Consensus as a Service

Lecture 10
CS 739
Spring 2012

Notes from reviews

Consensus

- How does it help?
  - Strong guarantees about properties
    - Elect a master
    - Discover a master
    - Find other replicas (membership)
- How should it be exposed?
  - Client Library
  - Paxos service (acceptors)
  - Lock service

Paxos as a library

- How use?
  - Clients manage their own replicas
  - Invoke paxos to pass proposals
  - Manage an internal state machine using Paxos
    - E.g. a log, election protocol, etc.
- Benefits:
  - No extra machines
- Drawbacks:
  - Client code must be written as state machine
  - Client code must have enough replicas to run paxos (enough for majorities)
  - Clients must be reliable enough to paxos to be efficient (rare failures)
  - Must be few enough clients for paxos to be efficient
  - Clients must have a good place to store state quickly
    - Log of actions

Paxos as a service

- Run a paxos service
  - Set of acceptors to vote & record outcomes
  - Distinguished proposer/learner to accept requests & provide replies
- Benefits:
  - Unreliable clients
  - Integrate with more code
- Drawbacks:
  - Doesn't address when state can change, how it changes, who knows it changed, who can change it

Consensus as Locks

- Lock service
  - Use consensus to make strong consistency guarantees
- Benefits
  - Can protect data held elsewhere (if ordering respected)
  - Fits programmer model of locking
- Drawbacks
  - Doesn't store data on its own

Chubby

- QUESTION: What is goal?
  - Expose a consistency service for applications
- Abstraction:
  - name space of small files
  - Strongly consistent operations on files
  - Advisory locks (only enforced by lock operations, not by file operations)
  - Notifications (synchronization)

Chubby Service

- Single master + replicas,
  - Paxos for consistent leader elections
  - Operations replicated to all replicas
  - Consistency enforced at master
  - QUESTION: Why so many single masters?
    - write throughput doesn't improve with multiple masters
    - Read operations get less consistent
    - Caching makes read performance less important

Chubby Design

- Why a file system?
  - Applications that want consistency often have to store data related to consistency
    - avoids need for separate service
  - Hierarchical names space easier to manage across a cluster
- Why a service and not a library?
  - A single client can get consistency without 5 replicas for availability (chubby provides extras)
  - Locks easier to reason about that consensus as a programming model (e.g. not deal with replicated state machines)

Chubby design goals

- Extreme scalability
  - 1000's of machines, 10,000 processes connected
  - Frequent checks by client code
    - polling if something changed, accessing shared state
  - Infrequent, coarse grained lock acquisition
    - Fine grained inherently too slow/expensive/failure prone
    - Allows stronger consistency during failures
      - can allow locks to be maintained across server failures
    - Often locks held by a primary/master, only changes ownership on failure

Granularity of consensus

- Fine grained
  - Used for updating individual objects (e.g. a single file)
- Coarse grained
  - Used for rare events (e.g. electing a leader)
- Which to use?
  - Observation: can use coarse-grained to provide fine grained with lower
    - Partition fine-grained operations to different masters
    - Store partitions in Chubby
    - Detect failures using notifications
    - Try to take over for failed node
  - Observation: fine grained operations always scale poorly and perform poorly

Chubby servers

- Essentially paxos
  - Single master runs paxos to update data
  - Election protocol when master fails
- What data is replicated with paxos?
  - Data in file system
  - Locks held
- Who services request
  - Reads: use a distinguished learner (the leader)
  - Writes: run paxos

Chubby operations

- What is needed for consistency?
  - Lock acquire/release
  - Data read/write
  - Stat: has anything changed (polling)
  - Compare-and-swap: allows updates without locks
- How use for leader election:
  - All clients try to get excusive lock, only one wins, then writes name in data

Efficient locking

- Leases: a lock with a timeout
  - Works if clocks are similar
  - Server gives lock to client for a fixed period of time
  - Client must renew lock or else it goes away
  - Handles case of client / server failure automatically
    - Client failure: server reclaims lock
    - Server failure: client loses lock
  - Provides failure detection
    - Client must renew lock periodically
  - Provides piggy-back opportunity for other messages
    - Attach other messages to renew message
- How handle server failure:
  - Would like to keep lock across server failure
  - Solution: grace period
    - Allows lock to be held but not used while server restarts
    - If get new lease before grace period over, don't need to release it

Scaling servers

- How reduce load from client checking polling or checking on things?
  - Sessions: aggregate all client state to piggyback all messages at once
  - Container for what goes away on failure
  - Enables caching, because client guaranteed to receive invalidations when session is alive
- Keep-alives provide fast failure notification

Chubby consistency mechanism

- Problem: use chubby to sequence operations to some other service (e.g. not chubby files)
  - may have reordering of chubby lock/unlock with operations in other service under failure
    - e.g. client grabs lock, issues request, then fails
    - Another client grabs lock, issues another request
      - first request could arrive after second
- Solution: sequencer; evidence that a lock is held
  - client grabs lock, gets sequencer, issues request, fails
  - next client grabs lock, gets sequencer issues request, fials
    - server verifies sequencer of first client, chubby says lock no loner held & reject

Event notification

- Think: like condition variables
  - Clients can be notified of useful events to avoid polling
    - file changed
    - file added to directory (perhaps representing new replica)
    - Lock acquired – new primary elected
  - Events delivered after the fact
    - state may not be true any more
    - Guaranteed to not see old state

Chubby Caching

- How do you cache?
  - Leases: record how long you can use a lock without contacting server
    - removes read requests
  - Aggregate with *sessions*: renew all leases on a session at a time
    - renew requests O(1) not O(cache size)
  - Negative caching: cache when open() fails
    - removes polling for non-existent files
  - Open files
    - repeatedly opening doesn't re-open file at server
  - Locks
    - don't release locks when done, but allow chubby to reclaim
- Result: traffic at Chubby server largely mutations
  - who owns lock
  - changes of file data

Failover

- What happens when a master fails?
  - Other servers detect
  - Elect one as leader (paxos…)
  - Clients learn of new master from DNS or someplace like that
  - Clients send keep alives to new server to establish session
    - Must provide lock handles

Consensus as a service

- Challenges
  - Caching failures can overwhelm servers
  - Client response to server failure
    - Why would clients restart on server failure?