P2 Client/Server File System

Akshay Parashar

Hemalkumar Patel

Kalpit Munot

Reetuparna Mukherjee

INTRODUCTION

Andrew File System(AFS) provides richer features than Network File System(NFS). It offers local cache copy to work on in the event of server outages, open-close consistency, hooks to receive updates from server. In this project, we have implemented AFS using FUSE. We implemented a subset of AFS features such as local cache copy and open-close consistency. Our design can tolerate file system crashes on both server and client sides, and can recover persisted data in a timely fashion. In addition to that, our design consists of a garbage collector that cleans intermediate files generated by our system.

IMPLEMENTATION AND PROTOCOL

POSIX APIs

We implemented the following POSIX APIs. We categorized them into local-only and local-remote. Local only calls don't require communication with the server. Local-remote calls may need to contact the server, either always or in-some cases, for successful operation.

- **Local-only:** read(), write(), utimens(), flush().
- **Local and remote calls:** getattr(), readdir(), mkdir(), rmdir(), creat(), unlink(), open(), release()

Terminology and Protocol Assumptions

There are terms and assumptions that we have used throughout this project. We believe that understanding them would bolster the reader's understanding of this report.

Terminology:

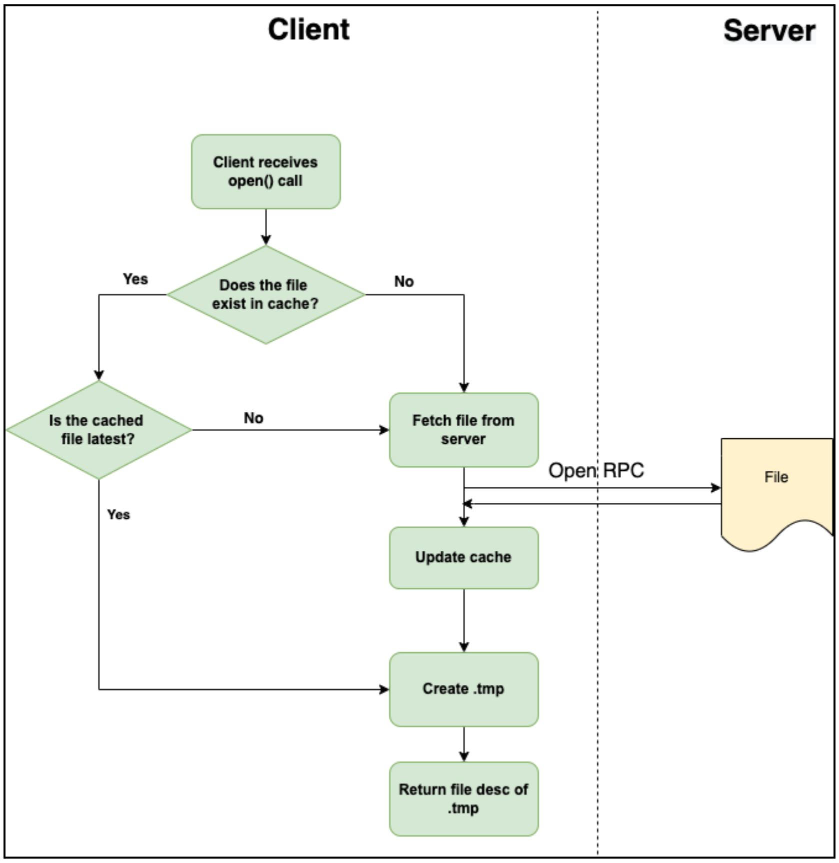
- **.tmp file** - Dirty file that have not been closed by the application process
- **.consistent file** - file that is closed by application process and is safe to be flushed on client/server

Protocol Assumptions:

1. Unless close() API is called, data is non-recoverable (dirty).

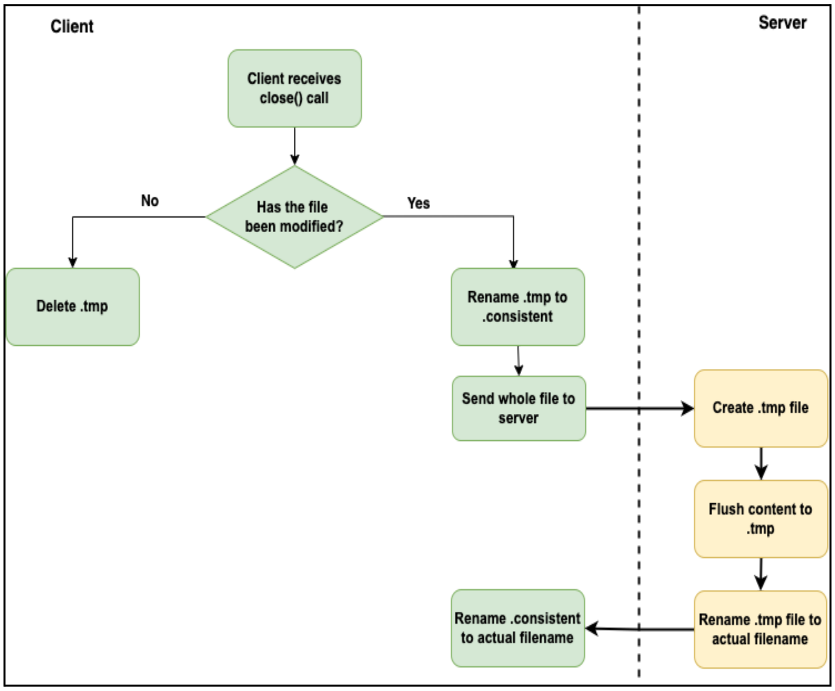
2. After close() is called and if server or client crashes, the data needs to be recovered by crash consistency protocol and updated on both the server and client
3. "Last Writer Wins" on both and Server and Client.

Open



- Client fetches the file on demand from the server **iff** file does not exist on cache (or) cached file is stale
- On fetching the file from server, client updates its cache
- Each application process is given its own .tmp file (i.e. copy of the cached file) descriptor to work on

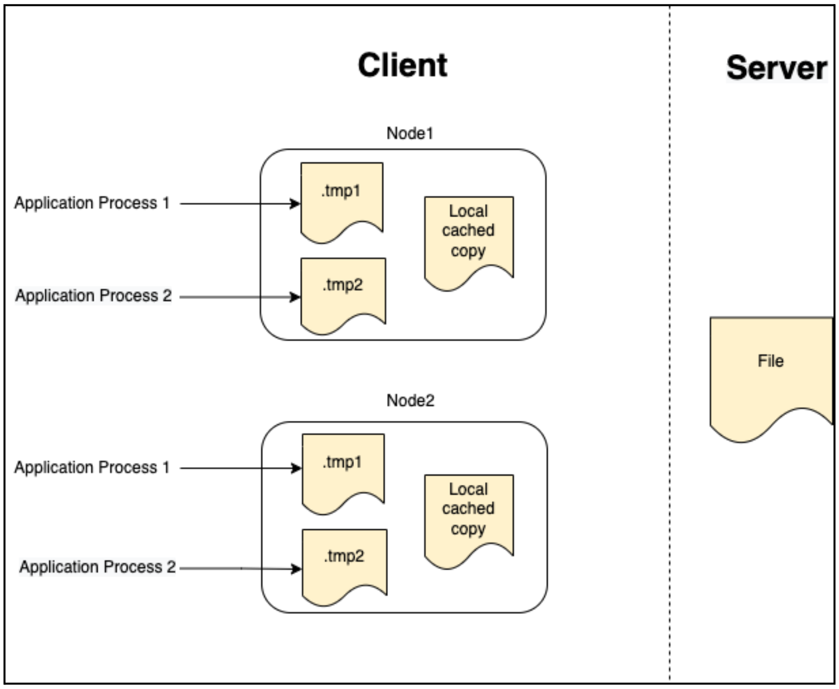
Close


Client
Server
Client receives close() call
Has the file been modified?
No
Yes
Delete .tmp
Rename .tmp to .consistent
Send whole file to server
Create .tmp file
Flush content to .tmp
Rename .tmp file to actual filename
Rename .consistent to actual filename

- Fuse-client checks for changes in .tmp file.
- .tmp file is deleted if there are no changes. Else, .tmp file state is changed to .consistent file (renamed)
- File is then sent to the server Server creates a .tmp for every close op, and then atomically renames it - LAST WRITE WINS
- After sending the file, .consistent file is used to create the cached copy for that node

Read/Write

- Read/ Write are local to each client. No server interaction is involved.
- Each application process performs read/write on its own copy of cached file (.tmp file created during open)



PERFORMANCE

For performance, we conducted tests with python scripts which emulated the scenarios and recorded the time taken by each call by decorating the API calls.

Case 1: The client creates the file for the first time

The client creates the file from variable sizes and stores them on the server. Calls recorded in this test were close and create calls. Figure 1 reports our findings.

Observations:-

- Create time is independent of the file size
- Close time increases with an increase in file size as expected

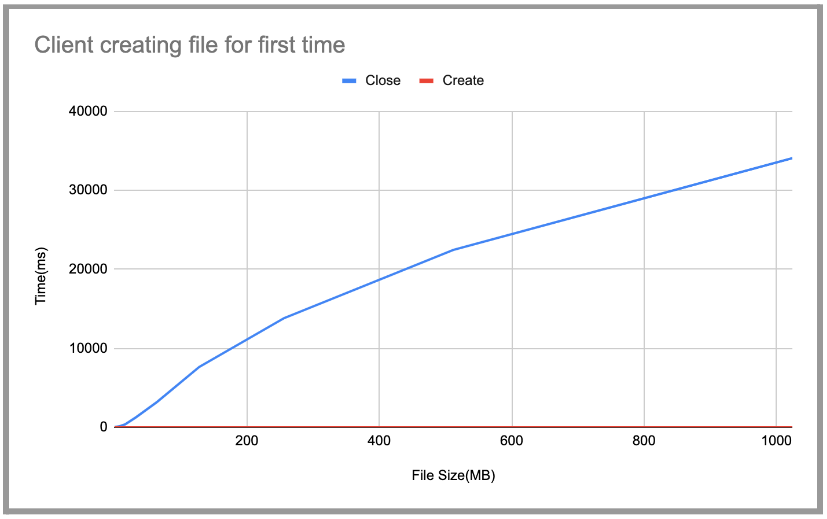


Figure 1. Client Creates the file, write data and closes it

Case 2: Client fetches file that existed in the server

For this test, the file is initially present on the server and is opened by the client. Calls recorded are open and close. Then we open the file again at the client locally and report the time to open and close for it [Figure 2].

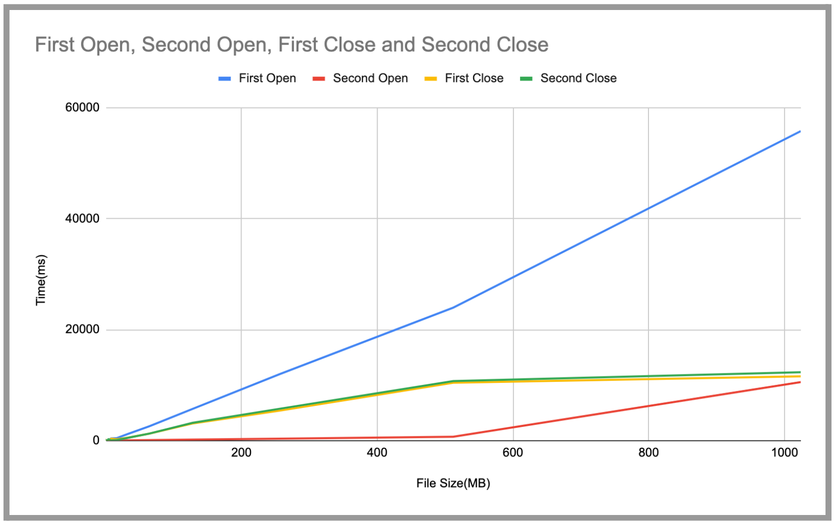


Figure 2. Client opens the file not present at local

Observation:-

- Only first open is increasing with file size. Since file is not changed by the client, other operations are not varying with file size

Case 3: Write vs Read

A file is written of variable size and write call monitored in this part. Same file is opened and read. Read api is monitored for this part[Figure 3].

Observations:-

- No server/GRPC communication is involved here
- As expected, read and write time increases with file size.
- These are slower than normal read/write because of fuse intercept

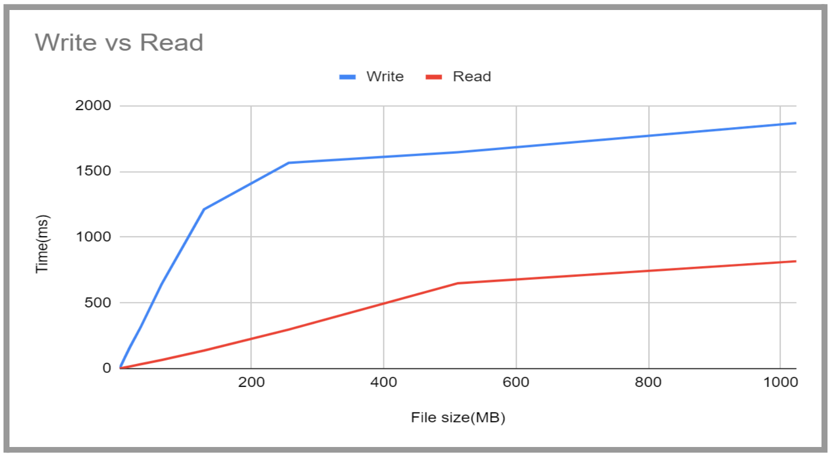


Figure 3. Client writes the file and reads the same file.

Case 4: Scalability

Clients processes are varied for this case. All clients are tested for writing a new file of 100 MB and then reading that file. Open API call for reading the file present at the server and close api for new file is reported here[Figure 4].

Observations:-

- Latency increases with increase in number of client processes
- This is due to the fact that client side fuse implementations are currently single threaded.

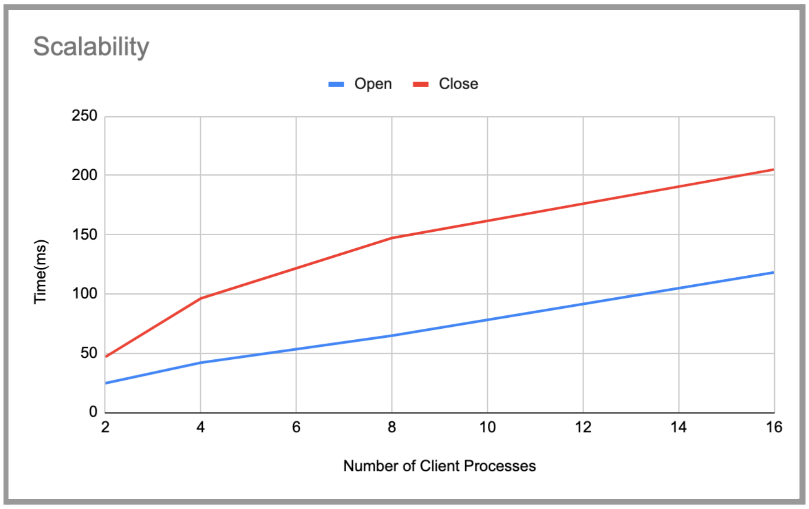


Figure 4. Number of clients perform file operations concurrently

Case 5: cost wrt gRPC stream packet size

The packet size is varied here from 4KB to 1MB. Same operations as case 4 are performed here and Open and close are reported in Figure 5.

Observations:-

- We experimented gRPC streams with different packet size during open() and close()
- Higher packet size improves open() and close() latency, and reaches optimal around 256KB

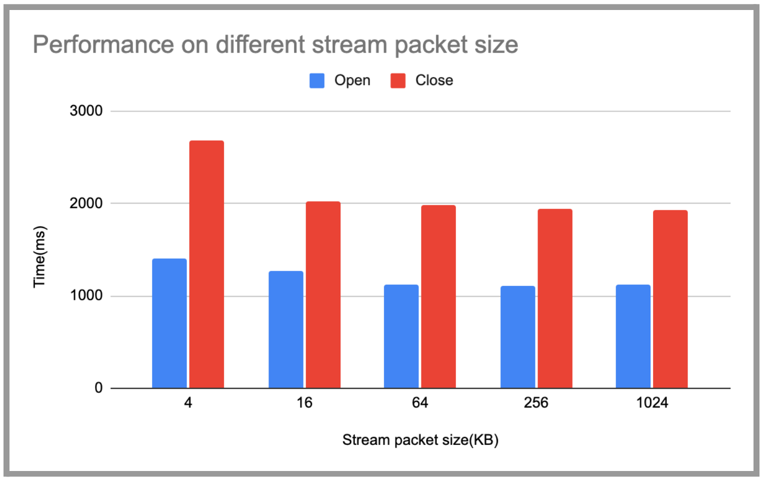


Figure 5. Cost for file operations with variable gRPC packet size

DURABILITY AND RELIABILITY

- Each application process per client creates a **.tmp** copy for their read/writes
- If the client/server crashes before application process could call close, the tmp file is dirty and is deleted during **garbage collection**
- Renaming a .tmp file to **.consistent** file on close saves the state of the file from dirty to consistent. It is an **atomic** operation, hence safe
- On restart, the client process deletes all the .tmp files and recovers the .consistent files to local cache and server.

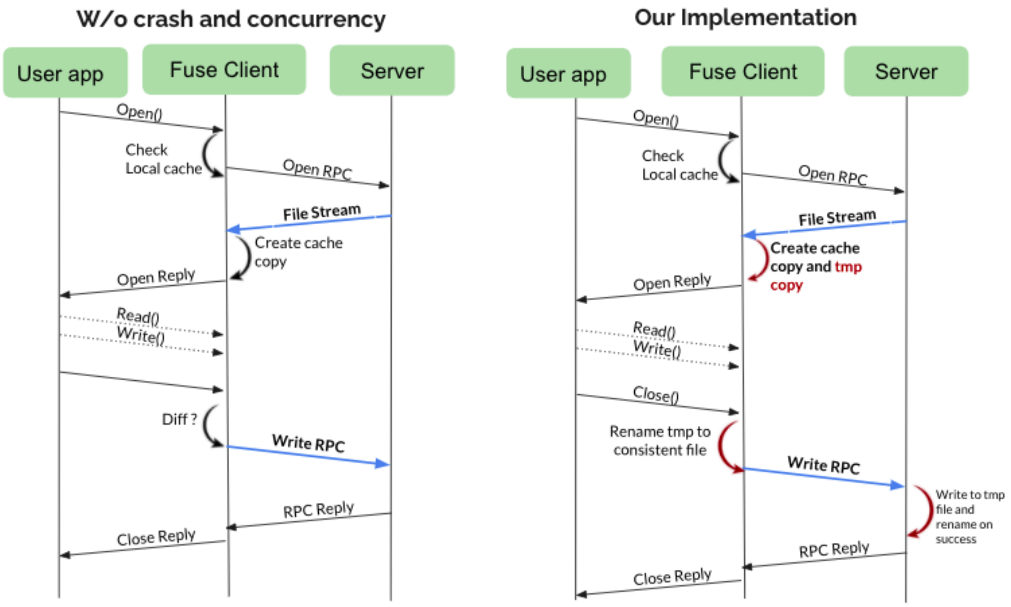


Figure 6. Simple AFS without crash and concurrency (left side) and proposed implementation (right side)

Figure 6 showcases overheads added in the proposed algorithm for concurrent client and crash recovery. Left side of the figure showcases a simple AFS protocol where the client/server need not implement concurrent clients separately. All of these clients read and write to a global cached copy. So there are issues like inconsistent data inside cached copy due to multiple writes. Also, there is no feature of crash consistency. Once the client crashes, there is no way of knowing if the cached file data changes are properly flushed by the client or if the client crashed in between updates (dirty file data). The blue arrows showcase the file streaming between client and server and depends on the file size. Hence, these are the most heavy RPC calls between the client and server.

The right side of Figure 6 showcases the proposed solution. To handle concurrent writes, each client creates a local copy of the cached file. It reads and writes to its unique tmp copy of the cached file. Once the close is called, we change the state of the file to .consistent files. .consistent files represent clean files whose data is properly written by the client and are closed. This way we can differentiate between the clients who crashed before calling close and ones who crashed after calling close. So, the recovery of .consistent clients can be done during the next client reboot.

Concurrent Read/Writes by Application Programmes

Since, each client creates its personal copy of the cached file, there would not be any inconsistent scenario of different clients writing to the same data. There are atomic renaming operations done to the temp files on close. Since, file system guarantees the rename (move operations) are atomic and safe from concurrency, it was the best choice for the problem statement of concurrent operations.

Server also handles this issue of concurrent writes by creating a temp file and flushing client data into the temp file and performing the rename operation in the end to the original file. Here, the last writer whose rename operation is performed last wins.

Overhead of Proposed Algorithm

As showcased in Figure 6, the main overhead of the proposed algorithm is to create a temporary copy during open for each client. This overhead is proportional to the file size and increases with increase in file size (as showcased in benchmark scenario). Other operations like atomic rename adds little overhead as it just changes the file path (using the same file structures). There is little overhead due to these calls in open() and close() APIs.

Crash Recovery

Once the client reboots, it checks for all the .consistent and .temp files inside the cache directory. It deletes all the .tmp files (dirty files need no recovery) and flushes all the .consistent files to the server. If there is no difference between the sent file and the server stored file, no change is made to the file. The server always updates the file according to the last writer reaching the server wins scenario. There are different crash scenarios in between the flow of open/close/read/write APIs as showcased in Figure 7 and Figure 8. We have attached demo videos showcasing these scenarios in detail and their expected results.

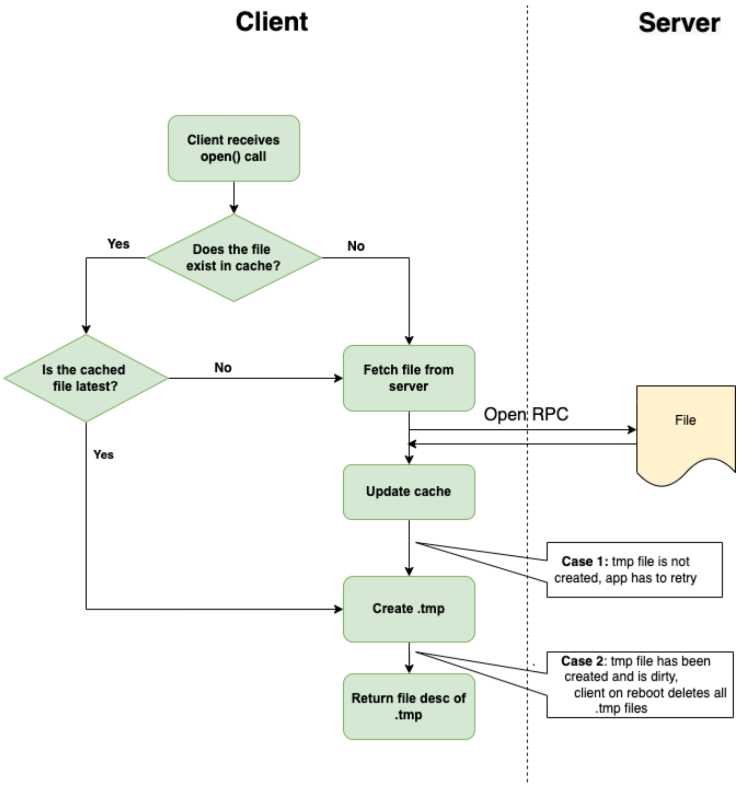

Client
Server
Client receives open() call
Does the file exist in cache?
Yes
No
Is the cached file latest?
No
Fetch file from server
Open RPC
File
Yes
Update cache
Case 1: tmp file is not created, app has to retry
Create .tmp
Case 2: tmp file has been created and is dirty, client on reboot deletes all .tmp files
Return file desc of .tmp

Figure 7. Possible crash scenarios during Open

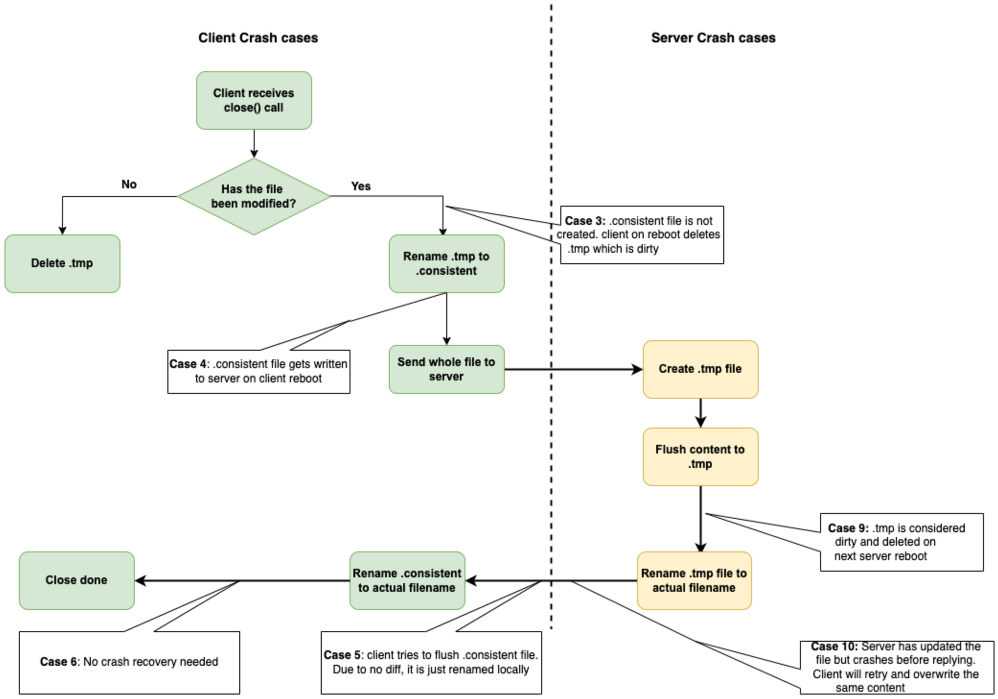

Client Crash cases
Server Crash cases
Client receives close() call
Has the file been modified?
No
Yes
Delete .tmp
Rename .tmp to .consistent
Case 3: .consistent file is not created. client on reboot deletes .tmp which is dirty
Case 4: .consistent file gets written to server on client reboot
Send whole file to server
Create .tmp file
Flush content to .tmp
Case 9: .tmp is considered dirty and deleted on next server reboot
Close done
Rename .consistent to actual filename
Rename .tmp file to actual filename
Case 6: No crash recovery needed
Case 5: client tries to flush .consistent file. Due to no diff, it is just renamed locally
Case 10: Server has updated the file but crashes before replying. Client will retry and overwrite the same content

Figure 8. Possible crash scenarios during Close

CONCLUSION AND LEARNINGS

It was interesting to see how the fuse works and realizing the power of gRPC stream in real use-case. Building AFS without supporting any of the durability and reliability was straightforward. However, making it fault-tolerant, reliable and crash-consistent required some re-thinking at the system-design level. Adding each one of them was a trade-off, as it made the system “correct”, but at the expense of costly operation. This is often the case with building highly reliable, highly fault-tolerant systems (and same with life, too!)

REFERENCES

1. https://en.wikipedia.org/wiki/Andrew_File_System
2. https://en.wikipedia.org/wiki/Network_File_System
3. libfuse/libfuse: The reference implementation of the Linux FUSE (Filesystem in Userspace) interface (github.com)