CS739 AFS Project

Marvin Tan, Yifei Yang, Rachit Tibrewal, Bijan Tabatabai

March 2022

1 Design and Rationale

Fetch	Returns the status, data, and last modification time on the server.
Store	Stores the data and returns the status and last modification time on the server.
Remove	Delete the specified file.
Create	Create a new file.
Rename	Change the name of a file or directory.
Makedir	Creates a new directory.
Removedir	Deletes the specified directory which must be empty.
Stat	Return status of a file or directory.
Readdir	Returns the list of files and directories in the directory.

Table 1: Server-Client Interface

We have designed and developed a filesystem with protocol similar to AFSv1[3]. The filesystem composed of two programs, a client and a server. The grpc calls available on the server program are listed in table 1.

The client is implemented as a FUSE filesystem. To keep performance at acceptable levels, whenever the client accesses a file from the server, it caches the entire file on its local disk and all subsequent accesses are done from that cache. To ensure cached copies remain consistent with what is seen on the server, we use a version file to store the last modified time of the server file at the time of retrieval. We utilize file attributes returned in stat() system call to store the version timestamp. The version file will be an empty file with the last modified time returned by stat() set to the last modified time of the server file(the version). Whenever a client opens a file, it sends a request to server to learn when the file was last modified. If that time is the same as the cache version timestamp, the cache has an up-to-date version of the file. However, if the server's version is newer than the cached version, the cached file is updated. Most filesystem system calls, e.g. mkdir and rmdir, are near one-to-one mappings with their corresponding RPCs, e.g. Makedir and Removedir. On the client side, we simply call the grpc function and does a couple extra steps. For unlink we simply remove any exist cache copies, but for create, since it also opens the file and returns the file descriptor, we also create a new empty cache, does a attribute call to the server and sets the version

file. The read and write system calls forward the corresponding operation to the client's local cache. On fsync, writes are only persisted to the local cache and not the server. Writes are only forwarded to the server with the Store RPC when the file is closed, or if a flush operation is called.

The server processes the RPCs by applying the request operations to its store of the filesystem. Because of this, the server's operation is mostly straight forward. The exception to the is the Store RPC, which is discussed in the next section.

2 Durability (1.3)

2.1 Filesystem Configuration

The server directory is mounted using ext4 with journal mode.

2.2 Crash consistency of local update protocol

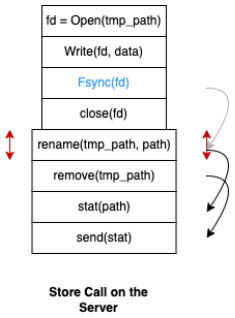

Figure 1: Store call on server durability diagram

2.3 Server Side Persistence

The server stores all files and directories in a targetdir which is passed as an argument to the server process. All metadata related operations like remove, rename, stat, makedir, removedir occur on the server itself. Store operation first writes the data to a random temporary file and then renames the file to the required file path.

3 Crash Recovery Protocol (1.4)

Specify what client should do after reboot; how to deal with possible client-side local persisted data?
The client does nothing extra after recovery, just performs as usual. For dirty cache data, when the next open occurs, we compare the last modification time between the cache version file and server file, if the server file is newer, which means the cache

copy is invalid, we discard the dirty cache data and use the server file, otherwise we use the cache copy.

Specify the actions that your file system does after the recovery clearly.

The server does nothing extra after recovery, just performs as usual. The client will retry the request when facing the server crash (with a user specified retry gap and retry limit). If the server recovers within the retry limit, the request will succeed and the operation will finish; otherwise a timeout error will be returned and the cache copy may be dirty. The dirty data will be handled by the same logic above.

Will your server send messages to the clients after rebooting? (You could assume to trust the client library).

No, server will not send messages to client after rebooting.

How your client would handle the RPC error code (e.g., GRPC's). It differs in that if the client FS finds RPC fails due to "the server crashed" vs. "you cannot create the file because it is existing".

We distinguish "server crash" and "operation error at server" by using "grpc::status" returned from the call. If the status is not ok then we conclude the server crashes. If it's ok then we check the "status" field in the call reply, where we specify the errno code set by the server side system call on error. If there is no error on the server side system call, the status field should be 0

Is there additional information needed to be persisted for the recovery to work?

For each open() at the client, we create a cache version file to record the last local modification time of the server file (i.e. version of the clean cache copy), which is used to check dirty and validity. Cache copy is dirty if cache copy is newer than the cache version file. Cache copy is valid if cache version file is as new as the server file (impossible to be newer than server file).

4 Performance (2.1)

We ran our experiments on a set of Cloudlab c220g5 machines [2]. One of the machines was designated as the server and the others as the clients. The server's storage directory is mounted on an ext4 filesystem with data journaling enabled on a 480GB Intel DC S3500 SATA SSD. The clients' data caches are mounted in a similar fashion.

4.1 Read Performance

We test the read performance by running a microbenchmark that measures the time it takes to open and read the contents of an uncached file on our filesystem. Then, since the file is now cached, it measures the time it takes to open and read the file again. We repeat this process 50 times, deleting the cached file between iterations, and report the average times.

Figure 2 shows these read performance measurements relative to the time it takes to read the same file on a local ext4 filesystem. We also measure how long it takes to read the files on a "passthrough" FUSE filesystem that simply calls the underlying system calls without any extra processing to show FUSE's overhead. As expected, reading an uncached file takes longer than reading a cached file because the client must ask the

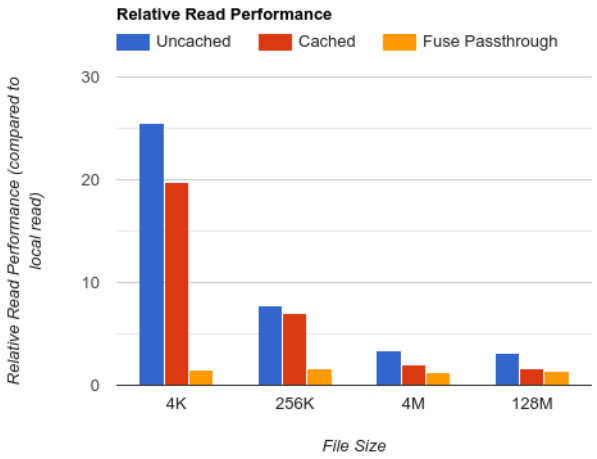

Figure 2: The time it takes to read a file in our system when it is uncached, cached, or read through a passthrough FUSE fs, relative to the time it takes to read the same file on a local ext4 fs.

server for a copy of the file before it can perform a read. Additionally, it takes small files relatively much longer to perform a read on our AFS implementation compared to large files. This is because whenever a file is opened, the client must check with the server to see if the cached version is still valid. Our measurements show that this process takes about 1.25ms to complete, which is significant when compared to the time it takes to read a 4 kilobyte file from disk. However, we do see that the read performance of large files in our system is comparable to the performance of the passthrough filesystem.

4.2 Write Performance

We measure the write performance of our filesystem in a similar manner to the read performance, except we consider the time it takes to open, write, and close the file.

?? shows the write performance measurements, along with the passthrough measurements, relative to the time it takes to perform the same write on a local ext4 file system. Writes take relatively longer than reads because in addition to writing the changes on the local disks, when the file is closed, the changes must be sent over the network to the server to be written again.

4.3 Scaling

Finally, we measure how our system performs under load. We do this by running our read microbenchmark on up to 20 different hosts and measuring how the performance compares to when it is run on only 1 host.

Figure 4 shows the results of this experiments with 5, 10, 15, and 20 clients accessing the server at once. We see that when there are 20 clients heavily excersing the

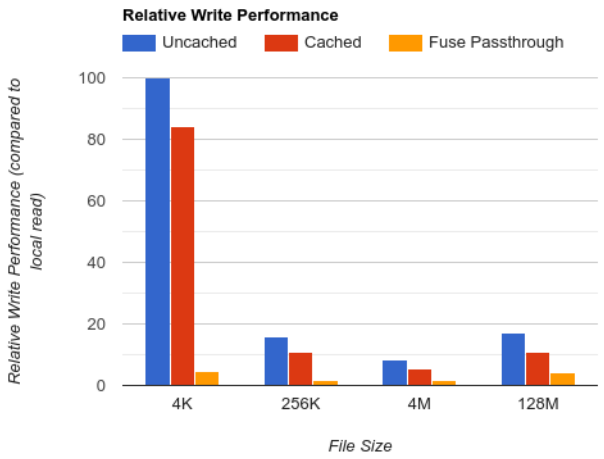

Figure 3: The time it takes to write a file in our system when it is uncached, cached, or written through a passthrough FUSE fs, relative to the time it takes to write the same file on a local ext4 fs.

Figure 4: The performance degradation of the server as more clients perform the read benchmark

server we see a modest 11% increase in read latency.

4.4 Potential Improvements

We have thought of several ways to potentially improve our performance that we regretfully did not have time to implement. First, currently, we always retrieve a file from the server on open if it is not already in the cache. However, if the file is opened with the O_TRUNC flag, we can skip this step because the data will be thrown out anyway. Second, we could have tried to compress large files before sending them over the network. Finally, the server calls the open and close system calls every time it accesses a file. Caching the file descriptors of frequently accessed files could reduce the time using those system calls.

5 Reliability (2.2)

5.1 AFS-like Protocol

In our implementation, we made care to match the consistency semantics listed in the AFS paper [3].

- **Write to an open file by a process on a workstation are visible to all other processes on the workstation immediately but are invisible elsewhere on the network**. In our implementation, this is followed as all processes on a client view and make updates to the same cached copy of the file. After a write operation is finished, all read operations after the write will see the update because the read is performed on the same cache copy as the write does, given that these operations are on a same workstation.

- **Once a file is closed, the changes made to it are visible to new opens anywhere on the network. Already open instances of the file do not reflect these changes.** In our implementation, on closing the server version of the file is updated. Subsequent opens on that file will use the latest version, because the cache validation check will fail and then the latest version of the server file will be fetched to client cache. However, already open instances of the file on other clients are not modified, because read and writes on already open instances of the file will interact only with the cache copy without updating it using the latest version of the server file.

- **All other file operations (such as protection changes) are visible everywhere on the network immediately after the operation completes.** As part of this assignment, access control is not implemented. However, operations like unlink, mkdir, and others are forwarded directly to the server, so they are visible by the time the RPC returns to the client.

- **Multiple workstations can perform the same operation on a file concurrently.** We follow the semantics that the last one of the concurrent writes will win. So in our implementation, concurrent writes will all update the server file

on closing. Due to the fact that on closing the client will send the whole file to the server,the last one of concurrent writes will override the changes of previous ones, which accords to the "last-win" semantics.

5.2 Evaluation

We evaluate the system behavior on crashing by inserting multiple crash points into both the client and server (demo link [1]). The setup is as follows:

- *Crash points for client.* Eight crash points are totally inserted. We insert two crash points into each of the four operations: *open*, *read*, *write*, and *flush*. For operations incurring RPC calls (*open* and *flush*), two crash points are inserted before and after the call respectively; for operations incurring operations on local cache copies (*read* and *write*), two crash points are inserted before and after the local operation.

- *Crash points for server.* Four crash points are totally inserted. We insert one crash point into the processing *Stat*, *Fetch* and *Store* RPC calls. We insert two crash points into *Store* RPC call, before and after the write operation of the server file.

- *Crash duration.* We let the crash last for a fixed amount of time (e.g. 500ms). To achieve this, we input the flag of the crash type when starting the client. The client keeps running but just throws error when reaching the specified crash point. After the fixed amount of time from crashing, the flag of the crash type will be reset and then the client will run normally.

- *Expected behaviors.* For client crash, we have tests for read crash and write crash. For read crash, the second read is issued with the correctness checked. For write crash, the second write is issued and then we use a final read to check correctness. For server crash, the client will be blocked because of retrying (with user-specified maximal number of times) and the operation will succeed after the server recovers.

References

[1] Crash demo. `https://drive.google.com/file/d/18ijz6xAHsPcNQ7vuldF_6RzAjgRraPIp/view?usp=sharing`.

[2] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, et al. The design and operation of {CloudLab}. In *2019 USENIX annual technical conference (USENIX ATC 19)*, pages 1–14, 2019.

[3] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81, 1988.