Operating Systems Design Project 2

Kushal Joseph Vallamkatt (kjv5316@psu.edu)

Introduction

In this short report, I discuss how I implemented the distributed striped file server using gRPC and protocol buffers, step by step.

Basic Striped Writes, Reads

- As a first step to the implementation of this project, I worked on writes and reads to the system, and striping them across the servers. Note: At this point of time, there is no token system. It is implemented later.
- The striped files stored on disk, on the Fileservers have names in a pattern `"ServerNumber_filename_ChunkNumber"`. (Here "filename" refers to the filename, without any extensions)
- I have delegated most of the logic work of the file system to the Metaserver. The Fileservers just **receive instruction**s from the client (via the metaserver) on which files to write/read and from what offsets within the files.
- For a given read/write operation, the client first sends a request to the metaserver. The metaserver sends back a list of instructions in the format {chunk-number, server-number, start-byte, end-byte}.
- Each "instruction" represents one Chunk, which in my code means one "stripe" of a file. A chunk is of the size `PFS_BLOCK_SIZE * STRIPE_BLOCKS`.
- For writes, this means *"Write a chunk (i.e, a stripe) #chunk-number to the fileserver #server-number, for the PFS file, starting from start-byte, and ending at end-byte (both inclusive)"*.
- For reads, it means exactly the same as above, but to read.
- Internally, I have logic in the fileserver which, based on the chunk number, figures out the offset to read/write within the striped files, and in the case of writes, the offset in the given buffer.
- The client hence knows, for each instruction, which Fileserver to forward it to. It sends this instruction to the respective file servers.

Token System

- After the basic reads, writes (and other operations like delete, fstat) worked fine, I began implementing the Token System.
- A client keeps with it a list of tokens granted by the Metaserver. These tokens are specific to a particular filename. Each token contains data showing the range of bytes (start-byte - end-byte), along with the mode allowed.
- When a client needs to read/write, it first checks if the range requested can be "covered" using one or more tokens, for the given mode.
- If it is **unable to cover the range**, the client will request for a Read/Write token for the entire range of bytes it is interested in, for the specific filename.
- The metaserver checks if this token collides with any other tokens handed out to other clients. Of course, read tokens will not "collide" with other read tokens.
- If there is a collision, the Metaserver sends back a **revocation message** to the current holder of the token (using **gRPC Bidirectional Streaming**).

- This message revokes the current token held by the other client, and if possible, splits the existing range, to maximize concurrency.
- For more context, the Metaserver actually sends back a revocation token, i.e, a token that the client should delete, and if possible, a set of new split tokens, which are "granted" to the client.
- During this process, the client who requested **waits on a condition variable**. Then, the server grants the requested token to the client. The client resumes once it receives this Grant, in the stream.

Cache

- I have used a byte-granular cache, for the reasoning that all my reads, writes are at byte granularity, and so is my token system.
- The cache stores a mapping of ranges (start byte - end byte) which maps to some given read data.
- Whenever the client requests to read some bytes, we first check if the cache's data can cover the given range. If not, we forward the request to the metaserver over the network
- Every read returned from the File System is cached in the structure mentioned above.
- Invalidations: In my implementation, the tokens a client holds are always consistent with the cache ranges. Whenever a client writes to a file, any clients with colliding ranges are sent revocations (via the gRPC bidirectional stream). I use these revocations to invalidate the particular range of the cache.
- I have not implemented write caching. The client will only cache reads. And pfs_read will return cached data, as long as it's still valid. This also means I have not implemented the relevant execstat variable counting for writes/writebacks.