Part 1

Design

For Part 1, we had to implement a basic GRPC client and server. We used `dfs-service.proto` to define our protocol (the requests supported, the request types, and the response types).

For my design, I implemented the following:

- `rpc StoreFile` with input `stream StoreRequest` and response `google.protobuf.Empty`
- `rpc FetchFile` with input `FileRequest` and response `stream FetchResponse`
- `rpc DeleteFile` with input `FileRequest` and response `google.protobuf.Empty`
- `rpc ListFiles` with input `google.protobuf.Empty` and response `ListResponse`
- `rpc GetFileStatus` with input `FileRequest` and response `FileStatusResponse`

Each `FileRequest` message type consists of just the file name. `StoreRequest` includes both file name and content bytes, `FetchResponse` includes just content bytes, and `ListResponse` contains a `map` of strings and `modified_time` (stored as an `int64`). The `FileStatusRequest` is just for the file name, and `FileStatusResponse` includes `size`, `modified_time`, and `creation_time`.

For the client-side implementation, I followed the `GRPC` documentation and tried to keep my implementation as simple as possible. The `Store` method sets a deadline, then calls the server-side `StoreFile` rpc. If the fiel is found, it then populates the `StoreRequest` until the entire file is processed. The client-side `Fetch` does the same thing but using the server's `FetchFile` rpc instead. Similarly, the `Delete` calls the `DeleteFile` rpc, `List` calls the `ListFiles` rpc, and `Stat` calls the `GetFileStatus` rpc.

The server-side impelmentation is also relatively straightfoward. `StoreFile` opens a file, reads the request, writes the file, then closes the file. `FetchFile` attempts to find the requested file, then streams the bytes until the end of the file, then closes the file. `DeleteFile` attempts to find the file, and deletes it if found. `ListFiles` iterates over each file in the directory and returns a `map` of all the results with the `file_name` and `modified_time`. Finally, `GetFileStatus` tries to find the request file, and if found, returns the `size`, `modified_time`, and `creation_time`.

Decisions and Tradeoffs

The main decisions I had to make in part 1 was just how much or how little information to include in each RPC call. For `StoreFile`, I decided to go for an empty response from the server. I thought about including a `bool` to indicate success and maybe a `string` for a possible error code or status message, but ultimately decided to go with the simplest possible response and to leave error/status code handling to the surrounding code. While the tradeoff is the error/status handling can't be done within the message, the benefit is that we can lower the memory footprint and speed up/simpify the transfer process. I made the same decision for the `DeleteFile` response, as well as the `ListFiles` request.

After storing/deleting, the client doesn't really need any information from the server other than whether the operation succeeded or failed, which can be handled outside of the message itself. Similarly, when a client calls `ListFiles`, it doesn't need to pass any information to the server, since the server will always return the same response (in contrast to a method like `Fetch`, where the client has to specify a file name).

Other than that, I also considered storing the `ListResponse` using a `repeated` type instead of `map`. Either would have worked here. Using `repeated` is basically like using an ordered list or array, while using `map` is like using a `dict` with key-value pairs. I ended up choosing to use `map` for the improved performance due to key-based access; the file names can serve as unique keys to quickly set and get the file sizes we need. However, if we use `ListResponse` sequentially, `repeated` would have better performance, so that was the main tradeoff. I think if I knew the client would use the `ListResponse` sequentially over using it to index and access specific file sizes, I would choose to use `repeated` instead of `map` here.

Flow of Control Graph

This flow of control graph shows the request and response types for the different methods. The server runs indefinitely, and different client requests trigger different responses, as detailed above in the `Design` section.

Testing

To test my Part 1 implementation, I ran the server, then issued different client requests to test normal cases and edge cases. I used three different types of files: image files, `.txt` files, and pdfs. I also created small files, i.e. `.txt` files with just one letter, and also downloaded high-resolution images (tens or hundreds of megabytes) to test large files. I put each type of file into the client but not the server, and tested storing them. I then did the opposite, putting them in the server but not the client to test fetch file. I tested deleting each file, listing the files, and also getting the file statuses of different files.

For each operation, I also tested files that did not exist (storing, fetching, deleting, and getting status of non-existent files). I also added a `sleep` into the server in order to test that the deadline was being handled correctly.

Part 2

Design Designs and Tradeoffs

Part 2 of this project builds upon the foundation created in Part 1. I reused the methods implemented in part 1, but modified some of them slightly and also added new RPC methods to support client-side cache-ing as well as the possibility of mutiple clients making interleaving requests.

Handling Cache-ing

The main changes I made to the existing RPC methods was to add `crc` and `mtime` to `StoreRequest`, and add `crc` to `FileStatusResponse`. This is because the cache-ing in my implementation relies heavily on `crc` to check whether or not the client or server has the same data for each file; if the `crcs` are not equal, then we check the modified time to determine what action to take.

One of the main new methods I had to flesh out was the `HandleCallbackList` method, which is called wheneer an asynchronous request is made to the server. This method is in charge of keeping the client and server files in sync. It gets a list of the files in the client's cache, and compares it to what it finds in the server. If the file exists in both the server and the client, then it checks the crcs; if equal, we assume they're in sync, and do nothing. If the crcs are not equal, we check the modified times; if the client was modified more recently, we call `store` to update the server; if the server was modified more recently, we call `fetch` to update the client. Otherwise, if a server file exists that the client doesn't have, we call `fetch` to get it; if the server has a file that the client doesn't, we call `delete` to delete the file from the server. Finally, if the client has files that the server doesn't, we call `store` to update the server.

An alternative to putting all this logic into `HandleCallbackList` would be to more granually handle these checks inside respective methods like `fetch` and `store` and `delete`, but having this logic inside `HandleCallbackList` helps us avoid the memory overhead associated with calling separate methods. I do have checks inside of each function (i.e. comparing crcs/mtimes in `Fetch`, `Store`, etc.), but having the extra checks inside the `HandleCallbackList` layer just helps avoid calling these methods when unnecessary.

To support `HandleCallbackList`, we have `ProcessCallback` on the server-side, which is very similar to `List`; here I have it pass a list of all its files to the client with the important attributes the client will need to compare files, including file name, size, mtime, ctime, and crc. I chose to have the server pass the client the info it needs to make these checks instead of checking in the server for the sake of simplicity; I wanted to cleanly separate where the cache-ing checks would occur, and I decided it made much more sense on the client side, since the client directory is in itself the "cache".

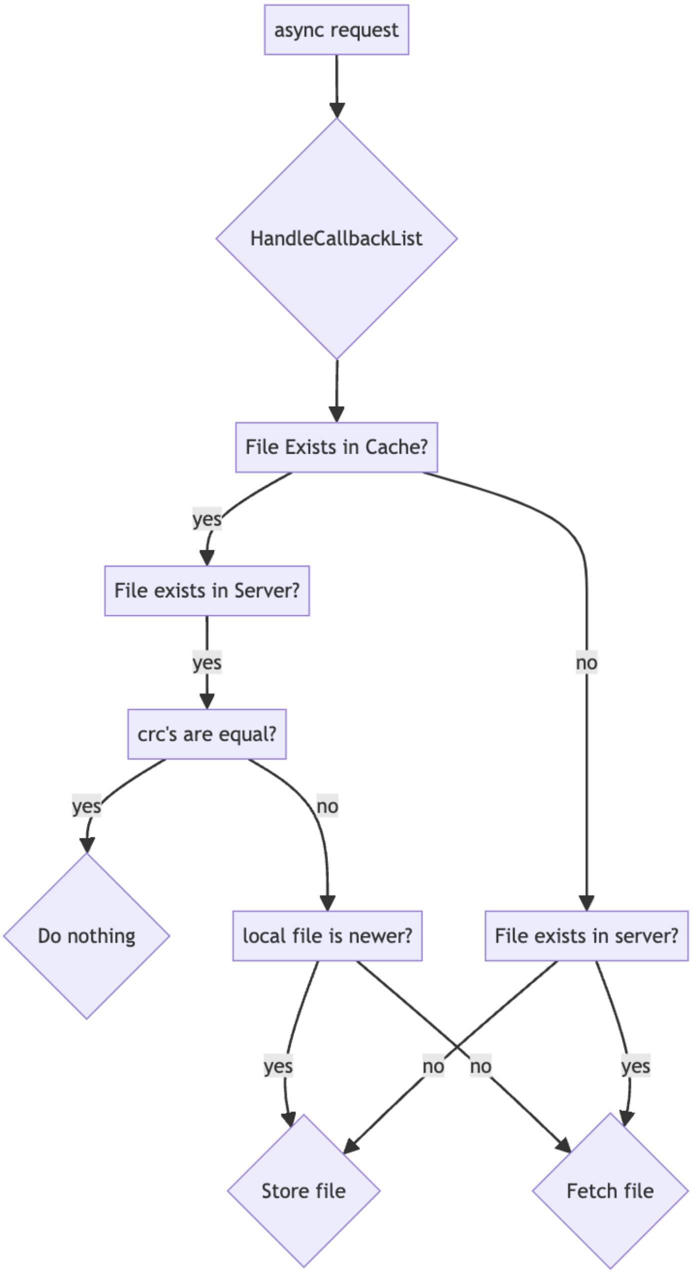
Handling Write Locks

Other than the cache-ing logic, the other main functionality we had to support in Part 2 is handling requests from multiple clients simultaneously without their requests interfering or overriding one another. Multiple readers are generally fine; we don't need to lock reads like `list` or `fetch` requests. However, `store` requests require synchronization, as multiple writers on the same file at once can lead to inconsistencies.

One possibility was to create a mutex on each file; only the owner of the mutex could edit that file. However, in order to avoid creating a large number of mutexes (which could be quite expensive computationally and memory-wise for a large number of files), I decided to create a `map` called `file_owner` to keep track of file locks and their owners, and a singular mutex (`file_owner_mutex`) for editing this map. Only having one mutex helps to decrease the complexity of synchronization mechanisms. However, the tradeoff for this is that it creates a bottleneck at the step of editing the `map`; if many clients try to acquire/release their lock at once, they will have to wait their turn, as only one can edit this `file_owner` map at a time.

This new mutex would support the new rpc methods: `RequestWriteLock` and `ReleaseWriteLock`. Before a `store` operation, a client will request a write lock, and the server will check whether the file either does not currently have an owner, or whether the client is already the owner of the file. If either is true, the server will return success of `true`, or will deny permission otherwise by setting success to `false`. If the client cannot obtain the lock, it will return `RESOURCE_EXHAUSTED`. Otherwise, if the client obtains the lock, it will proceed storing as usual. Finally, it request to release the lock, and on the server side, it will obtain the `file_owner_mutex` then empty the correct spot in the `file_owner` array.

Flow of Control Graphs



This flow of control graph shows the process of `HandleCallbackList` determining whether to fetch, store, or do nothing based on the state of the server and the cache.

Testing

To test Part 2, I started with a single server and a single client to test the basic cache-ing functionality. Similar to Part 1, I used files with varying sizes and types to test storing and fetching. Then, I tried files that existed in the client but not the server, files that existed in the server but not the client, and files that could not be found in the server to check whether the appropriate fetch/store methods were called and would keep the two in sync. I would also store duplicate text files, then modify the file on either the client or the server to see if the changes would propagate. To check whether cache-ing was happening for the same files, I would store the file then make sure `FetchFile` was not called on the server side.

After this, I mounted a second client and repeated the steps above, but I would alternate steps between clients to see if synchronization would happen between the server and both clients.

To test the write lock, I used the DFS stress test from the `6200-tools` repository (downloaded from https://github.gatech.edu/cparaz3/6200-tools). This test simulates a heavy concurrent workload with many clients making requests on the same files at the same time.

References

- GRPC C++ documentation: https://grpc.github.io/grpc/cpp/index.html
- basic GRPC C++ tutorial: https://grpc.io/docs/languages/cpp/basics/
- proto3 documentation: https://protobuf.dev/programming-guides/proto3/
- GRPC status codes: https://github.com/grpc/grpc/blob/master/doc/statuscodes.md
- GRPC deadlines (C++ code): https://grpc.io/blog/deadlines/
- P4L1 lecture material on RPC's
- `pr4` repository, including all the diagrams in the docs section: https://github.gatech.edu/gios-spr-24/pr4/tree/main/docs
- `grpc` examples: https://github.com/grpc/grpc/tree/v1.62.0/examples/cpp
- `grpc` protos: https://github.com/grpc/grpc/tree/v1.62.0/examples/protos
- `6200-tools` for testing: https://github.gatech.edu/cparaz3/6200-tools
- All the slack threads in channel `#project4` of the `Georgia Tech` Slack workspace. I read almost every thread in this channel to get my bearings and also to solve bugs in my code.