AFS-Fuse Client Server

Design

Our filesystem is designed to match the POSIX semantics of the minimum necessary API functions. We implemented open(), close(), creat(), unlink(), mkdir(), rmdir(), read(), write(), pread(), pwrite(), stat(), fsync(), and truncate(). To match POSIX semantics, write()/pwrite() are not persistent on disk until fsync() returns. Unlike POSIX, close ensures that the file is persistent. Additionally, creat, unlink, mkdir, and rmdir are all persistent upon returning.

As we were implementing something along the lines of AFSv1, the basic structure is a single server process communicating with multiple client FUSE processes. Each client FUSE caches entire files on open() and only flushes on close(). Writes to the server follows *last-writer-wins* policy, so the server copy of the file will entirely contain one client's copy, not a mix. The clients only fetch a file from the server when there is a possibility it has changed more recently than the cached copy, or when no cached copy exists.

In order to keep track of when a file has been modified by either the client or on the server, each client FUSE keeps an attribute file for each file in its cache. The attribute file contains a dirty flag indicating the cached copy has been modified since opening, and the most recent modified time from the server. Each time the attribute file is modified, it is flushed to disk. It is small enough that such flushes are atomic, meaning it will always contain meaningful information.

AFS Interface

GetAttr	Fetches the file attributes.
Fetch	Fetches the file. Uses read streaming
Store	Stores the file and returns the time attributes of the file.Uses write streaming
MkDir	Creates a new directory
RmDir	Removes a directory
Create	Create a new file
Unlink	Delete a file

Opening a file

When a file is opened, FUSE will first check if the file exists on the server. If not, the client

FUSE will call the Create RPC. If this returns successfully, the client FUSE will set the modified time and dirty flag in the attribute file. If the RPC returns that the file does already exist, it is fetched from the server. This is to account for the case in which another client wrote to the file between the initial check and the Create RPC.

If the initial check indicates the file does exist on the server, the check will return the modified time from the server. If there is no cached copy, or if the modified time in the cached copy's attribute file, the client FUSE will Fetch the file from the server. Note that if there had been a client crash previously, and there isn't a new version on the server, the local version of the file that is opened will be the same as it would have been before the crash.

To Fetch, the client FUSE first calls the Fetch RPC. This returns the contents of the file and the last modified time of the file. The file contents are saved to the cache, persisted with fsync, and then the attribute file is updated. This way if there is a client crash between saving the file and updating the attribute file, the file will simply be fetched again on the next open. While inefficient, this is not incorrect. However, there will never be a case where the attribute file indicates it has a more recent version of the file than it actually does.

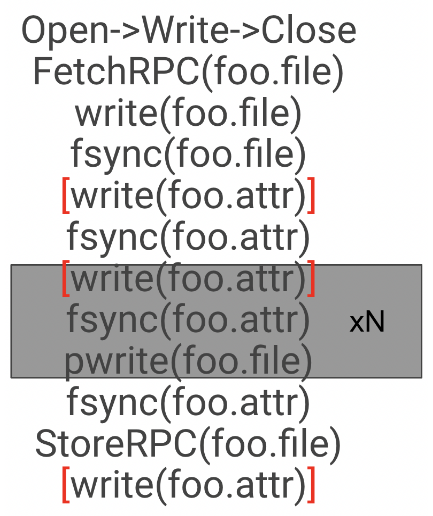
Modifying the file

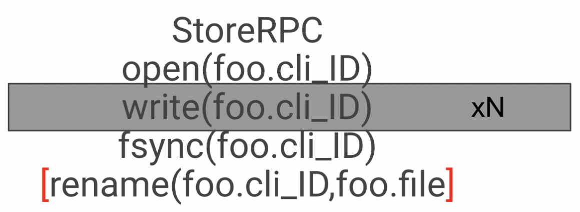
We have two ways of modifying a file, writes and truncations. In either case, the dirty flag in the attribute file is set to 1 before making the modification. This way if there is a crash, the attribute file may indicate a Store is needed when it is not (again inefficient but correct), but will never indicate a file is clean when it is dirty.

Closing a file

When a file is closed, first the dirty flag in the attribute file is checked. If it is clean, the file is simply closed. If it is dirty, the file is persisted with fsync and the Store RPC is called. On the server, the file is written to a temporary file, and renamed once writing is complete and the write is persistent on the server's disk. The name of the temporary file depends on the client, so if multiple clients were to Store concurrently, the last to finish would have their file entirely stored on the server. The server responds to Store with the modified time of the new file. This time is queried before renaming in case a Store from another client renames its temp file immediately after. When the client receives a response from the server, it updates the attribute file with the new modification time and sets the dirty flag to 0. If the client crashes before updating the attribute file, the next open will result in Fetching, which is unnecessary if the server's copy hadn't been changed, but it is not incorrect.

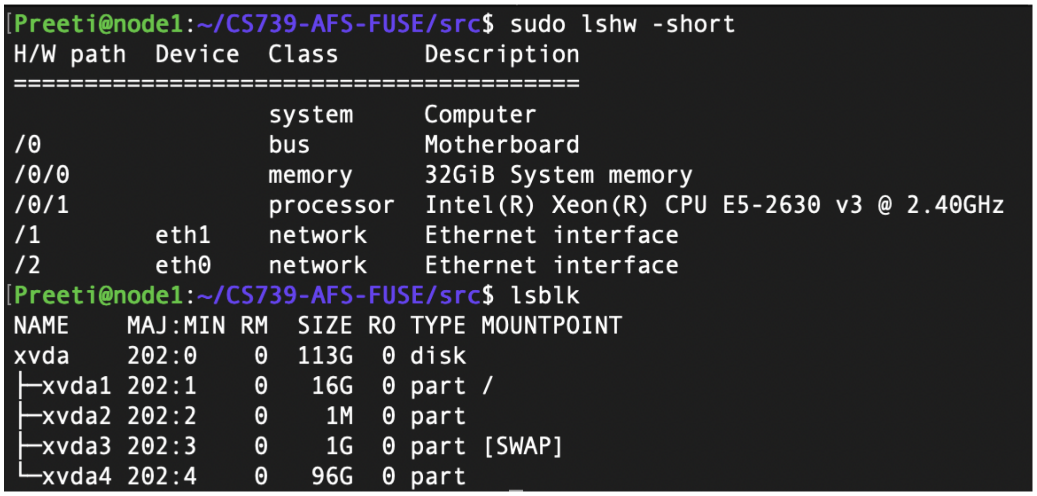
Protocol Diagrams





Hardware Platform

We performed the experiments on Cloudlab.



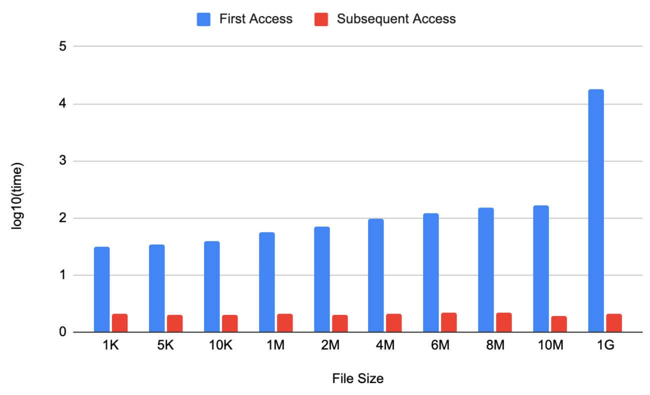
```
[Preeti@node1:~/CS739-AFS-FUSE/src$ sudo lshw -short
H/W path  Device  Class      Description
========================================
                  system     Computer
/0                bus        Motherboard
/0/0              memory     32GiB System memory
/0/1              processor  Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
/1        eth1    network    Ethernet interface
/2        eth0    network    Ethernet interface
[Preeti@node1:~/CS739-AFS-FUSE/src$ lsblk
NAME    MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
xvda    202:0    0  113G  0 disk
├─xvda1 202:1    0   16G  0 part /
├─xvda2 202:2    0    1M  0 part
├─xvda3 202:3    0    1G  0 part [SWAP]
└─xvda4 202:4    0   96G  0 part
```

Reliability Evaluation methodology

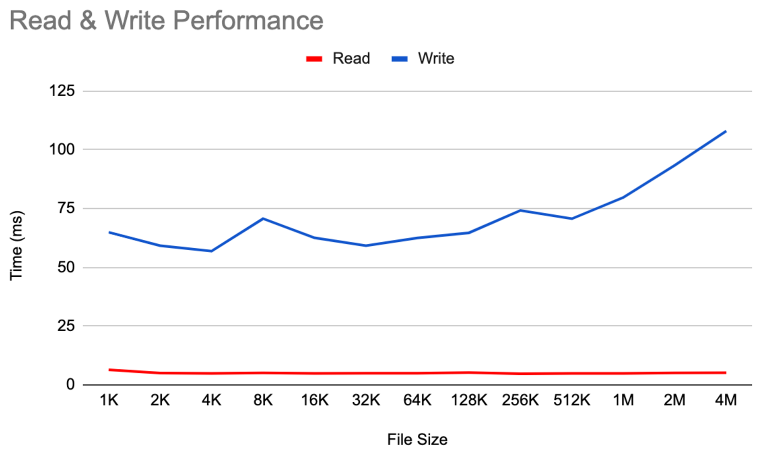
For simplicity, we performed the correctness and crash tests with a single machine performing both the roles of client and server. We added crash hooks to both the server and client FUSE process. These crashes are triggered by writing a code to a file, which the server and client processes read to decide if they should crash. Once the crash happens, the code can be cleared and the server/client can be restarted. For some client FUSE crashes, the user program accessing the filesystem must also be manually killed. See CRASH.txt for an explanation of the available crash hooks.

Performance

1. We measured the times for an `open()` call when the file is not cached (First Access) and when the file is cached and not modified on the server (Subsequent Access) by increasing the file sizes.



2. We measured the time for the sequences open(), pread(), close() and open(), pwrite(), close(). AFS stores the file if modified on close(), which increases the time as the file size increases whereas pread() just closes the file locally.



Note - The read() time doesn't increase with the increase in file size. This is probably due to kernel caching or internal caching by fuse (direct-io).

Optimizations

- If the open() flag is set to O_TRUNC, avoid fetching from the server
- Caching the attr files in the memory would help if the same file is opened again and again
- Use a Threadpool at the server to handle multiple clients concurrently
- Multiple end users at a client machine can be supported by writing the changes to a temp file (on for each user) and renaming the file on close()

- Implement a reply-cache on the server for non-idempotent operations.

Conclusion

We have implemented a distributed file system in the style of AFSv1, that generally follows POSIX semantics. We only deviated from POSIX when crash consistency demanded we persist data to disk before POSIX guarantees persistence. Thus our semantics are, in a sense, at least as strong as POSIX.

We focused our effort on providing safe crash behavior. On the server, this was easy as we did not use callbacks. On clients we needed to be a little more careful to ensure data users expect to be persistent, actually is during a client crash. Our focus on crash safety meant we did not have much time to optimize performance. While we did achieve adequate performance, our measurements exposed several places where we could have improved performance if there had been more time.