Java-based Implementation of Google's Chubby Lock Service using Raft Consensus Algorithm

Marco Fontana

March 28, 2024

This project explores the design and implementation of "Chubby", a distributed file system, designed by Google, that offers a lock service for loosely-coupled distributed systems. Chubby prioritizes consistency over availability and provides coarse-grained locking alongside reliable, low-volume storage. The platform enables users to perform various operations such as creating and deleting files, reading and writing text data, managing permissions, and receiving notifications about file system changes. The system was developed using Java and the Raft consensus algorithm.[1][2]

Contents

List of Figures

1 Goals/Requirements

The goals to be achieved by this project were:

- reading and understanding Google's Chubby distributed lock service paper
- implementing it using Java and Raft consensus algorithm
- testing it's robustness against crashes

1.1 Scenarios

Clients connect to a 'Chubby Cell' that is a 5-node network that represents a distributed file system that provides a set of services that each client may use through a command line interface that enables them to make common file system operations such as to create and delete files and directories, reading and writing file's data, managing permissions through access control lists (ACL) and to also subscribe to specific event types in order to receive notifications about changes in the environment, such as file content changes, directories added and removed and other clients trying to access an exclusively locked resource.

In order to connect to a Chubby Cell, clients need to authenticate, providing username, password and the cell name they want to connect to. Clients cannot perform any operation to the cell until they authenticate. (See figure 1).

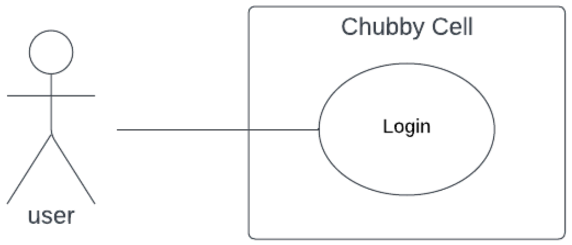


Figure 1: Use case diagram, user login.
Users may not perform any operation on the system until they authenticate

Upon authentication clients, as showed in figure 2, may open a new handle on a file or directory based on the handle type it can either be an exclusive lock on the resource, where the client may make write operations, or a shared lock, where the client may read the content of the resource. An initial handle of type read is given to the client to perform starting operations in the system.

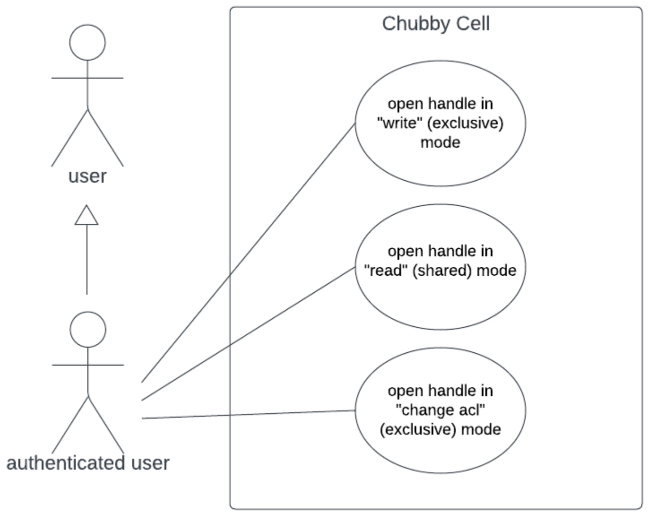

Chubby Cell
user
authenticated user
open handle in "write" (exclusive) mode
open handle in "read" (shared) mode
open handle in "change acl" (exclusive) mode

Figure 2: Use case diagram, operations that an authenticated user can do
Upon authenticating, clients may open handles on nodes to execute specific operations afterwards

Each handle gives the user the option to perform specific operations on the resource that it's currently holding.

As shown in figure 3, those operations are:

- `read`, with this handle type, clients may read both the file content and the ACL (Access Control List) of the resource.
- `write`, clients with this handle type may execute the same operations of the `read` handle, plus overwrite the file content and remove the resource from the system.
- `change_acl`, this handle type permits clients to manage permissions (read, write and change acl) to the resources.

Both write and change_acl handles permits exclusive access to the resource, meaning that other clients may not access the resource with another handle of the same type while another client has it open. On the other hand, read handle types, permit shared access to the resource, meaning that other clients may open handle of both types on it.

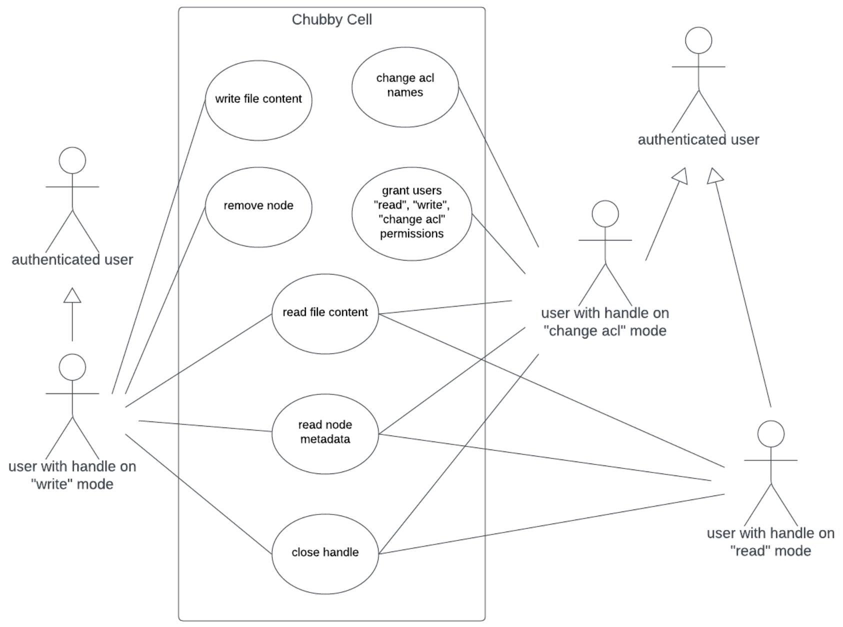

Chubby Cell
write file content
change acl names
remove node
grant users "read", "write", "change acl" permissions
read file content
read node metadata
close handle
authenticated user
user with handle on "write" mode
authenticated user
user with handle on "change acl" mode
user with handle on "read" mode

Figure 3: Use case diagram, commands that can be done based on handle type Read handles permit clients to read the node's data. Write and change_acl handles enable writing operations such as overwriting file content and changing the acl permissions

1.2 Self-assessment policy

The quality of the produced software may be assessed by the numerous tests that were made to make sure it works correctly in each scenario.

2 Background

Many aspects of distributed systems were taken into account in order to produce the software for the project.

First of all, since the CAP theorem asserts that it's not possible to guarantee availability, consistency and fault tolerance at any given time, chubby chooses to prioritize consistency and fault tolerance, meaning that in the event of a network partition, it will ensure data consistency at the expense of availability.

A Chubby cell is a collection of 5 servers, also known as replicas, that work together to provide a distributed lock service. To minimize the risk of simultaneous failures, the replicas are generally placed in different racks. In this system, one of the replicas is elected as the master. The master is responsible for handling all client requests and replicating its state to the other replicas. This replication ensures that all replicas maintain a consistent state, which is crucial for the overall reliability of the system. The mechanisms for state replication and master election are handled by the Raft consensus algorithm. This algorithm allows potentially unreliable machines to reach agreement about relevant features of the system. Each machine in the distributed system has a consensus module that is used to perform these actions. In the event of a master failure, the Raft algorithm automatically elects a new master from the remaining replicas. However, it's worth noting that maintaining such a system can be costly due to the need for multiple servers.

3 Requirement Analysis

- **Functional requirements**
  - authentication
  - create and delete files and directories
  - read and write file content and metadata
  - edit files and directories permissions
  - activate subscriptions to receive events about changes
- **Non functional requirements**
  - it has to guarantee consistency
  - it has be to available and reachable at any time
  - scalability can be easy achieved by adding new chubby cells to the system. Also, the most requested operations, 'keep alive' and 'read' file content, are easily scaled: the first one can be adapted by reducing lease time through increased time period between keep alive requests if the number of clients increase; and about the second one, 'read' operations on files content do not need to extract the data from the server, since it is cached, meaning that the node does not need to be retrieved from the database to provide the data. Since both those operations are the most requested ones, this drastically reduces server load when the number of clients increase.

4 Design

4.1 System structure

The structure of the system consists of various Chubby Cells, each one with its own name space isolated from the others. Clients may choose a chubby cell to connect to and perform operations on, through RPCs, as shown in figure 4.

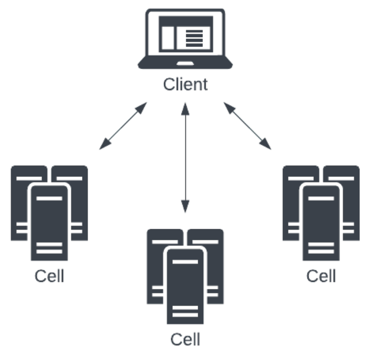


Figure 4: high level system structure
clients may connect to any cell, each one is separated from the others and has its own name space clients may work on

Each cell consists of 5 servers that guarantee consistency through replication of data. One of them is elected master and initiates read and write operations on the database, while the others, copy updates from the master; if a master fails, the other replicas run a new election and a new master is elected. (See figure 5).

4.2 Behaviour

The system's behaviour depends on the type of operation that is requested by the client.

The file system interface is similar to the one used in Unix, it consists of a hierarchy of files and directories separated by slashes, where a single slash character represents the root directory.

The system utilizes default nodes, to perform specific server-side operations, that cannot be accessed nor deleted. Those are nodes contained in the default path '/ls/cellName/acl/' and its child nodes.

The path '/ls/cellName/' consists of the root node '/', its child node 'ls' stands for 'lock service' and it's a node shared between all chubby cells, and 'cellName' is the name of cell. Clients may only create nodes in this cell's name space that have this specific starting path. The default path '/ls/cellName/acl/' contains a set of child nodes that keep records of clients that are permitted access to each node of the name space contained within the cell, and are identified by the ACL node's name: i.e. if client 'C' creates a node 'N.txt' and assigns it's write permission under the name 'N-write' a node identified by the absolute path '/ls/cellName/acl/N-write.txt' will be created and will contain an entry 'C' that assures that client 'C' has write permission on node 'N'. Other clients

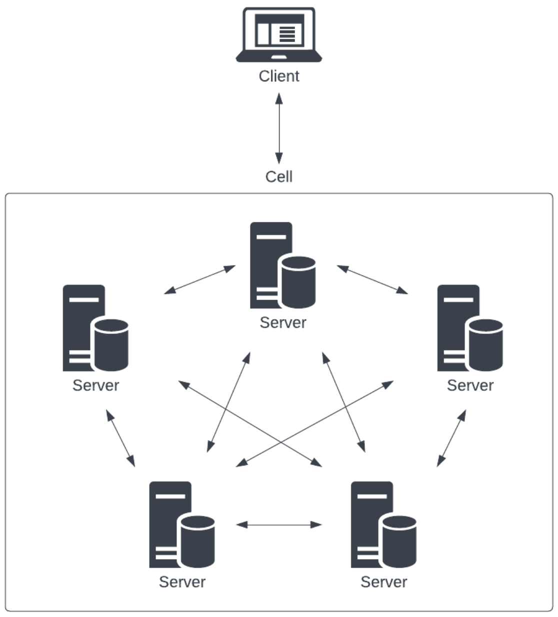


Figure 5: system structure, communication between servers inside a chubby cell Data sent from clients is stored and replicated among all servers to guarantee its consistency

may be added to the list afterwards. If not specified, each client has permission to write, read and change acl freely.

In order to synchronize clients' activities and avoid situations where clients may try to apply changes to resources simultaneously, exclusive locks are required to perform write actions on the nodes of the system. Non-critical operations, such as read operations, require a 'shared lock'. This shared lock guarantees the holder the right to read data from the node, while not preventing other clients from obtaining either exclusive or shared locks on the resource.

Each client once connected to the platform is granted a handle with shared lock on root node, this is the starting point from which a client may open other handles on resources stored into the system.

In order to open an handle, the command `open` is required. The user is expected to provide the absolute path of the node to be opened and which type of handle it will use. If the node is not already present into the system, it will be created along with its parent nodes. This is the only command that expects a path as argument and can only be executed from root node, once the handle is open, clients may operate on the node. The syntax to open a node with default parameters is the following:

```
open *absolute-path* *handle-type*
```

Optional arguments may be provided to set other parameters:

- `[node attribute]` indicates if the node that needs to be created has to be permanent or temporary. If not specified, a default value `permanent` is automatically set, otherwise user may set its type to `ephemeral` to make the node temporary. Temporary nodes are automatically deleted if no client has an open handle on it and it has no child nodes.

- `[lock delay]` indicates a time (in seconds) that will be used in case the holder has failed or becomes inaccessible, the server will prevent other clients from claiming the lock for the specified period of time. Once set, a keep-alive signal between client and server is periodically sent, to check for possible crashes. Clients may specify any lock-delay up to 60 seconds; this limit prevents other clients from making exclusive locks and making the resource unavailable for an arbitrarily long time.

- `[event subscriptions]` indicates a list of events to which clients may subscribe that are delivered asynchronously. Clients may set one or more of those subscriptions:

  - `file_contents_modified` a notification is sent if another client has modified the file content of the node currently hold by this client.
  - `child_node_added` a notification is sent if a direct child node of currently hold node is added.
  - `child_node_removed` a notification is sent if a direct child node of currently hold node is removed
  - `child_node_modified` a notification is sent if a direct child node of currently hold node is either added or removed. Note: other subscriptions that check if nodes are added or removed, will be replaced by this one, so that multiple notifications about the same events won't be notified.
  - `handle_invalid` a notification is sent if current handle has become invalid
  - `conflicting_lock` a notification is sent if another client is trying to access the node currently hold by this client with an exclusive lock (write or change acl)

So the full syntax of the `open` command, considering also optional arguments, is:

```
open *absolute-path* *handle-type* [node_attribute] [lock_delay] [event_subs]
```

Where arguments between '[' and ']' are optional.

Each time a client activates a subscription, an event watcher on the node's value is created and when the value changes and is put back into the kv store, a new event is detected and data stored at the moment of the subscription is checked, if it has changed, an asynchronous notification is sent to the client (see figure 7). For example to check if a file's content has changed, the old checksum is evaluated against the new one and if differences are detected the notification is sent. The same occurs if a client tries to access an exclusively locked resource with an exclusive lock, in that case the updated lock generation number is checked against its older value and if case of differences a notification is sent.

Each notification is sent only the first time an event is detected, so multiple changes won't get notified (except for `conflicting_lock` subscription, so that if multiple clients try to open an handle with exclusive lock on the resource, the lock holder will be notified by each of them, even if the same client tries to access it multiple times).

Once the handle is opened, user may execute various operations based on the type of handle that it's currently holding. Please refer to figure 6 to get an overview of how requests described here are processed.

The specific syntax is based on the type of handle and operation that has to be performed:

- `read` if the client has a handle of this type, it can perform those operations:
  - `read filecontent` reads the file content of the node. This data is cache-able, meaning that if a client has an open handle, multiple read requests on this data will reuse the response that was given to the client instead of retrieving the node's data from the database. If another client writes on this node, the file content that is going to be read by this client will always be the one that was present at the moment the handle was opened, ignoring possible file content updates made by other clients. In order to read updates a new handle has to be opened on the same node. Clients may activate the subscription `file_contents_modified` to get a notification when this happens.
  - `read acl` reads the acl names of this node. Default values are 'read', 'write' and 'change_acl' and are mirrored in the default path '/ls/cell-Name/acl/...', meaning that it contains three child nodes 'read.txt', 'write.txt' and 'change_acl.txt' that guarantee that each client may access the node with any permission (unless modified afterwards).
- `write` if the client has an handle of this type, it has an exclusive lock on the node, meaning that it will be the only one able to modify its content while in possession of the lock. All `read` operations mentioned above can still be executed, plus the following ones:

  - `write filecontent [content to be written]` overwrites the file content of the node with the one specified as argument. If a `read filecontent` operation is detected, the most updated version of the file content will be shown.
  - `remove` removes a node from the system, unlocking the node and opening a new handle on root node. A node can be removed only if no client has an open handle on it, and it has no child nodes.

- `change_acl` clients with this handle have exclusive access to this node's resources, since it also performs write operations on its content. All `read` operations mentioned earlier are still valid and can also be performed with this handle.

  - `write acl *ACLType* *newACLName*` replaces the 'read', 'write' or 'change_acl' permission name of this node with the specified one, removing the previous name from '/ls/cellName/acl/...' path and adding the new one instead. Clients without entries in the respective ACL will no longer be able to open handles with specified permission on the node, unless a client with 'change_acl' permissions adds them explicitly. Clients that have an open handle on the node can still perform operations until they close it.
  - `write addClient *ACLType* *client names*` adds an entry for each specified client into the ACL file contained in the default path '/ls/-cellName/acl/...', granting them the specified permission.

- clients with any handle type may also perform other utility operations:

  - `close`, closes the open handle and unlocks the node (also, if ephemeral and all the conditions mentioned above are met, removes it from the system). After those operations, a read handle on root node is opened. Put it simply, clients repeatedly traverse to and from root node each time a new handle is closed and created.
  - `list cmd`, responds with a list of all possible commands, and their syntax, that can be executed with the handle currently hold.
  - `list event`, responds with a list of all possible subscriptions that a client may request.
  - `list defnode`, responds with a list of all the default nodes of the name space.
  - `node metadata`, returns the metadata of the node currently hold
  - `node acl`, returns the ACL of the node currently hold
  - `ls [depth]`, returns the child nodes of the node currently hold by this client, an optional 'depth' parameter can be set to return child nodes based on their depth relative to current node, otherwise a default depth '1' is set, meaning that only direct child nodes are considered.

    – `exit`, exits the application gracefully.

In order to open another handle, the old one needs to be closed via `close` command, that closes the handle currently opened and opens a handle of type read (that is always guaranteed) on root node, going back to the initial state mentioned above.

4.3 Interaction

Upon receiving a request, the chubby cell offloads its parsing to a processor that evaluates the request and activates the corresponding methods in the name space. In figure 8 it's possible to see a simplified version of the parsing process that follows an `open` request to create a new node. The process creates the node, stores it into the kv store with the absolute path as key and its content and metadata as value, assigns to it the default ACL permissions and ultimately creates an handle on the node that permits the client to execute new operations on it until closed.

The 'write filecontent' command retrieves the node's value from the kv store, overwrites it's data and puts it back into the database. (See figure 9).

Other requests that a client may execute have a similar approach and interact with the same entities shown here, with minor differences between them. Those two cases where selected to show how entities interact with each other since they're the most complex ones and cover all major possible interactions that also occur between the other commands that clients may execute.

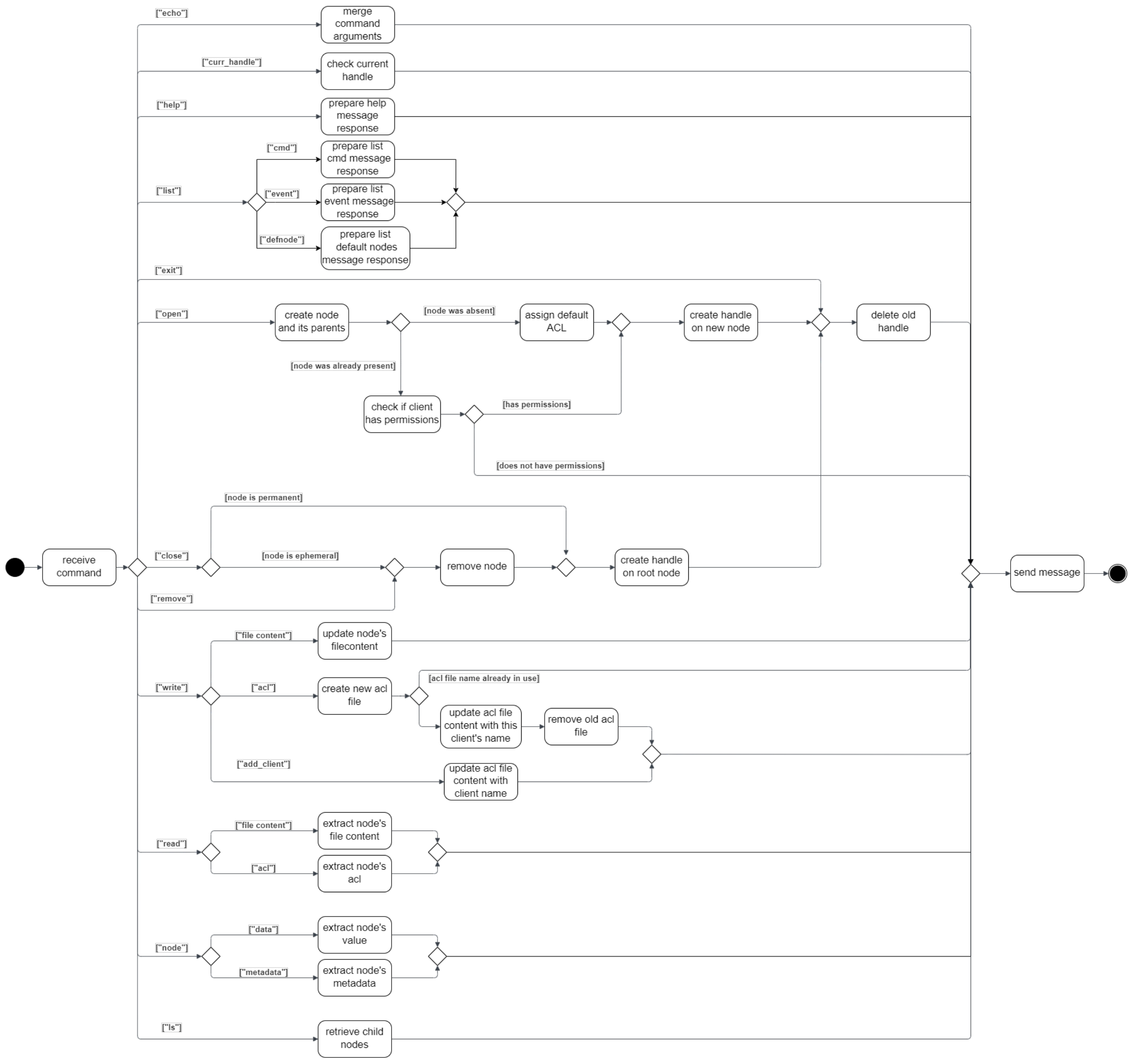


Figure 6: Activity diagram, how requests are processed
How the request is processed varies based on its type, each one eventually replies with a message that is sent back to the client about the result

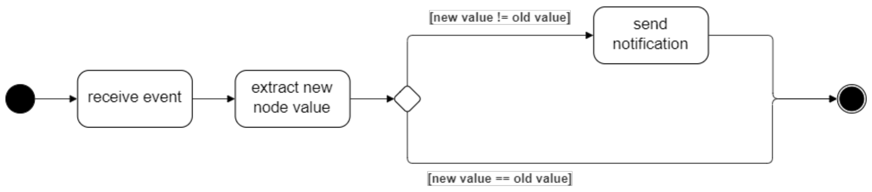


Figure 7: Activity diagram, how events are detected
If any modification is detected between previously stored value and current value, a notification about the changes is sent. This does not happen if values are identical

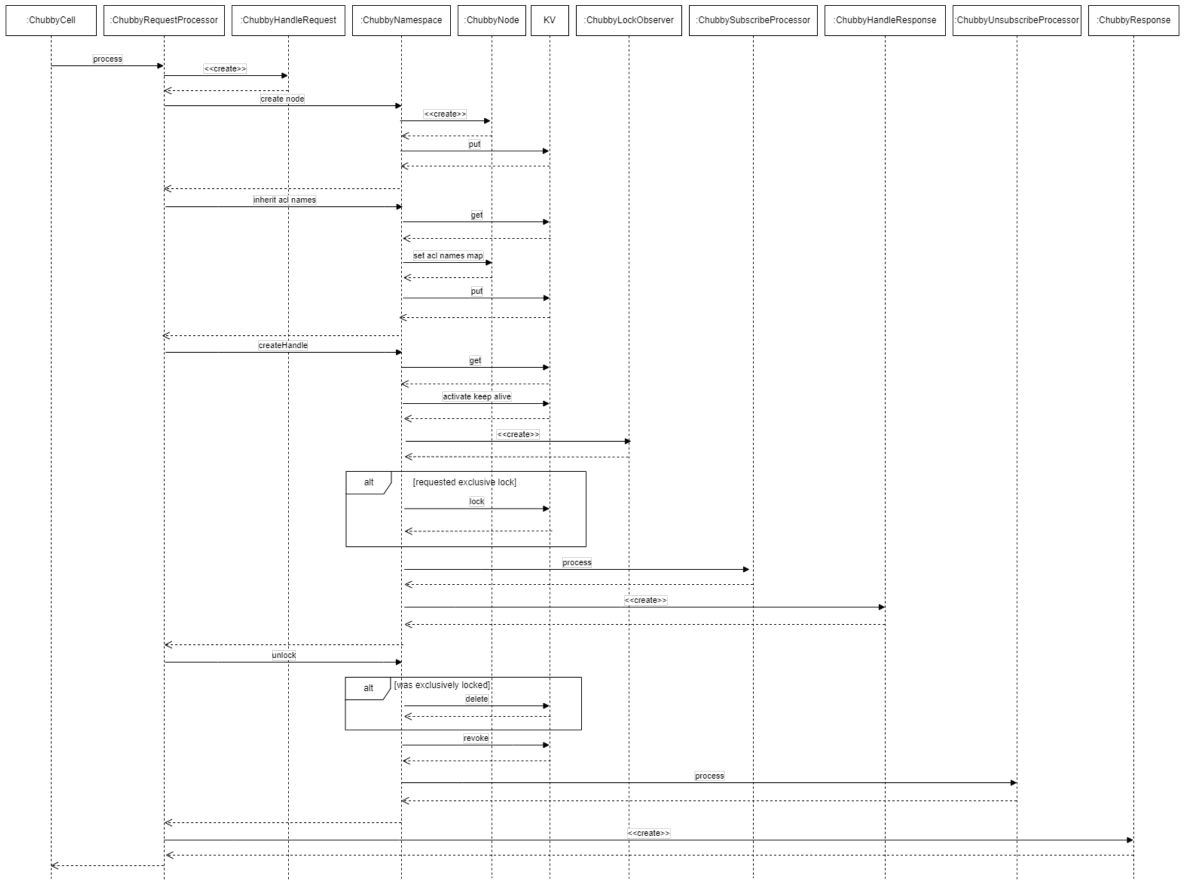


Figure 8: Sequence diagram, open command
The open command creates the node and its parents, assigns to it initial ACLs, creates a new handle on the node and removes the old one

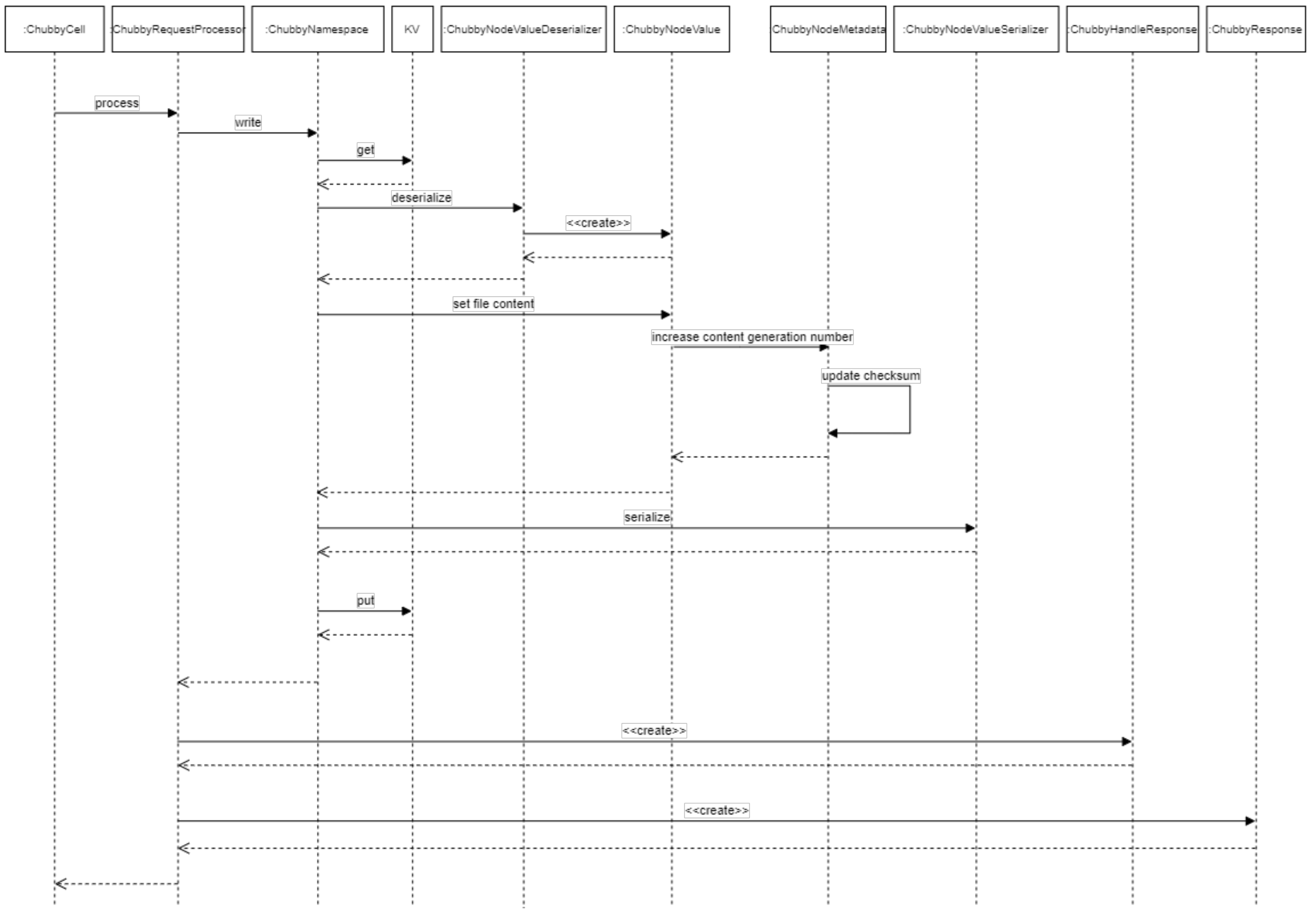


Figure 9: Sequence diagram, write command
Write command retrieves the file node from the kv store, overwrites its content, updates the checksum and puts it back into the database

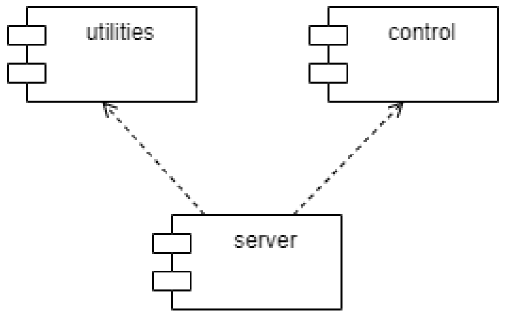


Figure 10: Components' diagram
The component 'server' has a dependency over 'utilities' and 'control', utilizing their content

4.4 Domain entities

Several entities have been created to model the system.

The module 'server' contains classes to model chubby cells, name spaces and their underlying logic. It has a dependency over the module 'utilities', that contains utility methods and custom exceptions, and the module 'control' that is responsible of modelling communication responses and node handles. (See figure 10).

Inside the module 'server' a package with the same name contains classes that model it. As shown in figure 11, class 'ChubbyCell' represents the cluster of servers with which the client communicates, each cell contains a 'Chubby-Namespace' that contains the logic to manage file and directories of the system. Clients send requests to the cell that invokes the entity 'ChubbyRequestProcessor', responsible of parsing client's requests and activating operations on the name space. If the request contains subscriptions to be activated, entities 'ChubbyLockObserver' and 'ChubbySubscribeProcessor' are invoked to manage specific event types about node handles and values.

Chubby files and directories are called 'ChubbyNodes', each one contains a 'ChubbyNodeValue' that represents the node's file content and 'ChubbyNodeMetadata' that contains specific attributes and settings of the node (as shown in figure 12):

- `checksum`, it's a 64 bits number that contains the checksum of this node's file content
- `instanceNumber`, it's a 64 bits number that increases by 1 for each parent node with the same name of this node, to correctly identify each node

- `contentGenerationNumber`, it's a 64 bits number that increases by 1 each time the file content is modified
- `lockGenerationNumber`, it's a 64 bits number that increases by 1 each time a client gets a lock on this node, if this node was previously free
- `lockRequestNumber`, it's a 64 bits number that increases by 1 each time a client *tries* to get a lock on this node, with any result.
- `lockClientMap`, it's a map that keeps track of the clients that have an open handle of any type on this node. Keys are the clients' usernames, values are the handle types.
- `aclGenerationNumber`, it's a 64 bits number that increases by 1 each time the ACL is modified
- `childNodeNumber`, keeps track of the current number of child nodes this node has.
- `aclNamesMap`, it's a map that keeps track of ACL names, keys are the permission type (read, write, change_acl) and values are the names associated to each of them.
- `chubbyNodeType`, it's an enumeration that indicates this node's type, it may be either 'file' or 'directory'.
- `chubbyNodeAttribute`, it's an enumeration indicating that this node is either a 'permanent' node or an 'ephemeral' (temporary) one.

In module 'control' a package with name 'message' contains an abstract class 'ChubbyMessage' that models each type of message that both client and server may send. Entity 'ChubbyRequest' contains the request sent by the client. Entities 'ChubbyError', 'ChubbyResponse' and 'ChubbyNotification' are sent from the server as response to previous requests. (See figure 13). Specifically, errors are sent when a request cannot be completed, responses when the client has to be notified about the successful execution of the latest request and notifications are sent asynchronously when an event matching any active subscription has been detected.

Additionally, a package 'control' manages the part of the request, sent by the client, and of response, sent by the server, that needs to provide or use an open handle on a node, typically, a 'ChubbyHandleRequest' contains the handle to be created that was requested by the client or that needs to be created after the command execution, while 'ChubbyHandleResponse' contains the handle that was created (and that will be used from now on) as response to previous command. The handle request contains the enumerations 'ChubbyEventType' that is an optional parameter that may contain subscriptions that are requested to be activated, and 'ChubbyHandleType' that contains the type of handle that needs to be created (write, read or change_acl). The 'ChubbyLockDelay' entity contains a value that sets the time between the occurrences of keep alive messages between client and server. (See figure 14).

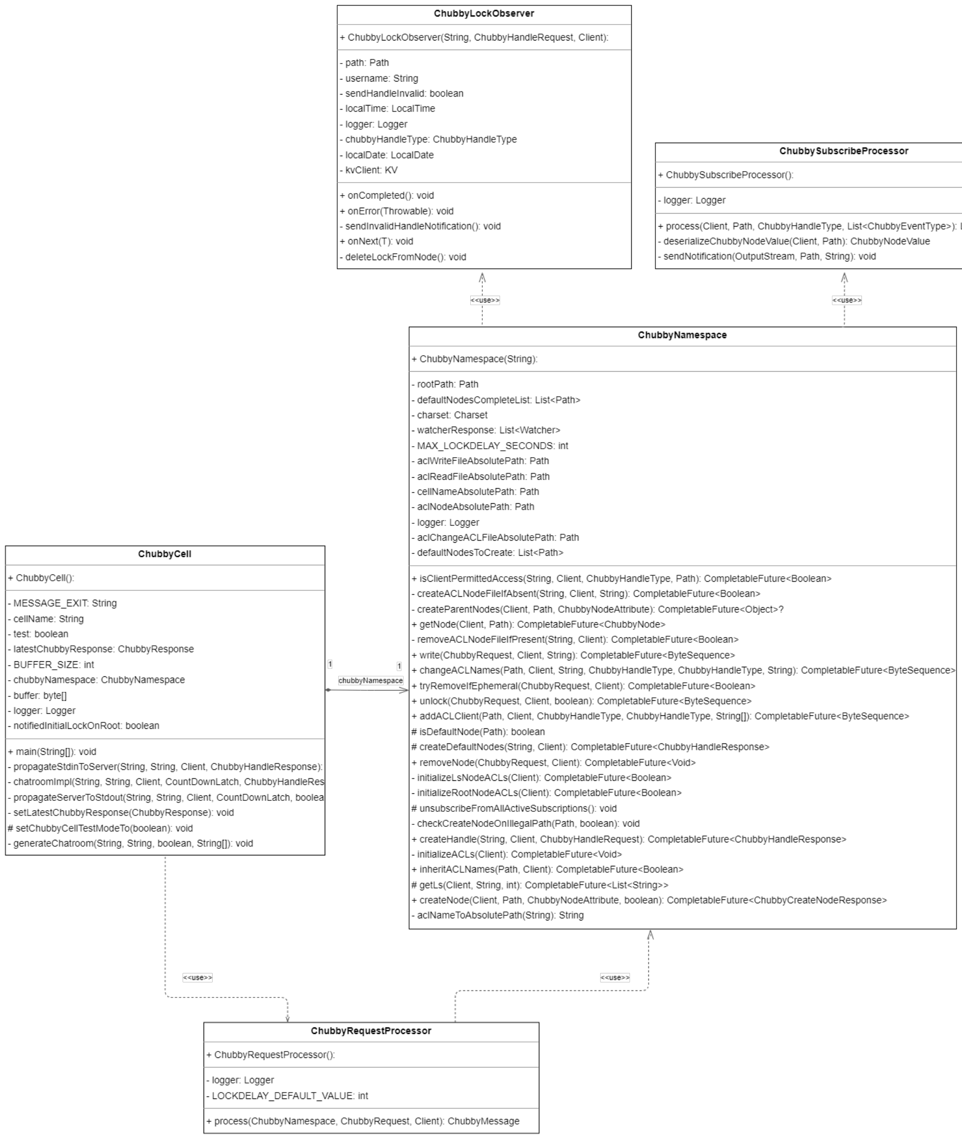

ChubbyLockObserver
+ ChubbyLockObserver(String, ChubbyHandleRequest, Client):
- path: Path
- username: String
- sendHandleInvalid: boolean
- localTime: LocalTime
- logger: Logger
- chubbyHandleType: ChubbyHandleType
- localDate: LocalDate
- kvClient: KV
+ onCompleted(): void
+ onError(Throwable): void
- sendInvalidHandleNotification(): void
+ onNext(T): void
- deleteLockFromNode(): void
ChubbySubscribeProcessor
+ ChubbySubscribeProcessor():
- logger: Logger
+ process(Client, Path, ChubbyHandleType, List<ChubbyEventType>): List<Watcher>
- deserializeChubbyNodeValue(Client, Path): ChubbyNodeValue
- sendNotification(OutputStream, Path, String): void
<<use>>
<<use>>
ChubbyNamespace
+ ChubbyNamespace(String):
- rootPath: Path
- defaultNodesCompleteList: List<Path>
- charset: Charset
- watcherResponse: List<Watcher>
- MAX_LOCKDELAY_SECONDS: int
- aclWriteFileAbsolutePath: Path
- aclReadFileAbsolutePath: Path
- cellNameAbsolutePath: Path
- aclNodeAbsolutePath: Path
- logger: Logger
- aclChangeACLFileAbsolutePath: Path
- defaultNodesToCreate: List<Path>
+ isClientPermittedAccess(String, Client, ChubbyHandleType, Path): CompletableFuture<Boolean>
- createACLNodeFileIfAbsent(String, Client, String): CompletableFuture<Boolean>
- createParentNodes(Client, Path, ChubbyNodeAttribute): CompletableFuture<Object>?
+ getNode(Client, Path): CompletableFuture<ChubbyNode>
- removeACLNodeFileIfPresent(String, Client): CompletableFuture<Boolean>
+ write(ChubbyRequest, Client, String): CompletableFuture<ByteSequence>
+ changeACLNames(Path, Client, String, ChubbyHandleType, ChubbyHandleType, String): CompletableFuture<ByteSequence>
+ tryRemoveIfEphemeral(ChubbyRequest, Client): CompletableFuture<Boolean>
+ unlock(ChubbyRequest, Client, boolean): CompletableFuture<ByteSequence>
+ addACLClient(Path, Client, ChubbyHandleType, ChubbyHandleType, String[]): CompletableFuture<ByteSequence>
# isDefaultNode(Path): boolean
# createDefaultNodes(String, Client): CompletableFuture<ChubbyHandleResponse>
+ removeNode(ChubbyRequest, Client): CompletableFuture<Void>
- initializeLsNodeACLs(Client): CompletableFuture<Boolean>
- initializeRootNodeACLs(Client): CompletableFuture<Boolean>
# unsubscribeFromAllActiveSubscriptions(): void
- checkCreateNodeOnIllegalPath(Path, boolean): void
+ createHandle(String, Client, ChubbyHandleRequest): CompletableFuture<ChubbyHandleResponse>
- initializeACLs(Client): CompletableFuture<Void>
+ inheritACLNames(Path, Client): CompletableFuture<Boolean>
# getLs(Client, String, int): CompletableFuture<List<String>>
+ createNode(Client, Path, ChubbyNodeAttribute, boolean): CompletableFuture<ChubbyCreateNodeResponse>
- aclNameToAbsolutePath(String): String
ChubbyCell
+ ChubbyCell():
- MESSAGE_EXIT: String
- cellName: String
- test: boolean
- latestChubbyResponse: ChubbyResponse
- BUFFER_SIZE: int
- chubbyNamespace: ChubbyNamespace
- buffer: byte[]
- logger: Logger
- notifiedInitialLockOnRoot: boolean
+ main(String[]): void
- propagateStdinToServer(String, String, Client, ChubbyHandleResponse):
- chatroomImpl(String, String, Client, CountDownLatch, ChubbyHandleRes
- propagateServerToStdout(String, String, Client, CountDownLatch, boolea
- setLatestChubbyResponse(ChubbyResponse): void
# setChubbyCellTestModeTo(boolean): void
- generateChatroom(String, String, boolean, String[]): void
1
1
chubbyNamespace
<<use>>
<<use>>
ChubbyRequestProcessor
+ ChubbyRequestProcessor():
- logger: Logger
- LOCKDELAY_DEFAULT_VALUE: int
+ process(ChubbyNamespace, ChubbyRequest, Client): ChubbyMessage

Figure 11: Class diagram, server package
Each cell has its own name space where resources are managed via request processor that elaborates the requests from the client and updates the name space accordingly. Subscribe processor and lock observer are activated from the name space to check for events in case of subscriptions

1

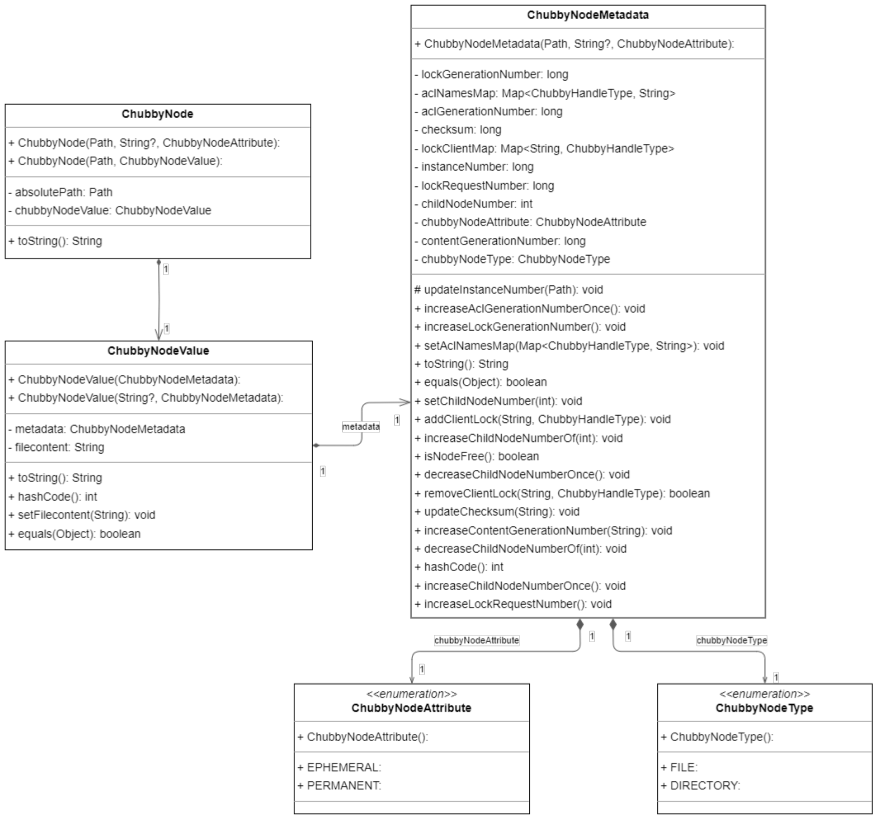
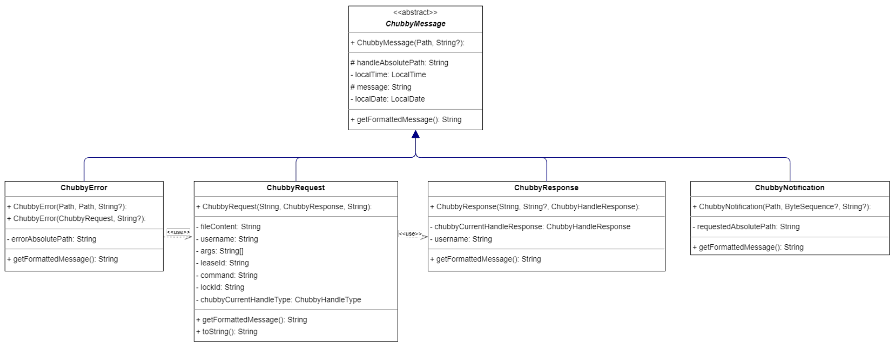
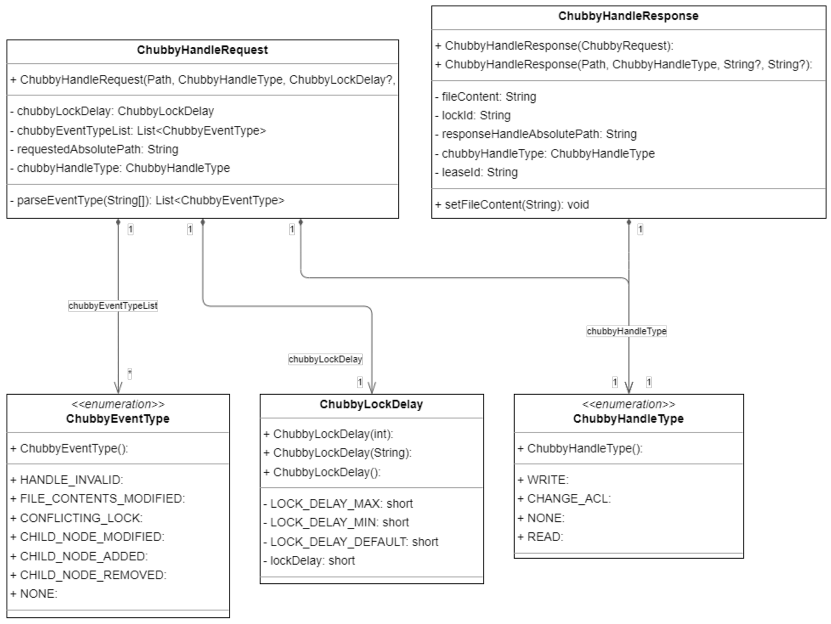

ChubbyNode
+ ChubbyNode(Path, String?, ChubbyNodeAttribute):
+ ChubbyNode(Path, ChubbyNodeValue):
- absolutePath: Path
- chubbyNodeValue: ChubbyNodeValue
+ toString(): String
1
1
ChubbyNodeValue
+ ChubbyNodeValue(ChubbyNodeMetadata):
+ ChubbyNodeValue(String?, ChubbyNodeMetadata):
- metadata: ChubbyNodeMetadata
- filecontent: String
+ toString(): String
+ hashCode(): int
+ setFilecontent(String): void
+ equals(Object): boolean
metadata
1
1
ChubbyNodeMetadata
+ ChubbyNodeMetadata(Path, String?, ChubbyNodeAttribute):
- lockGenerationNumber: long
- aclNamesMap: Map<ChubbyHandleType, String>
- aclGenerationNumber: long
- checksum: long
- lockClientMap: Map<String, ChubbyHandleType>
- instanceNumber: long
- lockRequestNumber: long
- childNodeNumber: int
- chubbyNodeAttribute: ChubbyNodeAttribute
- contentGenerationNumber: long
- chubbyNodeType: ChubbyNodeType
# updateInstanceNumber(Path): void
+ increaseAclGenerationNumberOnce(): void
+ increaseLockGenerationNumber(): void
+ setAclNamesMap(Map<ChubbyHandleType, String>): void
+ toString(): String
+ equals(Object): boolean
+ setChildNodeNumber(int): void
+ addClientLock(String, ChubbyHandleType): void
+ increaseChildNodeNumberOf(int): void
+ isNodeFree(): boolean
+ decreaseChildNodeNumberOnce(): void
+ removeClientLock(String, ChubbyHandleType): boolean
+ updateChecksum(String): void
+ increaseContentGenerationNumber(String): void
+ decreaseChildNodeNumberOf(int): void
+ hashCode(): int
+ increaseChildNodeNumberOnce(): void
+ increaseLockRequestNumber(): void
chubbyNodeAttribute
1
1
chubbyNodeType
1
1
<<enumeration>>
ChubbyNodeAttribute
+ ChubbyNodeAttribute():
+ EPHEMERAL:
+ PERMANENT:
<<enumeration>>
ChubbyNodeType
+ ChubbyNodeType():
+ FILE:
+ DIRECTORY:

Figure 12: Class diagram, node package
It contains the logic needed to model nodes, each of which has its own value and metadata



<<abstract>>
ChubbyMessage
+ ChubbyMessage(Path, String?):
# handleAbsolutePath: String
- localTime: LocalTime
# message: String
- localDate: LocalDate
+ getFormattedMessage(): String
ChubbyError
+ ChubbyError(Path, Path, String?):
+ ChubbyError(ChubbyRequest, String?):
- errorAbsolutePath: String
+ getFormattedMessage(): String
<<use>>
ChubbyRequest
+ ChubbyRequest(String, ChubbyResponse, String):
- fileContent: String
- username: String
- args: String[]
- leaseId: String
- command: String
- lockId: String
- chubbyCurrentHandleType: ChubbyHandleType
+ getFormattedMessage(): String
+ toString(): String
<<use>>
ChubbyResponse
+ ChubbyResponse(String, String?, ChubbyHandleResponse):
- chubbyCurrentHandleResponse: ChubbyHandleResponse
- username: String
+ getFormattedMessage(): String
ChubbyNotification
+ ChubbyNotification(Path, ByteSequence?, String?):
- requestedAbsolutePath: String
+ getFormattedMessage(): String

Figure 13: Class diagram, message package
It contains models for representing message requests and responses exchanged between clients and servers. Each message is formatted the same way, copying the format given by the chubby message class



ChubbyHandleRequest
+ ChubbyHandleRequest(Path, ChubbyHandleType, ChubbyLockDelay?,
- chubbyLockDelay: ChubbyLockDelay
- chubbyEventTypeList: List<ChubbyEventType>
- requestedAbsolutePath: String
- chubbyHandleType: ChubbyHandleType
- parseEventType(String[]): List<ChubbyEventType>
ChubbyHandleResponse
+ ChubbyHandleResponse(ChubbyRequest):
+ ChubbyHandleResponse(Path, ChubbyHandleType, String?, String?):
- fileContent: String
- lockId: String
- responseHandleAbsolutePath: String
- chubbyHandleType: ChubbyHandleType
- leaseId: String
+ setFileContent(String): void
1
1
1
1
chubbyEventTypeList
chubbyLockDelay
chubbyHandleType
*
1
1
1
<<enumeration>>
ChubbyEventType
+ ChubbyEventType():
+ HANDLE_INVALID:
+ FILE_CONTENTS_MODIFIED:
+ CONFLICTING_LOCK:
+ CHILD_NODE_MODIFIED:
+ CHILD_NODE_ADDED:
+ CHILD_NODE_REMOVED:
+ NONE:
ChubbyLockDelay
+ ChubbyLockDelay(int):
+ ChubbyLockDelay(String):
+ ChubbyLockDelay():
- LOCK_DELAY_MAX: short
- LOCK_DELAY_MIN: short
- LOCK_DELAY_DEFAULT: short
- lockDelay: short
<<enumeration>>
ChubbyHandleType
+ ChubbyHandleType():
+ WRITE:
+ CHANGE_ACL:
+ NONE:
+ READ:

Figure 14: Class diagram, handle package
Each handle request contains information about the handle that has to be created or owned by the client, subscriptions to activate and lock delay. The response contains the handle that was created and assigned to the client

5 Technologies Used

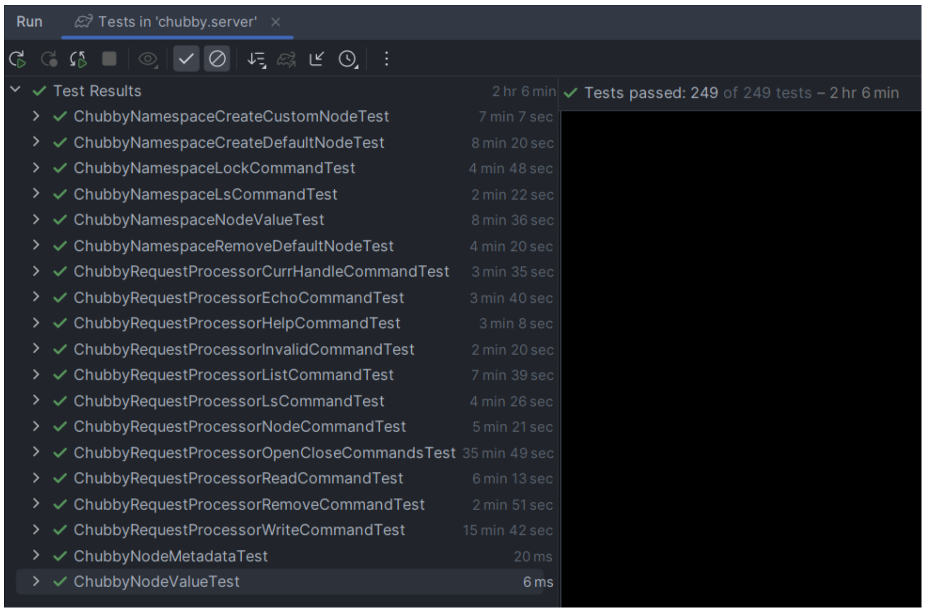
- Gradle, to automate the building process of the project, chosen for its ease of use and its support to project modularity, to separate the project in different modules that can be undertaken separately.
- Docker, to create clusters of chubby cells, chosen for its ease of use and the fact that it's supported by many different machines making it easy to create, run and deploy applications anywhere.
- Jetcd, to store files and directories as kv pairs controlled by Raft consensus algorithm. This was chosen for its robustness and because it contains several built-in functions that are mentioned in the original chubby paper [1] that were needed to fully implement each functionality that is described there, such as a distributed kv store to store files and directories and events detection.
- Gson, to transmit data in Json format between clients and servers. This was chosen for it's simplicity, robustness and its ability to easily convert complex objects to Json format.

6 Validation

In order to verify the correctness of the software produced, many types of tests, focusing on different aspects, were created.

The tests check if the node's values are correctly created and modified when necessary, and they also make sure that each command executes correctly producing the desired effect.

Note that the execution of all tests may require much time (as shown in figure 15), due to the number of tests made and especially the fact that each one needs to set up and tear down docker containers in order to correctly isolate each test, plus, a delay was added to make sure that containers are ready before the test execution begins.



```
Test Results                                              2 hr 6 min   Tests passed: 249 of 249 tests – 2 hr 6 min
  ChubbyNamespaceCreateCustomNodeTest                     7 min 7 sec
  ChubbyNamespaceCreateDefaultNodeTest                    8 min 20 sec
  ChubbyNamespaceLockCommandTest                          4 min 48 sec
  ChubbyNamespaceLsCommandTest                            2 min 22 sec
  ChubbyNamespaceNodeValueTest                            8 min 36 sec
  ChubbyNamespaceRemoveDefaultNodeTest                    4 min 20 sec
  ChubbyRequestProcessorCurrHandleCommandTest             3 min 35 sec
  ChubbyRequestProcessorEchoCommandTest                   3 min 40 sec
  ChubbyRequestProcessorHelpCommandTest                   3 min 8 sec
  ChubbyRequestProcessorInvalidCommandTest                2 min 20 sec
  ChubbyRequestProcessorListCommandTest                   7 min 39 sec
  ChubbyRequestProcessorLsCommandTest                     4 min 26 sec
  ChubbyRequestProcessorNodeCommandTest                   5 min 21 sec
  ChubbyRequestProcessorOpenCloseCommandsTest             35 min 49 sec
  ChubbyRequestProcessorReadCommandTest                   6 min 13 sec
  ChubbyRequestProcessorRemoveCommandTest                 2 min 51 sec
  ChubbyRequestProcessorWriteCommandTest                  15 min 42 sec
  ChubbyNodeMetadataTest                                  20 ms
  ChubbyNodeValueTest                                     6 ms
```

Figure 15: Validation test results
Those tests were made to verify the correctness of the software produced for each scenario

7 Deployment instructions

In order to successfully launch the project and its tests, make sure to have Docker, Gradle and Git installed. The commands showed in this section will only refer to 'windows' command syntax.

Clone the project using

```
git clone https://github.com/MarcoFontana48/jGoogleChubby.git
```

Once the project is downloaded, use the following command to set up 3 isolated chubby cells 'local', 'cell1', 'cell2', to connect to:

```
docker stack deploy -c ./docker-compose0.yml local;
docker stack deploy -c ./docker-compose1.yml cell1;
docker stack deploy -c ./docker-compose2.yml cell2;
gradle etcd_setup
```

To connect to the cells using a custom gradle task, use the following command to access the 'local' cell using an already registered user 'client0' with password 'password':

```
gradle run_client_0-local
```

Other tasks are already present to also connect to different cells using different clients, check file 'server/build.gradle.kts'

Once done, to delete all the containers previously created, use the command:

```
docker stack rm local; docker stack rm cell1; docker stack rm cell2
```

8 Usage example

8.1 how messages are formatted

Before showing some usage examples, it's necessary to explain how the messages are formatted. Each message type is formatted the same way:

```
[date time] username-messageType:(handleType)path
```

i.e. a typical request from a client 'client0' with an open handle on path '/ls/local/test.txt', that will be used to 'write' content in it, is the following:

```
'[2024-03-13 09:45:12] client0-request:(write)\ls\local\test.txt<
read filecontent'
```

and the response from the server:

```
'[2024-03-13 09:45:12] chubby-response:(write)\ls\local\test.txt>
hello world!'
```

8.2 examples

The moment a client connects to a cell, a notification from the server, containing a message, will be sent

```
'[2024-03-13 09:21:43] chubby-notification:\> acquired initial
shared lock on root '\' node'
```

This informs the client that an initial handle (read type) on root node has been successfully created.

In order to open a new handle use command 'open'; i.e. to open an handle on a 'test.txt' file that will be used to write file content (on a cell named 'local'), the following command may be executed:

```
'open /ls/local/test.txt write'
```

It will now be shown the request that was sent to the server and the response that it's sent back to the client:

```
'[2024-03-13 09:31:30] client0-request:(read)\< open /ls/local/
test.txt write'
'[2024-03-13 09:31:31] chubby-response:(write)\ls\local\test.txt>
successfully opened node'
```

This informes the client that the handle was successfully opened and that it now has an handle on the specified path '/ls/local/test.txt' of 'write' type. The client may now write file content in it:

```
'write filecontent hello world!'
```

```
'[2024-03-13 09:42:48] client0-request:(write)\ls\local\test.txt<
write filecontent hello world!'
'[2024-03-13 09:42:48] chubby-response:(write)\ls\local\test.txt>
node content updated successfully'
```

To read the file content, use the read command:

```
'read filecontent'
```

```
'[2024-03-13 09:45:12] client0-request:(write)\ls\local\test.txt<
read filecontent'
'[2024-03-13 09:45:12] chubby-response:(write)\ls\local\test.txt>
hello world!'
```

If a command that cannot be executed with current handle is sent to the server, an error response will be received:

```
'[2024-03-13 09:48:48] client0-request:(write)\ls\local\test.txt<
write acl read new_read_acl_name'
'[2024-03-13 09:48:48] chubby-error:\ls\local\test.txt> it's not
possible to change acl names of current node with 'WRITE' handle
type, acquire 'CHANGE_ACL' handle type before proceeding
```

Once done, close the handle using '`close`' command, that automatically unlocks the node and creates a new handle (read type) on root node.
About event subscriptions, if 'client0' opens a directory and activates the `child_node_added` subscription, those messages are going to be exchanged:

```
'[2024-03-13 10:01:36] client0-request:(read)\< open \ls\
local\test_dir change_acl child_node_added'
'[2024-03-13 10:01:36] chubby-response:(change_acl)\ls\
local\test_dir> successfully opened node'
```

If 'client1' now creates a child node of 'test_dir' on the same cell:

```
'[2024-03-13 10:02:03] client1-request:(read)\< open \ls\local
\test_dir\test_child_dir change_acl'
'[2024-03-13 10:02:04] chubby-response:(change_acl)\ls\local
\test_dir\test_child_dir> successfully opened node'
```

'client0' will receive the following notification:

```
'[2024-03-13 10:02:03] chubby-notification:\ls\local\test_dir>
number of child nodes from currently hold node has increased'
```

That informs the client that the number of child nodes has increased, and the directory from where the event took place: '\ls\local\test_dir'.

Many other examples are provided from the tests that were made, please refer to test classes for usage examples; also, once connected to a cell, the command 'help' may be executed for a detailed explanation of the possible commands that may be run.

9 Future works

The official Chubby paper also describes some optional functionalities that may be added to make the system scale more in case of necessity:

- **Proxies**, one or more servers may be added to receive requests in order to reduce traffic load on the master. This however adds a level of complexity, since more machines can fail and need to be handled.
- **Partitioning**, single name spaces contained inside a chubby cell may be partitioned into multiple isolated instances, each one with its own set of replicas and master.
- **Mirroring**, a 'global' cell that contains a copy of each directory and file resource created in each chubby cell at any given time.

References

[1] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. *Google Inc.*, 2006.

[2] Ousterhout John Ongaro, Diego. In search of an understandable consensus algorithm (extended version).