P2: Client/Server File System

Akshay Parashar
Hemal Patel
Kalpit Munot
Reetuparna Mukherjee

Protocol Semantics

POSIX Implementation

Local only calls:

- read()
- write()
- utimens()
- flush()

Local and remote calls:

- getattr()
- readdir()
- mkdir()
- rmdir()
- creat()
- unlink()
- open()
- release()

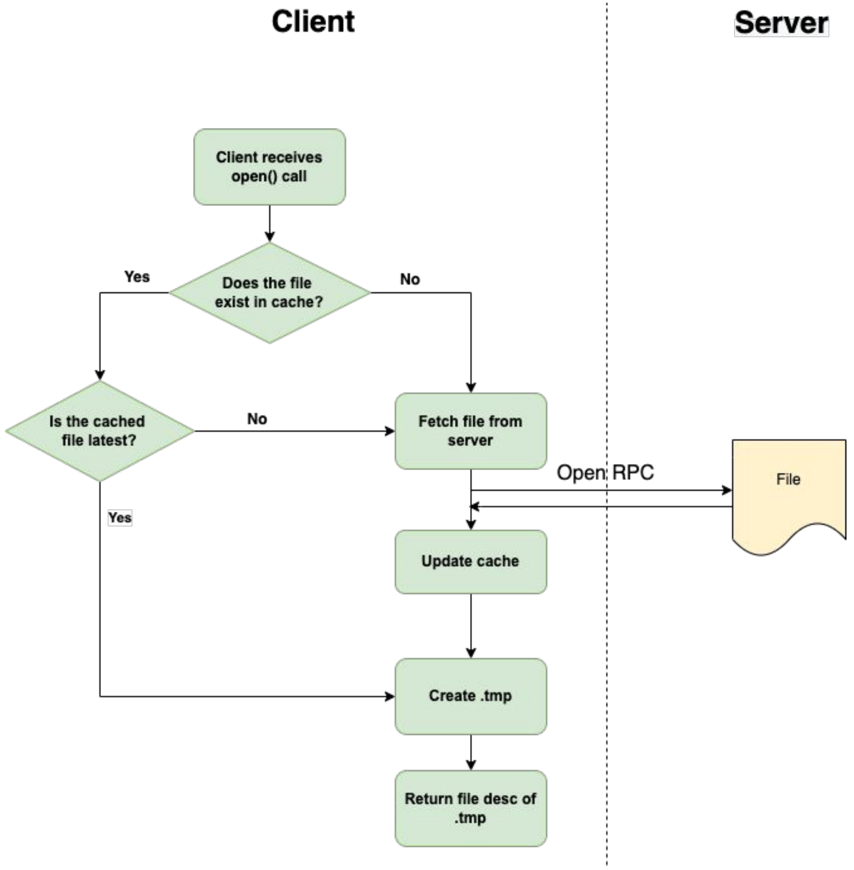
Terminology and Protocol Assumptions

- **Nomenclature:**
  - **.tmp file:** Dirty files that have not been closed by the application process
  - **.consistent file:** files that are closed by application process and are safe to be flushed on client/server

- **Assumptions**:
  - Unless close() API is called, data is non-recoverable (dirty).
  - After close() is called and if server or client crashes, the data needs to be recovered by crash consistency protocol and updated on both the server and client
  - "Last Writer Wins" on both and Server and Client.

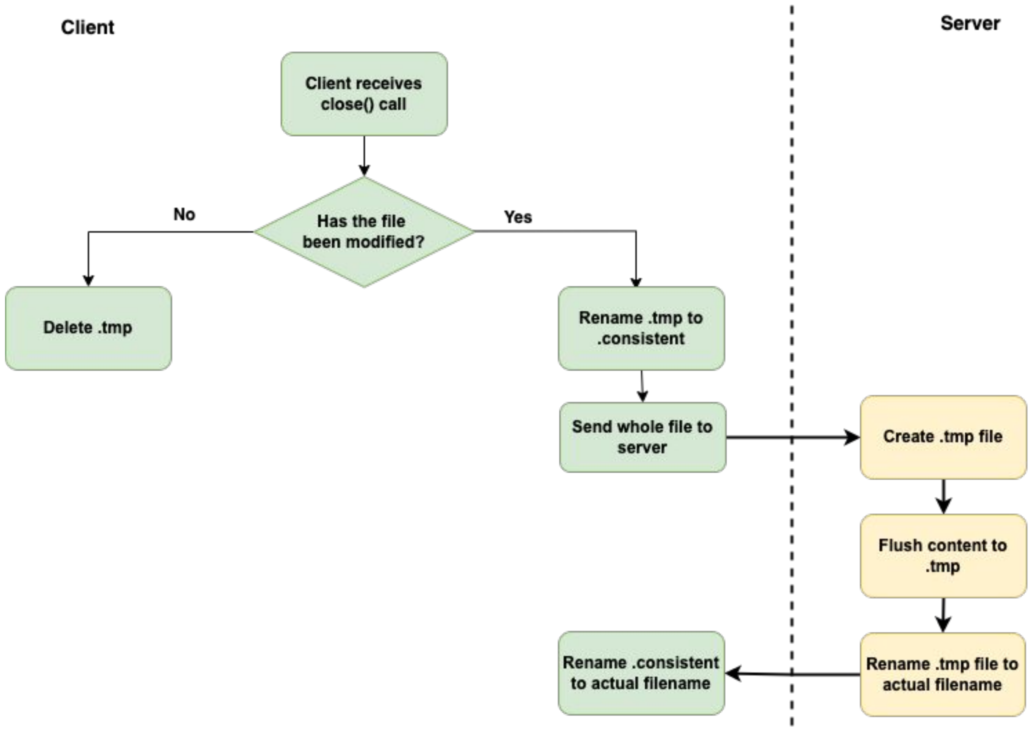
Open

- Client fetches the file on demand from the server iff
    - file does not exist on cache (or)
    - cached file is stale
- On fetching the file from server, client updates its cache
- Each application process is given its own .tmp file (i.e. copy of the cached file) descriptor to work on



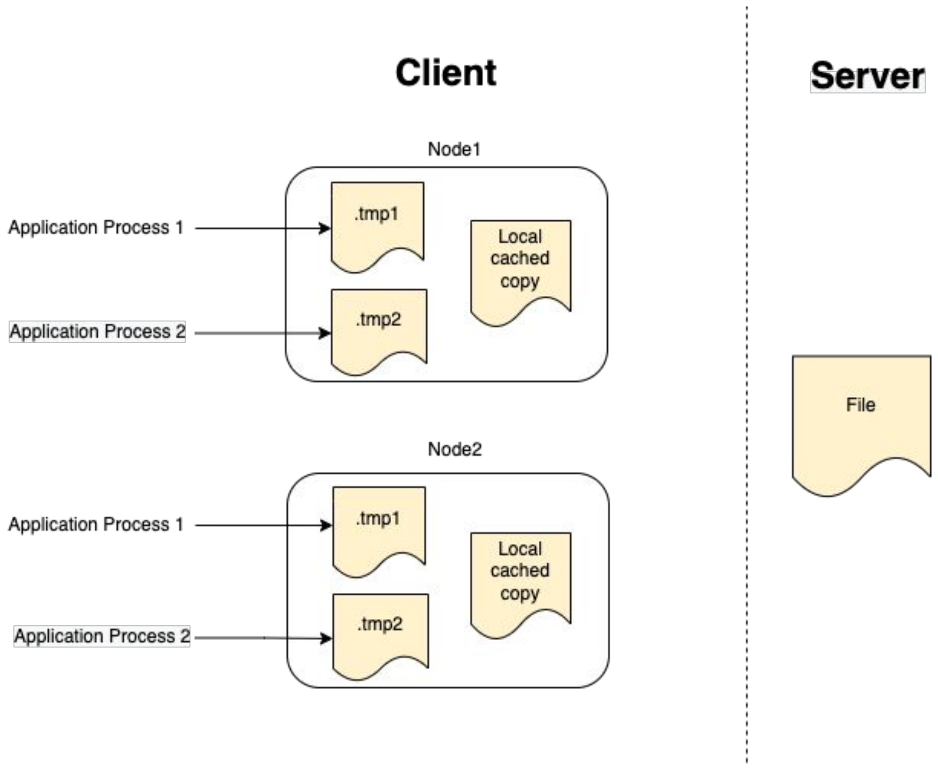
Close

- Fuse-client checks for changes in .tmp file.
- .tmp file is deleted if there are no changes,
- Else, .tmp file state is changed to .consistent file (renamed)
- File is then sent to the server
- Server creates a .tmp for every close op, and then atomically renames it - LAST WRITE WINS
- After sending the file, .consistent file is used to create the cached copy for that node


Client
Server
Client receives close() call
Has the file been modified?
No
Yes
Delete .tmp
Rename .tmp to .consistent
Send whole file to server
Create .tmp file
Flush content to .tmp
Rename .tmp file to actual filename
Rename .consistent to actual filename

Read/Write

- Read/ Write are local to each client
- Each application process performs read/write on its own copy of cached file (.tmp file created during open)
- No server interaction involved

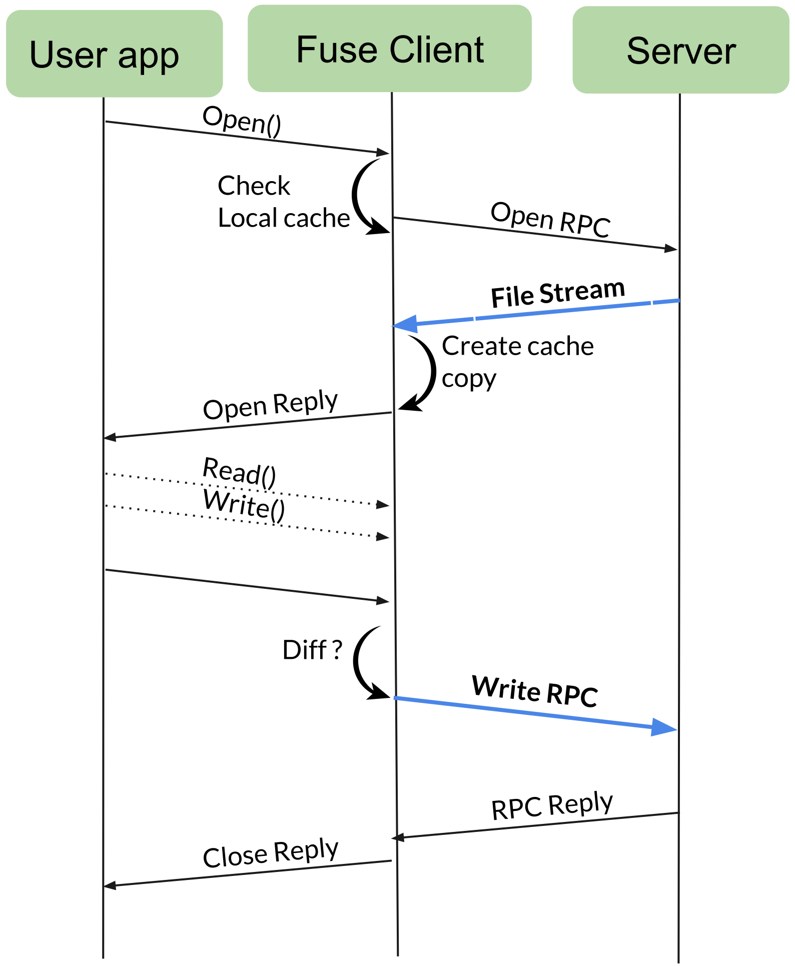


Durability and Reliability

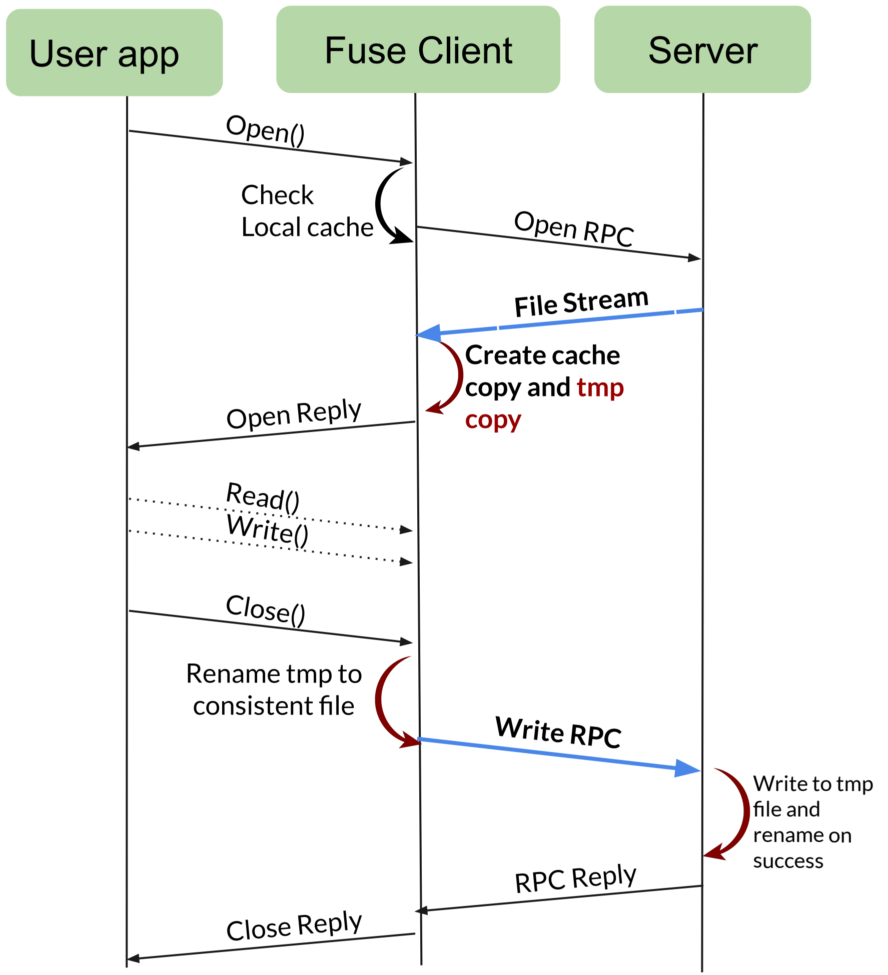
Durability: Crash Consistency Protocol

- Each application process per client creates a **.tmp** copy for their read/writes
- If the client/server crashes before application process could call close, the tmp file is dirty and is deleted during **garbage collection**
- Renaming .tmp file to **.consistent** file on close saves state of the file from dirty to consistent. It is an **atomic** operation, hence safe
- On restart, client process deletes all the .tmp files and recovers .consistent files to local cache and server.

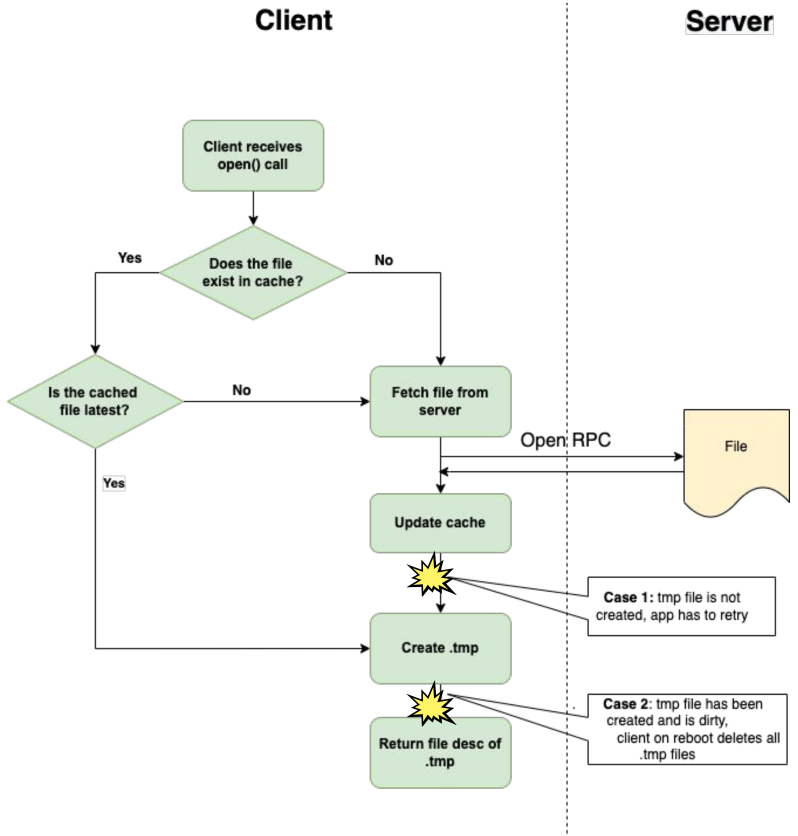
W/o crash and concurrency



Our Implementation

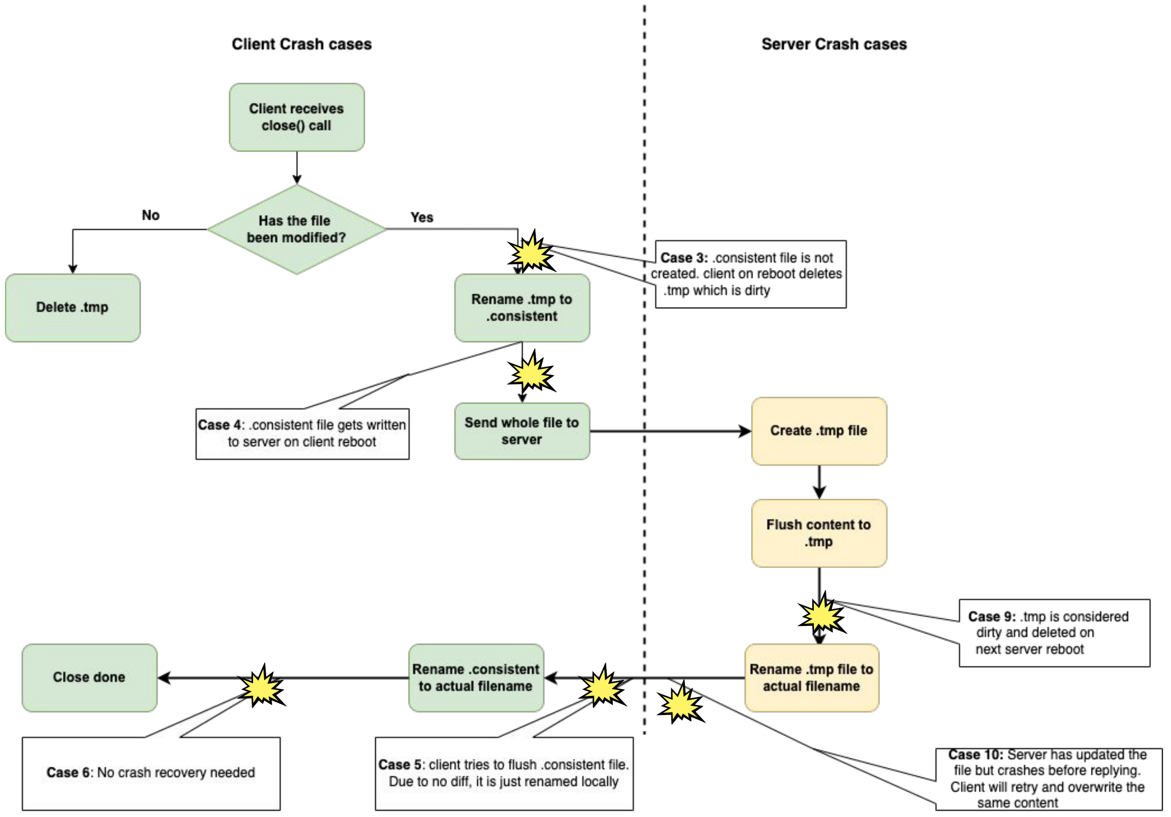


Crash Recovery Protocol, Open


Client
Server
Client receives open() call
Does the file exist in cache?
Yes
No
Is the cached file latest?
No
Yes
Fetch file from server
Open RPC
File
Update cache
Case 1: tmp file is not created, app has to retry
Create .tmp
Case 2: tmp file has been created and is dirty, client on reboot deletes all .tmp files
Return file desc of .tmp

–

Crash Recovery Protocol, Close


Client Crash cases
Server Crash cases
Client receives close() call
Has the file been modified?
No
Yes
Delete .tmp
Rename .tmp to .consistent
Case 3: .consistent file is not created. client on reboot deletes .tmp which is dirty
Case 4: .consistent file gets written to server on client reboot
Send whole file to server
Create .tmp file
Flush content to .tmp
Case 9: .tmp is considered dirty and deleted on next server reboot
Close done
Rename .consistent to actual filename
Rename .tmp file to actual filename
Case 6: No crash recovery needed
Case 5: client tries to flush .consistent file. Due to no diff, it is just renamed locally
Case 10: Server has updated the file but crashes before replying. Client will retry and overwrite the same content

Performance

Setup

Workload: File size ranges from 2KB to 1024 MB
Processes count: 2,4,8,16

Hardware Configuration:

- Cloud Provider: MSFT Azure
- OS: Linux (ubuntu 20.04)
- vCPU: 2
- RAM: 4GB

Cases

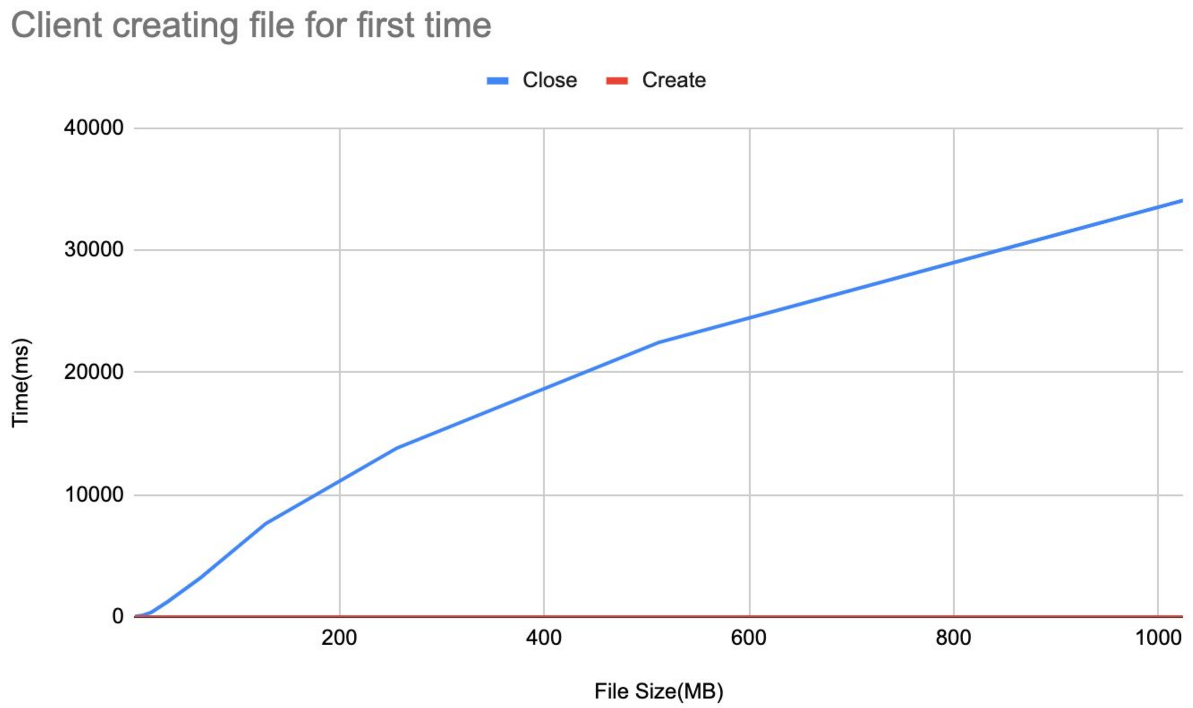
Number	Case description		Calls involved
1	Client creates the file for the first time		creat(), write(), **close()**
2	First vs Subsequent access - Client fetches file that existed in the server		**1st open()**, close() 2nd open(), close()
3	Write vs Read cost		**read(), write()**
4	Scalability (open() and close() cost wrt multiple client processes on the **same** machine)	(i) Client creates the file for the first time	open(), **close()**
		(ii) Client fetches file that existed in the server	**open(),** close()
5	Experimental (open() and close() cost wrt gRPC packet size)	(i) Client creates the file for the first time	open(), **close()**
		(ii) Client fetches file that existed in the server	**open(),** close()

***Costly operations highlighted in bold**

Case - 1: Client creates the file for the first time

Observations:

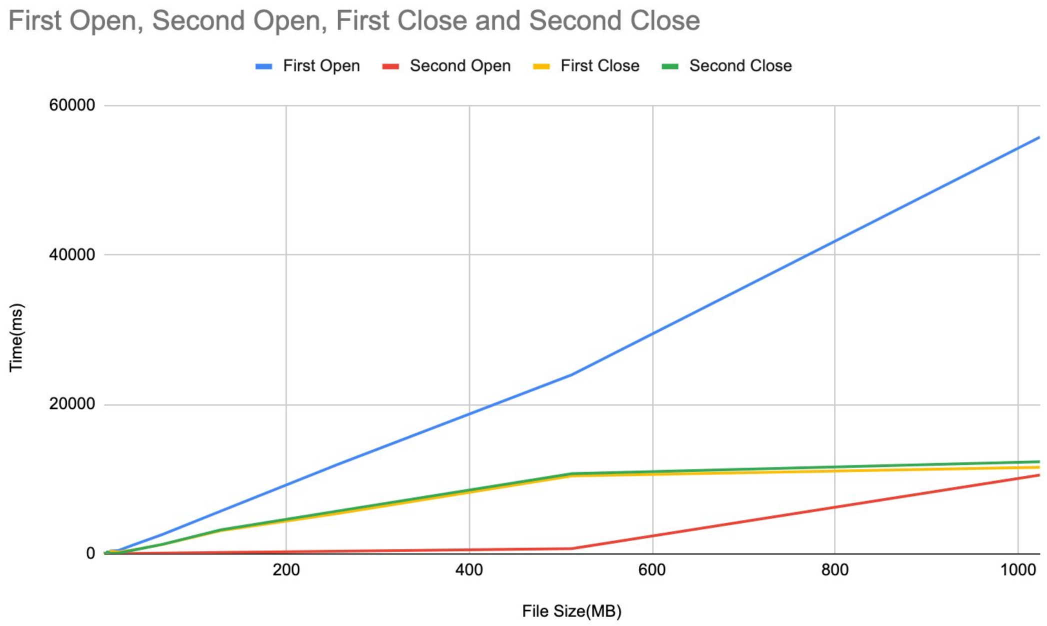
- Create time is independent to the file size
- Close time increases with increase in file size as expected



Case - 2: Client fetches file that existed in the server

Observations:

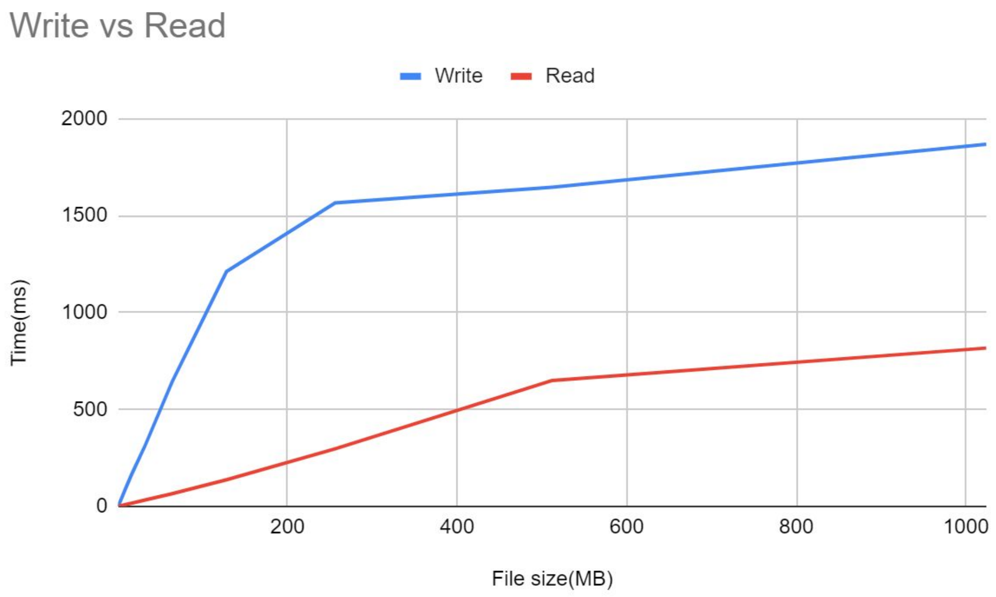
- Only first open is increasing with file size. Since, file is not changed by the client, other operations are not varying with file size



Case - 3: Write vs Read

Observations:

- No server/GRPC communication is involved here
- As expected, read and write time increases with file size.
- These are slower than normal read/write because of fuse intercept

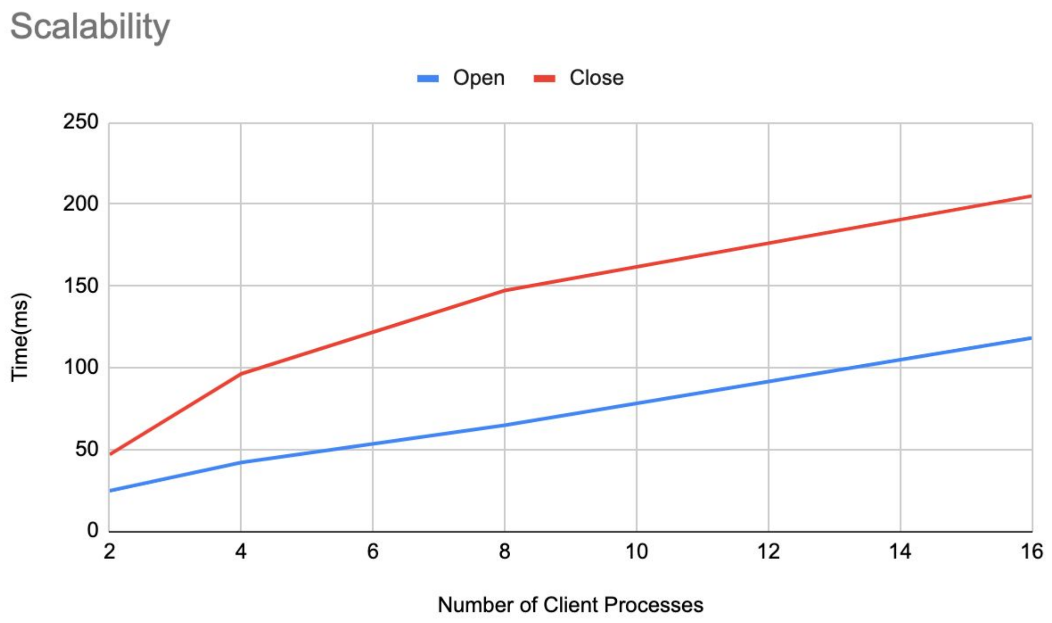


Case - 4: Scalability

Multiple application processes are spawned to perform open and close operations involving gRPC

Observations:

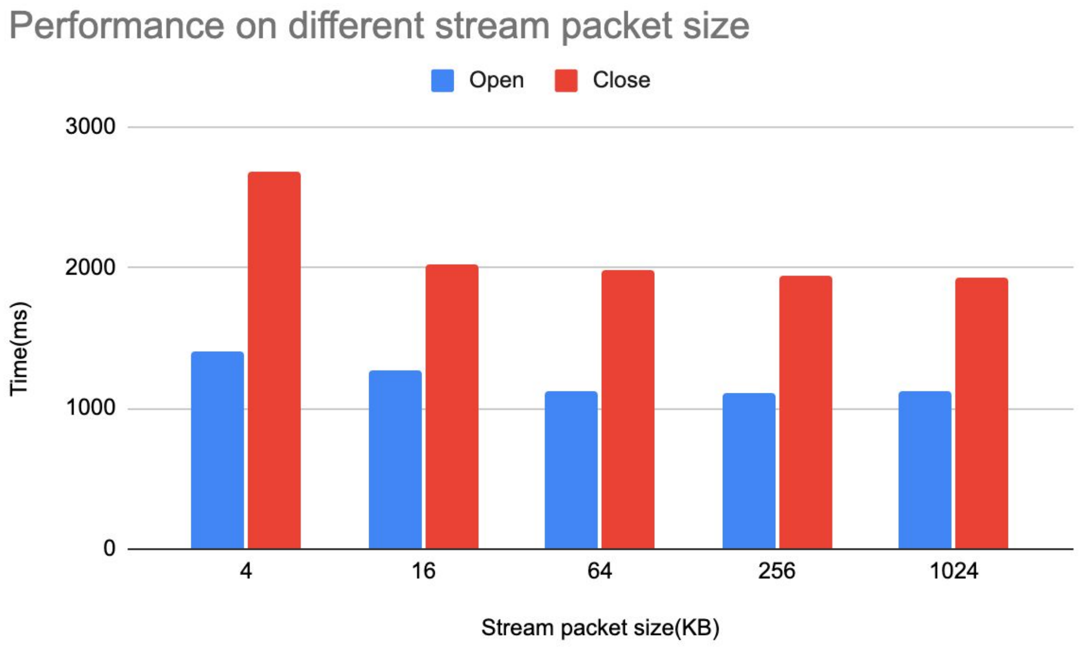
- Latency increases with increase in number of client processes
- This is due to the fact that client side fuse implementations are currently single threaded.



Case - 5: Experimental
(cost wrt gRPC stream packet size)

Observations:

- We experimented gRPC streams with different packet size during open() and close()
- Higher packet size improves open() and close() latency, and reaches optimal around 256KB



Performance Improvement Strategies

Only open() and close() calls involve network communication. So we focused on them for performance improvement.

1. Open() - getAttr to check last modified time, only fetch file if modified on server
2. Close() - Flush to server only if file was modified
3. Open/Close() - Streaming for concurrent reads and writes on client-server

Demos

Demos

1. E2E File-System Demo
2. Last Writer Wins Demo
3. Client Crash Demo
4. Server Crash Demo
5. gRPC Retry Demo

Thank you!