AFS (like) Distributed File System

Divyanshu Kumar Mohit Loganathan Salman Munaf

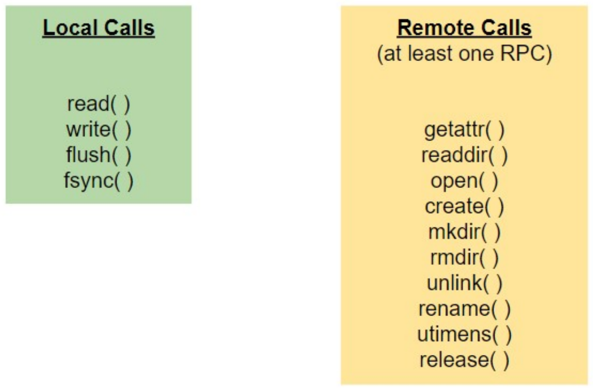
1 Introduction

This report presents the design and rationale of the AFS (like) distributed file system that we have implemented for the project. We have paid careful attention to follow AFS semantics and replicate its behavior. Crash consistency and recovery has been an important consideration in our design as we assume that clients and servers can fail anytime and we have added measures to ensure the data in file system is not corrupted. Lastly, we evaluate our implementation and demonstrate the performance and scalability of it.

2 System Design

2.1 POSIX APIs

We provide the basic set of POSIX file APIs to support file system functionality such as the ability to create, edit and delete files and directories. We have divided the file system APIs into two categories: Local and Remote calls. Local calls are handled locally by the client whereas Remote calls require atleast one RPC call to server. Figure 1 shows the list of file system APIs supported by our implementation.



Local Calls	Remote Calls (at least one RPC)
read( )	getattr( )
write( )	readdir( )
flush( )	open( )
fsync( )	create( )
	mkdir( )
	rmdir( )
	unlink( )
	rename( )
	utimens( )
	release( )

Figure 1: Supported File system APIs

In order to keep the report brief we will only explain the implementation of major file operations (code can be referred for further details). Figure 2 shows the flow of open() call. Upon file open, the client will only fetch the file from server if it is not present in the cache or if the file is modified on the server. After the file is fetched, a temporary copy of that file is created for that particular application and the file handle of that temporary file is returned to the application. The subsequent reads and writes are now performed by the applications on their cached copies. This design resolves the issue of stale cache by comparing the modification time between the files in cache and server and we are only fetching the file if it is updated on server or if it is not present in the cache. Moreover, creating a temporary file for each application allows us to support concurrent file accesses and writes by multiple applications.

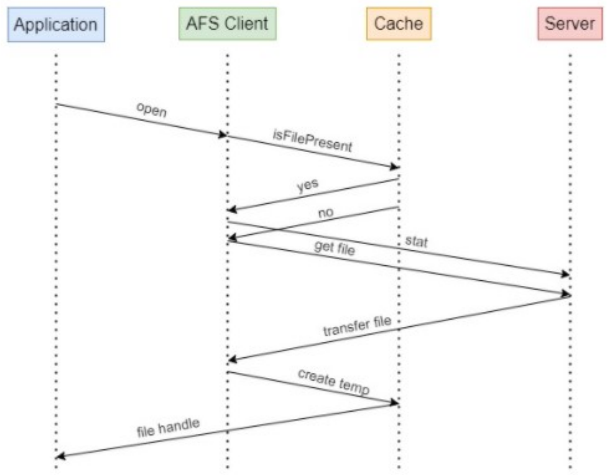


Figure 2: Flow of open() call

When a file is closed (Figure 3), the client checks to see if the file is modified at a later timestamp than server. If that is the case, we rename the file to a special recover file so that if client crashes before transferring the file at server, it can recover that file at reboot. The file is added to a shared queue and the close call is returned. Another thread takes care of transferring the file to server by polling the queue for files and sending them to server. Once the file is successfully sent to server the recovery file is removed by that thread. The idea of using a different thread to handle file transfer upon close() improves the perceived close time to the application as the file transfer takes place in the background.

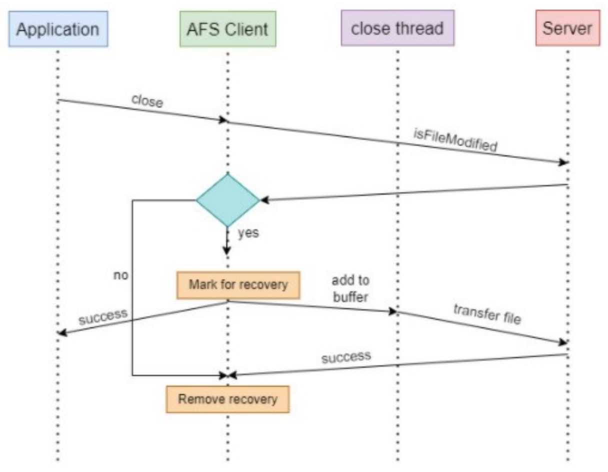


Figure 3: Flow of close() call

2.2 AFS Protocol and Semantics

The file system is based on the AFS protocol and semantics. It supports whole file caching on client side and all the read/write operations are directed to that copy. The updates are only visible to other clients when a file is closed and the updated file is flushed to server. As we are not using callbacks, we prevent stale cache by checking the modification time of file on server and fetch the file if it is changed.

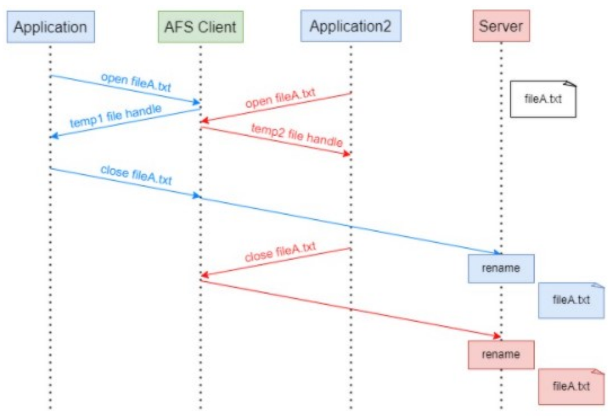


Figure 4: Last writer wins

The file system ensures that the last writer wins by using the atomic rename operation(Figure 4). The server creates a temporary file to receive data and once the transfer is complete, the file is renamed back to the original name. As the rename operation is atomic, the changes present in the last rename operation persist. The same rename operation is also used on the clients to ensure the changes of the last close persist.

2.3 Crash consistency

Our implementation takes inspiration from the protocols shared in the Alice paper to make the file system updates crash consistent. Figure 5 demonstrates that our local update protocol is crash consistent.

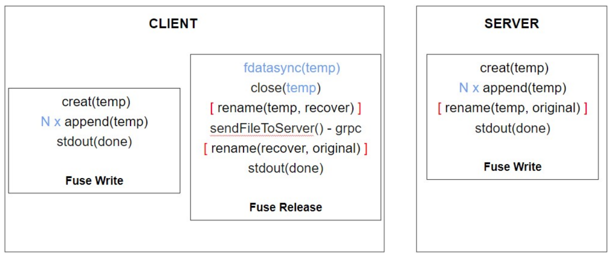


Figure 5: Crash consistent update protocol

When a file is opened a temporary file is created by the client and all the read/write operations are directed to that file. When close() is invoked, we change the name of the temporary back to its original name using the atomic rename operation. Similarly, on the server side when a file is successfully transferred by the client, the temporary file is renamed back to its original name. Hence, in our implementation the last rename operation wins.

We are using a protocol similar to the ext2 file system to maintain consistency. On initialization, the client scans the cache and checks if there are recovery (.recover) files present. It compares the file at server to see if recovery files are modified at a later timestamp than server. Lastly, it transfers all those files to the server.

2.4 Crash Recovery

There was a strong emphasis to make the file system fault tolerant and recover in case of failures.

2.4.1 Server Crash

In case of server crash (Figure 6), we have implemented a timeout retry mechanism with exponential backoff where the client retries the request until it receives a response from the server. If the operation is streaming, then the previous chunks are discarded and whole file is transferred again to the server.

2.4.2 Client Crash

Our file system supports the recovery of files on which a close() call is invoked. This is illustrated in Figure 7. When a close() call is invoked the file is persistently marked for recovery. In case of client crash, the client checks the file system on reboot if there is any recovery file. If a recovery file is present, modification time at client and server is compared and file is only sent to server if client's version of the file

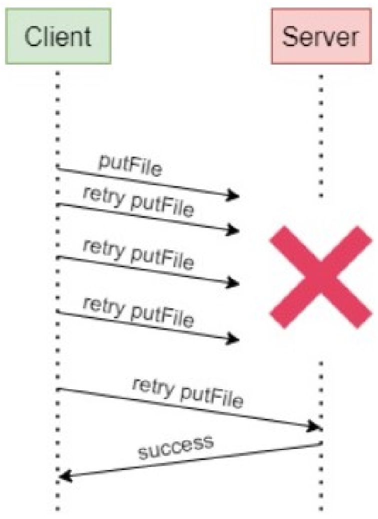


Figure 6: Server crash

is new. The file system does not support of recovery of unclosed files. This can be supported, but it is hard to distinguish between write finished files vs write unfinished files.

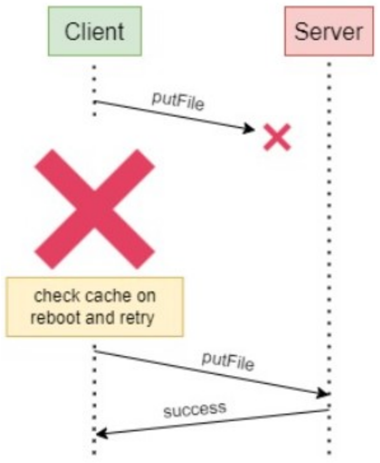


Figure 7: Client crash

3 Performance

We use the CloudLab machines to measure the performance of our distributed file system. System Configuration - 4 Node cluster :
Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz
Memory : 164941188 kB
CPU MHz: 1197.416, CPU max MHz: 3300.0000, CPU min MHz: 1200.0000, Virtualization: VT-x, L1d cache: 32K, L1i cache: 32K, L2 cache: 256K, L3 cache: 25600K

3.1 Time to complete file operations vs File Size

Figure 8 and 9 show the time it takes for various file operations with increasing file sizes. The performance on the localhost is faster compared to remote machines as there is no network delay, We can see in both the figures that the time it takes for first access (cold cache open) compared to subsequent access (warm cache open) is considerably higher as the file needs to be fetched from the server. Moreover, the time it takes to write a file is higher compared to reading a file. Lastly, the time to open and close file increases linearly with increasing file size as the data needs to be transferred between client and server.

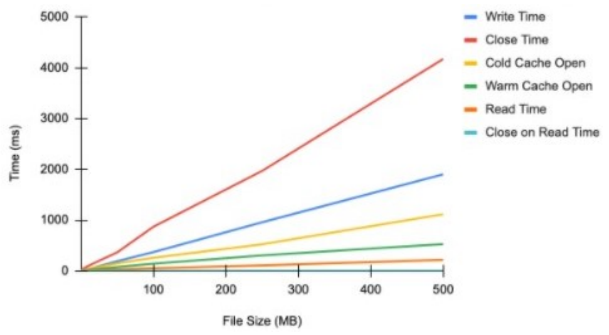


Figure 8: Performance of various operations vs File Size (Localhost)

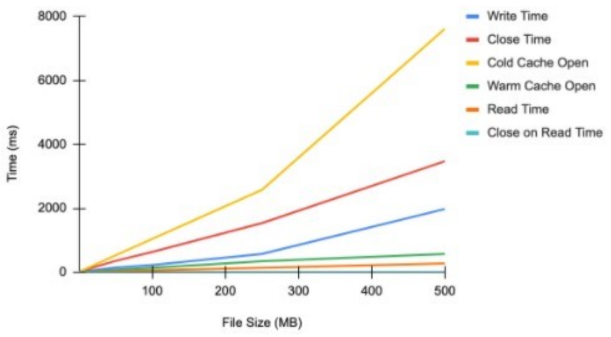
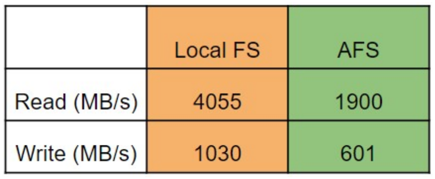


Figure 9: Performance of various operations vs File Size (Remote)

3.2 Our AFS vs Local FS(ext3)

As expected the performance of our file system is lower than the linux ext3 file system (Figure 10). However, it is not too bad in comparison when we consider the fact that our implementation is a user level process that requires significant re-routing of calls from kernel to AFS process and back. This especially appears in write case where the major bottleneck is not process level communication but probably the write overhead.



	Local FS	AFS
Read (MB/s)	4055	1900
Write (MB/s)	1030	601

Figure 10: Our AFS vs Local File System(ext3)

4 Scalability

4.1 Local File operations

Figure 11 exhibits that the time it takes for local operations on a 10MB file remain almost the same as we increase the number of applications.

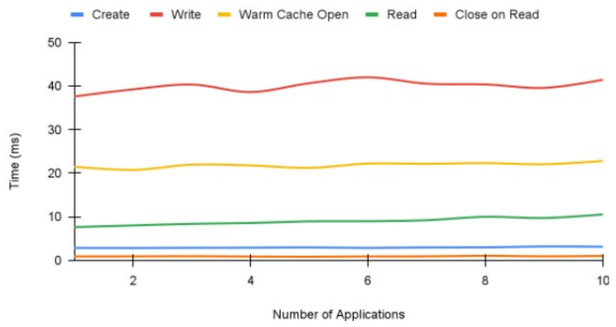


Figure 11: Scaling of local file operations

4.2 Remote File operations

Figure 12 and 13 show the time it takes for open() and close() call with increasing number of users. As the number of users increase, the performance of these operations degrade as the server becomes the bottleneck and all these operations have to go through it.

5 Improvements

We did several iterations over our base implementation to improve the performance of the file system.

5.1 Improving Open Time

In our base implementation, we observed that the first access time of file was huge as transferring the files to client over network takes time. Hence, we added the support of prefetching the files up to a certain threshold in the current directory.

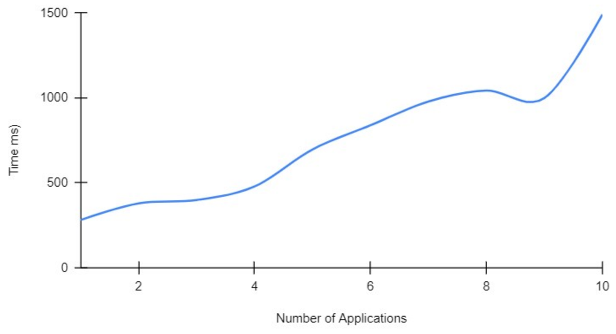


Figure 12: open() time vs Number of users

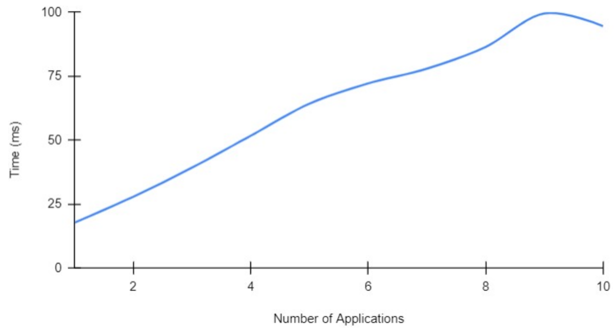


Figure 13: close() time vs Number of users

This led to a 87% improvement as demonstrated by Figure 14.

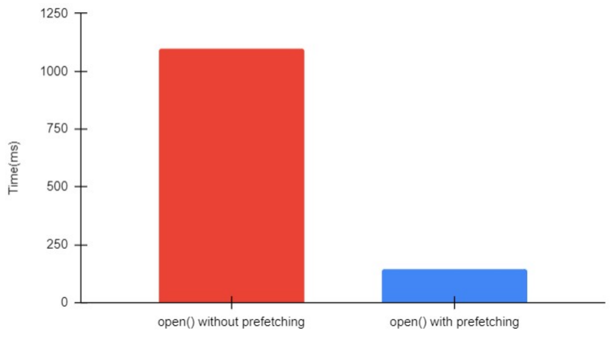


Figure 14: open() time with and without prefetching

5.2 Improving Close Time

We also observed that write buffering in the close() call was taking a lot of time as the entire file had to be flushed to server. We came up with the idea to spread fsync of write buffered data at write time. Figure 15 shows that fsync at every write is the best outcome.

However, the drawback is that it degrades the write throughput as demonstrated by the Figure 16. So we came up with a sweet spot to fsync 10% of the writes. This works for normal

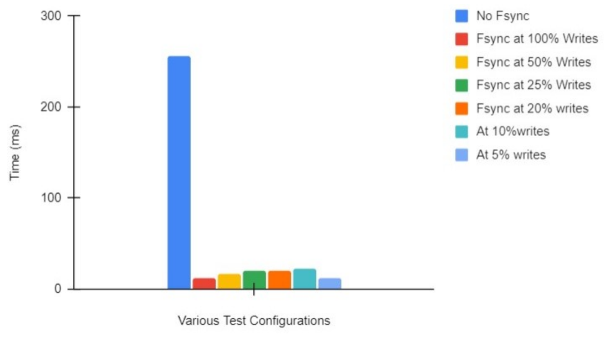


Figure 15: Average close() time as a function of fsync on writes

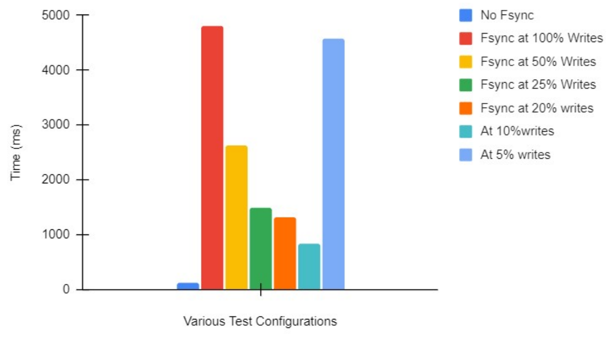


Figure 16: Average write time as a function of fsync

application as the writes are usually interspersed. This leads to a 46% improvement in time to close a file as shown in Figure 17.

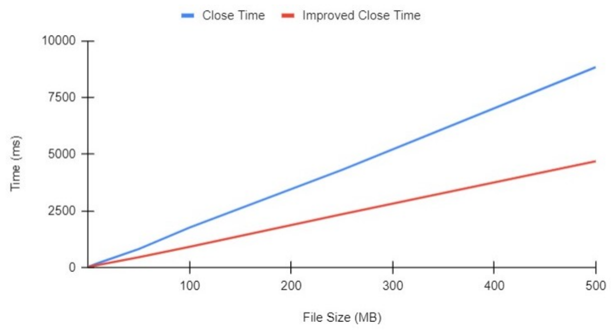


Figure 17: close() time with and without fsync on 10% of writes

Moreover, we observed that the time to close() a file was linked to the time it takes to transfer a file from client to server. Hence, we came up with an idea to use a shared queue and let other thread handle the task of transferring the files to server. This makes the close() call non-blocking as it is returned immediately and reduces the perceived close time to application. Figure 18 shows that it improved close() time by 54%.

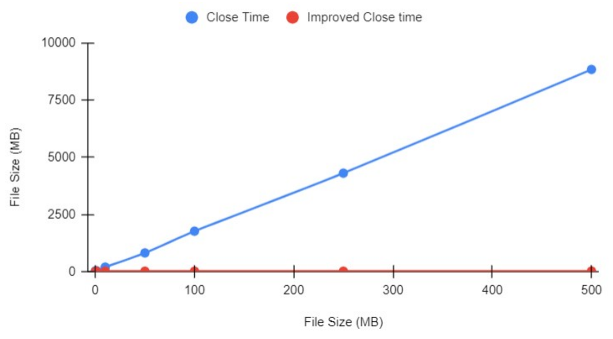


Figure 18: close() time with and without parallel close

Both of the above changes made the time it takes for the file to close independent of its file size. This is shown by Figure 19.

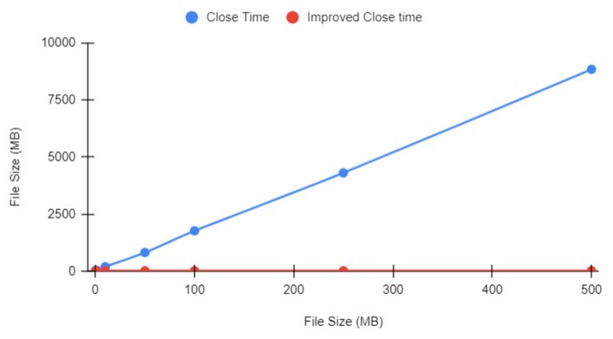


Figure 19: close() time with both fsync and parallel close

We introduced further optimizations such as only fetching and transferring files if it is modified, making temporary files to avoid intermixing of writes and streaming files in chunks of 1MB as opposed to sending complete file as one buffer. All of these optimizations improve the scalability of our file system.

6 Conclusion

In this report, we show that our design of the file system adheres to the AFS semantics. The system is fault tolerant and it recovers in case of client and server failures. It also provides crash consistency guarantees while also providing high performance and scalability. We present several insights derived from our analysis of this system.

References

[1] Howard et al. Scale and performance in a distributed file system. In *ACM Transactions cm Computer Systems, Vol. 6, No. 1*, 1988. http://pages.cs.wisc.edu/~dusseau/Classes/CS736/Papers/afs.pdf.

[2] FUSE. https://github.com/libfuse/libfuse.

[3] gRPC. https://github.com/grpc/grpc.

[4] Proto. https://developers.google.com/protocol-buffers/docs/cpptutorial.