

Introduction:

The purpose of this assignment was to implement the simplified file transfer version as discussed in the end to end argument paper. This assignment required the implementation of stream sockets using the java.net and the java.io packages.

The program has two entities a client and a server. The client first authenticates itself with the server for file access. After authentication, the client requests a file to be sent from the server. The server then encrypts the contents of the file and sends to the client. The server also sends a checksum and then the key to decrypt the message. The message is then decrypted and then written to the file. The contents of the file are read again and at the client end and a checksum is calculated. This is verified with that which is sent by the server. If the checksums match, the contents of the file at both ends are identical. Otherwise the client will request a retry. The retry is permitted only a certain number of times and after that the server will close the connection.

End to End Argument Principle:

"The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the end points of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible. (Sometimes an incomplete version of the function provided by the communication system may be useful as a performance enhancement.)"[1]

End to End Argument Meaning:

The end to end argument states that certain functionalities like error control, security and routing have to be implemented at the end points rather than the communication network. Even though the communication network may correctly transfer the contents of the file the errors at the application level or faults at the computer hardware level may cause the file transfer to be unsuccessful. Also the example of MIT's network in which each of the gateways had implemented checksums failed to prevent error in transmission as one of the gateways developed a transient error. So, it is suggested by the end to end argument that it is better to move up these functionalities into the application layer rather than rely on the transport layer.

End to End Argument Consequences:

The consequences of the end to end argument are that the application layer has to deal with the burden of the added functionalities which were being implemented by the communication layer. As suggested by the critical review of the end to end argument it is better to move these functionalities to the transport layer as nowadays the networks are quite reliable and faults occurring at the communication level are less and the transport layer also performs these functionalities. So if the application layer performs these functions it will cause redundancy checks and will waste time. The application layer must be able to reliably trust the transport layer and the application layer must not be overwhelmed by the increased functionalities which are tedious to implement and which results in redundancy.

Implementation of the Program:

This assignment has been completed using Java. I have used stream sockets from the Java package of java.net. The communication between the client and the server is done using the functionalities given in the java.io package. The functionalities in both the packages were combined to implement the program in this assignment.

Logics of the program:

I have implemented the program in two separate files placed in two different folders one for client and the other for server. Hence, there are two .java files which contain two main classes that is the mServer1 class and the mClient1 class.

Variables, objects and their functionalities in the mServer1 class:

Packages imported: java.io.*, java.util.* and java.net.*

Name of the class containing the main method: mServer1-This class contains the main method.

- 1.ServerSocket server: This creates a socket which is a server socket which will accept connections on the port number which is passed as argument.eg: I have passed 10444 as port number to accept connections.
- 2.Socket Sock: This creates a socket from the previous step of ServerSocket but it will be the server's socket. It defines the endpoint of communication between the server and client like a common portal.
- 3.PrintStream PS: It is used to send out the message to the output stream using the Socket to the client.
- 4.InputStreamReader IR: It is used to read the input stream from the client using the socket.
- 5.BufferedReader br: It is used to read the message sent by the client.
6. File f: It is used to create a new file instance by loading the file from the path specified.
- 7.f.exists(): It is used to check if the file exists on the server side.
- 8.boolean serverflag: It is used as a check to stop the retries after the retry limit has been reached.
- 9.int retrycounter: It is used to count the number of retries.
- 10.int retrycountervary: It is a parameter used to vary the number of retries.
- 11.FileInputStream fin: It is used to create an input stream from the given file which is passed as argument.
12. byte[] buf: It is a byte array used to store the contents of the file in byte format.
- 13.fin.read(): It is used to read the contents of the file from the inputstream of the file into the byte array passed as argument from the start position to the end position.

14. `java.util.zip.CRC32` x: Creates a class that computes the CRC32 checksum of a data stream.
15. `x.update(buf)`: Calculates the CRC32 checksum of the contents of the `buf` byte array.
16. `x.getValue()`: Used to get the CRC32 value calculated by the update method.
17. `OutputStream os` : It is used to represent the output stream which has to be sent from the Socket to the client from the server.
18. `BufferedOutputStream out`: It is used to write out the output stream to the socket from the server to the client.
19. `out.write(buf, 0, buf.length)`: It is the function used to write out the specified byte array called `buf` from starting index zero to its length.
20. `out.flush()`: It is used to flush out any unwritten buffered bytes.
21. `Sock.close()`: It is used to close the socket connection from the server side by calling the `close()` function.

Variables, objects and their functionalities in the `mClient1` class:

Packages imported: `java.io.*`, `java.util.*` and `java.net.*`

Name of the class containing the main method: `mClient1`-This class contains the main method.

Most of the functions and objects in the client are similar to the ones in the server class. The only difference is that they implement the functionalities of the client or the client side. The ones which are different are described below:

1. `Socket Sock`: This creates a socket for the client and passes in the address and the port number on which the server socket is listening.
2. `FileOutputStream inFile`: It is used to create an output stream `inFile` passing the file path in which the new file is created.
3. `InputStream is`: It is used to represent the input stream which has to be received from the Socket from the server to the client.
4. `BufferedInputStream in2`: It is used to receive the input stream to the socket from the server to the client.
5. `in2.read(b, 0, 1024)`: It is used to read the input stream into the byte array `b` of a specified index from zero to 1024.
6. `byte[] b`: It is used to store the inputstream containing the bytes of the message sent by the server from the file.
7. `boolean clientflag`: It is used as a check to stop the request of the client after the server sends a closing connection message.

String functions used:

1.equalsIgnoreCase: It is used to verify the message or string i.e., the string matches without upper or lower case check.

2.contains: It is used to check if the particular message or string contains a particular sequence of string characters.

Discussion of the various aspects of the implementation of my design as per End to End Argument:

Interaction Model:

1. The communication followed by the design is that of an Inter process communication as Inter process communication refers to the relatively low-level support for communication between processes in distributed systems, including message-passing primitives, direct access to the API offered by Internet protocols (socket programming) and support for multicast communication. The client process communicates with the server and the server also communicates with the client back.

2. The architecture is that of a client-server model. The client authenticates itself with the server and then requests a file to be sent from the server and the server sends the file. Also the communication entities are two namely the client and the server.

3. It follows the communication style of request-reply as the client each time requests the server of its service via message passing involving pair wise exchange of messages from the client to server and then server to client.

How the client and server interact in the program:

1. First the server starts its socket and is listening on a specified port number for the connection from the client.

2.The client then starts its socket specifying the address and the port number which will ensure that the server is uniquely identified and it will get connected to the intended server only.

3.Once the client connects to the server immediately the server sends an authentication required message and the client displays on its console this message.

4.The client then types in an authentication message like a password which is "HelloArvindserver" which if the letter sequence matches the client is given access.

5.If the password character sequence is wrong then the server sends the message that authentication has failed and closes the connection.

6.After authentication is done, the server sends an authentication approved message to the client and the client upon receiving that message will send a request for file message.

7. The server will then receive the request for file message and then will send a message indicating the name of the file to be sent. This will be displayed on the console of the client and the client will then enter the name of the file to be sent.
8. The server will then check if the file specified by the client exists. If it does then it will send message to the client indicating that the file is present and be ready to receive it.
9. If the file does not exist then the server will send a message to the client that the file does not exist and that the server is closing the socket connection.
10. The client upon receiving the message that the file is present on the server will be ready to receive it.
11. The checksum of the byte array contents is calculated using CRC32 class function in Java.
12. The server will first read the message contents from the file into a byte array and then using shift cipher in which the key is 5 and the bytes are shifted by adding 5 i.e., the key forward to each element in the byte array.
13. Then the byte array is then written to the output stream using sockets.
14. The client then reads the byte array from the input stream using sockets.
15. The server then sends the checksum to the client which the client will also receive from the socket.
16. The server then sends the key to the client which the client receives in string format and the client then converts it into integer format to decrypt the message accordingly.
17. The client then shifts back or subtracts 5 i.e., the key sent by the server to the byte array and then stores back the into the byte array at the client side.
18. The client then creates a file instance and stores the decrypted message into the file.
19. After that the message from the file is read into a byte array again and its checksum using CRC32 is taken and it is verified with the one that is received from the server.
20. If the CRC32 checksum matches then the client sends the message Success indicating that it is not necessary to resend the file. The client will set its flag as false and then exits the while loop.
21. The server on receiving the Success message from the client will then set its own flag as false and then close the socket and exit it's while loop.
22. If the CRC32 checksum does not match then the client will send the message Resend indicating that the file has to be resent as the contents are not matching.
23. The server will repeat the steps above and will count the number of retries.

24.If it has reached the limit of number of retries using retryvary variable then the server will then send a message to the client that the limit of retries are reached and the server is closing down the connection. It will set its flag as false and close the socket.

25.The client upon receiving the closing message will also set its flag as false and exit the while loop.

Interaction Model Diagrams:

Case 1:

The client does not send correct authentication credentials to the server.

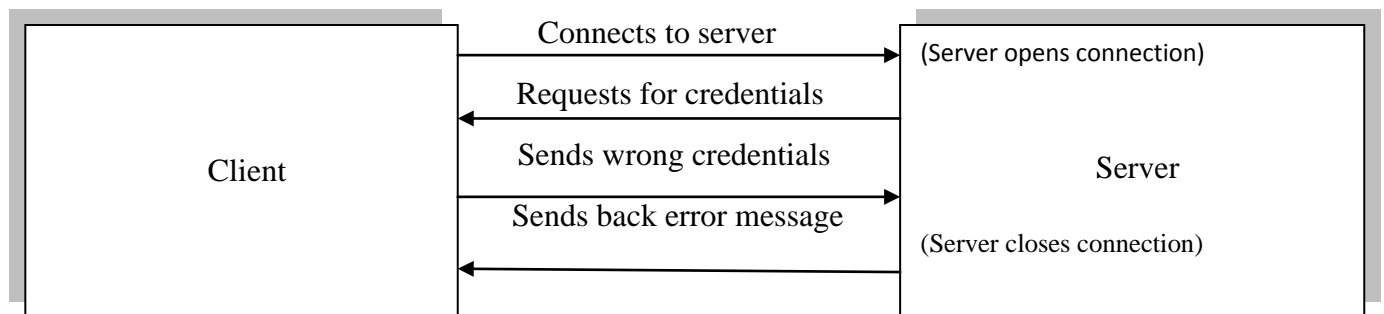


Fig 1.Case 1 Interaction Diagram

Case 2:

The client requests a file that does not exist on the server.

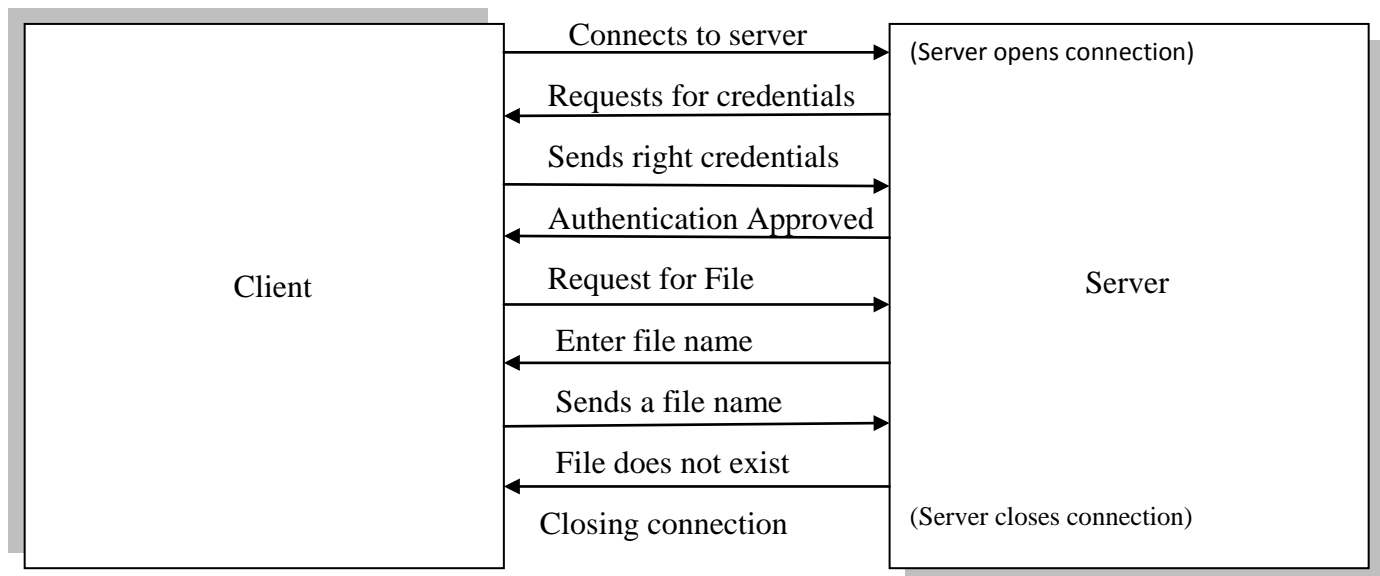


Fig 2.Case 2 Interaction Diagram

Case 3:

The file transfer is successful in first attempt.

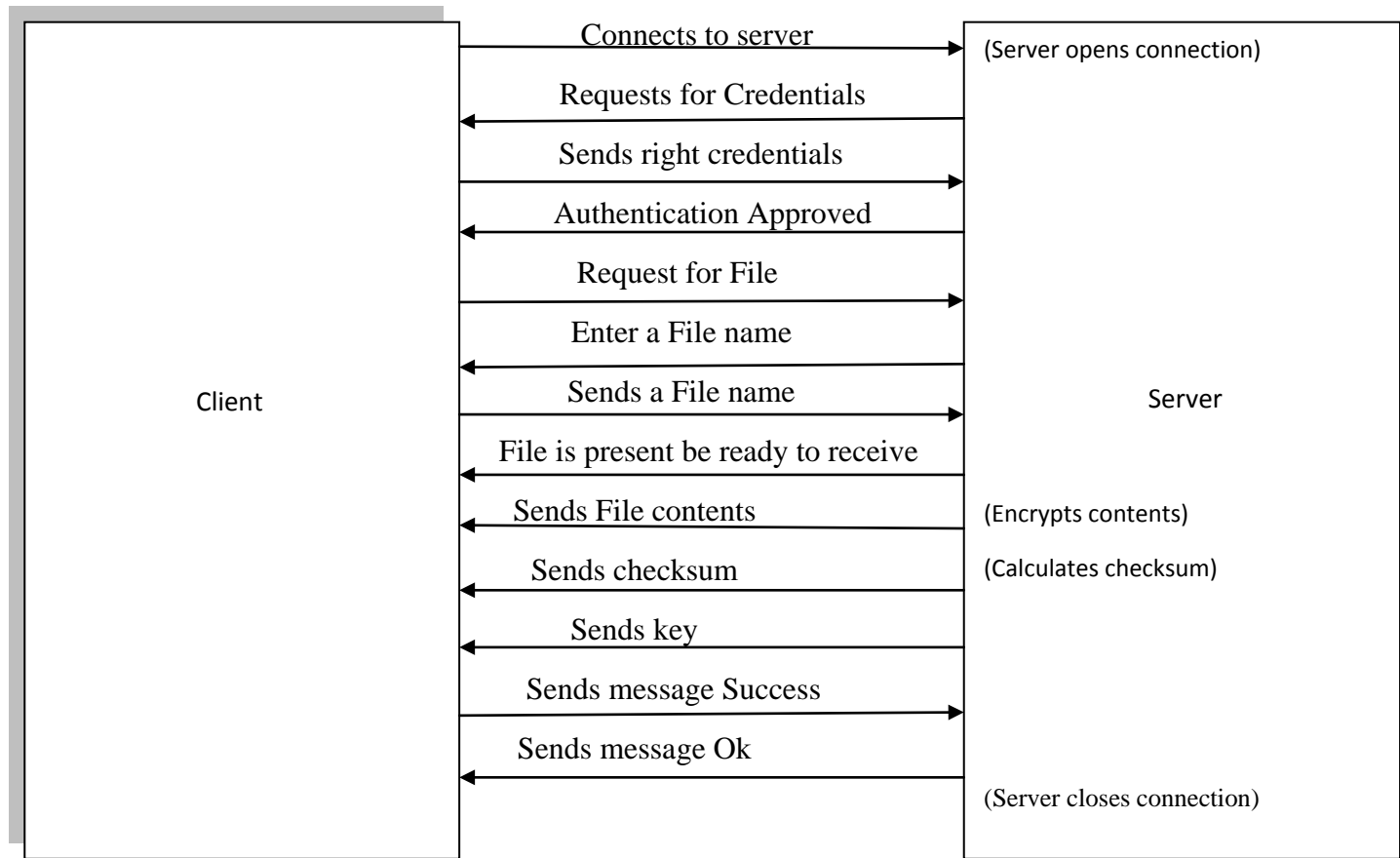


Fig 3.Case 3 Interaction Diagram

Case 4:

The file transfer is not successful in first attempt and retries and succeeds.

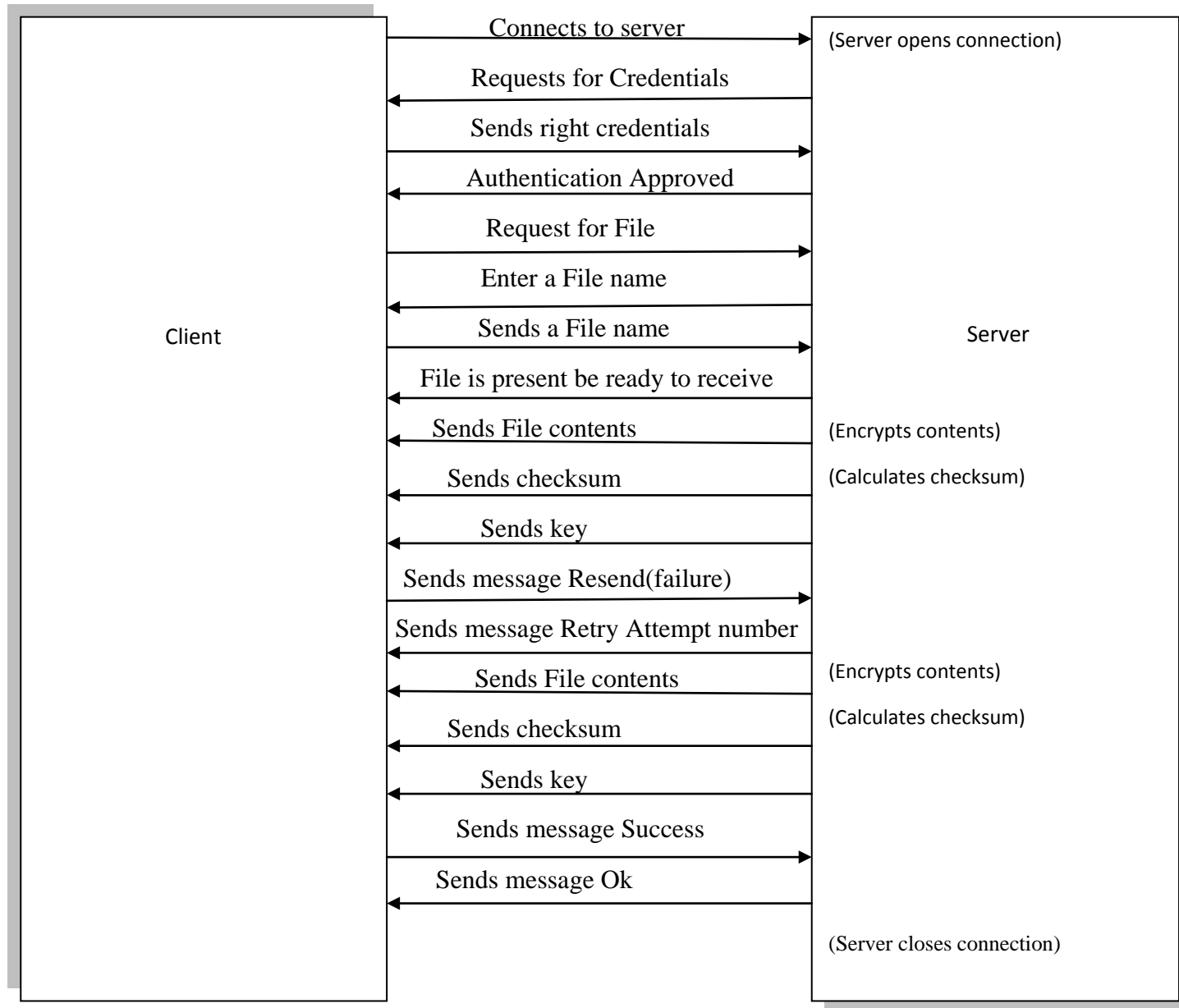


Fig 4.Case 4 Interaction Diagram

Case 4:

The file transfer is not successful in all attempts and retries till retry counter limit is reached.

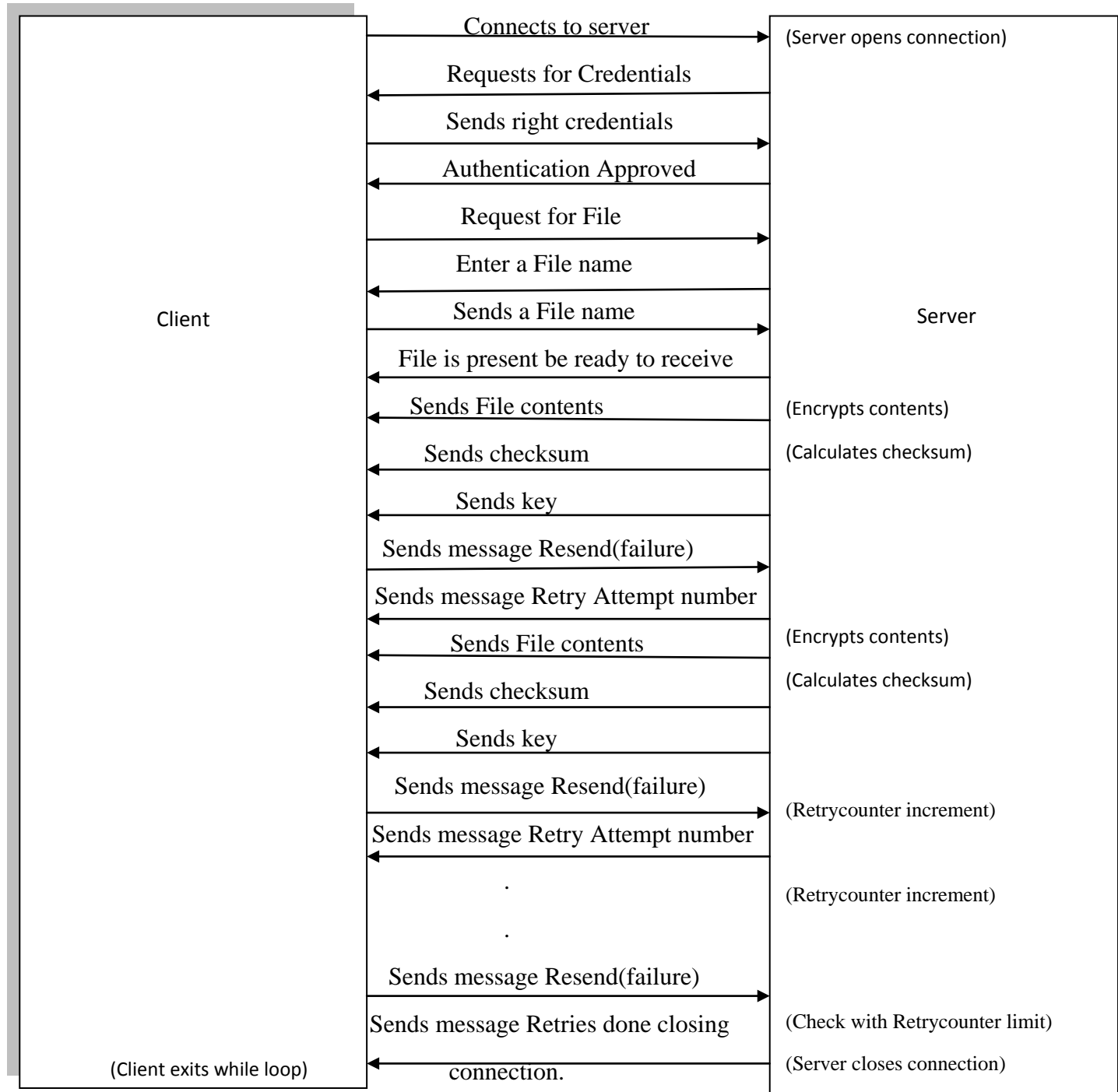


Fig 5.Case 5 Interaction Diagram

Failure Models:

1. In my design the checksum at the client's end is taken by reading the contents of the file again after transfer on the client machine from the server machine to ensure that the file was written correctly onto the client machine and then the checksum of its contents was taken. But while reading the file at server side the file was read wrongly into the byte array due to hardware or software fault then the checksum at server will be wrong as it calculates checksum from byte array and the file contents at client side will not match the file at server side but will match that of the byte array contents at server side. So the operation has failed and the file contents do not match.

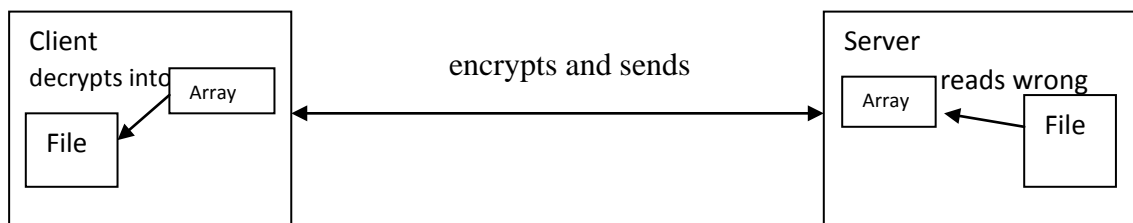


Fig 6. Diagram for faulty file content read at server end into byte array

2. If there is a persistent hardware error in the underlying network connection or the network gets disconnected midway then the file will never be transferred as the retry limit would be reached. Of course increasing the retry limit would not solve the problem. The network connection must be fixed.

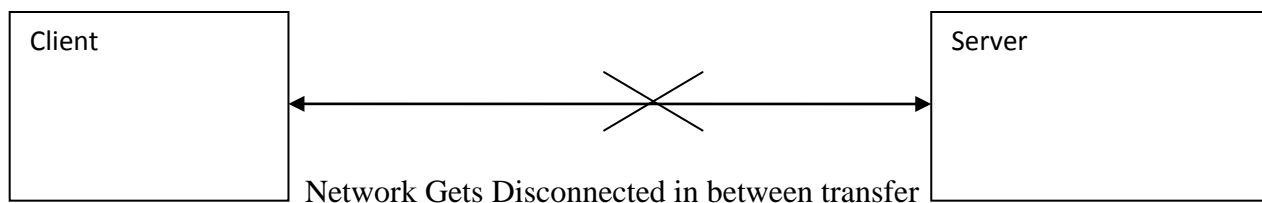


Fig 7. Diagram for Network Error/Failure

3. If memory is full at the client or the file size exceeds the client's available memory then the file transfer will cause unexpected behavior and will never be complete.

4. If the software due to some unknown reason writes the file contents wrongly into the system at the client side every time then the file transfer will never be successful. eg: The file size is 10Mb but the client has only 5 Mb free.

5. In my design, I have fixed the client byte array size to receive the file contents from the server as 1024 bytes. If the file at the server side exceeds 1024 bytes then there will be an overflow and the file transfer will not be successful.

Security Models:

1. In my design, the contents are transferred then the checksum and finally the key. So if there is a Man-in-the-Middle attack then the person can also copy the contents and then retrieve the message using the key impersonating as the server to the client and vice versa. The main problem is that the key is sent along the same channel as that of the encrypted message.

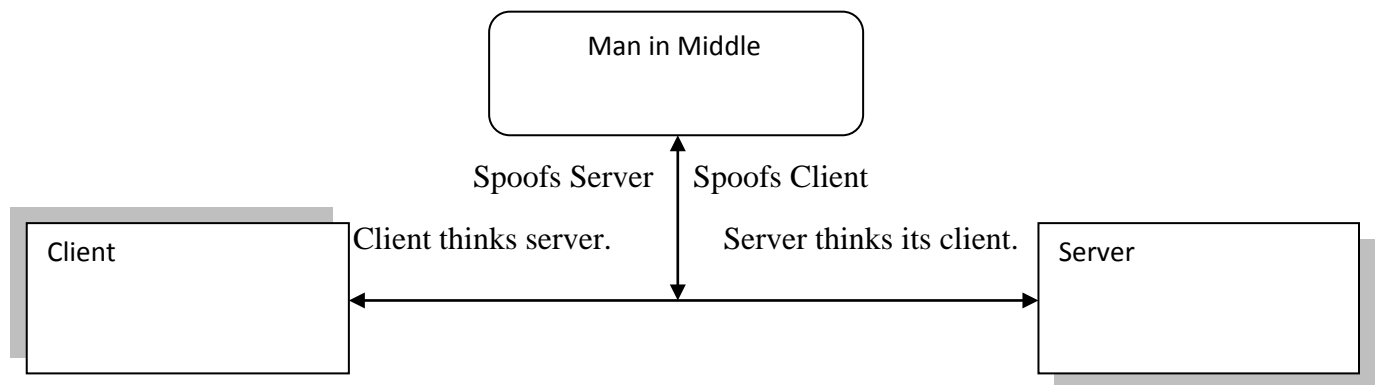


Fig 8. Man in The Middle Attack Diagram

Possible Solutions:

1. One solution is to send the key through another secure channel known only to the sender and receiver or a channel in which only they have access like a private or closed network.

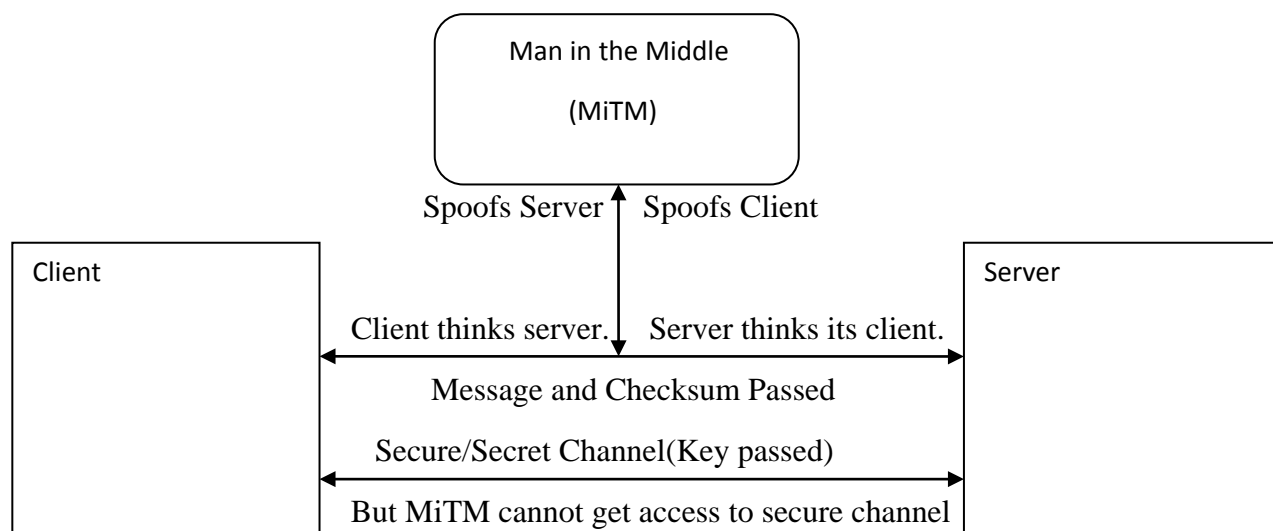


Fig 9. Man in the middle prevention via Secure Channel Diagram

2. Another solution would be to know the key beforehand without communicating on the network.

2. In my design, the address and the port number of the socket is fixed and hence if known to some attacker then he/she can send repeated requests and always close down the connection and the client will never be able to connect to the server.

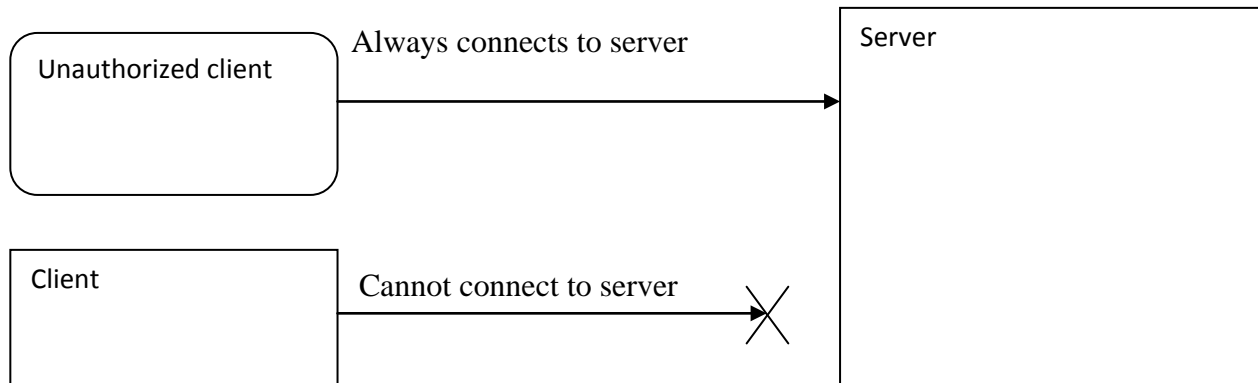


Fig 10. Unauthorized client connection Diagram

Possible Solution:

1. Only make the port number and address known to the genuine client and keep changing port numbers.

3. In my design, the server authentication password is simple and hence if brute forced can be known. But as the socket connection always closes on wrong credentials, that can be detected and the person querying can be banned after some attempts.

Sample Outputs and Explanation for 3 Cases:

Case 1:

In Case 1, the file transfer is successful and the program executes and performs the functionality as required. Here the client authenticates with the server and the server then asks for file name to be transferred. The client gives the name of the file that is available and the server transfers it. The server also sends the checksum and the key. The client then calculates the checksum and sends the message that the transfer is successful.

```

arvnair@sl253-rrpc02:~/NairA2/A2Case1/Client
login as: arvnair
arvnair@10.234.140.28's password:
Access denied
arvnair@10.234.140.28's password:
Last login: Tue Oct 28 12:21:30 2014 from 10.182.4.233
[arvnair@sl253-rrpc02 ~]$ ls
A2 dc files _files.submitted.log NairA2
A2 dc files _files.submitted.log NairA2
[arvnair@sl253-rrpc02 ~]$ cd NairA2
[arvnair@sl253-rrpc02 NairA2]$ cd A2Case1
[arvnair@sl253-rrpc02 A2Case1]$ ls
Client Server
[arvnair@sl253-rrpc02 A2Case1]$ cd Client
[arvnair@sl253-rrpc02 Client]$ ls
clientmakefile.sh mClient1.java
[arvnair@sl253-rrpc02 Client]$ javac mClient1.java
[arvnair@sl253-rrpc02 Client]$ java mClient1
Enter the Credentials to access server:
helloarvindserver
Authentication Approved!
Enter the file you want to send:
F1.txt
File is present Be Ready to receive it!
File transfer is successful!
[arvnair@sl253-rrpc02 Client]$

arvnair@sl253-rrpc01:~/NairA2/A2Case1/Server
login as: arvnair
arvnair@10.234.140.27's password:
Last login: Wed Oct 29 07:01:30 2014 from 140-182-65-83.ssl-vpn.iupui.edu
[arvnair@sl253-rrpc01 ~]$ ls
A2 dc files _files.submitted.log NairA2
[arvnair@sl253-rrpc01 ~]$ cd NairA2
[arvnair@sl253-rrpc01 NairA2]$ ls
A2Case1 A2Case2 A3Case3 Sampleoutputs
[arvnair@sl253-rrpc01 NairA2]$ cd A2Case1
[arvnair@sl253-rrpc01 A2Case1]$ ls
Client Server
[arvnair@sl253-rrpc01 A2Case1]$ cd Server
[arvnair@sl253-rrpc01 Server]$ ls
F1.txt mServer1.java servermakefile.sh
[arvnair@sl253-rrpc01 Server]$ javac mServer1.java
[arvnair@sl253-rrpc01 Server]$ java mServer1
[arvnair@sl253-rrpc01 Server]$
  
```

Fig 11.Sample output Screenshot for Case 1

Case 2:

Here in order to check if the retries are working or not I have introduced a failure that if the checksum is calculated by the client then while checking it becomes zero and never matches that of the checksum sent by the server. Hence, it retries every time and after the retry counter limit has been reached the server sends a message that retries are done and it is closing the connection. The client also then exits the while loop after this message.

```

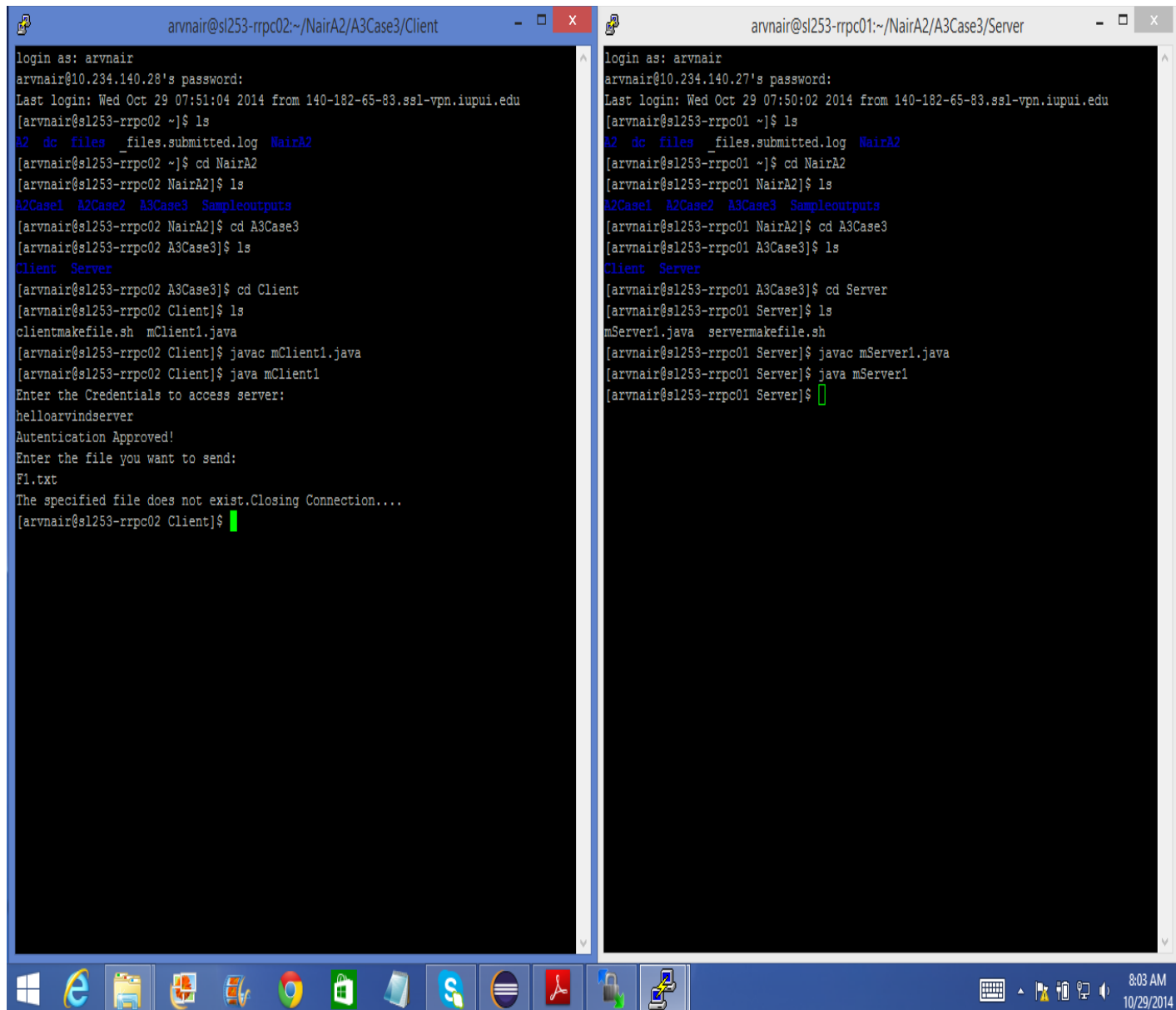
arvnair@sl253-rrpc02:~/NairA2/A2Case2/Client
login as: arvnair
arvnair@10.234.140.28's password:
Last login: Wed Oct 29 08:08:03 2014 from 140-182-65-83.ssl-vpn.iupui.edu
[arvnair@sl253-rrpc02 ~]$ cd NairA2
[arvnair@sl253-rrpc02 NairA2]$ cd A2Case2
[arvnair@sl253-rrpc02 A2Case2]$ cd Client
[arvnair@sl253-rrpc02 Client]$ java mClient1
Enter the Credentials to access server:
helloarvindserver
Autentication Approved!
Enter the file you want to send:
F1.txt
File is present Be Ready to receive it!
Retry server msg Retry attempt 1
Retry server msg Retry attempt 2
Retry server msg Retry attempt 3
Retry server msg Retry attempt 4
Retry server msg Retry attempt 5
Retry server msg Retry Attempts are done. Closing connection....
[arvnair@sl253-rrpc02 Client]$

arvnair@sl253-rrpc01:~/NairA2/A2Case2/Server
login as: arvnair
arvnair@10.234.140.27's password:
Last login: Wed Oct 29 08:04:38 2014 from 140-182-65-83.ssl-vpn.iupui.edu
[arvnair@sl253-rrpc01 ~]$ cd NairA2
[arvnair@sl253-rrpc01 NairA2]$ ls
A2Case1 A2Case2 A3Case3 Sampleoutputs
[arvnair@sl253-rrpc01 NairA2]$ cd A2Case2
[arvnair@sl253-rrpc01 A2Case2]$ ls
Client Server
[arvnair@sl253-rrpc01 A2Case2]$ cd Server
[arvnair@sl253-rrpc01 Server]$ ls
F1.txt mServer1.class mServer1.java servermakefile.sh
[arvnair@sl253-rrpc01 Server]$ java mServer1
Exception in thread "main" java.lang.NullPointerException
    at mServer1.main(mServer1.java:79)
[arvnair@sl253-rrpc01 Server]$ java mServer1
[arvnair@sl253-rrpc01 Server]$
  
```

Fig 12.Sample Output Screenshot for Case 2

Case 3:

In this case, I have removed the F1.txt file from the Server folder so when the Client requests for that particular file then it must show the expected output that the file does not exist and must close the connection. So, when the program in this case is run the client requests F1.txt file and the server checks if the file exists in its folder. Since it was deleted the server sends a message that the specified file does not exist and that it is closing the connection.



```
arvnair@sl253-rrpc02:~/NairA2/A3Case3/Client
login as: arvnair
arvnair@10.234.140.28's password:
Last login: Wed Oct 29 07:51:04 2014 from 140-182-65-83.ssl-vpn.iupui.edu
[arvnair@sl253-rrpc02 ~]$ ls
A2 dc files _files.submitted.log NairA2
[arvnair@sl253-rrpc02 ~]$ cd NairA2
[arvnair@sl253-rrpc02 NairA2]$ ls
A2Case1 A2Case2 A3Case3 Sampleoutputs
[arvnair@sl253-rrpc02 NairA2]$ cd A3Case3
[arvnair@sl253-rrpc02 A3Case3]$ ls
Client Server
[arvnair@sl253-rrpc02 A3Case3]$ cd Client
[arvnair@sl253-rrpc02 Client]$ ls
clientmakefile.sh mClient1.java
[arvnair@sl253-rrpc02 Client]$ javac mClient1.java
[arvnair@sl253-rrpc02 Client]$ java mClient1
Enter the Credentials to access server:
helloarvindserver
Authentication Approved!
Enter the file you want to send:
Fl.txt
The specified file does not exist.Closing Connection....
[arvnair@sl253-rrpc02 Client]$

arvnair@sl253-rrpc01:~/NairA2/A3Case3/Server
login as: arvnair
arvnair@10.234.140.27's password:
Last login: Wed Oct 29 07:50:02 2014 from 140-182-65-83.ssl-vpn.iupui.edu
[arvnair@sl253-rrpc01 ~]$ ls
A2 dc files _files.submitted.log NairA2
[arvnair@sl253-rrpc01 ~]$ cd NairA2
[arvnair@sl253-rrpc01 NairA2]$ ls
A2Case1 A2Case2 A3Case3 Sampleoutputs
[arvnair@sl253-rrpc01 NairA2]$ cd A3Case3
[arvnair@sl253-rrpc01 A3Case3]$ ls
Client Server
[arvnair@sl253-rrpc01 A3Case3]$ cd Server
[arvnair@sl253-rrpc01 Server]$ ls
mServer1.java servermakefile.sh
[arvnair@sl253-rrpc01 Server]$ javac mServer1.java
[arvnair@sl253-rrpc01 Server]$ java mServer1
[arvnair@sl253-rrpc01 Server]$
```

Fig 13.Sample Output Screenshot for Case 3

References: 1.*End-to-End Arguments in System Design*, J.H. Saltzer, D.P. Reed and D.D. Clark*, M.I.T. Laboratory for Computer Science.[1]

2. *A critical review of “End-to-end arguments in system design”*, Tim Moors, Polytechnic University.

3. *DISTRIBUTED SYSTEMS Concepts and Design Fifth Edition*, George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair, Addison Wesley Publications.

4. java.net and java.io packages from javadocs.

5. Class notes and slides on End-to-End ppt and End-to-End critical review ppt.