Part 1

Takeaways of Requirement

Define RPC service and message type in `.proto`

Setup service and deadline

Implement file operation via gPRC in client and server.

Flow control

Store: check file exist in client -> client setup deadline and writer -> send header (file name) to server by writer -> server check valid file name (not empty) -> client write file in chunks to writer -> server read file chunks from reader -> transfer done, return status code -> client check code for specific response (CANCELLED, DEADLINE_EXCEEDED , OK, ...)

Fetch: client setup deadline and reader -> server check file exist in server -> send header to client -> client check valid header -> server send file in chunks -> client read file in chunks -> transfer done -> server return status code -> client check response code

Delete: client setup deadline and service -> server check file existence -> remove file in server -> client get response code

Status: client setup deadline and reader -> server check file existence -> server read file status ans write to writer -> client reader file status from reader -> check response code

List: client setup deadline and reader -> server check directory existence -> server open directory and read file iteratively -> write each file status to writer -> client read return information -> check response code

Implementation and Decision

Proto:

```
rpc StoreFile (stream FileContent) returns (Result) {}
rpc FetchFile (Files) returns (stream FileContent) {}
rpc DeleteFile (Files) returns (Result) {}
rpc ListAll (NoArgs) returns (stream Result) {}
rpc FileStats (Files) returns (stream FileStatus) {}
```

Follow the gPRC basic tutorial to setup services and message type.

Store file to server have to contain file content in the request which may needs multiple time message transfer. in this case stream keyword is required to do so. And it work with reader and writer.

In the implementation, the structure is same setup deadline and service (by reader/writer) then process the request/received data. So I work on the Store at first. Than the following operations can use the same structure by change parameter and data process as necessary.

Store:

use `access()`, one way from Check file existence list in reference, to check the file existence

Set up deadline is follow the provided resource, but there is an error *missing template arguments before 'deadline'* try to apple the equation directly as parameter just solved the issue.

```
ClientContext context;

// std::chrono::time_point deadline = std::chrono::system_clock::now() +
std::chrono::milliseconds(deadline_timeout);

//missing template arguments before 'deadline'

context.set_deadline(std::chrono::system_clock::now() +
std::chrono::milliseconds(deadline_timeout));
```

Setup service with writer is no difficult by following the basic tutorial example.

To transfer file content, I use the same piece code from previous project. Read file size from `stat` than send in chunks.

Server receive file length before the actual content transfered and the chunk size is same at both side, so that server can track the transfer process.

By checking the error code chart we can see that `CANCELLED is 1`, `DEADLINE_EXCEEDED is 4` and others. in this way we can return status by checking the code.

In sample way **fetch** is swap the store code in client and server.

Delete only needs the file name at once when the PRC service be called. there no stream keyword needed.

List all file from server have to give the directory path, but it is a variable in server so that request parameter could be null. Since Null is reserved keyword, it has to to some other name as message type. To get files from given directory, I follow a example from IBM. One problem is that the IBM example using **S_ISDIR** is not meet the requirement of our project. I searched it and it come up with **S_ISREG**. former relate to directory, later relate to regular file.

Get file **status** from server, I choose to use reader/writer to transfer those status so stream keyword been used in response space. Actually it is not necessary because it can directly retrieve those data from define message structure.

One required information is create time of the file. the instruction named `crc` but it is not a available attribute from file stat. there is a `st_ctime`. however it cannot direct parse in to the set_xx() method and cause convert error. it has to convert to one of four support type of proto, int, than set it up.

Test

Just use provided command to test locally. `./bin/dfs-server-p1` `./bin/dfs-client-p1 [operation] [filename]`

I wasn't test every every response code. I test OK and NOT_FOUND to ensure the operation is correct which infer that other might be right. The possible problem is put in wrong place, it can be fixed after getting feedback from GradeScope.

To test the function work properly, I use the provided images. The problem I found that the program doesn't read the `sample_file` directory. So I copy the image to upper directory `server`. Than I just check the image after the related operation done.

For list and status operation, I print out relate information in server and client to determine the functionality.

Reference

gPRC: https://grpc.io/docs/languages/cpp/basics/

Deadline: https://grpc.io/blog/deadlines/

Check file existence: https://zhuanlan.zhihu.com/p/180501394

ifstream: https://www.w3schools.com/cpp/cpp_files.asp#:~:text=C%2B%2B%20Files%201%20C%2B%2B%20Files%20The%20fstream%20library,class%2C%20and%20the%20name%20of%20the%20file.%20
Check directory existence: https://www.w3schools.blog/check-if-directory-exists-cpp

Get file from directory: https://www.ibm.com/docs/en/zos/2.4.0?topic=functions-opendir-open-directory

Part 2

Takeaways of Requirement

Part 2 is based on part 1 but require write lock for each file. One file only can be modified by one client at a time.

Client has to check or update the latest file version via callback list

Call back list operation can be call in concurrent, so that it requres mutex lock .

Flow control

The flow control basically is same as part 1 but the write have to required before the service be called.

Take store as example:

check file exist in client -> client setup deadline -> **request write access** -> set up writer -> send header (file name) to server by writer -> server check valid file name (not empty) -> **server check access request record** -> client write file in chunks to writer -> server read file chunks from reader -> transfer done -> **remove the access record (lock)** -> return status code -> client check code for specific response (CANCELLED, DEADLINE_EXCEEDED , OK, ...)

If any check in server side is not passed the access lock will be erased before the return statement.

RequestWriteAccess: client setup deadline and service with required file and clientID -> server check the file existence and whether the client is free to access the file -> record to map or decline the request.

HandleCallbackList [Client] : for each file from callback list, check the existence locally, if not exist fetch it otherwise compare the local version and server version by checksum. If their checksum is different which means the two copies are not same. Then determine the latest version by modify time and update or fetch base on the compare result.

ProcessCallback [Server]: It as same as list file function from part 1 but not require to return status code. check directoty existence, open directory, get file status for each file.

Design and Implementation

To getting the access lock just direct call the **RequestWriteAccess** method. if the return status is OK than process is keep going otherwise return the error status from the called method.

InotifyWatcherCallback : this method required to handle race condition which mean we need a mutex lock. so a mutex lock at the method beginning and unlock it at the end.

HandleCallbackList [Client]: this method also require a mutex lock so that all the checks or comparsion of file from callback list is protected by a mutex.

Decisions

Same strategy that start with store function with lock than modify other function from part1.

RequestWriteAccess on client is just call the RPC service **WriteLock** like delete function.

WriteLock implementation on server using <filename, clientID> map to tracking the access lock. By checking the pair existence in the map we can determine whether the client request has authority to modify required file.

Mutex: in previous we using Pthead, in provided code there is no pthread included but give and . Based on educba example, just create a mutex variable than critical section can be protect by lock() and unlock() without initialize conditions.

CallbackList: is not used in any place in client, so that the override CallbackList is not implemented. However there is ProcessCallback in server which responsible to generate the callback list. It is no different than status function without return statement.

proto:

```
rpc CallbackList(Files) returns (FileList) {};rpc CallbackList(Files) returns (FileList) {};

message FileList {
   repeated FileStatus fstatus = 1;
}
```

From the basic tutorial only stream keyword be introduced. I try to use it at first but error report on the given code from following piece

```
 for(QueueRequest<FileRequestType, FileListResponseType>& queue_request : this-
>queued_tags) {
                this->RequestCallbackList(queue_request.context, queue_request.request,
                   queue_request.response, queue_request.cq, queue_request.cq,
queue_request.tag);
                queue_request.finished = true;
             }
```

After search there is another keyword **repeated** which working for returning list. and each repeated message type is an object in the list.

HandleCallbackList [Client]:

```
using FileListResponseType = FileList;

AsyncClientData *call_data = static_cast<AsyncClientData *>(tag);
```

This is the most difficulty part. It doesn't direct call the callbacklist service as part1 did and not have a required implemented method. It has to be figure how to getting the list. above list two line code is the key. **FileListResponseType** is desired list type, and **FileList** is defined list in proto. so we need to find the FileListResponseType somewhere. **call_data** is callback queue. it may have some useful information, by exploring the given option provided in call_data, context, reply, response_reader, and status. Only reply matches the **FileListResponseType**. Once successfully getting the list, the rest has no big problems.

Test

Since the part1 pass the test, functions in part2 except callback list should be fine if there is no error after successful compile.

However, it does report not compiled clearly error in gradescope. and there is a solution in piazza by switch two header file.

> #include "src/dfs-utils.h" //Canvas @1903
>
> #include "src/dfslibx-clientnode-p2.h"

Since callback list will be called at every time a RPC service be called it should be fine if store, fetch functions works without issue. Only relay on the feedback from gradescope.

Reference

Mutex: https://www.educba.com/c-plus-plus-mutex/

Shared memory example https://man7.org/linux/man-pages/man3/shm_open.3.html

Repeated and Stream https://learn.microsoft.com/en-us/dotnet/architecture/grpc-for-wcf-developers/streaming-versus-repeated

Suggestion

the search function in https://grpc.io is not working for me, always blank. And the tutorial not mentioned any thing about repeated. it cost lots time to find the solution. It has to be accessible somewhere in reference. (Or maybe it does have from google resource but blocked by region). instruction about callback list is better to give more hints or possible direction.