ZooKeeper FAQ

Q: What's the main take-away from this paper?

A: ZooKeeper's main academic contribution lies in the detailed design of a storage system specialized to fault-tolerant high-performance configuration management: watches, sessions, the choice of consistency, the specific semantics of the operations. It builds on existing work such as Chubby and Paxos.

A lot of what's interesting for us is the idea that one can obtain fault tolerance by keeping the critical state in fault-tolerant storage (ZooKeeper), and running the computation in non-fault-tolerant servers. For example, a MapReduce coordinator might keep state about jobs, task status, workers, location of intermediate output, &c, in ZooKeeper. If the coordinator fails, a new computer can be selected to run the MapReduce coordinator software, and it can load its state from ZooKeeper. This provides fault-tolerance for the coordinator without the complexity of state-machine replication (e.g. without having to write the MapReduce coordinator using a Raft library). You can think of this as providing fault-tolerance by making the state alone fault-tolerant, whereas use of Raft makes the entire computation fault-tolerant. This general pattern isn't new -- for example it's how database-backed web sites work -- but ZooKeeper is a good fit if your main concern is managing fault-tolerant services.

Q: What's the point of sessions?

A: A session consists of some state maintained by the client and ZooKeeper. The client tags each request with its session ID. If ZooKeeper doesn't hear from a client for a while because the client has failed or there's a network problem, the ZooKeeper leader will expire (destroy) the session.

One role of sessions is to manage the state required to guarantee FIFO client order, and to keep track of each client's watches.

More importantly, sessions are involved in the way ephemeral znodes work. When a client creates an ephemeral file, ZooKeeper remembers which client session created the file. If the ZooKeeper leader expires a session, then it will also delete all ephemeral files created by that session. In addition, and atomically with deleting the ephemeral files, ZooKeeper guarantees to ignore any further client requests from the expired session.

This arrangement works particularly well when applications implement lock-like constructions (e.g. leader election) in ZooKeeper. By representing the lock with an ephemeral file, applications can arrange that a lock will automatically be released if ZooKeeper thinks the lock holder has failed. Then another client can acquire the lock. But what if the original lock holder is actually alive, and continues to act as if it holds the lock by sending writes of the locked data to ZooKeeper? Because ZooKeeper stops honoring a session's requests at the moment ZooKeeper expires the session and deletes the ephemeral lock file, the original lock holder will automatically be prevented from changing anything stored in ZooKeeper.

Q: How does A-linearizability differ from linearizability?

A: A ZooKeeper client can send lots of "asynchronous" requests, without waiting for each to finish before sending the next. By the rules of ordinary linearizability, these requests are concurrent (they overlap in time), and therefor can be executed in any order. In contrast, ZooKeeper guarantees to execute them in the order that the client sent them. The paper calls this A-linearizability.

Q: Why are only update requests A-linearizable? Why not reads as well?

A: The authors want high total read throughput, so they want ZooKeeper replicas to be able to satisfy client reads and maintain watches without involving the leader. A given replica may not know about a committed write (if it's not in the majority that the leader waited for), or may know about a write but not yet know if it is committed. Thus a replica's state may lag behind the leader and other replicas. Thus serving reads from replicas can yield data that doesn't reflect recent writes -- that is, reads can return stale results.

Q: How does linearizability differ from serializability?

A: The usual definition of serializability is much like linearizability, but without the requirement that operations respect real-time ordering. Have a look at this explanation:
http://www.bailis.org/blog/linearizability-versus-serializability/

Section 2.3 of the ZooKeeper paper uses "serializable" to indicate that the system behaves as if writes (from all clients combined) were executed one by one in some order. The "FIFO client order" property means that reads occur at specific points in the order of writes, and that a given client's successive reads never move backwards in that order. Note that the guarantees for writes and reads are different.

Q: Why is it OK for ZooKeeper to respond to read requests with out-of-date data?

A: ZooKeeper is likely to yield data that is only slightly out of date: the leader tries hard to keep all the followers up to date, much as in Raft. So a follower may be a few writes (or batches of writes) behind the leader, but rarely much more than that.

Most uses of ZooKeeper have no problem with slightly-out-of-date read results. After all, even if ZooKeeper guaranteed to provide fresh results as of the time ZooKeeper executed the read, those results could easily be out of date by the time the reply arrived at the client. Because some other client might send a write request to ZooKeeper just after ZooKeeper replied to the read.

A typical use of ZooKeeper is for a MapReduce worker to register itself and look for work, perhaps in a ZooKeeper directory in which the MR coordinator writes task assignments. Suppose the worker checks for work every 10 seconds, or uses a ZooKeeper watch to be notified of any change in work assignments. In both cases, the worker is likely to hear about a new assignment somewhat after the assignment was made. But a little delay does not matter much (as long as each item of work is relatively long).

Q: What is pipelining?

A: There are two things going on here. First, the ZooKeeper leader (really the leader's Zab layer) batches together multiple client operations in order to send them efficiently over the network, and in order to efficiently write them to disk. For both network and disk, it's often far more efficient to send a batch of N small items all at once than it is to send or write them one at a time. This kind of batching is only effective if the leader sees many client requests at the same time; so it depends on there being lots of active clients.

The second aspect of pipelining is that ZooKeeper makes it easy for each client to keep many write requests outstanding at a time, by supporting asynchronous operations. From the client's point of view, it can send lots of write requests without having to wait for the responses (which arrive later, as notifications after the writes commit). From the leader's point of view, that client behavior gives the leader lots of requests to accumulate into big efficient batches.

A worry with pipelining is that operations that are in flight might be re-ordered, which would cause the problem that the authors discuss 2.3. If the leader has many write operations in flight followed by the creation of "ready", you don't want those operations to be re-ordered, because then other clients may observe "ready" before the preceding writes have been applied. To ensure that this cannot happen, Zookeeper guarantees FIFO for client operations; that is, ZooKeeper applies operations in the order that the client issued them.

Q: What does wait-free mean?

A: The precise definition: A wait-free implementation of a concurrent data object is one that guarantees that any process can complete any operation in a finite number of steps, regardless of the execution speeds of the other processes. This definition was introduced in the following paper by Herlihy:
https://cs.brown.edu/~mph/Herlihy91/p124-herlihy.pdf

Zookeeper is wait-free because it processes one client's requests without needing to wait for other clients to take action. This is partially a consequence of the API: despite being designed to support client/client coordination and synchronization, no ZooKeeper API call is defined in a way that would require one client to wait for another. In contrast, a system that provided a lock acquire operation that waited for the current lock holder to release the lock would not be wait-free.

Ultimately, however, ZooKeeper clients often need to wait for each other, and ZooKeeper does provide a waiting mechanism -- watches. The main effect of wait-freedom on the API is that watches are factored out from other operations. The combination of atomic test-and-set updates (e.g. file creation and writes condition on version) with watches allows clients to synthesize more complex blocking abstractions (e.g. Section 2.4's locks and barriers).

Q: How does the leader know the order in which a client wants a bunch of asynchronous updates to be performed?

A: The paper doesn't say. The answer is likely to involve the client's ZooKeeper library numbering its asynchronous requests, and the leader tracking for each client (really session) what number it should next expect. The leader has to keep state per session anyway (for client session timeouts), so it might be little extra work to track a per-session request sequence number. This information would have to be preserved when a leader fails and another server takes over, so the client sequence numbers are likely passed along in replicated log entries.

Q: What does a client do if it doesn't get a reply for a request? Does it re-send, in case the network lost a request or reply, or the leader crashed before committing? How does ZooKeeper avoid re-sends leading to duplicate executions?

A: The paper doesn't say how all this works. Probably the leader tracks what request numbers from each session it has received and committed, so that it can filter out duplicate requests. Lab 4 has a similar arrangement.

Q: If a client submits an asynchronous write, and immediately afterwards does a read, will the read see the effect of the write?

A: The paper doesn't explicitly say, but the implication of the "FIFO client order" property of Section 2.3 is that the read will see the write. That seems to imply that a ZK follower may block a read until the follower has received (from the ZK leader) all of the client's preceding writes. The follower is in a position to do this because a client session sends all requests (read and write) to the same follower, which therefor will be aware that the client has issued a write that hasn't yet appeared in the stream of committed operations from the leader.

Q: What is the reason for implementing 'fuzzy snapshots'?

A: ZooKeeper needs to write its state to disk so that it can recover from a power failure (by reading the data from the disk). It does this by appending every write operation to a log on the disk, and (to prevent that log from growing too long) it periodically writes a "snapshot" of its entire state (all the data) to disk and truncates the log. Thus the most most recent snapshot plus the log since that snapshot contain all the data.

A precise snapshot would correspond to a specific point in the log: the snapshot would include every write before that point, and no writes after that point; and it would be clear exactly where to start replay of log entries after a reboot to bring the snapshot up to date. However, creation of a precise snapshot requires a way to prevent any writes from happening while the snapshot is being created and written to disk. Blocking writes for the duration of snapshot creation might decrease performance a lot.

The "fuzzy" refers to the fact that ZooKeeper creates the snapshot from its in-memory database while allowing writes to the database. This means that a snapshot does not correspond to a particular point in the log -- a snapshot includes a more or less random subset of the writes that were concurrent with snapshot creation. After reboot, ZooKeeper constructs a consistent snapshot by replaying all log entries from the point at which the snapshot started, in order. Because logged updates in Zookeeper are idempotent and describe the state resulting from the client operation, the application-state will be correct after reboot and replay---some messages may be applied twice (once to the state before recovery and once after recovery) but that is OK, because they are idempotent. The replay fixes the fuzzy snapshot to be a consistent snapshot of the application state.

The Zookeeper leader turns the operations in the client API into idempotent transactions. For example, if a client issues a conditional setData and the version number in the request matches, the Zookeeper leader creates a setDataTXN that contains the new data, the new version number, and updated time stamps. This transaction (TXN) is idempotent: Zookeeper can execute it twice and it will result in the same state.

Q: How does ZooKeeper choose leaders?

A: Zookeeper uses ZAB, an atomic broadcast system, which has leader election built in, much like Raft. Here's a paper about Zab: http://dl.acm.org/citation.cfm?id=2056409

Q: How does Zookeeper's performance compare to other systems such as Paxos?

A: Zookeeper has impressive performance (in particular throughput). Three Zookeeper servers process 21,000 writes per second. Typical 6.5840 Rafts with 3 servers commit on the order of tens of operations per second (assuming a magnetic disk for storage) and maybe hundreds per second with SSDs.

Q: How big is the ZooKeeper database? It seems like the server must
have a lot of memory.

It depends on the application, and, unfortunately, the paper doesn't
report the authors' experience in this area. Since Zookeeper is
intended for configuration and coordination, and not as a
general-purpose data store, an in-memory database seems reasonable.
For example, you could imagine using Zookeeper for GFS's coordinator and
that amount of data should fit in the memory of a well-equipped
server, as it did for GFS.

Q: What's a universal object?

A: It is a theoretical statement of how good the API of Zookeeper is
based on a theory of concurrent objects that Herlihy introduced:
https://cs.brown.edu/~mph/Herlihy91/p124-herlihy.pdf. We won't
spend any time on this statement and theory, but if you care there is
a gentle introduction on this Wikipedia page:
https://en.wikipedia.org/wiki/Non-blocking_algorithm.

The authors appeal to this concurrent-object theory in order to show
that Zookeeper's API is general-purpose: that the API includes enough
features to implement any coordination scheme you'd want.

Q: When might one use a double barrier?

A: Here's a use for a single barrier: suppose you have a big
computation that proceeds in two phases. All of the results for phase
1 must be available before phase 2 can begin. Suppose you run each
phase in parallel on many workers (like MapReduce's Map, and then
Reduce). A barrier can be used to enforce "all workers must finish
phase 1 before any can start phase 2". If you have many phases, or a
loop in which each iteration is a phase, you might want a double
barrier.

Q: How does a client know when to leave one of the paper's double barriers?

A: Leaving the barrier involves each client watching the znodes for
all other clients participating in the barrier. Each client waits for
all of these znodes to disappear. Once they are all gone, the clients
all leave the barrier and continue computing.

Q: Is it possible to add more servers into an existing ZooKeeper
without taking the service down for a period of time?

A: It is -- although when the original paper was published, cluster
membership was static. Nowadays, ZooKeeper supports "dynamic
reconfiguration":

https://zookeeper.apache.org/doc/r3.5.3-beta/zookeeperReconfig.html

... and there is a paper describing the mechanism:

https://www.usenix.org/system/files/conference/atc12/atc12-final74.pdf

How do you think this compares to Raft's dynamic configuration change
via overlapping consensus, which appeared two years later?

Q: How are watches implemented in the client library?

A: The client library probably registers a callback function that will
be invoked when the watch triggers.

For example, a Go client for ZooKeeper implements it by passing a channel into "GetW()" (get with watch); when the watch triggers, an "Event" structure is sent through the channel. The application can check the channel in a select clause.

See https://godoc.org/github.com/samuel/go-zookeeper/zk#Conn.GetW.

Q: Why does the read lock on page 6 in the code jump to line 3 instead of line 2 like the write lock does?

A: Good catch; I believe it is a bug. The correct recipe is here: https://zookeeper.apache.org/doc/r3.1.2/recipes.html#Shared+Locks

Q: Section 5.2 says the request latency for a write is 1.2 milliseconds. Isn't that less than the time required for a single hard disk write?

A: I don't know what's going on here. Section 4 says each new log entry is forced to disk, and seems to imply that ZooKeeper waits for the data to be on disk before proceeding. The wait seems necessary for crash recovery -- it would be bad to tell a client that a write had succeeded, only to forget about it in a power failure. How long might a disk write take? Section 5 says they use mechanical hard drives, and appending an entry to a log file on disk takes at least half a rotation on average. A typical disk spins at 7200 RPM, which means a half rotation takes about 4 milliseconds, which is longer than the reported overall latency.

If the paper were published today, and said it used SSDs instead of hard drives, the 1.2 millisecond latency would make more sense, since one can write an SSD in much less than a millisecond. Another way to get fast writes is to interpose a battery-backed cache between the software and the disk -- the point of the battery is to preserve recent writes despite power failures, so they can be written to disk after the power is restored.

Q: Why did the authors choose the name ZooKeeper?

A: The apache zookeeper web site says this: "ZooKeeper: Because Coordinating Distributed Systems is a Zoo"