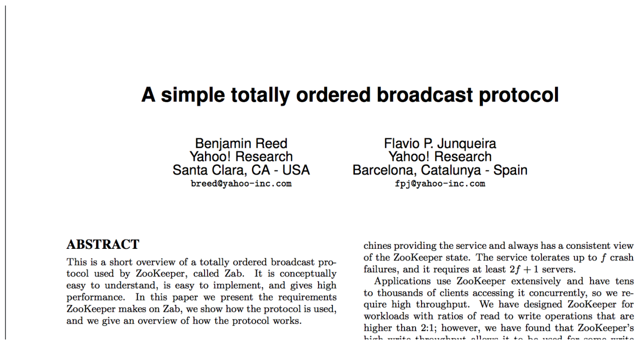
Distributed Systems

Day 19: Practical Consensus

The Part-Time Parliament

Leslie Lamport

This article appeared in *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133-169. Minor corrections were made on 29 August 2000.



A simple totally ordered broadcast protocol

Benjamin Reed
Yahoo! Research
Santa Clara, CA - USA
breed@yahoo-inc.com

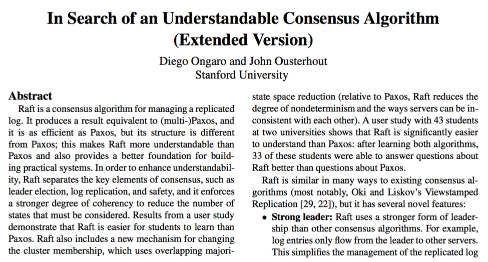
Flavio P. Junqueira
Yahoo! Research
Barcelona, Catalunya - Spain
fpj@yahoo-inc.com

ABSTRACT

This is a short overview of a totally ordered broadcast protocol used by ZooKeeper, called Zab. It is conceptually easy to understand, is easy to implement, and gives high performance. In this paper we present the requirements ZooKeeper makes on Zab, we show how the protocol is used, and we give an overview of how the protocol works.

chines providing the service and always has a consistent view of the ZooKeeper state. The service tolerates up to f crash failures, and it requires at least $2f + 1$ servers.

Applications use ZooKeeper extensively and have tens to thousands of clients accessing it concurrently, so we require high throughput. We have designed ZooKeeper for workloads with ratios of read to write operations that are higher than 2:1; however, we have found that ZooKeeper's



In Search of an Understandable Consensus Algorithm (Extended Version)

Diego Ongaro and John Ousterhout
Stanford University

Abstract

Raft is a consensus algorithm for managing a replicated log. It produces a result equivalent to (multi-)Paxos, and it is as efficient as Paxos, but its structure is different from Paxos; this makes Raft more understandable than Paxos and also provides a better foundation for building practical systems. In order to enhance understandability, Raft separates the key elements of consensus, such as leader election, log replication, and safety, and it enforces a stronger degree of coherency to reduce the number of states that must be considered. Results from a user study demonstrate that Raft is easier for students to learn than Paxos. Raft also includes a new mechanism for changing the cluster membership, which uses overlapping majority state space reduction (relative to Paxos, Raft reduces the degree of nondeterminism and the ways servers can be inconsistent with each other). A user study with 43 students at two universities shows that Raft is significantly easier to understand than Paxos: after learning both algorithms, 33 of these students were able to answer questions about Raft better than questions about Paxos.

Raft is similar in many ways to existing consensus algorithms (most notably, Oki and Liskov's Viewstamped Replication [29, 22]), but it has several novel features:

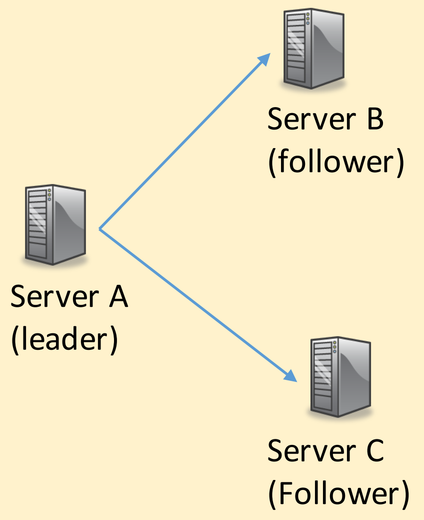
- **Strong leader:** Raft uses a stronger form of leadership than other consensus algorithms. For example, log entries only flow from the leader to other servers. This simplifies the management of the replicated log

1998
Paxos

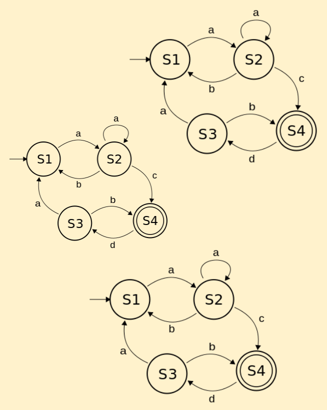
2008
ZAB

2014
Raft

Consensus in Practice

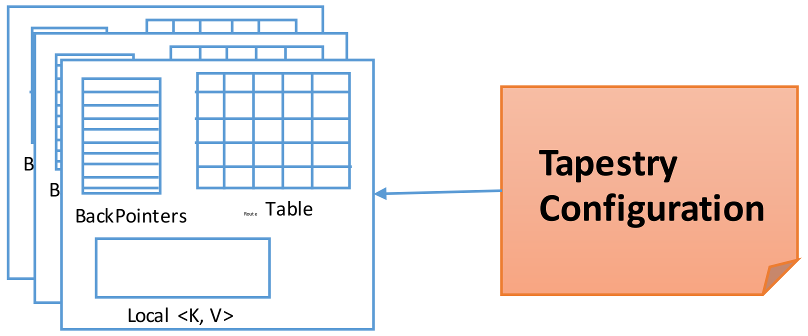


Consistent Data



Consistent Applications

Thus far, we've used consistency for application
Or all the application's data

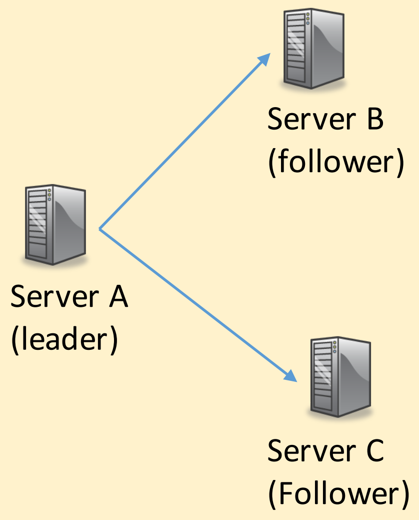


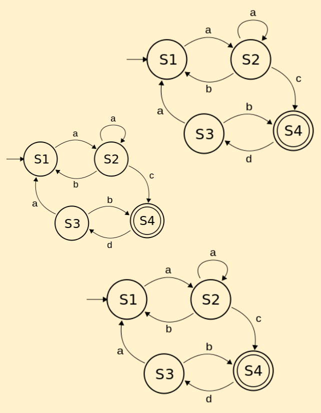
Tapestry Network

But metadata should
Be consistent

The data and the storage
Doesn't need to be consisten

Consensus in Practice



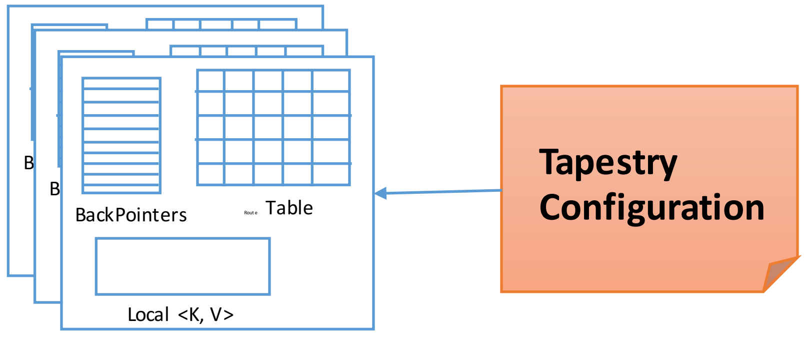


Consistent Data

Consistent Applications

Thus far, we've used consistency for application
Or all the application's data

Group Membership	Who is in my Tapestry cluster?
Configuration MetaData	What is the IP of the LiteMiner Master? How many nodes in Raft?
Distributed Locks	Who has locks on a file?



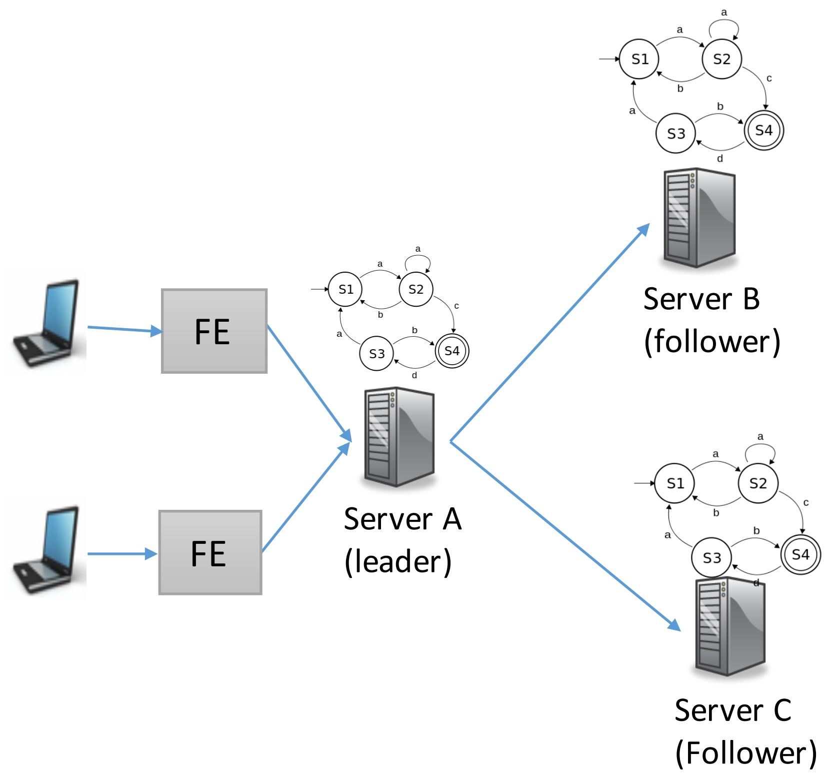
But metadata should
Be consistent

Tapestry Network

The data and the storage
Doesn't need to be consisten

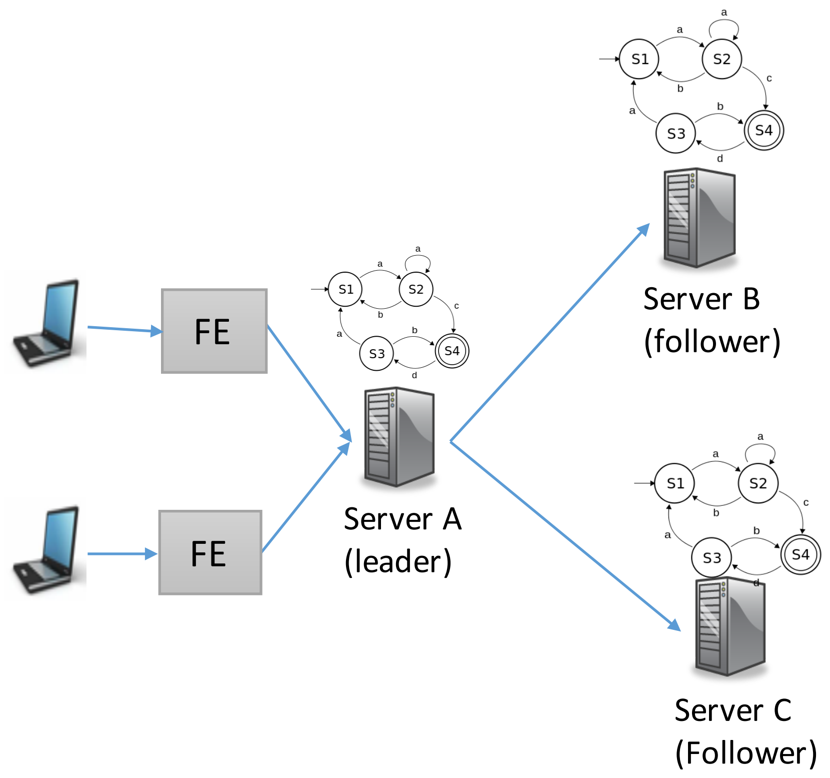
How would you implement a lock with Raft?

- Primitives exposed to the FEs
  - Lock()
  - Unlock()



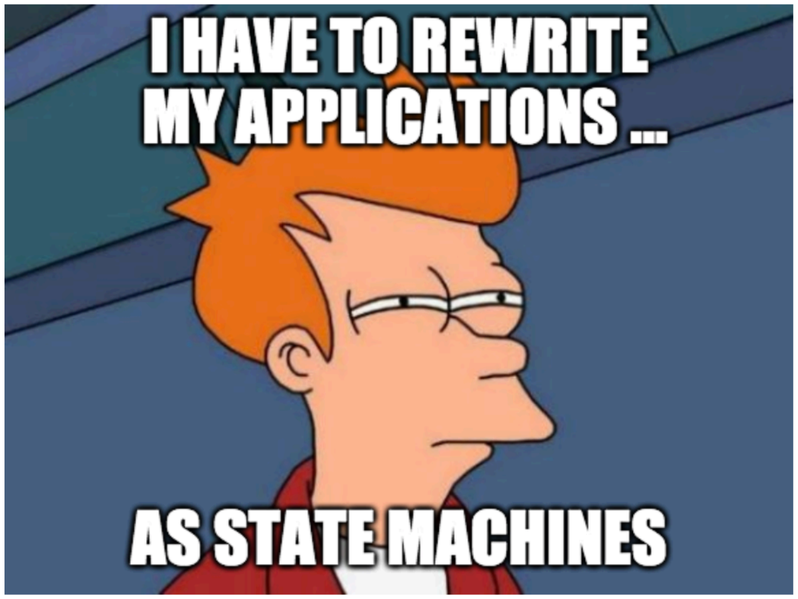
How would you implement a lock with Raft?

- Primitives exposed to the FEs
  - Lock()
  - Unlock()
- Challenges:
  - Locks must be implemented as a state machine
  - Must understand log-replication semantics



Life Before Cubby was far worse…

- 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!

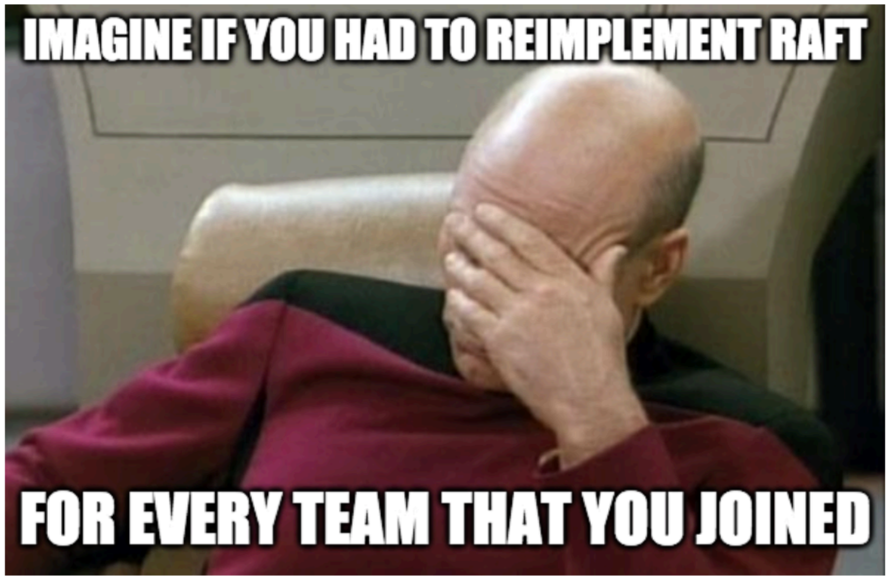


Before Chubby Came About…

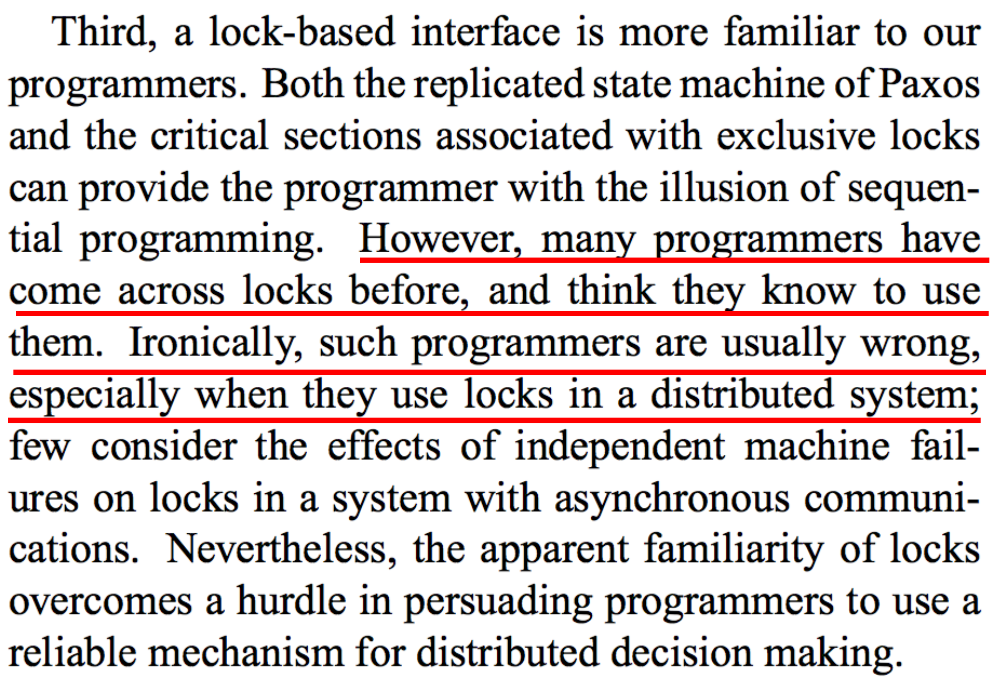
- Lots of distributed systems with clients in the 10,000s

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

- Unprincipled
  - Disorganized
  - Costly
  - Low availability


IMAGINE IF YOU HAD TO REIMPLEMENT RAFT
FOR EVERY TEAM THAT YOU JOINED

Which would you program with Locks? Or Raft?

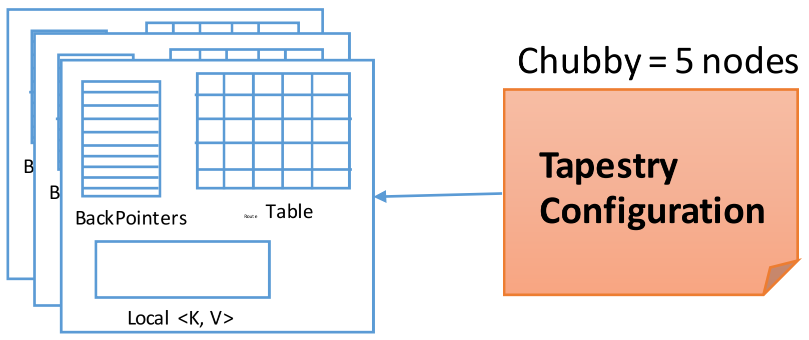


Third, a lock-based interface is more familiar to our programmers. Both the replicated state machine of Paxos and the critical sections associated with exclusive locks can provide the programmer with the illusion of sequential programming. However, many programmers have come across locks before, and think they know to use them. Ironically, such programmers are usually wrong, especially when they use locks in a distributed system; few consider the effects of independent machine failures on locks in a system with asynchronous communications. Nevertheless, the apparent familiarity of locks overcomes a hurdle in persuading programmers to use a reliable mechanism for distributed decision making.

Design requirements:

- Expose locks to developers
    - Locks are easier than redesigning as statemachines
- Locks can't be permanent
    - If locks are permanent and servers fail then locks are lost
- All servers in network should not be part of the service

Chubby = 5 nodes



Tapestry = 10000 nodes

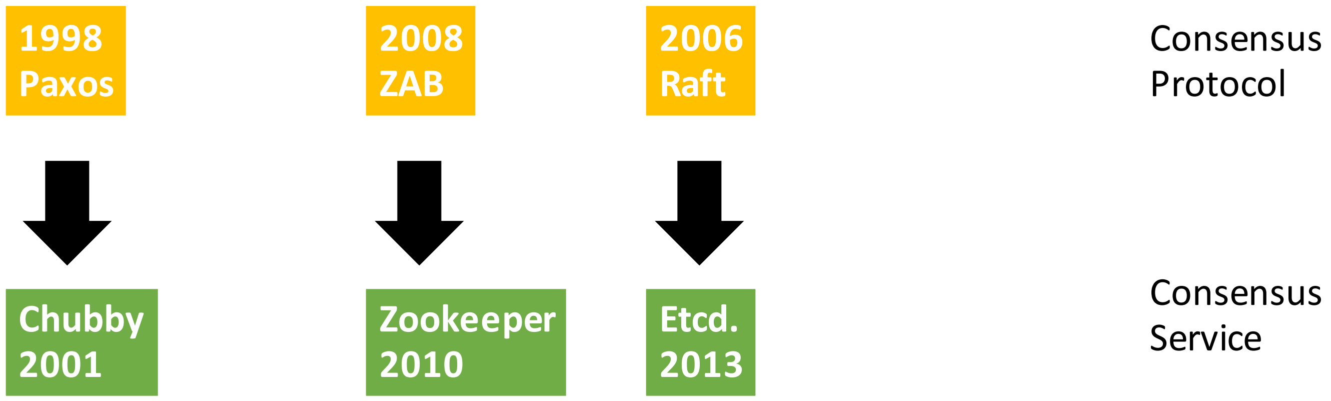
suggests that an event notification mechanism would be useful to avoid polling.

- Even if clients need not poll files periodically, many will; this is a consequence of supporting many developers. Thus, caching of files is desirable.
- Our developers are confused by non-intuitive caching semantics, so we prefer consistent caching.
- To avoid both financial loss and jail time, we provide security mechanisms, including access control.

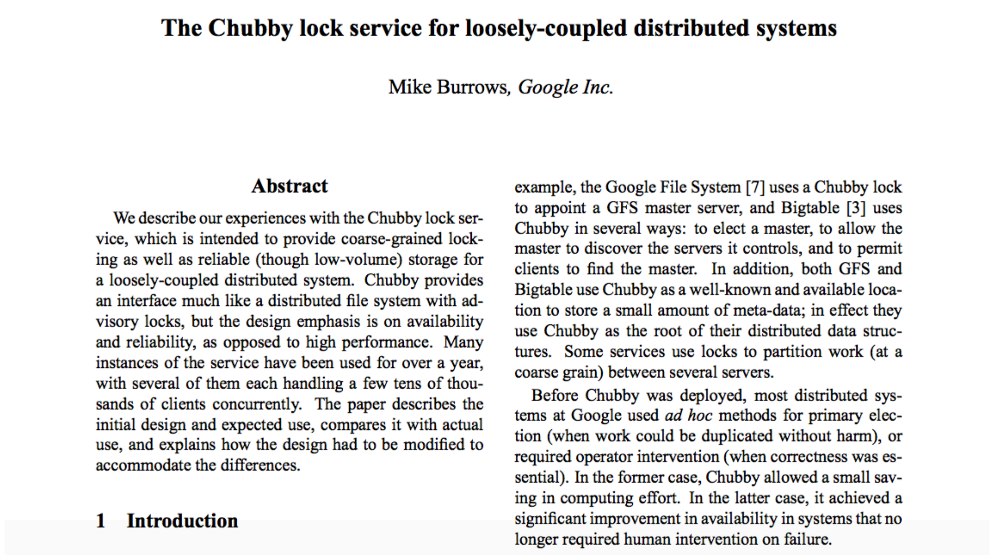
What is the Chubby Paper about?

"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 of consensus service based on well-known ideas
  - distributed consensus, caching, notifications, file-system interface



Chubby Design



The Chubby lock service for loosely-coupled distributed systems

Mike Burrows, *Google Inc.*

Abstract

We describe our experiences with the Chubby lock service, which is intended to provide coarse-grained locking as well as reliable (though low-volume) storage for a loosely-coupled distributed system. Chubby provides an interface much like a distributed file system with advisory locks, but the design emphasis is on availability and reliability, as opposed to high performance. Many instances of the service have been used for over a year, with several of them each handling a few tens of thousands of clients concurrently. The paper describes the initial design and expected use, compares it with actual use, and explains how the design had to be modified to accommodate the differences.

1 Introduction

example, the Google File System [7] uses a Chubby lock to appoint a GFS master server, and Bigtable [3] uses Chubby in several ways: to elect a master, to allow the master to discover the servers it controls, and to permit clients to find the master. In addition, both GFS and Bigtable use Chubby as a well-known and available location to store a small amount of meta-data; in effect they use Chubby as the root of their distributed data structures. Some services use locks to partition work (at a coarse grain) between several servers.

Before Chubby was deployed, most distributed systems at Google used *ad hoc* methods for primary election (when work could be duplicated without harm), or required operator intervention (when correctness was essential). In the former case, Chubby allowed a small saving in computing effort. In the latter case, it achieved a significant improvement in availability in systems that no longer required human intervention on failure.

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

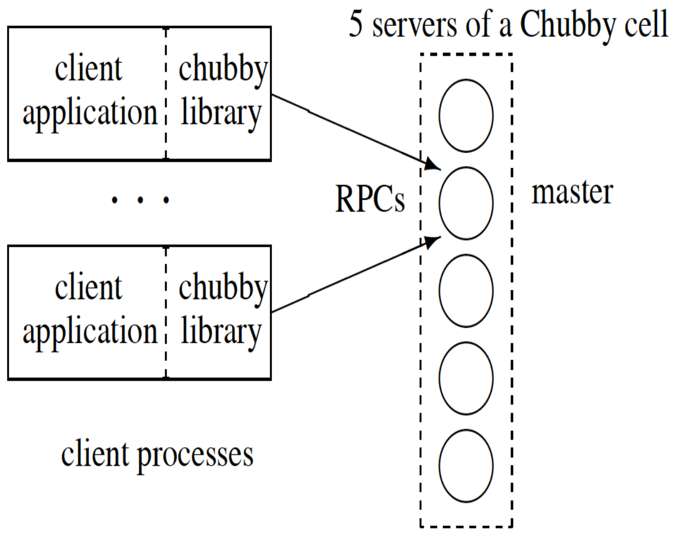
- Coarse vs. fine-grained locks
  - Fine-grained: grab lock before every event
  - Coarse-grained: grab lock for large group of events

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

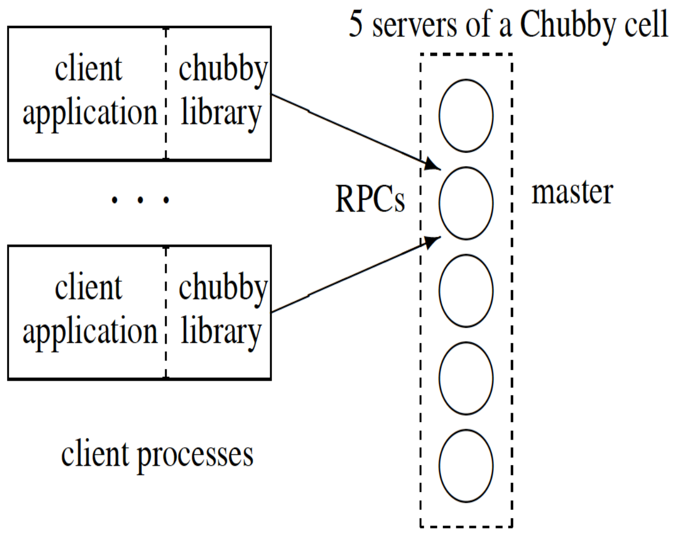
- Chubby cell: a small number of replicas (e.g., 5)
- Master is selected using a consensus protocol (e.g., Raft)



System Structure

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

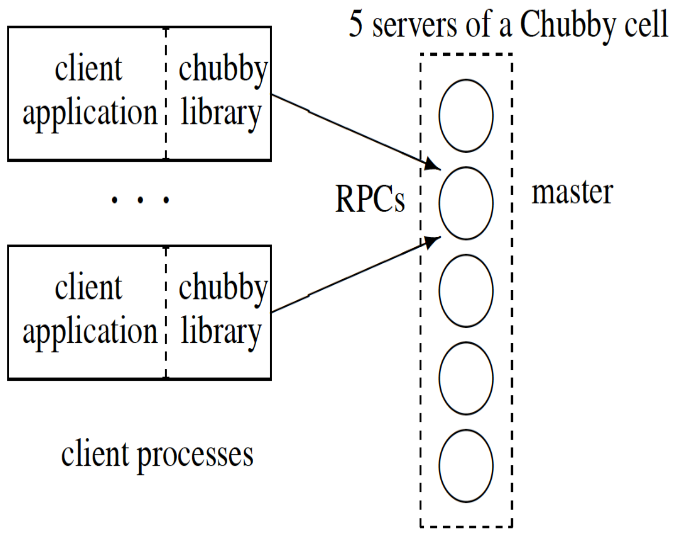
- Every replica server
  - Is listed in DNS
  - Direct clients to master
  - Maintain copies of a simple database



Read and Writes

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

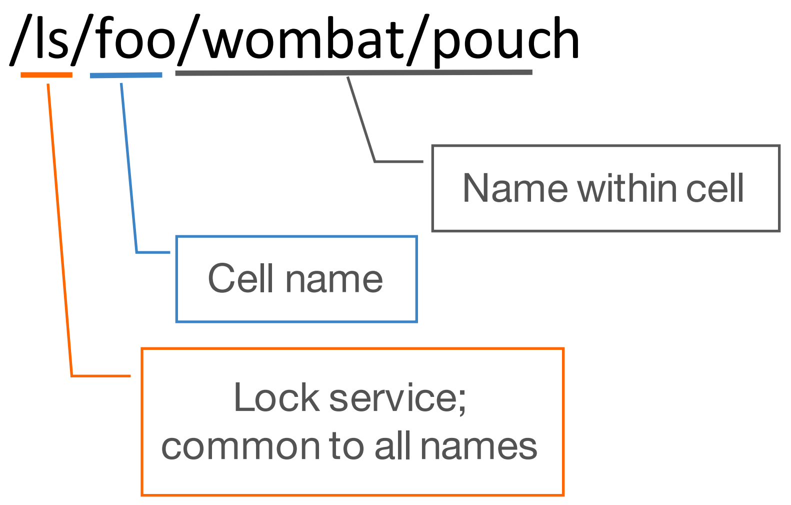
- Read
  - Master replies directly, as it has most up to date state
  - Reads must still go to the master



Chubby API and Locks

Simple UNIX-like File System Interface

- Bare bone file & directory structure
- /ls/foo/wombat/pouch



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