SimpleChubby: a simple distributed lock service

Jing Pu, Mingyu Gao, Hang Qu

1 Introduction

We implement a distributed lock service called SimpleChubby similar to the original Google Chubby lock service[1]. It intends to help implement distributed synchronization.

SimpleChubby exposes a simple Unix-like file system interface combined with lock primitives and event subscription. Leveraging such interface, we build three distributed synchronization applications, namely leader election, group membership, and distributed barriers.

SimpleChubby uses a server-client architecture, and we put great efforts to achieve fault tolerance where [1] does not provide many details. Server fault tolerance is achieved by combining the Viewstamp protocol [4] which we implement from scratch and a server-client cooperation mechanism to recover non-persistent server states. Server failure is transparent to users, and client failure is handled correctly by reclaiming its holding locks.

In the following, SimpleChubby API and its semantics are described in section 2, the system architecture and normal-case processing mechanism are described in section 3, the fault tolerance mechanism is discussed in section 4, and in section 5, three applications are shown to illustrate how SimpleChubby API is used to build distributed synchronization applications, and to give a detailed description about the system workflow.

2 User API

SimpleChubby user API consists of a file system interface for serialized file access and modification, a set of lock primitives, and an event subscription interface. The underlying file system is organized similar to Unix, which consists of a strict tree of files and directories. Both files and directories are called *nodes*.

```
/* File and directory operations */
FileHandlerId fileOpen(const string &fname,
        Mode mode);
void fileClose(FileHandlerId fdId);
bool fileDelete(FileHandlerId fdId);

/* Content read and write */
bool getContentsAndStat(FileHandlerId fdId,
        FileContent *content,
        MetaData *data);
bool setContents(FileHandlerId fdId,
        const FileContent &content);

/* Lock operations */
void acquire(FileHandlerId fdId);
bool tryAcquire(FileHandlerId fdId);
void release(FileHandlerId fdId);

/* Event subscription */
void registerEvent(ChubbyEvent e,
        EventCallback cb);
void deleteEvent(ChubbyEvent e);
```

Figure 1: SimpleChubby API

The full list of the API is shown in Figure 1. An application calls `fileOpen()` to retrieve a file handler associated with an opened node. Similar to UNIX's implementation, this function returns an integer type of identifier for each file handler, which is used by all other functions in the API. We simplify the `fileOpen()` interface by removing the argument of the parent handler, and always specify an absolute path of a node. The `mode` argument in the `fileOpen()` consists of a set of flags, including `READ`/`WRITE` mode, and create `NEW_DIRECTORY` flag. By setting `EPHEMERAL` flag, a client can create ephemeral nodes, which will be automatically removed when the connection fails.

Clients may also subscribe to a range of events during `fileOpen()`, including “file contents modified” and “lock state changed”, using the `mode` arguement. The events are delivered to the clients asynchronously. The event handler callback functions are supplied and registered by user applications beforehand, and will be called by the client library when corresponding events are received.

Our API supports a simplified failover model of Chubby. The server failures are transparent to user applications if the connection can be reconstructed within a certain grace period, otherwise an exception will be exposed to the applications and all the locks it holds are assumed to be invalid. If a client fails to connect to the leader server, its holding locks will be reclaimed after a certain timeout.

The user applications can use this API to perform various synchronization jobs. We provide three examples in Section 5.

3 System Architecture

3.1 Overview

SimpleChubby implementation consists of a client library linked to user programs, and a server executable running on each server. When user code invokes the client library interfaces to access files or perform lock operations, the client library initializes corresponding remote procedure calls (RPCs) to the leader server, waits for replies, and returns the result back to the user.

Client side. The client library offers a set of interfaces for users to access SimpleChubby lock service. It sends user requests to the leader server through PRCs, and receives any events from the leader server. The client library also maintains a set of states, including opened file handlers, subscribed events, and outstanding lock acquire requests. In addition to sanity checks, it uses this information to help server failover. Once an older leader fails, the client library tries to connect to a new leader, and send its states which are necessary to reconstruct server’s non-persistent states.

Server side. SimpleChubby runs on a group of servers to provide a reliable lock service. A server runs as either the leader server or a follower server, decided by a consensus protocol. When running as a follower server, it serves as a replica in the consensus protocol, and prepares to be promoted to a leader server.

When running as a leader server, it processes requests from clients and send events to clients. The leader server also monitors client failures, and clears the states associated with the failed clients. The leader server maintains two categories of states: persistent states and in-memory states. The persistent states, including lock ownership and file system content, are replicated among replicas, and thus accesses or updates to them go through the consensus protocol. The rest of states are not replicated through the consensus protocol, and thus will be lost after a leader change.

SimpleChubby does not make all the leader server states persistent because of the overhead of the consensus protocol. Instead, non-persistent states can be recovered using client side information. We talk about the failover mechanism in Section 4.

3.2 Server

The SimpleChubby leader server keeps two types of states, in-memory data structures that will be lost after a leader change, and persistent data that is replicated among leader and follower servers using the Viewstamp protocol.

3.2.1 In-Memory Data Structures

SimpleChubby uses a file handler mechanism. The file handler can be very handy when an application is sensitive to obsolete files. Since the server rejects any requests with invalid file handlers (i.e. meta-data in the file handler does not match with server’s record), the client detects that the node has been deleted or re-created after receiving a failure of an operation with the previously opened file handler. The server also rejects any fake file handlers by checking the digital signitures in them signed during successful `fileOpen()` operations.

However, maintaining the information of file handlers persistent on the servers can be challenging, since queries for file handlers are frequent and the lengths of the opened file handler lists dynamically change. Therefore, we choose to keep this information in memory instead of in persistent store, in order to reduce the overhead of the Viewstamp protocol. In the normal case, a file handler is created in `fileOpen()` and destoyed in either `fileClose()` or `fileDelete()`. All requests from clients except `fileOpen()` take file handlers as arguments, and the server returns failures if the file handlers do not match with server's records.

Another part of in-memory data structures is the queues of outstanding lock acquire requests for different nodes. Upon any `acquire()` request, the leader server adds the client's session information into the queue and blocks the RPC if the lock is held by another client. In a `release()` request, the server pops out a waiting session (if there is any) on the same lock, modifies the lock owner, and unblocks and returns the correspongding `acquire()` request.

We also store clients' request records for events in the leader server's memory. For each type of event, there are separate lists for different nodes. The client's session information will be added to the lists during `fileOpen()` request if event registration flags are set. Once a event happens, the server iterates through the list, and sends the event to the client if the session is still alive.

3.2.2 Persistent Data Store

The main part of persistent data is the Chubby file system. Each node has a full path name as the key, a string of content, the meta data defined in the SimpleChubby API, and a lock owner field. A normal node open returns the instance number of the node in order to form a file handler, while a node creation triggered by `fileOpen()` takes a unique ascending instance number from server and creates a node in the file system. All other operations on the filesystem (including the lock operations that change lock owners) take both the name and the instance number of the node as arguments, and will abort if the node does not exist or the instance number doesn't match the meta data.

Another persistent data is the value of the next instance number. This global value is increased by one every time a new node is created, and is passed to the node creation operation as discussed above. It works as a timestamp, which helps the server detect and reject the client requests with obsolete file handlers.

3.3 Replication and Leader Election

SimpleChubby implements a replicated state machine to store the persistent state of SimpleChubby servers, i.e., file content, lock ownership and etc., following the implementation described in [3]. It is based on the Viewstamp protocol [4], and the view-change part is implemented using Paxos [2].

The consensus implementation runs in the same process as the SimpleChubby server logic. The leader in a Viewstamp view is appointed as the SimpleChubby leader server.

Our consensus implementation targets a correct consensus behavior rather than pushing hard for good performance. To make our implementation simple, a replica loses all its data and states after rebooting, which increases the overhead of bringing up a reboot replica.

3.4 Client library

The client library of SimpleChubby is responsible for sending RPCs to the leader server, and returning the results to the user-level applications. To avoid overwhelming the server, the client process will block until a RPC gets replied. This implies that only one outstanding RPC is allowed between a client and the leader server.

In addition, the client library needs to monitor event messages from the server and execute up-call to the user level. Since a user thread might be blocked by such operations as `acquire()`, events have to be delivered asynchronously to another thread to guarantee that events are handled in time. With event subscription, a client can avoid polling state changes at the server. This is necessary for the applications we implemented in Section 5,

Finally, when the leader server fails, the client library needs to figure out the new leader after server-side view change, and helps to recover some non-persistent states at the server side. We will talk about the details in Section 4.

4 Fault Tolerance

4.1 Client Failure

During a client failure, the leader server releases all the locks previously held by the failed client after a certain timeout. This guarantees that locks are eventually released and no deadlock happens.

For garbage collection purpose, the server also deletes all the file handlers opened by the client, outstanding lock acquire requests sent by the client, and all the registered events from the client.

Each client maintains a TCP connection with the leader server, which is called a session. The session is dead if either end shuts down the TCP connection. And the leader server will assume a client failure once the session is dead. Notice that this is a simplified implementation. If a client dies before explicitly shutting down the TCP connection, the client failure cannot be detected.

4.2 Server Failover

When the leader server fails, the Viewchange protocol elects a new leader. After the leader change, all persistent data remain consistent. However, the in-memory data structures at the new leader need to be reconstructed.

Two methods are used for the reconstruction. First, a client eagerly sends some of its states to the new leader after a new connection is established, which helps recover the in-memory structures of the leader, including the queues of outstanding lock acquire requests and the lists of event subscriptions from clients. In particular, the client library keeps a copy of subscribed events and its outstanding lock acquire requests. After a leader change, a client connects to the new leader, sends reopen requests containing the previously subscription of events, and finally send an `acquire()` request if there is any outstanding request. On the other side, the leader accepts any reopen requests with valid file handlers, and adds the event subscriptions to its in-memory data structure.

However, the in-memory data structure of file handlers does not need to be reconstructed eagerly. The second method we use is to lazily recover the the file handler at the server side during the normal case of processing SimpleChubby requests. If the new leader receives a request with a file handler that is missing in the leader's memory, it checks the signiture of the file handler and initiates a reopen. If the reopen succeeds (i.e. the node exits and the meta data matches), the leader adds this file handler into its memory.

5 Application

We implemented three example user applications using the SimpleChubby service, namely leader election, double barrier, and group membership. These applications use the API described in Section 2, and are able to survive both client and server failures. They demonstrate that the functionality is correct and the interface is easy to use.

5.1 Leader Election

Leader election is an application that elects one out of a given group of clients as the leader. At any given time, at most one client among the group can be elected as the leader, and other clients should be able to find out who is the leader, if it exists.

We implement leader election as shown in Figure 2: all clients open a single specified file and

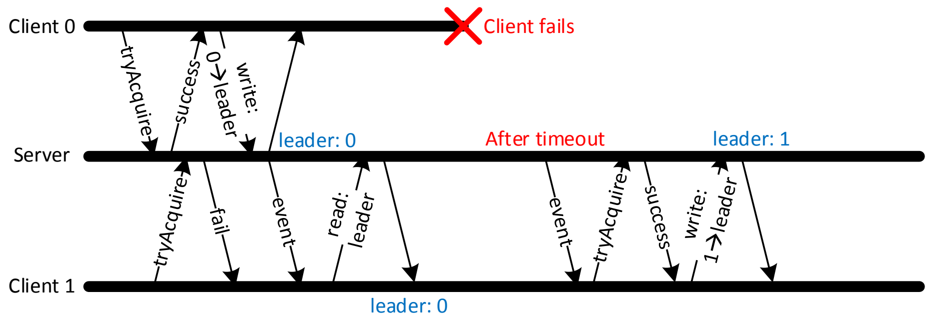

Figure 2: Leader election. Two clients compete to be the leader. Client 0 wins, write itself into file "leader", but fails later. The server reclaims the lock and notifies client 1, which becomes the new leader after acquiring the lock.

try to acquire its lock. The one who succeeds becomes the leader, and write its identity into the file. Others read the file to figure out who is the leader. The leader continues to hold the lock. Also, all clients subscribe to the "lock state changed" event, in order to compete for the new leader when the current leader dies. Clients also subscribe to the "file content modified" event to read out the new leader when changed.

5.2 Double Barrier

Double barriers enable a given number of clients to synchronize at the beginning and the end of a computation region. When the number of processes that have joined the barrier reaches the specified number, all processes start their computation, and leave the barrier together once they have all finished.

In SimpleChubby, double barrier is implemented using two (single) barriers. We implement the barrier as shown in Figure 3: all clients open, lock, and update one "count" file that store with the number of clients reaching the barrier. Then they wait until a "file content modified" event happens with another "passed" file. When the last client finds out that the required number achieves, it writes to the "passed" file to trigger the event, and cleans up the "count" file.

5.3 Group Membership

Group membership allows all the members inside a specific group to monitor the group membership changes, such as new member added, normal member leaving, or member failure.

We implement this application using the directory feature in SimpleChubby. When entering the group, the member creates a ephemeral file in the group directory with the name of itself, and query the directory content to get the list of all members. The members also subscribe to the "directory content modified" event of the group directory, so that they can obtain the most up-to-date membership. When leaving the group, the member deletes its file. The ephemeral feather ensures the correctness even at failure.

6 Conclusion

We implement a simplified version of Chubby. It provides a well-defined API to build distributed synchronization service , and handles client failure and server failover correctly. We build three applications on top of it to demonstrate its functionality.

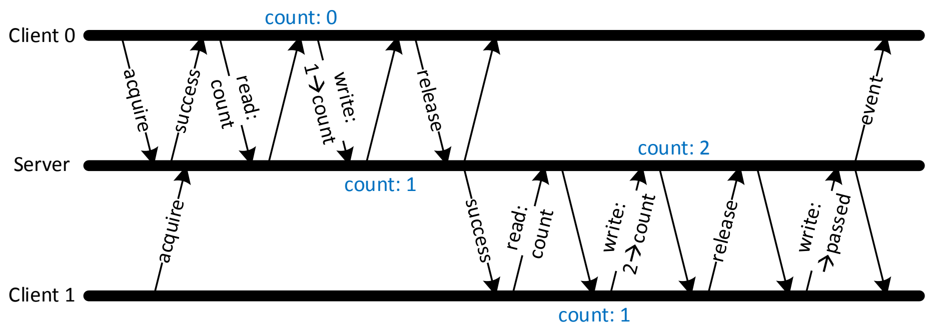

Figure 3: Barrier with two clients. Client 0 comes first and set the "count" file content to be 1. Client 1 comes later and set the "count" file content to be 2. Client 2 finds the requirement for the barrier is satisfied and writes to "pass" file, which ends the barrier by delivering events to all clients.

References

[1] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[2] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[3] D. Mazieres. Paxos made practical. Technical report, Technical report, 2007. http://www. scs. stanford. edu/dm/home/papers, 2007.

[4] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17. ACM, 1988.