Project 4 README

Spring 2023

Shide Qiu

Part 1 Understanding

In the first part of the assignment, several file transfer protocols using gRPC and Protocol Buffers need to be developed. The system should be able to handle both binary and text-based files.

The main challenge is to get familiar with gRPC service and implement following operations:

1. Fetch: fetch a file from a remote server and transfer its contents via gRPC.
2. Store: store a file to a remote server and transfer its data via gRPC.
3. Delete: delete a file on remote server.
4. List: list all files on the remote server including file name and modified time.
5. Get Status: get the attributes (size, modified time, and creation time) for a file on the remote server.

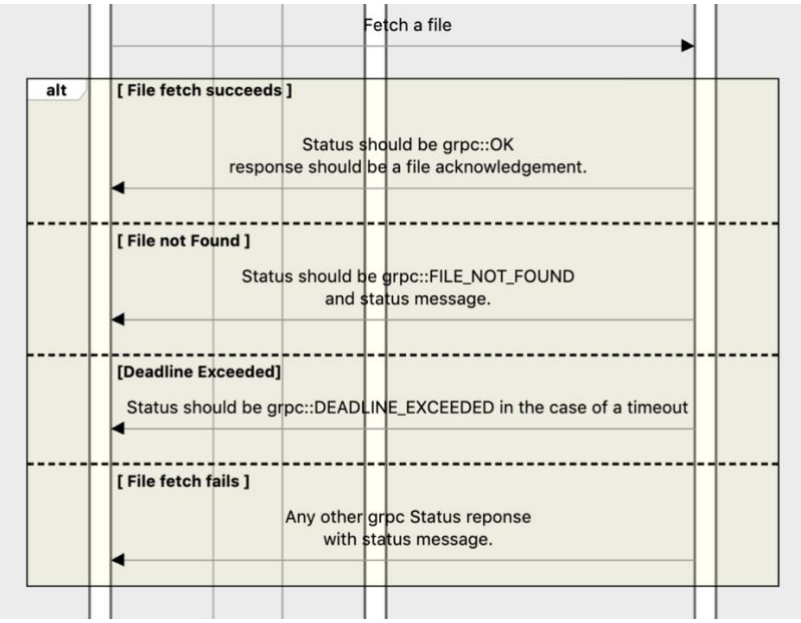
The client should be able to request each of the operations described above and the server will respond to those requests using the gRPC service methods specified in the proto buffer definition file. Also, the client should recognize when the server has timed out.

Part 1 Implementation and Testing

The workflow is clearly described in the “part1-sequence.pdf”. The fetch, store, delete, list, and get status operations are described in details below.

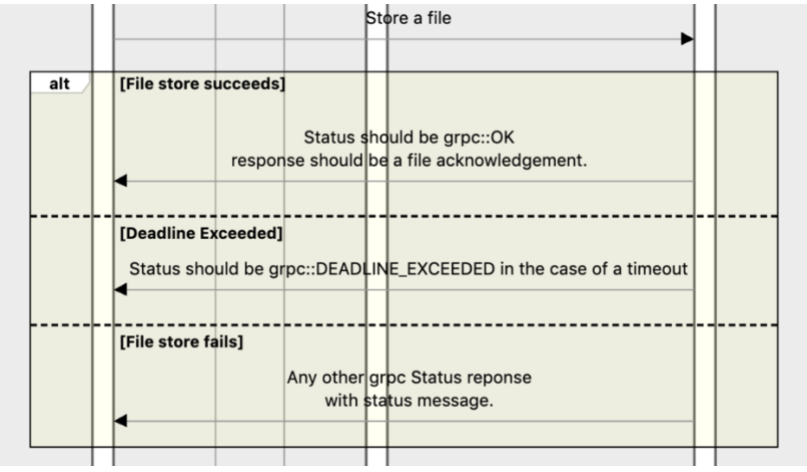
Fetch

Once the fetch request is called, the client will setup a timeout deadline and then start to send the fetch request containing the file name to the server. Once the server receives the request, it will firstly check if it has the file. If it doesn’t have the file, it will send a NOT_FOUND status back to the client. If it has the file, it will read the file and write a sequence of messages to send the file to the client. Since the gRPC message size is limited to 4MB by default (the file size could be larger than 4MB), server streaming RPC is used here. Writing a sequence of messages has been implemented in previous projects before so that we will not describe its details here. In each message, it will contain the file name, message size, and the content. Also, the server will check if the client’s timeout is triggered in every message. If the deadline is triggered, the server will send DEADLINE_EXCEED status to the server and stop writing messages. If the deadline is not triggered and all messages are sent, the server will return OK status to client. On the client side, it will receive and store the file if the status from the server is OK. If the status is not OK, it will not store the file and return the status received from the server.



Store

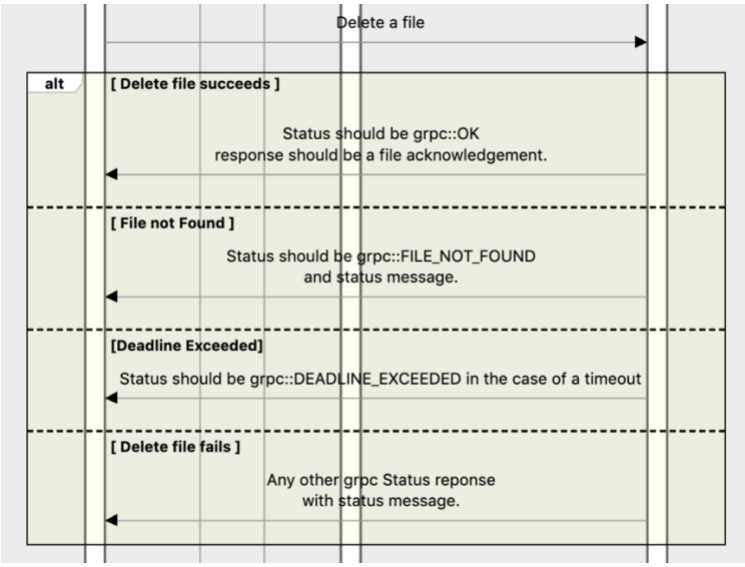
Once the store request is called, the client will setup a timeout deadline and then start to send the file to the server. Since the gRPC message size is limited to 4MB by default (the file size could be larger than 4MB), client streaming RPC is used to write a sequence of messages and sends them to the server. In each message, it will contain the file name, message size, and the content. Once the client has finished writing the messages, it waits for the server to read them and return its response. On the server side, it will firstly check the timeout deadline. If the deadline is triggered, it will return DEADLINE_EXCEEDED status to client. If not, it will store the received file and send “OK” status back to the client. The client will return the status received from the server.



Delete

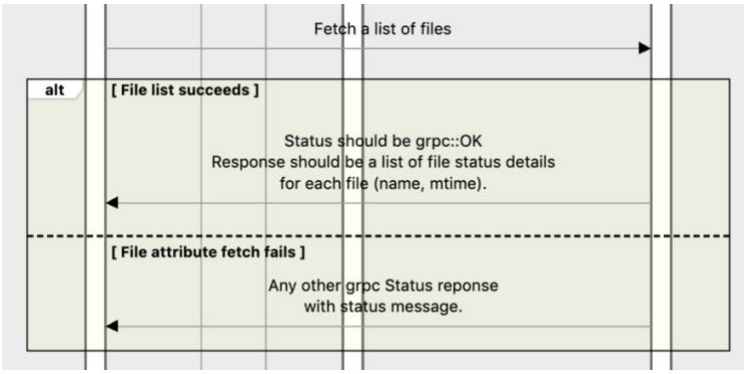
Once the delete request is called, the client will setup a timeout deadline and then send the delete request containing file name to the server. Unary RPCs are used since the client sends a single request to the server and gets a single response back. Once the server receives the request, it will firstly check the timeout deadline. If the deadline is triggered, it will return DEADLINE_EXCEEDED status to

client. If not, it will then check if it has the file. If not, it will send NOT_FOUND status back to the client. If it finds the file, it will delete the file and send OK status back to the client. The client will return the status received from the server.



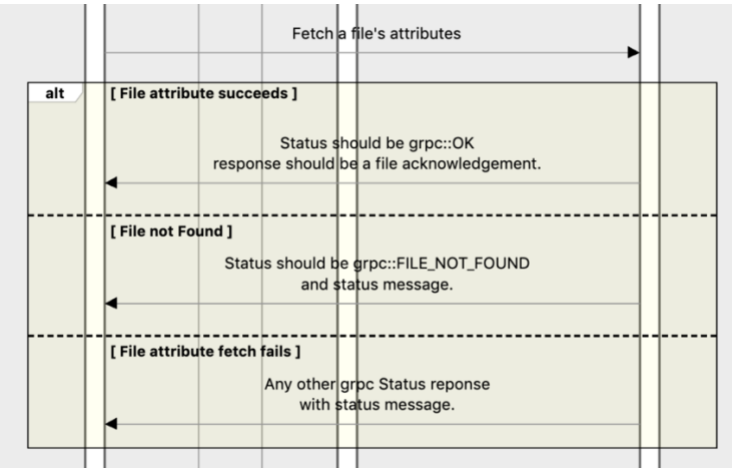
List

Once the list request is called, the client will setup a timeout deadline and then send the list request the server. The Unary RPCs are used here. Once the server receives the request, it will firstly check the timeout deadline. If the deadline is triggered, it will return DEADLINE_EXCEEDED status to client. If not, it will send all files' name and modified time under the directory back to the client and return the status OK. The client will return the status received from the server.



Get Status

Once the get status request is called, the client will setup a timeout deadline and then start to send the get status request containing the file name to the server. The Unary RPCs are used here. Once the server receives the request, it will firstly check the timeout deadline. If the deadline is triggered, it will return DEADLINE_EXCEEDED status to client. If not, it will send the file size, modified time and creation time back to the client. It will also return OK status to the client. The client will return the status received from the server.



The test is done in the VM. A couple print functions were added in both client and server side to print out the request and response. With those print functions, we launched the client while the server is running to check if the terminal messages was valid. It will make sure the gRPCs are set up and the messages are transferred between client and server as expected. Also, I created multiple files in client directory and server directory. All five operations are called separately and at the same time monitoring the server and client directory to make sure each operation is worked as expected. Then I created a large file in server and client, and tested the store and fetch operations to confirm that the streaming RPCs are worked as expected. In the end, I checked the timeout feature by adding sleep function on the server side before the server check the timeout. The status returned by the client is DEADLINE_EXCEEDED and it confirms that timeout is handled asa expected.

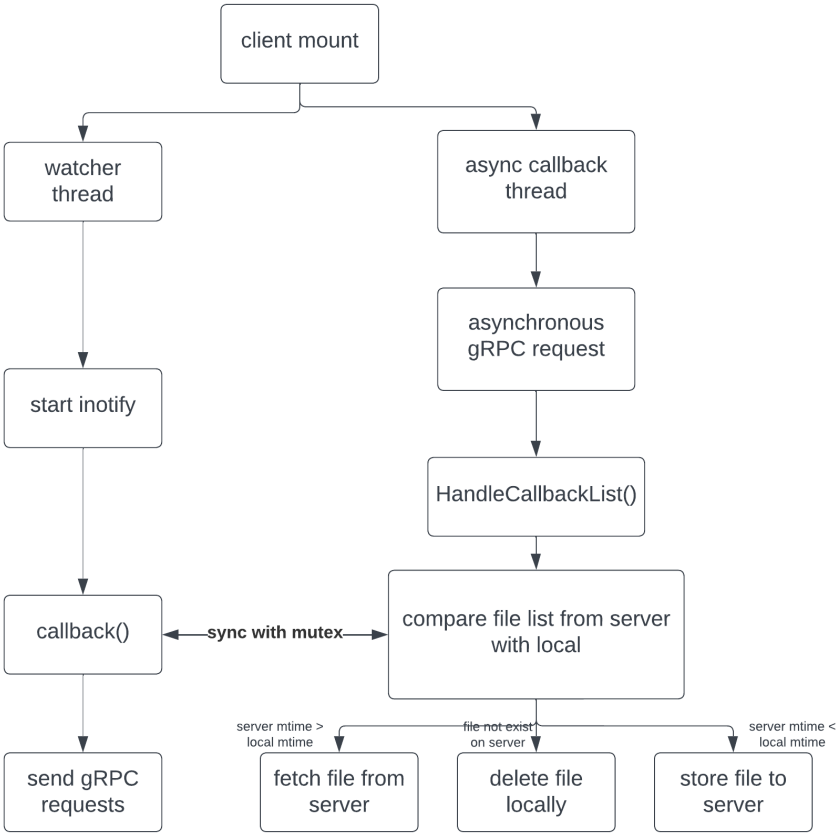
Part 2 Understanding

The part 2 is based on part 1, we will apply a weakly consistent cache strategy to the RPC calls created in part 1. Specifically, we will use a simple lock strategy on the server side to ensure that only one client may write to the server at any given time. Also, an asynchronous gRPC call need to be implemented in part 2. It will be used to allow the server to communicate to connected clients whenever a change is registered in the service. Therefore, the main challenge for part 2 is to build a mutex for writer locks and the synchronization between clients and server.

Part 2 Implementation and Testing

I used the workflow described by Kevin Anderson Lin on Piazza to implement the part2. The client-side workflow is plotted in the figure below. In addition to gRPC calls, each client includes a file watcher and an async callback. Once a client is mount, it will initiate an inotify watcher thread and an async callback thread. The inotify watcher monitors the client directory for any file changes. If it detects any changes, it will send gRPC request to the server to for update. The async callback thread requests

the server to notify the client of changes that the server knows about. If the server finds any changes in the server, it will invoke the callbacklist(). The client may fetch the file from the server, delete the file locally, or store the file to the server depending on the file status on client and the server. Such two thread are running concurrently so that there is a mutex implemented to synchronize these threads. On client side, the mutex is acquired before the callback() function and released after the gRPC request is sent. On the server side, the mutex is acquired before comparing file from server with local and released after the operations. Therefore, only one thread can make the update at one time.



As for the writer lock, it is implemented on the server side. There is a write_map maintained on the server to store the map of file name and client id. When the request write lock function is called, it will pass file name and client id as parameters. The function will check if the file name and client id pair is stored in the write_map. If a different client id exists, it will return RESOURCE_EXHAUSTED status. If the stored client id matches with the requesting client id, it will return the OK status and give the lock. If the stored client id is null, it will store the requesting client id with the file name as a pair. Then it will return the OK status and give the lock.

The fetch, store, delete, list, and get status operations are modified based on the part 1 implementation. The details are described below.

Fetch

On the client side, it will send a get status request containing file name to the server. On the server side, the get status is similar with what implemented in the part 1. However, it adds another crc

attribute. The crc is used to determine if a file has changed between the server and the client. The crc function is already implemented in the code. Once the client receives the file status, it will call the crc function to check if the file in the server is different with the file in the client. If there is no difference, it will return ALREADY_EXISTS status. If they are different, the client will send a fetch request containing file name to the server. The fetch operation on the server side is same with what implemented in part 1. The client will handle the response from the server as what it does in part 1. The timeout deadline is handled in the same way in part 1.

Store

On the client side, it will try to get the write lock first. If it fails to get the write lock, it will return the RESOURCE_EXHAUSTED status. If it gets the lock, it will get the crc of the client file and send the file to the server with file name, file size, file content, crc, and client id. On the server side, it will try to get the write lock first. If it fails, it will return the RESOURCE_EXHAUSTED status. If it gets the lock, it will then check the crc of the requested file on the server and compare it with the client's crc. If two crc are the same, it means they are the same file, and it will release the write lock and return ALREADY_EXISTS status back to client. If two crc are not the same, it will store the file on the server, release the write lock, and return OK status back to the client. The client will handle the response from the server as what it does in part 1. The timeout deadline is handled in the same way in part 1.

Delete

On the client side, it will try to get the write lock first. If it fails to get the write lock, it will return the RESOURCE_EXHAUSTED status. If it gets the lock, it will send the delete request containing file name and client id to server. On the server side, it will try to get the write lock first. If it cannot get the lock, it will return the RESOURCE_EXHAUSTED status. If it gets the lock, it will delete the file, release the lock, and return OK status to client. The client will handle the response from the server as what it does in part 1. The timeout deadline is handled in the same way in part 1.

List

The list operation is the same with what implemented in part 1.

Get Status

For the get status operation, the client-side implementation is the same with part 1. On the server side, the response has one more attribute crc. Thus, it adds get crc function of the requested file and add the crc into the response. All the other parts are the same with part 1.

The test is done in the local VM. A couple print functions are used for debugging purpose. Firstly, the write lock feature is tested by printing out the file name and client id whenever the lock is acquired. The terminal messages on both server side and client side are compared to understand which write lock is acquired. As for the synchronization between clients and server, it is tested when the client is mount on the running server. I deleted and added a file in server directory and check if the same file is removed or added in client directory. Also, I did the same operation in client directory and monitor the server directory to confirm it works as expected. In the end, I updated the file in client directory and monitor the server directory to check if the update is synchronized. Another update was done in server directory and the client directory was monitored for the synchronization.

Reference

1. GRPC C++ https://grpc.github.io/grpc/cpp/index.html
2. Protocol Buffers Documentation https://protobuf.dev/programming-guides/proto3/
3. gRPC C++ Examples https://github.com/grpc/grpc/tree/master/examples/cpp
4. gRPC and Deadlines https://grpc.io/blog/deadlines/
5. gRPC Basic Tutorials https://grpc.io/docs/languages/cpp/basics/
6. Piazza https://piazza.com/class/lco3fd6h1yo48k/post/1243