CS 739 Mini-Project 2 Report

Abhinav Agarwal Adil Ahmed Cong Ding Song Bian

9th March 2022

Contents

1 Overview

We implement a simplified AFS-like distributed file system, the Not-so-simple Multi-mode Reliable Distributed File System (NssMRDFS) which supports basic POSIX functionality. We design and implement a protocol for crash recovery and file consistency after crashes. We also test the functionality and measure the performance of our file system. We implement optimizations to tweak performance but mainly to improve reliability and present stronger consistency guarantees to users that require them. We run custom workloads to benchmark our system and present our takeaways and observations in this report.

2 Design

2.1 Key Principles

Our key design principles are as follows.

1. Clients do whole-file caching on `open()`. i.e. the entire file is streamed from the server. On subsequent opens, the timestamps of the client and server files are compared. If the file has not been changed, it is not re-downloaded. This is illustrated in Figure 1

2. Reads and writes are served from the local cache copy.

3. Clients sync the file to the server (using streaming) on `fsync()` or on `close()` if the file is dirty, as shown in Figure 2.

4. The last client to close the file overwrites the server copy.
5. File timestamps are compared to maintain consistency - no persistent state is required.

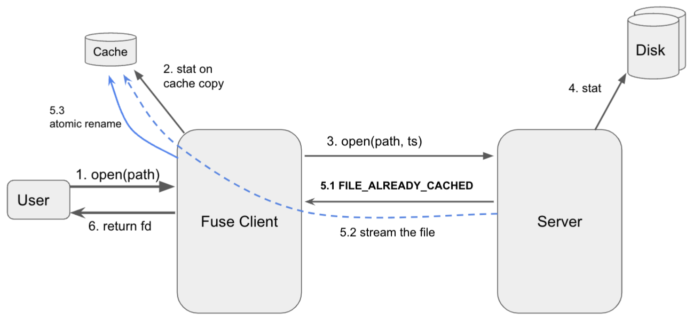

Figure 1: open() operation

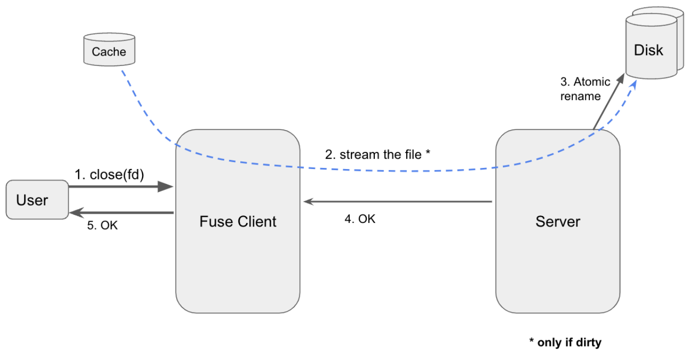

Figure 2: close() operation

The complete API interface is listed in Figure 3.

POSIX call	Fuse Intercept	Description
creat	create	Create a new file and return its fd
open	open / create	Open (or create) a file and return its fd.
stat	getaatr	Return info of the file.
read/pread	read*	Read n bytes into a buffer from file fd.
write/pwrite	write*	Write n bytes from a buffer to file fd.
unlink	create	Delete file.
fsync	create	Flush data to disk.
close	create	Close the file fd
mkdir	create	Create a new directory.
rmdir	create	Removes a directory
readdir	readdir	Returns a directory entry

Figure 3: API Interface (* denotes calls performed on local cache

2.2 Reliability and Crash Recovery

Our file system ensures reliability for both client and server:

1. Server crash recovery. Since server is stateless, we only need to clear temporary files.
2. Client crash recovery. There's no persistent state on clients, so crashed clients only need to delete its cache on recovery.

2.2.1 Multi-Mode Operation

We extend our file system to provide multi-mode operations (inspired from SplitFS [2]) which caters to user needs. We provide 2 configurable reliability modes that has different sets of consistency guarantees:

1. **POSIX mode.** The regular mode for the file system.

2. **SYNC mode.** Do a `fsync()` to server on each write call.

The difference in write calls is illustrated in Figure 4.

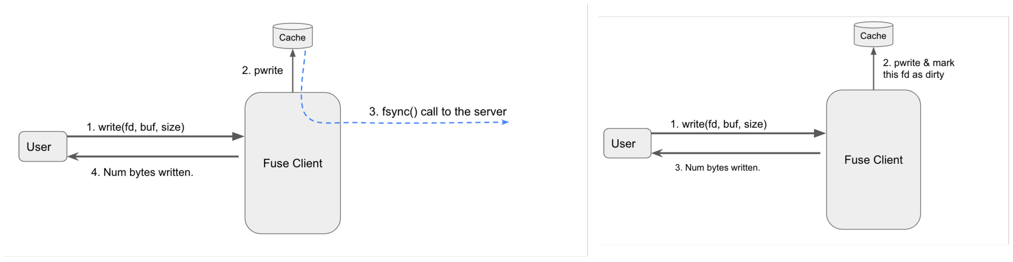

Figure 4: Left: write() in SYNC mode; Right: write() in POSIX mode

The users can choose which mode to use based on their needs. Different clients on the same machine can use different modes. We also plan to add this reliability mode per directory in the future.

3 Implementation

We developed our file system on top of the FUSE library and chose the gRPC library to set up the RPC between clients and server. We use C++ as our primary programming language and write 1700+ lines of code in total.

3.1 Client Caching

We use a directory in the client's `/tmp` folder to store cache entries. We use a flat folder structure where each filename is the hash of the file's complete AFS path. During download, the file is given a temporary name. An atomic rename to the final value is executed after the download is complete. The cache is cleared on every reboot. Partial writes may be lost in this manner unless the client is operated in SYNC mode.

3.2 Flushing files to server

We use a static directory on the ext4 filesystem to store server files and directories. When the client is streamed to the server during the `fysnc` or `close` operations, it is given a temporary filename. An atomic rename is executed after the transfer is complete.

3.3 Other Things We Tried

We tried to implement a smart and scalable prefetching mechanism that works as follows.

1. The client maintains a background thread which will monitor recently stat directories and resource pressure.

2. If the client is relatively idle, it will send a pre-fetch request to the server.

3. If the server is relatively busy, it will send a busy reject. Then the client does exponential backoff.

4. Otherwise, server responds with stat of all files (limited) in the directory.

5. Client responds with list of stale files and server sends a bulk download to client.

6. This bulk download can be cancelled in between if either server or client gets busy.

4 Evaluation

We run several benchmarks and conduct measurements to evaluate our final implementation.

4.1 Experimental Setup

Our experiments are conducted on 2 nodes on CloudLab [1], which is a large cloud infrastructure built mainly for academic research. Each node runs Ubuntu 18.04 LTS and uses Intel Xeon Silver 4114 2.20 GHz CPUs with 40 cores. Each node has 190GB of memory.

The sample workload used in our experiments consists of the below operations:

- create: create 500 empty files and write 8KB of data to each.
- read: read 500 files of size 8KB each.
- mkdir: create around 6.5k nested directories. The maximum depth of the nested directories is 6.
- scandir: scan all the directories.
- rmdir: recursively remove all directories.

4.2 Experimental Results

We measure and evaluate how various implementations perform on several key metrics. We note the performance impact in each case and reason about the differences observed.

4.3 Access Time

We set up two nodes on Cloudlab and run the client and server on different nodes. Then we measure the time taken for the first access (for different sizes) and compare it with subsequent accesses.

As Figure 5 shows, the latency for initial access increases with the increase in file size. When the client first opens the file, the file is streamed from the server to the client. For subsequent opens, the file is cached locally. No RPC is included in this process, thus varying the file size doesn't impact the access time.

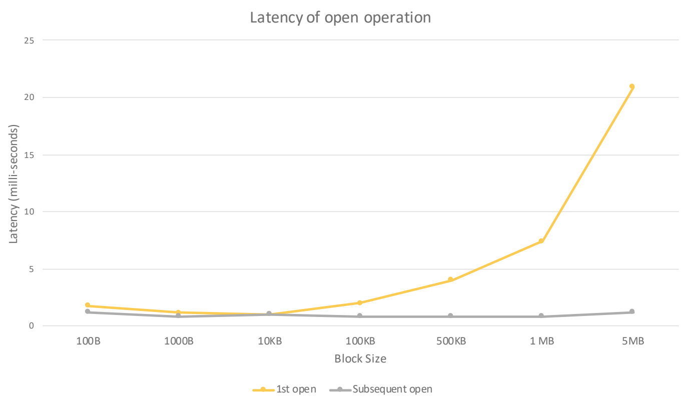

Figure 5: Access Time

4.4 Read/Write throughput

We evaluate the performance of write and read operations by varying the block size. From Figure 6, we can observe that the throughput of `write_ts POSIX` and `read_ts POSIX` increases with increasing block size. This is as with increase in block size, the number of write operations will decrease due to the larger buffer size and result in better throughput. However, the throughput of `write_ts SYNC` will decrease after a block size of 100 KB as the amount of data flushed to the server on each write grows proportionally.

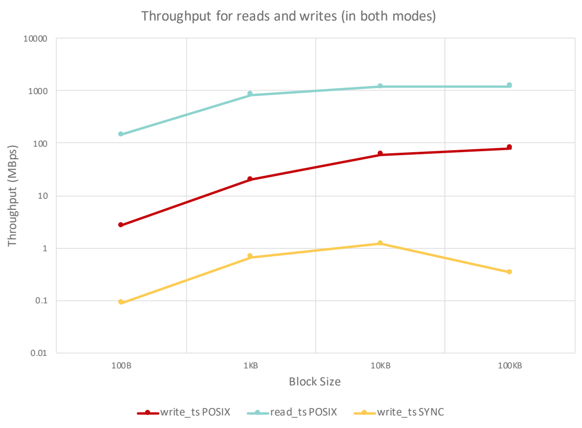

Figure 6: Read and Write Throughput

4.5 Scalability

We run the server on one node and set up multiple clients on a separate node. We then run the same workload simultaneously on multiple clients and evaluate the performance. We vary the number of clients each time and compare the performance for each type of operation. The results are presented in Figure 7.

The elapsed time is approximately the same for one client and two clients. There is a steady performance degrade as more clients are added as the frequent RPC calls saturate the network link to the server, while still achieving respectable performance. In general, our file system scales steadily for several clients with room for further optimizations.

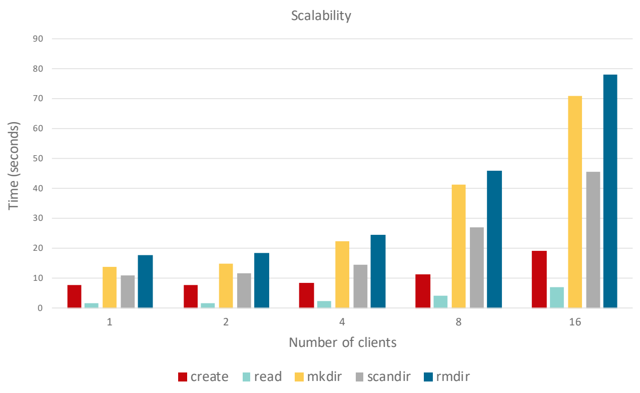

Figure 7: Scalability

4.6 Comparison with CSL

In this experiment, we compare the performance of our implementation with CSL. From Figure 8, we can observe that the latency of our system is much larger than that of CSL, indicating a lot of room for improvement.

5 Conclusion

We develop an AFS-like distributed file system from scratch and verify the functionality and reliability of our protocol. We extend the file system to support multi-mode operations and measure the performance of the file system.

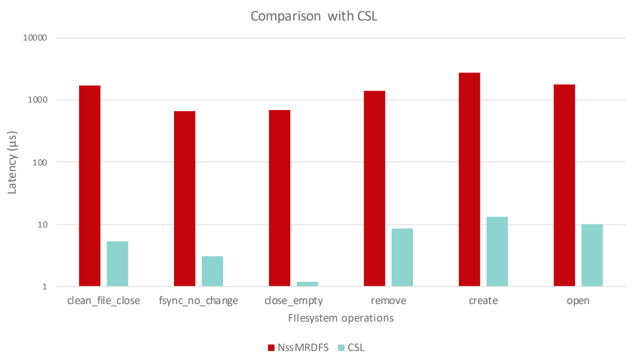

Figure 8: Comparison With CSL

References

[1] Cloudlab. `https://www.cloudlab.us/`, 2018.

[2] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 494–508, 2019.