Project README file

This is **Yufeng Xing's** Readme file.

Readme Instructions

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?**

Summary Description

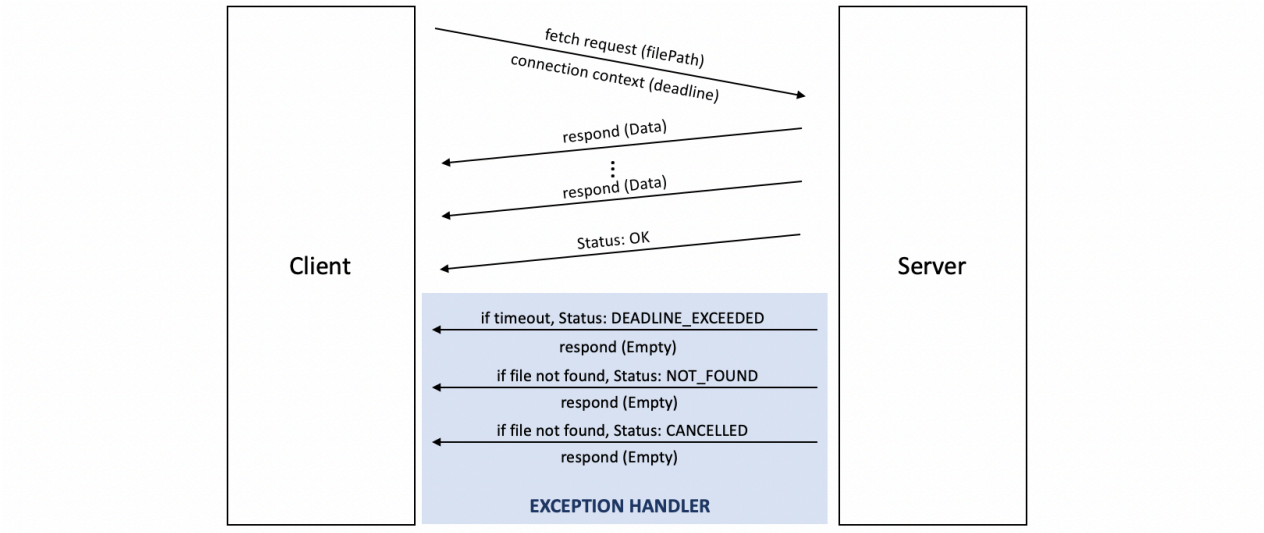
1. Part 1 - gRPC Basic Services

In this part, there are 5 methods provided by the server,

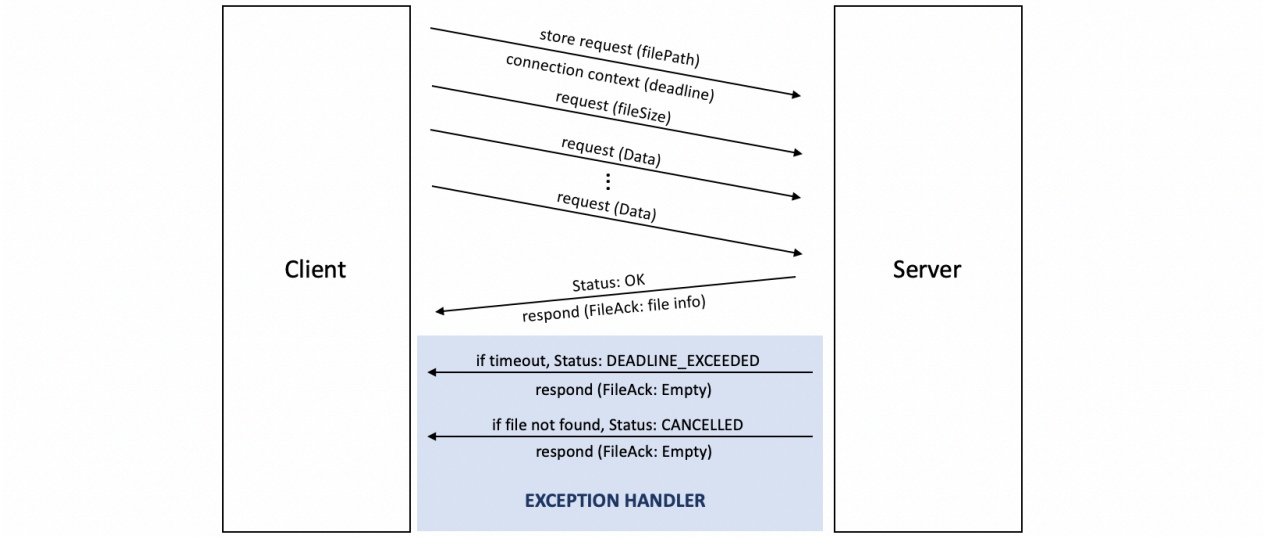
- `PutFile` is the method to store files on the server
- `GetFile` is the method to fetch files from the server
- `RemoveFile` is the method to delete files from the server
- `ListFile` is the method to list all files on the server, the listed information includes filename `fileName` and modified time `mtime`
- `GetStatus` is the method to get the status of a file on the server, the information includes filename `fileName` , modified time `mtime` , and created time `ctime`

In the `PutFile` method and `GetFile` method, because we have to get streamed data from local or the remote server, a `stream` message for file transformation called `Data` is implemented in these methods. What's more, beause we have multiple files on the server, we also have a message called `FileList` which contains a `repeated` type `FileAck` message. The `FileAck` message is defined as the acknowledgement of the file, which contains the file information `fileName` and `mtime` from the server.

When we **fetch** a file from the server, the client should first send a request message with the file name `fileName` it would like to fetch. This file name is a relative location and it should be resolved by the `WrapPath` function on the server. Then the method `GetFile` in the server stub is called and the server should check whether or not this file exists by file status calls. The gRPC's `ClientReader` should be called in the client and `ServerWriter` should be called from the server in order to maintain a send/recv data stream between the client and the server.



When we want to **store** a file on the server, we can have a similar procedure. We can use the client for sending the file, but we have to rewrite the logic of sending the data. Because in the fetching method, the `ofs` end is set to maintaining the send/recv stream. This brings us a problem because the server don't know the `fileName` and the `fileSize` information. So it doesn't know what is the file name for storage and when it could stop receiving the data. In order to deal with these problems, before we send the file data we wish to store, we have to send the `fileName` and the `fileSize` in the first two `bucket` (in our implementation, a data block of `6200` bytes in the data stream is called a `bucket`). The server will store the file and after it finishes, it will send a response called `FileAck` containing the basic file information containing the file name and the modified time `mtime`.



The implementation of **listing** the file method `ListFile` is a littile bit tricky because we use a repeated `proto3` data type for message `FileAck` so that we can list file informations for multiple files. In this method, we have to traverse the whole dictionary and output the information of all the file in this directory. We don't need to manage the nested directories because it is not required by the instruction. The `repeated FileAck` type in the `FileList` data type is named by `info` and it should be called by `add_info` for adding more data in this type. The `add_info` function will return a pointer pointing to the `FileAck` data type and we can then assign the corresponding data of each file in this type.

```
FileAck* fileInfo = respond->add_info();
fileInfo->set_name(fileName);
fileInfo->set_mtime(fileStatus.st_mtime);
```

Finally, the **delete** method and the **status** method are quite simple becaue they only requires simple request-respond scenarios. `RemoveFile` is responsible for finding the file in the request on the server, and if the file exists, the server will delete the file. The `GetStatus` is even easier because it will not modify any files on the server. It will only check whether we have a file on the server, and if we have this file, we will send the file information including the file name `fileName`, the file size `fileSize`, the file modified time `mtime` , and the file creation time `ctime`.

2. Part 2 - Distribution System

In the part 2, we have to implement a distributed system. In this part, we have to support multiple client communications with the server and we should be able to properly lock the files for synchronization. So the server in this part should be able to know if a file is already locked by some other clients and which lock the file has. To deal with these problems, we have to maintain two mappings, one is called the ID table `IDT` and it is used for mapping the `fileName` to the `ClientID`. The other table is called the mutex table `MUT` and it is used for mapping the `fileName` to the file mutex.

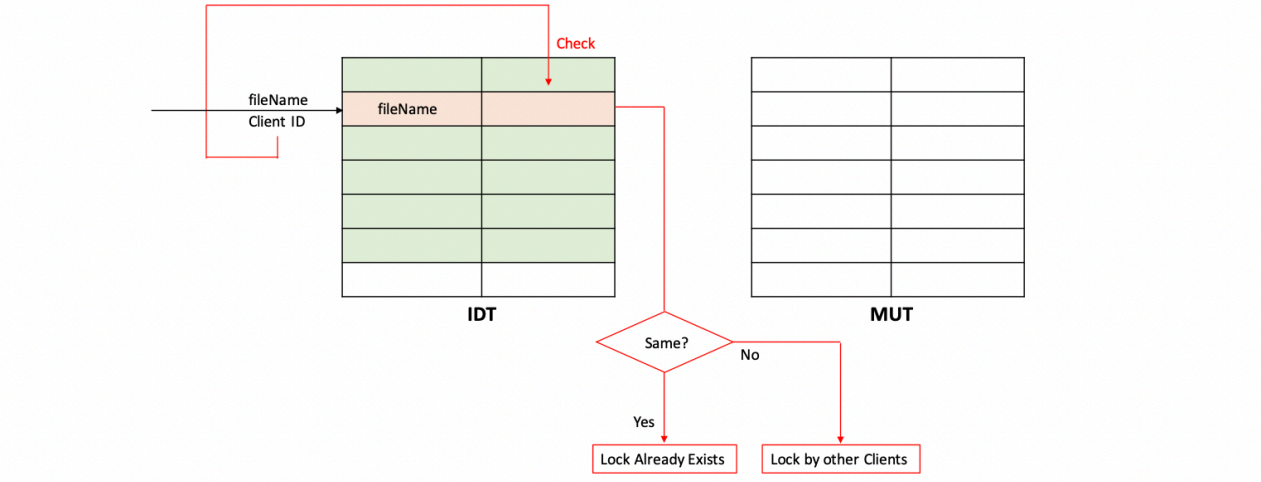
This locking method is implemented by the `WriteLock` method and it can be used to check whether the lock of a file is aquired by some other clients. This method will first check if this file is acquired by some other client.

If it is acquired by other client, we will abandon this lock acquirement. The conditions will be,

- a file name in IDT exists
- the corresponding ID in the IDT not match the current ID

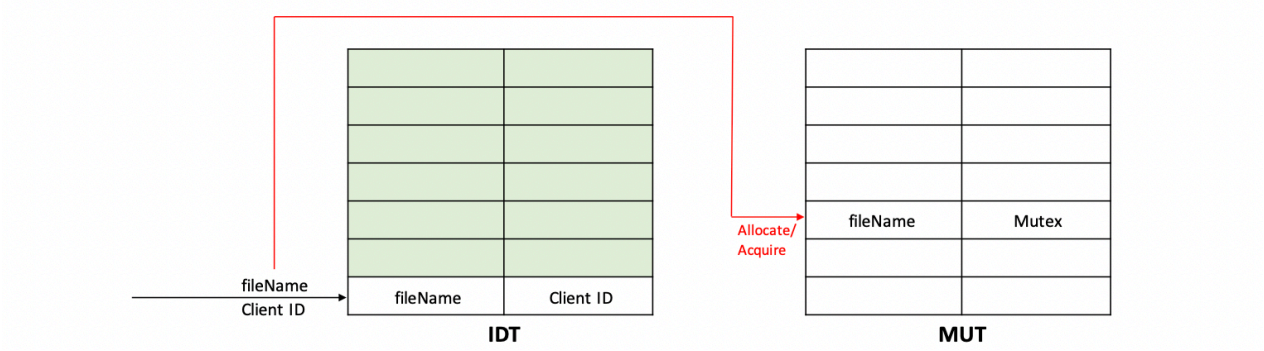
If the file lock is acquired by the current client, we will be okay.

- a file name in IDT exists
- the corresponding ID in the IDT matches the current ID. The conditions will be,



However, if this file is not acquired by some other clients, we can then allocate or acquire a lock to this client. The conditions will be,

- a file name in IDT doesn't exist



To speed up the mappings of initially existing files on the server, we can also initialize the MUT in the constructor `DFSServiceImpl` where we will assign a mutex for each of the existing file on the server.

When we change any files on the server, the `HandleCallbackList` will be called by the `inotify` and we have to find out whether we have to update the file in this method. In this method, we have to travese all the files in the current remote directory. The remote server will call `CallbackList`, which is handled by the `ProcessCallback` function in our server. It will return a list of file information on the server to the client. Then there will be three cases,

- We haven't got this file locally: fetch it from the server
- The file exists locally but the local mtime > remote mtime: delete remote file and store it to the server
- The file exists locally but the local mtime < remote mtime: delete local file and fetch it from the server

Created Problems

1. Problem 1 - Set Deadlines

To maintain the connection between the server and the client, the client has to set a deadline time. After deadline is set in the client, the client will be `cancelled` in the future schedule if the server doesn't response in a short period of time. The server should check whether the context in the client is cancelled and if there's no respond, we should reply with `DEADLINE_EXCEEDED` error code.

To implement this deadline in the client, we have to set the context deadline by `set_deadline` method,

```
using std::chrono::system_clock;
using std::chrono::milliseconds;

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

if (status.error_code() == StatusCode::DEADLINE_EXCEEDED) {
    // log
    dfs_log(LL_ERROR) << "Server timeout";
    return StatusCode::DEADLINE_EXCEEDED;
}
```

And in the server, we will check if the context is `cancelled` by,

```
// check if timeout
if (context->IsCancelled()) {
    // log
    dfs_log(LL_ERROR) << "Request expired, connection close.";
    // return error status
    return Status(StatusCode::DEADLINE_EXCEEDED, "Request expired");
}
```

2. Problem 2 - Using `dfs_log`

As it is mentioned in the project direction `readme.md` file, a logging utility `dfs_log` is provided in this assignment for tracking the system outputs and errors, and it can also be used for debugging. In my implementation `LL_SYSINFO` and `LL_ERROR` are used quite often. These log levels can be treated similar to `cout` and `endl`.

For example, if we want to simply output `Helloworld` as the system output, we can directly use the output log stream like,

```
dfs_log(LL_SYSINFO) << 'Helloworld';
```

The other debugging log levels like `LL_DEBUG`, `LL_DEBUG2`, and `LL_DEBUG3` are also used for implementing the callback functions in part 2 but in general, they are not as frequently used as `LL_SYSINFO` and `LL_ERROR`.

3. Problem 3 - File I/O

In the previous projects, we have done the file input or output mainly by `fopen` or `open` because we used C in those projects. Now, it's a better idea to use `ifstream` for file inputs and `ofstream` for file outputs because these methods are much easier to use, and it is quite useful in implementing the store file method and the fetch file method.

We can create an input file stream by,

```
ifstream ifs(filePath);
```

And then the `bytesSend` size of data should be looply read into the `buffer` (which is used to temporarily hod the file data),

```
ifs.read(buffer, bytesSend);
```

After reading this file to the last bit and send it to the other end, we have to close the input file stream by,

```
ifs.close();
```

The output stream can be used in a similar way. First, we will create an output file stream by,

```
ofstream ofs;
```

Then, we can open the out put file with the filg `ios::out` and `ios::trunc`. `ios::out` means that we will allow writing permission to that file and `ios::trunc` means that we will ignore and replace the current existing file if there is one exists and we will create a new file for writing,

```
ofs.open(filePath, ios::out|ios::trunc);
```

After writing to this file, we have to close the output file stream by,

```
ofs.close();
```

4. Problem 4 - Getting File Information

In this project, we have to check the file existence, file size `fileSize`, the modified time `mtime`, and the creation time `ctime` in an efficient way. And in C++, the simplest way to do so is by calling the `stats` function. For example, we can get the information of a file in the `filePath` by,

```
struct stat fileStatus;                    // structure for file info
stat(filePath.c_str(), &fileStatus);
```

The structure named `fileStatus` will then contain all the information we need including the `ctime` by `fileStatus.st_ctime`, and the `mtime` by `fileStatus.st_mtime`. Note that these valuse are in `time_t` type and it is actually a `int64` type. So that we can simplying using these values as the 64-byte integers.

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
```

```
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for file system I/O */
    blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

However, if the file doesn't not exist in the `filePath`, then this fuction will return `-1`. So the existence of the file can be simply checked by,

```
// if file not found
if (stat(filePath.c_str(), &fileStatus) == -1) {
    // log
    dfs_log(LL_ERROR) << "ERROR: " << filePath << " file not found\n";
    // return status code StatusCode::NOT_FOUND
    return Status(StatusCode::NOT_FOUND, "File Not Found");
}
```

5. Problem 5 - Traverse Directory

We should also be able to travese all the files in a given directory for the listing method, and then we should be able to get all the file objects in this directory. To open a given directory, we can use a `DIR` type pointer named `dir` and then call `opendir` to open the given `mount_path` by,

```
DIR *dir;
dir = opendir(mount_path.c_str());
```

Then, we have to repeatedly call the `readdir` function for getting file objects (called `entry`) from this directory until we get `NULL` by,

```
while ((entry = readdir(dir)) != NULL) {
  ...
}
```

Then, the name of each entiry can be easily get by accessing the `d_name`,

```
fileName = entry->d_name;
```

In Linux, all the file descriptors are stored as inodes and we can check the entity type we get by accessing its `st_mode` information. The `S_ISDIR` function can be called to check whether the current `entity` is a directory. And if it is a directory, according to the instructions, we cam simply skip this entity.

Problem 6 - CRC Checksum

To check whether the file content is modified, we can also use a CRC checksum. Even though the `mtime` can be used to determine if we should fetch or store, but it will be a burden because we `mtime` can be updated without any file modifications. Therefore, in order to determine whether a file has already existed, we can call `dfs_crc_checksum` which is already provided. We can get a CRC checksum result by

```
crc_checksum = (int) dfs_file_checksum(filePath, &crc_table);
```

So when we fetch or store the files, we will also send the checksum to the other end. If the CRC checksum value on the server is the same as the value on the client, the fetch or store operations will be necessary because the file is the same on both ends. Therefore, the server will send back the status code `ALREADY_EXISTS` by the instructions. However, if the checksum value is different, we have to update the file to its newest state.

Tests

Compile and Move Testing Files

```
$ make protos
$ make part1
$ make part2
$ cd ./mnt/server/sample-files/
$ cp * ..
$ cd -
```

For part 1,

```
/* Test storing the file */
// client
$ cd ./mnt/client
$ echo "Helloworld" > test.txt
$ cd -
$ ./bin/dfs-client-p1 store test.txt
// server
$ ./bin/dfs-server-p1

/* Test storing the a not existing file */
// client
$ ./bin/dfs-client-p1 store hello.txt
// server
$ ./bin/dfs-server-p1

/* Test storing the an empty file */
// client
$ cd ./mnt/client
$ echo "" > hello.txt
$ cd -
```

```
$ ./bin/dfs-client-p1 store hello.txt
// server
$ ./bin/dfs-server-p1

/* Test fetching the file */
// client
$ ./bin/dfs-client-p1 fetch gt-einstein.jpg
// server
$ ./bin/dfs-server-p1

/* Test fetching a not existing file */
// client
$ ./bin/dfs-client-p1 fetch file-not-found
// server
$ ./bin/dfs-server-p1

/* Test deleting the file */
// client
$ ./bin/dfs-client-p1 delete gt-einstein.jpg
// server
$ ./bin/dfs-server-p1

/* Test deleting a not existing file */
// client
$ ./bin/dfs-client-p1 fetch file-not-found
// server
$ ./bin/dfs-server-p1

/* Test listing the files */
// client
$ ./bin/dfs-client-p1 list
// server
$ ./bin/dfs-server-p1

/* Test getting the file status */
// client
$ ./bin/dfs-client-p1 stat gt-einstein.jpg
// server
$ ./bin/dfs-server-p1

/* Test getting the status of a not existing file */
// client
$ ./bin/dfs-client-p1 stat file-not-found
// server
$ ./bin/dfs-server-p1
```

For part 2,

```
// server
```

```
$ ./bin/dfs-server-p2
// Client 1
$ ./bin/dfs-client-p2 mount
// Client 2
$ ./bin/dfs-client-p2 mount

// Terminal 1
$ cd ./mnt/client
$ echo "Test again" > test.txt
$ mkdir -p hello
$ echo "a testing file" > new.txt
$ rm test.txt
$ rm -rf hello

// Terminal 2
$ cd ./mnt/server
$ echo "Test again" > test.txt
$ mkdir -p hello
$ echo "a testing file" > new.txt
$ rm test.txt
$ rm -rf hello
```

Next Improvement

I think part 1 is perfectly handled with, but we can make it better for the synchronization of part 2. In the part 2, the gradescope is not responsible for all the goals mentioned in part 2's instruction and there are several improvements we can make in the future.

1. File Creation

When we create a new file locally, it will not be stored directly to the server. But if we create a new file on the server, it will immediately be fetched to the client.

2. File Deletion

This is quite opposite from our goals because when a client delete a file locally, we should also delete the related file on the server. However, in our implementation, the server file will not be deleted.

All the problems above are the possible improvements in the future.

Known Bugs/Issues/Limitations

1. Part 1 Empty File Testing Problem

I think the testing for part one should add a test about storing an empty file on the server. When I test this locally, I run into bugs because my server never quits and the clients will be interrupted by the deadline. But my code can still pass

2. Part 1&2 Directory Problem

In the beginning, the testing files are listed in the `/mnt/server/sample_files` directory and it makes no sense for doing so because these files can be directly put into the root directory `/mnt/server`. I figure this out by checking the piazza and some other students have the same confusion as me. I think it would be a better idea to put all the testing files in the root directory `/mnt/server` of the server in the beginning.

3. Part 2 Readme Error

In the readme instruction, the checksum function is mistakenly expressed by,

```
std::uint32_t crc = file_checksum(filepath, this->crc_table);
```

However, it should be,

```
std::uint32_t server_crc = dfs_file_checksum(filepath, &this->crc_table);
```

The instructions in the `servernode` are good but I think we should also update the readme instructions.

This issuse is also mentioned in the **piazza**.

4. Changing Goals in Part 2

This is just a general suggestion on the testing instances of the part 2. I am quite sure my code is not reponsible for all the features listed in the instructions, and it will take more time for me if I keep implementing these features. I think you can either make the goals simpler by changing the instructions because you will not test all the case, or you should develop the testing cases further more because the current tests are obviously not enough.

References

1. gRPC deadlines
2. proto3
3. gRPC Starter
4. gRPC Examples
5. File stat
6. File open
7. ifstream
8. ofstream
9. List files in a directory
10. opendir
11. readdir
12. inode
13. inotify
14. Piazza