Consensus Services for Loosely-coupled distributed Systems

Zookeeper and Chubby
Motivation

• Distributed systems need a service that provides
  – Synchronization (who is the leader?)
  – Membership (who are active nodes in my service?)
  – Configuration metadata (what are environment info?)

• Distributed systems need a service that is
  – reliability
  – availability
  – easy-to-understand semantics
  – performance, throughput, latency only secondary
Before Chubby Came About…

• Lots of distributed systems with clients in the 10,000s

• How to do primary election?
  – Ad hoc (no harm from duplicated work)
  – Operator intervention (correctness essential)

• Unprincipled
  • Disorganized
  • Costly
  • Low availability
What is this paper about?

“Building Chubby was an engineering effort ... it was not research. We claim no new algorithms or techniques. The purpose of this paper is to describe what we did and why, rather than to advocate it.”

• Design based on well-known ideas
  – distributed consensus, caching, notifications, file-system interface
Life Before Cubby

• Distributed systems developers..
  – Implement Raft (well actually Paxos)
    • Application must be written as a state machine
    • Potential performance problems
      – Quorum on 5 is easier over quorum of 10K nodes
  – Shared critical regions (Exclusive locks)
    • Hard to code/understand
      – People think they can … but they can’t!
Chubby Design
Design Decisions: Motivating Locks?

• Lock service vs. consensus (Raft/Paxos) library

• Advantages:
  – No need to rewrite code
    • Maintain program structure, communication patterns
    • Can support notification mechanism
  – Smaller # of nodes (servers) needed to make progress

• Advisory instead of mandatory locks (why?):
  – Holding a lock called F neither is necessary to access the file F, nor prevents other clients from doing so
Design Decisions: Lock Types

• Coarse vs. fine-grained locks
  – Fine-grained: grab lock before every event
  – Coarse-grained: grab lock for large group of events

• Advantages of coarse-grained locks
  – Less load on lock server
  – Less delay when lock server fails
  – Less lock servers and availability required

• Advantages of fine-grained locks
  – More lock server load
  – If needed, could be implemented on client side
System Structure

- Chubby cell: a small number of replicas (e.g., 5)
- Master is selected using a consensus protocol (e.g., Raft)
System Structure

• Clients
  – Send reads/writes only to the master
  – Communicates with master via a chubby library

• Every replica server
  – Is listed in DNS
  – Direct clients to master
  – Maintain copies of a simple database
Read and Writes

• **Write**
  
  – Master propagates write to replica
  
  – Replies after the write reaches a majority (e.g., quorum)

• **Read**
  
  – Master replies directly, as it has most up to date state
  
  – Reads must still go to the master
Chubby API and Locks
Simple UNIX-like File System Interface

• Bare bone file & directory structure

• `/ls/foo/wombat/pouch`

- Cell name
- Name within cell
- Lock service; common to all names
Simple UNIX-like File System Interface

• Bare bone file & directory structure

• /ls/foo/wombat/pouch

• Does not support, maintain, or reveal
  – Moving files
  – Path-dependent permission semantics
  – Directory modified times, files last-access times
Nodes

• Node: a file or directory
  – Any node can act as an advisory reader/writer lock

• A node may be either permanent or ephemeral
  – Ephemeral used as temporary files, e.g., indicate a client is alive

• Metadata
  – Three names of ACLs (R/W/change ACL name)
    • Authentication build into ROC
  – 64-bit file content checksum
Locks

• Any node can act as lock (shared or exclusive)

• Advisory (vs. mandatory)
  – Protect resources at remote services
  – No value in extra guards by mandatory locks

• Write permission needed to acquire
  – Prevents unprivileged reader blocking progress
Locks and Sequences

• Potential lock problems in distributed systems
  – A holds a lock L, issues request W, then fails
  – B acquires L (because A fails), performs actions
  – W arrives (out-of-order) after B’s actions

• Solution 1: backward compatible
  – Lock server will prevent other clients from getting the lock if a lock become inaccessible or the holder has failed
  – Lock-delay period can be specified by clients
Locks and Sequences

- Potential lock problems in distributed systems
  - A holds a lock L, issues request W, then fails
  - B acquires L (because A fails), performs actions
  - W arrives (out-of-order) after B’s actions

- Solution 2: sequencer
  - A lock holder can obtain a sequencer from Chubby
  - It attaches the sequencer to any requests that it sends to other servers
  - The other servers can verify the sequencer information
Design: Events

• Client subscribes when creating handle

• Delivered async via up-call from client library

• Event types
  – File contents modified
  – Child node added / removed / modified
  – Chubby master failed over
  – Handle / lock have become invalid
  – Lock acquired / conflicting lock request (rarely used)
Design: API

• Open() (only call using named node)
  – how handle will be used (access checks here)
  – events to subscribe to
  – lock-delay
  – whether new file/dir should be created

• Close() vs. Poison()

• Other ops:
  – GetContentsAndStat(), SetContents(), Delete(), Acquire(), TryAcquire(), Release(),
    GetSequencer(), SetSequencer(), CheckSequencer()
Chubby: Providing Performance
Why scaling is important?

• Clients connect to a single instance of master in a cell
  – Much more client processes that number of machines
  – Note: Master and client have same CPU/Memory

• Existing mechanisms:
  – **Partition**: More Chubby cells(consistent hashing)
  – **Increase lease time**: from 12s to 60s to reduce KeepAlive messages
    (dominant requests in experiments)
  – **Client caches**: reduces reads/keepalive not writes
  – **Add new type of servers**: Add Proxy servers
Caching

- Client caches file data, node meta-data
  - Write-through held in memory

- Master keeps list of what clients may have cached

- Strict consistency: easy to understand
  - Lease based
  - Master will invalidate cached copies upon a write request
  - Do not want to alter preexisting comm. protocols

- Handles and locks cached as well
  - Event informs client of conflicting lock request
Caching and Invalidation

- writes block, master sends invalidations
- clients flush changed data, ack. with KeepAlive
- data *uncachable* until invalidation acked
  - allows reads to happen without delay
New Mechanisms

• Proxies:
  – Handle KeepAlive and read requests, pass write requests to the master
  – Reduce traffic but reduce availability

• Partitioning: partition name space
  – A master handles nodes with hash(name) mod N == id
  – Limited cross-partition messages
Scaling: Proxies

• Proxies pass requests from clients to cell
  – A layer of middle management between cell and clients

• Can handle KeepAlives and reads **NOT WRITES**
  – Not writes, but they are $<< 1\%$ of workload

• KeepAlive traffic by far most dominant

• Disadvantages:
  – additional RPC for writes / first time reads
  – increased unavailability probability
  – fail-over strategy not ideal (will come back to this)
Scaling: Partitioning

- Namespace partitioned between servers
  - N partitions, each with master and replicas

- Little cross-partition comm. desirable
  - permission checks
  - directory deletion
  - caching helps mitigate this
Chubby: Providing Availability
Sessions and Keep-Alives

• A client sends keep-alive requests to a master

• A master responds by a keep-alive response

• Immediately after getting the keep-alive response, the client sends another request for extension

• The master will block keep-alives until close the expiration of a session

• Extension is default to 12s
When things fail...

• Failure == missing keep alive msgs

• Server
  – Delete ephemeral files w/o open handles after an interval
  – Delete associated cache

• Client: discard in memory state
  – Sessions, handles, locks, cached data ...
Locks and Asynch Failures

- Asynch failure: delayed, ordered or lost messages

- Two solutions
  - Sequence #s: get Lock and a Sequence-ID (seq-#)
    - Submit requests with Lock-Seq-ID
    - Leader checkers Lock-seq-ID to make sure request is current
    - Lock-seq-ID is incremented when a new lock is acquired
  - LockDelay: delay giving out new locks after the lock holder dies
    - Outstanding requests should arrive within that time period
Chubby: Practical, warstories
Use and Observations

- Many files for naming
- Config, ACL, meta-data common
- 10 clients use each cached file, on avg.
- Few locks held, no shared locks
- KeepAlives dominate RPC traffic

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>time since last fail-over</td>
<td>18 days</td>
</tr>
<tr>
<td>fail-over duration</td>
<td>14s</td>
</tr>
<tr>
<td>active clients (direct)</td>
<td>22k</td>
</tr>
<tr>
<td>additional proxied clients</td>
<td>32k</td>
</tr>
<tr>
<td>files open</td>
<td>12k</td>
</tr>
<tr>
<td>naming-related</td>
<td>60%</td>
</tr>
<tr>
<td>client-is-caching-file entries</td>
<td>230k</td>
</tr>
<tr>
<td>distinct files cached</td>
<td>24k</td>
</tr>
<tr>
<td>names negatively cached</td>
<td>32k</td>
</tr>
<tr>
<td>exclusive locks</td>
<td>1k</td>
</tr>
<tr>
<td>shared locks</td>
<td>0</td>
</tr>
<tr>
<td>stored directories</td>
<td>8k</td>
</tr>
<tr>
<td>ephemeral</td>
<td>0.1%</td>
</tr>
<tr>
<td>stored files</td>
<td>22k</td>
</tr>
<tr>
<td>0-1k bytes</td>
<td>90%</td>
</tr>
<tr>
<td>1k-10k bytes</td>
<td>10%</td>
</tr>
<tr>
<td>&gt; 10k bytes</td>
<td>0.2%</td>
</tr>
<tr>
<td>naming-related</td>
<td>46%</td>
</tr>
<tr>
<td>mirrored ACLs &amp; config info</td>
<td>27%</td>
</tr>
<tr>
<td>GFS and Bigtable meta-data</td>
<td>11%</td>
</tr>
<tr>
<td>ephemeral</td>
<td>3%</td>
</tr>
<tr>
<td>RPC rate</td>
<td>1-2k/s</td>
</tr>
<tr>
<td>KeepAlive</td>
<td>93%</td>
</tr>
<tr>
<td>GetStat</td>
<td>2%</td>
</tr>
<tr>
<td>Open</td>
<td>1%</td>
</tr>
<tr>
<td>CreateSession</td>
<td>1%</td>
</tr>
<tr>
<td>GetContentsAndStat</td>
<td>0.4%</td>
</tr>
<tr>
<td>SetContents</td>
<td>680ppm</td>
</tr>
<tr>
<td>Acquire</td>
<td>31ppm</td>
</tr>
</tbody>
</table>
Use: Outages

• Sample of cells
  – 61 outages over few weeks (700 cell-days)
  – due to network congestion, maintenance, overload, errors in software, hardware, operators

• 52 outages under 30s
  – applications not significantly affected

• Few dozen cell-years of operation
  – data lost on 6 occasions (bugs & operator error)
Today

• Google’s Chubby
  – Motivation
  – Design choices
  – Scaling/Performance
  – Availability

• Yahoo’s ZooKeeper (Now Apache’s Zookeeper)
  – Differences with Chubby

• Summary
Zookeeper = Chubby without locks

• File system-based API
  – Similar types: Ephemeral, persistent, sequential
  – Different API calls

• Performance
  – Caching and watches (ZK’s invalidation)
    • Similarly one shot!

• Availability
  – Keepalive, leader election
## File Types

<table>
<thead>
<tr>
<th>File types</th>
<th>Chubby</th>
<th>ZooKeeper</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ephemeral</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Permanent</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Sequential</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>File API</td>
<td>Hierarchical</td>
<td>Yes</td>
</tr>
<tr>
<td>Predefined path</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

- Ephemeral: the znode will be deleted when the session that created it times out or it is explicitly deleted.
- Permanent: explicitly deleted by client.
- Sequence: the path name will have a monotonically increasing counter relative to the parent appended.

```
/ <-- Ephemerals created by Session X
  ↓
  services
  ↓
  YaView
  ↓
  servers
  ↓
  stupidname
  ↓
  morestupidity
  ↓
  locks
  ↓
  s-1
  ↓
  apps
  ↓
  users
```
Filesystem API

/ls/foo/wombat/pouch

Cell name

Name within cell

Lock service; common to all names

services

YaView

servers

stupidname

morestupidity

locks

read-1

apps

users
Read/Write Interactions

- **Writes**
  - All go through leader
  - Requires quorum

- **Reads**
  - ZK: go to any node
    - Higher performance
  - Chubby: go to leader
    - Performance limitation
  - Both: Clients cache and server invalidates cache
    - Chubby: invalidation is non-optional
    - ZK: must explicitly register for invalidation requests