Zircon

A Simple, Composable Distributed Filesystem

Cel Skeggs (cela@mit.edu) | Robert Rusch (rusch@mit.edu)

2018-05-11, 6.824 Final Project, https://github.com/celskeggs/zircon tag v0.1.0

Intro

Zircon is a simple, composable distributed file system inspired by distributed filesystems such as AFS, GFS and Ceph. It is built out of an interacting set of services, including an etcd server and many custom components, organized into a block storage layer and a file system layer.

Our primary goal was to build a sufficiently simple, extensible, and composable system that can be the basis for the authors' further experiments with distributed file systems, find current systems lacking in these regards. For our specific implementation, we also focused on scalability and fault tolerance and attempted to achieve availability and performance.

We achieved our primary goals, but our system is sufficiently incomplete to satisfactorily achieve fault tolerance, scalability, availability, or performance. This appears to be primarily a result of an insufficiently mature implementation, rather than any fundamental errors in our design.

Design

High-Level Overview

Our system contains a number of different, interlocking modules, as shown in Figure 1. Six of the modules (Etcd, Chunkservers, Frontends, Metadata Caches, Sync Servers, and Management Services) are executed as independent services, available over an RPC interface or providing no interface. In addition, four of the modules (the Memory Storage Backend, the File Storage Backend, the Block Storage Library, the Filesystem Library) are provided as libraries used by other modules. The only user-facing application currently implemented is a FUSE filesystem, which allows arbitrary Linux applications to use Zircon as a POSIX-like filesystem without any changes.

Block Storage API

The API exported by Zircon's block storage layer to its clients (filesystem or otherwise) is fairly straightforward -- it supports creation, reading, updating, and deletion of "chunks", which are 8 MB blocks of data. Chunk numbers are assigned on creation. Each read and write is performed on a *Version*, which is an iteration of a particular chunk. Reads will read any later version, if available, but Writes and Deletes will fail if attempted on old versions -- this allows for optimistic concurrency between different clients. Versions increment monotonically, and only the most recently written version is available for reads.

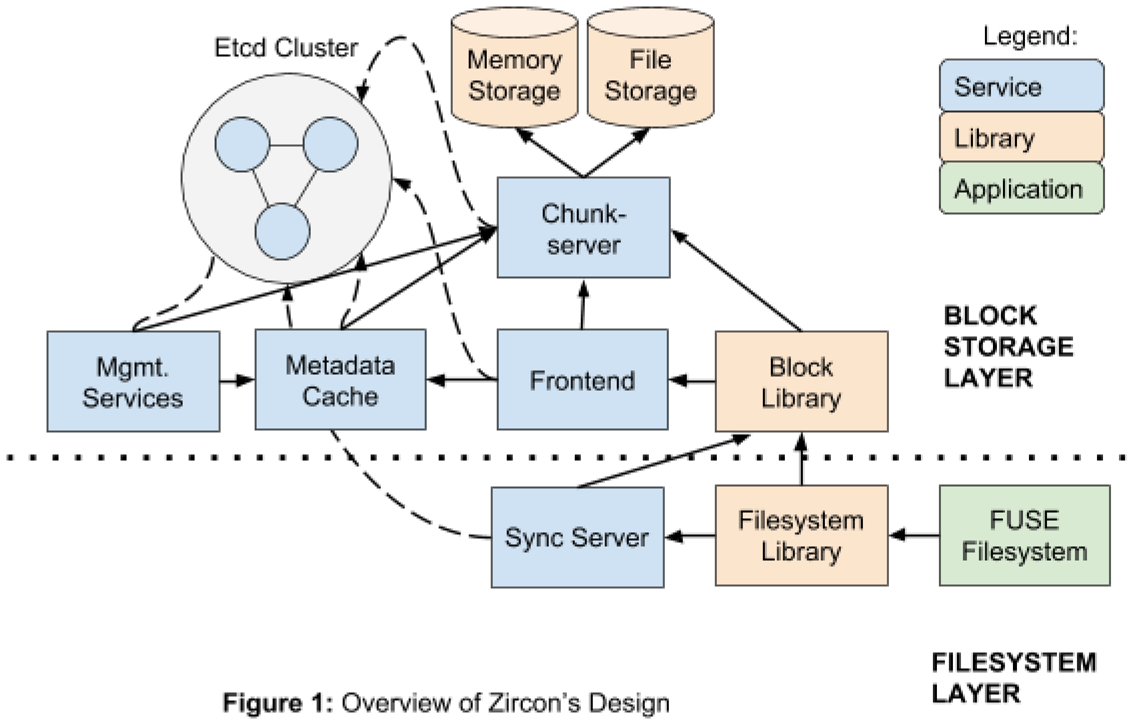

Figure 1: Overview of Zircon's Design

Etcd

We use etcd, a distributed key/value store built on Raft, as the coordination kernel for our system. We originally considered using Zookeeper, but etcd appears to be a reasonable modern alternative, providing tools like watches and transactions that allow for the construction of more complicated concurrency primitives. Etcd, as a Raft-based system, follows a majority-quorum model to ensure consistency.

Chunkservers

As the unit of storage for Zircon, we have chunkservers, which are very simple servers that just store data and a small amount of metadata. Although they connect to etcd to register their existence, they have no other active roles in the cluster: they simply respond to requests as they are received.

Chunkservers support a small set of fundamental operations: Add, Delete, List, Read, Start Write, Commit Write, and Update Version. To conserve bandwidth from a remotely located client to a datacenter, chunkservers support receiving a Start Write request (which contains written data, unlike a Commit Write request) and replicating the write to other specified chunkservers.

Rather than store one copy of each chunk on a chunkserver, we store a matrix of chunk-version data. One version for each chunk is marked as active and is the only version served to clients.

Writes to a chunkserver look like this:

- The client uploads data to be written at a certain offset as a Start Write request, one per chunkserver.
- The frontend reserves a version number to write new data to.
- The frontend performs a Commit Write on each chunkserver. The new version is not served yet.
- If all commits succeed, the version is updated, and all chunkservers are notified and start serving it.

Metadata

We planned to store all chunk metadata directly in etcd originally, but it turned out that etcd's upper limit of 8 GB of stored data could limit our system to somewhere on the order of 64 TB of user-visible storage. This limit could be a practical concern, so we decided to include an additional layer of indirection: we store our metadata in the storage system itself, and put the much smaller *metametadata* in etcd. Each metadata chunk contains two parts: an allocation bitset and an array of metadata entries. Each metadata/ metametadata entry contains a version and replica list, among others information.

Management Services

A number of different management services help keep Zircon in a healthy and functional state:

- A replicator to re-replicate chunks without a sufficient number of valid replicas.
- A chunk balancer to rebalance storage usage across our chunkservers.
- A garbage collector to delete and free up space taken by unreferenced chunks.
- A recovery service to help repair failures caused by other server crashes.

This functionality is not critical; an unmanaged cluster can work fine in the absence of failures. This allows us to independently implement, evaluate, and replace these cluster maintenance functions.

Frontend

We separate out the code that handles coordination of writes into the frontend server from the block storage client library, so that the complicated back-and-forth communication can be kept within the cluster network, and avoid needing to travel over a (likely high-latency) link to the client system.

File System

Our filesystem has two layers: a synchronization server and a filesystem interface library. The sync server provides read/write locks over etcd, and the filesystem interface library adds the block storage API.

This system currently stores one file per chunk, but this is intended to be changed so that a chunk can include multiple files, and a file can include multiple chunks, to support a variety of file sizes. A file is stored as a length plus the contents of the file, and a directory is just stored as an array of directory entries.

The read/write lock system synchronizes accesses similarly to POSIX semantics. We use read locks on different tree nodes as we traverse to each file (and read/write those files), and only use write locks for updating the filesystem structure, not for data changes.

Implementation

Our implementation focused on modularity as a mechanism to achieve our primary goals: each service is abstracted away behind a Go interface that can be swapped out for other implementations, helping us achieve composability and extensibility and to be able to effectively build automated tests for our code.

Future Goals

For real world usage, the stability, correctness, and fault tolerance of file systems are critical. Many more tests would need to be written and applied in order for this system to be trusted with any amount of important data, even more so given its distributed nature.

Zircon's current performance could also be improved significantly. Benchmarking currently indicates a number of potential performance improvements, such as using more granular locking in certain parts of the metadata cache leasing system.

Additionally, garbage collection and recovery services still need to be implemented, and we haven't addressed fault tolerance in terms of server crashes during active operations to the extent that we need to.