CS 739 - AFS-like Distributed FS

-LincoInspartacus James
-Kaustubh Khare
-Prabhav Adhikari
-Madhav Kanbur

High level architecture:

- Client implements whole file caching and replicates the server directory structure. On open, it sends a request to the server and fetches the whole file and stores it in its local disk. All the read/writes will be local. On close, the dirty file will be flushed back to the server.
- Server is stateless in our design and the client keeps track of the server's modified timestamp.

POSIX API‘s supported:

- Open
- Close
- Mkdir
- Rmdir
- Unlink
- Creat
- Stat
- Read/pread
- Write/pwrite

Client design:

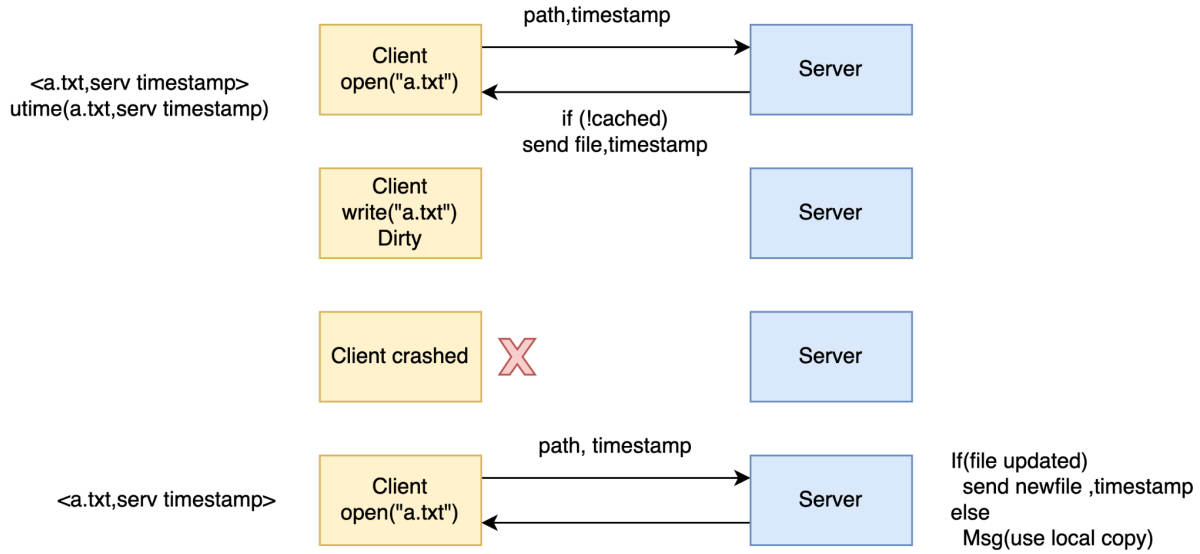
Cache layer:

- Client keeps track of server modified timestamp persistently as well as in-memory structure (hash map) to be in sync with the server and avoid extra fetches if the latest copy of the file is cached.
- Every open request contains the path of the file and the latest modified timestamp on the client side. If the file is not cached, the timestamp value will be set to 0.
- In order to maintain the **open to close** consistency, every open call will be sent to the server and the modified timestamp is checked against the one in the server. In case of matching timestamps, the server sends a message to use the local file on the client side. If it doesn’t match, then the new file is sent along with its modified timestamp.
- On receiving the file, the client updates its in-memory/persistent hashmap and also the timestamp of the file saved locally using utime() system call.
- All the read/writes will be sent locally.

- On close, the client checks the modified timestamp of the file with the previously stored one in the hashmap structure to determine if the file is dirty or not. The dirty file will be flushed back to the server and the latest modified timestamp will be obtained as a response.

Client crash recovery:

- Consider the below scenario in which the client has crashed after making a series of updates to the file locally.
- The client will not lose the data on subsequent open if the file has not been modified on the server. It will check the status by sending the timestamp on open requests,if it is updated, the server will send the new file. If not the client can continue working on the previously modified file and can send this file on the subsequent close.



Server design:

- Server is stateless in the design
- Server includes the errno in its response to the requests it received from the clients for normal POSIX operations and a separate error code (EIO) is used to distinguish the gRPC error in case of connection issues.

Server crash recovery:

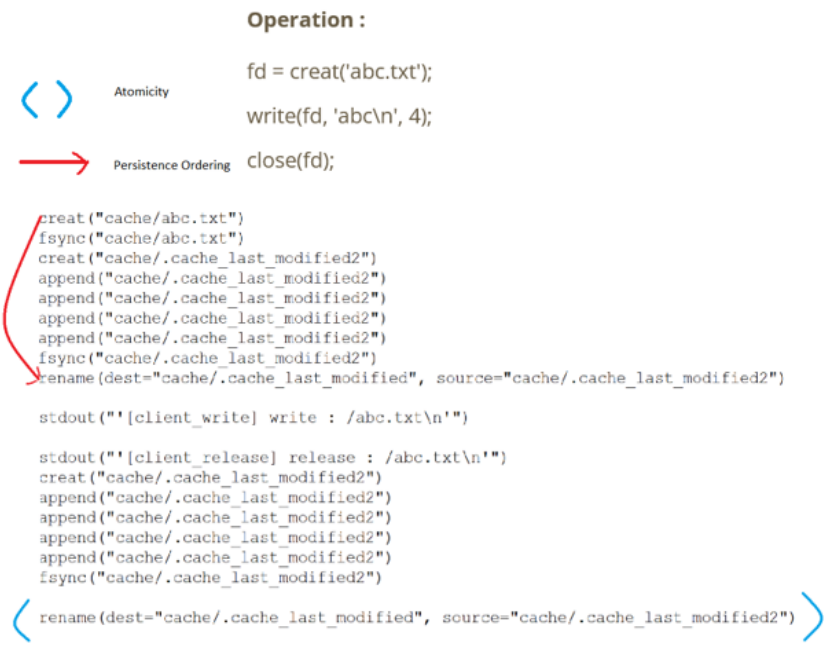
- On reboot, no extra steps are required from the server. It can receive subsequent requests from the clients after reboot.

Durability

Server: Since our server is stateless, we don't need to maintain any special data structures etc. The only concern is to ensure that files are persisted to disk safely in the close() function. This is achieved by using a local update protocol (write to temp file, rename to destination) for file updates. Furthermore, data mixing never occurs.

Client: To test the durability of our client code, we used the ALICE tool to find vulnerabilities for 2 common cases – 1. creation-writing-closing, 2. Unlinking of a file.

Case 1 : Create() – write() – close(). For unknown reasons (possibly a bug in ALICE), the write() system call doesn't show up in ALICE's logs. Nevertheless, 2 vulnerabilities were detected under the default APM –

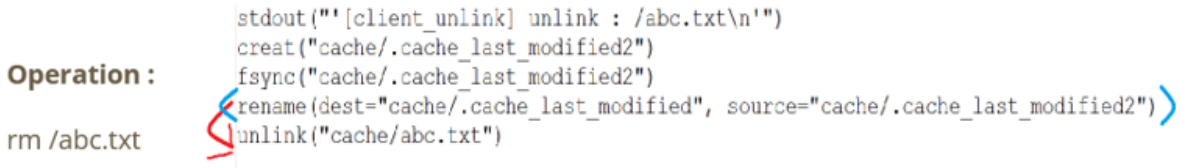


a. [ORDERING] create("cache/abc.txt") -> rename(".cache_last_modified", ".cache_last_modified2");

b. [ATOMICITY] rename(".cache_last_modified", ".cache_last_modified2");

However, since we use the **ext3-ordered** configuration as our underlying FS, none of these vulnerabilities affect us, as per recent literature (OTRUNC APPEND -> Any OP is unaffected and DirOps are atomic).

Case 2 : Unlinking a file



Vulnerabilities :

a. [ORDERING] rename(“.cache_last_modified”, “.cache_last_modified2”) -> unlink(“cache/abc.txt”)

b. [ATOMICITY] rename(“.cache_last_modified”, “.cache_last_modified2”)

For ext3-ordered, DirOp -> Any op remains unaffected, and as before DirOps are atomic.

Reliability

1. **Basic AFS-like protocol :** We fetch a file from the server and show how full-file caching works. The second read is performed locally. Lastly, we append some data to the file and flush it back to the server.

Video link -
https://drive.google.com/file/d/1BeRa02EJ80U8m3jEIK5kP5Uewa4tiAFo/view?usp=sharing

2. **Client Crash :** In a controlled setup, we crash the client FUSE process just after the client finishes a write() but before it calls close(). In this case, the user continues working with their dirty data after recovery and finally flushes the file to the server upon close().

Video link (part 1) –
https://drive.google.com/file/d/1wOO8ibMUbBLNRqXl4lLqXJVi_7uXP5wU/view?usp=sharing

Video link (part 2) –
https://drive.google.com/file/d/1--Fk_BEw1Y5m3w1MHl_FSVDgjwwyoRPS/view?usp=sharing

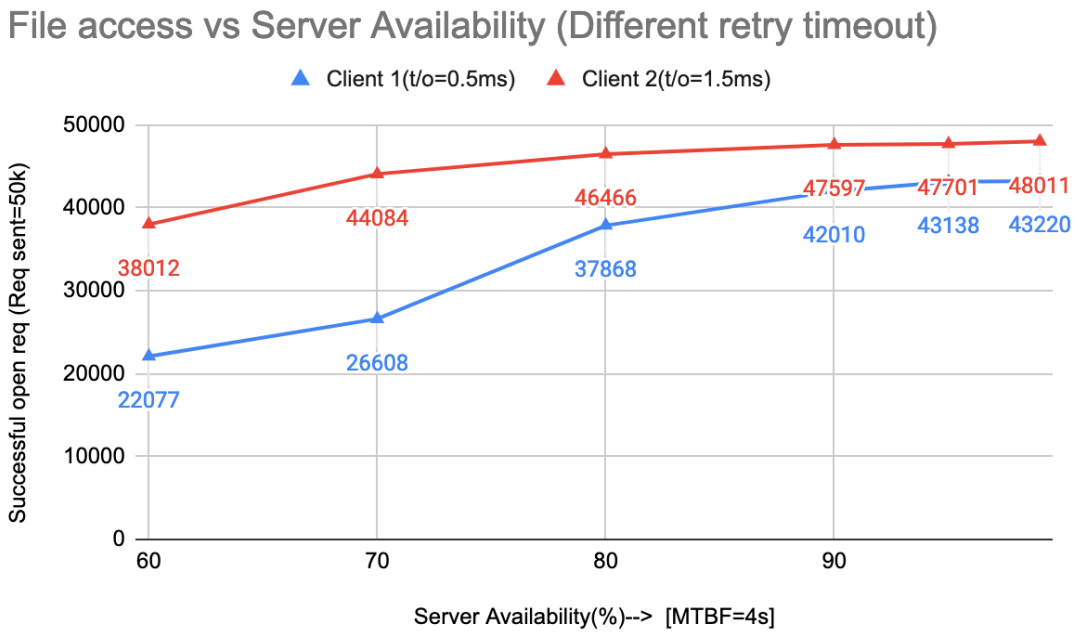
3. **Server Crash :** We crash the server when it's writing to a file in the close() function. Upon recovery, we see that the old contents are still present thanks to our local update protocol. The user is never presented with data corruption/mixing.

Video link -
https://drive.google.com/file/d/1t8uoKtcniTPdPfsExuYx7AMB-clXkzvW/view?usp=sharing

Performance

File Access vs Server Availability

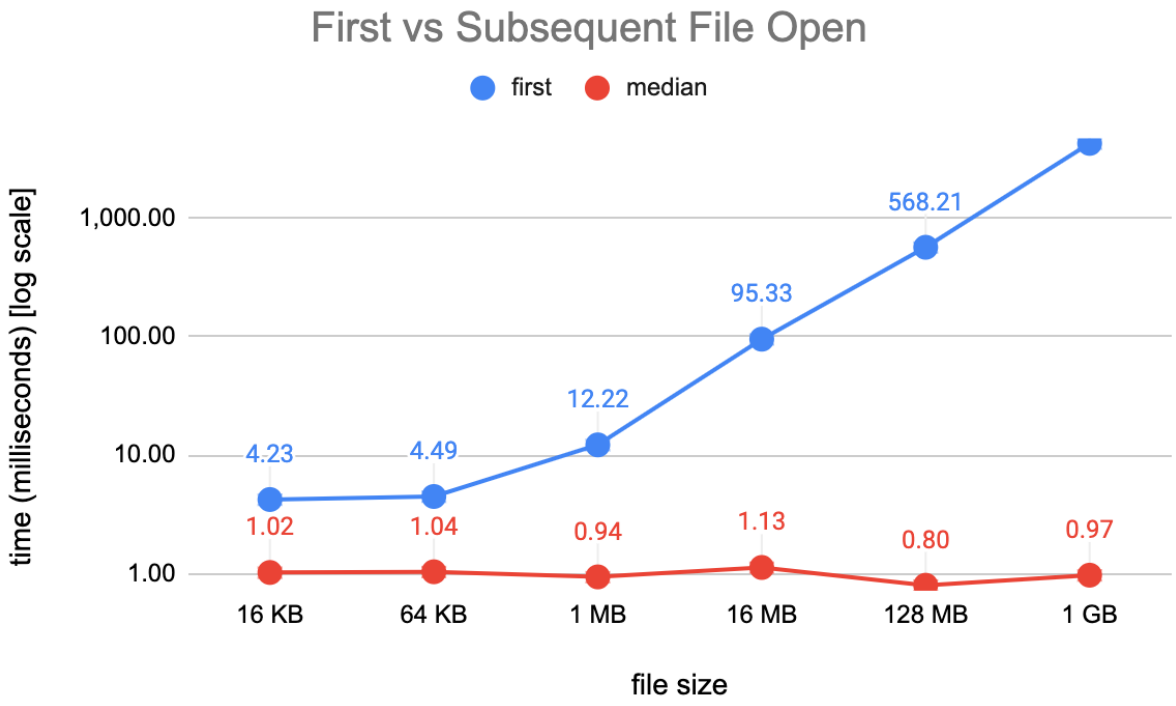
- Studied the impact of server's availability (MTBF=4s) on the clients having varied timeouts.
- The server's MTBF is kept at 4 seconds and the MTTR is varied to adjust server's availability.
- Two clients with different retry timeouts are accessing the same file in the study.
- From the graph, it can be observed that the number of successful open requests in the client 2 with increased retry timeout is more than the client 1 with less timeout.
- In summary, even though the server availability is low, the retry timeout in the client can be adjusted in such a way to have more successful file accesses.



File Open

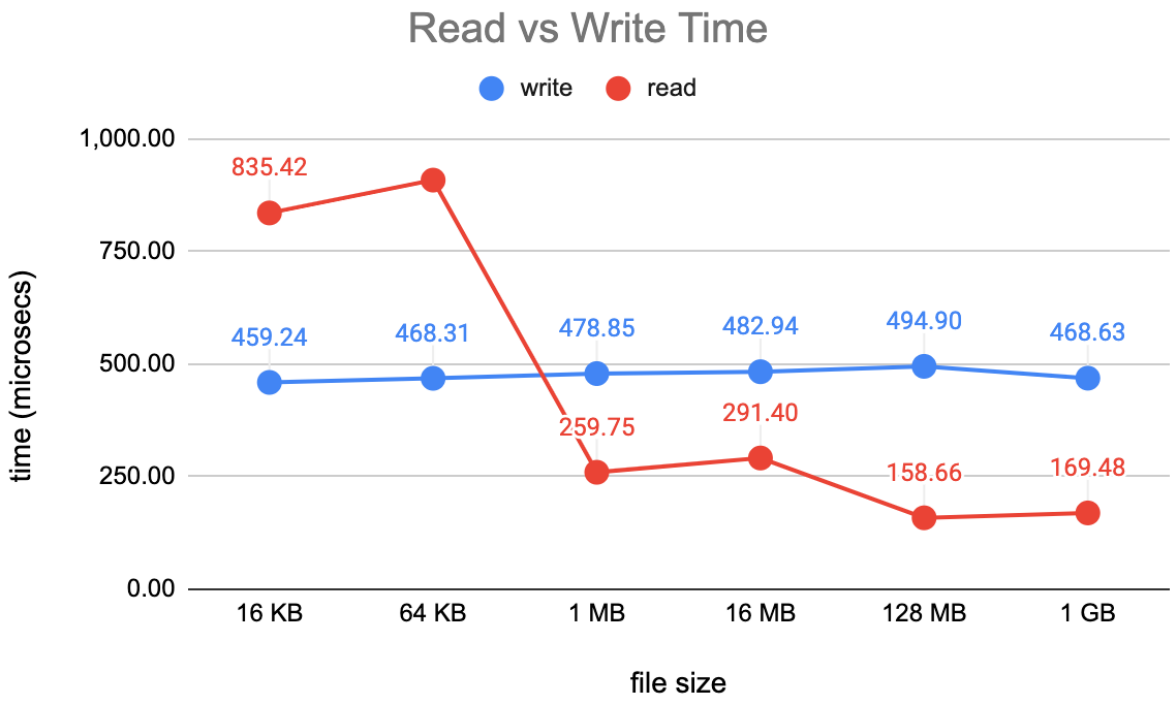
We compare the time required to open files of varying sizes. We consider 2 cases here:

1. File is not in the cache
   As seen in the figure below, the blue line represents the case when the file is not cached. The time required to open the file increases linearly as we increase the file size. This is expected because the file is being fetched by the client from the server over the network using gRPC and thus the network transfer time is the dominating factor.
2. File is in the cache
   As seen in the figure below, the red line represents the case when the file is previously cached and there have been no changes to the file. Thus the same cached file is reused and , as expected, we observe that the time required to open the file remains almost the same with increasing file size.



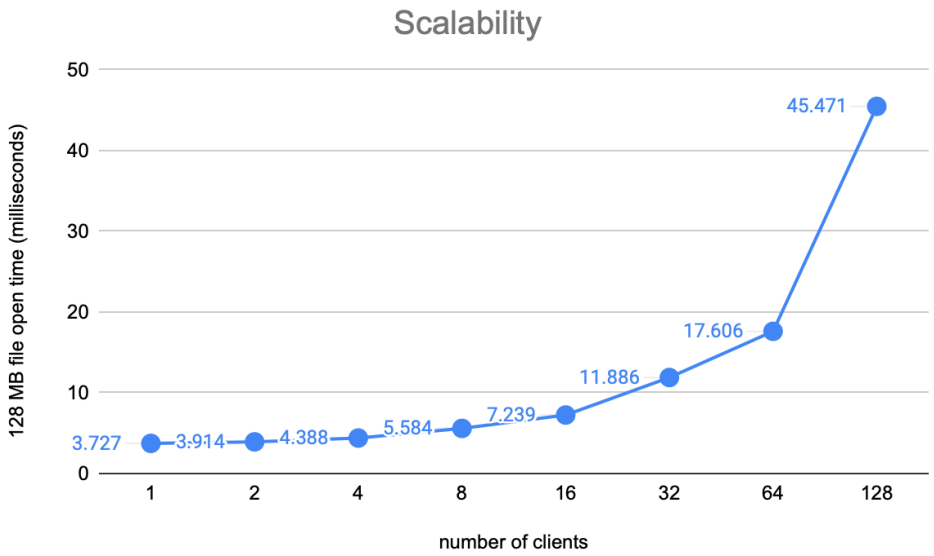
Read and Write

We compare the time required to read and write files of varying sizes. We observed a surprising discrepancy here. Even with increasing file sizes, write times remained similar and the read times decreased. Even after running the experiment multiple times we observed the same pattern but couldn't figure out the reasons for this deviation.

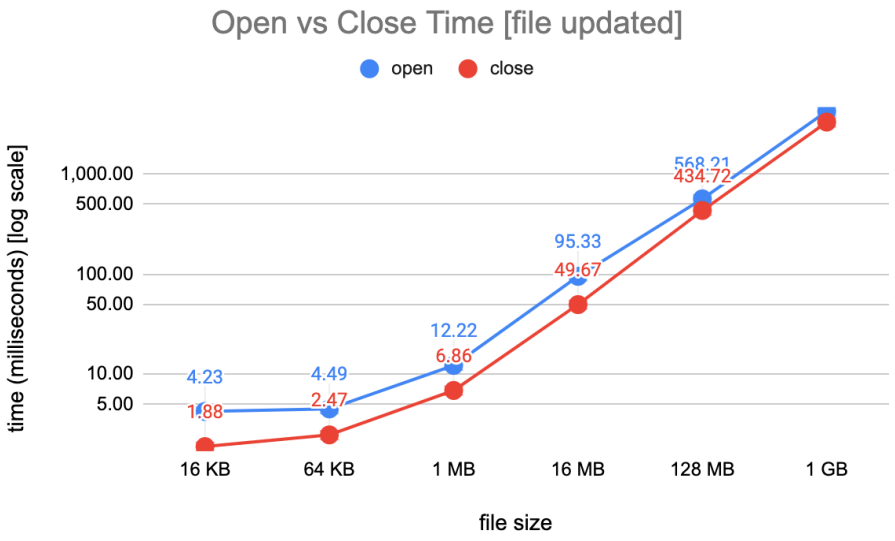


Scalability

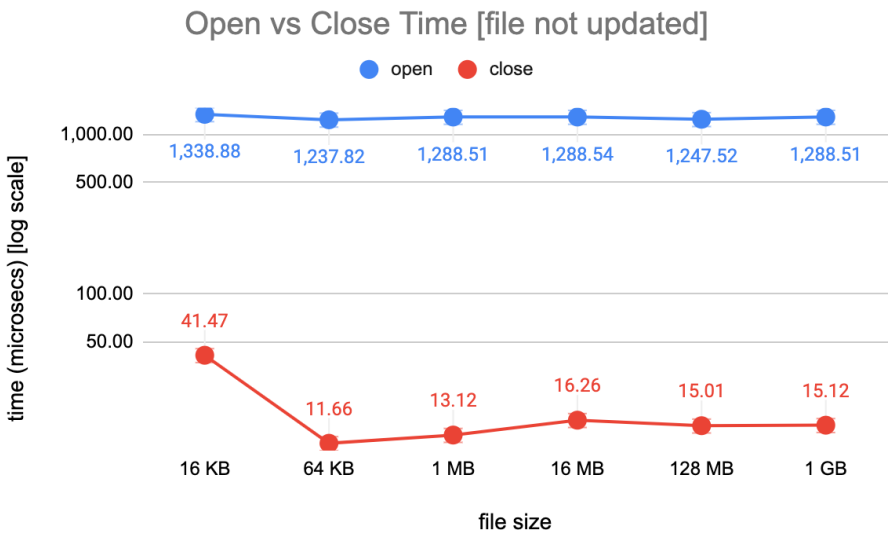
We measured the scalability of the system by spinning up multiple clients and measured the average time required by each of them to concurrently open a 128MB file which wasn't cached locally. As seen in the graph below, with an increasing number of clients, the time required to fetch and open the file increases almost linearly until 64 clients. However, with 128 clients we observe a drastic increase in the time for opening the files which suggests that we reached the threshold performance to scale linearly for our setup with 128 clients.



Open and Close Times

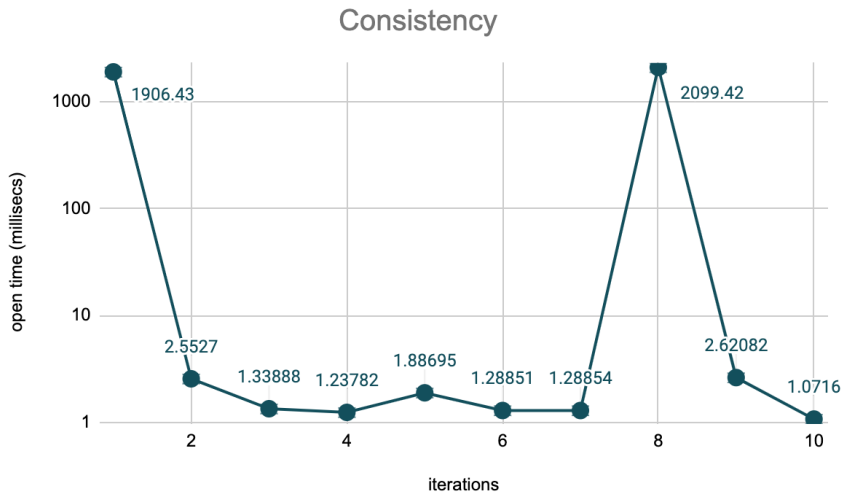


For increasing file sizes, open and close time keeps increasing. This is caused due to whole file fetching (in case of opens) and flushes (in case of close).



When the whole file is not fetched, the close is faster than open, as there is no network overhead. The times don't change across file sizes as expected.

Consistency Protocol



The file was being operated multiple times, and the time taken for each open was noted. The first peak is expected due to the file being fetched as discussed earlier. The file was changed in the server before the eighth open, resulting in the second peak - almost similar to the first open - which is expected. The file was fetched again because of our consistency protocol that the client must see the updated file if it is updated on subsequent opens.

Conclusion

Whole file caching with AFS-like semantics allows us to perform very quick local read/writes with the only expensive operations being the first open() and close(). We had to make multiple conscious decisions considering the tradeoffs between performance and reliability. We learnt how ensuring reliability might affect performance and vice-versa. Lastly, this was a very interesting project which was full of learnings and fun.