ZooKeeper FAQ

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

A: The authors want high total read throughput, so they want replicas to be able to satisfy client reads without involving the leader. A given replica may not know about a commited 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. One thing that's going on here is that the guarantees for writes and reads are different.

Q: What is pipelining?

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 to talk about in 2.3. If a the leader has many write operations in flight followed by write to 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 the client operations are applied in the order they have been issued.

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 supported 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 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 3 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 server may block a read until the server has received (from the leader) all of the client's preceding writes. ZooKeeper probably manages this by having the client send, in its read request, the zxid of the latest preceding operation that the client submitted.

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

A: 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 would decrease performance a lot.

The point of ZooKeeper's fuzzy snapshots is 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. Because updates in Zookeeper are idempotent and delivered in the same order, 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: It has impressive performance (in particular throughput); Zookeeper would beat the pants of your implementation of Raft. 3 zookeeper servers process 21,000 writes per second. Your raft with 3 servers commits on the order of tens of operations per second (assuming a magnetic disk for storage) and maybe hundreds per second with SSDs.

Q: How does the ordering guarantee solve the race conditions in Section 2.3?

If a client issues many write operations to various z-nodes, and then the write to Ready, then Zookeeper will guarantee that all the writes will be applied to the z-nodes before the write to Ready. Thus, if another client observes Ready, then all the preceding writes must have been applied and thus it is ok for the client to read the info in the z-nodes.

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 master 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: How does a client know when to leave a barrier (top of page 7)?

A: Leaving the barrier involves each client watching the znodes for all other clients participating in the barrier. Each client waits for all of them to be gone. If they are all gone, they 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?

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 actually 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?

It depends on the implementation. In most cases, 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.