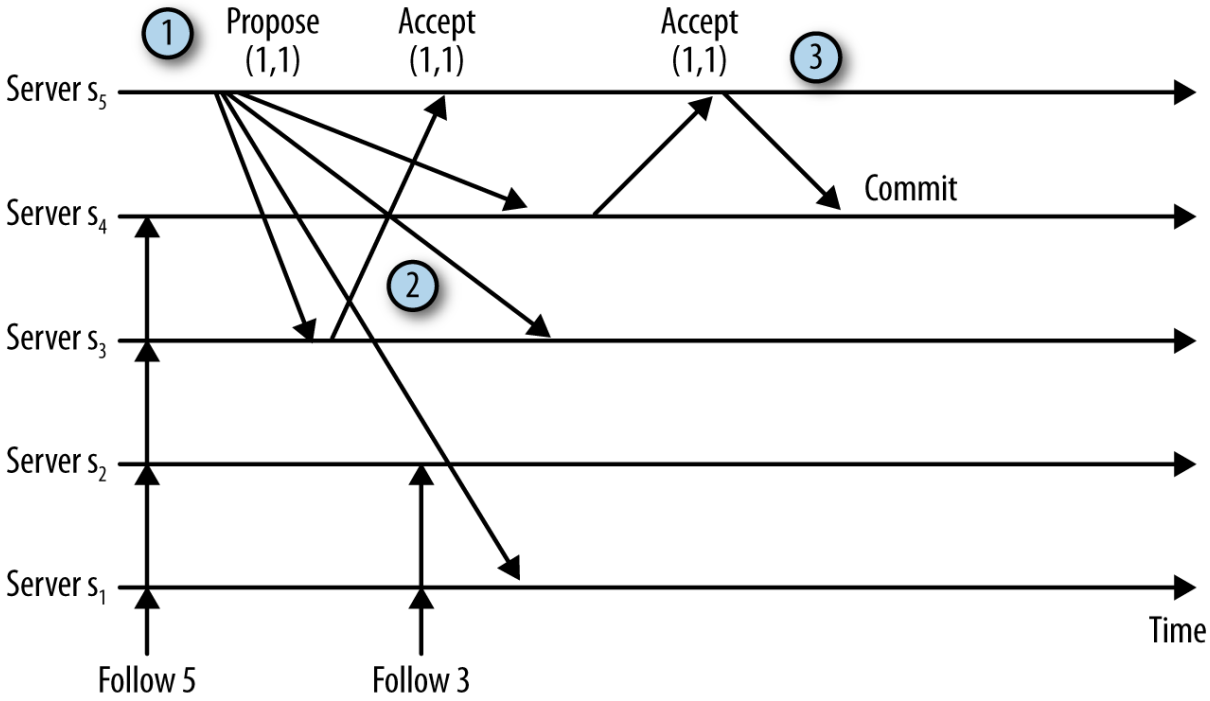
which the transaction was created. The epoch number increases each time a new leader election takes place.

The follower may accept proposals from epoch 4 during the recovery period of epoch 6, before it starts accepting new proposals for epoch 6. Such proposals, however, are sent as part of epoch 6's messages. Most of the difficulty with implementing a broadcast protocol is related to the presence of concurrent leaders, not necessarily in a split-brain scenario. Multiple concurrent leaders could make servers commit transactions out of order or skip transactions altogether, which leaves servers with inconsistent states.

ZAB guarantee :

1. An elected leader has committed all transactions that will ever be committed from previous epochs before it starts broadcasting new transactions.
2. At no point in time will two servers have a quorum of supporters.

With Zab, a leader does not become active until it makes sure that a quorum of servers agrees on the state it should start with for the new epoch. The initial state of an epoch must include all transactions that have been previously committed, and possibly some other ones that had been accepted before but not committed. It is important, though, that before the leader makes any new proposals for epoch e, it commits all proposals that will ever be committed from epochs up to and including e – 1.



1. Server s_5 is elected and servers s_1–s_4 follow s_5.
2. Server s_3 is elected and servers s_1 and s_2 follow s_3.
3. Server s_5 receives an acknowledgment from server s_4, forming a quorum of responses, after server s_3 is elected and supported by a quorum (servers s_1–s_3). An acknowledgment from server s_5 is implicit.

In the figure, l is server s5, l′ is s3, Q comprises s1 through s3, and the zxid of T is ⟨1,1⟩. After receiving the second confirmation, s5 is able to send a commit message to s4 to tell it to commit the transaction. The other servers ignore messages from s5 once they start following s3. Note that s3 acknowledged ⟨1,1⟩, so it is aware of the transaction when it establishes leadership.

We have just promised that Zab ensures the new leader l′ does not miss ⟨1,1⟩, but how does it happen exactly? Before becoming active, the new leader l′ must learn all proposals that servers in the old quorum have accepted previously, and it must get a promise that these servers won't accept further proposals from previous leaders. In the example in Figure 9-5, the servers forming a quorum and supporting l′ promise that they won't accept any more proposals from leader l. At that point, if leader l is still able to commit any proposal, as it does with ⟨1,1⟩, the proposal must have been accepted by at least one server in the quorum that made the promise to the new leader. Recall that quorums must overlap in at least one server, so the quorum that l uses to commit and the quorum that l′ talks to must have at least one server in common. Consequently, l′ includes ⟨1,1⟩ in its state and propagates it to its followers.

Recall that when electing a leader, servers pick the one with the highest zxid. This saves ZooKeeper from having to transfer proposals from followers to the leader; it only needs to transfer state from the leader to the followers. ZooKeeper uses two different ways to update the followers in order to optimize the process, sends the missing transactions [DIFF in the code] or does a full snapshot transfer[SNAP in the code].

Unlike followers, though, observers do not participate in the voting process we discussed earlier. They simply learn the proposals that have been committed via INFORM messages. Both followers and observers are called learners.

One main reason for having observers is scalability of connections and read requests. By adding more observers, we can serve more clients and more read traffic with a smaller penalty to the throughput of writes. Note that the throughput of writes is driven by the quorum size. If we add more servers that can vote, we end up with larger quorums, which reduces write throughput. Adding observers, however, is not completely free of cost; each new observer induces the cost of one extra message per committed transaction. This cost is less, however, than that of adding servers to the voting process.

Another reason for observers is to have a deployment that spans multiple data centers. Scattering participants across data centers might slow down the system significantly because of the latency of links connecting data centers. With observers, update requests can be executed with high throughput and low latency in a single data center, while propagating to other data centers so that clients in other locations can consume them.

SERVER PIPELINE :

STANDALONE :

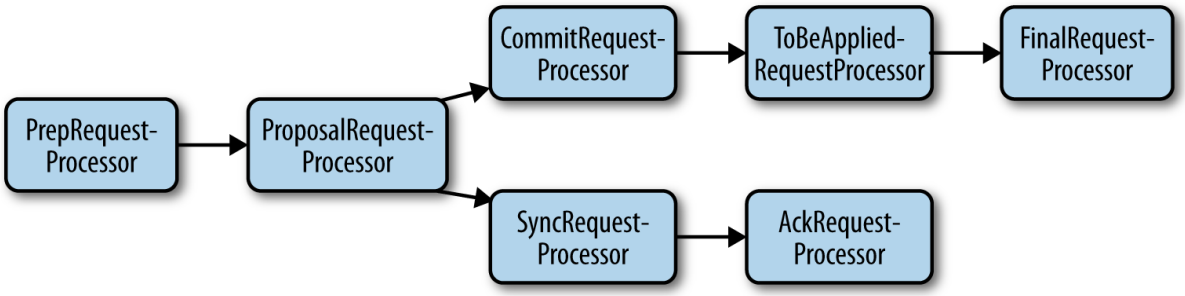
PrepRequestProcessor accepts a client request and executes it, generating a transaction as a result.

SyncRequestProcessor is responsible for persisting transactions to disk.

FinalRequestProcessor applies changes to the data tree when the Request object contains a transaction.

QUORUM MODE :

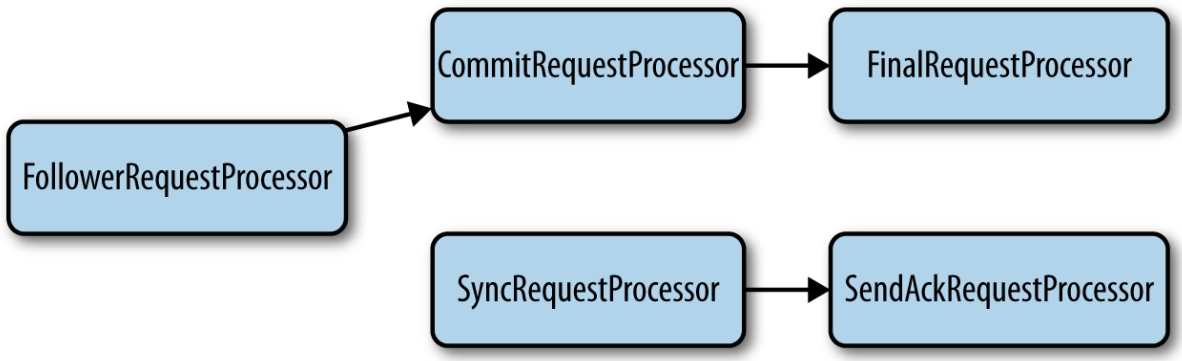
Leader server :



ProposalRequestProcessor forwards all requests to CommitRequestProcessor, and write requests to SyncRequestProcessor. It persists transactions to disk and ends by triggering AckRequestProcessor, a simple request processor that generates an acknowledgment back to itself.

CommitRequestProcessor commits proposals that have received enough acknowledgments. The acknowledgments are actually processed in the Leader class (the Leader.processAck() method), which adds committed requests to a queue in CommitRequestProcessor. ToBeAppliedRequestProcessor consists list of to-be-applied requests contains requests that have been acknowledged by a quorum and are waiting to be applied.

Followers and observers:



FollowerRequestProcessor, which receives and processes client requests. CommitRequestProcessor, additionally forwarding write requests to the leader. CommitRequestProcessor forwards read requests directly to FinalRequestProcessor, whereas for write requests, CommitRequestProcessor must wait for a commit before forwarding to FinalRequestProcessor.

SyncRequestProcessor processes the request, logging it to disk, and forwards it to SendAckRequestProcessor, which acknowledges the proposal to the leader. After the leader receives enough acknowledgments to commit a proposal, the leader sends commit messages to the followers (and sends INFORM messages to the observers). Upon receiving a commit message, a follower processes it with CommitRequestProcessor. For observers, it is not necessary to send acknowledgment messages back to the leader or persist transactions to disk.

Because writing to the transaction log is in the critical path of write requests, ZooKeeper needs to be efficient about it. ZooKeeper uses the following tricks, group commits consist of appending multiple transactions in a single write to the disk. Flushing here simply means that we tell the operating system to write dirty pages to disk and return when the operation completes.

Modern disks have a write cache that stores data to be written to disk. If the write cache is enabled, a call to force does not guarantee that, upon return, the data is on media. Instead, it could be sitting in the write cache. To guarantee that written data is on media upon returning from a call to FileChannel.force(), the disk write cache must be disabled. Operating systems have different ways of disabling it.

Padding consists of preallocating disk blocks to a file. This is done so that updates to the file system metadata for block allocation do not significantly affect sequential writes to the file. If transactions are being appended to the log at a high speed, and if blocks were not preallocated to the file, the file system would need to allocate a new block whenever it reached the end of the one it was writing to. This would induce at least two extra disk seeks: one to update the metadata and another back to the file.

Leave a comment

Comment

Post Comment