Design Rationale

Let's explore the rationale behind Chubby's architecture.

We'll cover the following

- Why was Chubby built as a service?
- Why coarse-grained locks?
- Why advisory locks?
- Why Chubby needs storage?
- Why does Chubby export a Unix-like file system interface?
- Chubby in terms of CAP theorem

Before we jump into further details and working of Chubby, it is important to know the logic behind certain design decisions. These learnings can be applied to other problems of similar nature.

Why was Chubby built as a service?#

Let’s first understand the reason behind building a service instead of having a client library that only provides Paxos distributed consensus. A lock service has some clear advantages over a client library:

- **Development becomes easy:** Sometimes high availability is not planned in the early phases of development. Systems start as a prototype with little load and lose availability guarantees. As a

service matures and gains more clients, availability becomes important; replication and primary election are then added to design. While this could be done with a library that provides distributed consensus, a lock server makes it easier to maintain the existing program structure and communication patterns. For example, electing a leader requires adding just a few lines of code. This technique is easier than making existing servers participate in a consensus protocol, especially if compatibility must be maintained during a transition period.

- **Lock-based interface is developer-friendly:** Programmers are generally familiar with locks. It is much easier to simply use a lock service in a distributed system than getting involved in managing Paxos protocol state locally, e.g., `Acquire()`, `TryAcquire()`, `Release()`.

- **Provide quorum & replica management:** Distributed consensus algorithms need a quorum to make a decision, so several replicas are used for high availability. One can view the lock service as a way of providing a generic electorate that allows a client application to make decisions correctly when less than a majority of its own members are up. Without such support from a service, each application needs to have and manage its own quorum of servers.

- **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.

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.

Why coarse-grained locks?#

Chubby locks usage is not expected to be fine-grained in which they might be held for only a short period (i.e., seconds or less). For example, electing a leader is not a frequent event. Following are some main reasons why Chubby decided to only support coarse-grained locks:

- **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. With fine-grained locks, even a brief unavailability of a lock server would cause many clients to stall.
- **Fewer lock servers are needed:** Coarse-grained locks allow many clients to be adequately served by a modest number of lock servers with somewhat lower availability.

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. Holding a specific lock is neither necessary to access the file, nor prevents others from doing so.

Other types of locks are mandatory locks, which make objects inaccessible to clients not holding the lock. 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.

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.

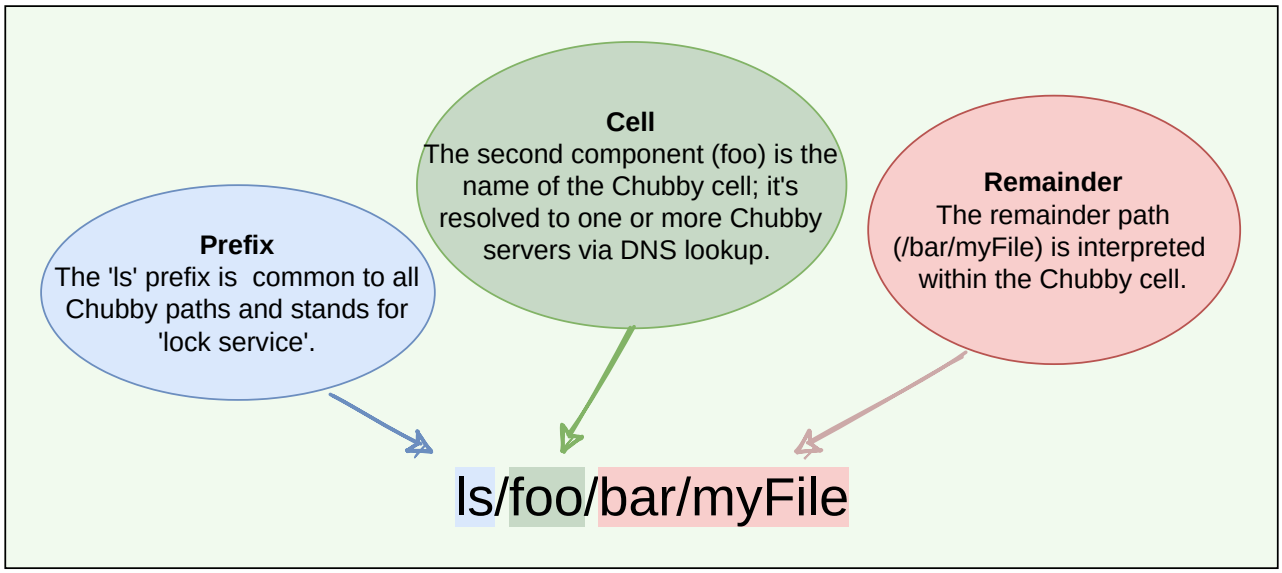
Not having a separate service for sharing the results reduces the number of servers that clients depend on. Chubby's storage requirements are really simple. i.e., store a small amount of data (KBs) and limited operation support (i.e., create/delete).

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

Recall that Chubby exports a file system interface similar to 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`

The main reason why Chubby's naming structure resembles a file system to make it available to applications both with its own specialized API, and via interfaces used by our other file systems, such as the Google File System. This significantly reduced the effort needed to write basic browsing and namespace manipulation tools, and reduced the need to educate casual Chubby users. However, only a very limited number of operations can be performed on these files, e.g., Create, Delete, etc.



Breaking down Chubby paths

Chubby in terms of CAP theorem#

As proved by CAP theorem (https://www.educative.io/collection/page/5668639101419520/5559029852536832/5998984290631680), no application can be highly available, strongly consistent, and partition tolerant at the same time. Since Chubby

guarantees strong consistency by ensuring that all read and write requests go through the master, technically, we can categories Chubby as a **CP** system.

Chubby's architecture is based on a master-backup design which ensures strong consistency. If the master dies, Chubyy chooses a new master from the backup servers. Chubby continues to operate as long as no more than half of the servers fail, and it is guaranteed to make progress whenever the network is reliable. If the Chubby servers experience network partitioning, the service will become unavailable.

On the other hand, Chubby is optimized for the case where there is a stable master server, and there are no partitions. In this case, it delivers a very high degree of availability. So, overall, Chubby provides guaranteed consistency and a very high level of availability in the common case.

← Back

High-level Architecture

Next →

How Chubby Works

Mark as Completed

Report an Issue