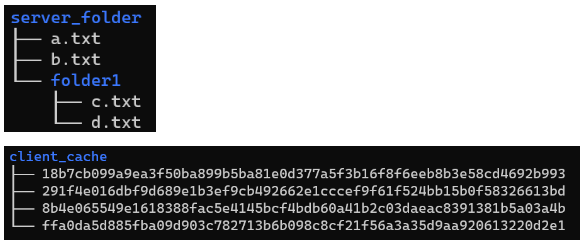
CS 739 Project 2 : Client/Server File System

Group 10
Cody Tseng, Do Men Su, Vyom Tayal, Chahak Tharani

0. Design

0.1. Cache folder structure



```
server_folder
├── a.txt
├── b.txt
└── folder1
    ├── c.txt
    └── d.txt
```

```
client_cache
├── 18b7cb099a9ea3f50ba899b5ba81e0d377a5f3b16f8f6eeb8b3e58cd4692b993
├── 291f4e016dbf9d689e1b3ef9cb492662e1cccef9f61f524bb15b0f58326613bd
├── 8b4e065549e1618388fac5e4145bcf4bdb60a41b2c03daeac8391381b5a03a4b
└── ffa0da5d885fba09d903c782713b6b098c8cf21f56a3a35d9aa920613220d2e1
```

Figure 1. Cache folder structure

The local cache's folder structure is shown in Figure 1, where the file name is the SHA256 value of the server's pathname. This implementation can support different file system naming rules on servers and clients and also eliminate the need to maintain an in-memory map.

0.2. Read-write sharing

One big difference in our implementation compared to POSIX is that we do not support write-sharing. The system will copy the content of the file to a new "dirty file" when the user issues a write and redirect the write to the dirty file. We made this decision since it is more consistent with the whole system and experiments on the CSL machine show similar behavior.

0.3. Dirty file handling after system crash

We considered several different approaches to try and keep the dirty files. However, some approaches will potentially flush corrupted data to the server, e.g., flush all dirty on reboot; and others will greatly reduce performance, e.g., maintain a journal for the dirty file. So we decided to drop all the dirty files upon reboot.

Due to this decision, we will send the data to the server during a fsync call, since there will be no reason to flush the data to the local cache. (Also, the visible on fsync behavior is also observed on the CSL machines.)

0.4. Design summary

The overall system design can be seen in Figure 2. Notice that there are two folders on the client, cache and temp. When the user opens a file, it first downloads to the temporary folder. After the download completes, the system will rename it to the hash value of the pathname and move to the cache folder. If the user issues a write, the system will copy the file locally and redirect the application's write. Finally, when the user closes the file, we will send it to the server which also uses the same renaming trick.

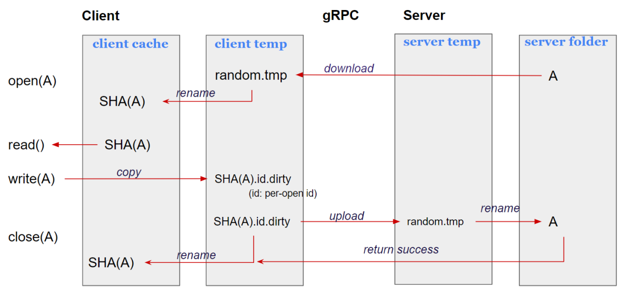


Figure 2. Overall system design

1. Functionality & Correctness

1.1. Posix Compliance

POSIX	FUSE	POSIX	FUSE

open()	open()	read()
		pread()
			read()

creat()	create()	write()
		pwrite()
			write()

unlink()	unlink()	close()	flush()
mkdir()	mkdir()		release()
rmdir()	rmdir()	fsync()	fsync()
stat()	getattr()	rename()	rename()

Table 1. POSIX function implemented

Table 1 shows the POSIX function we implemented and their corresponding FUSE operation. An optional rename function is also implemented. The biggest POSIX spec violation is no write-sharing, as mentioned in 0.2.

1.2. AFS Protocol and Semantics

We defined our RPC interface as shown in Table 2. Most calls are one-to-one mapping to FUSE operations with some exceptions. For example, since the system uses whole-file cache, it uses local copy during reading and writing. It will recheck with the server using a getattr call during an open call, which means that if two clients have the same file in the local cache and one client updates it, the other will technically have a stale cache. However, the stale cache will be updated upon the next open call.

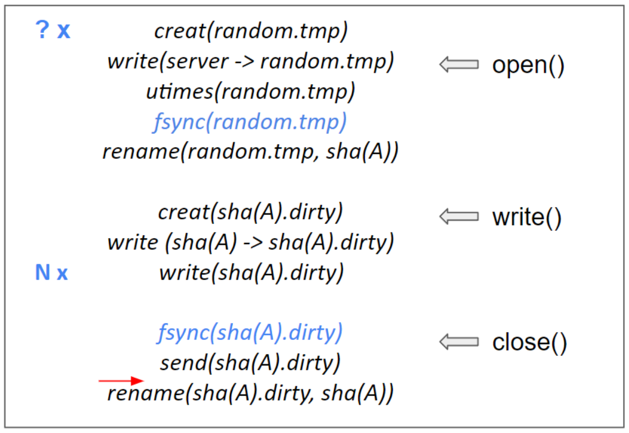
Additionally, during a close or a fsync call, the system will send the data to the server, combined with the use of a temporary transfer file on the server, the system ensures that no data will be mixed.

getattr	Return the stat of a file
download	Return the binary content and modify time of a file
upload	Send the binary content and modify time of a file
mkdir	Create a directory
rmdir	Remove a directory
create	Create an empty file
rename	Rename a file
unlink	Unlink (delete) a file

Table 2. RPC Interface

1.3. Durability

We don't want to impose too many restrictions on the underlying file system, so we tried to insert fsync when needed. However, handling non-atomic directory operations will require a magnitude of additional work, so our implementation will require a file system with atomic directory support. In our experiments, ext4 with data ordered mode is used.

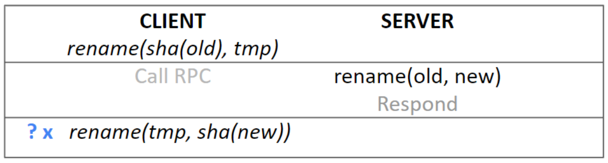


```
? x      creat(random.tmp)
       write(server -> random.tmp)          ⟸ open()
         utimes(random.tmp)
         fsync(random.tmp)
       rename(random.tmp, sha(A))

         creat(sha(A).dirty)                ⟸ write()
       write (sha(A) -> sha(A).dirty)
N x      write(sha(A).dirty)

         fsync(sha(A).dirty)                ⟸ close()
         send(sha(A).dirty)
       rename(sha(A).dirty, sha(A))
```

Figure 3. Local update of open, write, close

The system guarantees atomic writes, that is, the changes will only be visible after a successful close() or fsync(). In some rare cases, such as the client crash immediately after sending data to the server (red arrow in figure 3), the client might have an outdated cache file on its system, but it will be updated on the next open call. The fsync calls shown in the graph also help us to reach such a guarantee.

The rename procedure is surprisingly tricky, we initially went with a local rename after server rename approach. However, an issue will arise if the server rename successfully but the client doesn't get the response; as another client might rename an older file to the old name, causing the original client to use its outdated local cache on the next open call. We solve this issue by first renaming the file to a temporary name and only commit after the server responds successfully.



```
          CLIENT                      SERVER
    rename(sha(old), tmp)
          Call RPC                 rename(old, new)
                                      Respond
? x rename(tmp, sha(new))
```

Figure 4. Rename

1.4. Crash Recovery Protocol

The crash recovery protocol of our implementation is simple. On client recovery, it will clear its temporary folder while keeping its cache folder, since all the files in the cache folder are clean. Similarly, on server recovery, it will also clear its temporary folder. No special messages are exchanged and no additional metadata is stored on disk. Additionally, On Server failure, all calls will respond ENONET or error no network.

3. Performance Measurement

3.0. Experiment Setup

We performed our experiments on cloudlab machines. As shown in figure 5, our server was running on a dedicated physical machine and the clients were running on 15 virtual machines on 3 hosts.

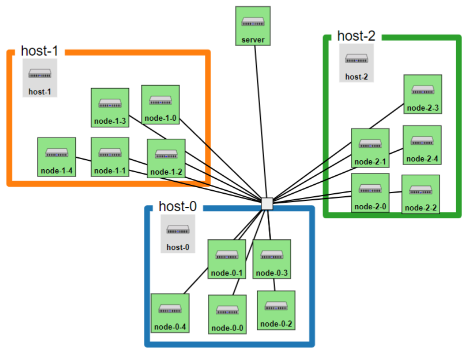


Figure 5. Experience setup on cloudlab

3.1. Performance Measurement

3.1.0. Read / Write Performance

We measured the performance of reading and writing to files of different sizes. The first read of a file requires the client to download the file from the server; the subsequent read can read from the locally cached files. The write always uploads the file to the server after closing the file on the client.

Figure 6 shows the result of our experiment. We can see that the first read is significantly slower than the subsequent reads. The first read operation downloads the file from the server and writes it to its local cache, so its performance is impacted by the network bandwidth and disk writing performance. As a result, the performance of the first write is bounded by the bottleneck, disk writing speed. However, though the subsequent reads do not produce any RPC calls and are only bounded by disk reading speed, its throughput is still less than 1/3 of EXT4 disk read throughput.

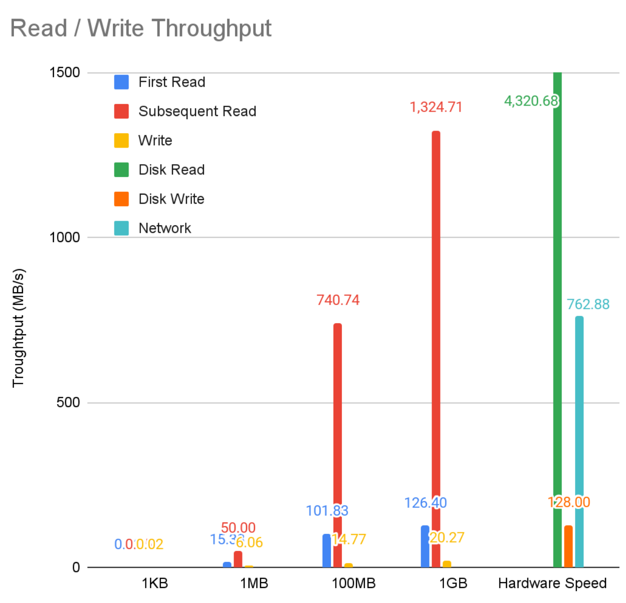


Figure 6. Read / Write Throughput

This figure shows the throughput of our file system and hardware limitation for comparison. The disk read / write speeds are measured with a local EXT4 file system and the network throughput is measured with iperf3.

3.1.1. Real-world Workload Benchmark

To test the performance of our file system, we ran 4 different real-world workloads on it.

- **Copy**: Copy an entire vscode (v1.65) source tree to fuse. The source tree is about 80MB and there are about 5000 files.
- **Remove**: Delete the vscode source tree with rm -rf.
- **SQLite**: Insert 2 rows in a large SQLite database (~3GB).
- **Make**: Compile htop 3.1.2.

We measured the performance of these workloads with different numbers of clients. The resulting performance is shown in figure 7. The copy and SQLite workloads require a large number of disk writing operations; thus, their runtime is much higher than the remove and make workloads. We can also see that the runtime slightly increases as the number of clients increases.

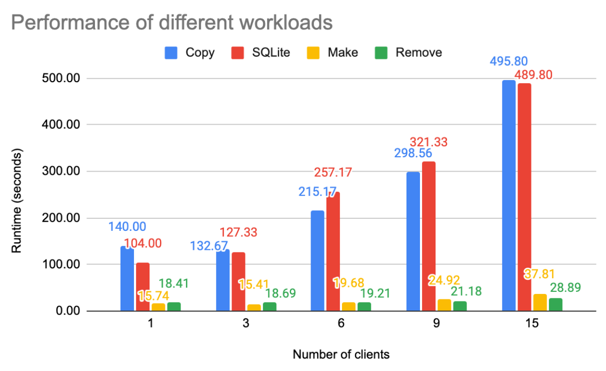


Figure 7. Performance of Different Workloads

3.1.2. Optimizations

To improve the performance of our file system, we think of two different optimizations.

- **Btrfs**: Use Btrfs to avoid copying the whole file on writes. Without Btrfs, our system creates the dirty file by copying the original file. This optimization explicitly tells Btrfs to copy the file with the copy-on-write mechanism.
- **Getattr Cache**: Cache getattr results for 1 second. This implementation avoids sending multiple getattr RPC calls for consecutive getattr.

To compare the performance improvement, we use two baselines for comparison:

- **Normal Implementation**
- **No Consistency**: This implementation reduces the consistency guarantee by not creating a dirty file when writing.

The performance of these 4 implementations is shown in figure 8. The getattr cache improves the performance of make workload. We reason that make relies on reading the file modification time, producing many getattr calls. Btrfs improves the performance of SQLite workload. Since the operation of inserting a few rows into a big SQLite database modifies a small proportion of a large file, the Btrfs copy-on-write mechanism can avoid copying the entire file. However, Btrfs implementation runs much slower on the copy workload. A similar result also happens if we run our normal implementation on Btrfs.

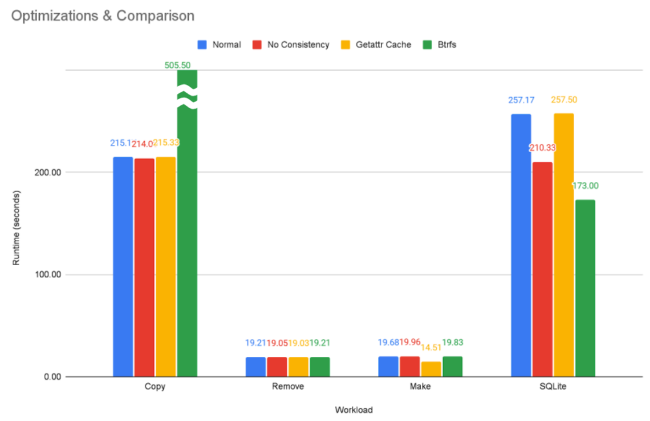


Figure 8. Performance of Different Optimizations

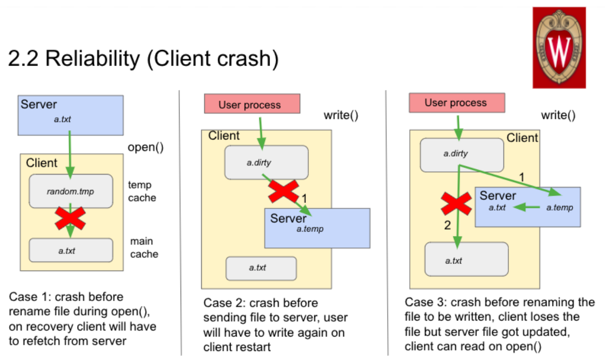
3.2. Reliability

To test that our file-system works like AFS-like protocol, we ran experiments to test the following scenarios:

- check file attributes on open()
- whole file is cached on the client and the next read will read from the cache instead of taking from server again
- writes are done to the local copy and flushed to server on close
- different clients/processes will get server's copy and update their local copies separately and last writer will win in updating the server copy which will then be retrieved by clients and local caches will be consistent again
- local cache is persisted to disk on clients
- when multiple processes and multiple nodes write to the same file, last writer wins

We experiment with various crash scenarios:

3.2.1. Client Crash Recovery


2.2 Reliability (Client crash)

Case 1: crash before rename file during open(), on recovery client will have to refetch from server

Case 2: crash before sending file to server, user will have to write again on client restart

Case 3: crash before renaming the file to be written, client loses the file but server file got updated, client can read on open()

Crash Scenario 1:
We crash the client after it fetches the file copy from the server to its temp cache but before it can rename it to its main local cache copy.

Recovery test: We observe that on next request of the copy, the client has to refetch the file from the server, which is in accordance to AFS protocol, if we copied to main cache file instead of temp and the client crashed in between the copy, it will have an incomplete version of the file but the time would be similar to that on server hence it won't refetch which is not expected. Therefore, temp cache file if renamed to main cache copy after complete write and rename is an atomic operation in ext4.

Crash Scenario 2:
We crash the client after it has written to the local cache copy but the written file is not yet sent to the server.

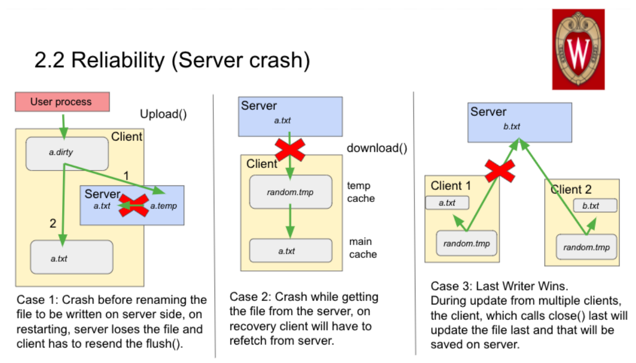
Recovery test: We observe that the client loses the written temp copy on restart. If the write was in the main cache copy, or the temp cache copy was renamed before the write on the server completes, we would not know if we have the complete data or partial data in our local cache if the client crashes in between write operations. Hence, we write to temp cache file and rename to main local cache only after successful write to server. This ensures reliable data on client and server.

Crash Scenario 3:
We crash the client after the write has been successful on the server but the rename operation has not been completed on the client.

Recovery test: We observe that the server has the latest copy of the file but the client has a stale copy of the file which will not pose a problem because the next access of the file after client reboots will check for the file modification time on the server and seeing a latter copy, will copy the file (which it wrote) to its local cache.This is expected and ensures reliability because client cannot rename the file to its main cache before send has completed for the reason mentioned in Crash Scenario 2. Even though the dirty file is deleted from client , this poses no issues in further functioning of the file system.

3.2.2. Server Crash Recovery

2.2 Reliability (Server crash)



Case 1: Crash before renaming the file to be written on server side, on restarting, server loses the file and client has to resend the flush().

Case 2: Crash while getting the file from the server, on recovery client will have to refetch from server.

Case 3: Last Writer Wins. During update from multiple clients, the client, which calls close() last will update the file last and that will be saved on server.

Crash Scenario 1:
Crash before renaming the file to be written on server side, on restarting, server loses the file and client has to resend the flush().

Recovery test: We observe that the server will delete all it's temporary files on restart and therefore we will lose the updated file for which the flush was called on the server side. If the write was to the main file, then on restart we would not know whether the file was written to completely or if the update was pending and therefore the server might save a corrupt file. Hence we write to temp files and then rename it to the actual name once the full write is complete to ensure reliability of data on the server.

Crash Scenario 2:
Crash while getting the file from the server, on recovery client will have to refetch from the server.

Recovery test: Here, the server will crash during the download of the file from the server to the client which will lead to no response to the client. The handling is pretty straightforward here as no state needs to be maintained and an open operation is idempotent so the client can just retry the open and get the correct file data in the next request.

Crash Scenario 3:
Last Writer Wins.
During update from multiple clients, the client, which calls close() last will update the file last and that will be saved on the server.

Recovery test: This is more of a protocol design rather than a crash and reliability test. As per the protocol, if 2 clients make an update to the same file and close it, then the client calling the close last wins, i.e. the update from the later client will be saved to the file and the update from the earlier client will be lost. This is by design and therefore it handles ambiguity amongst the different clients in the sense that the system is now made deterministic as it makes sense to make the last update to the file, in the case of competing flushes.