readme-student.md

CS6200 Distributed File System using GRPC

Parin Patel : ppatel480@gatech.edu

Project Description

In this project, We are designing and implementing a simple distributed file system (DFS). First, we will develop several file transfer protocols using gRPC and Protocol Buffers. Next, we are incorporating a weakly consistent synchronization system to manage cache consistency between multiple clients and a single server. Our Distributed file system supports following commands.

- Fetch : It will pull the file from remote server if file isnt locked.
- Store : It will try to store file if it has file lock.
- List : Lists all file on server.
- Stats : It returns stat object from server , which contains information about size , modified time , creation time , file path etc
- Lock : It hasnt exposed from command line but can be used to acquire write lock on file.
- Mount : It mounts remote file system on localserver , it supports write,deletion or remote changes to be reflected on local and remote system. Similar to sync operation.
- Delete : It deletes server/remote file as well as file on all other mount clients.

Source code using a combination of C++14, gRPC, and Protocol Buffers to complete the implementation.

Part 1 Description :

Part 1 implements following service into our server , we have mentioned this [above](#Project Description). :

- Fetch
- Store
- List
- Stats
- Delete

Design:

We will be using IFStreams/OFStreams during given project as client is single threaded & in future requirement only one client can acquire write lock on file . Thus all write operations are atomic. Secondly , each client/Server thread performing request will have its own copy of IFStreams/OFStreams Objects copy. This will allow use non-thread safe objects in over code. More on this limitation later.

GRPC :

We are using `stream` GRPC whenever we need to transfer file data or any large data. This allows use to reduce the memory buffer neeeded to serialize messages for GRPC. We will be following Server & Client based method to transfer the file or any required information. Client will be sending a request and either receive go ahead to transfer data or return one of the failure condition.

Server Side:

- Fetch :

1. Receive request from client.
2. Validate if deadline isn't exceeded else return `::grpc::StatusCode::DEADLINE_EXCEEDED` .
3. Translate remote path to local path.
4. Check if file exist locally , if not send `::grpc::StatusCode::NOT_FOUND` .
5. Sets `file_stats` to new files stats and send payload back .
6. Try to Open existing and start reading/streaming the file till `EOF` .

7. If read is successful check set `file_transfer_status` to `FILE_TRANSFER_SUCCESS` else `FILE_TRANSFER_FAILURE` and return with with `::grpc::Status::OK`.

- Store

1. Receive request from client.
2. Validate if deadline isn't exceeded else return `::grpc::StatusCode::DEADLINE_EXCEEDED` .
3. Translate remote path to local path.
4. Try to Open existing / Create new file. and start writing the file till `EOF` .
5. If write is successful check set `file_transfer_status` to `FILE_TRANSFER_SUCCESS` else `FILE_TRANSFER_FAILURE`.
6. Sets `file_stats` to new files stats and send payload back with `::grpc::Status::OK` .

- List

1. Receive request from client.
2. Validate if deadline isn't exceeded else return `::grpc::StatusCode::DEADLINE_EXCEEDED` .
3. Try to Open current directory. and start reading till reaching end of list .
4. Ignore entries which are either `.` or `..` or are hidden(starts with `.`) , otherwise start reading stats for each entry.
5. Close the directory and send payload back with `::grpc::Status::OK` .

- Stats

1. Receive request from client.
2. Validate if deadline isn't exceeded else return `::grpc::StatusCode::DEADLINE_EXCEEDED` .
3. Translate remote path to local path.
4. Check if file exist locally , if not send `::grpc::StatusCode::NOT_FOUND`.
5. Try to Open existing and start reading stats.
6. Sets `file_stats` to new files stats and send payload back with `::grpc::Status::OK` .

- Delete

1. Receive request from client.
2. Validate if deadline isn't exceeded else return `::grpc::StatusCode::DEADLINE_EXCEEDED` .
3. Translate remote path to local path.
4. Check if file exist locally , if not send `::grpc::StatusCode::NOT_FOUND`.
5. Try to delete existing file.
6. send payload back with `::grpc::Status::OK` .

Client Side

- Fetch :

1. Creates context for server and sets deadline.
2. Sets request path and sends payload .
3. Open local file for writing and start writing streaming payload till `EOF` .
4. If write is successful and `file_transfer_status` == `FILE_TRANSFER_SUCCESS` then return `OK` else return with incoming return code.

- Store

1. Creates context for server and sets deadline.
2. Sets request path, stats and sends payload .
3. Open local file for writing and start reading streaming payload till `EOF` .
4. If read is successful and incoming response `file_transfer_status` == `FILE_TRANSFER_SUCCESS` then return `OK` else return with incoming return code.

- List

1. Creates context for server and sets deadline and send it.
2. sets file_map with required fields (file path and mtime).
3. return with incoming return code.

- Stats

1. Creates context for server and sets deadline.
2. Sets request path and sends payload .
3. Update / memcopy to *file_status.
4. Return with incoming code.

- Delete

1. Creates context for server and sets deadline.
2. Sets request path and sends payload .
3. Return with incoming code.

Message and Method Structure:

GRPC Protos:

```
service DFSService {
    //  A method to store files on the server
        rpc store_file(stream  file_stream) returns (file_response);
    //  A method to fetch files from the server
        rpc fetch_file(file_request) returns (stream file_stream);
    //  A method to delete files from the server
        rpc delete_file(file_request) returns (file_response);
    // A method to list all files on the server
        rpc list_file(empty) returns (stream file_list);
     // A method to get the status of a file on the server
        rpc stat_file(file_request) returns (file_response);
}

// Response for Fetch or Request for Store
message file_stream{
        string file_name = 1;
        bytes file_stats = 2;
        bytes file_data = 3;
};

message file_request{
        string file_name = 1;
};

message file_response{
        string file_name = 1;
        int32 file_transfer_status = 2;
        bytes file_stats = 3;
};

message file_list{
        bytes files = 1;
}

message empty {

}
```

Tests:

```
#!/bin/bash

bin/dfs-server-p1 -d 3 -m mnt/server/sample-files
```

```
#!/bin/bash
set -e
TXT_FILE_NAME='parin.txt'
```

```
BIN_FILE_NAME='gt-klaus.jpg'

for i in "list" "store" "fetch" "stat" "delete" ; do
if [[ $i == "list" ]] then
    bin/dfs-client-p1 -d 3 -m /mnt/client list
elif [[ $i == "stat" ]] then
    bin/dfs-client-p1 -d 3 -m /mnt/client $i ${BIN_FILE_NAME}
    bin/dfs-client-p1 -d 3 -m /mnt/client $i ${TXT_FILE_NAME}
elif [[ $i == "delete" ]] then
    bin/dfs-client-p1 -d 3 -m /mnt/client $i ${BIN_FILE_NAME} && ls -la
    bin/dfs-client-p1 -d 3 -m /mnt/client $i ${TXT_FILE_NAME} && ls -la
else
    bin/dfs-client-p1 -d 3 -m /mnt/client $i ${BIN_FILE_NAME} && \
        diff -s mnt/server/${BIN_FILE_NAME} /mnt/client/${BIN_FILE_NAME}
    bin/dfs-client-p1 -d 3 -m /mnt/client $i ${BIN_FILE_NAME} && \
        diff -s mnt/server/${TXT_FILE_NAME} /mnt/client/${TXT_FILE_NAME}
fi
done
```

Part 2 Description :

Part 2 extends service implemented by part 1 into our server , we have mentioned in problem statement. :

- Support write locks for whole file. If file is locked , other clients will fail to read/write/stats/delete the file.
- CRC Checksum: Validate file changes using CRC checksum.
- Local file Caching: Support whole file local cache , changes need to be updated to locally first and then propagated to remote server.
- Client initiated deletion / Mount: if the file has changed / deleted locally then the change needs to be broadcast to all other clients who are caching the file locally.
- Date base sequences : assuming client/server are using correct timestamps , Latest modified file is source of truth.

Design:

We will be using a map with path as key and client id as value to keep track of file locks. To keep track of client side deletion and deletion broadcast we will be utilizing Callback function to get current directory file list and take appropriate action. For Syncing file during mount we will be fetching every file to local cache and if different file with same path exist we will making sure copy with last modified time is kept. if any changes happen to local system it will be broad-casted to all other nodes.

GRPC :

Similar to previous part we will be using `stream` to transfer files , Although data structure of request and response have changed to accomdate locks, client id and few other stats useful for comparing file content.

Server Side:

Design is quite same as part1 , I shall be just listing modified bits and pieces.

- **Fetch**: We will be checking for any locks on server before we read the local copy to send.
- **Write** :

1. same as Part 1 above till server gets request.
2. Acquire mutex to check for file_lock_map , once mutex is acquired check for file lock.
   a. if current client or no one owns it then acquire the lock by adding entry in map , then same as part 1 , write the file
   b. If Lock is already aquired, send `StatusCode::RESOURCE_EXHAUSTED` .

- **Stats** : same as Part1 , just check for lock before reading stats.
- **Delete** : same as Part1 , check for lock before deleting file. To broadcast delete we will be implementing this part in handle callback.
- **List** : Same as Part1
- **mount_path**: Sets Async threads to check incoming Callback request in `ProcessQueuedRequests` .
  - **ProcessQueuedRequests** : Acquire `queue_mutex` mutex lock before calling : `CallbackList` , so that current & another Async thread dont run into race condition.
  - **CallbackList** : It is quite similar to list command , few changes is it also includes few other info as shown in file_object below.

Client Side:

- **Fetch**: Same as Part 1
- **Write** : Same as Part 1, just asks for lock
- **Stats** : same as Part1
- **Delete** : same as Part1
- **List** : Same as Part1
- **InotifyWatcherCallback** : Acquire : `inotify_mutex` mutex lock before calling : `HandleCallbackList` , so that `InotifyWatcherCallback` & another Async thread that is handing some read/write dont run into race condition.
- **HandleCallbackList** : At initialization of client `mount` = False.
  1. Get list of all local files in mounted directory
  2. Receive in remote file list from callback resposne.
  3. for every entry in remote list,
     - if remote file path exist in local and CRC is same (File is unmodified) then DO Nothing
     - if remote file path exist in local and CRC is different then:
       - if remote file mtime > local : Fetch
       - if remote file mtime < local : Store
     - if remote file path doesnt exist in local :
       - if `mount` == False : Fetch
       - else : Issue Delete Command
     - Remove entry from local list.
  4. If count(local_list) > 0 :
     - if `mount` == False : Fetch
     - else : Delete local copy
  5. Set `mount` = True

Message & other notable Structures :

GRPC Message Structures:

```
message file_stream{
  string file_name = 1;
  bytes file_stats = 2;
  bytes file_data = 3;
  string client_id = 4;
  uint32 file_CRC = 5 ;
};

message file_request{
  string file_name = 1;
  string client_id = 2;

};

message file_response{
  string file_name = 1;
  string client_id = 2 ;
  int32 file_transfer_status = 3;
  bytes file_stats = 4;
  uint32 file_CRC = 5 ;
  bool file_lock = 6;
};

message file_list{
  bytes files = 1;
  int32  file_length = 2;
}

message empty {
  string name = 1; // Required due to part2/src/dfslibx-clientnode-p2.h:200
}
```

File Object structure used in list_files and Handle Callback.

```
struct file_object {
    char file_path[256];
    std::int32_t mtime;
    std::uint64_t file_size;
    std::int32_t create_time;
    std::uint32_t file_crc;
};
```

Tests

```
#!/bin/bash

bin/dfs-server-p1 -d 3 -m mnt/server/sample-files
```

```
#!/bin/bash
set -e
TXT_FILE_NAME='parin.txt'
BIN_FILE_NAME='gt-klaus_LARGE.jpg'

for i in "list" "store" "fetch" "stat" "delete" ; do
if [[ $i == "list" ]] ; then
    bin/dfs-client-p1 -d 3 -m /mnt/client list
elif [[ $i == "stat" ]]
then
    bin/dfs-client-p1 -d 3 -m /mnt/client $i ${BIN_FILE_NAME}
    bin/dfs-client-p1 -d 3 -m /mnt/client $i ${TXT_FILE_NAME}
elif [[ $i == "delete" ]]  ;
then
    bin/dfs-client-p1 -d 3 -m /mnt/client $i ${BIN_FILE_NAME} && ls -la
    bin/dfs-client-p1 -d 3 -m /mnt/client $i ${TXT_FILE_NAME} && ls -la
else
    bin/dfs-client-p1 -d 3 -m /mnt/client $i ${BIN_FILE_NAME} && \
        diff -s mnt/server/${BIN_FILE_NAME} /mnt/client/${BIN_FILE_NAME}
    bin/dfs-client-p1 -d 3 -m /mnt/client $i ${BIN_FILE_NAME} && \
        diff -s mnt/server/${TXT_FILE_NAME} /mnt/client/${TXT_FILE_NAME}
fi
    bin/dfs-client-p1 -d 3 -m /mnt/client $i ${BIN_FILE_NAME} &
    bin/dfs-client-p1 -d 3 -m /mnt/client $i ${BIN_FILE_NAME}  || echo "FILE IS LOCKED!!!!"
done
```

mount :

```
bin/dfs-client-p1 -d 3 -m /mnt/client mount &
watch -n 1 ls -la /mnt/client
```

& Run above script in different shell and watch the ls -la output, include sleep if needed in bash script to watch comfortably.

Problems & Improvements:

- I ran into issue reading and writing through ifstream/ofstream as per ifstream doc:

> If the input sequence runs out of characters to extract (i.e., the end-of-file is reached) before n characters have been successfully read, the array pointed to by s contains all the characters read until that point, and both the eofbit and failbit flags are set for the stream.

If total byte % BUFFER SIZE != 0 then it wasnt flushing last chunk , thus resulting in incorrect transfer. This issue was resolved by flushing last buffer if `file_reader.gcount() > 0`.

- Server and Client is stateless thus , crash by either system can cause hanging /orphaned locks.
    - solution 1 : use timestamped base locks. (preferred) .

    ```
    #define UNLOCKED false;
    ```

```
#define LOCKED true;
struct file_lock {
    std::string client_id;
    time_t locktimestamp;
};

std::map<std::string, struct file_lock > lock_map{};
bool check_for_file_lock(const std::string &file_path, const std::string &client_id) {
    if (lock_file_map.count(file_path) == 0) {
    return UNLOCKED;
    } else if (lock_file_map[file_path].client_id == client_id && lock_file_map[file_path].client_id <
               time(NULL) - 300  ) {
                  return UNLOCKED;
                  }
    else return LOCKED;
}
```

- solution 2 : use journaling on server side to keep track.

References

- gRPC C++ Reference
- Protocol Buffers 3 Language Guide
- gRPC C++ Examples
- C++14 cppreference
- CPlusPlus
- CS6200 Piazza