Jordan Tannenbaum
Fall 2024
CS 6200

Distributed File System Design Document

Part 1: Building the RPC Protocol Service

Project Design and Implementation
The design of the RPC protocol service was based largely on the provided example code in the gRPC C++ documentation.

Here, the design centers on the five service calls defined in the dfs-service.proto file. These calls include Store(), Fetch(), Delete(), List(), and Stat(). For each call, there is a unique request and response message. For example, the request and response for Delete() are DeleteRequest and DeleteResponse, respectively.

The service call Store() is a client-to-server streaming RPC. This means that Store will continuously stream requests from the client to the server via the gRPC Client Writer until certain conditions are met.

Store()'s request message is StoreRequest which contains a file name and a file chunk. Store()'s response message is StoreResponse, which is a file acknowledgement containing a file name and mtime.

On the client-side, Store() sets the gRPC timeout deadline, confirms that the file it would like to store exists in the client directory, and opens the gRPC Client Writer. Here, the Client Writer, via the service stub, calls Store() and passes it a StoreRequest containing the name of the file to be stored. Next, the client creates a buffer and begins a loop that fills the buffer with file data, sets the StoreRequest file chunk to said buffer, and uses the Client Writer to write that StoreRequest to the server. This process continues until the entire file is sent. The client then returns the appropriate StatusCode, such as DEADLINE_EXCEEDED or OK, based on the server's Status. For OK statuses, the client will perform an additional check to ensure that a StoreResponse with a file acknowledgement has been received.

On the server-side, the server loops through a gRPC Server Reader which continuously reads the StoreRequests and checks if the client has cancelled the request or if the deadline has been exceeded. Upon receiving the first StoreRequest, the server will open an ofstream buffer to store the file. The server will then begin a while loop where in each iteration of the loop, the server will check if the deadline has been exceeded. If it has not, the file chunk received in the StoreRequest will be written to this buffer. This loop continues until the Client Writer stops receiving requests. The server will then return the appropriate Status based on the success of the write operations. Before sending an OK Status, the server will populate its StoreResponse with a file acknowledgement.

The next service call, Fetch(), is a server-to-client streaming RPC. Here, FetchRequest contains the name of the file to be fetched and FetchResponse contains a file chunk along with a file name and mtime for file acknowledgement.

For the client, Fetch() sets the timeout deadline and instantiates the gRPC Client Reader which makes the Fetch() call via the service stub. The client then follows a similar logic as the server did for Store(), looping through its Client Reader and writing file chunks to memory until no more responses arrive for the Reader to read. Once this concludes, the client reports a StatusCode that corresponds to the server's Status. Like with Store() the client confirms the FetchResponse contains a file acknowledgement before returning an OK StatusCode.

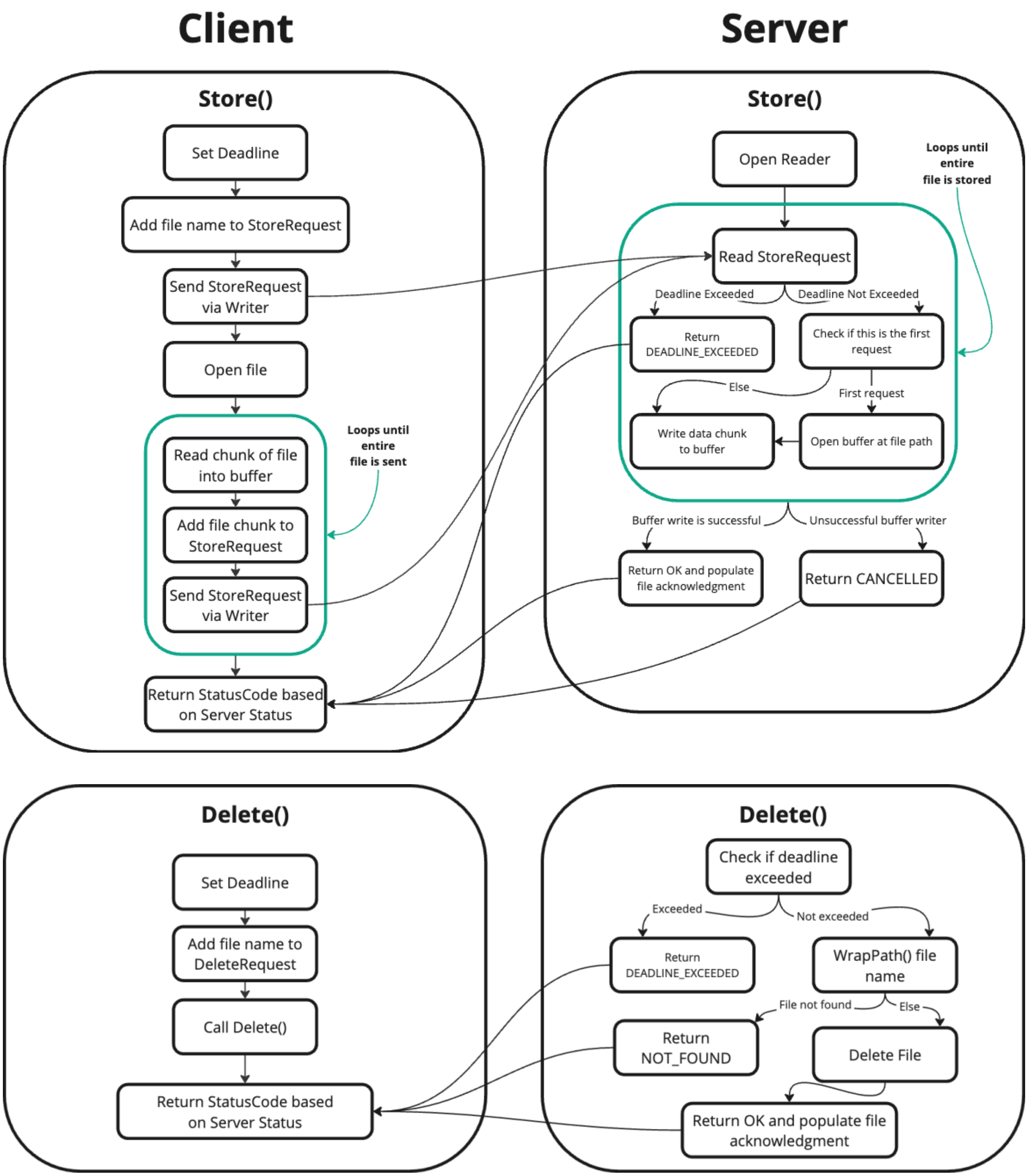
Delete(), List(), and Status() are all simple RPCs with no streaming involved. Delete() uses a DeleteRequest, containing a file name, and a DeleteResponse which is a file acknowledgement. List() has a ListRequest, which is empty since List() provides data on all files, and a ListResponse that contains a file list of FileData objects. FileData is another message type used exclusively for ListResponse. It stores a file's name and mtime. Finally, Status() uses a StatusRequest which contains the name of the file to get the status of. For a response, it uses StatusResponse which contains the name, size, mtime, and crc of the requested file.

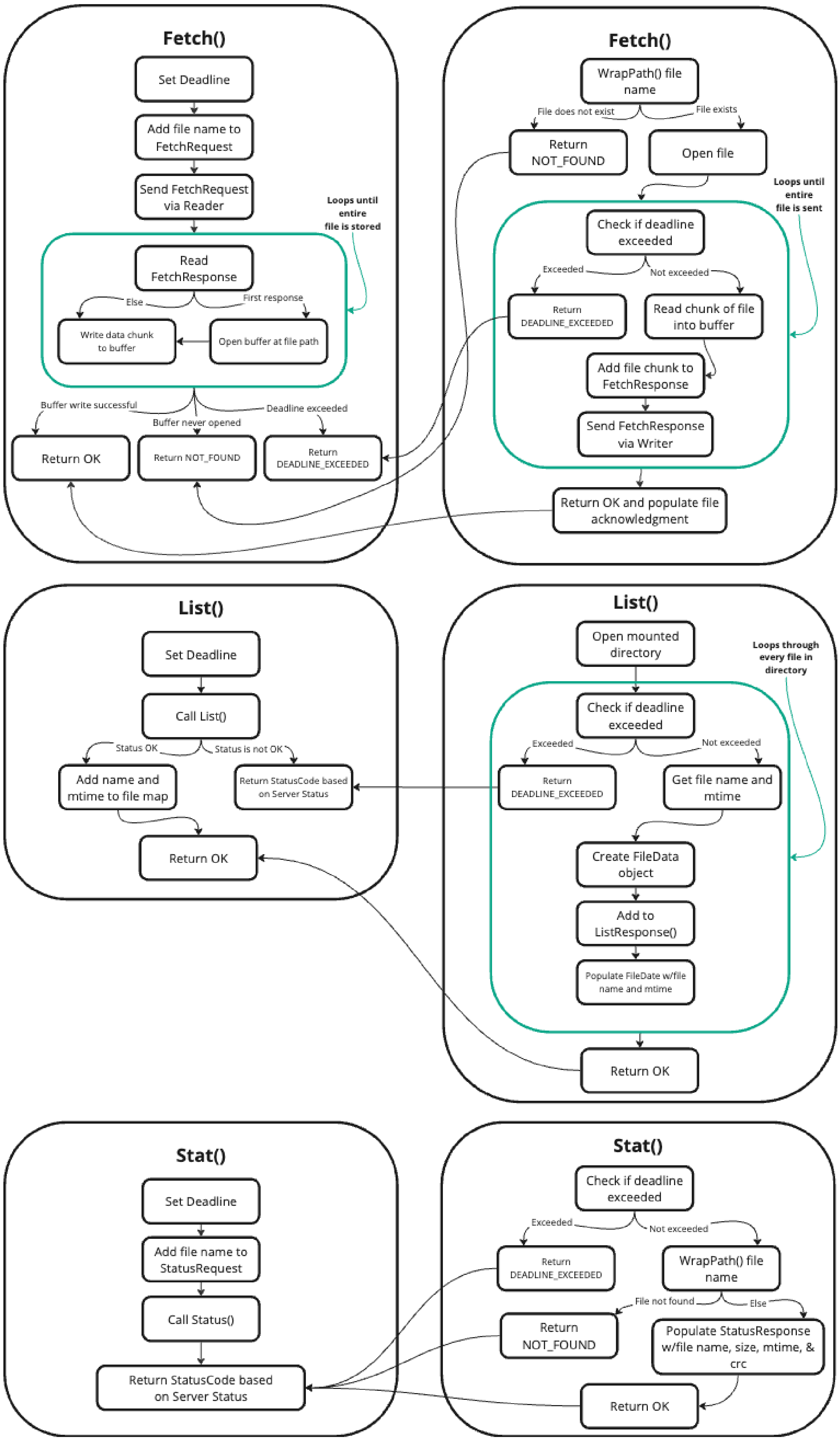
The client side of these service calls begin essentially the same with each client populating its respective request, setting its timeout deadline, and using the service stub to pass its request to its associated service call. The clients then examine if the server returned and DEADLINE_EXCEEDED or CANCELLED Status and return their own associated StatusCodes. Otherwise, the clients ensure that a file acknowledgment was returned and return the OK StatusCode. That said, List() includes additional functionality before returning with an OK StatusCode where the ListResponse()'s file list is iterated through and each file's name and mtime are added as key and value respectively to the file_map.

Likewise, the server counterparts for Delete() and Stat() are largely the same. Both first check if the deadline is exceeded and return the DEADLINE_EXCEEDED Status if it is. Otherwise, both check if the requested file exists and returns NOT_FOUND if it does not. Here, Delete() and Stat() diverge. Delete() will populate its file acknowledgement and delete the file before returning the correct Status based on if Delete() was successful. On the other hand, Stat() will acquire the file's size, mtime, and crc. It then populates the corresponding parameters, along with the file name, in its StatusResponse. If it can do so without issue, it returns an OK Status. Likewise, if the file is not found or the deadline is exceeded, NOT_FOUND and DEADLINE_EXCEEDED would be returned respectively.

That said, List()'s server function is more unique. It begins by opening the directory of the mounted path. It then loops through each file in the directory and adds name and mtime to a FileData object. This FileData object is then added to the ListResponse's file list. This process continues until every file in the mounted directory is added to the file list and the function concludes, returning the OK Status if successful.

Control Flow




Fetch()
Set Deadline
Add file name to FetchRequest
Send FetchRequest via Reader
Loops until entire file is stored
Read FetchResponse
Else
First response
Write data chunk to buffer
Open buffer at file path
Buffer write successful
Buffer never opened
Deadline exceeded
Return OK
Return NOT_FOUND
Return DEADLINE_EXCEEDED
Fetch()
WrapPath() file name
File does not exist
File exists
Return NOT_FOUND
Open file
Loops until entire file is sent
Check if deadline exceeded
Exceeded
Not exceeded
Return DEADLINE_EXCEEDED
Read chunk of file into buffer
Add file chunk to FetchResponse
Send FetchResponse via Writer
Return OK and populate file acknowledgment
List()
Set Deadline
Call List()
Status OK
Status is not OK
Add name and mtime to file map
Return StatusCode based on Server Status
Return OK
List()
Open mounted directory
Loops through every file in directory
Check if deadline exceeded
Exceeded
Not exceeded
Return DEADLINE_EXCEEDED
Get file name and mtime
Create FileData object
Add to ListResponse()
Populate FileDate w/file name and mtime
Return OK
Stat()
Set Deadline
Add file name to StatusRequest
Call Status()
Return StatusCode based on Server Status
Stat()
Check if deadline exceeded
Exceeded
Not exceeded
Return DEADLINE_EXCEEDED
WrapPath() file name
File not found
Else
Return NOT_FOUND
Populate StatusResponse w/file name, size, mtime, & crc
Return OK

Trade-offs
The primary trade-off was the decision to use unique request and response messages for each service call. This is because many of these requests, like FetchRequest, DeleteRequest, and StatusRequest, contain the same information and therefore making each request unique is redundant. However, separating these requests and responses ensures that if additional functionality needs to be added to one of these calls, it could be done without impacting the other calls. Moreover, based on the example code in the gRPC C++ documentation, it appeared that unique requests and responses for each call is standard convention. As such, despite the additional overhead, the decision to use unique requests and responses was made.

Testing
PR4 Part 1 testing was largely manual. For each service call, core functionality was tested with .png, .jpg, and .txt file types. For example, for Delete(), the function was called on a 3000 line .txt file, a 1 line .txt file, a 100x100 .png, a 1000x1000 .png, 100x100 .jpg, and 1000x1000 .jpg.

Once these initial tests were passed, additional tests were run for Store() and Fetch() with .txt files. For Store(), first a file called TestDoc.txt containing the text "Hello World!" was stored on the server. Then, in the client's copy of TestDoc.txt, the text was changed to "Hello Fall 2024!" The Store() function was again called on TestDoc.txt. Passing this test ensured that existing files could successfully be overwritten. Similarly, the same test was then conducted where TestDoc.txt on the client contained the text "Hello!" and TestDoc.txt on the server contained "Hello Fall 2024!" Passing this test ensured that when overwriting, extra bytes that were used in the old file but not in the new file are cleared. These same tests were done in reverse for Fetch() where it was ensured that overwriting worked properly when writing from server to client.

Part 2: Implementing the Distributed File System (DFS)

Project Design and Implementation
Here, whenever a file is created, modified, or deleted from a client or server, the change should be propagated to the other client(s) and/or server. As such, additional functionality was added to some of the previously described service calls for multithreading support and new functionality was created to support asynchronous updates.

To enable coherent multithreading, the RequestWriteAccess() service call was created. This call accepts a WriteRequest containing a file name and client ID and returns a WriteLock as a response. WriteLock itself is empty as the client can determine the success of the RequestWriteAccess() function based on the returning Status.

As a simple RPC, the client side of RequestWriteAccess() is like that of Stat(), Delete(), and List(). Here, the client sets the timeout deadline, populates the WriteRequest with the client ID and filename, and then makes the RequestWriteAccess() call via the service stub.

It then returns the appropriate StatusCode based on the Status of the server. Notably, if the server cannot provide write access to the client, a RESOURCE_EXHAUSTED StatusCode will be returned.

The server side of the function centers around the write_locks map and its corresponding write_locks_mutex. This map contains a key-value pair of filenames to client IDs, making it a dictionary denoting which files are being written to by which clients. This is mutex locked to ensure that multiple server threads cannot edit the map at once, preventing race conditions where different server threads try to give different clients write access to the same file.

As such, when RequestWriteAccess() is called on the server, it locks the write_locks_mutex, checks if lock is free and if it is, adds the file name and client ID to the write_locks map before returning the OK Status. Additionally, if the client already has the lock, then the function will simply return the OK Status. Likewise, if the lock is held by another client, the RESOURCE_EXHAUSTED Status will be returned.

As the name suggests, the goal of RequestWriteAccess() is for the client to request write access for a file. As such, this function is now called in the client-side versions Store() and Delete() before attempting the previously described functionality of Part 1. On the server-side, these functions now ensure that the client has the lock before initiating their core functionality. Moreover, these functions oversee releasing the locks once they complete their function. For example, if a client calls Store(), the client will first get a write lock on a given file via RequestWriteAccess() before calling the server-side Store(). Once the server-side Store() finishes storing the file, it will then release the client's write-lock.

The choice to release locks server-side was an explicit design decision. Initially, there was an additional ReleaseWriteAccess() function which the client called after using Store() or Delete() to release their locks. However, it became apparent that if the client were to crash or otherwise misbehave, it opens the possibility to ReleaseWriteAccess() never being called and the client maintaining their lock indefinitely. As such, this functionality was moved to the server.

Asynchronous functionality is supported through two channels. The first is the inotify channel which reads data from the client and writes it to the server. The second channel is through the CallbackList RPC which reads data from the server and writes to the client. To ensure that these two channels do not overwrite each other, the client has an async_mutex which is locked whenever either channel attempts to add, modify or delete a file. For the inotify channel, this is before the callback function is called and for the CallbackList RPC, this is every time a result is available in the completion queue.

Because the inotify callback is largely boilerplate, the bulk of the asynchronous implementation deals with the CallbackList RPC. On the server-side this service call is implemented in the ProcessCallback() function. To begin, this function locks the write_locks_mutex to ensure that no more Store() and Delete() operations will be completed before starting its core functionality. This ensures that the file and delete lists are as recent as possible. The function then works to populate its CallbackListResponse with two lists where each list is composed of FileInfo messages. These FileInfo messages contain a file's name, size, mtime, and crc.

The first list, called file list, is a list of all the files in the server's directory. This implementation is essentially identical to the previously described List() function from Part 1; however, dot files are ignored. The second list, called delete list, contains all the files that have been deleted on the server.

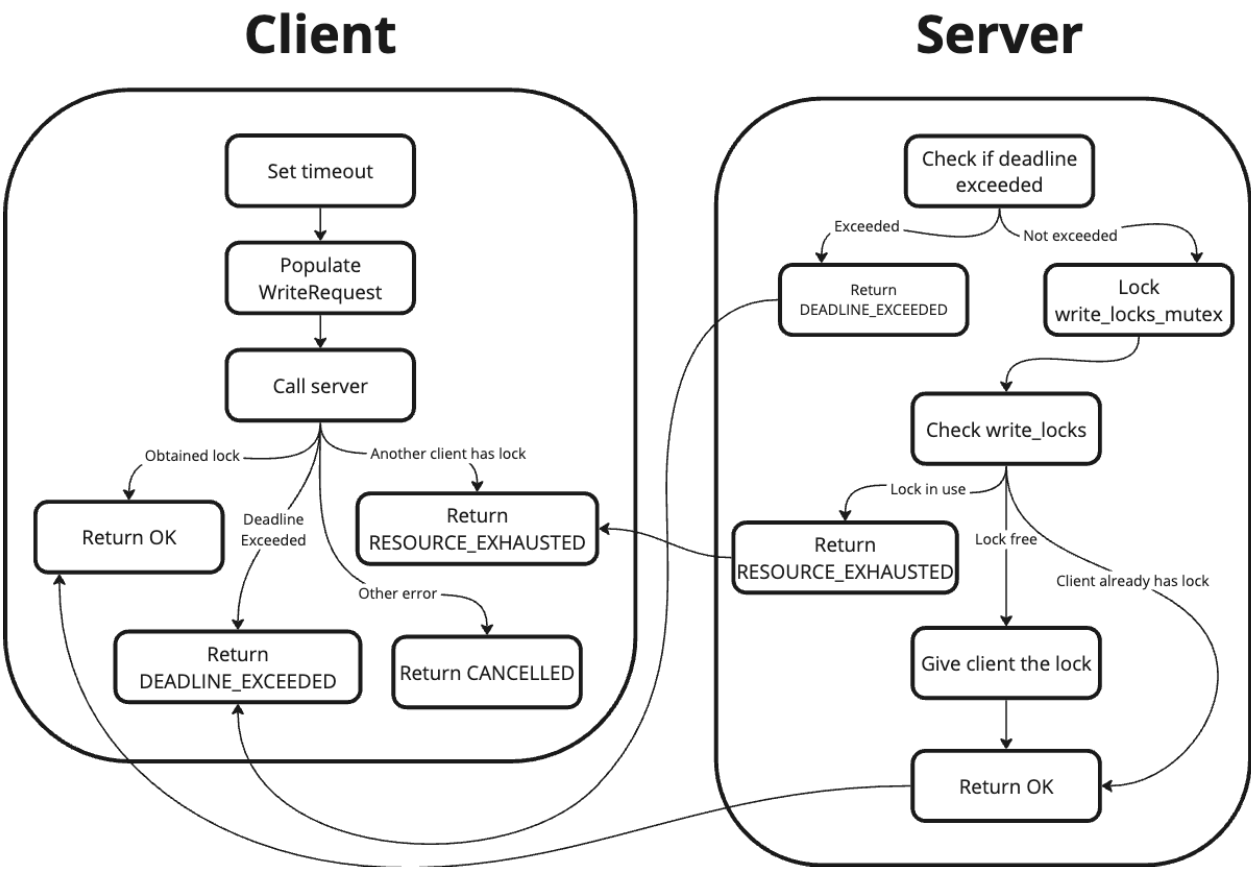
To create the delete list, a tombstoning method was used in the server-side Delete(). Here, a tombstone, which is a record of a deleted file, is created before every deletion. It contains the file's name, size, mtime, and crc. It is then stored in a mutex-protected tombstones map with the file name as the key. To then create the delete list, ProcessCallback() simply locks the tombstone mutex and iterates through the map, populating a FileInfo object with each record's information and adding it to the CallbackResponse delete list.

The reason this tombstoning was implemented was to avoid unintentional file recreation when Delete() is called. For example, if one client deletes a file from the server, another client that has that deleted file may reupload that file to the server, thinking that the server was simply missing the file. However, by creating a record of deleted files, the client's file can be compared against the server's deleted file record to determine if that file should be uploaded to the server or if it should instead be deleted off the client.

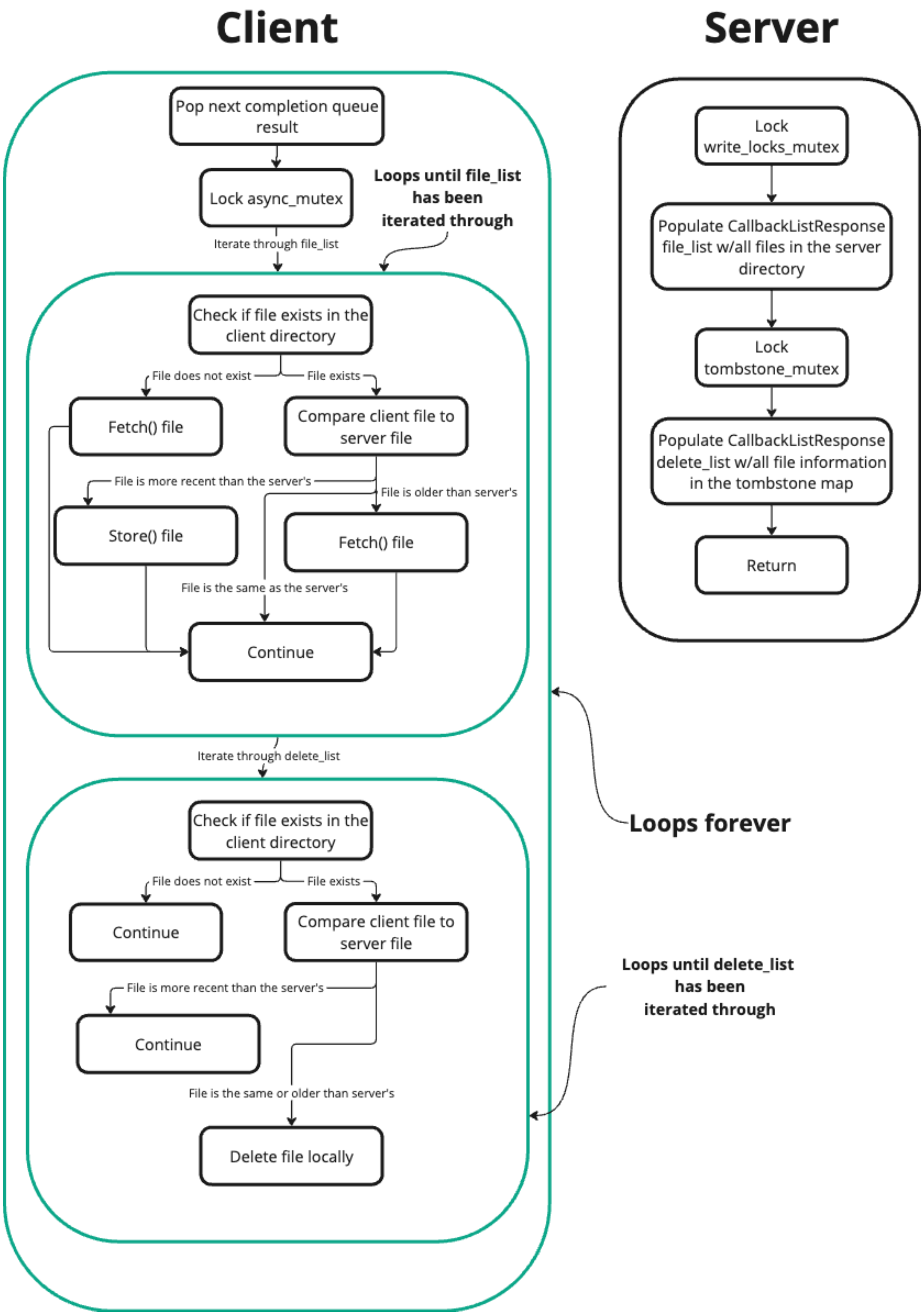
Once these lists are created, they are sent to the client which processes them in HandleCallbackList(). Here, once gaining access to the async_mutex, each list is iterated through. First is the file list. If a given file does not exist in the client directory, the file is then Fetch()ed. Otherwise, the file is compared between the client and server. If the client has a more recently updated version of the file, determined by comparing mtimes, the client will Store() its version of the file. Likewise, if the server's file is newer, the client will Fetch() the file.

The client will then process the delete list. If a file is found on the delete list and on the client, as determined by comparing mtimes, crcs, file names, and file sizes, the client will delete its copy of the file. This process will continue until the entire delete list has been iterated through and the client will await the next response from ProcessCallbackList().

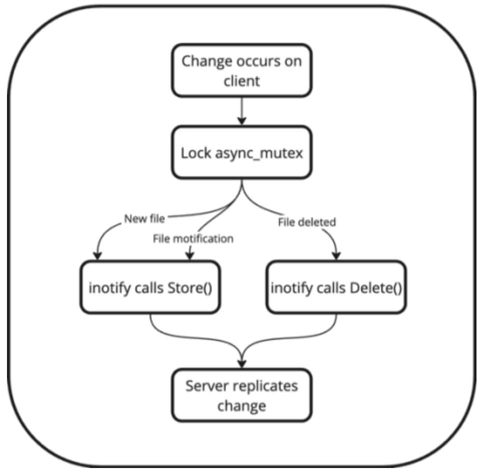
Control Flow – RequestWriteAccess



Control Flow – CallbackList



Control Flow – inotify



Trade-offs

There were two key trade-offs in this implementation. The first was the decision to not have a separate ReleaseWriteAccess() function. As discussed in the previous section, the decision to release write access within the Store() and Delete() functions was to prevent indefinite lock holding by clients that have crashed. However, from a functional programming perspective, this decision adds additional overhead to the Store() and Delete() functions as they now have extra functionality that is not entirely part of their core use-case. As a result, it could be argued that releasing write access should be separate function and that clients should be programmed to gracefully exit like and in doing so relinquish their locks. However, given the scope of this project, this did not appear necessary and thus the decision to have slightly more bloated, but safer versions of Store() and Delete() was made.

The second trade-off was the use of tombstoning. While the form of tombstoning used ensures that the clients know exactly which files should and should not be on the server, this does come with somewhat notable overhead given that metadata about each deleted file is stored. However, given the scope of this assignment, it seemed unlikely that memory limitations would be met that would prevent more tombstones from being created. That said, in an enterprise environment, some type of "janitor" function would likely need to be implemented to clean out the tombstone map at recurring intervals. However, for the scope of this assignment, the decision was made that the extra memory overhead was worth the functionality.

Testing

Part 2 testing was similar to Part 1's. Once again, each service call's core functionality was tested with a 3000 line .txt file, a 1 line .txt file, a 100x100 .png, a 1000x1000 .png, 100x100 .jpg, and 1000x1000 .jpg. This was then followed by the additional overwrite tests described in Part 1's testing section.

In addition to the Part 1 tests, Part 2’s asynchronous functions were also tested. Here, three client instances were created, each with its own directory. The first two instances were asynchronous while the third simply handled command line arguments. Here, the third instance called Store() and Delete() for various files and the server and other client directories were examined immediately after to ensure that the correct changes took place.

For example, when TestDoc2.txt was deleted via the third client, it was ensured that this file was deleted from the two other clients and the server as well.

Moreover, files were then added, removed, and modified from each of the asynchronous clients and the server. It was then ensured that these changes were reflected in the remaining clients and/or server. This process was then repeated for the previously mentioned file types to ensure consistency across file size and format.

References

Asynchronous Callback API Tutorial. (2024, June 18). GRPC. https://grpc.io/docs/languages/cpp/callback/

Asynchronous-API tutorial. (2022, February 17). GRPC. https://grpc.io/docs/languages/cpp/async/

Basics tutorial. (2024, November 25). GRPC. https://grpc.io/docs/languages/cpp/basics/#server

c - How to set the modification time of a file programmatically? (2010, February 2). Stack Overflow. https://stackoverflow.com/questions/2185338/how-to-set-the-modification-time-of-a-file-programmatically

C++ remove() - C++ Standard Library. (2024). Programiz.com. https://www.programiz.com/cpp-programming/library-function/cstdio/remove

Can we have functions inside functions in C++? (2019, May 8). Stack Overflow. https://stackoverflow.com/questions/4324763/can-we-have-functions-inside-functions-in-c

Casting to void and Back to Original_Data_Type**. (2012, September 5). Stack Overflow. https://stackoverflow.com/questions/12275321/casting-to-void-and-back-to-original-data-type

Difference between std::mutex lock function and std::lock_guard? (2016, July 12). Stack Overflow. https://stackoverflow.com/questions/38340378/difference-between-stdmutex-lock-function-and-stdlock-guardstdmutex

GIOS Staff, & GIOS Student Contributers. (2024). *GIOS Slack*. GIOS Slack. https://gatech.enterprise.slack.com/archives/CF8H98PL4

Google LLC. (2024). *Protocol Buffers Documentation*. Protobuf.dev. https://protobuf.dev/

GRPC C++ Documentation. (2024). Github.io. https://grpc.github.io/grpc/cpp/index.html

How do you add a repeated field using Google's Protocol Buffer in C++? (2009, November 20). Stack Overflow. https://stackoverflow.com/questions/1770707/how-do-you-add-a-repeated-field-using-googles-protocol-buffer-in-c

How do you loop through a std::map? (2014, October 9). Stack Overflow. https://stackoverflow.com/questions/26281979/how-do-you-loop-through-a-stdmap

How to find if a given key exists in a std::map. (2009, December 21). Stack Overflow. https://stackoverflow.com/questions/1939953/how-to-find-if-a-given-key-exists-in-a-stdmap

ifstream::is_open. (2023). Cplusplus.com. https://cplusplus.com/reference/fstream/ifstream/is_open/

Moore, P. (2024). *PR4 Part 1 Sequence Diagram*. GitHub. https://github.gatech.edu/gios-fall-24/pr4/blob/main/docs/part1-sequence.pdf

Mutex in C++. (2023, November 11). GeeksforGeeks. https://www.geeksforgeeks.org/std-mutex-in-cpp/

Protocol buffers - unique numbered tag - clarification? (2014, November 9). Stack Overflow. https://stackoverflow.com/questions/26826421/protocol-buffers-unique-numbered-tag-clarification

Remove a key from a C++ map. (2014, February 28). Stack Overflow. https://stackoverflow.com/questions/10038985/remove-a-key-from-a-c-map

Scope with Brackets in C++. (2011, February 22). Stack Overflow. https://stackoverflow.com/questions/5072845/scope-with-brackets-in-c

Sheerin, G. (2018, February 26). *gRPC and Deadlines*. GRPC Blog. https://grpc.io/blog/deadlines/

Wikipedia Contributors. (2024, April 2). *Tombstone (data store)*. Wikipedia; Wikimedia Foundation. https://en.wikipedia.org/wiki/Tombstone_(data_store)