Zookeeper

Introduction

- Key idea in the past was to try to develop primitives to solve coordination
    - For example, some services in the past focus specifically on queues, leader election, configuration, locks
- ZooKeeper attempts to move away from implementing primitives and instead give an API to enable developers to implement their own primitives
    - Doesn't use blocking primitives like locks
    - Uses wait free data objects in a hierarchy
    - Provides order guarantees for operations:
        - FIFO client ordering
            - This allows clients to issue multiple outstanding operations at a time
            - Achieves through a pipelined architecture
        - Linearizable writes
            - Uses Zab, a leader-based atomic broadcast protocol to order updates
    - Not optimized for high frequency writes
    - Allows clients to watch data and receive messages when there are updates, which allows clients to keep caches
    - Very fast though (tens of thousands of reads per second)
        - Our Raft implementations only support 10s
    - Used primarily for configuration / coordination, not storing global data
        - General idea is that this is used for storing mission critical data (i.e. configuration / progress) while non-fault-tolerant programs read from this data and attempt to carry out tasks
        - I.e. GFS coordinator uses Zookeeper to update configs and manage everything and all of the workers read from Zookeeper and update job progress
    - Different style of making distributed systems
        - Adds fault tolerance to your state rather than the entire computation (which is what Raft does)
        - Making your application with Raft requires major restructuring that can be a pain

Service Overview

- Definitions:
    - client = user of ZooKeeper
    - server = process providing ZooKeeper
    - znode = data node in ZooKeeper
    - data tree - hierarchical namespace containing znodes
    - session - clients establish these when they connect and use them to issue requests
- znodes:
    - Uses standard UNIX notation for file paths to access data tree
    - Can store data but not meant for general data storage
        - Instead, the existence of the znodes is meant to signify information
        - Some znodes in common locations can be used to carry metadata
    - Two types:
        - Regular: clients create and delete them explicitly
        - Ephemeral: clients create them and can either delete them or have them expire when session ends
    - Can have a sequential flag which attaches to their name a monotonically increasing counter
    - Watches are initiated by issuing a read with a watch flag
        - One-time triggers that are unregistered once triggered or session closes
- API:
    - `create(path, data, flags)`

    - `delete(path, version)`
    - `exists(path, watch)`
    - `getData(path, watch)`
    - `setData(path, data, version)`
    - `getChildren(path, watch)`
    - `sync(path)` - waits for all updates pending at start of operation to propagate to the server the client is connected to (path argument is ignored)
    - All of the above have synchronous and asynchronous versions
    - Update methods take a version number which can be used to perform conditional updates
        - If version is -1, then no checking
- Guarantees
    - Updates are linearizable while reads are not, so reads are done at replicas and can return data that may have been updated by other clients
    - However a client's requests are in FIFO order so they will see the writes they make
- Can be used to keep track of a server configuration:
    - When a new leader is elected, it can delete a "ready" znode, make all of its changes, and then recreate "ready"
    - Since client operations are FIFO, these will execute in order
    - This means that a process will know that a new configuration is ready iff the ready node is there
- If only a single znode is used to store configuration, we could also just use a watch to achieve the same effect
    - Watches only tell you that data is stale, not the new data, so it doesn't matter if multiple updates occur after the watch is fired because you have to read the data yourself anyway and then you can just set another watch
- Group membership can be accomplished via ephemeral nodes
- Simple locks can be implemented with ephemeral nodes
    - Herd effect can occur with simple implementation because a bunch of people are waiting for the same lock and will all fight for it the second it is deleted
        - We can get around this by having each client come in and join a queue using the sequential tag and only attempt to get the lock when it sees it is lowest in queue (by watching the person who is next lowest)
    - Read / write locks can be implemented similarly to above
        - Write locks work exactly as the herd locks do
        - Read locks will just wait until the next lowest write lock is deleted
        - This is because reads have to block until the write is done and afterwards can then be done concurrently