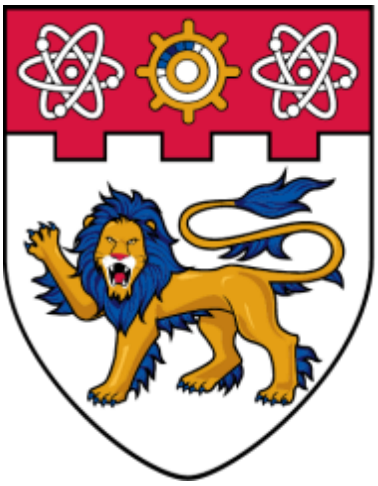

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

SC4051 Distributed Systems

Semester Project

Submitted By: Zeng Ruixiao

Matriculation Number: U2220375D

Demonstration Time: April 8 16:00-16:15

1. Introduction

In this project, it needs to design and implement a system for remote file access based on client-server architecture.
The files are stored on the local disk of the server which provides a set of service to remotely access the files.
User can use the client program interface to invoke these services and get respond after sending request.
Communication between client and server is carried out using UDP.
The system should satisfy the requirements below:

1.1. Read: user provides file pathname, an offset and the number of reading bytes, and server should return the specified length of bytes starting from the offset byte of the specified file.

1.2. Write: user provides file pathname, an offset and the bytes need to write in, and server should insert the specified bytes at the offset byte of the specified file.

1.3. Monitor: user provides file pathname and monitor interval to monitor a specified file for interval time. During monitoring, user can receive notice and file content once it is updated. After the interval time, monitor will be expired and deleted from server. And a certain user cannot do anything else when they are monitoring, but it allows multiple clients to monitor a file concurrently.

1.4. Cache: after user read from server, client should retain it in a buffer to speed up read operation. It uses approximate one-copy update semantics to maintain cache consistency. And freshness interval t should be set as an argument in command at client starting.

1.5. System needs to implement 2 other operations. One of them is idempotent and the other is non-idempotent. In implementation, project is chosen the delete as non-idempotent operation, the move as half-idempotent operation (in system status, but not in reply) and overwrite as idempotent operation (in both system status and reply).

2. Architecture Design

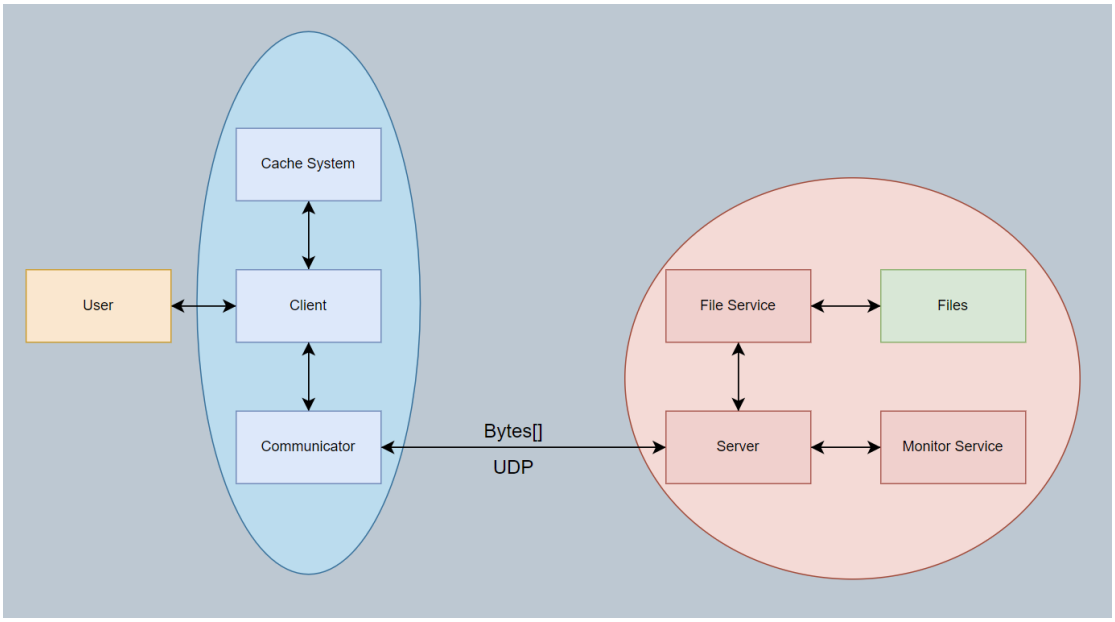

Fig.1 Architecture of project

3. Message Format

3.1. Request:

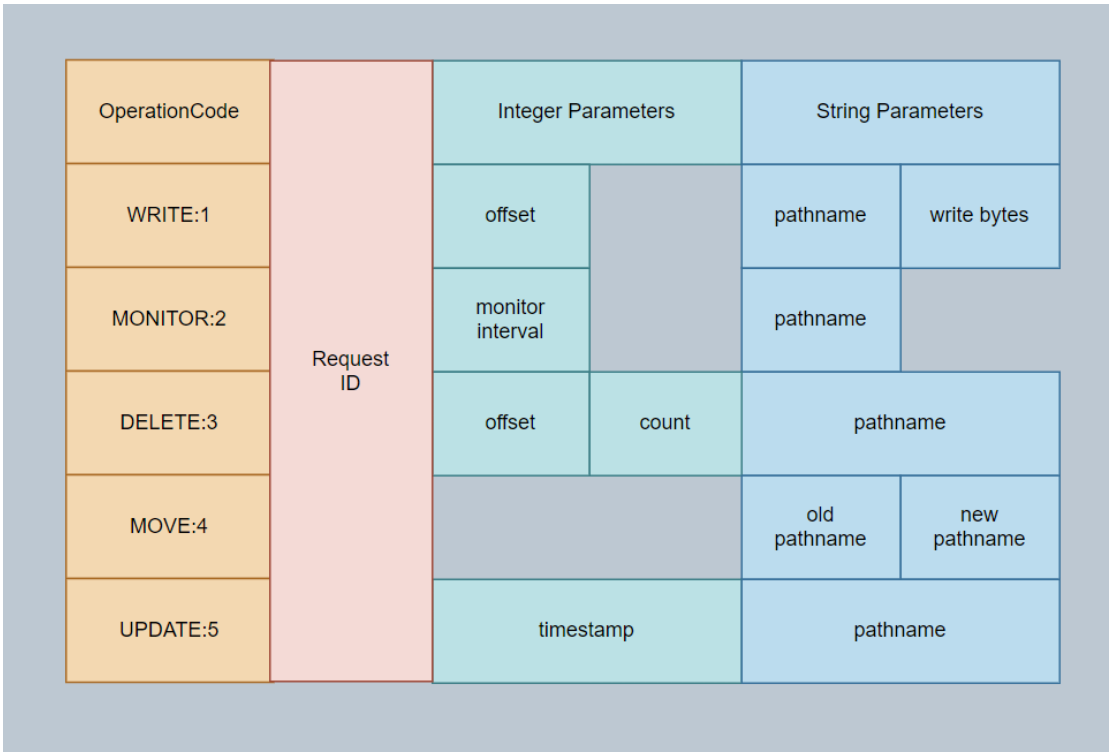

Fig.2 Request Structure

Operation code is 1 byte. Request ID is 4 bytes.
For integer parameter, each unit block is 4 bytes. Timestamp use int64_t type which is 8 bytes.
For string parameter, it is consisted by the string length in int (4 bytes), and its content store in chars (1 byte for each).

3.2. Response:

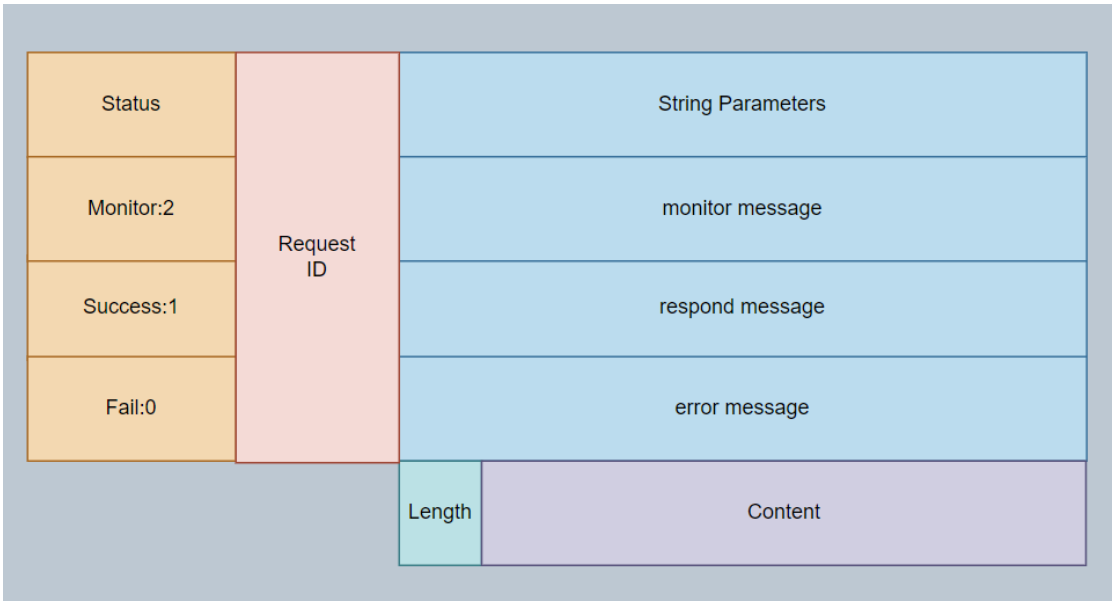

Fig.3 Response Structure

Status code is 1 byte.
For string parameter, it is consisted by the string length in int (4 bytes), and its content store in chars (1 byte for each).
Request ID is used to filter repeat response, and the monitor message has request ID with a int 0 for padding.

3.3. Special tackle with update response message:

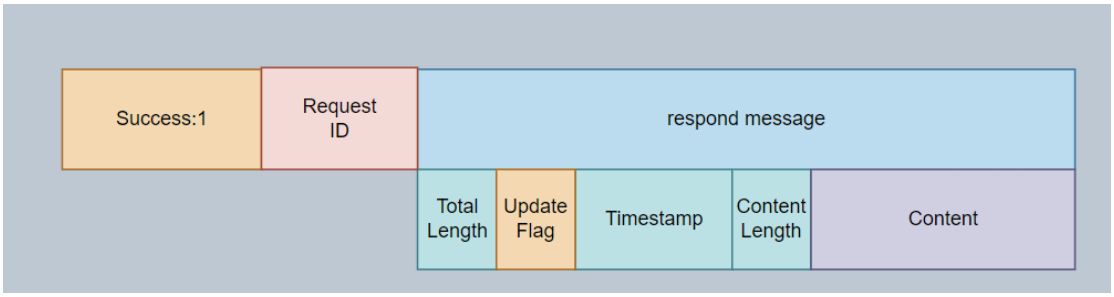

Fig.4 Update Response Structure

For transmission of timestamp, it uses a nested structure to encapsulated data.
The content part is optional if update flag is false.

4. Server Implementation

Server is implemented with Java, containing the classes below.

4.1. Entity class:

4.1.1. Service:

It stores a single request information including request ID, client address, client port, process status, successful status and response message.
It would be used to filter the repeat request.

4.1.2. OperationRequest:

It stores all possible parameters would appear in a request which can be passed between classes.

4.1.3. FileMonitor:

It stores a single monitor information from a client include pathname, starting time, time interval, client address, client port and expired time.
It would be used for monitor function.

4.2. Core Class:

4.2.1. Server:

The main class during running. It manages the file service, monitor service and service history. It receives byte array from client **Unmarshalling** it to the **OperationRequest** object which stores the unmarshalled datum and handle the request through invoking 3 classes mentioned above. It sends the response to client after marshalling.

4.2.2. FileService:

This class used to process the file operation. It contains 6 methods: write, delete, move, readAll, updateCheck and overwrite. Each method uses a 2-D byte array to pass the error message in parameters, it would not be mentioned below again.
Write needs 3 parameters: String pathname, int byteOffset, byte[] data; It can insert the data into the byteOffset byte in file pathname.
Delete needs 3 parameters: String pathname, int byteOffset, int delCount; It can delete delCount bytes content starting from byteOffset byte in file pathname.
Move needs 2 parameters: String pathname, String newpathname; It can move the file pathname to newpathname, including rename.
ReadAll needs 1 parameter: String pathname; It can return the full content of file pathname.
UpdateCheck needs 2 parameters: String pathname, long lastUpdate; It can return whether the file pathname is updated after lastUpdate.
Overwrite needs 3 parameters: String pathname, int byteOffset, byte[] data; It can overwrite the data starting from the byteOffset byte in file pathname.

4.2.3. MonitorService:

It maintains an ArrayList of **FileMonitor** objects. It can add, remove and find the monitors with pathname. It can select the ArrayList for a certain file and remove the FileMonitor if it is expired.

4.2.4. ServiceHistory:

It maintains an ArrayList of **Service** objects. It can check whether the service processed and resend the response without repeat process.

There is an additional class **PacketDropSimulator**, it can drop the packet with a random number comparing with the setting drop probability to simulate the real packet loss.

5. Client Implementation

Client is implemented with C++, containing the classes below.

5.1. Entity class:

5.1.1. ReceivedMessage:

It stores all possible parameters would appear in a response. It can be passed between classes.

5.1.2. CacheItem:

It stores the cache information about a single file including file path, last update time and content.

5.2. Core Class:

5.2.1. Client:

The main class during running. It provides a command user interface for user. It can pack the data and pass it to Communicator and process the response datum for demonstration to user. It can read from cache for better performance.

5.2.2. CacheManager:

It maintains an unordered map of CacheItem with key of their pathname. It can add, update or remove a CacheItem. It can check whether the CacheItem is fresh. It can obtain CacheItem information.

5.2.3. Communicator:

This class will communicate with server through UDP with vector<uint8_t> type datum. It implements timeout retransmission and monitoring file update during monitor.

5.2.4. Marshalling:

Marshall the original data package into vector<uint8_t> type byte array.

5.2.5. Unmarshalling:

Unmarshall the vector<uint8_t> type byte array to **ReceivedMessage** object.

6. Function implementation

6.1. File operation

All direct file operation method is in the class **FileService**. During running, it is instantiated in the main program, and it will invoke correspond method to process the request.
Specially, the read operation is deprecated as it can be implemented in update operation. (see 6.3 for details)

6.2. Monitor

On server side, monitor request is stored by a **FileMonitor** object. It stores client information and monitoring file information. During running, server will instantiate a **MonitorService** object to store all **FileMonitor** objects.
For each monitor request, it adds a new **FileMonitor** object to **MonitorService**.
If a request modifies a file, we will find all monitor request on this file (and remove expired monitors during finding, do not remove it before a new monitor message need to send).

6.3. Cache in approximate one-copy update approach

On the client side, for each file, it uses a **CacheItem** object to store. It contains the file path, last update timestamp and content.
For each first read operation, it invokes the update operation to obtain the full content and intercept target bytes in client and store the full file in cache.
For later read operation for a file in cache, it checks whether the file is fresh. If not, it will send the update request for obtaining latest version.
For modified operation which may change the file status including write, delete and move, once they are invoked in client, the cache will erase the file.
If a file is not in cache, it can be obtained from server.
If a file in cache is fresh, it can directly be read through cache.
If a file in cache is not fresh, it can update from server.
If a file in cache is modified, it should be erased from cache.
So, the file can keep consistency with server.

7. Fault-tolerance measure

To avoid anomaly situations caused by unstable network or unexpected input, the methods below are used to tackle these problems.

7.1. Timeout retransmission:

On the client side, if it cannot receive a response in an appropriate time (timeout is set to 2 seconds), the message will try to retransmit. To avoid the infinite retries, the max retries in this project is 7. It is implemented in class **Communicator**.

7.2. Filter duplicate requests & Maintenance histories:

Request ID is used to identify a request in a certain client-server communication. It is generated by client side start from 1 and add 1 for each time. Requests send from different clients may share a same request ID.
On the server side, it may receive a duplicate request as the use of retransmission in 7.1. It can store the first processed request information in a **Service** object. Each information consists of client-request information and processing information. Client-request information contains the client address, client port and request ID, which is used to detect whether the service was received before as it is bijection between request and (address, port, ID) tuple.
Processing information would contain the processed status, successful status and response message after the request processed for at least 1 time. It can retransmit the response based on this part information if it receives a repeat request. In project, a **ServiceHistory** object will manage the service and check whether a new request is appeared before.
On the client side, as it may receive a duplicate response caused by retransmission. When it receives a response, it would unmarshall the byte array and obtain a request ID from it. And it will compare it with current request ID. If they are different, it indicates that the response does not correspond to goal request. So that it can correctly drop the repeat response. It is implemented in class **Communicator** and class **Unmarshalling**.

7.3. Other problems:

On the server side, if it cannot process the request because of some exceptions including but not limited to file not found or I/O error, it will return the error reason as the response message.

8. Invocation Semantic

8.1. At-least-once

With the retransmission function, it can ensure that the request is processed at least once. If it is not processed, it should be the problem of network connection.

8.2. At-most-once

With the filter duplicate request and maintenance histories, it can ensure each non-idempotent operation request processed for at most once.

8.3. Experiment

8.3.1. Design:

Change the running mode of both the server and the client to debugging, add breakpoints before their corresponding sending and receiving functions, and use breakpoints to control the timeout mechanism to determine retransmission.

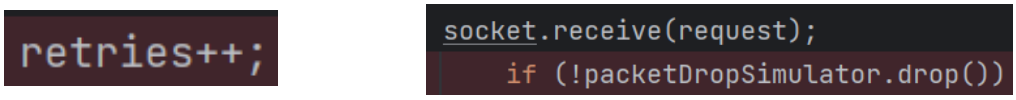

Fig.6 Breakpoints in client code(left) and server code(right)

The dark red line shows the breakpoint.

On client side, the breakpoint is set at the retry counter addition.

On server side the breakpoint is set at package drop which is after receiving request.

8.3.2. Steps:

a) Start client and server in the debug mode and set breakpoints.

b) Input the request information at client (it will send a request once input finish), it will test for delete, overwrite, move(not redo when receive repeat one), move(redo).

c) Resume the client for few times, and client send more requests to server.

d) Resume the server for few times. It will start process requests and send responses.

e) Resume the client. It will resend a request and receive the responses before.

8.3.3. Result:

All client side is start with a menu; it would not repeatedly appear in client result.

And the color of block is the part of steps. abcde

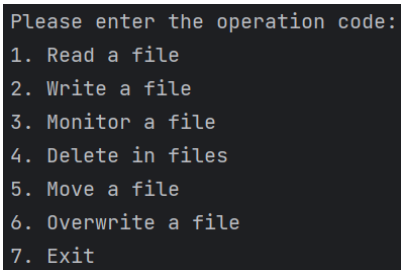

Fig.7 client menu

8.3.3.1. Delete

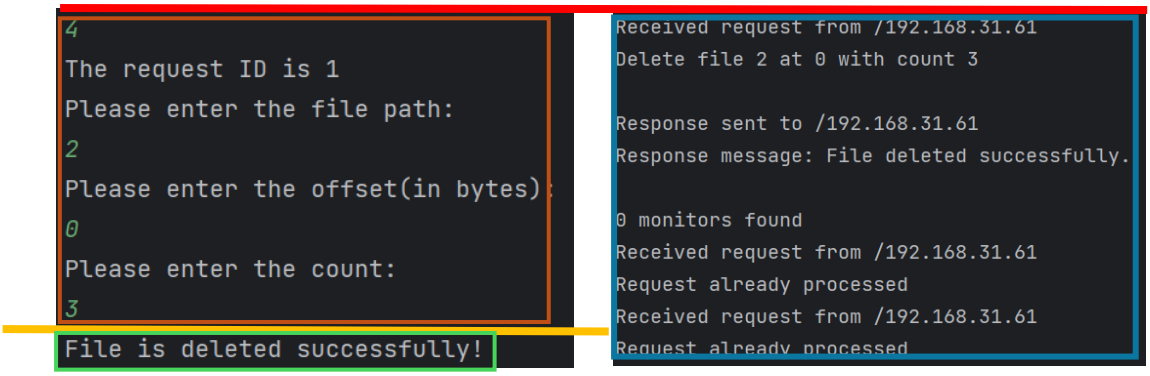

Fig.8 delete result in client(left) and server(right)

It indicates that the server automatically filters the repeat delete request and send the same response.

8.3.3.2. Overwrite

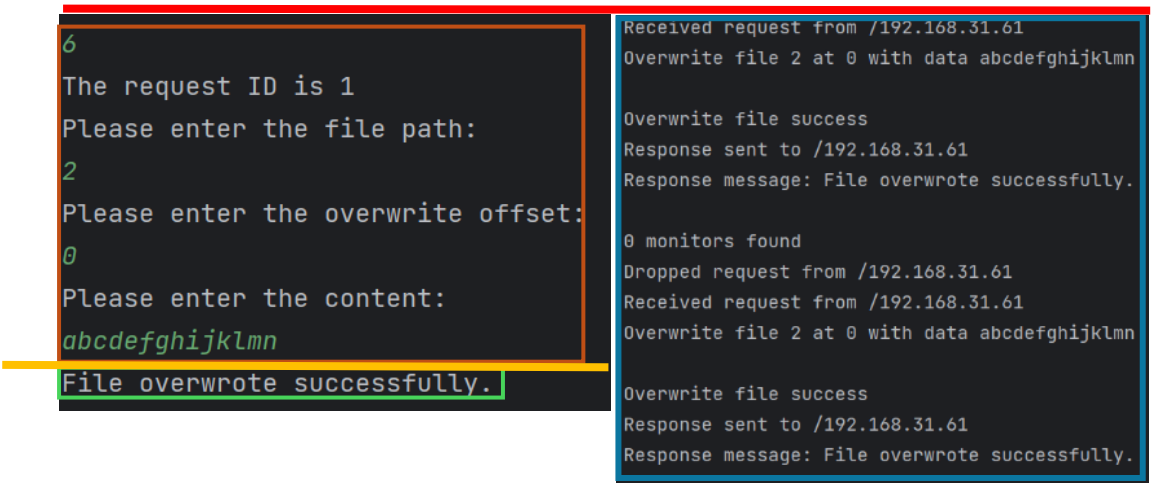

Fig.9 overwrite result in client(left) and server(right)

It indicates that the server redoes the overwrite request.

8.3.3.3. Move (not redo)

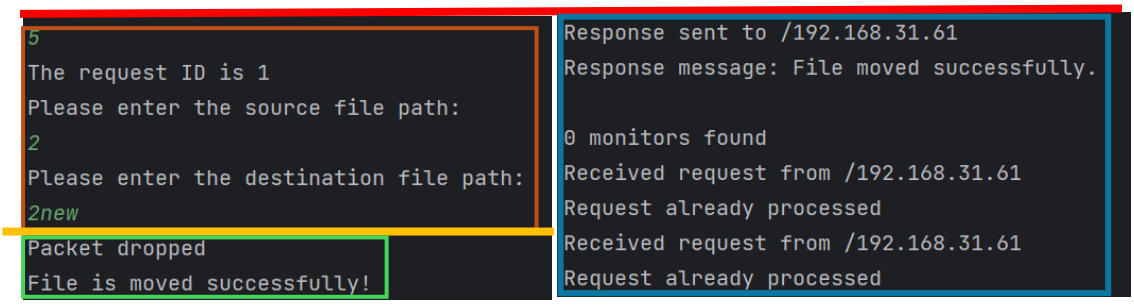

Fig.10 move without redoing result in client(left) and server(right)

It performs similarly to the delete and it successfully implemented the request and transmitted the correct response.

8.3.3.4. Move(redo)

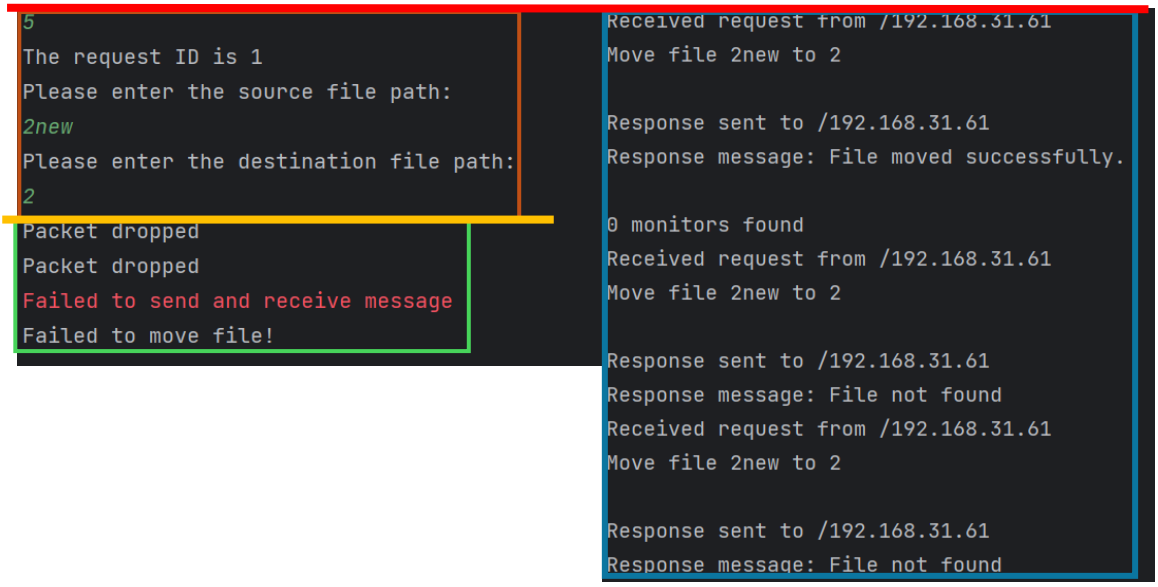

Fig.11 move with redoing result in client(left) and server(right)

It implements the file move but send the wrong message back as the packet drop. If it is necessary to reply for client, the server should not redo the move request.