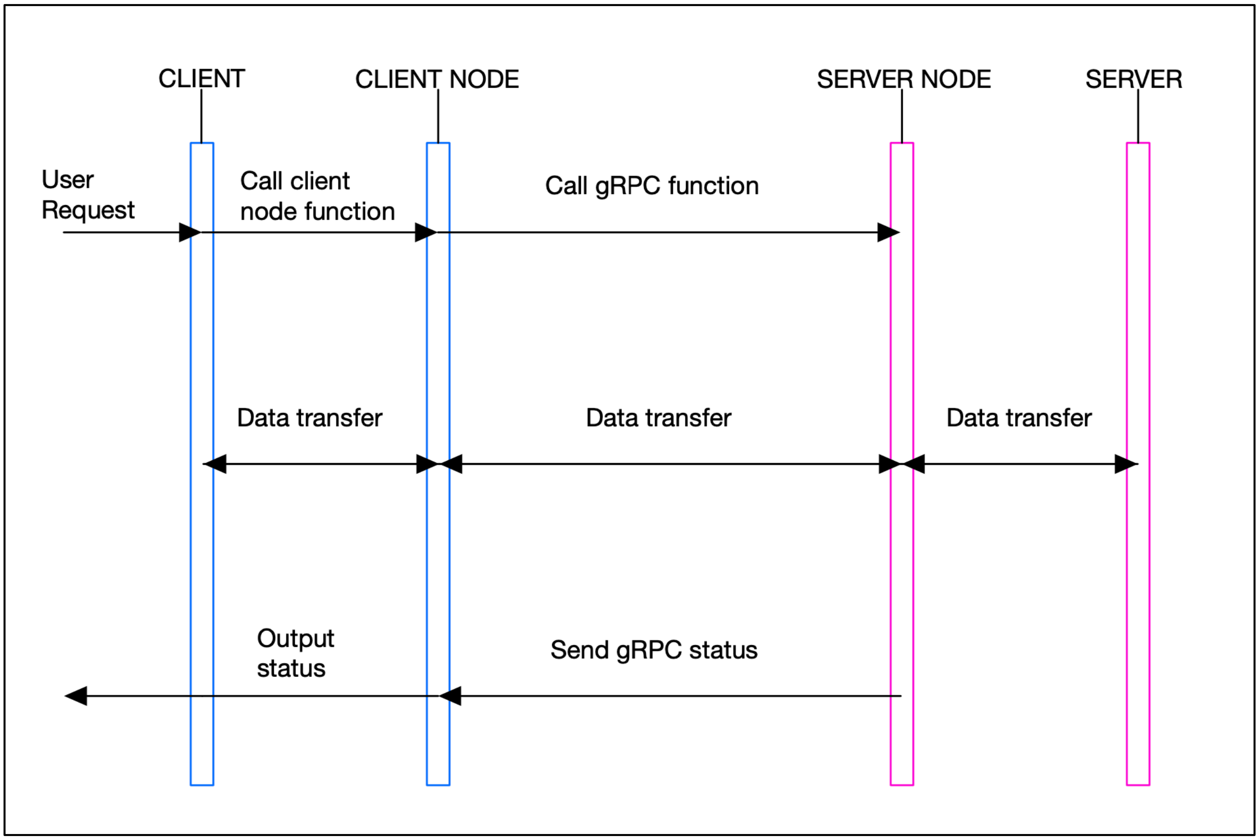
Table of Contents

Project Design:

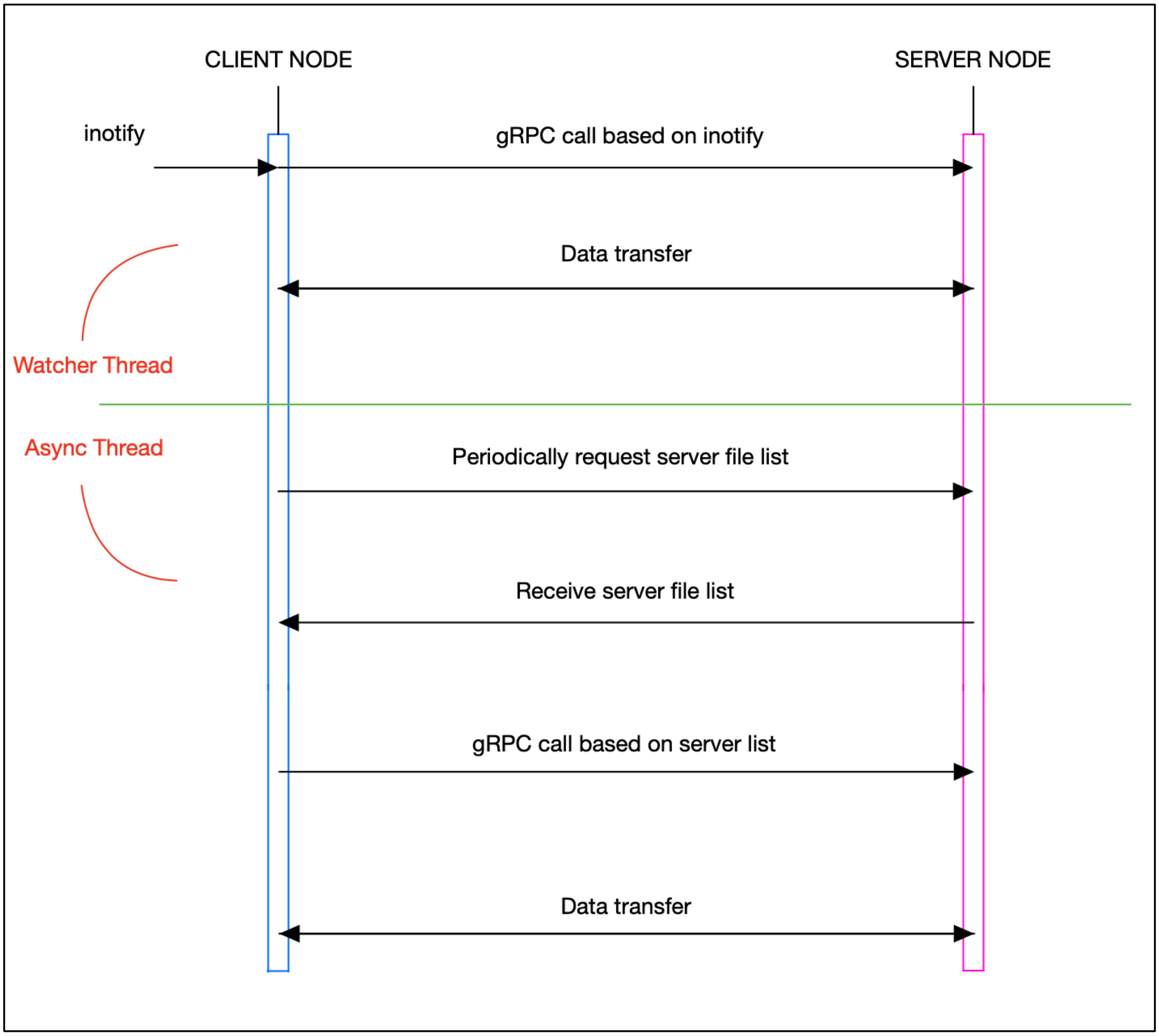
Part1 - Client-Server Protocol Service using gRPC

1. A client-server pair communicate using gRPC.
2. The server supports methods to store a file, fetch a file, delete a file, get stats for a file, and list all files.
3. The client node (depending on what the server returns) should return,
    - gRPC::OK – if all went well
    - gRPC::DEADLINE_EXCEEDED – if client deadline expired at server
    - gRPC::NOT_FOUND – if file not found at server
    - gRPC::CANCELLED – if any other error occurred



Part2 - Distributed File System

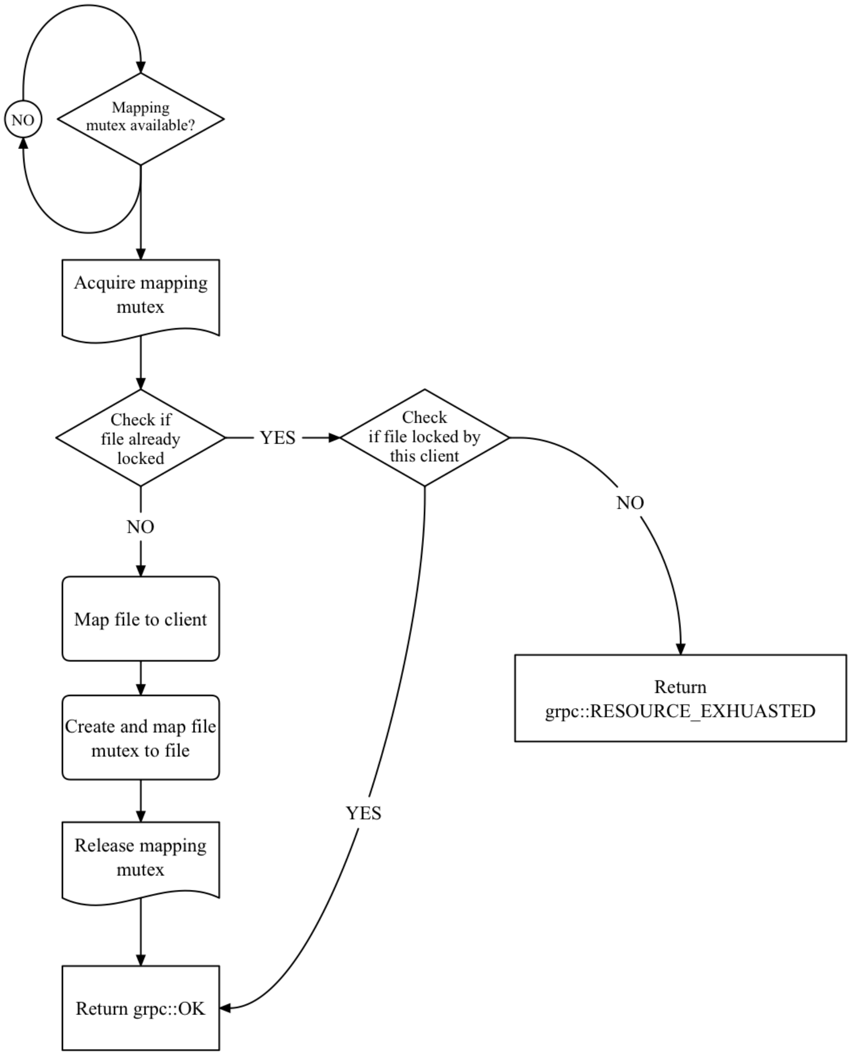
1. A DFS based on the gRPC methods discussed above comprising of one server and multiple clients.
2. There are 2 types of threads in a client node,
   - Watcher threads: Use the inotify system commands to monitor the client directory and store the file or delete the file at the server based on what happened at the client. [1]
   - Async thread: Periodically request the server file list, compare the list against the client list, and make appropriate gRPC calls to the server to synchronize the lists.
3. There will be file locks at the server, so only one client can store or write to a file at one time. [2]



Critical choices and Trade-offs.

Server writer lock in part2 DFS

- Before a client stores or deletes a file, they need to get a writer lock, so only one client stores/deletes a file at one time.
- The client requests a writer lock for the particular file before it can proceed with the store or delete process. If the lock is not available, the attempt will fail.
- For this, the server keeps 2 maps – one that maps the file to the client id, and another that maps the file to its writer mutex. [3]
- Each time a client requests a lock, first we check if the file is already locked. If not, the filename is mapped to the client id and added to the map. A new mutex is created for the file and also added to the mutex map.
- When a client requests to store or delete a file, we first check if the client already has a lock before proceeding.
- The 2 maps are also accessed using a mapping mutex.

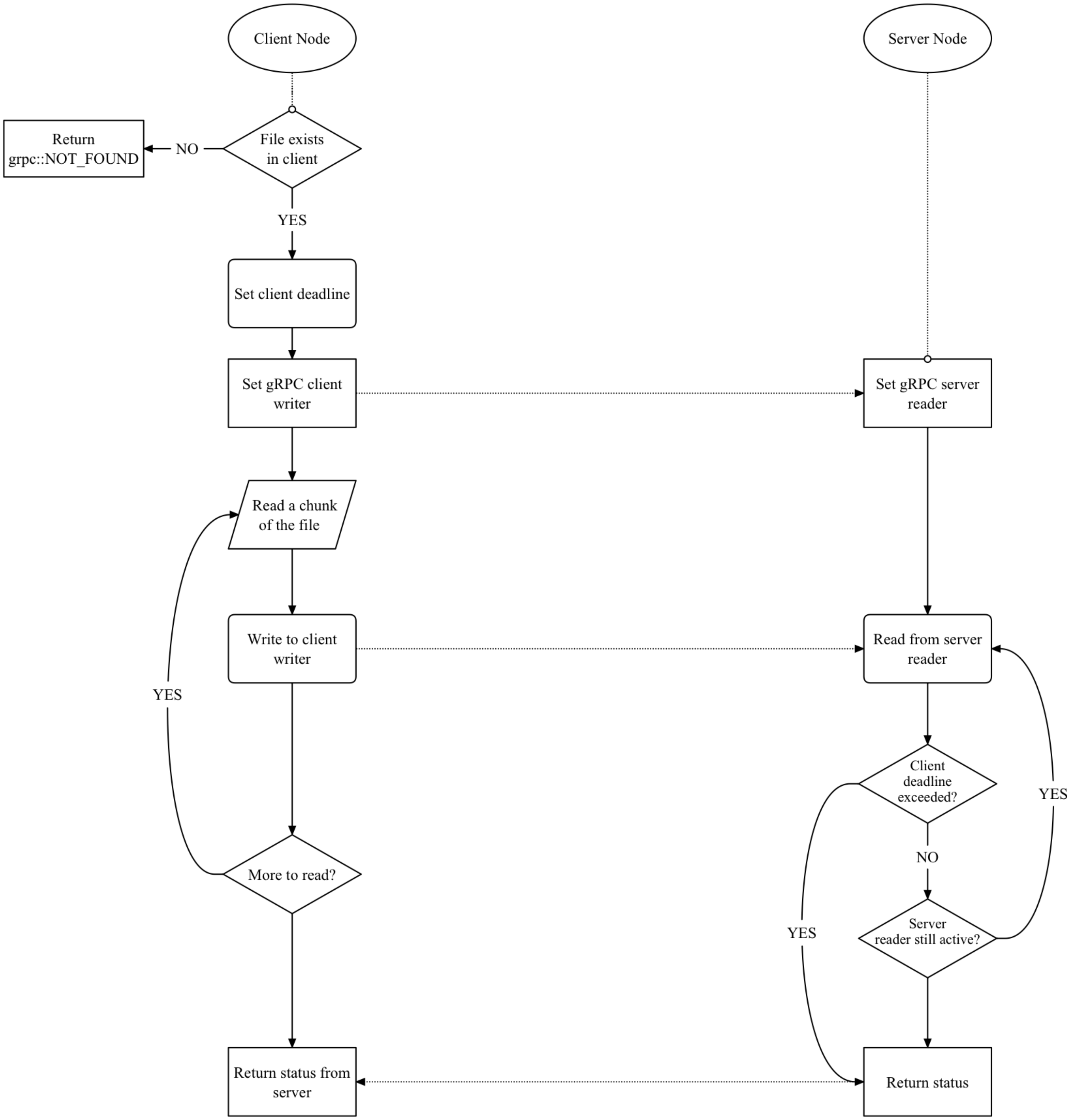

NO
Mapping mutex available?
Acquire mapping mutex
Check if file already locked
YES
Check if file locked by this client
NO
NO
Map file to client
Create and map file mutex to file
Release mapping mutex
YES
Return grpc::RESOURCE_EXHUASTED
Return grpc::OK

All the locks used to avoid race conditions in part2 DFS

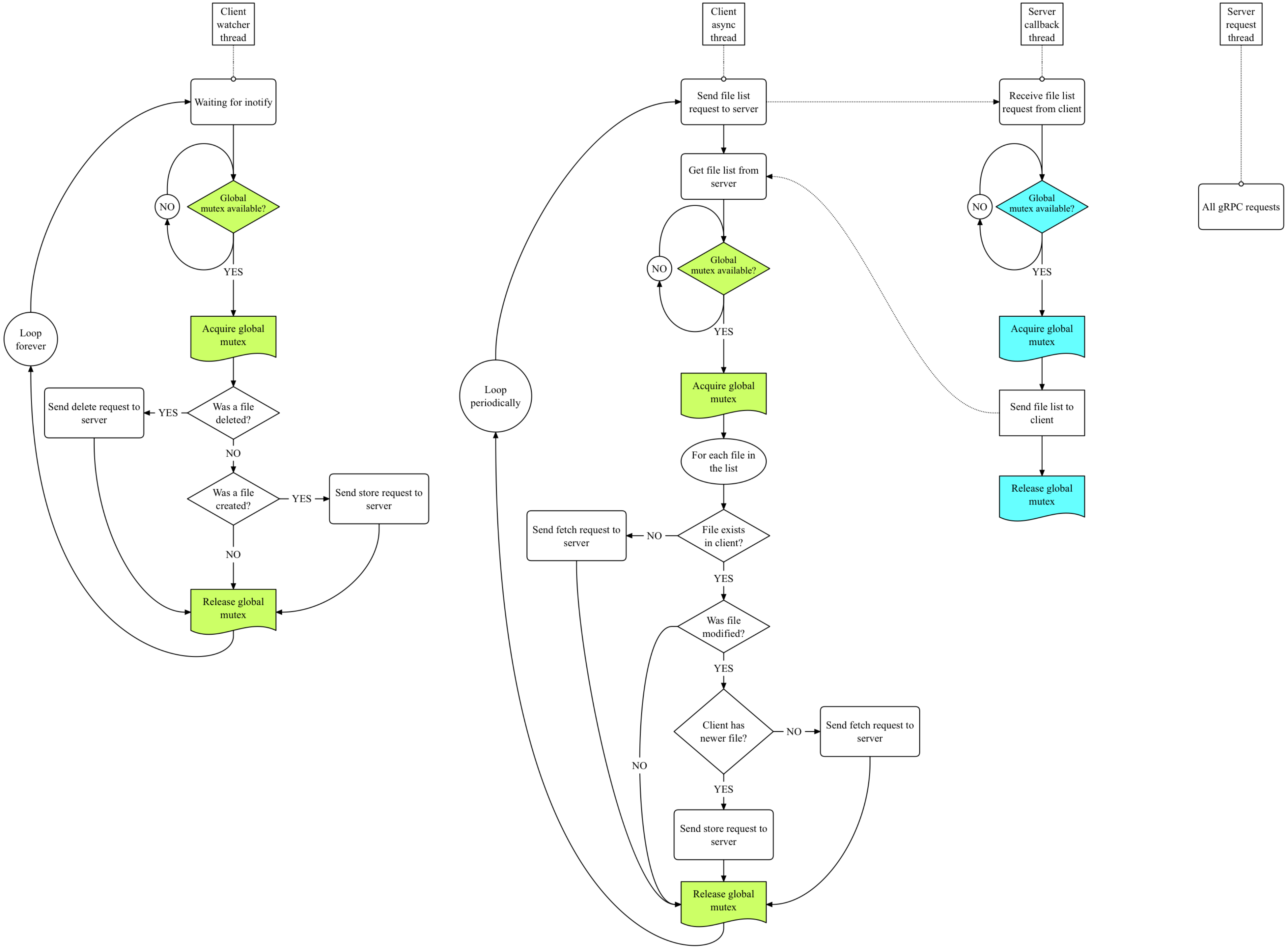
- Client node global mutex: Used to make sure only one of the watcher threads and the async threads are making gRPC methods calls. [4] [5]
- Server node global mutex: Used to make sure only one client's process call back is worked on at one time.
- Writer lock mapping mutex: Used to make sure only one thread is accessing the list of writer locks at one time.
- Filename lock mutex: Used to make sure only one client is modifying a file at one time.

Flow of control

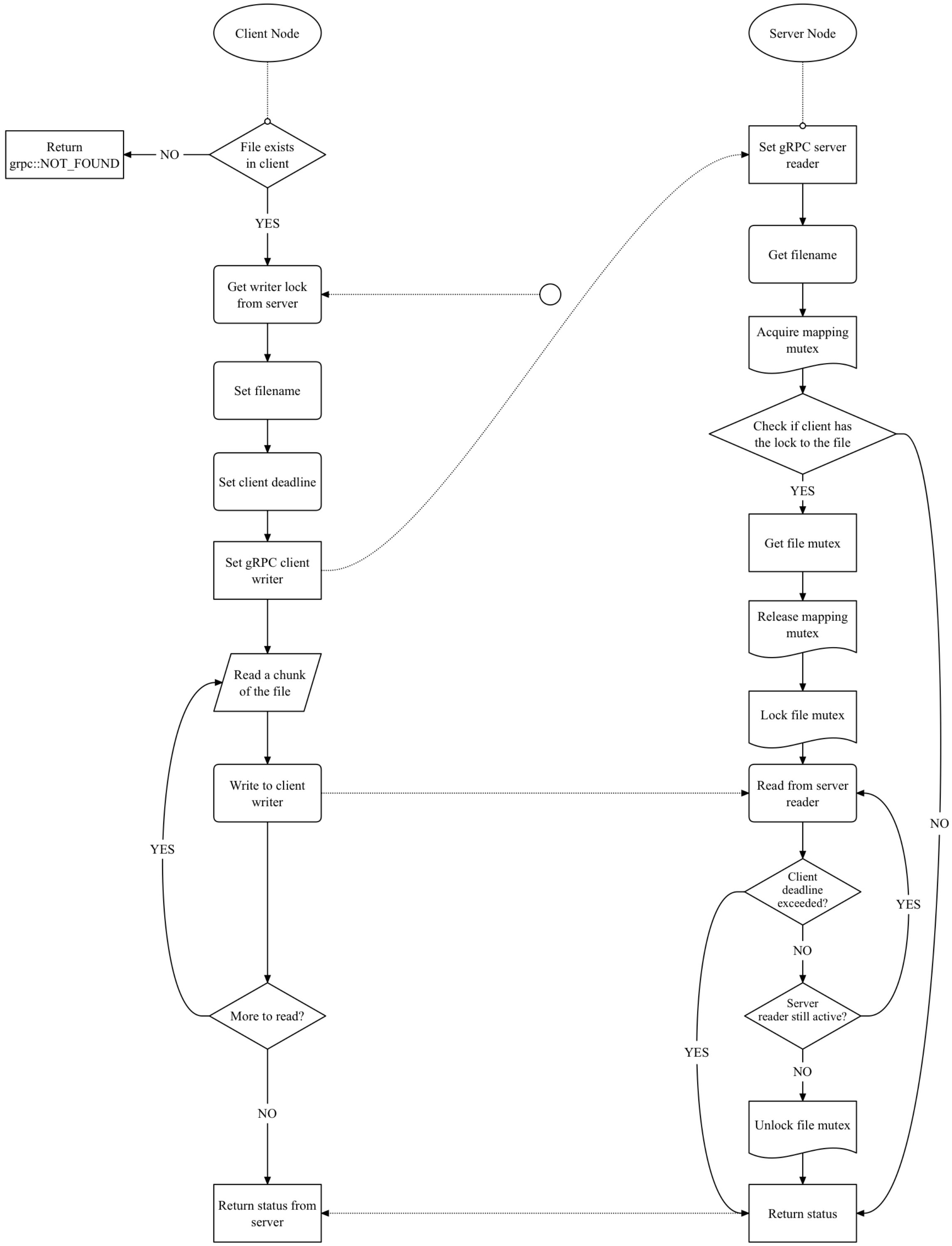
Part1 – Synchronous store method (for brevity only the store method shown for part1) [6] [7]



Part2 – DFS general flow [7]



Part2 – Store method (for brevity, only the store method shown for part2) [7] [6] [5] [2] [3]



Code Implementation

Part1 – Protocol [6] [4]

1. There are 5 methods each for store a file, fetch a file, delete a file, get stats for a file, and to get the server file list.
2. The store, fetch and file list methods use streaming message types.

Part1 - Client [7] [6] [5]

1. The user runs the client executable with the appropriate command (fetch/store/list/delete/stat) and the filename (if required).
2. This calls the appropriate function in the client node (*DFSClientNodeP1*).
3. For the store method, the filename is wrapped to get the file path (*WrapPath*).
4. The file size is calculated (*stat, st_size*) to keep track of the progress if using fetch or store.
5. The client deadline is set (*set_deadline*).
6. The appropriate gRPC call is made and the status is returned.

Part1 - Server [7] [6] [2]

1. The server would be listening on a known port.
2. The actual gRPC service method implementations are in the server node (*DFSServiceImpl*).
3. If required, the filename received from the call is wrapped to get the file path (*WrapPath*).
4. The client deadline is checked periodically to make sure it has not expired (*IsCancelled*).
5. The appropriate task is completed and depending on how it went, the appropriate status is returned.

Part2 – Protocol [7] [6]

1. In addition to the protocol methods in part1, we have a method to get a writer lock and a call back list to handle asynchronous requests for listing files.

Part2 - Client [1] [7] [6] [5]

1. The *inotify* callback gets called each time *inotify* signals a change to a file in the client directory (*InotifyWatcherCallback*).
2. Periodically the client requests the server file list and the asynchronous thread in the client node goes through the file list (*HandleCallbacklist*) and calls the appropriate fetch or store methods depending on if the file exists in the client or who has the newer file.
3. Only one thread between the inotify callback and the callback list handler should be able to call the gRPC methods, and this is made sure using a global mutex.
4. Before a file is stored or deleted to the server, the client needs to get a writer lock for the specific file from the server, so only one client modifies a file at one time.
5. By using the gRPC methods, both the watcher and the async threads work to keep the server and client list synchronized.

Part2 - Server [2] [4] [7] [6] [3]

1. The server node has a function specifically to handle asynchronous requests from the client for the server file lists (*ProcessCallback*).
2. In addition, just like part1, the server node has the actual implementations of the methods defined in the protocol.
3. The server node has a method to acquire a writer lock, before a client can store or delete a file. The server uses 2 maps - one to map the filename to the client and the other to map the filename to the file mutex.
4. For the store and delete methods, the server first checks if the particular client has the writer lock before proceeding.
5. At the server, the asynchronous callback and requests are handled by separate threads.

Testing

- The testing for this project was done using Python.
- For part1, the Python *subprocess* module was used to automatically run a series of client requests with an ever-running server. [8]
- Various file types with sizes ranging from 1KB to 100MB were used for the testing.
- The contents of the files were checked each time in fetch and store method tests.
- Requests for small files, large files, non-existent files, etc. were used to make sure the server handles them correctly.
- The client deadline time was reduced to make sure the client and the server handles them correctly.
- For the part2 DFS, multiple clients and one server were kept running, and a Python script kept changing the files in the clients and the server, randomly.
- At the end of each test, the file lists in each of the client and the server were compared to make sure they are the same. [9]
- ***All the Gradescope tests passed for each of the project sections.***

Works Cited

[1] M. Kerrisk, "inotify(7) — Linux manual page," [Online]. Available: https://man7.org/linux/man-pages/man7/inotify.7.html.

[2] A. Raman, *Multithreaded GETFILE Server Code.*

[3] cplusplus.com, "map," [Online]. Available: http://www.cplusplus.com/reference/map/map/.

[4] A. D. Birrell, "An Introduction to Programming with Threads," 1989.

[5] A. Raman, *Multithreaded GETFILE Client Code.*

[6] The Linux Foundation, "Introduction to gRPC," [Online]. Available: https://grpc.io/docs/what-is-grpc/introduction/.

[7] A. Gavrilovska, *Georgia Institute of Technology, CS6200 Lectures.*

[8] Python Software Foundation, "Subprocess management," [Online]. Available: https://docs.python.org/3/library/subprocess.html.

[9] Python Software Foundation, "File and Directory Comparisons," [Online]. Available: https://docs.python.org/3/library/filecmp.html.