```
6.824 2020 Lecture 8: Zookeeper Case Study

Reading: "ZooKeeper: wait-free coordination for internet-scale systems", Patrick
Hunt, Mahadev Konar, Flavio P. Junqueira, Benjamin Reed.  Proceedings of the 2010
USENIX Annual Technical Conference.

What questions does this paper shed light on?
  * Can we have coordination as a stand-alone general-purpose service?
    What should the API look like?
    How can other distributed applications use it?
  * We paid lots of money for Nx replica servers.
    Can we get Nx performance from them?

First, performance.
  For now, view ZooKeeper as some service replicated with a Raft-like scheme.
  Much like Lab 3.
  [clients, leader/state/log, followers/state/log]

Does this replication arrangement get faster as we add more servers?
  Assume a busy system, lots of active clients.
  Writes probably get slower with more replicas!
    Since leader must send each write to growing # of followers.
  What about reads?

Q: Can replicas serve read-only client requests form their local state?
   Without involving the leader or other replicas?
   Then total read capacity would be O(# servers), not O(1)!

Q: Would reads from followers be linearizable?
   Would reads always yield fresh data?
   No:
     Replica may not be in majority, so may not have seen a completed write.
     Replica may not yet have seen a commit for a completed write.
     Replica may be entirely cut off from the leader (same as above).
   Linearizability forbids stale reads!

Q: What if a client reads from an up-to-date replica, then a lagging replica?
   It may see data values go *backwards* in time! Also forbidden.

Raft and Lab 3 avoid these problems.
  Clients have to send reads to the leader.
  So Lab 3 reads are linearizable.
  But no opportunity to divide the read load over the followers.

How does ZooKeeper skin this cat?
  By changing the definition of correctness!
  It allows reads to yield stale data.
  But otherwise preserves order.

Ordering guarantees (Section 2.3)
  * Linearizable writes
    clients send writes to the leader
    the leader chooses an order, numbered by "zxid"
    sends to replicas, which all execute in zxid order
    this is just like the labs
  * FIFO client order
    each client specifies an order for its operations (reads AND writes)
    writes:
      writes appear in the write order in client-specified order
      this is the business about the "ready" file in 2.3
    reads:
      each read executes at a particular point in the write order
      a client's successive reads execute at non-decreasing points in the order
      a client's read executes after all previous writes by that client
```

```
        a server may block a client's read to wait for previous write, or sync()

Why does this make sense?
  I.e. why OK for reads to return stale data?
       why OK for client 1 to see new data, then client 2 sees older data?

At a high level:
  not as painful for programmers as it may seem
  very helpful for read performance!

Why is ZooKeeper useful despite loose consistency?
  sync() causes subsequent client reads to see preceding writes.
    useful when a read must see latest data
  Writes are well-behaved, e.g. exclusive test-and-set operations
    writes really do execute in order, on latest data.
  Read order rules ensure "read your own writes".
  Read order rules help reasoning.
    e.g. if read sees "ready" file, subsequent reads see previous writes.
         (Section 2.3)
         Write order:      Read order:
         delete("ready")
         write f1
         write f2
         create("ready")
                           exists("ready")
                           read f1
                           read f2
         even if client switches servers!
    e.g. watch triggered by a write delivered before reads from subsequent writes.
         Write order:      Read order:
                           exists("ready", watch=true)
                           read f1
         delete("ready")
         write f1
         write f2
                           read f2

A few consequences:
  Leader must preserve client write order across leader failure.
  Replicas must enforce "a client's reads never go backwards in zxid order"
    despite replica failure.
  Client must track highest zxid it has read
    to help ensure next read doesn't go backwards
    even if sent to a different replica

Other performance tricks in ZooKeeper:
  Clients can send async writes to leader (async = don't have to wait).
  Leader batches up many requests to reduce net and disk-write overhead.
    Assumes lots of active clients.
  Fuzzy snapshots (and idempotent updates) so snapshot doesn't stop writes.

Is the resulting performance good?
  Table 1
  High read throughput -- and goes up with number of servers!
  Lower write throughput -- and goes down with number of servers!
  21,000 writes/second is pretty good!
    Maybe limited by time to persist log to hard drives.
    But still MUCH higher than 10 milliseconds per disk write -- batching.

The other big ZooKeeper topic: a general-purpose coordination service.
  This is about the API and how it can help distributed s/w coordinate.
  It is not clear what such an API should look like!

What do we mean by coordination as a service?
  Example: VMware-FT's test-and-set server
```

```
    If one replica can't talk to the other, grabs t-a-s lock, becomes sole server
    Must be exclusive to avoid two primaries (e.g. if network partition)
    Must be fault-tolerant
  Example: GFS (more speculative)
    Perhaps agreement on which meta-data replica should be master
    Perhaps recording list of chunk servers, which chunks, who is primary
  Other examples: MapReduce, YMB, Crawler, etc.
    Who is the master; lists of workers; division of labor; status of tasks
  A general-purpose service would save much effort!

Could we use a Lab 3 key/value store as a generic coordination service?
  For example, to choose new GFS master if multiple replicas want to take over?
  perhaps
    Put("master", my IP address)
    if Get("master") == my IP address:
      act as master
  problem: a racing Put() may execute after the Get()
    2nd Put() overwrites first, so two masters, oops
    Put() and Get() are not a good API for mutual exclusion!
  problem: what to do if master fails?
    perhaps master repeatedly Put()s a fresh timestamp?
    lots of polling...
  problem: clients need to know when master changes
    periodic Get()s?
    lots of polling...

Zookeeper API overview (Figure 1)
  the state: a file-system-like tree of znodes
  file names, file content, directories, path names
  typical use: configuration info in znodes
    set of machines that participate in the application
    which machine is the primary
  each znode has a version number
  types of znodes:
    regular
    ephemeral
    sequential: name + seqno

Operations on znodes (Section 2.2)
  create(path, data, flags)
    exclusive -- only first create indicates success
  delete(path, version)
    if znode.version = version, then delete
  exists(path, watch)
    watch=true means also send notification if path is later created/deleted
  getData(path, watch)
  setData(path, data, version)
    if znode.version = version, then update
  getChildren(path, watch)
  sync()
    sync then read ensures writes before sync are visible to same client's read
    client could instead submit a write

ZooKeeper API well tuned to synchronization:
  + exclusive file creation; exactly one concurrent create returns success
  + getData()/setData(x, version) supports mini-transactions
  + sessions automate actions when clients fail (e.g. release lock on failure)
  + sequential files create order among multiple clients
  + watches -- avoid polling

Example: add one to a number stored in a ZooKeeper znode
  what if the read returns stale data?
    write will write the wrong value!
  what if another client concurrently updates?
    will one of the increments be lost?
  while true:
    x, v := getData("f")
    if setData(x + 1, version=v):
      break
  this is a "mini-transaction"
    effect is atomic read-modify-write
  lots of variants, e.g. test-and-set for VMware-FT

Example: Simple Locks (Section 2.4)
  acquire():
    while true:
      if create("lf", ephemeral=true), success
      if exists("lf", watch=true)
        wait for notification

  release(): (voluntarily or session timeout)
    delete("lf")

  Q: what if lock released just as loser calls exists()?

Example: Locks without Herd Effect
  (look at pseudo-code in paper, Section 2.4, page 6)
  1. create a "sequential" file
  2. list files
  3. if no lower-numbered, lock is acquired!
  4. if exists(next-lower-numbered, watch=true)
  5.   wait for event...
  6. goto 2

  Q: could a lower-numbered file be created between steps 2 and 3?
  Q: can watch fire before it is the client's turn?
  A: yes
     lock-10 <- current lock holder
     lock-11 <- next one
     lock-12 <- my request

     if client that created lock-11 dies before it gets the lock, the
     watch will fire but it isn't my turn yet.

Using these locks
  Different from single-machine thread locks!
    If lock holder fails, system automatically releases locks.
    So locks are not really enforcing atomicity of other activities.
    To make writes atomic, use "ready" trick or mini-transactions.
  Useful for master/leader election.
    New leader must inspect state and clean up.
  Or soft locks, for performance but not correctness
    e.g. only one worker does each Map or Reduce task (but OK if done twice)
    e.g. a URL crawled by only one worker (but OK if done twice)

ZooKeeper is a successful design.
  see ZooKeeper's Wikipedia page for a list of projects that use it
  Rarely eliminates all the complexity from distribution.
    e.g. GFS master still needs to replicate file meta-data.
    e.g. GFS primary has its own plan for replicating chunks.
  But does bite off a bunch of common cases:
    Master election.
    Persistent master state (if state is small).
    Who is the current master? (name service).
    Worker registration.
    Work queues.

Topics not covered:
  persistence
  details of batching and pipelining for performance
```

```
    fuzzy snapshots
    idempotent operations
    duplicate client request detection

  References:
    https://zookeeper.apache.org/doc/r3.4.8/api/org/apache/zookeeper/ZooKeeper.html
    ZAB: http://dl.acm.org/citation.cfm?id=2056409
    https://zookeeper.apache.org/
    https://cs.brown.edu/~mph/Herlihy91/p124-herlihy.pdf  (wait free, universal
    objects, etc.)
```