Skinny: a simplified version of Chubby

Cody Tseng
UW-Madison

Wuh-Chwen Hwang
UW-Madison

Ping-Han Chuang
UW-Madison

Abstract

We implemented Skinny, a distributed lock service that resembles Chubby, on top of Raft consensus protocol implementation NuRaft and remote procedure call framework gRPC. It has most of the Chubby's features, including locks, small file read/write, events, ephemeral files, and client-side caching, and is consistent and highly-available. We conducted performance evaluations of Skinny.

1 Introduction

Chubby [1] is a lock service built on top of loosly-coupled distributed system, presented by Google. It is intended to be used as a course-grained lock service with storage for small files (perhaps configurations), and is guaranteed to be highly-available and consistent. The system exposes an interface similar to a file system with advisory reader/writer locks. In this project, we built a Chubby-like distributed lock service, Skinny, and implemented most features of Chubby. The report is structured as follow: we first define the scope of our project (supported features), and introduce the structure of our system Skinny. We then dive into the server-side implementation, which covers how Skinnymanages sessions, files and directories, file handles, locks, events, and cache invalidation. Following it is the details of the client library, which consists of automatic Skinnyleader finding, session management, and file cache. The server-client and client-user interfaces follows, and we showcased the correctness tests and performance evaluations. We listed the lessons learned from this project in the end, concluding this report.

2 Supported Features

We start with implementing the core features and gradually include many Chubby's handy functionalities, including:

- client session
- file open/close/delete
- file read/write
- reader/writer (share/exclusive) lock
- single kind of event
- directory
- client-side cache
- ephemeral files

We do not implement access control list (ACL), snapshot, proxy, lock sequencer/lock-delay, and lock upgrading (from reader to writer lock). There is only one kind of events (notify while file/directory changes).

3 System structure

The server (cluster) and the client library collectively build the Skinnyservice, giving users a consistent view of the lock (and file system) state, and hide server failures. Figure 1 showcases the high-level structure of the system.

3.1 Server

3 or 5 servers compose a server cluster, which is called a Skinnycell. Each server has a user-facing port that runs gRPC server, handling client requests. The Raft consensus protocol is run across all servers, which is used as the leader election protocol, and all states (locks, files, sessions, and file handles of each session) changes are replicated through such protocol. In our implementation, we use NuRaft, a lightweight C++ Raft implementation library presented by eBay [3]. Different to Chubby, we

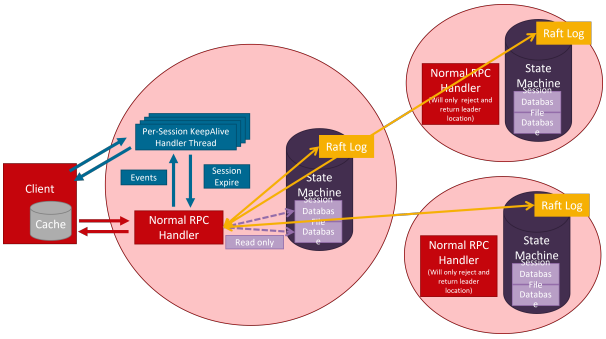

Figure 1: System structure diagram

only keep states in memory instead of persisting them. As Raft already tolerates failures and gives us linearizable consistency, we believe it is sufficient to keep states in memory. If a server is not the leader, it will reject client requests and respond with an error with the current leader number.

3.2 Client Library

The client library has addresses of all servers, gives a user-friendly API to the end user, and invokes RPC calls to server on behalf of the user. In particular, it creates a session on start, keeps the session active by sending keep-alive RPC calls in the background until the user explicitly requests to tear down , on which it sends end-session RPC to the server to drop the session. If the leader (of servers) fails, client library automatically finds the newly elected leader and resume the session, creating the illusion to end users that Skinnyservice is highly-available. Client library is also responsible for handling events and caches files to expedite reads.

4 Design details

4.1 File, directory, and file handle

Files and directories are hash-mappings from filename to the file content and metadata. We use the entire path name as the key to identify a file, which is identical to Chubby's approach. File metadata indicates whether a file is ephemeral, and keeps track of which session(s) is/are the lock owner(s), and if there is only a single lock owner, whether the lock is held in shared (reader) or exclusive (write) mode. To distinguish old file handles that refer to the deleted files instead of their newly created counterparts that shares the same pathnames, there are instance number and "file exists" fields that keep track of the file version. Sessions that opened the file is also kept so that events and cache invalidation messages can be sent to appropriate client sessions.

A directory is a file with "is_directory" field set as true, and its file content is all the containing files' name, delimited by '
0'.

A file handle in a number given to the client to represent the opened file. The server's session library keeps track of files opened by each session by maintaining an array of opened files' pathname, and the instance number of the files when they are opened. File handle is such entry's vector index. This is crucial for distinguishing valid file handles from those that refer to files being deleted. If the file handle's instance number is different from that of the current file's metadata, it means that the file has been deleted and re-created after this file handle was created, and all operations on such file handle should be rejected.

4.2 Locks

Reader-writer locks are implemented so that multiple readers can acquire a shared lock at the same time, given that there is no writer (which acquires the exclusive lock). We provide both blocking (Acquire) and non-blocking (TryAcquire) interfaces to the user. Lock sequencer and lock delay are Chubby's approaches to prevent late, invalid requests from the previous lock owner to be successfully executed. Since we do not provide interfaces to check whether users hold the lock before handling the request, we implemented neither of these. Moreover, since lock states are fully replicated in our system, simply checking who is the current lock owner should prevent the invalid requests from execution, so we still do not need these techniques.

4.3 Events

Users can listen to events to the files they opened. When opening a file or a directory through client library, the user can specify a callback function, which will be called when the client library receives the event notification from the server. Currently, users cannot specify which event they want to listen to. Instead, a notification will be sent when a file content is changed or deleted, or when a directory has new files or existing files are deleted. Events notifications are piggybacked in KeepAlive RPC calls response, described in the later subsection.

4.4 Cache invalidation

Client caches file contents that it opened and read. The leader server remembers sessions that has opened the file in the file metadata to keep file caches coherent at all time. As such, when multiple clients open a file and one of them requests a write, the (leader) server has to

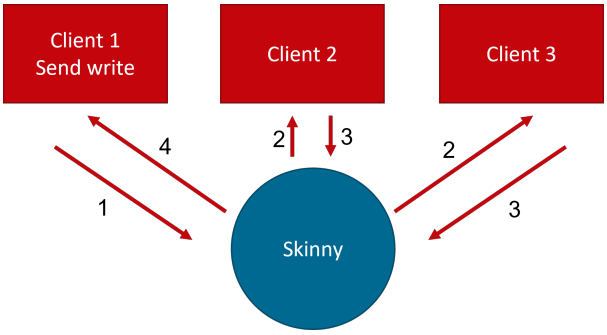

Figure 2: Cache invalidation procedure

send cache invalidation notifications (which is also piggybacked in KeepAlive's response) to all other sessions that open the file and cannot return the write requests until it receives notification acks from those sessions (or discovers such sessions are dead). Figure 2 shows the steps of cache invalidation.

4.5 Session management

Client session is maintained by clients keep sending heartbeat messages called "KeepAlive" to the server. If the server does not hear from the client for a while (currently 5 seconds), the session times out and is revoked. On the other hand, if the client does not receive the response to KeepAlive call for a certain amount of time (we set it to 10 seconds) after invoking the call, it thinks the leader server went down and initiates the protocol that attempts to resume its session at the new leader.

On the client side, on receiving the response of the previous KeepAlive call, the client library will check whether the response contains event notifications or cache invalidation messages, and react appropriately. After that, it will immediately send a new KeepAlive request to the leader server. On the server side, a designated session-managing thread is created per session (we called it "KeepAlive thread", or "KAThread"). This thread waits until (1) a new KeepAlive request arrives: it resets the timer and return the call with the queued cache invalidation message or event notification, if any; (2) a cache invalidation or event notification comes to this session: depending on there is a blocked KeepAlive request, piggyback the message in the response or save the message in queue; (3) a EndSession request arrives, or time out: kill the session.

4.6 Leader failover

We rely on Raft to detect and conduct a view change, which elects the new leader. As such, we do not need to implement master lease, which is used in Chubby that promises all non-leader servers will not elect a new leader for a certain period of time. NuRaft exposes an interface for all servers learning the current leader. The newly elected leader knows the active sessions (by state replication) but does not know the values of their countdown timer, which apparently should not be replicated. It launches the new KeepAlive handling threads, which effectively resets the session expiration timer to the longest possible duration.

Client library observes leader failures by not receiving the response of KeepAlive call within timeout, currently set to 10 seconds. It invalidates cache and keeps attempting to resume the session on the other server, which will accept the request if it is the new leader. Before client library finds the new leader, it will block all client requests.

5 RPC call and user API

RPC calls are communications between the client library and the server, and the client library exposes a set of user APIs to the end user.

5.1 RPC Call

Including session management (StartSession, EndSession, KeepAlive), file open, close, and delete, read (GetContent) and write (SetContent), and lock acquisition (Acquire, TryAcquire, Release). Except for opening a file/directory, all file/directory related RPC calls takes file handles instead of pathname, thereby keeping most states on the server-side.

```
(session_id) StartSession ()
() EndSession (session_id)
(fh) Open (session_id, path, is_directory)
() Close (session_id, fh)
(content) GetContent (session_id, fh)
() SetContent (session_id, fh, content)
(res, msg) Delete (session_id, fh)
(res, msg) TryAcquire (session_id, fh, ex)
(res, msg) Acquire (session_id, fh, ex)
(res, msg) Release (session_id, fh, ex)
(fh, event_id)
    KeepAlive (session_id, ack_event)
```

5.2 User API

Client library exposes APIs that are similar to that used by the communication between the server and itself, except that there is no session management interfaces (the library handles them), and that user can specify whether they are opening a file or a directory (to set the

"is_directory" metadata field) and pass in the callback to be invoked when the library receives event notifications.

```
(constructor)[calls StartSession() RPC]
(destructor)[calls EndSession() RPC]
(fh) Open (path, callback, is_ephemeral)
(fh) OpenDir (path, callback, is_ephemeral)
() Close(fh)
(content) GetContent(fh)
() SetContent(fh, content)
() Delete(fh)
(bool) TryAcquire(fh, ex)
() Acquire(fh, ex)
() Release (fh)
```

6 Correctness Testing

To ensure the correctness of aforementioned functionalities, we wrote and ran a few tests. Surprisingly, testing takes much more time than we expected, as a lot of race condition arises. We bind the C++ client library to Python code, so we can both use the service interactively (for initial checks) and write automated test suites/launch and kill servers in PyTest. We tested basic read/write and locks with thousands of client sessions, as well as events, client cache invalidation, and leader failover. The following is a list of tests we conducted.

Read/write

- Single writer, many writes
- Single writer, one write, many readers

Lock

- Single user Acquire, TryAcquire, Release; Acquire after Release
- Multiple users contending (non-blocking) the same lock
- Multiple users wait (blocked), acquire, and release the same lock
- Single writer Acquire after multiple readers Release
- Multiple readers Acquire after a single writer Release

Event

- Event callback invocation
- Built a distributed barrier, which is used in performance measurement

Cache

- Cache invalidation under client failures

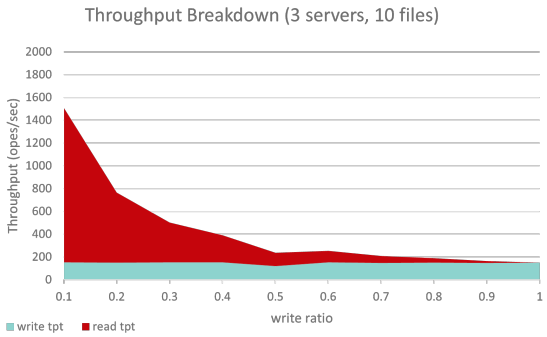

Figure 3: Throughput breakdown, varying read/write ratio. 150 clients access 10 files on a Skinny cluster of 3 nodes.

Leader failure

- Client library correctly finds the newly elected leader

7 Performance Measurement

We stress-test our system and measure the throughput . We also measure single read/write request latency.

7.1 Hardware

We spawned 10 CloudLab [2] c8220 machines to conduct our performance measurement. Each c8220 node has two Intel E5-2660 v2 CPU that has 10 logical cores each, running at 2.20 GHz. With hyper-threading enabled, there are 40 logical cores per node. Each node has 256GB DDR4 memory.

7.2 Throughput measurement

The server side is a Skinnycell that consists of 3 or 5 nodes running the server program. The (C++) test program spawns 30 client libraries, each opens a certain amount of files and keeps issuing read/write requests to those files for 30 secs. We ran 5 such test program, so there are 150 clients sending requests simultaneously. We used the barrier implemented on Skinnyto synchronize the clients. We change the requests' read/write ratio, the number of server nodes, the number of files and measure the throughput (operations per second).

From Figure 3, we observe that when changing the read/write ratio (x-axis), the read throughput drops while the write throughput remains around 200 ops/sec. This indicates that our system may be bounded by the write requests handling. Our implementation forces every opened file to be cached, so for each write requests, the server has to invalidate all 150 clients' cache files. We

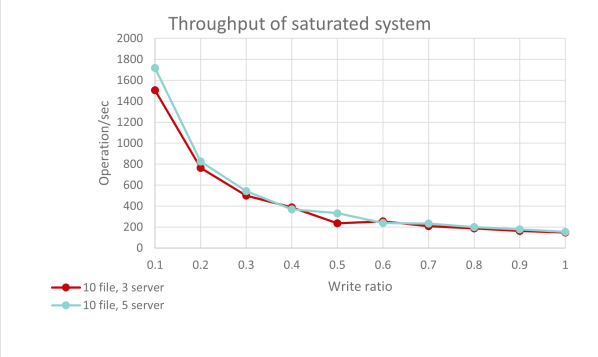

Figure 4: Throughput, varying read/write ratio. 150 clients access 10 files on a Skinny cluster of 3/5 nodes.

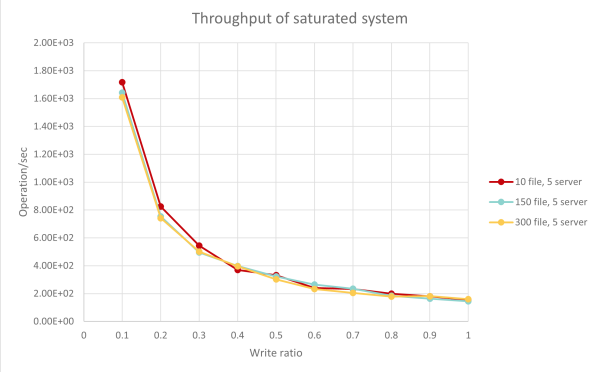

Figure 5: Throughput, varying read/write ratio. 150 clients access 10/150/300 files on a Skinny cluster of 5 nodes.

guess the network, flooded with these cache invalidation requests, might be the bottleneck.

We thought that the number of server nodes may affect the throughput, as state replication needs to go to more servers. However, Figure 4 shows that the difference is negligible, perhaps because all server nodes are in the same data center (probably even the same rack), so the network round trip is not a significant factor. Another thing we thought that might affect the throughput is the number of files we opened and read/written, as those are the files . However, Figure 5 shows again that there is no significant difference when we change the number of files from 10 to 150 and to 300. We think regardless of those variations, our system is bottlenecked by the cache invalidation traffics induced by write requests, which implies that the cache coherence protocol needs more tweaks.

7.3 Latency

We tested latency for reading and writing a single file, running the test for ten times and take the average. We ensure this is the first read, so cache is not participated. The average read latency (that goes to server) is 368 microseconds, and the average write latency is 1517 microseconds.

8 Lessons learned

8.1 Clearly defined where to put logic

In hindsight, we should've clearly defined weather each piece of logic belong to the state machine or the RPC handler. We started with writing logic in RPC handling functions and eventually moved most of them into the state machine functions. That makes lots of code that can be run concurrently be executed in serial. We should put as little logic inside the state machine as possible so that most of the execution can be parallelized. What is worse, since the state machine runs serially, lots of concurrent tools moved from the code in RPC handler, such as locks and conditional variables, caused the program to hang at weird time point and took us quite some time to figure out the deadlock. Should we have a clear cut between what can before writing the code, we could have run more portions of the logic in parallel and avoid lots of bugs caused by wrongly use concurrent tools inside state machine.

8.2 Use synchronized container

When a data structure is shared by multiple threads, we currently uses the classic mutex approach, where the developers declare a mutex and are expected to acquire the lock before modifying the data. However, during testing we found many bugs that can be attributed to forgetting to acquire a lock, acquiring a wrong lock or releasing a lock too early. Therefore, we believed that using a library that provide synchronized containers, e.g. *Folly* from *Facebook*, can reduce many bugs that we encountered.

References

[1] BURROWS, M. The chubby lock service for loosely-coupled distributed systems. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2006).

[2] DUPLYAKIN, D., RICCI, R., MARICQ, A., WONG, G., DUERIG, J., EIDE, E., STOLLER, L., HIBLER, M., JOHNSON, D., WEBB, K., AKELLA, A., WANG, K., RICART, G., LANDWEBER, L., ELLIOTT, C., ZINK, M., CECCHET, E., KAR, S., AND MISHRA, P. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (July 2019), pp. 1–14.

[3] EBAY. Nuraft c++ implementation of raft core logic as a replication library, n.d.