Chain Replication

Introduction

- Storage systems implement operations so clients can store / retrieve / change data
  - File systems: read and write operations access a single file and are idempotent (behavior is always the same)
  - Database system: operations may each access multiple objects and are serializable (can be atomic and ordered)
    - Serializable does not impose real time restrictions on operations while linearizable does
- Storage systems / services are somewhere between:
  - Store objects
  - Support queries to return value derived from single object
  - Support update to atomically change state of single object according to some computation based on prior state of object
- We want to support high availability, throughput, and strong consistency
  - Chain replication supports this
- Storage service guarantees a reply for each request it receives
  - If a client doesn't receive a reply, it re-issues a request
    - This means some requests can be ignored, but this is fine
  - Since query operations are idempotent but updates aren't, we have to be careful to make sure updates aren't performed twice

Chain Replication

- Servers are assumed to be fail-stop:
  - Each server halts in response to failure instead of making incorrect state transitions
  - A server's halted state can be detected by the environment
- If an object is replicated on t servers, $t - 1$ servers can fail without compromising availability
- The servers replicating a given object *objID* are linearly ordered to form a chain
  - First server is called the *head*
  - Last server is called the *tail*
  - The reply for each request is generated and sent by the tail
  - Each query is directed to the tail of the chain and processed there atomically
  - Each update request is directed to the head of the chain
    - Request is processed atomically and then forwarded to the next element
- Writes only respond after they have been processed all of the way to the tail
  - Reads can return as soon as possible, and they will not see any pending updates
  - This is fine because it is still linearizable
- We have a master that observes the chain and reconfigures it to eliminate failed servers
  - Informs clients which server is the head and which is the tail
  - Is assumed to never fail or can use something like Paxos or Raft
- Failures:
  - Head fails
    - Set new head to old head's successor
    - This can cause any requests sent only to the old head to be ignored, but that's fine since client can resend when it doesn't receive a response
  - Tail fails
    - Set new tail to old tail's predecessor
    - No other problems besides maybe having to send new responses that original tail didn't
  - Failure of intermediate S

        - Remove S from the chain but we need to do a bit of bookkeeping to determine if there are any requests that we need to forward from S's predecessor to S's successor
        - Each server maintains a list of update requests that have been forwarded to its successor but have not been acknowledged by the successor
            - When it receives acknowledgement from its successor that it successfully finished applying the operation, then it removes from this list
    - Handling these failures is fast and simple because the topology means we already know who is going next
- When chains get too short, we need to add to them so that they can tolerate more failures
    - Simplest to just add a server to the end of a chain to make it the new tail
    - Forwarding the new state to this server can be done in parallel with processing requests from clients

Primary / Backup Protocols

- Chain replication is a form of a p/b approach which itself is an instance of a state machine approach to replica management
- In p/b we have:
    - A primary that imposes a sequencing on client requests (to ensure strong consistency)
    - Distribution of requests from primary to backup
    - Waits for acknowledgment from backups
    - After receiving a reply from backups it replies to client
- If a primary fails, one of the backups is rpomoted
- In chain replication, the primary is split between the head and the tail
    - Faster than P/B because in CR the tail can respond to queries without waiting while in P/B we have to wait for acknowledgment from backups for prior updates before responding
- In P/B latency is parallel while in CR it is serial
- Experiments show that CR yields higher throughput than P/B
- For the longest time, these dominated data replication / paxos / raft / quorum systems
    - Recently, has begun to shift

Sharding

- If you have too much data to fit on a single replica group, you need to shard across many replica groups
- Not so great system: each group of three servers serves a single shard / chain
    - Some servers will be more loaded than others
    - The primary in each group will be slow while others have idle capacity
    - Replacing a failed replica takes a long time since we have to fetch over the network (risk other replicas failing in the meantime)
    - shard A: S1 S2 S3
      shard B: S4 S5 S6
- Better system (rndpar)
    - Split data into many more shards than servers
        - Then each server could be used in multiple shards / chains
        - shard A: S1 S2 S3
          shard B: S2 S3 S1
          shard C: S3 S1 S2
        - A server is primary in some groups, backup in other
    - Major improvements in repair speed because if a server was part of M shards, then those M chains / shards can build a new one in parallel
    - However these improvements are undermined by the fact that a few failures can wipe out all replicas of a shard