CSE 486/586 Distributed Systems
Google Chubby Lock Service

Steve Ko
Computer Sciences and Engineering
University at Buffalo

Recap

- Paxos is a consensus algorithm.
  - Proposers?
  - Acceptors?
  - Learners?
- A proposer always makes sure that,
  - If a value has been chosen, it always proposes the same value.
- Three phases
  - Prepare: "What's the last proposed value?"
  - Accept: "Accept my proposal."
  - Learn: "Let's tell other guys about the consensus."

Recap: First Requirement

- In the absence of failure or msg loss, we want a value to be chosen even if only one value is proposed by a single proposer.
- *P1. An acceptor must accept the first proposal that it receives.*

Recap: Second Requirement

- But the first requirement is not enough!
  - There are cases that do not provide any consensus.
- We need to accept multiple proposals.
- Then we need to guarantee that once a majority chooses a value, all majorities should choose the same value.
  - I.e., all chosen proposals have the same value.
  - This guarantees only one value to be chosen.
  - This gives our next requirement.
- *P2. If a proposal with value V is chosen, then every higher-numbered proposal that is chosen has value V.*

Recap: Strengthening P2

- OK; how do we guarantee that?
- Can acceptors do something?
  - Yes!
- So we can strengthen P2:
- *P2a. If a proposal with value V is chosen, then every higher-numbered proposal accepted by any acceptor has value V.*
- By doing this, *we have change the requirement to be something that acceptors need to guarantee.*

Recap: Strengthening P2

- But guaranteeing P2a might be difficult because of P1.
- Scenario
  - A value V is chosen.
  - An acceptor C never receives any proposal (due to asynchrony).
  - A proposer fails, recovers, and issues a different proposal with a higher number and a different value.
  - C accepts it (violating P2a).

Recap: Combining P1 & P2a

- Then can proposers do anything about that?
- *P2b. If a proposal with value V is chosen, then every higher-numbered proposal issued by any proposer has value V.*
- Now we have changed the requirement P2 to *something that each proposer has to guarantee*.

How to Guarantee P2b

- *P2b. If a proposal with value v is chosen, then every higher-numbered proposal issued by any proposer has value V.*
- Two cases for a proposer proposing (N, V)
  - If a proposer knows that there is and will be no proposal N' < N chosen by a majority, it can propose any value.
  - If that is not the case, then it has to make sure that it proposes the same value that's been chosen by a majority.
- (Rough) Intuition for the first case
  - If there's a proposal chosen by a majority set S, then any majority set S' will intersect with S.
  - Thus, if the proposer asks acceptors and gets replies from a majority that it *did not and will not* accept any proposal, then we're fine.

"Invariant" to Maintain

- P2c. *For any V and N, if a proposal with value V and number N is issued, then there is a set S consisting of a majority of acceptors such that either*
  - *(A) no acceptor in S has accepted or will accept any proposal numbered less than N or,*
  - *(B) V is the value of the highest-numbered proposal among all proposals numbered less than N accepted by the acceptors in S.*

Paxos Phase 1

- A proposer chooses its proposal number N and sends a *prepare request* to acceptors.
- Maintains P2c.
- Acceptors need to reply:
  - A promise to not accept any proposal numbered less than N any more (to make sure that the protocol doesn't deal with old proposals)
  - If there is, the accepted proposal with the highest number less than N

Paxos Phase 2

- If a proposer receives a reply from a majority, it sends an *accept request* with the proposal (N, V).
  - V: the highest N from the replies (i.e., the accepted proposals returned from acceptors in phase 1)
  - Or, if no accepted proposal was returned in phase 1, any value.
- Upon receiving (N, V), acceptors need to maintain P2c by either:
  - Accepting it
  - Or, rejecting it if there was another prepare request with N' higher than N, and it replied to it.

Paxos Phase 3

- Learners need to know which value has been chosen.
- Many possibilities
- One way: have each acceptor respond to all learners
  - Might be effective, but expensive
- Another way: elect a "distinguished learner"
  - Acceptors respond with their acceptances to this process
  - This distinguished learner informs other learners.
  - Failure-prone
- Mixing the two: a set of distinguished learners

Problem: Progress (Liveness)

- *There's a race condition for proposals.*
- P0 completes phase 1 with a proposal number N0
- Before P0 starts phase 2, P1 starts and completes phase 1 with a proposal number N1 > N0.
- P0 performs phase 2, acceptors reject.
- Before P1 starts phase 2, P0 restarts and completes phase 1 with a proposal number N2 > N1.
- P1 performs phase 2, acceptors reject.
- …(this can go on forever)
- How to solve this?

Providing Liveness

- Solution: elect a distinguished proposer
  - I.e., have only one proposer
- If the distinguished proposer can successfully communicate with a majority, the protocol guarantees liveness.
  - I.e., if a process plays all three roles, Paxos can tolerate failures *f* < 1/2 * *N*.
- Still needs to get around FLP for the leader election, e.g., having a failure detector

CSE 486/586 Administrivia

- More practice problems & example final posted
- Quick poll: Android platform class
- PhoneLab hiring
  - Testbed developer/administrator
- Anonymous feedback form still available.
- Please come talk to me!

Google Chubby

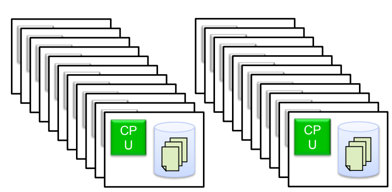
- A lock service
  - Enables multiple clients to share a lock and coordinate
- A coarse-grained lock service
  - Locks are supposed to be held for hours and days, not seconds.
- In addition, it can store small files.
- Design target
  - Low-rate locking/unlocking
  - Low-volume information storage
- Why would you need something like this?

Google Infrastructure Overview

- Google File System (GFS)
  - Distributed file system
- Bigtable
  - Table-based storage
- MapReduce
  - Programming paradigm & its execution framework
- These rely on Chubby.
- Warning: the next few slides are intentionally shallow.
  - The only purpose is to give some overview.

Google File System

- A cluster file system
  - Lots of storage (~12 disks per machine)
  - Replication of files to combat failures



Google File System

- Files are divided into chunks
  - 64MB/chunk
  - Distributed & replicated over servers
- Two entities
  - *One master*
  - Chunk servers

Google File System

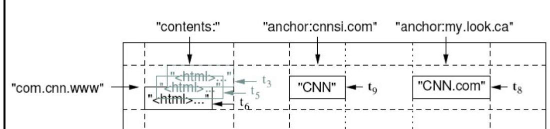
- Master maintains all file system metadata
  - Namespace
  - Access control info
  - Filename to chunks mappings
  - Current locations of chunks
- Master replicates its data for fault tolerance
- Master periodically communicates with all chunk servers
  - Via heartbeat messages
  - To get state and send commands
- Chunk servers respond to read/write requests & master's commands.

Bigtable

- Table-based storage on top of GFS
- Main storage for a lot of Google services
  - Google Analytics
  - Google Finance
  - Personalized search
  - Google Earth & Google Maps
  - Etc.
- Gives a large logical table view to the clients
  - Logical tables are divided into *tablets* and distributed over the Bigtable servers.
- Three entities
  - Client library
  - *One master*
  - Tablet servers

Bigtable

- Table: rows & columns
  - *(row, column, timestamp) -> cell contents*
- E.g., web pages and relevant info.
  - Rows: URLs
  - Columns: actual web page, (out-going) links, (incoming) links, etc.
  - Versioned: using timestamps

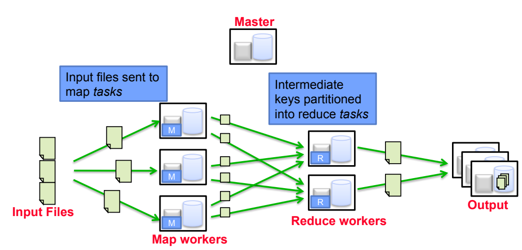


MapReduce

- Programming paradigm
  - Map: (key, value) → list of (intermediate key, intermediate value)
  - Reduce: (intermediate key, list of intermediate values) → (output key, output value)
  - Programmers write Map & Reduce functions within the interface given (above).
- Execution framework
  - Google MapReduce executes Map & Reduce functions over a cluster of servers
  - *One master*
  - Workers

MapReduce

- Execution flow



Common Theme

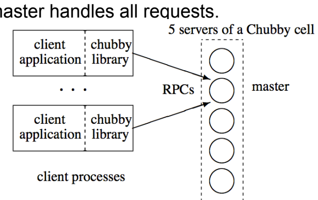
- One master & multiple workers
- Why one master?
  - This design simplifies lots of things.
  - Mainly used to handle meta data; it's important to reduce the load of a single master.
  - No need to deal with consistency issues
  - Mostly fit in the memory → very fast access
- Obvious problem: failure
  - We can have one primary and backups.
  - We can then elect the primary out of the peers.
- How would you use a lock service like Chubby?

Chubby

- A coarse-grained lock service
  - Locks are supposed to be held for hours and days, not seconds.
  - In addition, it can store small files.
- Used for various purposes (e.g., the master election) for GFS, Bigtable, MapReduce
  - Potential masters try to create a lock on Chubby
  - The first one that gets the lock becomes the master
- Also used for storing small configuration data and access control lists

Chubby Organization

- Chubby cell (an instance) has typically 5 replicas.
  - But each cell still serves tens of thousands of clients
- Among 5 replicas, one master is elected.
  - Any one replica can be the master.
  - They decide who is the master via Paxos.
- The master handles all requests.



Client Interface

- File system interface
  - From a client's point of view, it's almost like accessing a file system.
- Typical name: /ls/foo/wombat/pouch
  - ls (lock service) common to all Chubby names
  - foo is the name of the Chubby cell
  - /wombat/pouch interpreted within Chubby cell
- Contains files and directories, called *nodes*
  - Any node can be a reader-writer lock: reader (shared) mode & writer (exclusive) mode
  - Files can contain a small piece of information
  - Just like a file system, each file is associated with some meta-data, such as access control lists.

Client-Chubby Interaction

- Clients (library) send *KeepAlive* messages
  - Periodic handshakes
  - If Chubby doesn't hear back from a client, it's considered to be failed.
- Clients can subscribed to *events*.
  - E.g., File contents modified, child node added, removed, or modified, lock become invalid, etc.
- Clients *cache data* (file & meta data)
  - If the cached data becomes stale, the Chubby master invalidates it.
- They Chubby master *piggybacks events or cache invalidations on the KeepAlives*
  - Ensures clients keep cache consistent

Client Lock Usage

- Each lock has a *"sequencer"* that is roughly a version number.
- Scenario
  - A process holding a lock L issues a request R
  - It then fails & lock gets freed.
  - Another process acquires L and perform some action before R arrives at Chubby.
  - R may be acted on without the protection of L, and potentially on inconsistent data.

Client API

- open() & close()
- GetContentsAndStat()
  - Reads the whole file and meta-data
- SetContents()
  - Writes to the file
- Acquire(), TryAcquire(), Release()
  - Acquires and releases a lock associated with the file
- GetSequencer(), SetSequencer(), CheckSequencer()

Primary Election Example

- All potential primaries open the lock file and attempt to acquire the lock.
- One succeeds and becomes the primary, others become replicas.
- Primary writes identity into the lock file with SetContents().
- Clients and replicas read the lock file with GetContentsAndStat().
- In response to a file-modification event.

Chubby Usage

- A snapshot of a Chubby cell

time since last fail-over	18 days
fail-over duration	14s
active clients (direct)	22k
additional proxied clients	32k
files open	12k
naming-related	60%
client-is-caching-file entries	230k
distinct files cached	24k
names negatively cached	32k
exclusive locks	1k
shared locks	0
stored directories	8k
ephemeral	0.1%

stored files	22k
0-1k bytes	90%
1k-10k bytes	10%
> 10k bytes	0.2%
naming-related	46%
mirrored ACLs & config info	27%
GFS and Bigtable meta-data	11%
ephemeral	3%
RPC rate	1-2k/s
KeepAlive	93%
GetStat	2%
Open	1%
CreateSession	1%
GetContentsAndStat	0.4%
SetContents	680ppm
Acquire	31ppm

- Few clients hold locks, and shared locks are rare.
  - Consistent with locking being used for primary election and partitioning data among replicas.

Acknowledgements

- These slides contain material developed and copyrighted by Indranil Gupta (UIUC).