CS739 P2: A Simple Distributed File System

Yatharth Bindal, Benita Britto,
Ethan Brown, Wiley Corning

Section 1: System Design

System Overview

- Server-client communication over GRPC
- POSIX API Compliance
  - open(), close(), creat(), unlink(), mkdir(), rmdir(), read(), write(), pread(), pwrite(), stat(), readdir(), release()
- Clients check local cached files against the server during open()
  - If file not found locally or server's timestamp doesn't match, fetch server's version
  - Local cached file is then consistent with server before any read() or write()
  - Whole-file caching

Client Behavior

- Reads and writes are performed locally and don’t require RPC calls
- Clients flush modified file data to the server during close()
  - Last write request to the server wins, enforced by GRPC request queuing and server file locking
  - Other clients update their caches when they open the file, or write their own data if they close an already-open version
- Client dirty-file logs: One central log vs multiple .tmp file logs
  - The central log approach records file paths of locally-modified files during writes, and deletes entries when files are committed to server during close
  - The .tmp log approach records local writes in .tmp files and deletes them on close
  - No need to send file data to server on close if the file wasn’t modified

Server RPCs

`Fetch`	Reads file data from the server.
`Store`	Stores file data on the server, returning the server's timestamp.
`Remove`	Deletes a single file.
`Rename`	Renames a single file.
`Mknod`	Creates a new file with the specified name.
`GetFileStat`	Reads the server's `stat` information for a single file.
`TestAuth`	Compares client and server modify times for a single file.
`MakeDir`	Creates a new directory with the specified name.
`RemoveDir`	Removes a specified directory if it is empty.
`ReadDir`	Reads a list of entry names present on the server in a given directory.
`FetchWithStream`	Streaming version of Fetch.
`StoreWithStream`	Streaming version of Store.

Crash Consistency

- Underlying Linux file system: ext4 with data=ordered
  - Crash consistency via journal; atomic rename and fsync
  - Does *not* guarantee strong ordering among all system calls, e.g. write and rename
- Persistent client cache validated against server timestamps
  - Server's modify_time is returned from Store, written to client file metadata
  - TestAuth validates cache file if modify time matches server's
- Server persistence
  - Atomic Store: write to temp file and rename
  - Per-file read/write locks prevent mixed or incomplete data
  - Always commit changes before replying

Client Crash Recovery

- Client cache isn't invalidated on crash/reboot
  - Client revalidates cache files as-needed with open
- Assume a crash permanently disconnects all open file handles
  - Purge dirty-file logs in the cache
- Future Improvement: client close-logs track which files have been flushed but not known to be committed to the server
  - Extension of dirty-file log approaches
  - Entries recorded to close-log on fsync() and initiating close()
  - Entries removed from close-log upon successful close() and response from server
  - On client init, commit all pending local files from the close-log to the server

Server Crash Recovery

- Server implements atomic file system interactions
  - Crash cannot result in inconsistent server state
  - If request is not completed, we expect the client to retry it
- Clients resend requests with exponential backoff
  - Eventually returns an error code after timeout
  - Server does not persist client information
- If a server fails to execute a filesystem call, return the error code to client
  - E.g., trying to stat a missing file returns ENOENT
  - Client should use error codes to infer when a non-idempotent request has succeeded

Optimizations

- gRPC Streaming
  - Necessary for large-file transactions
- Server memory mapping
  - Hold recently accessed files in memory for faster Fetch() response

Update Protocol Diagrams

Central Log Approach (V1)

N x { write(file)
append(log)
fsync(log) }
...
fsync(file)
write_rpc(file)
creat(newlog)
? x { append(newlog)
fsync(newlog) }
remove(log)
[rename(newlog, log)]

.tmp Log Approach (V2)

N x { creat(log)
write(file) }
...
fsync(file)
write_rpc(file)
remove(log)
utime(file,tmod_server)

Server Store()

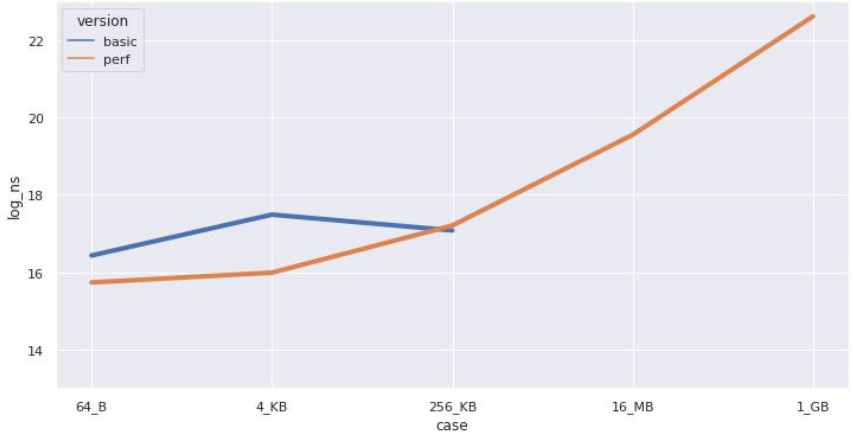
...
creat(tmp)
write(tmp)
fsync(tmp)
[rename(tmp,file)]
stat(file)
...

Section 2: Results

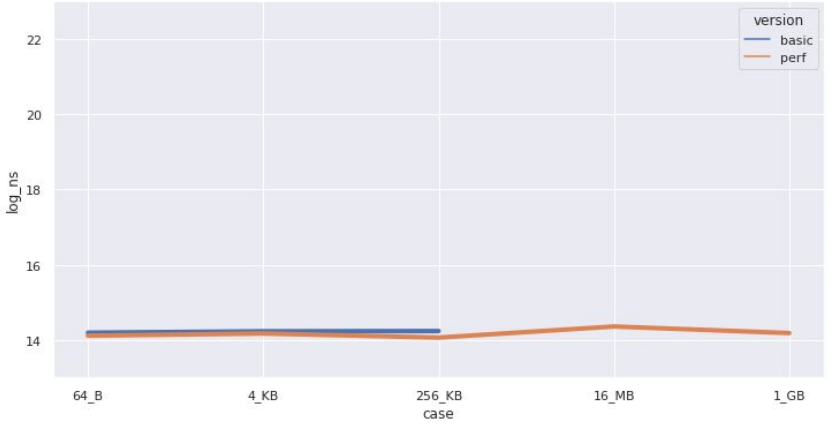
Methodology

- Testing environment: Azure VMs for server and client
  - 1 VM for server, 1 VM with multiple client processes & directories
  - Ubuntu 20.04, 2 vcpus, 4 GB memory, SSD
- Performance experiments run with 100 trials
- Timing Mechanisms
  - Date command in bash for timing overall workloads
  - std::chrono timing used for client/server functions
- Simulated crashes using dereference to null pointer

Performance: Impact of Client-side Caching

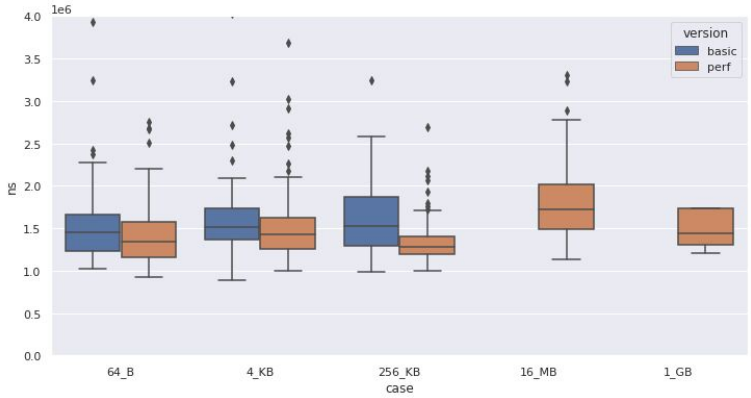


Log of time taken for **first** open()
(test case: cat file to /dev/null)

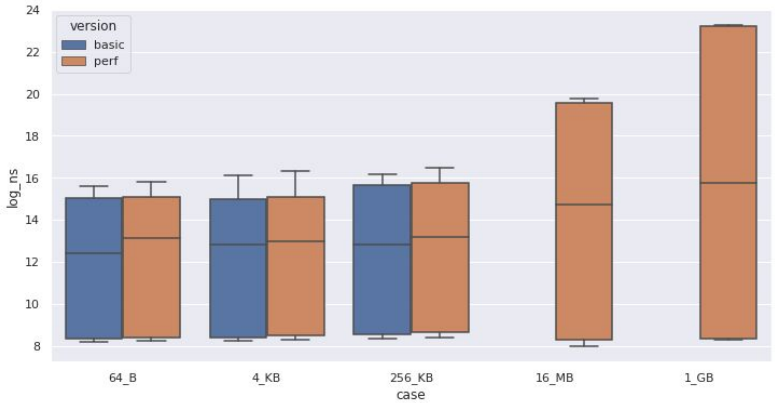


Log of time taken for **median** open()
(test case: cat file to /dev/null)

Performance: Impact of Optimizations

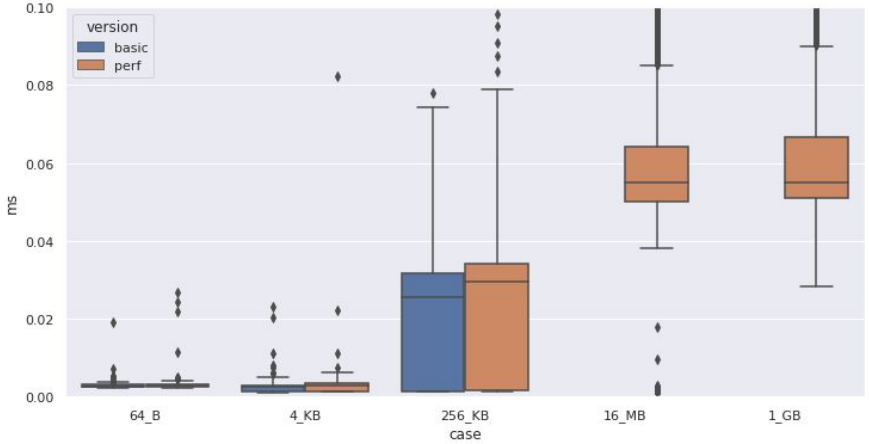


Time taken for open()
(test case: cat file to /dev/null)

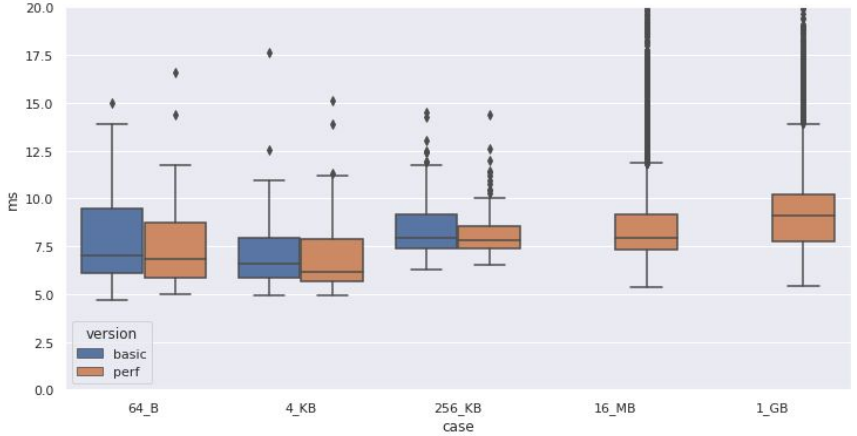


Log of time taken for release()
(test case: copy file into network folder)

Performance: Read and Write



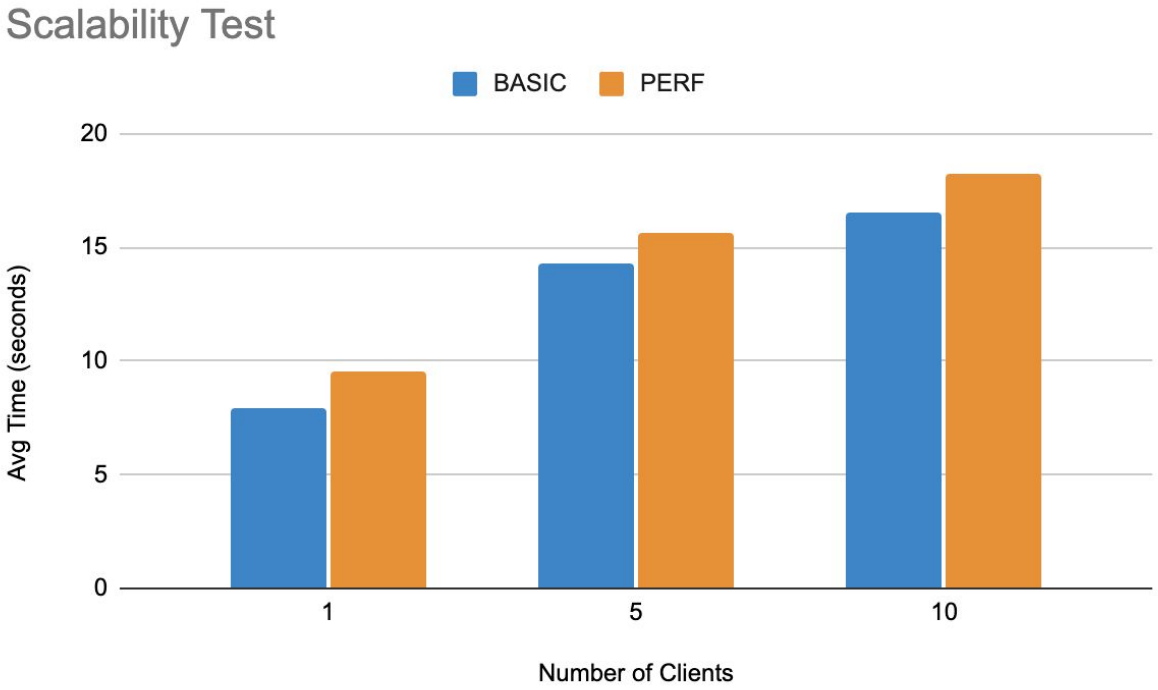
Time taken for read()
(test case: cat file to /dev/null)



Time taken for write()
(test case: copy file into network folder)

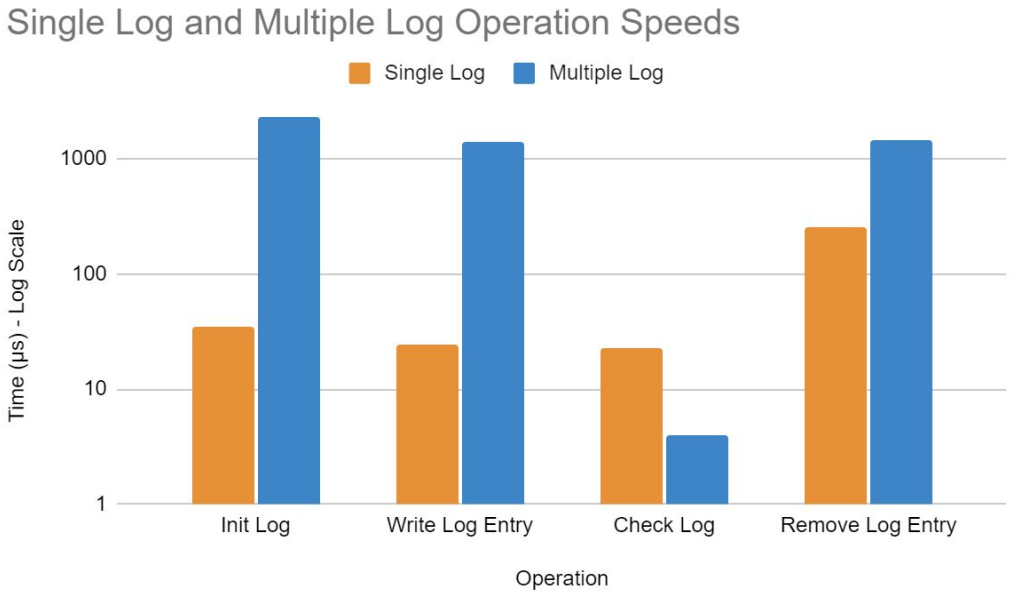
Performance: Scale

- Test case: copy 4 source code files (size in KB); make



Client Dirty-File Logs

- .tmp file operations for multiple-logs have higher overhead than single-log, except for checking files modified
- No significant difference in either approach between increasing # of log entries



(Tests run with 100 log entries)

Client Reliability (20 pts)

- Recovery from simulated crashes at various FUSE functions/RPC calls
  - Demo
- Local cache invalidated after client crash
  - Re-fetches file data from server because modified timestamp changed during writes
  - Resumes normal operation using last known stable data reflected on the server

Server Reliability (20 pts)

- Recovery from simulated crashes
  - Demo

- Points of danger?
  - Crash while writing file: handled by temp file mechanism
  - Crash while responding to FETCH: handled by GRPC
  - Other thread interacting with a file while it is being written: handled by locks
  - Crash before responding to non-idempotent request: not explicitly handled
    - Upon retry, request will fail and return a filesystem error