WiscAFS

Aadi Swadipto Mondal Mohil Patel Rahul Uday Chakwate
Project Group 1/8

1 WISC AFS DESIGN

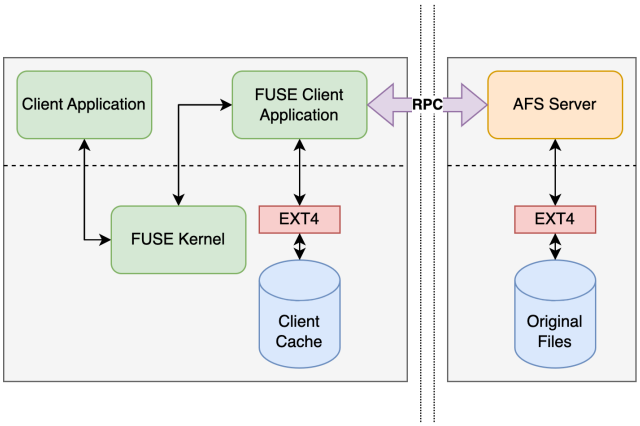

Figure 1: Basic Design

1.0.1 Key Design Principles. Figure 1 describes the basic design of WiscAFS. The following points describe the key design principles of WiscAFS.

(1) The server does not maintain any non-persistent state. This is a well-thought decision to aid in server crash consistency.
(2) We use the last modified timestamp of files to track when they are updated. Clients completely rely on server timestamps to track such updates. This relieves us from the assumption that server and client clocks are synchronized.
(3) Client cache can store copies of files in the server. A file in the client cache can have three states:
   (a) The file can be absent in the cache.
   (b) The file is marked temporary (having extension **.fusetmp**) which denotes that the file is currently open by one or more processes in the client machine.
   (c) The file is marked permanent (having extension **.fuseper**) which denotes that no process has the file currently opened in the client machine.

   We assume that files in our file system don't have these two extensions. Files in the cache are stored in the format **<filename>.<server-timestamp>.<extension>**. The **server-timestamp** is the last-modified time of the file last known to the client. This gets updated only when the client tries to open or close the file. Note that the **server-timestamp** can be stale if multiple clients are accessing the same file.
(4) Client is not stateless. The client tracks open files and maintains particularly three states: file-descriptor when first opened, number of times the file is simultaneously opened (open is called multiple times for the same file before calling close), and dirty bit of the file (whether the file has ever been written since first open).

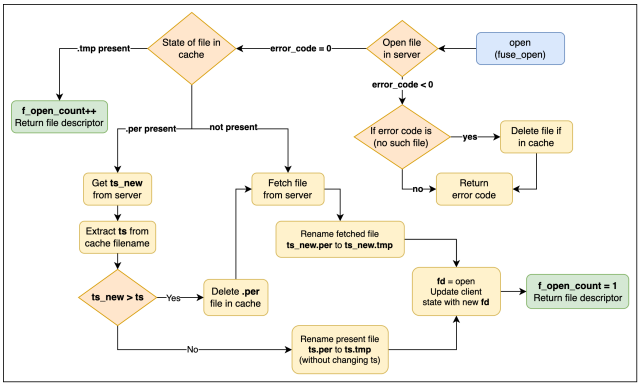

Figure 2: Open Protocol

1.0.2 Open Protocol. The open protocol, as shown in figure 2, has the following steps.

(1) Open call in client first tries to open the file in server with the exact same flags passed by the application (in client). If it fails, we check if the error code is *"File not found"*, in that case, we remove any cache copies if present.
(2) On a successful opening in the server, we look for the state of the file in the cache.
   (a) In case the file is already opened, we update the client state and return the same file descriptor. Our goal is to provide **first-open-to-last-close consistency**. If one file

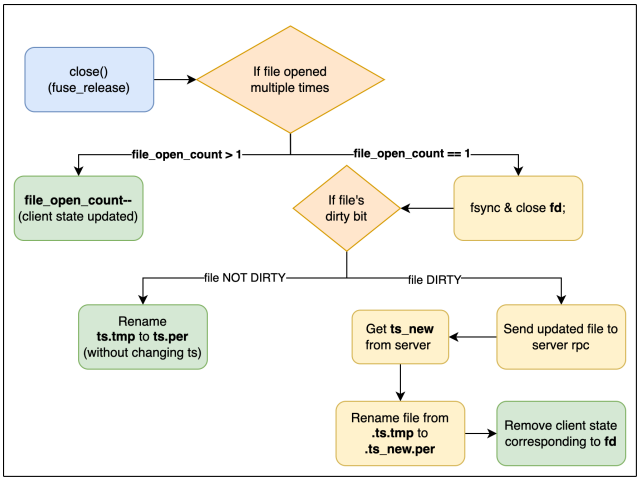

Figure 3: Close Protocol

is opened multiple times (without close), we update the count of times the file is opened. the FUSE process in the client maintains only one file descriptor for such simultaneous opening of the same file.

(b) In case the file is not present in the cache, the file is fetched from the server via gRPC streams. After that, the file is marked as temporary as described previously. Open RPC also fetches the last modified timestamp of the file which is recorded with the filename in the client cache.

(c) In case the file is present in the cache, we fetch the last modified timestamp from the server and compare it with the value recorded in the client cache. If we see that the file in the client cache is stale (the file has been updated in the server since the last close in the client), we simply delete the file and act as if the file was absent in the client cache. If the client cache is not stale, we simply mark the file temporary as described previously.

(3) If the file was not previously opened, we add a new entry in the client state. If the file is ever written later, the client state is updated to mark the file dirty.

1.0.3 Close Protocol. The close protocol, as shown in figure 3, has the following steps.

(1) If the file was opened simultaneously, we simply update the client state to reflect that close was called. If this is not the last close, nothing more is done.

(2) If the file was unlinked in opened state, we simply close the file on client and return success.

(3) If this is the last close, the dirty bit is checked for. In case the file is dirty, we use gRPC streaming to send the file to the server. The server initially writes the file to a temporary file. After the server receives the entire file, it renames the temporary file (assumed atomic) to the actual filename. The server sends a success code on completion.

(4) Upon receiving the success code from the server, the client marks the file as permanent as described previously. Similar to open RPC, close RPC also fetches the last modified timestamp from the server, which is updated is recorded with the filename in the client cache.

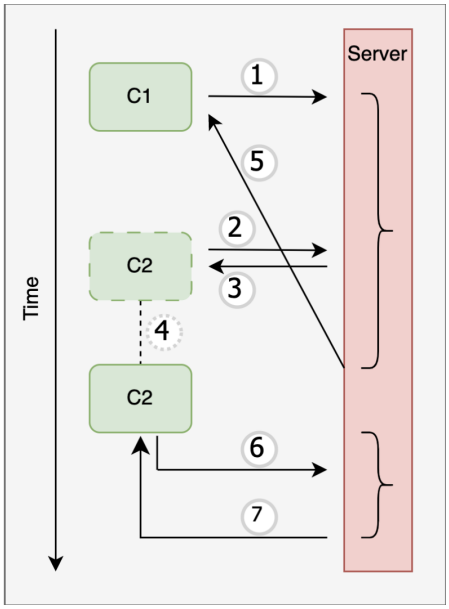

Figure 4: Close Wait Protocol

1.0.4 Close Wait Protocol. Files may be big and take time to send from the client to the server. It is possible that another client tries to close the file when one client is interacting with the server during file-close. A scenario with the detailed steps is shown in figure 4 where client 2 requests file-close when client 1 is interacting with the server. In this scenario, when client 2 sends a file-close request to the server (step 2), the server responds with a wait message (step 3). Client 2 randomly waits using

an exponential back-off algorithm (fails after 16 retries) and re-sends file-close requests after waiting (step 6). If the server is not interacting with any other client for that particular file, the server serves the request for client 2.

1.0.5 Crash Consistent Design. On client crash and subsequent restart, the client clears all files in the cache marked temporary (essentially the opened files during the crash). Since the client state is only composed of opened files and their file descriptors, it is justifiable to lose them as they are only local to the FUSE process. The client can lazily fetch files lost during a crash from the server as and when required. Files that were closed (marked as permanent) remain in the cache and are available for use after the client restarts. This ensures that the file is never left in a corrupted state in the client cache.

The server is inherently stateless except when the client is sending the updated file to the server (the server initially writes in a temporary file and then performs a rename operation) during file-close. Similar to the client, the server deletes the temporary files in an event of a crash. The client can send the file again if the server crashed in between. This ensures that the file is never left in a corrupted state on the server. After the crash, the old version of the file is available for use.

1.0.6 Unlink. Unlink forwards call to the server (deletes from the cache if successful). We also track files that were deleted while the file was open. To comply with POSIX semantics, such files are ignored upon close as described previously in the close protocol.

1.0.7 Utimens. There are primarily two major challenges:

(1) We depend on the monotonic increase of the last modified time for server files.
(2) We cannot allow the client time to overwrite server time (this happens when files are moved from outside the mount point directory).

To counter these challenges, we ensure that last modified time is never set to old times for a newly created file. For an already existing file, the maximum of the current last modified time and value given by the application for the new value, ensuring monotonic increase of time. Finally, utimens is never directed to the server.

This can be a potential issue if applications depend on last modified time of files when files are moved across directories within the WiscAFS mount point or from outside to WiscAFS mount point. The server will not see any updates as a result of utimens since it is never destined to go to the server.

1.0.8 GetAttr & GetXAttr. Write on opened files (directed to client cache copy) may change the file size. GetAttr and GetXAttr fetch all information from the server via RPC, except for the file size which is overwritten to the client cache file size.

1.0.9 File Permissions. Files are always created with all permissions (777). This is to handle complex permission issues like writing only for the first time, read-only thereafter. Operations related to file permissions, specifically **chown** and **chmod** are NO-OP operations.

1.0.10 Directories. All directory-related operations are forwarded to the server. Directories in the cache are created when the file is opened in client. Directories are deleted when empty directories are left after the file unlinks (or *File Not Found* error codes).

2 RESULTS

2.0.1 Filebench. Table 1 shows the results of filebench across wiscAFS and unreliablefs. In the results, **mongo** is a macro benchmark and all the other benchmarks are microbenchmarks.

There are two interesting observations between unreliablefs and wiscAFS. First is all the operations where we see similar latency and throughput numbers for both filesystems. These operations are readfile, writefile and sync (fsync) operations. We expect this to happen because in wiscAFS these operations execute directly on local cache copy without any interaction with the server, so they show similar measurements as unreliablefs. Additionally, note that in seqwrite's writefile wiscAFS is showing slightly worse performance than unreliablefs. In the write operation, although it is entirely on the local cache file, we update the client state to track things like dirty bit, etc in wiscAFS. This could be the reason for the slower performance.

In contrast, the second is those operations that show a considerable degradation in measurements for wiscAFS compared to unreliablefs. This includes operations like statfile, closefile, openfile, deletefile, etc. All these operations involve server interaction in wiscAFS, which significantly degrades the performance compared to unreliablefs.

Lastly, there are a few discrepancies that we could not understand. In mongo benchmark readfile1 is showing

Workloads	Ops	Tput(mb/s) wiscAFS	Lat(ms/ops) wiscAFS	Tput(mb/s) unreliablefs	Lat(ms/ops) unreliablefs
create	appendfile	118.883	8.367	161.48	6.166
createfiles	closefile	0	4.294	0	0.031
	writefile	0.047	1.796	0.489	0.063
	createfile	0	14.639	0	0.112
createrand	sync	0	14.183	0	14.449
	appendfile	76.194	5.057	101.441	3.41
delete	deletefile	0	78.713	0	1.328
readrand	readfile	63.991	0.015	64.991	0.014
writerand	writefile	0.794	2.441	4.879	0.393
seqread	seqreadfile	5459.415	0.181	5420.266	0.183
seqwrite	writefile	120.184	8.29	157.779	6.317
stat	statfile	0	66.022	0	0.026
writefsync	sync	0	22.725	0	22.855
	appendfile	5.556	1.379	69.082	0.087
mongo	deletefile1	0	6.153	0	0.135
	closefile2	0	0.037	0	0.016
	readfile1	0.703	0.112	39.061	0.049
	openfile2	0	8.861	0	0.033
	closefile1	0	0.042	0	0.016
	appendfile1	0.333	1.637	19.826	0.068
	openfile1	0	5.28	0	0.035

Table 1: Filebench results

bad performance in wiscAFS & in createfiles benchmark writefile operation is bad for wiscAFS. We suspect that this could be due to workload properties (things like small writes, small reads, files not in local cache during read, etc.).

2.0.2 xv6 and Parallel-Build. Both xv6 and parallel-build workloads work. The videos for the same are in the presentation slides. For xv6 we are able to build and run xv6 on top of wiscAFS. And for parallel-build we are able to build leveldb. We also see improvement in build time as we increase the number of threads

3 EVALUATION

3.0.1 Client Cache Consistency. Along with the basic cache consistency test (test 1) provided to us, we implemented 3 more tests to evaluate wiscAFS as shown in the slides. In test 2, client B modifies a file which is then modified by client A. Upon reopening the file, we assert that client B should see the updated contents and not the old contents in its cache. In test 3, we make sure that a file is not flushed to the server on a write call but only on a close file call. In test 4, we let both clients run asynchronously for many iterations and let them open, read or modify the file in a probabilistic way and close the file so that we are not certain of the cache state at any point in time. Finally, we evaluate if the file on the server is in a consistent state and not a mixed state.

3.0.2 Client Crash consistency. We designed tests to evaluate the crash consistency of wiscAFS. In the client crash test, client A opened and modified the file, but crashed before closing the file and did not flush its updates. Upon reboot, we verify that the file is re-fetched from the server during the open call and client A sees the server content.

Note on server crash. We observed that when the server crashes and reboots, gRPC is facing issues re-initializing the communication with the server since the gRPC channel is broken between the server and the client. Thus, we also had to kill the client and let it reboot to establish a connection with the server.

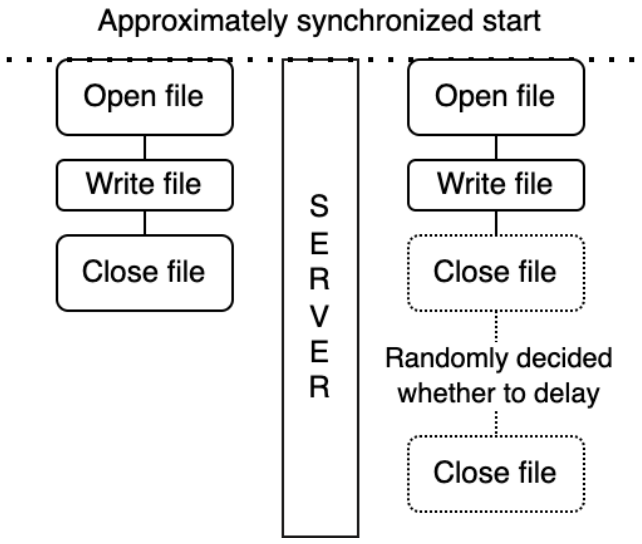

Figure 5: Option A: Who is the winner?

3.0.3 Option A: Who is the winner? In this test, we evaluated the impact of network randomness on WisCAFS as shown in figure 5. We have two clients A and B opening, modifying, and closing a file on the server. We observe that there exists a fixed average delay between the arrivals of the two requests on the server when we synchronize the clients. We then inject a variable delay with a fixed probability of 50% in the flush call of client B. This variable delay will impact the probability of client B winning as described by the following analytical model. This is a good analyses of random delays that is usually introduced due to randomness in networks.

$$\text{when fixed delay} >> \text{variable delay}$$

$$p(\text{B winning}) = 0.5$$

$$\text{since any client can randomly be a winner}$$

$$\text{when fixed delay} << \text{variable delay}$$

$$p(\text{B winning}) = p(\text{delay in A}) * p(\text{B winning})$$

$$+p(\text{no delay in A}) * p(\text{B winning})$$

$$= 0.5 * 1 + 0.5 * 0.5 = 0.75$$

We intend to study this transition of probability through experimentation. This transition is shown in Figure 6 which plots probability against variable delay (in ms). The average arrival time difference between client A and B calls is plotted in Figure 7.

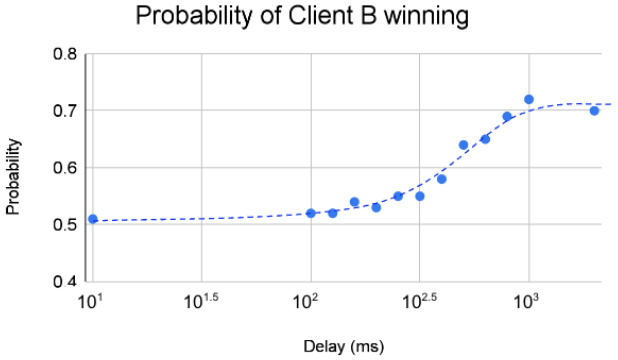

Figure 6: Probability of Client B winning. X-axis is the delay in milliseconds, y-axis gives the probability.

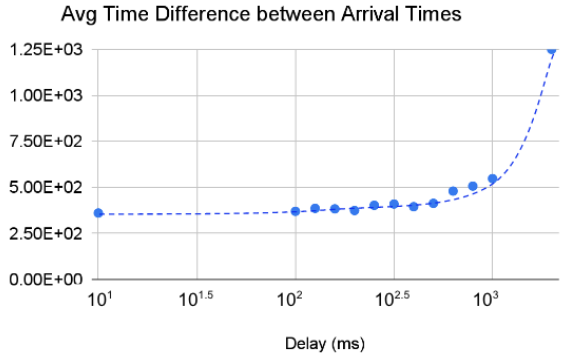

Figure 7: Average time difference (in ms) between the arrival of flush calls of client A and client B plotted against delay (in ms).

4 DURABILITY

For durability we designed *errinj-alice-delay* and *errinj-alice-reorder*. In *errinj-alice-delay*, when the error is injected we delay the operation by 1 filesystem call duration, i.e. it will be executed when the next fs call happens. Similar in the *errinj-alice-reorder* we execute the operation after running the next filesystem call. Both are implemented using a simple queue storing the error-injected operation.

Figure 8 shows the workload that we designed to show the potential durability issue in crash conditions. This workload is inspired by the Git application issue found in the ALICE paper. In the workload designed, we are adding *errinj-alice-delay* in the rename operation (shown in red). From the client application's perspective, rename shows success so it won't be worried about its

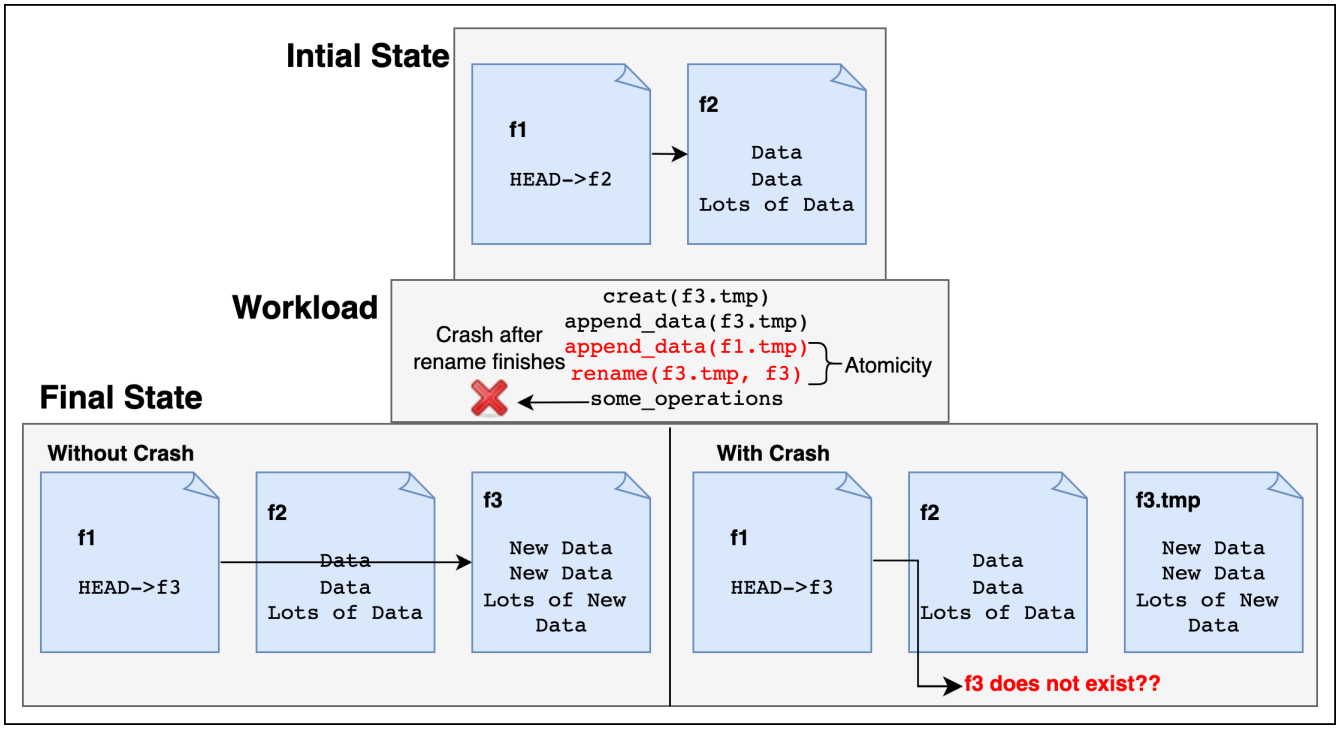

Figure 8: Durability Workload

state. But due to delay if there is a crash after rename operation it is possible that the client will see a corrupted state on the disk because the operation never really got pushed to persistent storage.

5 TAKEAWAYS

5.0.1 *Filesystem & POSIX Semantics.*

- close() (fuse_release) operation executes lazily and ignores the return error code.
- POSIX semantics allow for an open file to be unlinked. wiscAFS needs to handle this scenario while doing the close operation.
- In ***mv*** command, it creates a new file and uses ***utimens*** to update the last modified time in past. If the designed system assumes a monotonic increase in time, this can cause issues.

5.0.2 *Permissions.*

POSIX semantics allow ***create*** call to open a file in *rdwr* mode for the first time and change its permission to *read-only* after it is closed. We tried to emulate this operation but it proved hard as we wanted to keep the server stateless. Thus in our implementation, everything has *rwx* for all the files.

5.0.3 *Consistency.*

Because of variable network latency, the last writer wins semantics in AFS should only be seen when the server receives the request. We cannot see it from the client's perspective. Due to network latency even if client B's request was sent later in (real) time, client A can be the winner if its request arrived the last.

5.0.4 *Durability.*

Applications may assume filesystem persistence properties which may not be true due to added optimizations. This can lead to data corruption in potential crash conditions.

5.0.5 *Integrating large C & C++ code.*

Integrating large C & C++ codebases can be complicated. It took us few days just to get a basic integration of unreliablefs (C) with grpc (C++) and compile it end-to-end.