CSCI 53700: Distributing Computing

Assignment Number 2

Submitted by: Anushka H Patil

Aim: This assignment is intended to reinforce the principles of the inter-process communication and the end-to-end argument. The aim is to develop a simplified version of a ftp program (similar to the one described in the end-to-end argument paper) using the stream sockets available in java.net package.

Model

My design is built on a client-server model architecture. The communication and data transfer take place using the stream sockets available in java.net package.

Socket: It is one endpoint of a two-way communication link between two programs running on different computers on a network. A socket is bound to a port number so that the transport layer can identify the application that data is destined to be sent to.

- **Server**: The server class extends *Thread*, which helps serve multiple clients concurrently. Following are the tasks performed by the server:
    - Client Authentication
    - Client Registration
    - File Transfer

  The communication required to perform the above tasks is thru *socket stream*. The server creates an instance of the ServerSocket and waits for the client to make a connection request. My server runs on a 10.234.136.55 and has a socket that is bound to port number 4444. When server receives a request, it accepts the request and binds it with the socket. Upon reception, the server gets a new socket, clientSocket, bound to the same local port, 4444, and has its remote endpoint set to the client's address and port. At this time, the new Socket object puts the server in direct connection with the client, we can then access the output and input streams to communicate with the client respectively until the socket is closed.

- **Client**: The server creates a new Socket object *socket*. The Socket constructor uses the host and port number of the server to establish the connection i.e. to make a connection request. When the server accepts the connection, it obtains an input output stream from it to communicate with the server.

Tasks:

- **Client Authentication**: The client sends its login credentials to the server. I have used a *user.txt* file to maintain the credentials for all of the clients at the server side. The server on reception of these credentials validates it with the content in the text file. If the credentials match, then the server responds a "*Successful*" message else "*Unsuccessful*" message.
- **Client Registration:** The client first registers itself with the server. On reception of these values, the server first checks if the username exists. If the username exists, the server responds with an error message and requests the client to enter another username. The server again checks whether the username exist, and the loop continues. If the username doesn't exist, then the server saves the credentials.
- **File Transfer**: After successful login or registration, the server asks the client where it wants to continue to transfer file or exit. If the client chooses to transfer file, server will

request the it for a filename. If the file exist, the server will transfer it else will respond "*File doesn't exist*" and the clients socket will be closed. The server will encrypt the file and compute the checksum of the data before transferring it. On the reception of the file, the client will decrypt it, compute a checksum and validate it with the server's checksum. If the checksums validate successfully, it indicates successful file transfer else the client requests the server to retransmit the file. The client will make 5 attempts to receive correct file. After 5 retries, a failure message will be displayed if the transfer is still unsuccessful and the socket connection will be closed.

My Design Highlights:

- **Multithreading**: The server has the capacity to service many clients and many requests simultaneously. Since, the server class extends Thread it makes it really convenient to create a new socket for every new client and service the client's request on a different thread. The server's main thread runs in a while loop to listen for a new connection request. I have overridden the threads run method which contains all the above functionalities implemented in it.
- **Encryption Mechanism**: I used a simple mechanism to encrypt the file at server's end and likewise decrypt it at client's end. I have reversed the String while sending it from the server, on reception of this string the client reverses the received string to obtain the original file.
  Encryption server side:

```
while (read != null) {
    String encrypt = "";
    // encryption -- reverse the string
    for (int i = read.length() - 1; i >= 0; i--) {
        encrypt += read.charAt(i);
    }
    output.println(encrypt);
                    }
```

  Decryption client side:

```
    while (!read.equalsIgnoreCase("null")) {
        String decrypt = "";
        // decryption -- reverse the string
        for (int i = read.length() - 1; i >= 0; i--) {
            decrypt += read.charAt(i);
        }
}
```

- **End-End Argument Implementation**: To verify the integrity of the file, I have made use of the *hashcod*e() function to compute the checksum of the file.
  The server sends this checksum to the client. After encrypting the received data, the client computes the checksum. If both of these checksum match, it indicates file transfer successful. Checksum is a way of ensuring the integrity and truthfulness of a file when it is received at the client's end. This checksum validation is performed at the ends which ensures that if any attempt is made to change the content will result in different checksum value. In this way we can detect whether the original file is altered or not. Since we perform it at the ends, any undesirable corruption during the communication can be easily noticed. Thus, using this mechanism my design provides a reliable transfer by validating the checksum for the entire data stream.

- **Byzantine Behavior**: To exhibit byzantine behavior at the server end, I have made use of a random number. If the random number is less than 0.5, it alters the value of the server checksum. The server then sends this wrong value to the client which results in end-end checksum error. The client thus has to request retransmission for the requested file. This is how I have introduced a byzantine behavior into my system.

Run Samples:

1) Server

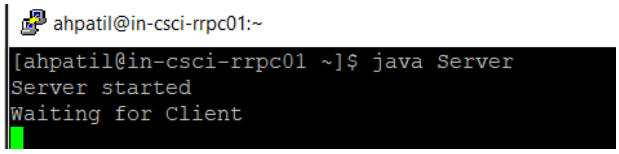

fig 1. Starting a server

2) Client Registration

a. Successful Registration

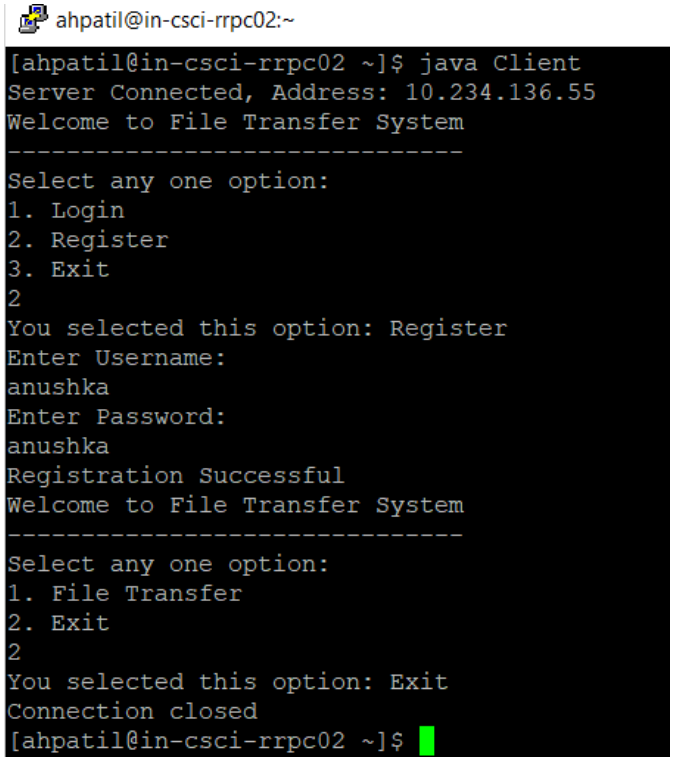

fig 2.a. Client Registration

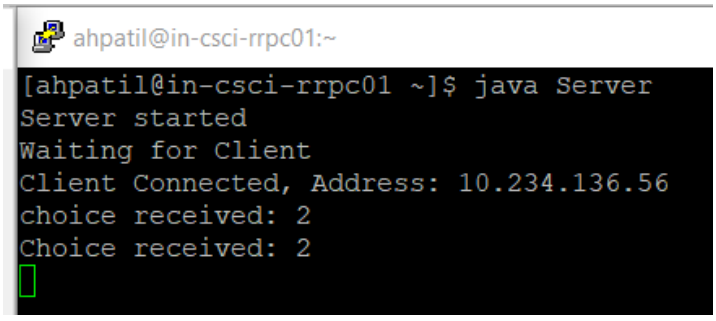

fig 2.b. Server

The client registers itself with the server. The server saves the credentials for future session.

b. Username already exists

```
[ahpatil@in-csci-rrpc03 ~]$ java Client
Server Connected, Address: 10.234.136.55
Welcome to File Transfer System
------------------------------
Select any one option:
1. Login
2. Register
3. Exit
2
You selected this option: Register
Enter Username:
anushka
Enter Password:
anushka
Username already exist, please try another username
Enter Username:
```

fig 2.c. Username already exist

The client enters username which already exist. The server sends him responds saying the username exist, please try another username.

3) Client Login

a. Successful Login

```
ahpatil@in-csci-rrpc02:~
1. Login
2. Register
3. Exit
1
You selected this option: Login
Enter Username:
anushka
Enter Password:
anushka
Authentication successful
Welcome to File Transfer System
------------------------------
Select any one option:
1. File Transfer
2. Exit
1
You selected this option: File Transfer
Enter the filename:
file1
Attempting to Receive File...
Receiving File...
File Transmission Successful
Connection closed
[ahpatil@in-csci-rrpc02 ~]$
```

```
ahpatil@in-csci-rrpc01:~
[ahpatil@in-csci-rrpc01 ~]$ java Server
Server started
Waiting for Client
Client Connected, Address: 10.234.136.56
choice received: 2
Choice received: 2
Client Connected, Address: 10.234.136.56
choice received: 1
anushka verified
Choice received: 1
Sending File: file1 to Client: anushka
Client Connection Closed, Address: 10.234.136.56
```

fig 3.a. Client logins using the credentials and is successfully authenticated by the server

The client enters the registered credentials and thus the server validates it successfully

b. Authentication Error

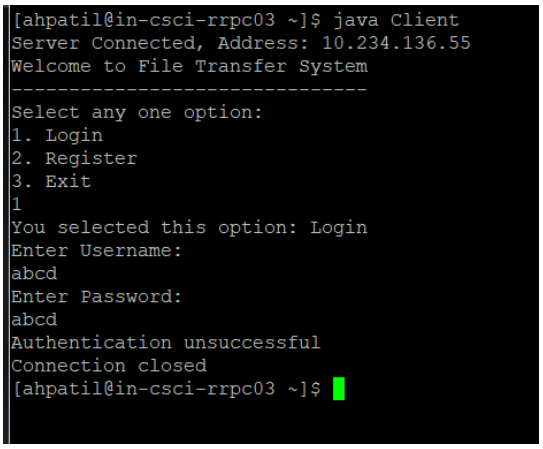

fig 3.b. Client logins using the credentials and is successfully authenticated by the server

The client enters the registered credentials and thus the server validates it successfully

4) File Transfer

a. Successful Transfer

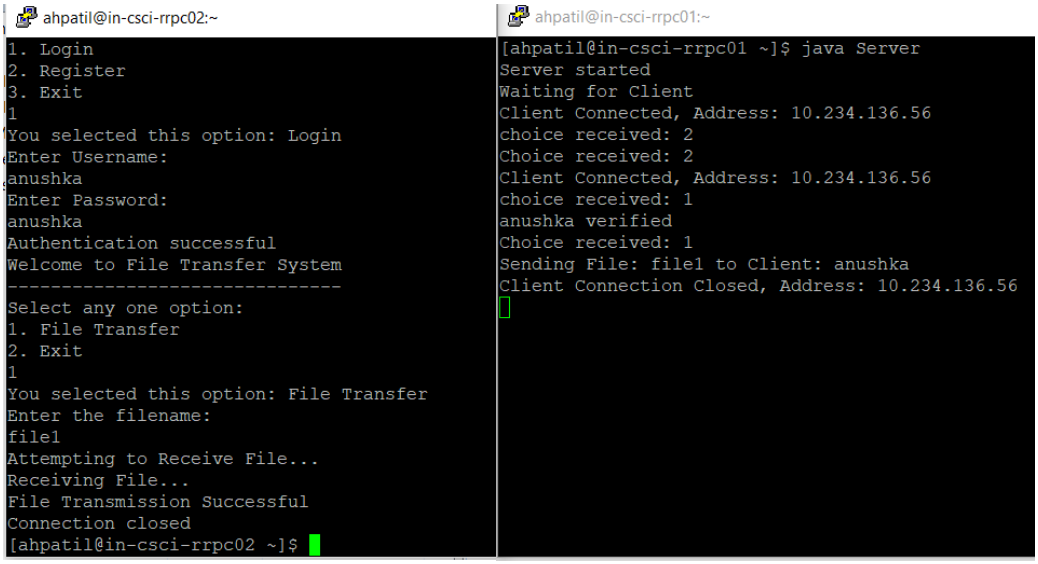

fig 4.a. After successful login, the client request for file1

After the client is successfully authenticated, the client requests the server for file1. The server checks if the file exist. In this case the file exist, thus the server encrypts the file and sends it to the client. On reception of the file. The client decrypts the file, validates the checksum. If the checksum is successfully validated, it indicates successful file transmission.

b. Error

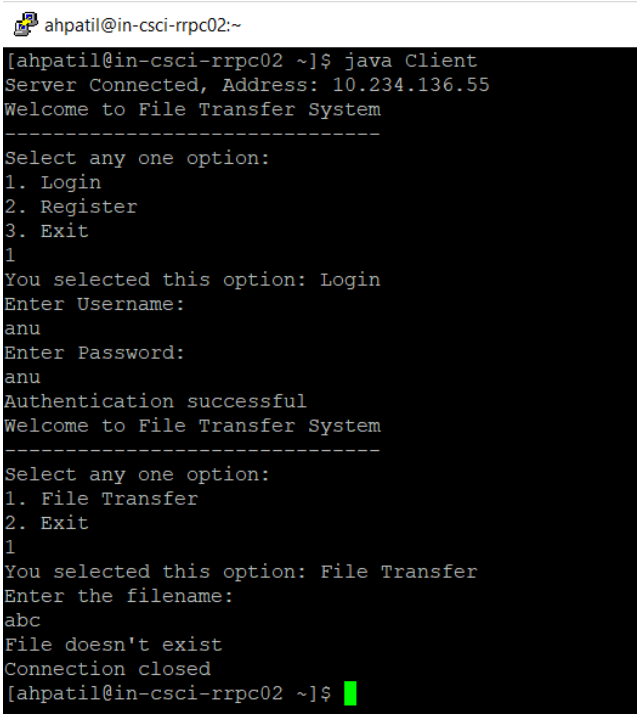

fig 4.b. After successful login, the client request for abc which doesn't exist

5) Multiple Client request

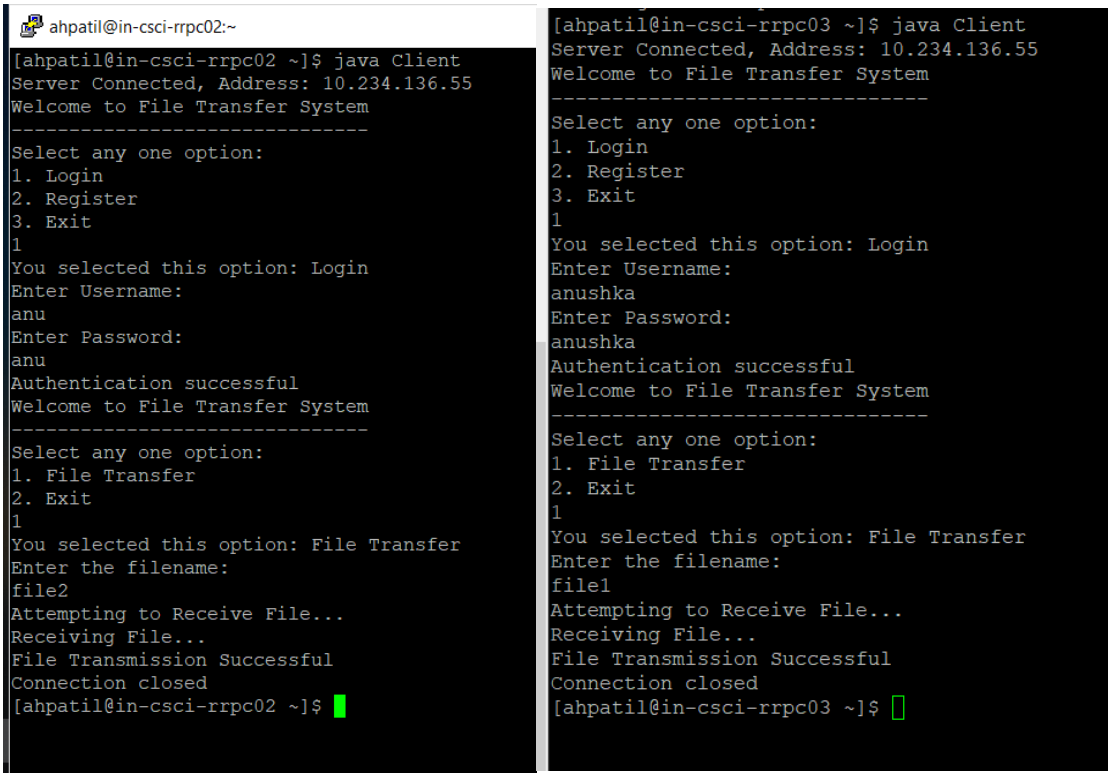

Fig 5.a. Client1: anu (left) and Client anushka (right)

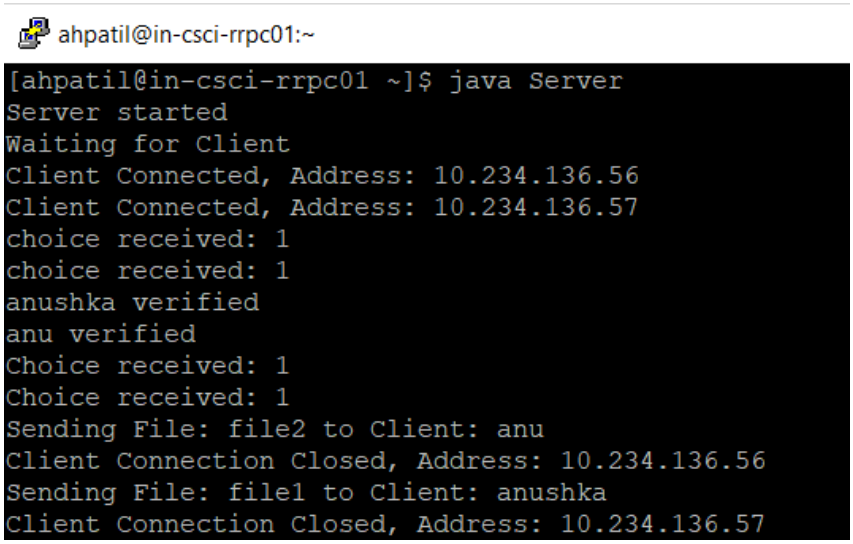

Fig 5.b. Server receiving requests from both the clients simultaneously

Client 1 (anu) and Client 2 (anushka) send request simultaneously to the server. The server serves both the clients. The figures above show how both the clients are served by the server.

6) Byzantine Failure

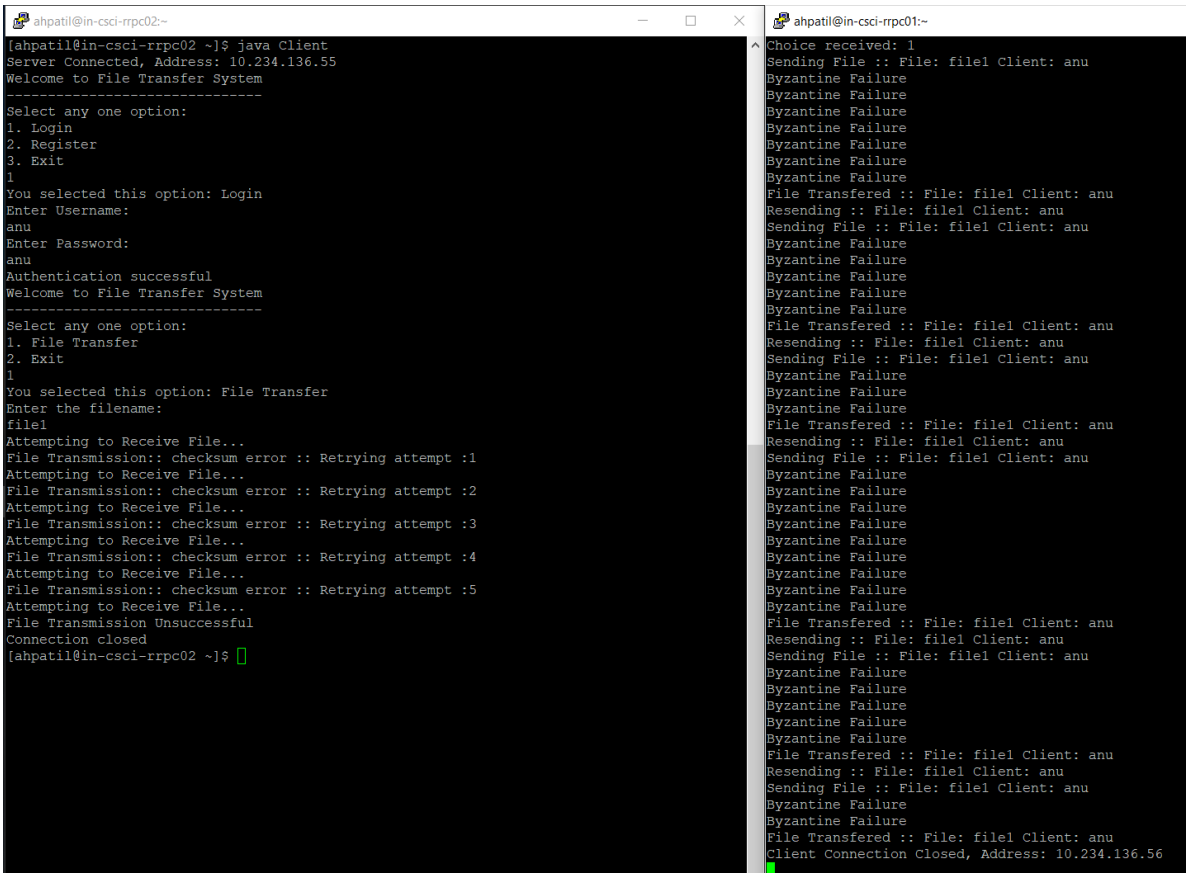

Fig 6.a. Byzantine Behavior

If there is a checksum error, it makes 5 attempts to obtain the correct file. After 5 attempts it displays message saying, “File Transmission Unsuccessful” and the socket is closed.

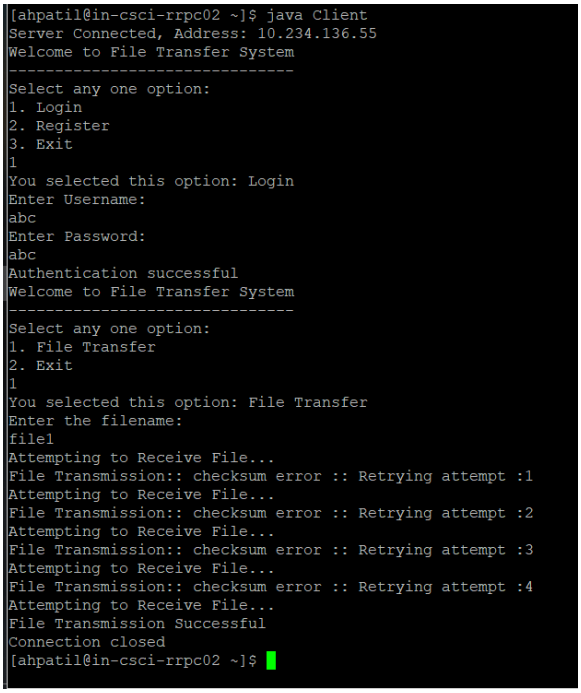

Fig 6.b. Client instance

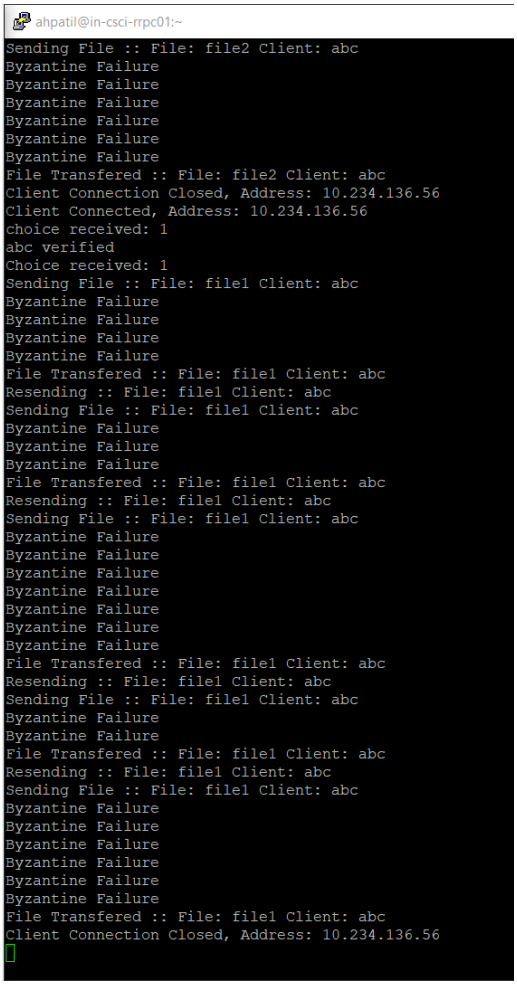

Fig 6.c. Server Instance

The client receives a checksum error for first 4 attempts on 5th attempt it successfully receives the file.

Pros:

- The server can serve multiple clients concurrently
- The model has end-end mechanism to ensure the integrity and truthfulness of the file
- The model makes use of encryption mechanism to provide security for file being transferred
- Capable of transferring all sizes of data

Cons:

- The client remains blocked until the server processes it's request. There is no timeout mechanism thus, the client might be blocked until the server is closed or any error occurs.
- Since, the server creates a thread for every client. If there are numerous clients simultaneously there will be numerous threads. It will be requiring more computation to manage when the amount is in millions.

Conclusion:

Thus, I have successfully designed a file transfer program which reinforces the principles of the inter-process communication and the end-to-end argument. The model allows secure file transfer by providing data protection through encryption. The model has end-end checksum implementation which ensures the integrity and truthfulness of the file being transferred.