Detailed Design of Chubby: Part I

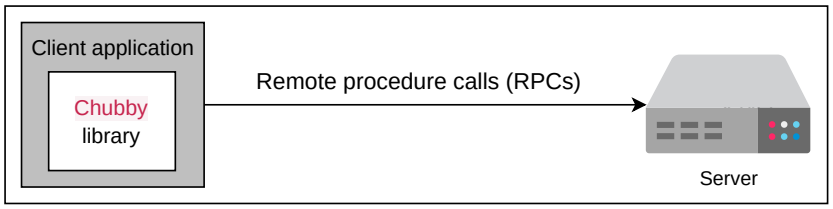
Learn about the components, different types of requests, and the namespace of the server in Chubby.

We'll cover the following

Components of the Chubby system

Communication between clients and servers happens through the client library linked with the client application.

> **Note:** Chubby also has another component called a proxy server (this is an optional component that we discuss later).

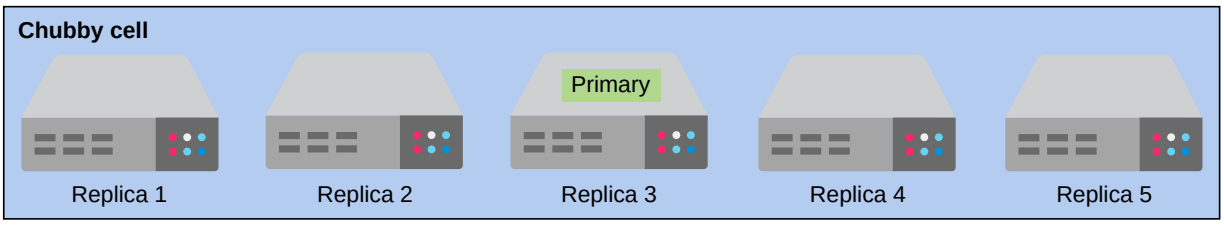


Client application using Chubby service via Chubby's client library

If we look at the illustration above, there is a single server providing the Chubby service. If that server fails, the service will be down. To handle this, Chubby deploys a set of servers called a **Chubby cell**. A Chubby cell usually consists of five servers placed in different racks to reduce correlated failures. The servers placed in the same rack may share the electric power. For example, in case of a short circuit, the whole rack will be unavailable, and if we keep all the servers of a Chubby cell in the same rack, all of them will be unavailable. So, the servers are placed in different racks to reduce the system's unavailability. We may distribute replicas across data centers to guard against the full data center failure. However, doing so makes data consistency trickier and adds latency.

Servers

All the servers in the Chubby cell are replicas of each other and maintain copies of Chubby's database. Out of all the replica servers in the Chubby cell, one is elected as the **primary** using a consensus protocol. The rest of the servers in the Chubby cell are called **secondary replicas** or simply **replicas**.



A Chubby cell

Primary replica: The primary replica has to get the votes of a majority of replicas and a guarantee from those replicas that they won't choose a different primary replica for a small period (a few seconds). This guarantee is known as the **primary replica lease**. Under the condition that the primary replica continues to receive a majority of the vote, the replicas periodically renew the primary replica lease.

Replicas must have a leader who can initiate read and write operations, called a primary replica.

- The primary replica itself entertains all the read operations. This way, the client knows that there is no other primary replica, and it can continue to communicate with it.
- The primary replica also performs the write operations, but these are also propagated to other replica servers to perform as well.

It also keeps a cell's database containing a list of machines (IP addresses) on the cell.

Replica servers: The primary replica has all the data a relevant client would want, and it can satisfy read and write operations from the client. However, if it fails, there must be some replica servers that maintain a copy of the database in the primary replica. The replicas copy the database using consensus protocol (making sure each write request is not acknowledged until it is accepted by the majority of the replicas). They can also elect a new primary replica or renew an older lease of a primary replica by re-electing it with majority votes using the consensus protocol.

Since the primary replica and the other replica servers contain copies of the same database, the difference between them regarding their duties/responsibilities/roles is listed below:

- Replica servers are responsible for electing a primary replica and providing backup to the overall data, for in case the primary fails, they elect a new primary replica.
- The primary replica is responsible for entertaining all the read requests and write requests. It is also responsible for propagating write requests to the replica servers.

Cell database: All the replica servers keep the data, metadata, and a list of all the servers operating in the cell in this database.

Point to ponder

Question

Why do we use a primary replica to reply to all write and read operations?

Hide Answer

All read requests are satisfied only by the primary replica, which is updated all the time due to the write operations also happening on it. Hence, the process above is secure, provided the primary replica lease has not ended. This process will ensure that read processes are not occurring while a write is being processed, which might update the data that will be read.

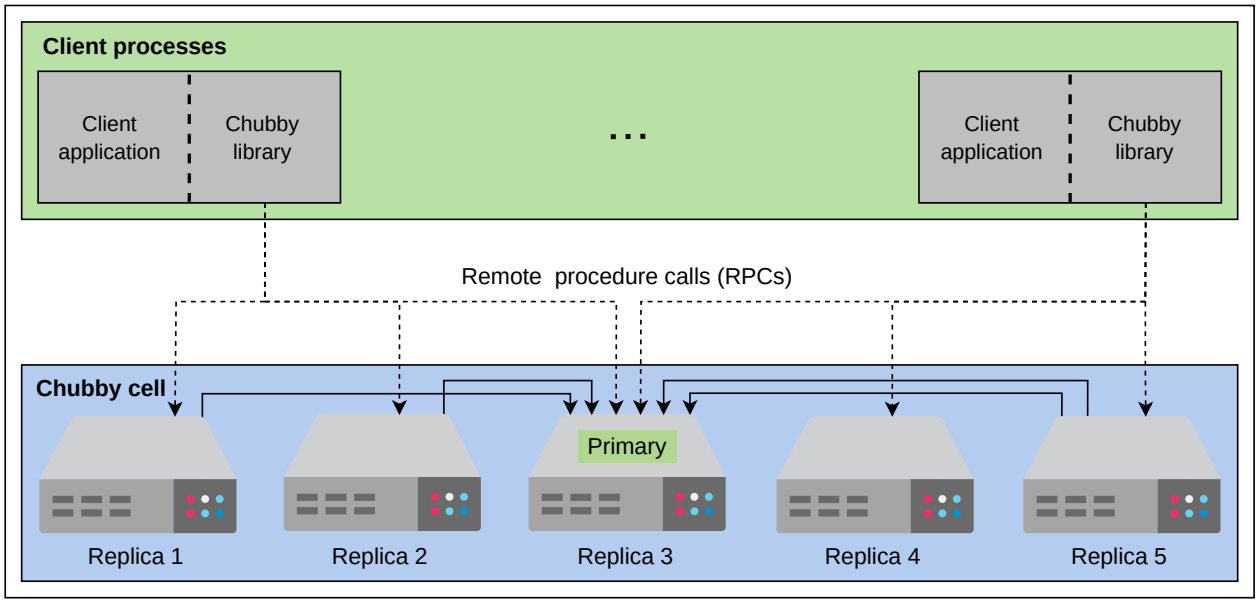
Conducting reads and writes at the same node (the primary replica) helps our system ensure strong data consistency. However, the disadvantage of doing so is that there is a limit on read and write throughput that we can extract from a single node.

Let’s view the overall design of Chubby.

Communication between components

Chubby’s design is shown in the following illustration. There are client processes that want to use the Chubby service. Each client process interacts with the Chubby server through the `Chubby` library. To perform any operation on Chubby, the client has to find the primary replica server because that is responsible for the read or write operations on the database.

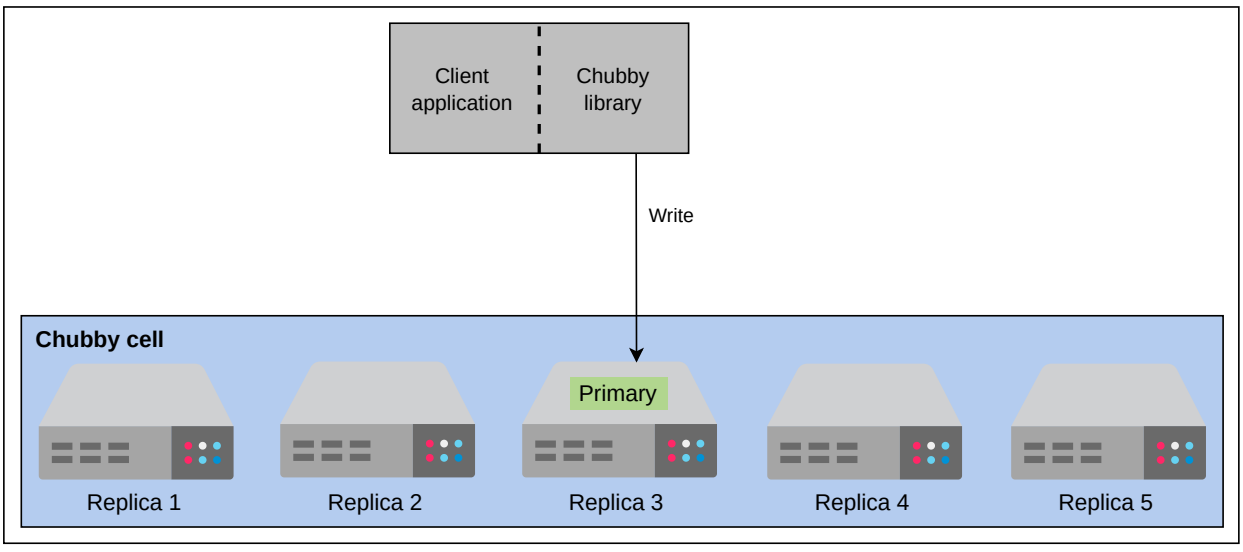
The list of servers in the Chubby cell is placed on the Chubby DNS server. Clients get this list from the DNS and send the primary replica location request to all servers in that list. All the non-primary replica servers return the location of the primary replica (probably the IP address) in response to such requests because they know the primary replica. After finding the primary replica’s location, the client then directs all its requests to the primary replica. It continues to do so until the primary replica stops responding or it indicates that it is no longer a primary replica.



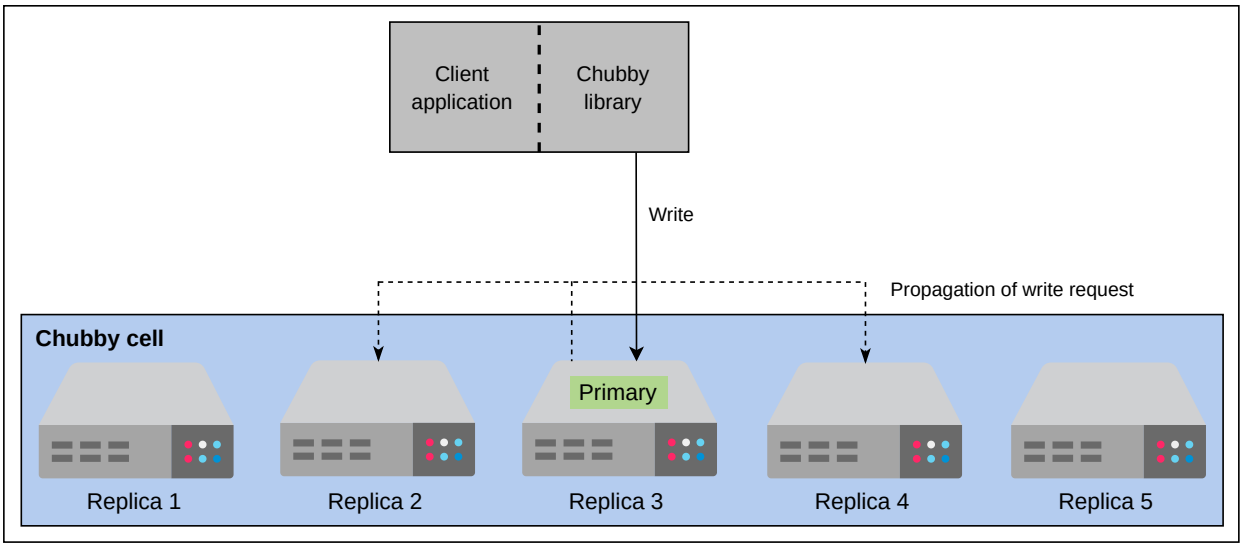
The clients finding the primary replica

There are two types of requests–write and read.

- When a primary replica receives a write request, it propagates the request via the consensus protocol to all replica servers. Write requests are asynchronous and are only acknowledged until they are propagated to the majority of replica servers. Making sure that the write request has propagated to the majority of replica servers signifies that most replica servers are updated and can be elected as a new primary replica in case of failure. Chubby does not wait for the propagation of requests to all the replica servers because these can fail and not respond for a few hours, which can stall a write request for too long.

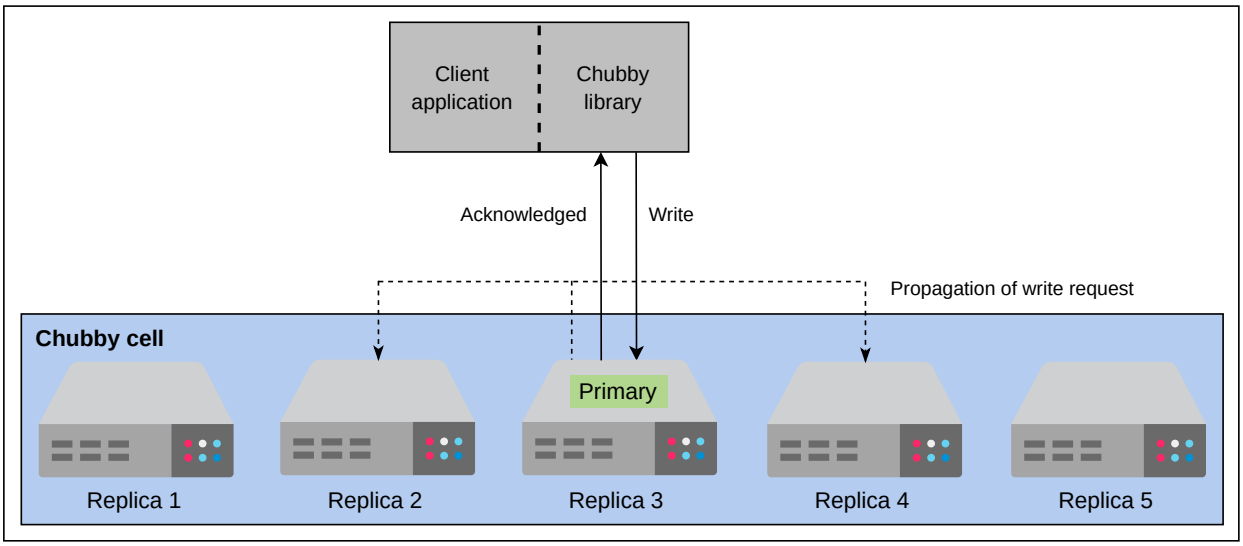


The client sends a write request to the primary replica



The write request is propagated to the majority of replica servers

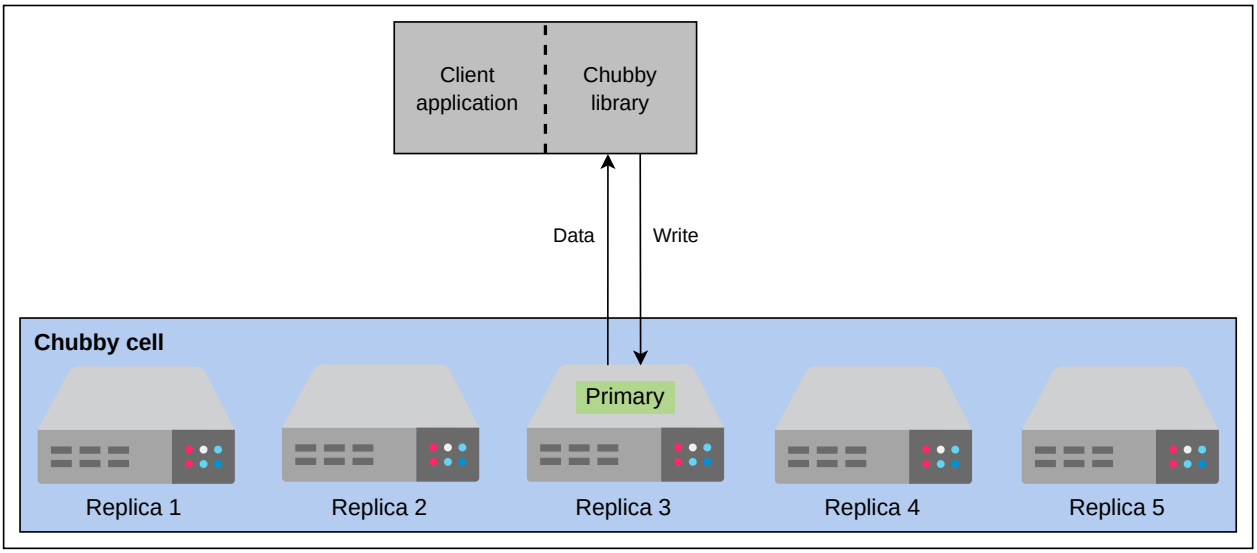
2 of 3



The write request is acknowledged

3 of 3

- The read requests are satisfied by the primary replica.



Read request

Inside a server

Let's go through the design details of the server.

Files and directories

Chubby's file system is similar to UNIX. It is composed of a tree of files and directories (containing either the data or metadata of applications), which are separated by a slash (`/`). A usual name looks like the following:

```
ls/foo/wombat/pouch
```

In the example above, `foo` is the name of a particular Chubby cell and `wobat/pouch` is a node inside the cell.

Question

Why is Chubby's naming structure so similar to a file system?

Hide Answer

It is made similar to a file system to make it readily accessible to applications with their own API and with interfaces used by file systems like GFS. This is done without spending any effort on writing basic namespace manipulation and growing tools.

The programmers are well-versed with the file system API, and such familiarity helps reduce the learning curve of Chubby.

The naming design does differ from UNIX in some aspects, and those differences are there to ease distribution. These differences include:

- Operations that can move files between directories are not exposed
- Directory-modified times are not maintained
- Path-dependant permission semantics are not used–access to a file is controlled by permissions on the file instead of the directory's permissions
- Last access times are also not exposed

Question

Why are these differences introduced in the naming structure?

Hide Answer

Not exposing directory-changing operations for files, not maintaining directory modified times, and not having path-dependant permission semantics allows different primary replicas of Chubby (in case a primary replica fails and another starts) to serve files in different directories without having to keep track of the directory of a file by copying unneeded metadata to all replicas.

Chubby caches metadata and data on clients. It updates it whenever it changes on the primary replica, so if we start keeping track of the last access time of a file, it would get updated even on a read operation. This will cause Chubby to cache it again, which is not necessary because read operations do not cause any change on data itself. So, not exposing the last access times is there to make caching the file metadata easier.

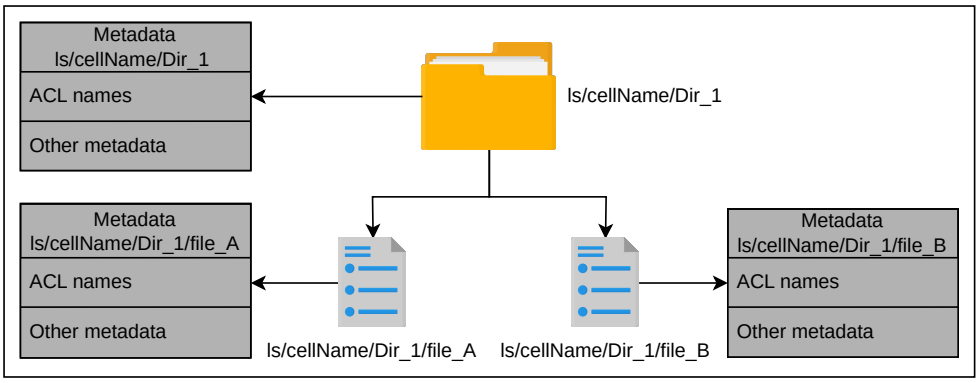
Node

The namespace contains files and directories. Each file or directory is known as a **node**. Each node has a unique name (the name includes the whole path) within a cell, and it has no hard link or sym-link. Nodes can be categorized into two types based on what they represent:

- Ephemeral
- Permanent

Ephemeral nodes are there to show that a client is alive to other clients and permanent files are there to advertise data/metadata of a service. Both of these nodes can be specifically deleted. However,

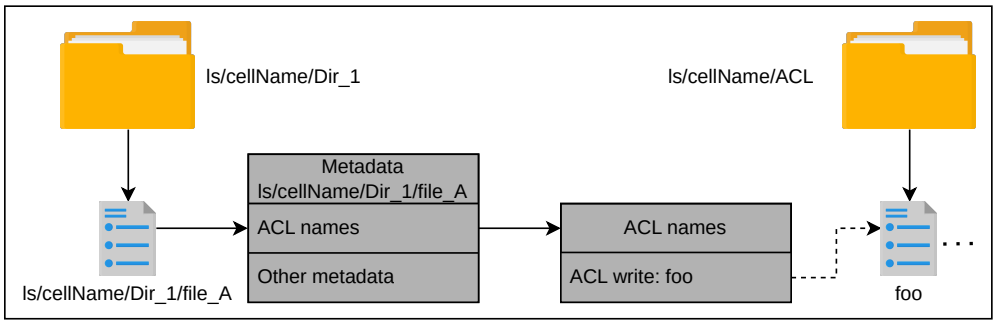
ephemeral nodes can also get deleted when no client has them open.



Nodes and their metadata

Access control lists (ACLs) are files in the ACL directory of the namespace. These files contain a simple list of authorized clients' names. They are readily available for services that want to use similar authorizing mechanisms. Therefore, the users are authorized by mechanisms built into the RPC system.

Each node has metadata, including ACL file names. There are three ACL names because we need to keep track of the read ACL names, write ACL names, and changed ACL names. A node uses them to keep a check on writing, reading, and modifying ACL names. When a new node is created, it inherits the ACLs of its parent directory unless they are overridden. For example, if a file A's write ACL name is `foo`, there is a file named `foo` in the ACL directory, and it contains an entry `user_1`, then `user_1` has permission to write A.



Write the ACL of file_A

Other metadata: Each node also has four monotonically increasing numbers of 64 bits that help clients detect changes in files and directories. Whenever a node is created, all these numbers start from 0. These numbers include:

1. **The instance number:** If a new node is created with a name similar to a previously created node, it will have an instance number greater than the previous one. Let's say we create a node `ls/cellName/Dir1/file_A` that is already present in the cell with instance number 0. It will be created with instance number 1.
2. **The content generation number (CGN):** It increases when a file's content is generated. The CGN of `ls/cellName/Dir1/file_A` after writing content in it becomes 1 and will increase every time

content is generated in it.

3. **The lock generation number:** It is incremented when a node's lock is held. If a client holds a lock at `ls/cellName/Dir1/file_A` its lock generation number will become 1.
4. **The ACL generation number:** It is incremented when the ACL name of a node is written. When ACL names of `ls/cellName/Dir1/file_A` are written, it's ACL generation number will be incremented to 1.

All the replicas, along with data, also copy this metadata into their replicated databases. For clients to be able to identify if the files differ, Chubby also has a 64-bit file-content checksum.

Handle

- Clients open a node and get a handle (non-negative integers that act as unique identifiers to open nodes) to it (similar to UNIX file descriptors). These handles include:
    - **Check digits:** These are used to prevent clients from creating and guessing handles.
    - **Sequence number:** It informs a primary replica if a handle is created by itself or a previous primary replica.
    - **Mode information:** It is given to a primary replica for state recreation when a restarted primary replica is presented with an old handle.