wiscAFS

Evan Wireman, Mahitha Pillodi, Ruthvika Reddy Loka

1 Design Overview

This section describes our intended design, however there are some bugs in our implementation. Much of our design is simple, we tried to only touch the pieces we needed to. This mainly pertained to the open, close, and getattr operations. We used a hash of a file's *struct stat* to act as an indicator of any file changes on the server. For the rest of this paper, this hash will be referred to as an s-hash.

1.1 Open

When a file is first created, the client sends a server a *mknod* RPC call, at which point the server initializes the file. When the client proceeds to open the file, it first checks if the file exists locally. If it does, the client verifies that the s-hash it has stored for the file matches the server's s-hash. If this is true, the client opens its local version of the file. Otherwise, it talks to the server and pulls the server's version of the file, along with its current s-hash. The client then caches this file and adds the hash to a map. Any subsequent reads, writes, or flushes are done to this local version of the file.

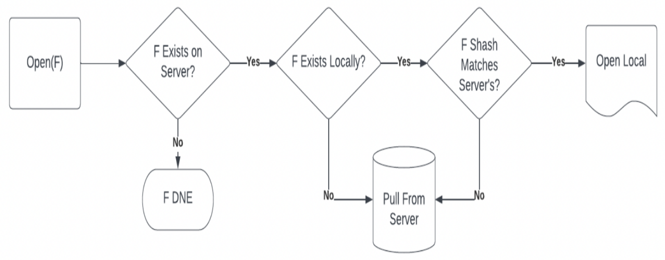


Figure 1: Open

1.2 Close

Whenever a client modifies a file with path */path/to/file*, a .diff file is spawned at the path */path/to/file.diff*. This serves as an indicator that the client has made local changes that need to be communicated to the server. Thus, when the client closes the file, it checks if there is a .diff file. If so, it knows it needs to read in the whole file and transmit it. Otherwise, it merely closes the local version of the file.

If the server receives a close (in our implementation, we called the actual RPC call write) request, it writes over whatever is in its local version of the file. It also updates its own s-hash map, storing a new s-hash for the file now that it has been modified.

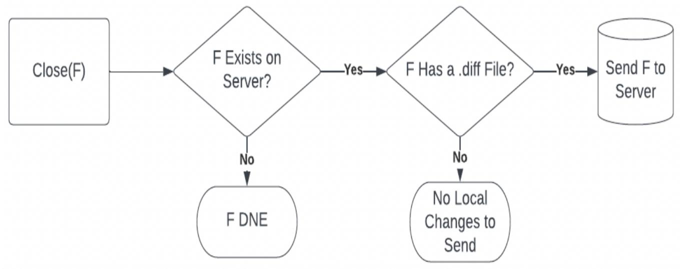


Figure 2: Close

1.3 Getattr

The client first checks if it has the file cached. If so, it pulls the s-hash for the file in question from the server and compares it to its own stored s-hash. If the two s-hashes match, the client knows its local version is either up to date or newer than the server's version, so it returns local information. Otherwise, it pulls the *stat struct* from the server.

1.4 Other Ops

mkdir, readdir, and rmdir all directly communicate with the server to maintain a consistent directory tree. unlink was treated as a delete, since we were not concerned with links, and triggers both a client and server delete. rename also directly communicates changes with the server to maintain consistency with file names.

1.5 Shash Semantics

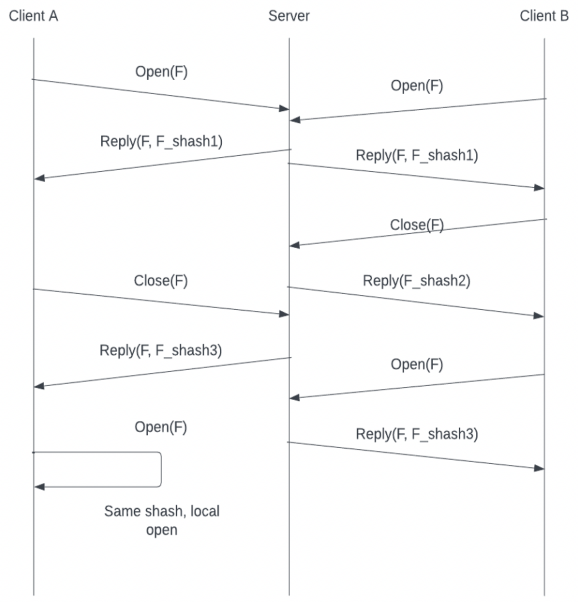


Figure 3: Shash Semantics

2 Results

While most of the filebench workloads ran, we realized our *create* and *delete* implementation dramatically slowed down some of the tests. For that reason, we adapted some of them to generate less files. Additionally, for whatever reason, our implementation only seems to work in debug mode. This may be an issue with our machine's configuration, we could not identify why this was the case. Unfortunately, the extra I/O to *stdout* likely played a role in slowing down our implementation.

Workload	ms/op
filemicro_create.f	21.702
filemicro_createfiles.f	30.213
filemicro_createrand.f	24.769
filemicro_delete.f	345.662
filemicro_rread.f	0.125
filemicro_rwritedsync.f	0.634
filemicro_seqread.f	0.146
filemicro_seqwrite.f	41.311
filemicro_statfile.f	160.623
filemicro_writefsync.f	25.965
fileserver.f	FAILED
mongo.f	24.231
varmail.f	138.642
webserver.f	244.642

3 Implications

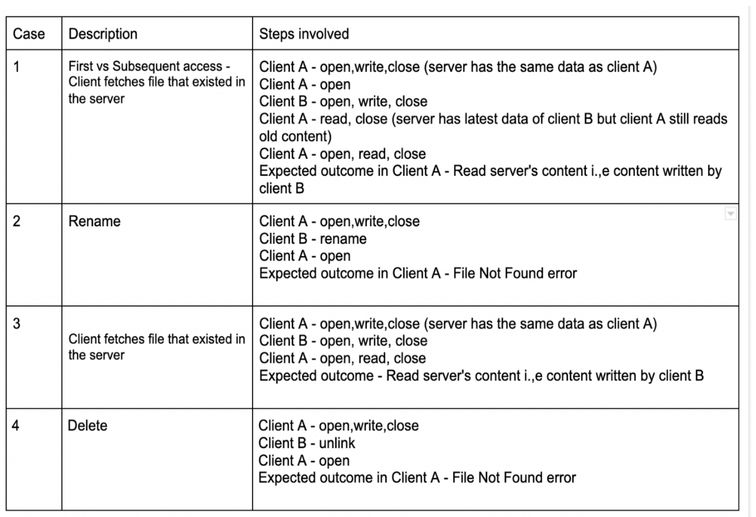
From our results, we feel that there is much room for improvement. Our implementation shines, as one would assume it would, when it works on a single, local file. However, once there is frequent creations/deletions of files, our implementation starts to falter

One way we could improve our implementation is by not alerting the server at file creation. The FUSE *create* method calls *mknod* and *open*. We send a *mknod* request to the server at each file creation. However, we would save a few RPC calls if we were to keep a local copy and just send it to the server at the first time it is closed. This would also save us from having a "ghost file" on the server if the client were to crash before closing an open file.

Another improvement we could make is fully utilizing the RPC payload size. We stuck to 2mb RPC payloads because we found that the maximum size was 4mb. However, this leaves us with nearly 2mb of essentially wasted space in each packet. Our original though process was to remain relatively conservative, however now we believe that this is impacting our performance.

4 Consistency Tests

We designed four test cases to verify our project's functionality. The below figure summarizes our tests.



Case	Description	Steps involved
1	First vs Subsequent access - Client fetches file that existed in the server	Client A - open,write,close (server has the same data as client A) Client A - open Client B - open, write, close Client A - read, close (server has latest data of client B but client A still reads old content) Client A - open, read, close Expected outcome in Client A - Read server's content i.,e content written by client B
2	Rename	Client A - open,write,close Client B - rename Client A - open Expected outcome in Client A - File Not Found error
3	Client fetches file that existed in the server	Client A - open,write,close (server has the same data as client A) Client B - open, write, close Client A - open, read, close Expected outcome - Read server's content i.,e content written by client B
4	Delete	Client A - open,write,close Client B - unlink Client A - open Expected outcome in Client A - File Not Found error

Figure 4: Scenarios

5 Conclusion

This project was fun and challenging. Although I wish we were able to complete a more complete implementation, we are proud of what we have. For normal command line file ops, such as *ls* and *touch*, it works just fine. That being said, there is huge room for improvement. This project helped us to greater appreciate some of the insights discovered in some of the papers we have read this semester.