WiscAFS - AFS like Client/Server file system

Project group 7:
Deep Jiten Machchar - machchhar@wisc.edu
Vishnu Ramadas - vramadas@wisc.edu
Varun Kaundinya - kaundinya@wisc.edu

1 Introduction and Design

We have built an AFS-like distributed file system on top of FUSE, specifically unreliablesfs [1]. We use the gRPC library to communicate between a server and clients [2]. During FUSE mount, a client object gets initialized and establishes connection with the server.
Our filesystem implementation caches entire files locally and enforces a close-to-open consistency model. We map the server files to a local cache by maintain a file named using the server file inode numbers. We write back the updated files to server only after a close, provided they were modified by a write. We track writes by maintaining a dirty bit. Our implementation follows last writer win policy as part of the close-to-open consistency.

2 Results

During the demonstration, we showed filebench results obtained with debug prints enabled. We regenerated all filebench results after removing them and have summarized those results below. Since we were close to finishing the last parts of the consistency and durability tasks, we completed it after the demo and have summarized our findings below.

2.1 Filebench

The performance of the filebench benchmarks when run for the complete fileset are shown in table below.

Table 1: Performance measurements for Filebench Workloads

Benchmark	Operations	Operations/sec	Rd/Wr	Bandwidth (mb/sec)	Time per Operation (ms/op)
filemicro_create	948	94.788	0/95	94.7	10.528
filemicro_createfiles	51	5.099	0/2	0.0	191.800
filemicro_createrand	1001	100.087	0/91	45.3	9.970
filemicro_delete	54	6.749	0/0	0.0	1878.598
filemicro_rread	0	0	0/0	0.0	0.0
filemicro_rwritedsync	0	0	0/0	0.0	0.0
filemicro_seqread	0	0	0/0	0.0	0.0
filemicro_seqwrite	938	93.788	0/94	93.7	10.528
filemicro_statfile	65943	6593.626	0/0	0.0	3.017
filemicro_writefsync	42992	4298.636	0/4295	33.6	0.231
fileserver	188	18.798	1/6	0.6	1593.980
mongo	260	25.997	4/4	0.1	38.375
varmail	2358	39.296	6/6	0.1	399.638
webserver	434	43.372	15/0	0.3	1029.254

2.2 Consistency: Who is the winner?

We invoked Client B writes from Client A through a script that SSHes into Client B. Each script writes the same amount of data to the same location on the server. The client that closes the file last will send it across the last. The server will therefore retain the data written by the client that closed last. We noticed that this setup results in Client B winning always, i.e. data written by Client B was always written to server last. We believe this is due to the delay in SSHing into Client B, which guarantees that B starts slighly after A does. We gradually added delays to Client A's close. We swept the delay from 780 micro-seconds to 810 micro-seconds and saw the probability of winner shift from B always winning to A always winning (Figure: 1).

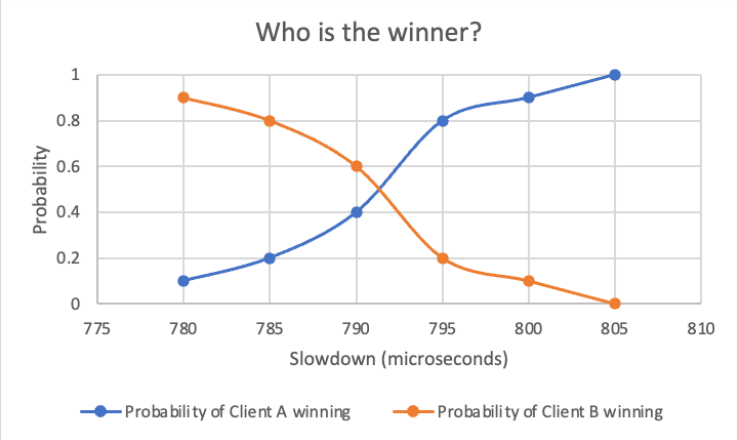

Figure 1: Sweep of delay on one client vs. winner

We also measured the RPC round trip latency, i.e. the time difference between the client-side RPC call to close a file and the client receiving a response from the server after the close happens. We did this for various filesizes and our findings are captured below

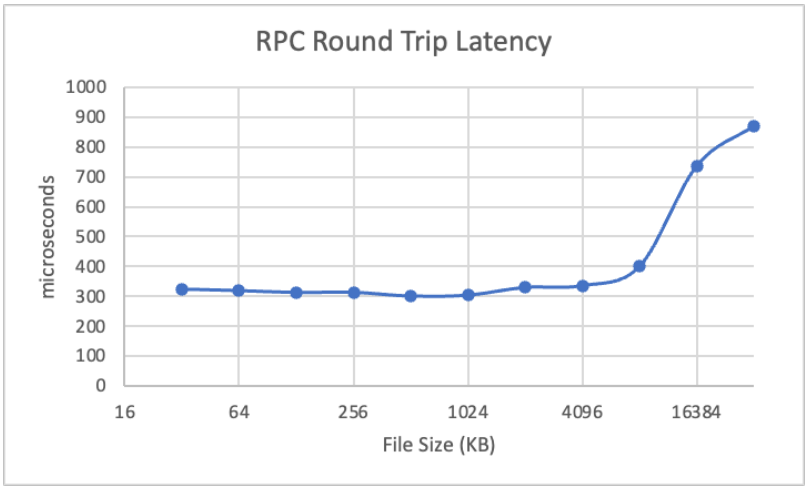

Figure 2: Write latency

2.3 Durability

During the project demonstration, we had a basic version of *alice_delay* where we were maintaining a queue to delay a write till fsync happens. We later added error injection capabilities for both *alice_delay* and *alice_reorder*. Our durability infrastructure is as follows -

- We maintain a queue for all writes and immediately return to the application with a success code
- All reads look through the queue for a match and then read data from the match.
- During an fsync, we write data before flushing it out to server. We also delete the entries from the queue during fsync. If the error inject is of type reorder, we randomly interleave writes to the disk.
- Test cases
  - With reorder error, we generated a test case where a file is first written and then read before closing. These reads will always get right data since the queue returns the required data. But after closing, the final data would be corrupted in the file as the writes get reordered. A new read after open would see this corrupted data.
  - A crash before an fsync will make the file system lose data in the queue. So after restart, the application would see inconsistent data.

  We have included a demonstration of these in the link below.
- Demo - durability_reorder_corrupt.mkv

3 Learnings

We didn't have much low-level systems experience before this course. The contents of the project were especially new to us. This was a great learning experience as we got to work on a lot of new things such as GRPC, FUSE, and file system calls. In addition to learning about distributed file systems, we also picked up good coding practices and complex debugging insights. Some of the key learnings are,

- Got exposure to AFS-like file system. Coming up with our own protocol and consistency semantics involved deep reasoning about various aspects of such systems
- Took quite some time to set up a cross-compiled unreliablefs build, which blocked us from working in parallel during the initial days. This is something we could plan better in the future.
- Few pointer casting mistakes like not casting char* to std::string caused data corruption which took considerable time to debug. We realised we need to read-up on C/C++ compatibility and support.
- Learnt a few good coding practices like controlling prints through debug macros
- We didn't do code reviews when we implemented major changes. In hindsight we believe a group code review would have helped solve several issues before hand.

4 Improvements

Our system is a bare-bones one and some of the improvements that we think would make it more robust are -

- Do a getattr (lstat) call on open and see if the modified time matches, and only then do a full file read. Currently, to keep it simple, we do a full file read regardless of the modified time. This change should help speed up read-only accesses to a great extent.
- We think our system performance can be improved further by packing data more efficiently instead of sending whole objects during read/write streaming. This way, the 4KB size constraint of the gRPC can be utilized to the best extent.
- Our crash logic is naive. We ensure data integrity on a server crash by doing an atomic rename of a temp file. We do not support a reply cache which can handle crashes for non-idempotent operations. On client crashes, we simple reset the cache and expect the client to re-fetch data. Logging operations will allow us to continue using the cache even after a client restart. This should also give significant performance gains over our current implementation.

References

[1] https://github.com/ligurio/unreliablefs/tree/master/unreliablefs

[2] https://grpc.io/docs/languages/cpp/