```
6.824 2017 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.

Why are we reading this paper?
  Widely-used replicated state machine service
    Inspired by Chubby (Google's global lock service)
    Originally at Yahoo!, now outside too (Mesos, HBase, etc.)
  Open source
    As Apache project (http://zookeeper.apache.org/)
  Case study of building replicated services, given a Paxos/ZAB/Raft library
    Similar issues show up in lab 3
  API supports a wide-range of use cases
    Application that need a fault-tolerant "master" don't need to roll their own
    Zookeeper is generic enough that they should be able to use Zookeeper
  High performance
    Unlike lab 3's replicate key/value service

Motivation: many applications in datacenter cluster need to coordinate
  Example: GFS
    master has list of chunk servers for each chunk
    master decides which chunk server is primary
    etc.
  Other examples: YMB, Crawler, etc.
    YMB needs master to shard topics
    Crawler needs master that commands page fetching
      (e.g., a bit like the master in mapreduce)
  Applications also need to find each other
    MapReduce needs to know IP:PORT of GFS master
    Load balancer needs to know where web servers are
  Coordination service typically used for this purpose

Motivation: performance -- lab3
  dominated by Raft
  consider a 3-node Raft
  before returning to client, Raft performs
    leader persists log entry
    in parallel, leader send message to followers
      each follower persist log entry
      each follower responds
  -> 2 disk writes and one round trip
    if magnetic disk: 2*10msec = 50 msg/sec
    if SSD: 2*2msec+1msec = 200 msg/sec
  Zookeeper performs 21,000 msg/sec
    asynchronous calls
    allows pipelining

Alternative plan: develop fault-tolerant master for each application
  announce location via DNS
  OK, if writing master isn't complicated
  But, master often needs to be:
    fault tolerant
      every application figures how to use Raft?
    high performance
      every application figures how to make "read" operations fast?
  DNS propagation is slow
    fail-over will take a long time!
  Some application settle for single-point of failure
    E.g., GFS and MapReduce
    Less desirable

Zookeeper: a generic coordination service
  Design challenges:
    What API?
    How to make master fault tolerant?
    How to get good performance?
  Challenges interact
    good performance may influence API
    e.g., asynchronous interface to allow pipelining

Zookeeper API overview
  [diagram: ZooKeeper, client sessions, ZAB layer]
  replicated state machine
    several servers implementing the service
```

```
    operations are performed in global order
      with some exceptions, if consistency isn't important
  the replicated objects are: znodes
    hierarchy of znodes
      named by pathnames
    znodes contain *metadata* of application
      configuration information
        machines that participate in the application
        which machine is the primary
      timestamps
      version number
    types of znodes:
      regular
      empheral
      sequential: name + seqno
        If n is the new znode and p is the parent znode, then the sequence
        value of n is never smaller than the value in the name of any other
        sequential znode ever created under p.

  sessions
    clients sign into zookeeper
    session allows a client to fail-over to another Zookeeper service
      client know the term and index of last completed operation (zxid)
      send it on each request
        service performs operation only if caught up with what client has seen
    sessions can timeout
      client must refresh a session continuously
        send a heartbeat to the server (like a lease)
      ZooKeeper considers client "dead" if doesn't hear from a client
      client may keep doing its thing (e.g., network partition)
        but cannot perform other ZooKeeper ops in that session
    no analogue to this in Raft + Lab 3 KV store

Operations on znodes
  create(path, data, flags)
  delete(path, version)
      if znode.version = version, then delete
  exists(path, watch)
  getData(path, watch)
  setData(path, data, version)
    if znode.version = version, then update
  getChildren(path, watch)
  sync()
   above operations are *asynchronous*
   all operations are FIFO-ordered per client
   sync waits until all preceding operations have been "propagated"

Check: can we just do this with lab 3's KV service?
  flawed plan: GFS master on startup does Put("gfs-master", my-ip:port)
    other applications + GFS nodes do Get("gfs-master")
  problem: what if two master candidates' Put()s race?
    later Put() wins
    each presumed master needs to read the key to see if it actually is the master
      when are we assured that no delayed Put() thrashes us?
      every other client must have seen our Put() -- hard to guarantee
  problem: when master fails, who decides to remove/update the KV store entry?
    need some kind of timeout
    so master must store tuple of (my-ip:port, timestamp)
      and continuously Put() to refresh the timestamp
      others poll the entry to see if the timestamp stops changing
  lots of polling + unclear race behavior -- complex
  ZooKeeper API has a better story: watches, sessions, atomic znode creation
    + only one creation can succeed -- no Put() race
    + sessions make timeouts easy -- no need to store and refresh explicit timestamps
    + watches are lazy notifications -- avoids commiting lots of polling reads

Ordering guarantees
  all write operations are totally ordered
    if a write is performed by ZooKeeper, later writes from other clients see it
    e.g., two clients create a znode, ZooKeeper performs them in some total order
  all operations are FIFO-ordered per client
  implications:
    a read observes the result of an earlier write from the same client
    a read observes some prefix of the writes, perhaps not including most recent write
      -> read can return stale data
    if a read observes some prefix of writes, a later read observes that prefix too
```

```
Example "ready" znode:
  A failure happens
  A primary sends a stream of writes into Zookeeper
    W1...Wn C(ready)
  The final write updates ready znode
    -> all preceding writes are visible
  The final write causes the watch to go off at backup
    backup issues R(ready) R1...Rn
    however, it will observe all writes because zookeeper will delay read until
      node has seen all txn that watch observed
  Lets say failure happens during R1 .. Rn, say after return Rj to client
    primary deletes ready file -> watch goes off
    watch alert is sent to client
    client knows it must issue new R(ready) R1 ...Rn
  Nice property: high performance
    pipeline writes and reads
    can read from *any* zookeeper node

Example usage 1: slow lock
  acquire lock:
   retry:
     r = create("app/lock", "", empheral)
     if r:
       return
     else:
       getData("app/lock", watch=True)

    watch_event:
       goto retry

  release lock: (voluntarily or session timeout)
    delete("app/lock")

Example usage 2: "ticket" locks
  acquire lock:
     n = create("app/lock/request-", "", empheral|sequential)
   retry:
     requests = getChildren(l, false)
     if n is lowest znode in requests:
       return
     p = "request-%d" % n - 1
     if exists(p, watch = True)
       goto retry

    watch_event:
       goto retry

  Q: can watch_even fire before lock it is the client's turn
  A: yes
     lock/request-10 <- current lock holder
     lock/request-11 <- next one
     lock/request-12 <- my request

     if client associated with request-11 dies before it gets the lock, the
     watch even will fire but it isn't my turn yet.

Using locks
  Not straight forward: a failure may cause your lock to be revoked
    client 1 acquires lock
      starts doing its stuff
      network partitions
      zookeeper declares client 1 dead (but it isn't)
    client 2 acquires lock, but client 1 still believes it has it
      can be avoided by setting timeouts correctly
      need to disconnect client 1 session before ephemeral nodes go away
      requires session heartbeats to be replicated to majority
        N.B.: paper doesn't discuss this
  For some cases, locks are a performance optimization
    for example, client 1 has a lock on crawling some urls
    client will do it 2 now, but that is fine
  For other cases, locks are a building block
    for example, application uses it to build transaction
    the transactions are all-or-nothing
    we will see an example in the Frangipani paper

Zookeeper simplifies building applications but is not an end-to-end solution
  Plenty of hard problems left for application to deal with
```

```
  Consider using Zookeeper in GFS
    I.e., replace master with Zookeeper
  Application/GFS still needs all the other parts of GFS
    the primary/backup plan for chunks
    version numbers on chunks
    protocol for handling primary fail over
    etc.
  With Zookeeper, at least master is fault tolerant
    And, won't run into split-brain problem
    Even though it has replicated servers

Implementation overview
  Similar to lab 3 (see last lecture)
  two layers:
    ZooKeeper services  (K/V service)
    ZAB layer (Raft layer)
  Start() to insert ops in bottom layer
  Some time later ops pop out of bottom layer on each replica
    These ops are committed in the order they pop out
    on apply channel in lab 3
    the abdeliver() upcall in ZAB

Challenge: Duplicates client requests
  Scenario
    Primary receives client request, fails
    Client resends client request to new primary
  Lab 3:
    Table to detect duplicates
    Limitation: one outstanding op per client
    Problem problem: cannot pipeline client requests
  Zookeeper:
    Some ops are idempotent period
    Some ops are easy to make idempotent
      test-version-and-then-do-op
      e.g., include timestamp and version in setDataTXN

Challenge: Read operations
  Many operations are read operations
    they don't modify replicated state
  Must they go through ZAB/Raft or not?
  Can any replica execute the read op?
  Performance is slow if read ops go through Raft

Problem: read may return stale data if only master performs it
  The primary may not know that it isn't the primary anymore
    a network partition causes another node to become primary
    that partition may have processed write operations
  If the old primary serves read operations, it won't have seen those write ops
   => read returns stale data

Zookeeper solution: don't promise non-stale data (by default)
  Reads are allowed to return stale data
    Reads can be executed by any replica
    Read throughput increases as number of servers increases
    Read returns the last zxid it has seen
     So that new primary can catch up to zxid before serving the read
     Avoids reading from past
  Only sync-read() guarantees data is not stale

Sync optimization: avoid ZAB layer for sync-read
  must ensure that read observes last committed txn
  leader puts sync in queue between it and replica
    if ops ahead of in the queue commit, then leader must be leader
    otherwise, issue null transaction
  in same spirit read optimization in Raft paper
    see last par section 8 of raft paper

Performance (see table 1)
  Reads inexpensive
    Q: Why more reads as servers increase?
  Writes expensive
    Q: Why slower with increasing number of servers?
  Quick failure recovery (figure 8)
    Decent throughout even while failures happen

References:
  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.)