CS739 P2: A Simple Distributed File System

Yatharth Bindal, Benita Britto, Ethan Brown, Wiley Corning

Our distributed file system is implemented using the gRPC framework to facilitate communications between a client FUSE file system and remote file server. The client supports a basic POSIX-compliant API (*open, close, creat, unlink, mkdir, rmdir, read, write, pread, pwrite, stat, readdir, release*). Our system provides reliability guarantees in line with AFS v1; beyond this, our design efforts focused on performance rather than additional reliability features.

System Design

Protocol

When a user process attempts to open a file that is not cached, our client retrieves the file's content from the server, writes it to a cache directory on the client's disk, and returns a handle to the cache file. If the file is already cached when *open* is called, the client sends its copy's *modify* timestamp to the server; if the server's timestamp differs, then the cached file is considered stale and the client will request a fresh version.

The *read* and *write* calls for a file are passed through to its local copy. When a file is closed after a *write*, the client sends this new version to be stored on the server; the server responds with the file's new *modify* timestamp, which is applied to the client's cached copy. The server applies concurrent updates in the order received, providing last-writer-wins semantics.

All other supported POSIX operations are passed to the server as RPCs; for example, a call to *readdir* will return the server's view of the directory. If the server encounters a filesystem error when handling a request, its resulting OS error code will be returned directly to the client user process (e.g. ENOENT when trying to *stat* a missing file).

`Fetch`	Reads file data from the server.	`TestAuth`	Validates a cached file.
`Store`	Stores file data on the server.	`MakeDir`	Creates a new directory.
`Remove`	Deletes a single file.	`RemoveDir`	Removes an empty directory.
`Rename`	Renames a single file.	`ReadDir`	Reads entry names in a directory.
`Mknod`	Creates a new file.	`FetchWithStream`	Streaming version of Fetch.
`GetFileStat`	Reads file metadata on the server.	`StoreWithStream`	Streaming version of Store.

Table 1: List of RPCs supported by our file system

Client crash consistency

Ext4 with data=ordered ensures persistence in the underlying file system via journaling, with atomic renames and fsyncs. This consistency guarantee does not necessarily extend to strong ordering among all system calls like write. The persistent client cache is validated against the server after a crash as files are opened. We assume file handles and dirty-file logs are invalidated by a crash and discard them on recovery.

Server crash consistency

In our server implementation, any request that modifies the persistent state is executed as an atomic sequence of operations. Most notably, the server handles a Store request by writing the new content to a

temporary file, flushing this file to disk, and renaming the file. If the server crashes during this process, the disk will remain in a consistent state upon restart. We rely on the underlying ext4 filesystem's journaling mechanism to ensure that system calls such as *rename* and *mkdir* are atomic.

The server does not return from an RPC until its effects have been persisted to disk. If the client does not receive a response, it will retry the RPC several times with an exponential backoff, eventually failing with an ETIMEDOUT error code. The server has no explicit mechanism for dealing with duplicate requests. If a non-idempotent request is executed twice, the server will typically surface a filesystem error code that will need to be interpreted by whatever user process originated the request. Our system therefore provides at-least-once semantics (within the client's overall timeout period).

The server's in-memory data structures are intended to be transient. When the server is restarted after a crash, it requires no explicit recovery mechanism to resume normal operations.

Server concurrency behavior

Our server uses a per-file locking mechanism to maintain safety while processing concurrent requests. The server maintains an in-memory map that stores a separate mutex for each file path; a single, global mutex guards access to this map as mutexes are created upon first access. When the server handles a request that will modify the state of a file, it first waits to acquire a unique lock on its mutex. Requests that read file state will instead acquire a shared lock. We therefore allow simultaneous reading of a file, but prevent simultaneous writing (which might result in an inconsistent state) and reading (which might return inconsistent data) while a write operation is ongoing.

The mutex data structure is "flat" in the sense that locking a file does not lock any of its parent directories. Because we permit *rmdir* only for empty directories and do not support renaming directories, this flat structure does not result in a safety risk.

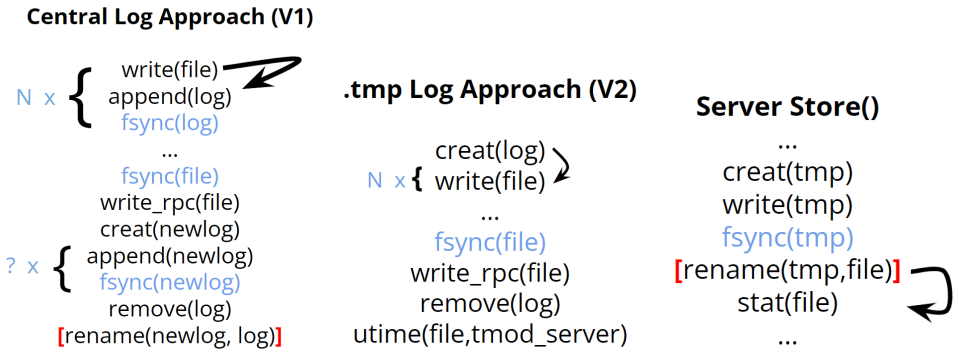


Fig: Update-protocol diagrams for the client (under each log approach) and for the server. As in the ALICE paper, arrows indicate an ordering dependency.

Performance Improvements

Our client uses a dirty-file log to track whether files are modified between *open* and *close*, allowing unmodified files to be closed without contacting the server. We developed two alternative implementations of this log: a single-log approach which records dirty file paths as entries in a central log file, and a multiple-log approach which creates .tmp log files for each dirty file. In the single-log version, a file's path is appended to the central log whenever it is written locally; these lines are cleared only when the client closes the file and commits it to the server. In the multiple-log version, a .tmp file is created on *write* to

indicate a file is dirty and then deleted when the corresponding file is committed to the server. Both approaches clear the log on client restart, and any modified but uncommitted files will be overwritten with the server version.
Our final design for evaluation uses the multiple-log implementation due to the possibility of crash-safety problems in single-log updates.

We used an in-memory cache on the server to hold recently accessed files of sizes less than 1MB, allowing the server to handle *Fetch* requests for these files without performing a more-expensive disk read. We also implemented an additional client mode that uses gRPC streaming to perform *Fetch* and *Store* operations; we found that this mode was necessary to support large files (e.g. 16MB or higher).

Evaluation

Methodology

We evaluated our file system's performance and reliability using a testing environment of two Azure cloud VMs, configured with the same specifications and using the same dedicated client and server VM across all runs. The VMs were each configured with Ubuntu 20.04, 2 vcpus, 4 GB memory, and used SSDs rather than HDDs to improve underlying disk performance.

For complex benchmark scripts, we collected rough end-to-end timing measurements using the Linux *date* command. We used the *std::chrono* monotonic clock in our client and server code to collect more precise measurements of individual components of our system.

To evaluate reliability, we triggered controlled client and server crashes by dereferencing a null pointer and causing a simple segmentation fault. Crash points were inserted into our code via a macro function and controlled with a mixture of per-crash function arguments and overall global variables set at compile time. The resulting crashes terminate either the server or client processes, with client crashes making the FUSE file system inoperable until unmounted and reinitialized.

We developed a test suite to measure the performance of basic read and write operations and to demonstrate our consistency guarantees. In our read-focused benchmark, we *cat* a file from the network directory to */dev/null*; in the *write*-focused benchmark, we copy a file into the network directory. These tests were conducted for two system versions (a "basic" version and a "performance" version) and for five different file sizes (64B, 4KB, 256KB, 16MB, and 1GB).

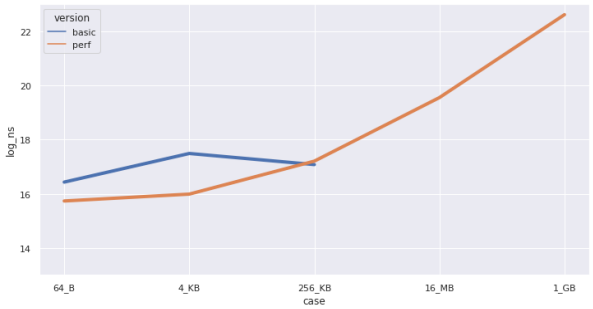
Due to limitations in gRPC outgoing packet size, we found that our 16MB and 1GB files could not be transferred by the "basic" version of our file system (which does not utilize streaming). The "performance" version, which enables streaming, is able to perform these transfers.

Under each combination of the above factors, we took measurements over 100 iterations of our test script (except for file sizes of 1GB) and evaluated the minimum, maximum, average, median, and standard deviation of their durations. In the 1GB condition, we performed 5 iterations of the read benchmark (due to its long duration) and 3 iterations of the write benchmark (as processes that ran with higher iterations were killed by the system).

We evaluated the performance of our client dirty-log implementations over 100 iterations and across several log sizes (explained further below), and collected the same statistics.

Results on Client-side Caching

In the read-focused benchmark, we observe that the first call to *open* takes significantly longer than subsequent calls, since the first call must *Fetch* the file content while subsequent calls can utilize the cache. The calls to *open* in our write-focused benchmark We also observe a roughly linear correlation



between file size and benchmark duration.

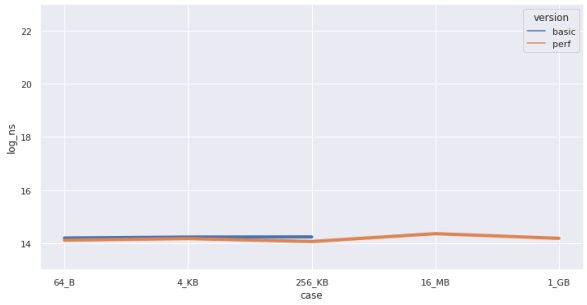


Fig: Log of time taken for first open()

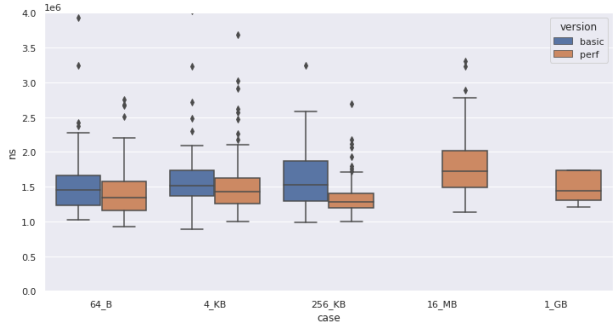
(Test case: cat file to /dev/null)

Fig: Log of time taken for median open()

(Test case: cat file to /dev/null)

Impact of Optimizations

For our read-focused benchmark, we found that the “performance” version outperformed the “basic” version by a slight margin (as seen for small file sizes) on *open* calls. We notice this difference when the server in-memory cache is warm. For the *close* / *release* operation, the “performance” version was slightly outperformed by the “basic” version in cases where both were applicable, possibly due to the additional bookkeeping overhead required for streaming.



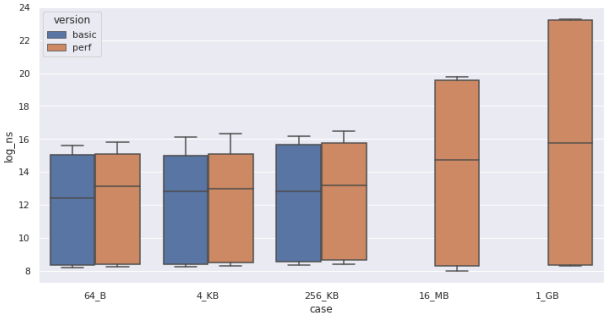


Fig. Time taken for open()

(test case: cat file to /dev/null)

Fig. Log of time taken for release()

(test case: copy file into network folder)

Performance of *read* and *write* calls

In the

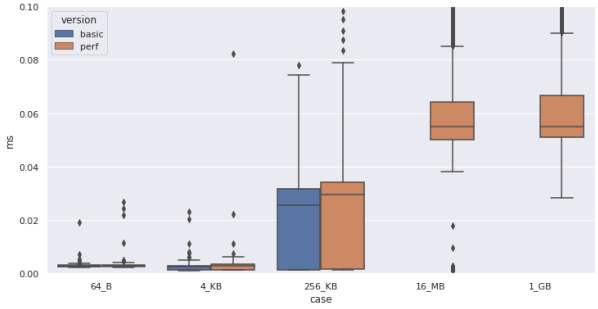


Fig. Time taken for read()

(test case: cat file to /dev/null)

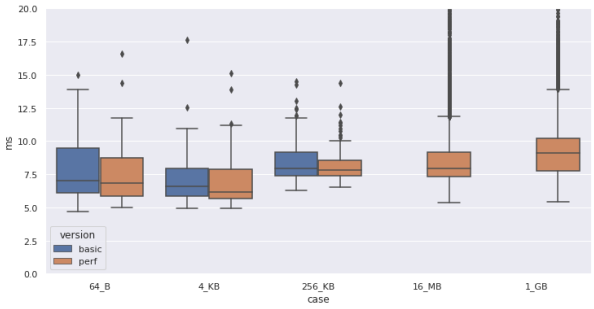
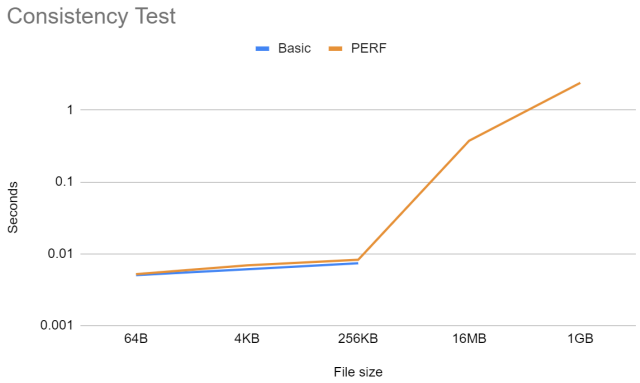


Fig. Time taken for write()

(test case: copy file into network folder)

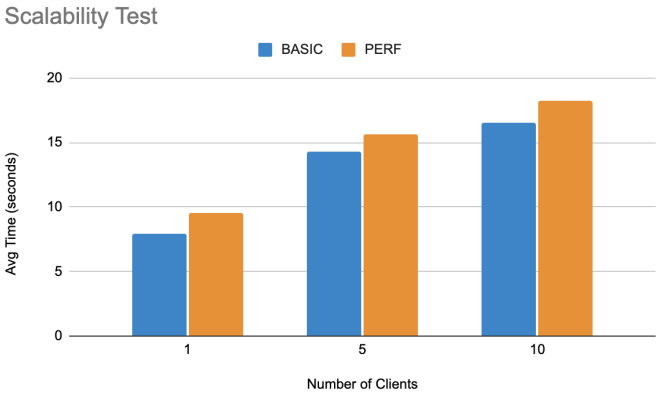
Consistency

We tested consistency by invoking the *cat* command on the client and modifying the timestamp of the same file on the server in parallel. This causes the client's *TestAuth* call to invalidate its cache, requiring it to *Fetch* the file from the server. This causes the read to take longer than if there was no modification to the file (as seen in read fig for read vs write performance).



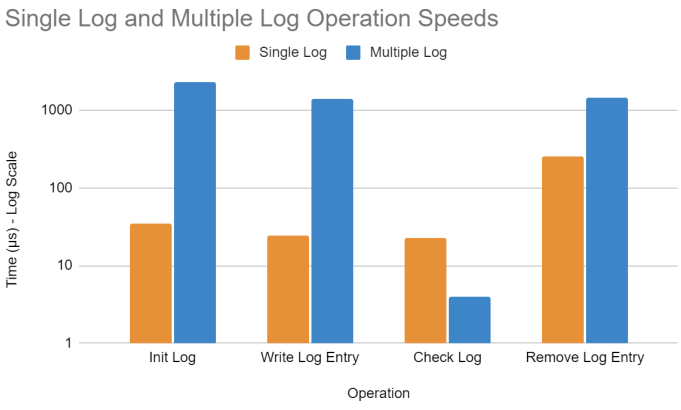
Scalability

We ran a test to copy 4 source code files of sizes ~5KB and ran make for a total of 100 iterations. Clients were spawned as different processes on the same machine due to lack of resources. Our performant system does not outperform the basic system as there is an overhead of adding files to our in-memory map for very small files.



Client Cache Log Comparison

We compared our two approaches to tracking dirty-file status by timing different log operations for each approach which are performed as part of client file system calls. The most significant results are summarized in the graph below. Operations were run on pre-existing logs with 100 entries, and log files were flushed from the underlying file system cache to isolate results across experiment iterations.
We found that the single-log approach outperformed the multiple-log .tmp file approach for the operations of clearing the logs on init,



writing a single log entry, and removing all log entries for a file that is committed to the server.
We hypothesize that most of the overhead in the multiple-log approach comes from suboptimal .tmp file creation and deletion (particularly dealing with paths for recursive subdirectories), and that it could be improved in future work. We also found that the operation to check the log and see if a file had been modified performed faster under the multiple-log approach, and thus the frequency of file writes or closes will impact which approach performs best for different workloads.
We also timed operations on empty logs, logs with 50 entries, logs with 100 entries, and logs with 100 entries which did not contain the specified file entry in the cases of checking file modification and deleting an entry. There was no statistically significant difference between the various operation speeds of either log system when compared across different log sizes. In workloads with thousands of log entries or frequent enough log accesses to cache the logs in memory, the multiple-log and single-log approaches may scale better respectively.

Server and Client Crash Experiments

We verified our client and server safely recovered from crashes at several points during critical function calls, and that file data on either end could be safely restored to a current or recent stable state. For our live demonstration we triggered crashes in the client open and close functions; we confirmed that a client was both able to continue using a valid cached file after recovering from a crash near the end of open, and able to retrieve up-to-date data from the server following a crash during close without causing corrupted files or committing incomplete data to the server. We also triggered a crash during the server write to disk function, and confirmed that the restarted server would serve the modified file after restarting and that the waiting client could properly retry and eventually continue uninterrupted operation during a server crash.

Future work

Our dirty-file log can be extended to create a similar close-file log, which would create log entries upon beginning a client fsync or close call and remove entries upon successfully committing data to the server. Then, after a client crash and restart, all files recorded in the close-log could be flushed to the server during client init using a modified client close call; this would ensure that any file changes that were written and intentionally committed locally by the user will still be communicated to the server after crash recovery. We partially implemented a prototype for the close-log design, but encountered crash consistency issues with log management which prevented enabling it in the final prototype.

An asynchronous implementation of the server would yield performance improvements in multi-client scenarios. Our server-side file-locking mechanism could also be improved by allowing concurrent *Store* requests for the same file to write different temp files in parallel, acquiring a unique lock only for the duration of the *rewrite* call; thanks to Remzi for suggesting this. We could also improve reliability by implementing a replicated configuration for our server; a simple primary-backup mechanism would reduce downtime during crash recovery and provide at least some protection against disk hardware faults.