Purpose

* Configuration important in systems
  + operational parameters for system processes
* Zookeeper service for different coordination needs
  + an API that enables developers to implement their own primitives
* Zookeeper uses wait-free data objects organized hierarchically like a file system
* Coordination kernel
  + wait-free coordination service with relaxed consistency guarantees
* Coordination recipes
  + show how zookeeper used to build higher level coordination primitives
  + eg. blocking, strongly consistent primitives

Guarantees

* FIFO client ordering of all operations
  + clients can submit operations asynchronously
  + all requests from a given client are executed in order that they were sent by client
* linearizable writes
  + reads requests processed locally at each replica
* **Zab** =leader based atomic broadcast protocol
  + servers process reads locally so Zab not used in reads
* Clients should cache leader

Zookeeper

* clients submit requests through client API
* **client** = user of zookeeper service
* **server** = process providing zookeeper service
* **znode** = in-memory data node in zookeeper data, organized in a data tree
  + Regular type: clients manipulate by creating and deleting them explicitly
  + Ephemeral: system can remove them automatically when session that created them terminates
  + when created with sequential flag: monotonically increasing number added
* **session** = clients establish session when connect to zookeeper
  + obtain session handle
  + each session has a timeout
    - zookeeper considers client faulty if does not receive anything from session for more than timeout
  + Session ends when client closes session handle or Zookeper detects client faulty
* Zookeeper gives clients a set of znodes
  + file system path notation to find znode
* watches
  + when client issues read operation with watch flag, server promises to notify client when information returned has changed
* Not designed for general data storage
  + metadata/configuration data
* API pg 3-4
  + special points:
    - sync = apply all pending write requests before processing read
* Examples
  + new leader takes charge of a system
    - new leader can designate a path as the *ready znode*
    - other processes only use configuration when *ready znode* exists
    - new leader makes configuration change by deleting ready, updating configuration znodes, then creating ready
    - If process sees ready znode, it must see all config changes
    - What if client sees ready before new leader deletes it?
      * if client puts watch on a znode, notification happens before sees new state of system after change is made
  + Config management
    - config stored in znode zc
    - processes obtain config by reading zc with watch flag set to true
    - If config updated, processes are notified
  + Group Membership
    - designate znode zg to represent group
    - whenever process member of group starts, creates ephemeral child under zg
    - If process fails or ends, its ephemeral node is deleted
  + Locks
    - lock is znode
    - To acquire lock, client tries to create designated znode with ephemeral flag
    - If create succeeds, client holds hock
      * Otherwise, client can read znode with watch flag set
      * notified if current leader dies or deletes znode
      * Try again once that happens
      * Problem is herd effect
        + if many clients waiting to acquire lock, all vie for it at same time
    - Simple lock without herd effect
      * Lock znode L
      * All clients who want lock create ephemeral znode under L sequentially
      * To try to acquire, read children of L and if a client's sequential number is lowest, it has lock
      * Otherwise, put a watch on the znode with sequential num right before client's number
      * Now only 1 client will get a notification when lock is released
    - Read/Write Locks
      * Write lock is same as above
      * Read lock only checks if there are any write znodes before it
    - Double Barrier
      * When enough processes (barrier threshold) joined barrier, processes start computation and leave barrier once finished
      * barrier represented with znode b
      * every process creates a znode as child of b when ready to enter
      * removes when ready to leave
      * Processes can enter barrier when number of child znodes of b exceeds barrier threshold
      * processes can leave when all processes removed their children
      * Ready child created by process that causes number of children to exceed barrier threshold
* Implementation pg 8
  + replicating zookeeper data on each server
    - in memory database conataining entire data tree
    - log updates to disk, and writes are on disk before applied to in memory database
    - keep replay log of committed operations and generate snapshots periodically
  + read requests serviced from local replica
  + requests that change state of service (write requests) processed by an agreement protocol
    - forwarded to the leader
    - followers receive message proposals consisting of state changes from leader and agree upon the state changes
  + replicas never diverge (but may lag behind)
  + transactions are idempotent
  + When leader receives write req, calculates state of system when write is applied and transforms it into transaction that captures new state
  + Leader broadcasts changes though Zab
  + Zab
    - atomic broadcast protocol
    - majority quorums to decide on a proposal
    - can only work if majority of servers are correct
    - guarantees changes broadcast by a leader are delivered in the order they were sent and all changes from previous leaders are delivered to an established leader before it broadcasts its own changes
    - TCP used
    - leader chosen by Zab is the zookeeper leader
  + Snapshots
    - fuzzy snapshots
      * do not lock zookeeper state to take snapshot
      * dfs of tree atomically reading each znode's data and meta-data and writing to disk
      * May result in zookeeper applying a subset of state changes during snapshot
      * state changes are idempotent so can apply them twice as long as order is same
  + Client-Server Interactions
    - servers process writes in order and do not process other writes or reads concurrently
    - reads are processed locally
      * tagged with zxid = last transaction seen by server
        + server with less zxid will not reply to anything until caught up
      * to guarantee value returned not stale, client calls sync before read operation
        + read now reflects all changes that happened before sync was issued
    - zxid used in interactions between client and server to guarantee that server has a view at least as recent as client
    - client sends heartbeats to avoid session timeout
      * leader determines that there has been a failure if no other server receives anything from a client session within session timeout