Distributed Locking Service : Chubby, Introduction

Goal :

Build a highly available & consistent service that can stores small objects and provide locking mechanism over the objects.

Research Paper

Chubby_Credit_Google_Inventor

Use cases :

It started with the requirement of reliable locking service.

Few standard places where it is used widely across the distributed domain are

- Leader/Master election
- Naming Service (DNS)
- Distributed locking mechanism
- Storage (small objects that rarely changed

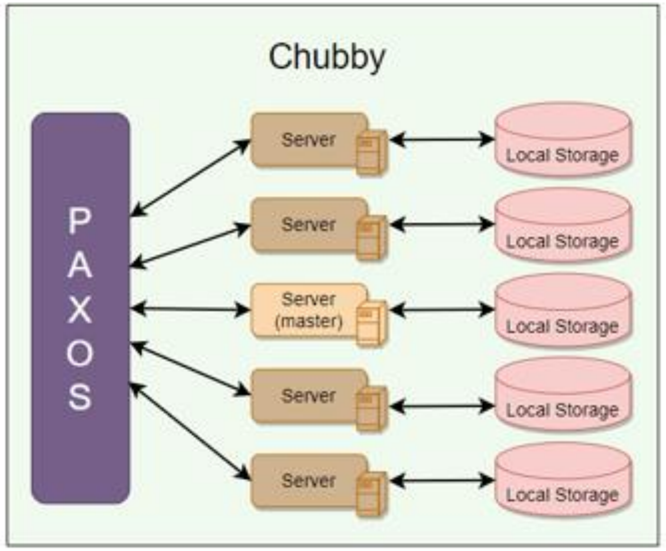
How has it structured?

Internally, it is implemented as a key/value store that also provides a locking mechanism on each object stored in it.

Who build it?

Google had built this system as utility to support their different service like **GFS**, **Bigtable** etc.

Another open-source alternative to Chubby is **Apache Zookeeper**.

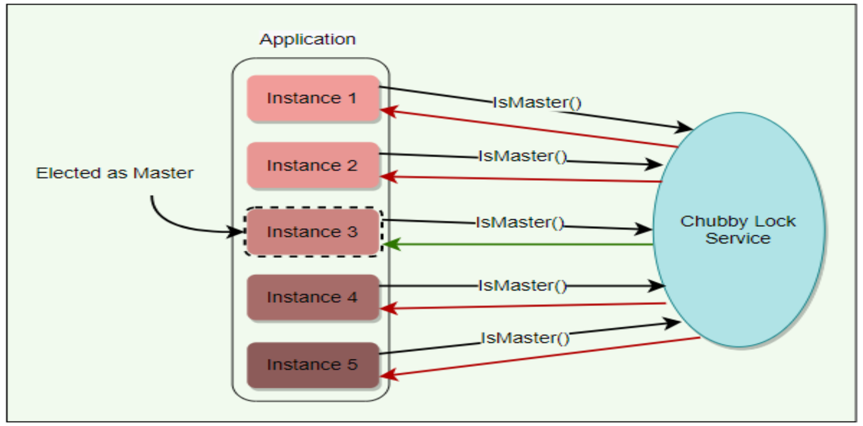


Distributed Locking Service : Chubby, Use-cases with more info

Leader/Master election :

Any lock service can be seen as a consensus service, as it converts the problem of reaching consensus to handing out locks.

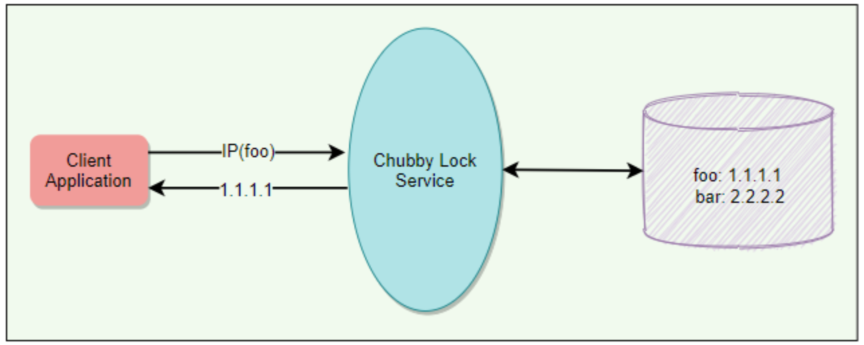
an application can have multiple replicas running and wants one of them to be chosen as the leader. Chubby can be used for leader election among a set of replicas, e.g., the leader/master for GFS and BigTable.



Naming Service (like DNS)

Let's understand how Google has used Chubby instead of DNS (Name Service) for discovering the service inside Google.

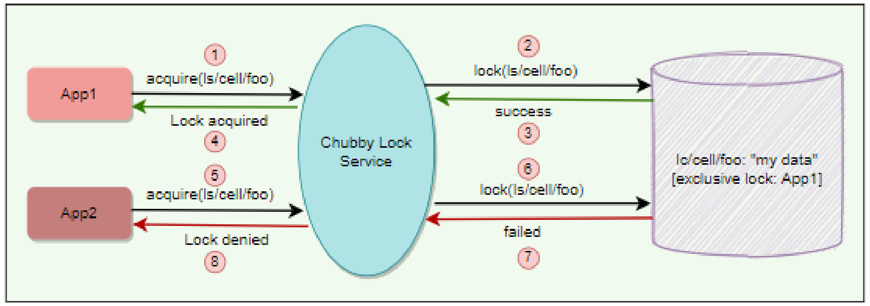
DNS service can't be updated too frequently because of design of time-based caching update. So, to handle this situation and Google's internal requirement of frequent update required in discovery, google start using Chubby for Naming Service.



Distributed locking mechanism

Chubby provides a developer-friendly interface for coarse-grained distributed locks (as opposed to fine-grained locks) to synchronize distributed activities in a distributed environment.

In other words, we can say that Chubby provides mechanisms like semaphores and mutexes for a distributed environment.



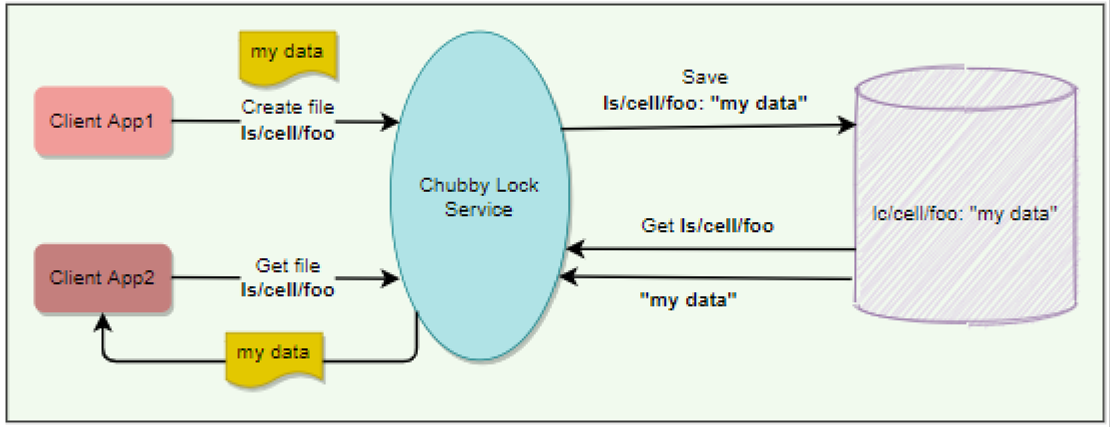
Distributed Locking Service : Chubby, Use-cases with more info

Storage (small objects that rarely change) :

Chubby provides a Unix-style interface to reliably store small files that do not change frequently (Complementing the service offered by GFS, meaning GFS is for Large files and Chubby could be used for small files). Applications can then use these files for any usage like DNS, configs, etc.

GFS and Bigtable store their metadata in Chubby.

Some services use Chubby to store ACL files (Access Control List)



How Chubby started (Background)

Chubby describes a certain design and implementation done at Google in order to provide a way for its clients to synchronize their activities and agree on basic information about their environment.

At a high level, Chubby provides a framework for distributed consensus.

So, as you know, Google using Paxos for consensus, and so our next point of discussion will be how chubby and Paxos are mixing.

Distributed Locking Service : Chubby & Paxos

Paxos

Paxos plays a major role inside Chubby.

Readers familiar with Distributed Computing recognize that getting all nodes in a distributed system to agree on anything (e.g., election of primary among peers) is basically a kind of distributed consensus problem.

Distributed consensus using Asynchronous Communication is already solved by **Paxos protocol**, and Chubby uses ***Paxos underneath to manage the state of the Chubby system*** at any point in time.

Let's Start understanding of High-level Architecture?

Ok, let's learn some of the common terminology used inside the Chubby Architecture.

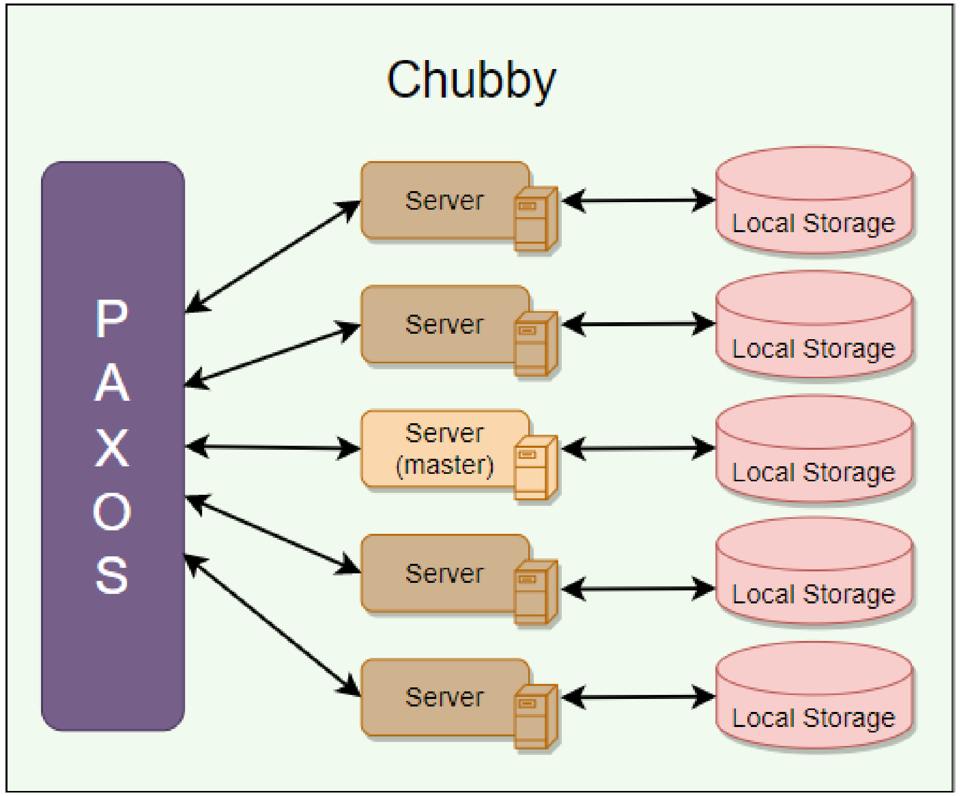
Chubby cell

Chubby cell is basically a cluster. As per initial thought of an architect, Chubby cell mostly confined within the data center. But later for introducing better Fault tolerant to the system, they decide to replicate across the multiple geography.

A Single chubby cell have two major components (Server, Client), and they will communicate via RPC (Remote Procedure Call).

Chubby servers

- A chubby cell consists of a small set of servers (typically 5) known as replicas.
- Using Paxos, one of the servers is chosen as the master who handles all client requests. If the master fails, another server from replicas becomes the master.
- Each replica maintains a small database to store files/directories/locks. The master writes directly to its own local database, which gets synced asynchronously to all the replicas.
- For fault tolerance, Chubby replicas are placed on different racks.



Distributed Locking Service : Chubby Architecture – Common terminology

Chubby Client Library

Client applications use a Chubby library to communicate with the replicas in the chubby cell using RPC.

Chubby APIs

Chubby exports a file system interface like POSIX but simpler.
It is represented by hierarchical data structure like Tree, and its structure mentioned below…

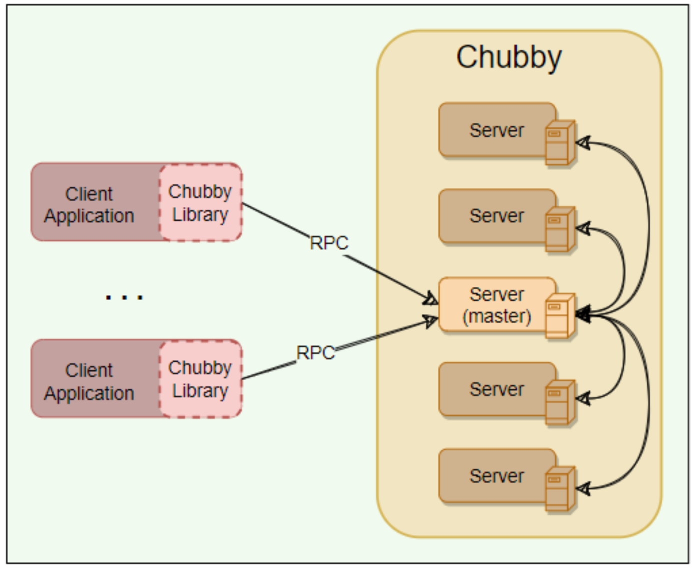
It consists of a strict tree of files and directories in the usual way, with name components separated by slashes.

File format: /ls/chubby_cell/directory_name/.../file_name

Chubby was originally designed as a lock service, such that every entity in it will be a lock.
Later, its creators realized that it is useful to associate a small amount of data with each entity.
So, as of now, each entity in Chubby can be used for **locking** or storing **a small amount of data** or **both**, i.e., storing small files with locks.

Majorly Chubby APIs are divided into four major category

- *General*
- *File*
- *Locking*
- *Sequencer*



Index	APIs Category	APIs Name	APIs Brief Information
1	General	Open	Opens a given named file or directory and returns a handle.
		Close	Close an open handle
		Poison	Allows a client to cancel all Chubby calls made by other threads
		Delete	Deletes the file or directories
2	File	GetContentsAndStat	Returns (atomically) the whole file contents and metadata associated with the file. ***But discraged to use it***
		GetStat	Returns just the metadata
		ReadDir	Returns the contents of a directory - names and metadata of all childrens
		Setcontents	Writes the whole contents of a file (atomically).
		SetACL	Write new Access Controller list information
3	Locking	Acquire	Acquires a lock on a file.
		TryAcquire	Tries to acquire a lock on a file; it is a non-blocking variant of Acquire.
		Release	Releases a lock.
4	Sequencer	GetSequencer	Get the sequencer of a lock. A sequencer is a string representation of a lock.
		SetSequencer	Associate a sequencer with a handle.
		CheckSequencer	Check whether a sequencer is valid.

Distributed Locking Service : Chubby Architecture – Important Design decisions

Chubby built as a service, Why?

One primary reason for creating the chubby was to create Paxos distributed consensus.
So, for handling this situation there two possibilities here

- ✓ Create client library
- ✓ Create as a service

So, the next point here is to understand that What are the advantage of creating service over library?

- **Lock based interface is developer friendly**

To provide the atomicity in operation, we should have some consensus state which is managed in system. This is the requirement. Now, if by chance we will try this management at client end than it means every client must manage Paxos state locally at real time which is very difficult task in distributed system, so easier way is to manage Lock service with some simple interface like Acquire(), TryAcquire() and Release().

- **Provide quorum & replica management**

Distributed consensus algorithms need a quorum to decide
Since, we have multiple replicas to produce high availability so managing quorum at client end
In this structure is very complex structure to deal. So, to handle this situation it is better you handle this on server itself through master server.

- **Broadcast functionality**

Clients and replicas of a replicated service may wish to know when the service's master changes;
This requires an event notification mechanism.
Such a mechanism is easy to build if there is a central service in the system.

- **Development becomes easy**

In all above method we realized that Chubby as a service helps more in development instead of Chubby as a client Library.

The arguments above clearly show that building and maintaining a central locking service abstracts away and takes care of a lot of complex problems from client applications.

Distributed Locking Service : Chubby Architecture – Important Design decisions

Coarse-grained locks, Why?

I remember, I read the intention of inventor from start was not design this for Fine grained locking mechanism.

These locks should not be used in situations where the locks are held for a short period of time.

When electing a leader, for example, Chubby locks will be ideal, since this isn't a frequent event.

Reason behind this approach in design are mentioned below

- **Less load on lock server**

Coarse-grained locks impose far less load on the server as the lock acquisition rate is a lot less than the client's transaction rate.

- **Survive server failures**

As coarse-grained locks are acquired rarely, clients are not significantly delayed by the temporary unavailability of the lock server.

Why advisory locks?

Chubby locks are advisory, which means it is up to the application to honor the lock. Chubby doesn't make locked objects inaccessible to clients not holding their locks. It is more like record keeping and allows the lock requester to discover that lock is held.

Chubby gave following reasons for not having mandatory locks:

✓ To enforce mandatory locking on resources implemented by other services would require more extensive modification of these services.

✓ Mandatory locks prevent users from accessing a locked file for debugging or administrative purposes. If a file must be accessed, an entire application would need to be shut down or rebooted to break the mandatory lock.

✓ Generally, a good developer practice is to write assertions such as assert("Lock X is held"), so mandatory locks bring only little benefit anyway.

Chubby in terms of CAP theorem?

So, overall, Chubby provides guaranteed consistency and a very high level of availability in the common case.

Why Chubby needs storage?

Chubby's storage is important as client applications may need to advertise Chubby's results with others. For example, an application needs to store some info to:

✓ Advertise its selected primary (leader election use case)
✓ Resolve aliases to absolute addresses (naming service use case)
✓ Announce the scheme after repartitioning of data.

Not having a separate service for sharing the results reduces the number of servers that clients depend on. Chubby's storage requirements are simple. i.e., store a small amount of data (KBs) and limited operation support (i.e., create/delete).

Why does Chubby export a Unix-like file system interface?

Chubby exports a file system interface like Unix but simpler. It consists of a strict tree of files and directories in the usual way, with name components separated by slashes.

File format: /ls/cell/remainder-path

Distributed Locking Service : Brief Info about Chubby Working

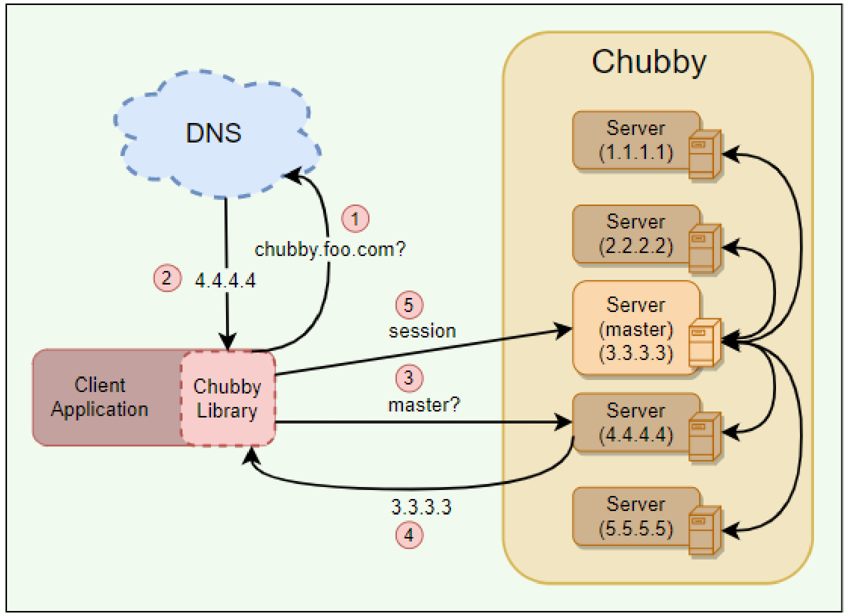
Chubby Working?

Service Initialization

Upon init there are two steps which will be done
1. Using Paxos protocol, a master is getting selected
2. The information about the master is persisted in storage must shared among the all-other replicas

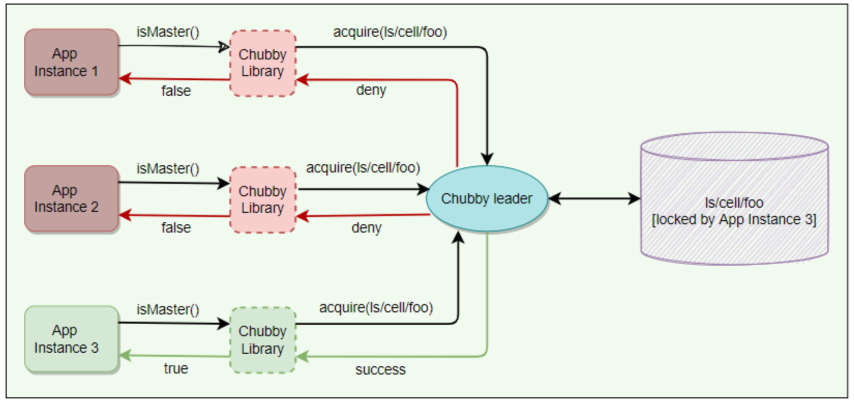
Client Initialization

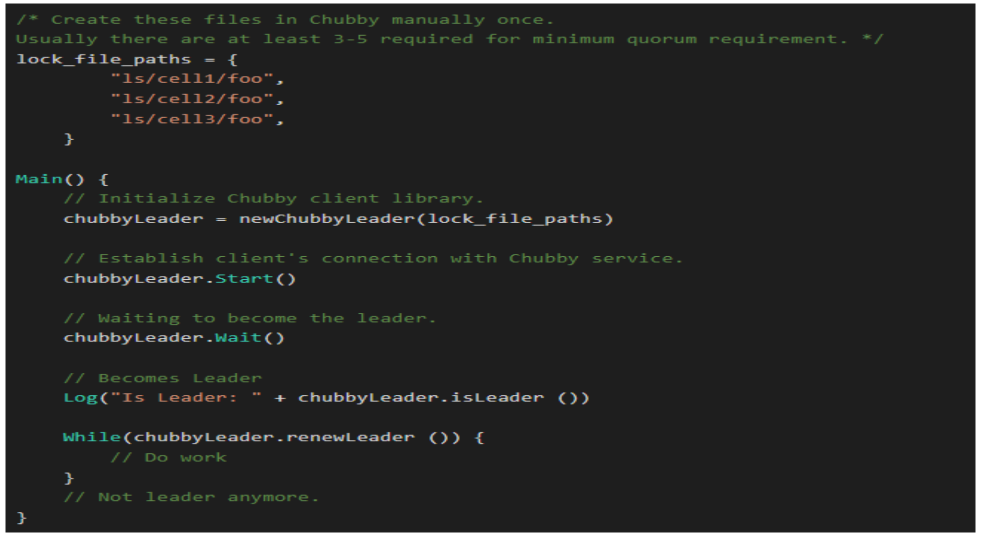
Upon init of client, following steps are performed
- ✓ Client contacts the DNS to know the client replicas
- ✓ Client calls any chubby server directly using RPC
- ✓ Chubby server will return the current master server if the current server is not the master server
- ✓ Also, client cache the master information and used this until the master server became the nonresponsive.



Leader election example using Chubby

Once the master election starts, all candidates attempt to acquire a Chubby lock on a file associated with the election. Whoever acquires the lock first becomes the master. The master writes its identity on the file, so that other processes know who the is current master.





```
/* Create these files in Chubby manually once.
Usually there are at least 3-5 required for minimum quorum requirement. */
lock_file_paths = {
        "ls/cell1/foo",
        "ls/cell2/foo",
        "ls/cell3/foo",
    }

Main() {
    // Initialize Chubby client library.
    chubbyLeader = newChubbyLeader(lock_file_paths)

    // Establish client's connection with Chubby service.
    chubbyLeader.Start()

    // Waiting to become the leader.
    chubbyLeader.Wait()

    // Becomes Leader
    Log("Is Leader: " + chubbyLeader.isLeader ())

    While(chubbyLeader.renewLeader ()) {
        // Do work
    }
    // Not leader anymore.
}
```

Chubby Working : File, Directories and Handles

Chubby Working?

Chubby file system interface is basically a tree of files and directories, where each directory contains a list of child files and directories. Each file or directory is called a node.

Nodes

✓Any node can act as an advisory reader/writer lock.
✓Node could be even ephemeral or permanent

Metadata

Metadata for each node includes ACL(Access Control Lists), four monotonically increasing 64-bit numbers, and a checksum.

Monotonically increasing 64-bit numbers: These numbers allow clients to detect changes easily.

- ✓ An instance number: This is greater than the instance number of any previous node with the same name.
- ✓ A content generation number (files only): This is incremented every time a file's contents are written.
- ✓ A lock generation number: This is incremented when the node's lock transitions from free to held.
- ✓ An ACL generation number: This is incremented when the node's ACL names are written.

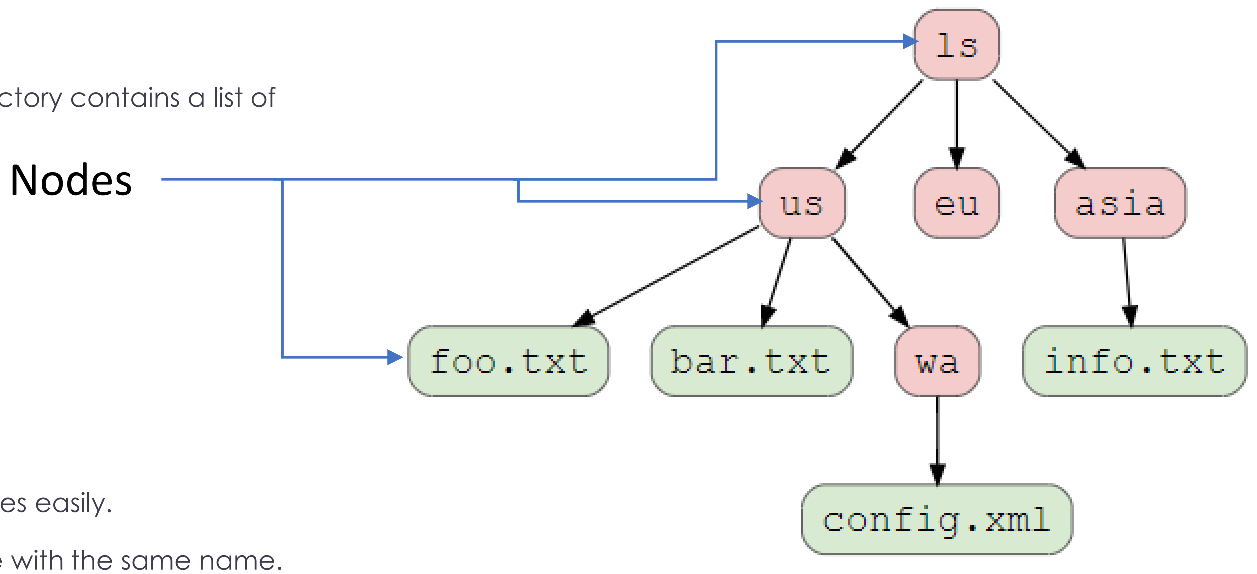
Checksum: Chubby exposes a 64-bit file-content checksum so clients may tell whether files differ.

Handles

Client's open nodes to obtain handles (that are analogous to UNIX file descriptors).
This is sharing some information regarding the handles to current master.

This handle have three parts

- ✓ Check digits: Prevent clients from creating or guessing handles, so full access control checks are performed only when handles are created.
- ✓ A sequence number: Enables a master to tell whether a handle was generated by it or by a previous master.
- ✓ Mode information (provided at open time): This mode information is an instruction to current master regarding the recreation of state is required or not?



An ephemeral **node that will disappear when the session of its owner ends**.

Chubby Working : Locks, Sequencers & Lock-delays

Locks

Locks are the way chubby node can work in this system.
There are two different approach of Locking required for Write/Read mode.

For Write, Exclusive lock is required so that writing would happen atomically.
For Reading, Sharing lock is suffice, because this time, system don't have any atomicity requirement.

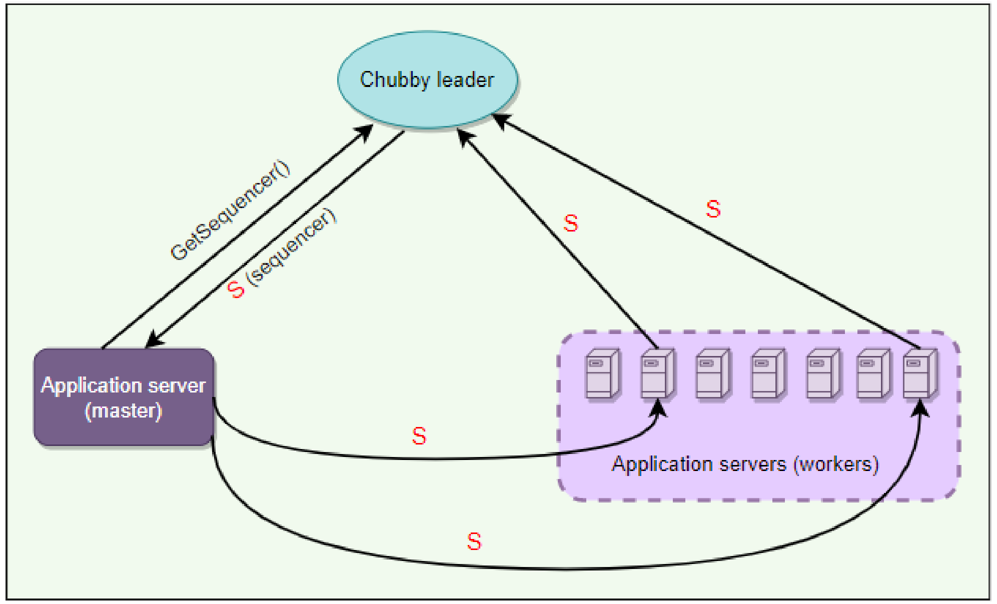
So, different locking mechanism is required at different time and chubby system is providing the same to do the needful.

Sequencer

Sequencer = **Name of the lock** + Lock mode (exclusive or shared) + **Lock generation number**

An application's master server can generate a sequencer and send it with any internal order to other servers.

Application servers that receive orders from a primary can check with Chubby if the sequencer is still good and does not belong to a stale primary (to handle the 'Brain split' scenario).



Lock-delay

If a client releases a lock in the normal way, it is immediately available for other clients to claim, as one would expect. However, if a lock becomes free because the holder has failed or become inaccessible, the lock server will prevent other clients from claiming the lock for a period called the lock-delay.

- ✓ Clients may specify any lock-delay up to some bound, defaults to one minute.
- ✓ lock-delay protects unmodified servers and clients from everyday problems caused by message delays and restarts.

An ephemeral **node that will disappear when the session of its owner ends**.

Chubby Working : Sessions & Events

What is a chubby sessions?

A Chubby session is a connection between a Chubby cell and a Chubby client.

- ✓ It is for the defined interval time, by maintaining the periodic handshaking
- ✓ Client's handles, locks & cached data only valid until the session is alive

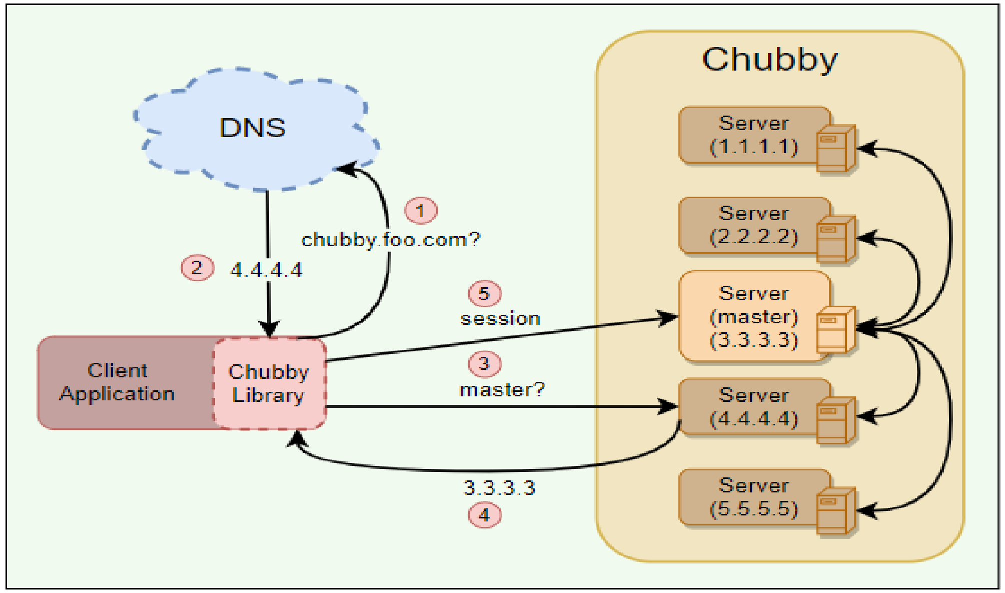
Session Protocol

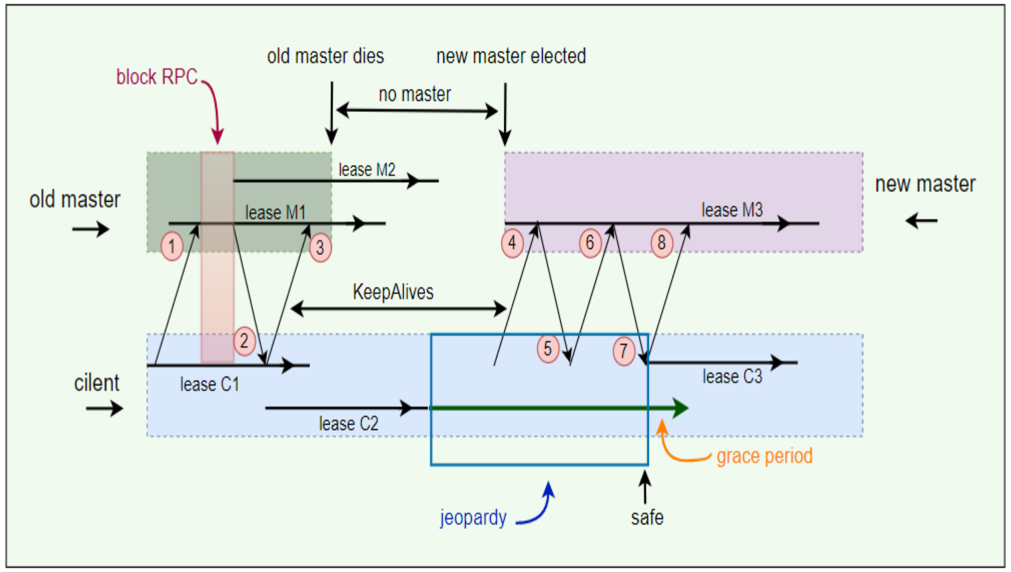
- Client requests a new session on first contacting the master of Chubby cell.
- Each session has an associate lease, which is a time interval during which the master guarantees to not terminate the session unilaterally. The end of this interval is called 'session lease timeout.'
- The master advances the 'session lease timeout' in the following three circumstances:
  - When the master responds to a KeepAlive RPC from the client
  - When a master failover occurs
  - On session creation

KeepAlive

Purpose?
To maintain the constant session between client and chubby cell.
Following are the basic steps of responding to a KeepAlive.

- On receiving a KeepAlive (step "1" in the diagram below), the master typically blocks the RPC (does not allow it to return) until the client's previous lease interval is close to expiring.
- The master later allows the RPC to return to the client (step "2") and thus informs the client of the new lease timeout (lease M2).
- The master may extend the timeout by any amount. The default extension is 12s, but an overloaded master may use higher values to reduce the number of KeepAlive calls it must process. Note the difference between the lease timeout of the client and the master (M1 vs. C1 and M2 vs. C2).
- The client initiates a new KeepAlive immediately after receiving the previous reply. Thus, the client ensures that there is almost always a KeepAlive call blocked at the master.





An ephemeral **node that will disappear when the session of its owner ends**.

Chubby Working : Sessions & Events

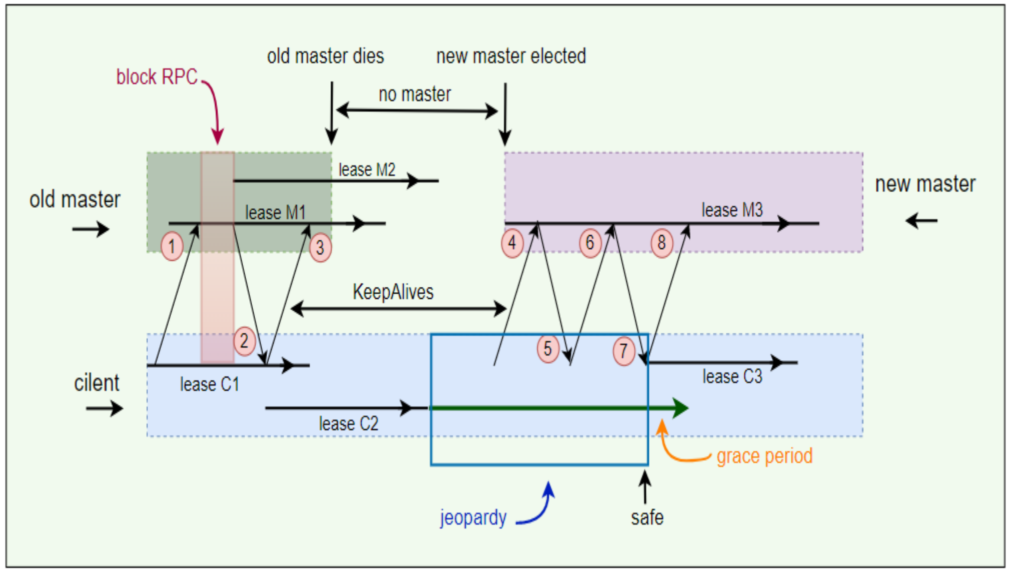
Session optimization

Piggybacking events: KeepAlive reply is used to transmit events and cache invalidations back to the client.

Local lease: The client maintains a local lease timeout that is a conservative approximation of the master's lease timeout.

Jeopardy: If a client's local lease timeout expires, it becomes unsure whether the master has terminated its session. The client empties and disables its cache, and we say that its session is in jeopardy.

Grace period: When a session is in jeopardy, the client waits for an extra time called the grace period - 45s by default. If the client and master manage to exchange a successful KeepAlive before the end of client's grace period, the client enables its cache once more. Otherwise, the client assumes that the session has expired.



Failovers

The failover scenario happens when a master fails or otherwise loses membership. Following is the summary of things that happen in case of a master failover:

The failing master discards its in-memory state about sessions, handles, and locks.
Session lease timer is stopped. This means no lease is expired during the time when the master failover is happening. This is equivalent to lease extension.
If master election occurs quickly, the clients contact and continue with the new master before the client's local lease expires.
If the election is delayed, the clients flush their caches (= jeopardy) and wait for the "grace period" (45s) while trying to find the new master.

An ephemeral **node that will disappear when the session of its owner ends**.

Chubby Working : Master Election & Chubby Event

How to initialize the newly elected Master?

- **Picks epoch number**: It first picks up a new client epoch number to differentiate itself from the previous master. Clients are required to present the epoch number on every call. The master rejects calls from clients using older epoch numbers. This ensures that the new master will not respond to a very old packet that was sent to the previous master.
- **Responds to master-location requests** but does not respond to session-related operations yet.
- **Build in-memory data structures**:
  - ✓ It builds in-memory data structures for sessions and locks that are recorded in the database.
  - ✓ Session leases are extended to the maximum that the previous master may have been using.
- **Let clients perform KeepAlives** but no other session-related operations at this point.
- **Emits a failover event to each session**: This causes clients to flush their caches (because they may have missed invalidations) and warn applications that other events may have been lost.
- **Wait**: The master waits until each session acknowledges the failover event or lets its session expire.
- **Allow all operations to proceed**.
- **Honor older handles by clients**: If a client uses a handle created prior to the failover, the master recreates the in-memory representation of the handle and honors the call.
- **Deletes ephemeral files**: After some interval (a minute), the master deletes ephemeral files that have no open file handles. Clients should refresh handles on ephemeral files during this interval after a failover.

Chubby Event?

Chubby supports simple events.

Events are delivered to the client asynchronously via callback from the chubby library .

Clients need to subscribe the **event/range of events** while creating handle

Few example for such events are mentioned below

- ✓ File contents modified
- ✓ Child node added, removed and modified
- ✓ Chubby master failed over
- ✓ When handle becomes invalid?
- ✓ Lock Acquired?
- ✓ Any Conflict of lock situation with another client

And so on…

Additionally , Chubby also sharing standard Session Events mentioned below

Expired : If the session timeout
Safe: When your session survived some communication issues
Jeopardy: When session lease time out and grace period begins

An ephemeral **node that will disappear when the session of its owner ends**.

Chubby Working : Caching

Chubby Cache

Chubby System is read biased, and so one thought will automatically come into mind of an architect that Caching must be the one of the important component in this system. So, let's learn the caching used in Chubby, next in this section.

To reduce read traffic, Chubby clients cache **file contents, node metadata, and information** on open handles in a consistent, **write-through cache** in the client's memory.

Because of this caching, Chubby must maintain consistency between a file and a cache as well as between the different replicas of the file.

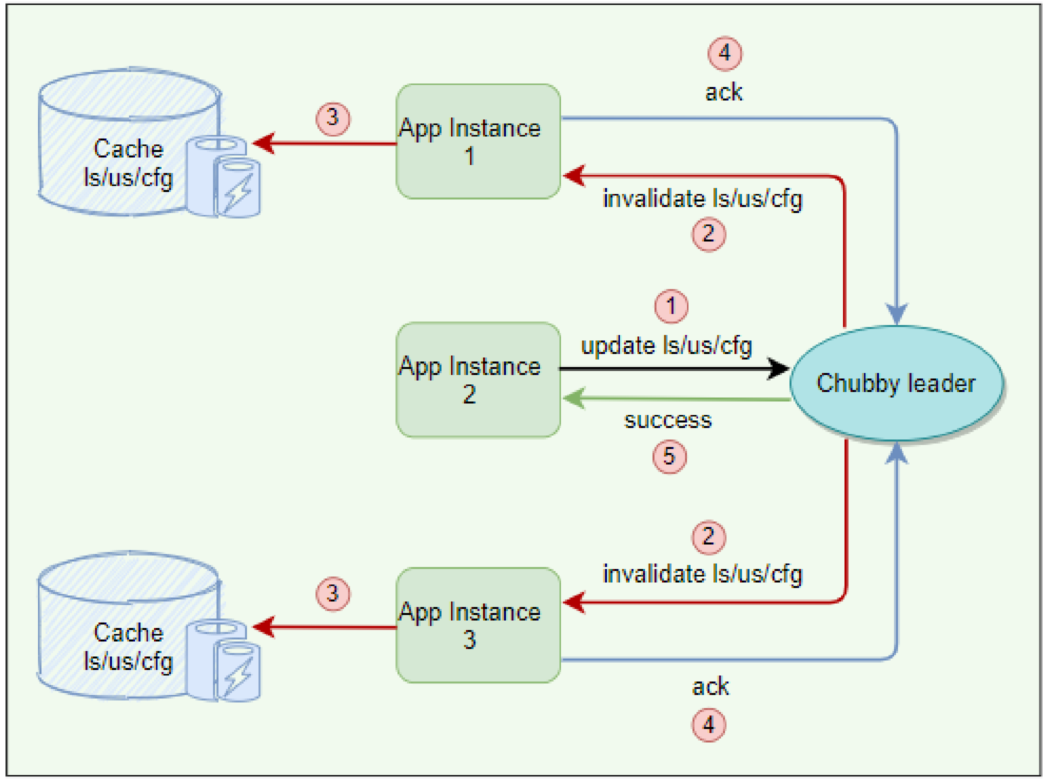
So, from above it is clear that cache eviction policy in this system must be some thing like time based.

Chubby clients maintain their cache by a lease mechanism and flush the cache when the lease expires.

Cache Invalidation

Below is the protocol for invalidating the cache when file data or metadata is changed:

- ✓ Master receives a request to change file contents or node metadata.
- ✓ Master blocks modification and sends cache invalidations to all clients who have cached it. For this, the master must maintain a list of each client's cache contents.
- ✓ For efficiency, the invalidation requests are piggybacked onto KeepAlive replies from the master.
- ✓ Clients receive the invalidation signal, flushes the cache, and sends an acknowledgment to the master with its next KeepAlive call.
- ✓ Once acknowledgments are received from each active client, the master proceeds with the modification. The master updates its local database and sends an update request to the replicas.
- ✓ After receiving acknowledgments from most replicas in the cell, the master sends an acknowledgment to the client who initiated the write.



An ephemeral **node that will disappear when the session of its owner ends**.

Chubby Working : Caching – Q&A

Chubby Cache – Q&A

Q: While the master is waiting for acknowledgments, are other clients allowed to read the file?

Ans: **Yes**, during the time the master is waiting for the acknowledgments from clients, the file is treated as 'uncachable.'

This means that the clients can read the file but will not cache it.

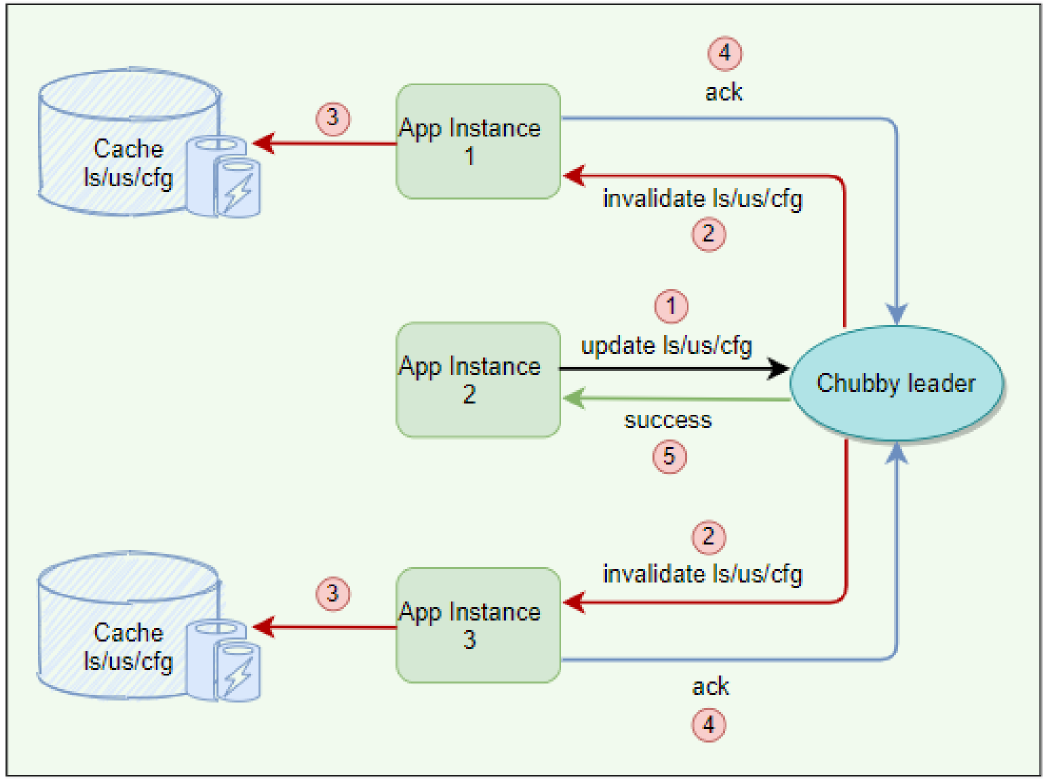
This approach ensures that reads always get processed without any delay. This is useful because reads outnumber writes.

Q: Are clients allowed to cache locks? If yes, how is it used?

Ans: **Yes**, Chubby allows its clients to cache locks, which means the client can hold locks longer than necessary, hoping that they can be used again by the same client.

Q: Are clients allowed to cache open handles?

Ans: **Yes**, Chubby allows its clients to cache open handles. This way, if a client tries to open a file it has opened previously, only the first open() call goes to the master.



Chubby Working : Database

How Chubby using database?

Chubby usage database for storage.

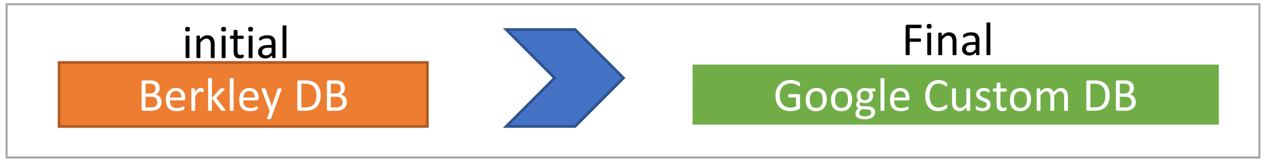
In initial days of chubby implementation, for database , inventor used Berkley DB and replicated it for better Fault tolerance.

But later they can know that Berkley db. is not enough for risk mitigation, and so they decided to build their own database.

So, they implemented their own custom database for the following characteristics

✓ Simple key-value pair database using write ahead logging & snapshotting's
✓ Atomic operation only, no need of general transactional based database
✓ Database log is distributed among replicas using Paxos

So, **One key learning here is sometime being an architect or innovator also , you are not sure about your requirement in detail. Meaning you have idea but not much in specific to requirement level, even in that situation we must proceed further and change the component if they need to change for betterment of overall your system.**



Backup

Why backup is important?
- To Support the recovery, incase of failure.

Approach of backup in Chubby?

All database transactions are stored in a transaction log (a write-ahead log).
If you keep appending the log than this log himself become unmanageable in size, so this approach won't work.

Backup contd.

So, the actual approach in chubby system is like

- Keep transaction log (a write-ahead log)
- Once snapshot is taken, delete the transaction log
- Start fresh for logging the transaction post snapshot
- at any time, the complete state of the system is determined by the last snapshot together with the set of transactions from the transaction log.
- Backup - databases are used for disaster recovery
- Backup - database initialize the database of a newly replaced

Mirroring

Mirroring is a technique that allows a system to automatically maintain multiple copies.

- Chubby allows a collection of files to be mirrored from one cell to another.
- Mirroring is commonly used to copy configuration files to various computing clusters distributed around the world.
- Mirroring is fast because the files are small.
- Event mechanism informs immediately if a file is added, deleted, or modified.
- Usually, changes are reflected in dozens of mirrors worldwide under a second.
- Unreachable mirror remains unchanged until connectivity is restored. Updated files are then identified by comparing their checksums.
- A special "global" cell subtree /ls/global/master that is mirrored to the subtree /ls/cell/replica in every other Chubby cell.
- Global cell is special because its replicas are in widely separated parts of the world. Global cell is used for:
  - ✓ Chubby's own access control lists (ACLs).
  - ✓ Various files in which Chubby cells and other systems advertise their presence to monitoring services.
  - ✓ Pointers to allow clients to locate large data sets such as Bigtable cells, and many configuration files for other systems.

Chubby Working : Scaling

How Chubby system scaling?

Chubby's clients are individual processes, so Chubby handles more clients than expected. At Google, 90,000+ clients communicating with a single Chubby server is one such example.

Next point here is how to make sure that this system scale at max, and for that what would be your engineering decision

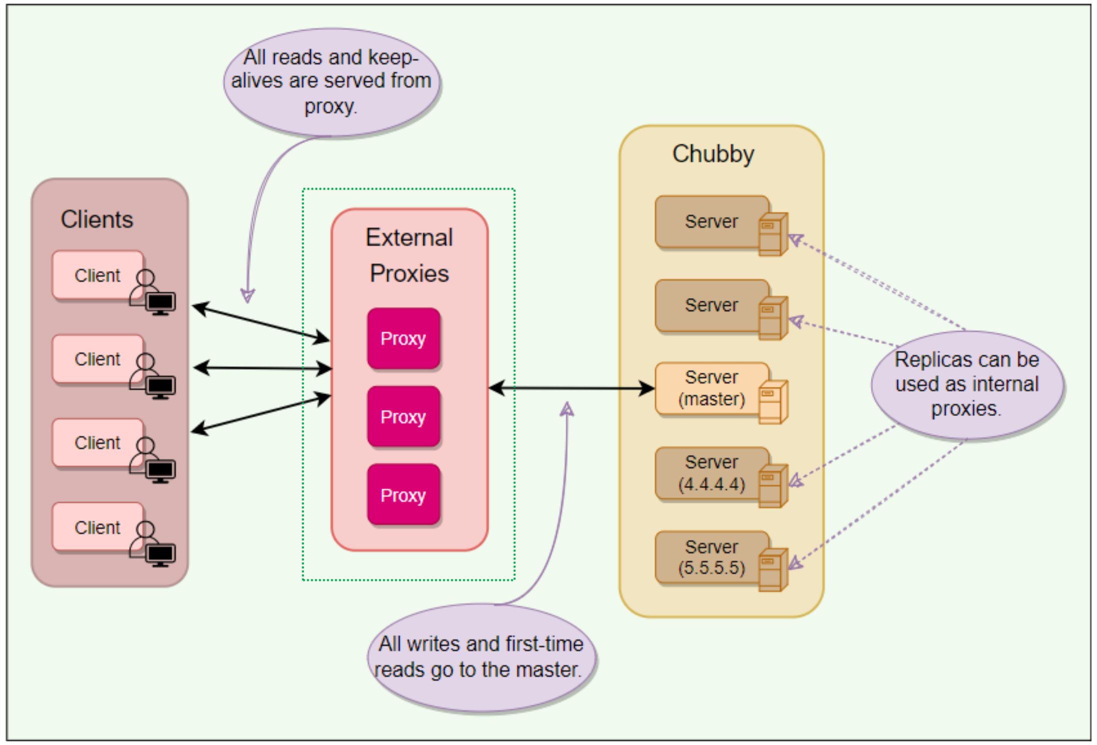
✓ **Minimize request rate** -> How you do it?
By creating more and more chubby cell, you will give the opportunity and closer chubby cell to each client. Which will eventually decrease the request rate to the chubby server.

✓ **Minimize KeepAlive load-**> How you do it?
Honestly speaking, KeepAlive load is like server is hitting request continuously and it is one area which require to improve so that server resource is not wasted for this. So, by increasing client lease time from 12s(default) to 60 second.

✓ **Caching** -> How you do it?
This is also one of the good decision took by chubby client implementation to reduce the workload of chubby server. Chubby client is usually caching – file data, metadata, and handles.

✓ **Simplified protocol conversions** -> How you do it?
As per the chubby architect, they suggested to introduce additional layer between the chubby client and server like **proxy**, which will convert complex protocol requirement to simplified protocol, and it improves the actual server computing requirement less, and so the server resource is available for other clients. In proxies also there are two different kind of proxies possible here, Internal and External proxies. Here also chubby architect has done excellent work to think like this. Therefore, this service has service at that much scale.

Here is a quick caveat : **"All writes and first-time reads pass through the cache to reach the master. This means that proxy adds an additional RPC for writes and first-time reads. This is acceptable as Chubby is a read-heavy service."**


All reads and keep-alives are served from proxy.
Clients
Client
Client
Client
Client
External Proxies
Proxy
Proxy
Proxy
Chubby
Server
Server
Server (master)
Server (4.4.4.4)
Server (5.5.5.5)
Replicas can be used as internal proxies.
All writes and first-time reads go to the master.

An ephemeral **node that will disappear when the session of its owner ends.**

Chubby Working : Learnings

How Partitioning helps in Chubby system scaling?

Partitioning

Chubby's interface (files & directories) was designed such that namespaces can easily be partitioned between multiple Chubby cells if needed. This would result in reduced read/write traffic for any partition, for example:

- ✓ ls/cell/foo and everything in it, can be served by one Chubby cell, and
- ✓ ls/cell/bar and everything in it, can be served by another Chubby cell

There are some scenarios in which partitioning does not improve:

- When a directory is deleted, a cross partition call might be required.
- Partition does not necessarily reduce the KeepAlive traffic.
- Since ACLs can be stored in one partition only, so a cross partition call might be required to check for ACLs.

Learnings

Lack of aggressive caching: Initially, clients were not caching the absence of files or open file handles. An abusive client could write loops that retry indefinitely when a file is not present or poll a file by opening it and closing it repeatedly when one might expect they would open the file just once. Chubby educated its users to make use of aggressive caching for such scenarios.

Lack of quotas: Chubby was never intended to be used as a storage system for large amounts of data, so it has no storage quotas. In hindsight, this was naive. To handle this, Chubby later introduced a limit on file size (256kBytes).

Publish/subscribe: There have been several attempts to use Chubby's event mechanism as a publish/subscribe system. Chubby is a strongly consistent system, and the way it maintains a consistent cache makes it a slow and inefficient choice for publish/subscribe. Chubby developers caught and stopped such uses early on.

Developers rarely consider availability: Developers generally fail to think about failure probabilities and wrongly assume that Chubby will always be available. Chubby educated its clients to plan for short Chubby outages so that it has little or no effect on their applications.

Chubby Working : Summary

Summary

- Chubby is a distributed lock service used inside Google systems.
- It provides coarse-grained locking (for hours or days) and is not recommended for fine-grained locking (for seconds) scenarios. Due to this nature, it is more suited for high-read and rare write scenarios.
- Chubby's primary use cases include naming service, leader election, small files storage, and distributed locks.
- A Chubby Cell basically refers to a Chubby cluster. A chubby cell has more than one server (typically 3-5 at least) known as replicas.
- Using Paxos, one server is chosen as the master at any point and handles all the requests. If the master fails, another server from replicas becomes the master.
- Each replica maintains a small database to store files/directories/locks. Master directly writes to its own local database, which gets synced asynchronously to all the replicas for fault tolerance.
- Client applications use a Chubby library to communicate with the replicas in the chubby cell using RPC.
- Like Unix, Chubby file system interface is basically a tree of files & directories (collectively called nodes), where each directory contains a list of child files and directories.
- Locks: Each node can act as an advisory reader-writer lock in one of the following two ways:
  - ✓ Exclusive: One client may hold the lock in exclusive (write) mode.
  - ✓ Shared: Any number of clients may hold the lock in shared (reader) mode.
- Ephemeral nodes are used as temporary files, and act as an indicator to others that a client is alive. Ephemeral nodes are also deleted if no client has them open. Ephemeral directories are also deleted if they are empty.
- Metadata: Metadata for each node includes Access Control Lists (ACLs), monotonically increasing 64-bit numbers, and checksum.
- Events: Chubby supports a simple event mechanism to let its clients subscribe for a variety of events for files such as a lock being acquired, or a file being edited.
- Caching: To reduce read traffic, Chubby clients cache file contents, node metadata, and information on open handles in a consistent, write-through cache in the client's memory.
- Sessions: Clients maintain sessions by sending KeepAlive RPCs to Chubby. This constitutes about 93% of the example Chubby cluster's requests.
- Backup: Every few hours, the master of each Chubby cell writes a snapshot of its database to a GFS file server in a different building.
- Mirroring: Chubby allows a collection of files to be mirrored from one cell to another. Mirroring is used most commonly to copy configuration files to various computing clusters distributed around the world.

An ephemeral **node that will disappear when the session of its owner ends**.

Chubby Working : Summary

System design patterns

Here is a summary of system design patterns used in Chubby.

1. Write-Ahead Log: For fault tolerance and to handle a master crash, all database transactions are stored in a transaction log.

2. Quorum: To ensure strong consistency, Chubby master sends all write requests to the replicas. After receiving acknowledgments from most replicas in the cell, the master sends an acknowledgment to the client who initiated the write.

3. Generation clock: To disregard requests from the previous master, every newly-elected master in Chubby uses 'Epoch number', which is simply a monotonically increasing number to indicate a server's generation. This means if the old master had an epoch number of '1', the new one would have '2'. This ensures that the new master will not respond to any old request which was sent to the previous master.

4. Lease: Chubby clients maintain a time-bound session lease with the master. During this time interval, the master guarantees to not terminate the session unilaterally.

References and further reading

Chubby paper
Chubby architecture video
Chubby vs. ZooKeeper
Hierarchical Chubby
Bigtable
Google File System

An ephemeral **node that will disappear when the session of its owner ends**.

Distributed Locking Service : Chubby

If this Editorial can answer the following discussion points , then it served my purpose.

How does Chubby ensure fault tolerance and reliability of data?

How does Chubby ensure high availability in case of master failure?

How does Chubby ensure file data integrity?

What happens when a Chubby replica fails permanently?

How do clients find the master using Chubby?

During cache invalidation, while the master is waiting for acknowledgments from clients, can other clients still read the file?

Chubby stores its database snapshot in a GFS cluster. How does Chubby avoid cyclic dependency if that GFS cluster depends upon Chubby for electing its master?