Distributed File System

Author: Yuan Yang (yyang998)

This project is developed under Ubuntu Linux 20.04 environment with Vagrant and VirtualBox.

1. Part 1 - Building the RPC protocol service (single threaded)

Design

In this part of the project, gRPC is used to implement the remote procedure calls between two processes Client and Server. Client prepares the requests and sends the requests via services defined using protocol buffers. The implementations of the service APIs are completed on the Server side, and Client side is in charge of preparing the requests, calling methods on Server and processing the responses.
Protocol Buffer is used to serializing and structuring data (request and response) that need to be transmitted between Client and Server through remote procedure call.
Five services are implemented in this part of the project:

1. Store(): Client requests to store a file onto Server.
2. Fetch(): Client requests to fetch a file from Server to local.
3. Delete(): Client requests to delete a file on Server.
4. ListFiles(): Client requests to get a list of files that stored on Server's mount path.
5. GetStatus(): Client requests to get some attributes (modified time, created time, file size) of a file on Server.

In this part of the report, the control flow of the project will be discussed and explained for each service separately.

Store():

For the Store() method, since a transmission of a file is involved and file content might need to be transferred in chunks by Client instead of sending in full in one batch, a **Client Streaming RPC** should be used for Client to write and send sequences of content to Server side.

The flow and general steps of the Store() method is:

- Client prepares the requests: file name that the client wants to upload and the streaming file chunks;
- Client calls the method implemented on Server to sends to requests to Server;
- On the Server side, server opens a file with this specific file name and prepare to receive file content.
    - If the file exists on Server, open it.
    - If the file not exists on Server, create and open it.
- Client receives a ClientWriter for it to write file content chunks into the service input parameter.
- Server reads the file content chunks from the service input parameter and writes the content to the file opened on Server.

- Server prepares and sends the RPC response with acknowledgement with file name and modified time.

Fetch():

For the Fetch() method, since a transmission of a file is involved and file content might need to be transferred and sent in chunks by Server, a **Server Streaming RPC** should be used for Server to write and send sequences of content and sends to Client side.

The flow and general steps of the Fetch() method is:

- Client prepares the requests: file name that the client wants to fetch;
- Client calls the method implemented on Server to sends to requests to Server;
- On the Server side, server opens a file with this specific file name and prepare to send file content.
    - If the file exists on Server, open it using ifstream and prepare reading file.
    - If the file not exists on Server, return error code.
- Client receives a ClientReader for it to read file content chunks from the service returned response prepared by server.
- Server reads the file and writes the file content chunks to service returned response.
- Client reads the content from response using ClientReader and writes to local file.

Delete():

The flow and general steps of Delete() method is:

- Client prepares the requests: file name that the client wants to delete;
- Client calls the method implemented on Server to sends to requests to Server;
- On the Server side, server try to find the file. If found, server delete the file.
    - If delete existing file success, go to next step.
    - If delete existing file fail, return error status code to client.
- Server prepares and sends the RPC response with acknowledgement with file name and modified time.

ListFiles():

The flow and general steps of ListFiles() method is:

- Client prepares the requests: the input is empty since a list of all files is needed, not a specific file;
- Client calls the method implemented on Server to sends to requests to Server;

- On the Server side, Server opens its mounted directory. For each file in this directory:
  - Server prepares and sends the RPC response with current file's attribute.
- Client receives the response and insert the file and its attributes received from Server to a local map.

GetStatus():

The flow and general steps of Delete() method is:

- Client prepares the requests: file name that the client wants to get status of;
- Client calls the method implemented on Server to sends to requests to Server;
- On the Server side, server try to find the file.
  - If found, server prepare the response based on status of the file on Server.
  - If not found, return error code.
- Server sends the RPC response with file status.
- Client receives the response and get status of the file.

Choices and Code Implementations

- The communication strategy for RPC call status between Client and Server: `grpc::Status` and `grpc::StatusCode` are the features that used to communicate the status of RPC calls between Client and Server.
- I/O of the files on both Server and Client sides are implemented using `std::fstream` class.
  - `ifstream` for reading data from files.
  - `ofstream` for writing data into files.
    - Previous data stores in the file will be truncated in default and then write new data in.
    - If the file not exists, the file will be created and then have data written in.
- How to judge if an item in directory is file or not when implementing ListFiles():
  - Get the status of a file and check its `st_mode` : the item is a file of `st_mode` is `S_IFREG` .\
- Handling gRPC deadlines: (deadline is specified by user when start running the client)
  - Deadline needs to be specified by Client to indicate how long the Client is willing to wait for an RPC to complete and sends response back. If the deadline exceeds before Server completes the requests, the RPC will be terminated.
  - Set deadline in RPC context on Client side:

    ```
    std::chrono::system_clock::time_point curDDL = std::chrono::system_clock::now()
    context.set_deadline(curDDL);
    ```

  - Check if deadline exceeds on Server side:

```
if (context->IsCancelled())
{
    dfs_log(LL_ERROR) << "DDL EXCEED for processing file:" << curPath;
    return Status(StatusCode::DEADLINE_EXCEEDED, "Deadline exceeded.");
}
```

Tests

The tests in this part is just to test different functionalities/services implemented for the RPC protocol, including if a service can be completed successfully or it returns the right status code and error code.

- Different types of files with different size are used to test: the files I used for testing is basically the same batch of the files used for testing in Project 3 (detials can be found in my readme report for Project 3).
- Store(upload) a file from Client to Server: tested for scenarios that this specific file exists and not exists on Server.
- Fetch/Delete/GetStatus a file from Server to Client: tested for scenarios that this specific file exists and not exists on Server.
- List files on Server: Server mounted directory is empty or not, the items in directory are all not actual files. Print out the list of files on Client sides to check if the list is the same as the list of files on Server.
- Test services with different `deadline_timeout` : default is 10000, tested for 5000 and 20000.

2. Part 2 - Completing the Distributed File System

Design

Most of the part of the project is identical to Part 1, except that we need to implement it in multithreaded system and also need to keep Client mounted directory and files in it up to date with Server (Server also need to up to date with client). In order to implement these add-on functions, some essential attributes of the files (Check Sum and Modified Time) are needed to transmitted and

compared between Server and Client in Part 2. Also, since the system is multithreaded and the services contains I/O of files and directories, several mutex mechanisms are needed for all the resources that will be accessed by several threads. Moreover, on the Client side, a `HandleCallbackList()` function needs to be implemented for system to automatically watch for the modification in the directory and sync the Client and Server side based on certain mechanisms (will be explained later in this report).

Synchronization Design

Multithreading system requires the synchronization design, especially this system involves multiple modifications and I/O operations for every file.

- Mutexes and Resources used for Synchronization Design:
  - **On Client side,** the mounted directory is protected by a mutex.
  - **On Server side**, several mutexes are used:
    - The mounted directory is protected by a mutex.
    - A map is implemented `<file_name, {mutex to this file, owner of this mutex}>` : Each file in the mounted directory is protected by a mutex, and have an owner string registered to it to indicate which Client thread is currently holding the mutex.
      - While another client is holding the mutex, an error code with `RESOURCE_EXHASUTED` needed to be returned to let current client know the mutex is occupied.
    - This map is also protected by a mutex.

    The detailed implementation of mutexes and how the files are couple with mutexes and owner string is explained later: **Choices and Code Implementations**.
- Another design here is to implement an extra service call `RequestWriteLock()` to let Client acquire write lock for a specific file from Server. The general process is:
  - Client prepares the requests: file name that the client wants to acquire mutex of;
  - Client also need to store current ClientID as metadata and pass it to Server via context.
  - Client calls the method implemented on Server to sends to requests to Server;
  - On the Server side:

    a. Lock the map that holds the mutexes of each file;

    b. Try to find the file name among the keys of the map:
      - If the file has a lock -
        - if OwnerID associated to the file in map is an empty string - set the owner ID to be the current requesting client's ID: acquire lock succeed.
        - if OwnerID associated to the file in map is not an empty string: compare the lock owner ID with ClientID sent from Client:
          - different ID indicates the lock is already been held by another Client: acquire lock failed.

            - same ID indicates the lock is already been held by this current Client: acquire lock succeed.
        - If the file has no lock:
            - insert into the map with a new mutex for this file and set the owner ID to be the current requesting client's ID: acquire lock succeed
    c. Unlock the map that holds the mutexes of each file;
  - Server prepares and sends the RPC response with status code and message.
- Mutex releasing mechanism when it is no longer held by a specific Client:
  i. Lock the map that holds the mutexes of each file;
  ii. Clear the OwnerID associated the file name in map without deleting the mutex (replace with empty string).
  iii. Unlock the map that holds the mutexes of each file;

Async Thread: Design on the Handling of Asynchronous Event Calls

The provided Part 2 skeleton code contains the initial set-ups of the mechanisms for automatically monitor the changes in mounted directories using two threads (Watcher and Async). If identifies some modifications of Client and Server, the comparison of these two sides will be invoked automatically. Comparison mechanism - with Client's mounted directory protected by a mutex:

1. First get a list of files with file name, modified time on Server side
2. Client compares the files in its directory with the file list (need modified time stored in the list as well) obtained from Server, for each ServerFile on Server:
   - Case 1 - if the file with the same name as ServerFile exists on Client, compare the modified time of ServerFile and the ClientFile on Client.
     - Case 1a - if modified time of ServerFile is smaller than ClientFile: call Store() method to upload the ClientFile to Server since ServerFile is out of date.
     - Case 1b - if modified time of ServerFile is larger than ClientFile: call Fetch() method to get the ServerFile from Server and update the ClientFile since ClientFile is out of date.
     - Case 1c - if modified time of ServerFile equals to ClientFile: do nothing.
   - Case 2 - if the file with the same name as the ServerFile not exists on Server:
     - Call Fetch() method to get the ServerFile from Server.

   During this process, directories are protected by mutex indicated in last section Synchronization Design.

Modifications for the services implemented in part 1 to adapt the changes made during synchronization design:

The five methods do need some lock mechanism:

- ListFiles() and CallbackList() are basically the same:
  - the operation with the mounted directory should be protected.
- Store():
  - Client needs to send the check sum, modified time, client thread id to Server via context.
  - Client needs to acquire the write lock of the file on the Server before actually sending the file to Server.
  - On the Server side, if there is a file already exists with the same name as the requested, the check sum should be compared:
    - If the check sum of the two files are the same on Client and Server, no need to upload and save the file again on the Server side, just to further compare modified time of the two files and return `ALREADY_EXISTS` error code to client. Need to make sure the modified time of both files are synced:
      - If modified time on Client is smaller than modified time on Server, do nothing, release the lock and return (Client side handles the scenario by setting the file's modified time on Client to be the same as Server);
      - If modified time on Client is larger than modified time on Server, update the modified time of the Server file to be the modified time on Client;
    - If the check sum of the two files are not the same, do regular Store() with file mutex locked.
  - After regular Store() task finished, current client needs to release the lock associated the file.
- Fetch():
  - If the file client wants to fetch exists on Client already, Client needs to send the check sum and modified time to Server via context.
  - On the server side, finds the file mutex and locks it.
    - Check sum and modified comparison is the same as Store() - introduced above in Store() method.
  - After regular Fetch() task finished, mutex should be unlocked.
- Delete():
  - Client needs to send the client thread id to Server via context.
  - Client needs to acquire the write lock of the file on the Server before actually deleting the file on Server.
  - Server deletes the file while protecting by the file mutex.
  - After regular Delete() task finished, current client needs reto lease the lock associated the file.

Choices and Code Implementations

- There are several possible Synchronization Design Strategies to choose from in this project:
  - Option 1: Server maintains a map that each file is associated with a mutex and an owner ID (explicit lock with a convinient way to check the lock owner at current time point in system).
  - Option 2: Server maintains a map that each file is associated with an owner ID (implicit lock).
  - Option 3: Server maintains a map that each file is associated with a mutex (explicit lock).

  At the end, I choose to use Option 1 to implement the system. Using an explicit lock is more convenient when creating a critical section in the code and makes the code more readable and easy to debug. Moreover, having an owner ID associated to file provides the mechanism to check the owner and knows what is going on internally. In regard of how to release the mutex, with option 1, I can just clear the owner ID field after finishing certain operation on file.
- Implementation details of Synchronization Design Option 1 - how Server maintains a map that each file is associated with a mutex and an owner ID:
  - The key of the map is the file name, and the value in map is a self-defined `struct ServerFilePair` which contains the mutex pointer and the current owner ID associated to the file. This map also need to be protected by mutex while accessing it.

```
struct ServerFilePair
{
    std::unique_ptr<std::mutex> curMutex;
    std::string curOwnerClientID;
};
std::map<std::string, ServerFilePair> serverFileLockMap;
std::mutex serverFileLockMapMutex;
```

- How to add a mutex with no owner to a file - insert in map with empty owner ID and new mutex pointer:

```
serverFileLockMapMutex.lock();
serverFileLockMap[curFileName].curOwnerClientID = std::string();
serverFileLockMap[curFileName].curMutex = std::make_unique<std::mutex>();
serverFileLockMapMutex.unlock();
```

- How to release a lock from a client - clear the `curOwnerClientID` :

```
serverFileLockMapMutex.lock();
((serverFileLockMap.find(filename_requested))->second).curOwnerClientID.clear(); //
serverFileLockMapMutex.unlock();
```

- Different from Part1, especially for Store() and Fetch(), the I/O operations needs extra care. When using `ofstream` to write data into the file, we need to make sure there is actually data coming in before opening the stream. For example:
  - When Client tries to fetch a file `x` from server, if `x` exists on both Client and Server sides and the two check sums are equal, Server will only return `ALREADY_EXISTS` status code and not be sending any actual file content to Client. If the `ofstream` is opened on Client side before knowing this, 0 bytes will be stored on Client side and overwrites the old file which has the contents in it.
- Why use unique pointer for file mutexes `std::unique_ptr<std::mutex>` ?
  - Unique pointer cannot be copies to other pointers, avoiding different files sharing the same mutex.

Tests

- All the tests in Part1 are also used to test the basic functionality of Part2.
- Other tests are designed specifically to test the Synchronization and Async callbacks. Some examples of tests I used are listed here:
  - With several files in Server's mounted directory and no file at all in Client's mounted directory, after mounting both Client and Server, all the files on Server should be automatically fetched to Client and then the modified time on both sides should be updated to the latest same one. The files on both sides should be the same with the same sizes.
  - When Client and Server are already mounted, delete a file on Client, the file will be automatically fetch from Server to Client.
  - When Client and Server are already mounted, replace a file on Client with a file with same name but older modified time, the Fetch() should automatically start.
  - When Client and Server are already mounted, replace a file on Server with a file with same name but older modified time, the Store() should automatically start.
  - On client side, call Store() on a file. After uploading file to Server, do Store() on the same file again. The write lock should be acquired successfully by client calling second Store() => after operation finished, write lock on a file should be released and ready to be acquired by another client.
- Helder's stress test posted on Piazza: https://piazza.com/class/lco3fd6h1yo48k/post/1212

References

RPC related:

- https://protobuf.dev/programming-guides/proto3/
- https://grpc.io/docs/what-is-grpc/introduction/
- https://levelup.gitconnected.com/understanding-grpc-a-practical-application-in-go-and-python-f3003c9158ef
- https://cs.brown.edu/courses/csci1310/2020/assign/labs/lab8.html
- https://grpc.github.io/grpc/cpp/classgrpc_1_1internal_1_1_reader_interface.html
- https://piazza.com/class/lco3fd6h1yo48k/post/1274
- https://piazza.com/class/lco3fd6h1yo48k/post/1200
- https://grpc.io/docs/languages/cpp/basics/
- https://grpc.io/docs/what-is-grpc/core-concepts/
- https://grpc.github.io/grpc/cpp/classgrpc_1_1_status.html
- https://grpc.io/blog/deadlines/
- https://grpc.github.io/grpc/core/md_doc_statuscodes.html
- https://github.com/grpc/grpc/tree/master/examples/cpp
- https://github.com/grpc/grpc/tree/master/examples/cpp/metadata
- https://dev.to/techschoolguru/upload-file-in-chunks-with-client-streaming-grpc-golang-4loc
- https://betterprogramming.pub/grpc-file-upload-and-download-in-python-910cc645bcf0
- https://stackoverflow.com/questions/46105216/file-transfer-using-grpc
- https://github.com/grpc/grpc-web/issues/517
- https://www.youtube.com/watch?v=i9H3BaRGLEc
- https://itnext.io/downloading-files-using-golang-grpc-f07e4a16a536
- https://medium.com/codex/distributed-services-using-grpc-100743363c6b
- https://piazza.com/class/lco3fd6h1yo48k/post/1243
- https://piazza.com/class/lco3fd6h1yo48k/post/1212
- https://piazza.com/class/lco3fd6h1yo48k/post/1324
- https://github.com/grpc/grpc/blob/master/examples/cpp/metadata/greeter_server.cc
- https://grpc.github.io/grpc/cpp/classgrpc_1_1string__ref.html
- https://grpc.io/blog/deadlines/#checking-deadlines
- https://www.youtube.com/watch?v=gnchfOojMk4&t=144s
- https://piazza.com/class/lco3fd6h1yo48k/post/1353

cpp related:

- https://www.simplilearn.com/tutorials/cpp-tutorial/ifstream-in-cpp
- https://learn.microsoft.com/en-us/cpp/cpp/how-to-create-and-use-unique-ptr-instances?view=msvc-170
- https://stackoverflow.com/questions/11617552/c-assigning-null-to-a-stdstring
- https://cplusplus.com/reference/string/
- https://stackoverflow.com/questions/2185338/how-to-set-the-modification-time-of-a-file-programmatically
- https://cplusplus.com/reference/fstream/
- https://stackoverflow.com/questions/612097/how-can-i-get-the-list-of-files-in-a-directory-using-c-or-c
- https://stackoverflow.com/questions/146924/how-can-i-tell-if-a-given-path-is-a-directory-or-a-file-c-c
- https://cplusplus.com/reference/map/multimap/
- https://www.geeksforgeeks.org/map-find-function-in-c-stl/
- https://man7.org/linux/man-pages/man2/utime.2.html
- https://en.cppreference.com/w/cpp/io/c/remove
- https://stackoverflow.com/questions/15388041/how-to-write-stdstring-to-file
- https://www.educba.com/c-plus-plus-ofstream/
- https://cplusplus.com/reference/mutex/mutex/
- https://cplusplus.com/reference/istream/basic_istream/read/