Project README file

This is **YOUR** Readme file.

Project Description

Please see Readme.md for the rubric we use for evaluating your submission.

We will manually review your file looking for:

- A summary description of your project design. If you wish to use grapics, please simply use a URL to point to a JPG or PNG file that we can review
- Any additional observations that you have about what you've done. Examples:
    - **What created problems for you?**
    - **What tests would you have added to the test suite?**
    - **If you were going to do this project again, how would you improve it?**
    - **If you didn't think something was clear in the documentation, what would you write instead?**

Known Bugs/Issues/Limitations

Please tell us what you know doesn't work in this submission

References

Please include references to any external materials that you used in your project

Part 1 - Building a RPC protocol service

For part 1 of this project, I had to build a series of different remote procedure calls with message types that does a series of different functions on files from a remote server. The functions enabled are fetching a file from the server, storing a file onto the server, listing all the files available on the server, and getting available attributes on a file. Before starting on those functionalities, I experimented with a simple SayHello function as shown in the tutorials that were listed on the project readme. Doing this, helped me better understand how the server and client connect and talk to each other through this rpc system with protocol buffers. After this, I moved on to working on the functionalities. The functionalities were simple in nature, as I've done similar implementations just in a different setting on previous projects. The message types I created were mostly unique to each rpc call, but I tried to make some generic enough where they can be reused. The store takes in a stream of bytes of the file to be stored on the server and returns the file name the content was stored to on the server side. The fetch was pretty much the reverse of the store, it takes in a file name which is passed into the server and then the server gets the file by that name and returns the content of the file to the client. The client in turn stores that file content in the file of the same name on the client side. The delete function does as its name suggests, it takes in a file name and gives it to the server which in turn deletes the file from the server. Now it returns an Empty message, which means it returns nothing. The only thing notable from the return to the client is the status of the request returned. The list function takes in an empty message because it needs nothing. The client would use the stub to request the list from the server, and the server would return a list of files available on the server. Now the return message type uses the key word repeated, which basically enables the message to contain a list of the same message type, which in my case was FileName which just stored the file's name. The last function was the GetStatus which the client takes in a file name and gives that to the server. The server then returns relevant information about the file requested such as file size, creation date, and last modified data which were all available as file stats for c++. For most of this part, I followed this diagram: https://github.gatech.edu/gios-spr-21/pr4/blob/master/docs/part1-sequence.pdf.

All the methods before the service stub calls the method on the server side, it adds a deadline in which serves to determine whether not the deadline was exceeded for a call. I would first define the methods in the proto file and then later have methods of the same name on the servernode file. There I would fully define what each function does.

To test this part, I just manually tested all the methods with a variety of files that I got off the web, created and from the sample files folder. For store I first tested a small file to make it was storing it on the server correctly, then after that was determined to be true, I would store a larger file, for example thos jpg files from the sample files folder. I took the same approach for the fetch and delete methods. For the list methods, I had the list method on the client side print out the result that was given back from the server to determine that it was listing the correct files only and not any extra folders or anything. For the get status method, I would get the status of the method and then just compared the values of what was returned to actually properties of the file to make sure it matches.

The first problem I ran into was not really knowing where to start with this part. The RPC made sense in my head, but since this was my first time actually implementing it, I didn't really know where to begin. So after going through Piazza, after seeing another classmate's suggestion, I decided to start with a simple program that I could do using RPCs. With that in mind, I decided to recreate the SayHello program for RPCs as shown in the tutorials and examples. AFter completing that, the task seemed less duanting. Another problem that I ran into was with storing and fetching. When I was first starting, I ran into a lot of problems with getting the correct bytes back from either the client or server. I was receiving less than I was sending. Turns out with the message protocol buffers, when I set a value in the message, I can specifiy the size of that message field. When I started doing that, I began to get the correct bytes sent from both the client and server. One more problem that I ran into was getting the time functionalities. Since c++ didn't really have any standard libraries for time, I had to use the one provided by the google protocol buffer library. This was all new to me, so it took a while to understand how to use it. For example, the we had to set deadlines for the RPC calls, so the deadline was set by getting the current time using the google protocol buffer and then adding the set deadline time to it then adding that as a option to the client context. Later in the project for the file stats part, I had to convvert the standard time in c++ to the one used by google protocol buffer because that was the only available time type for the message buffers.

Part 2 - Completing the Distributed File System

For part 2 of this project, we had to build and completely a very basic distributed file system or DFS. The main difference between this part and part 1 is that the rpc calls made in part 1 is synchronous, whereas for part 2 its asynchronous. This will allow the server to listen and communicate with whichever client is connected and made a change on their mount path. This is the mount command for this part. The client uses the handle callback method and inotify method to check if the mounted directory of the client has any changes. Depending on the change made, the respective rpc method will be called to the server from the service stub. The existing functions such as fetch, store, list, delete, and get status were all copied over from part 1. The main difference being that, all those functions were edited to be asynchronous, meaning mutex locks were added around the functions and only one client can access the file at a time. If the client is trying access the same file that another client has a lock on, they are denied access. The file can only be access when the client can get the lock. To determine whether not the client has the lock to the file, I used the std container map with a key value pair of the file name and the client id. The getwritelock function would check the file name that the client is requesting and check that the filename in the map points to the client id of the client. If it does, then the client already has the lock, if the filename exists in the map

but points to a different client id, that means another client currently is accessing the file and thus access is denied. If the filename isn't in the map, the filename and client id are added giving the client access to the lock. This check is done on all those file methods to prevent multiple clients trying to write to a file at the same time. In the callback, we had to do a variety of checks to determine which method gets called. If the client mount path doesn't have the file, the fetch method is called to get it from the server. Otherwise if the server modified time is greater than the client modified time for the file, it fetches, and if its the vice versa condition, it stores the file from the client onto the server. For most of this project, I followed the design laid out by this diagram: https://github.gatech.edu/gios-spr-21/pr4/blob/master/docs/part2-sequence.pdf.

I used two locks for the store and fetch methods so I could have one lock mainly lock the file it was actively using and the other lock would be put on the directory to prevent other clients from making major changes to the directory.

For this part, I took the same first approach as for part 1, as these methods should still work as expected and not be broken. Once those tests were completed I moved on to testing the writelock and making sure that functionality was working. Since I was only testing with one client, to fully test the get write lock method I would hardcode in some values in the server side method to add some key pair values to the map first so when the client makes a call, it would return Resource exhausted and exit. I did this testing method to test both the getwritelock wrapper around the store and fetch methods. To test mount, I would start out with all the files from sample files folder and move them into the client folder and then would do a variety of tests, such as moving them out, deleting them and so on. I also put in a text file on the client side and edited that file to make sure the client was updating the server with the updated version of the text file.

One problem ran into was the shared mutex and using it. At first I tried to use the shared_mutex, but it kept throwing me errors when I tried compiling the code, when I switched to using shared timed mutex, all those errors went away, and it started working. One of the very first problems I ran into was trying to figure out how to implement the async calls needed in the callback in both client and server. After implementing the code in the client, I didn't implement the callback properly in the server and thus the service was not working as expected. It wasn't till I took a step back and looked at the server code and realized there was a process callback method that I didn't implement. Going off that, I wasn't fully sure on how to test the mount command till I actually read through a piazza post posted by one of my classmates. Another problem I ran into was how I was opening the file when I was prepping it on the client for fetch or on the server for store. I was getting alternating checksums for the same file when I tried to store or fetch multiple times in a row. I later figured out that was due to me opening the file too early to be written into and when it was opened and then later closed, all the content in the file was lost if the checksum was the same. That caused two of the tests on gradescope to fail.

Recommendations

In the test suite, I would've added more variety testing of the methods since this is an asynchronous service. Doing this will fully test the DFS and determine whether or not its working as expected. Maybe more edge cases such as a client trying to store the same file another client is trying to fetch from the server or a client trying to delete a file while another client is trying to store/fetch it at the same time.

I think for part 2, the documentation on how the mount command wasn't very clear. Like I mentioned earlier, it wasn't till I read a classmates piazza post that I got a good understanding of what the mount was doing and how to actually trigger those callbacks. The student's response on that piazza post was very well documented that it was easy to read and understand at the same time.

References

I used this repo to get started with the SayHello RPC programs: https://github.com/grpc/grpc/tree/v1.35.0/examples/cpp/helloworld. Message buffers: https://developers.google.com/protocol-buffers/docs/reference/google.protobuf. time utils from google: https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.util.time_util. More RPC understanding: https://grpc.io/docs/languages/cpp/basics/. how to trigger callbacks: https://piazza.com/class/kjwcg2w7lw87li?cid=1571.