

# IMPLEMENTATION REPORT

Prepared By: Group F

Team Members:

Gayatri Hiregoudar

Ksenia Fedorovna

Rojesh Uprety

Romaan Khadeer

**Design Specification  
For  
Simple Reliable Transport Protocol (SRTP)  
And  
File Transfer Application**

**Assignment 1**

## Contents

Introduction .....	4
Protocol Functionality .....	4
SRTP Protocol .....	5
System Design Model .....	5
FTP layer .....	7
Ftclient .....	7
Ftserver .....	7
Reliable Layer .....	7
Functions implemented on the client side are: .....	7
Concurrency .....	8
Go Back n Layer .....	8
Circular Queue: .....	9
Handling Duplicate Packets and Acknowledgements .....	10
Flow control .....	10
Acceptance Testing .....	12
Instruction for Use .....	12
Test Results .....	12
Test Case I: General Flow of the program and SRTP implementation .....	12
Test case II: Re-transmission of the same file .....	14
Test case III: Establishing the connection between the client and the server with different port number .....	15
Test Case IV: Implementation of the concurrency of the system .....	16
Test case V: Implementation of the concurrency, both the clients trying to send the same file .....	17
Test Case VI: Testing network performance using Iperf .....	18
Test Case VII - X: Using Netem command to evaluate network reliability .....	19
References .....	22

## Introduction

We have implemented the key functionalities of a simple, reliable transport protocol on top of UDP, as a part of a file transfer application. UDP provides an unreliable connectionless transport service over network layer.

The main characteristics of UDP are:

- SDUs not larger than 65507 bytes
- Loss or out of order SDU packets
- SDUs delivered to the destination are not corrupted

## Protocol Functionality

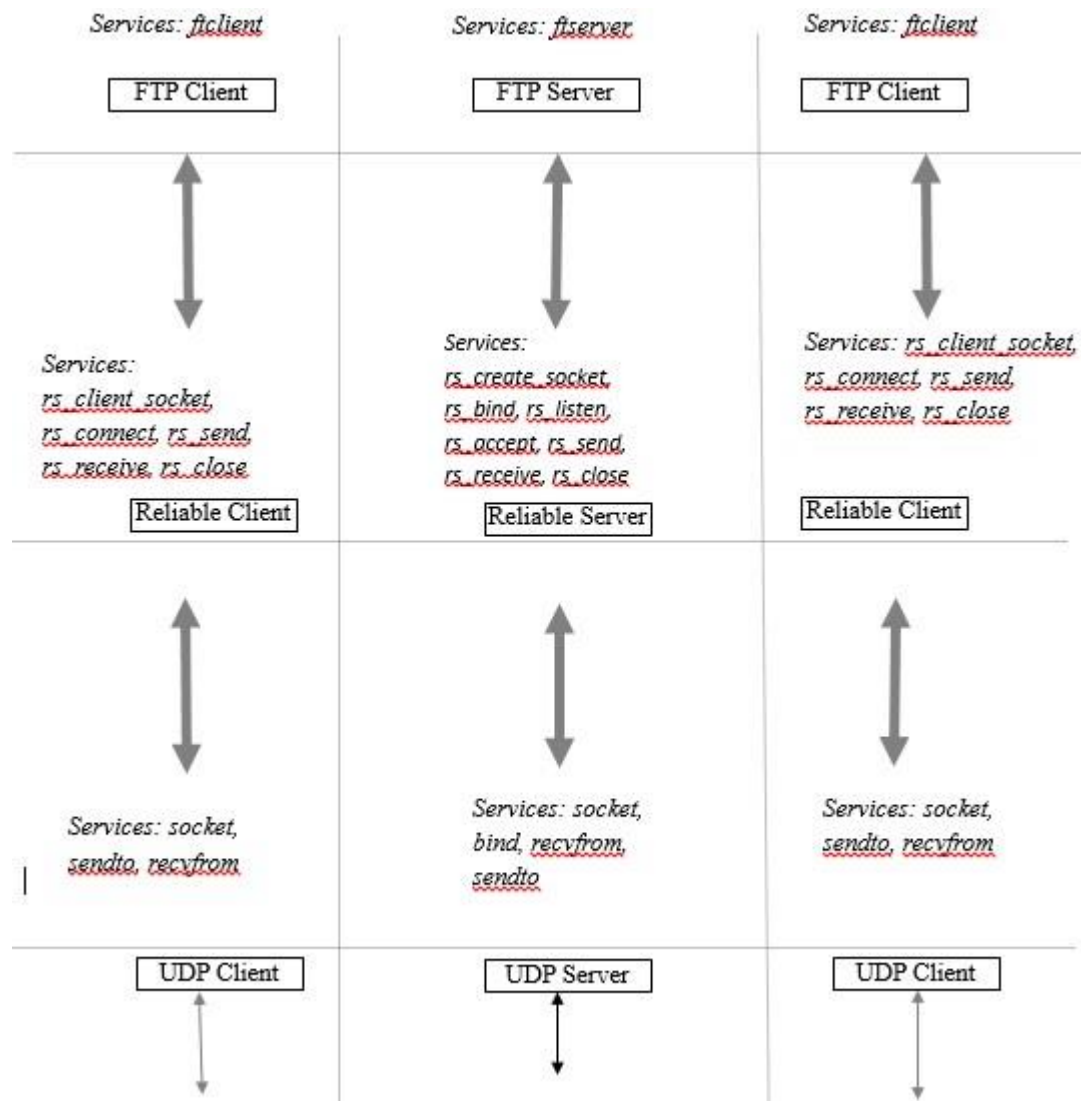
Since UDP is connectionless and unreliable we implement the following functionality:

- Reliability  
To ensure that the packets are delivered reliably with no loss or data corruption even over an unreliable network we use **Go back n** ARQ mechanism.
- Ordering of data  
To ensure that the packets are delivered in the same order as they were sent we use **CRC** error detection code. We also deal with duplicate packets responsibly.
- Connection Oriented  
To ensure reliability **connection states** are maintained at both server and client side. However data traverses only from client to server and server responds with only control packets (ACK).

Our transport protocol also deals with **concurrency**; where multiple client requests service from the server simultaneously. This is achieved by creating a child at the server side to deal with any new request.

## SRTP Protocol

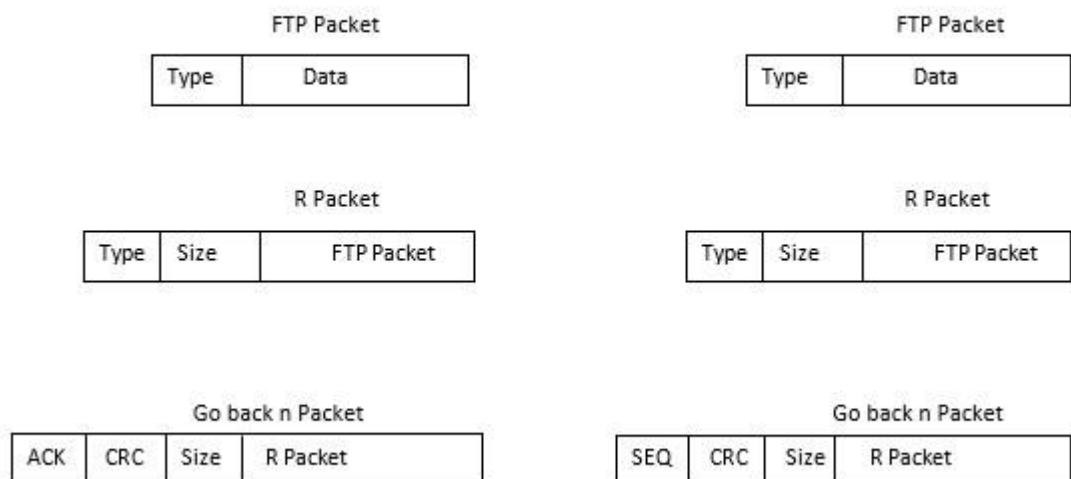
### System Design Model



**Fig 1: Three Layered Architecture for client server communication**

The protocol that we have implemented has three layers:

- FTP layer (Top most) - In this layer we accept a filename or data as input from the user, package it and pass it on to the lower layer.
- State of connection and concurrency Layer (Middle) – This layer establishes connection between client and server and manages the states of connection. This layer also implements concurrency to accept multiple requests from different clients.
- Reliable UDP layer (Lowest) – This layer ensures that the data sent from the client is delivered exactly in the same order without any loss. This is achieved by implementing **go back n** ARQ mechanism where any lost package is retransmitted. Checksum controls the ordering and redundancy of packages. The server just acknowledges the client on receiving the packet.



**Fig 2: Packet format in different layers.**

## FTP layer

The concept of this layer is provide interaction between user and server. Thus, we have two applications running, ftclient on the user side and ftserver for processing the user needs.

### Ftclient

This application receives file name as input along with address of the server and port number the user wants to communicate with. Ftclient parses the input parameters and forms ftp package which consist of the type of data and the actual data itself. Then the package is passed down to the lower layer (reliable and arq layer).

API: *ftclient [-d] <server> <port> <filename>*

### Ftserver

It is the other side of interaction, where data is received. Once the filename or file data is received it sends acknowledgment. We can either choose to read the file or discard it.

API: *ftserver [-d] <port>*

## Reliable Layer

The next layer is Reliable layer. It includes the functionality to establish and maintain connection. Concurrency is also implemented as a part of this layer.

On the **client side** file name or data along with server and port information is received as parameters from the FTP layer and then a connection is established. The client then reads the file data, sends the data to the server, and closes the connection when data transfer is complete.

This layer encapsulates the type of data, size of the packet and FTP packet to form an R packet and pass it down to the GO back n layer.

### Functions implemented on the client side are:

**Rp\_connect()** : The Rp\_connect() function is used basically to ensure that the client is connected to the server. This function is different from TCP's "connect ()" in few ways. First of all TCP has a constant connection between client and server, while R\_connect() is like a fake TCP connect() where we mimic TCP's behaviour. We use two way handshake principle: client requests service from the server and the server responds with available port number. This functionality will further be valuable to implement concurrency.

**Rp\_send()** : Once the server name and port number has been received this function send the data to the server

**Rp\_close()** : After all the data transfer and acknowledgement has happened, this function closes the connection between the client and server.

On the **server side** we take the allocated port number, creates a socket, binds the server address and port number to the socket. Now the server actively listens to any incoming requests at this port. Each time there is a new request from the client, the server responds the client with the available port where it can connect to.

***Rp\_initialise()*** : Initialises the data structures on the server

***Rp\_listen()*** : Opens the port and starts listening to the incoming requests

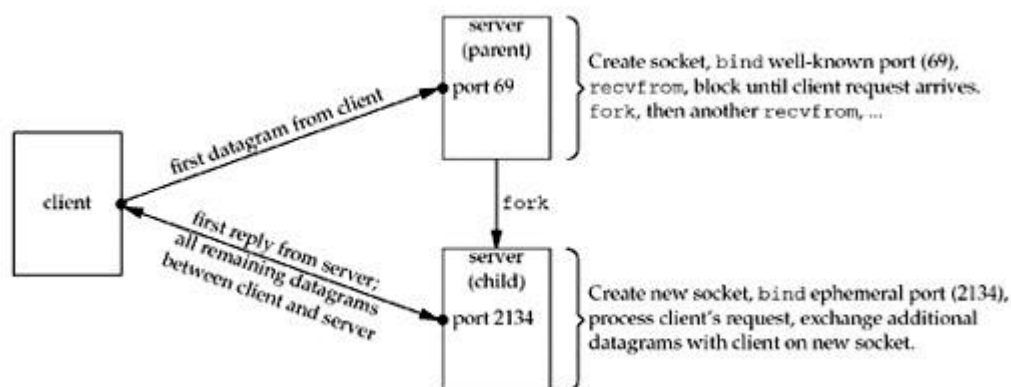
***Rp\_accept()*** : Accepts the connections

***Rp\_receive()*** : Receives the data sent

***Rp\_close()*** : Closes the connection on completion of the transfer

## Concurrency

Our protocol accepts multiple requests from clients simultaneously and respond to the client with ACK. Whenever there is a new request from the client, the server creates a child to handle request from the client with available port number. The child then binds to the new port and responds to the client



**Fig 3: Process involved in stand-alone concurrent UDP server [1]**

## Go Back n Layer

This layer ensures the data reaches the destination without any loss and in the same order sent. We use go back n ARQ to retransmit the lost packets and CRC to check the packet ordering. Go back n protocol has been implement using ***circular queue***.



On the **client side**, we take the packet received from the upper layer and place it in the **circular queue**. Sender's transmission window is set to 5, i.e., Five SDU's are transmitted at once across the network to the server. The packet sent from the client will have sequence number, CRC, size of the packet and R packet encapsulated within it. Once the server responds with ACK for each SDU, SDU is removed from the queue and window is moved ahead to accommodate the next packet to be sent. If the server does not respond within a specified amount of time, say 2 sec on our case the SDU's are resent. The packets are retransmitted for a maximum of three times before terminating the connection due to irresponsiveness from the server. Once the server responds with ACK for the packets sent, the client signals the server to terminate the connection before terminating itself.

On **server side**, the server receives the packet from the client and Checks for the correctness of the data and adds an ACK to the packet and sends it back to the client upon receiving the correct packet. If not ACK is not sent to the client and the server waits for the SDU to be retransmitted. The receiver window is 1, which means the server can receive only one packet at a time.

The following APIs are offered by the Go-BackN-Layer on client side to the above layers:

- go\_connect -> Establish the initial communication with the server
- go\_sender -> Send the packets reliably
- go\_receive -> Pass the received control data on the network to the above layer.

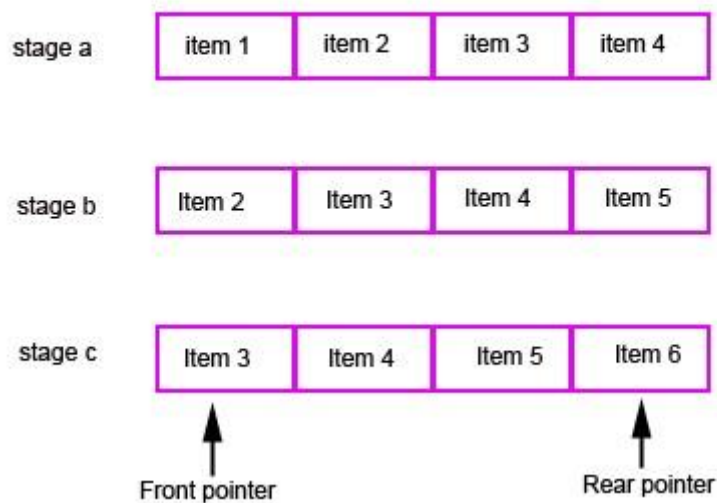
The following APIs are offered by the Go-BackN-layer on server side to the above layers:

- init\_go\_backn\_receiver -> Initialize the go backn receiver and start listening to the defined port.
- receive\_go\_backn\_receiver -> For every incoming packet, a relevant call back connection is made and the packet is handled by the above layer registered function.

### Circular Queue:

Sliding window in our protocol is implemented using Circular Queue. In Circular Queue items are inserted at the rear end of the queue and are read off from the front. As a result there is a constant stream of data flowing in and out of the queue, therefore using memory efficiently. There is a variable which keeps track of the length of the queue. We have a function called "prepare\_transmit()" which takes all the data to be transferred over the network, traverse through the queue and sends the packets beginning from the left till the end of Queue. Meanwhile on server side we have "go\_receive" which receives the packets sent and responds the client with ACK. The client waits until the server receives the packet and responds. "move\_window()" function slides the window ahead upon receiving the ACK for the packets by deleting the packets which are acknowledged from the queue.

### A circular queue (circular buffer)



**Fig 4: Circular Queue Buffer [2]**

## Handling Duplicate Packets and Acknowledgements

The server has buffer size of 1. It only accepts the packet that it is expecting. Rest of the packets will be simple discarded by sending acknowledgments.

### Flow control

The concept of flow control is to avoid flooding, when no-acknowledgement messages would overrun processes over and over again. To resolve the issue something is need to be done to increase round trip time (RTT). Thus, RTT is an important metric of if the connection is being flooded. To measure and manage the metric we could have used the basic technique [3]:

- In addition to sequence number of the packet, we also fix the time when the packet was sent.
- After an acknowledgement has been received we define the difference in local time between the time we receive the ACK and the time we have sent the packet. That difference is RTT time for the packet.
- Due to variability of frequency with which packets arrive, we should do some adjustment to RTT. We will move a percentage of the distance between the current RTT and the packet RTT. A common practice show 10% seems to work well. This is called an exponentially smoothed moving average. The effect achieved with it is noise in the RTT is smoothed out with a low pass filter.
- Finally we will discard packets that have exceeded a maximum preset RTT. Hence, the sent queue doesn't expand eternally. We can consider a packet not acked within a second has been lost, and one second is the value of our maximum RTT.

- With RTT defined we can manage our flow control. A large RTT indicates we should send data less frequently. When it is within acceptable ranges, we can try increasing the frequency of sending data.

## Acceptance Testing

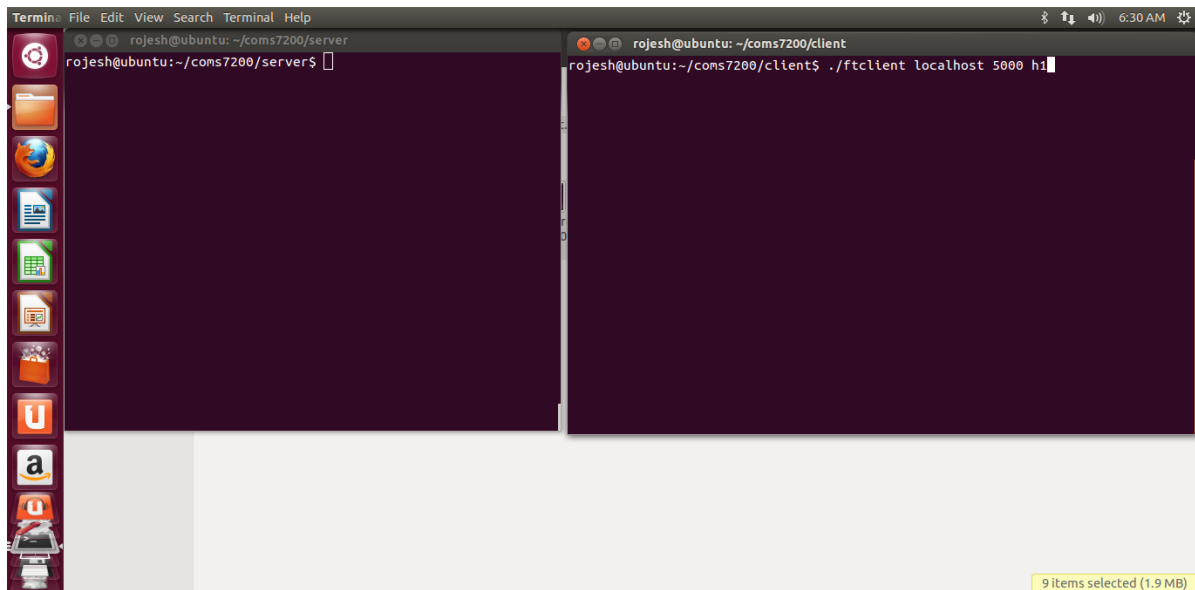
### Instruction for Use

- The system is built is using C- programming in Linux platform.
- Source code of client and server are run separately
- Server should be running first before the client can actually communicate with it
- Multiple clients can be run to ensure concurrency of the program.

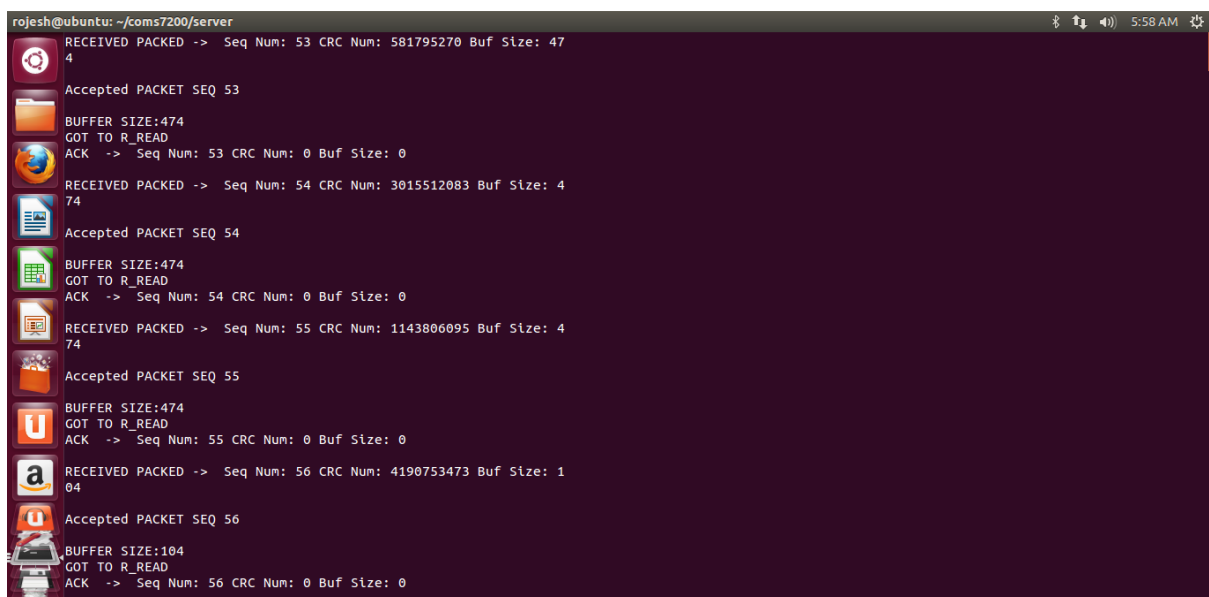
### Test Results

#### Test Case I: General Flow of the program and SRTP implementation

Test Id	Summary	Expected Results	Comments
T001	Running the server and then client and sending the file over to the server.	<p>When the server code is run, it stays idle and waits for the client to initiate to establish a connection</p> <p>On executing the client side, it tries to establish the connection with the server first.</p> <p>On successful connection, file is transferred to the server which in turn send an acknowledgement.</p>	The codes run normally and provides the output as expected.



**Fig 5: Screenshot showing server and client running in separately. Server is in idle state and client trying to communicate with server using port no. 5000**



**Fig 6: Screen shot of server receiving file from client.**

```

Terminal
rojesh@ubuntu: ~/coms7200/server
Accepted PACKET
ACK -> Seq Num: 97 CRC Num: 1902772870 Buf Size: 474
RECEIVED PACKED -> Seq Num: 98 CRC Num: 3329149040 Buf Size: 474
Accepted PACKET
ACK -> Seq Num: 98 CRC Num: 99413806 Buf Size: 474
RECEIVED PACKED -> Seq Num: 99 CRC Num: 2896155083 Buf Size: 474
Accepted PACKET
ACK -> Seq Num: 99 CRC Num: 1339536614 Buf Size: 474
RECEIVED PACKED -> Seq Num: 100 CRC Num: 3641711637 Buf Size: 474
Accepted PACKET
ACK -> Seq Num: 100 CRC Num: 1034412544 Buf Size: 474
RECEIVED PACKED -> Seq Num: 101 CRC Num: 39748476 Buf Size: 474
Accepted PACKET
ACK -> Seq Num: 101 CRC Num: 2821486432 Buf Size: 474
RECEIVED PACKED -> Seq Num: 102 CRC Num: 579109368 Buf Size: 474
Accepted PACKET
ACK -> Seq Num: 102 CRC Num: 680395719 Buf Size: 474

rojesh@ubuntu: ~/coms7200/client
WAIT FOR TIMEOUT OR ACK: RETRY 0
Sender: BEFORE LEFT MOVE
Left Ptr: 3 Right Ptr: 1 Buffer Count: 3
Inx_number:0 1 2 3 4
Seq_number:102 98 99 100 101
Ack_number:102 98 99 100 101
Seg Number:0 0 0 0 0
Seg Size :0 0 0 0 0
Sender: AFTER LEFT MOVE
Left Ptr: 1 Right Ptr: 1 Buffer Count: 0
Inx_number:0 1 2 3 4
Seq_number:102 98 99 100 101
Ack_number:102 98 99 100 101
Seg Number:0 0 0 0 0
Seg Size :0 0 0 0 0
FILE TRANSFER COMPLETED
rojesh@ubuntu:~/coms7200/client$
rojesh@ubuntu:~$

```

**Fig 7: Screenshot showing the successful transfer of file from the client to serve**

Test case II: Re-transmission of the same file.

Test ID		Summary	Excepted Results	Comments
T002		Trying to resend the same file h1 to the server.	Since the file has already been sent, the server should send an acknowledgement to the client and display a message accordingly.	File transfer was not initiated. Transmission error message display.

```

Terminal
rojesh@ubuntu: ~/coms7200/server
rojesh@ubuntu:~/coms7200/server$ ./ftserver 5000
RECEIVED PACKED -> Seq Num: 1 CRC Num: 0 Buf Size: 474
Accepted PACKET SEQ 1
FTP TYPE: 1
FTP SIZE: 256
R TYPE: 4
R SIZE: 466

rojesh@ubuntu:~/coms7200/client
rojesh@ubuntu:~$ cd coms7200 ./client
rojesh@ubuntu:~/coms7200$ cd client
rojesh@ubuntu:~/coms7200/client$ ^C
rojesh@ubuntu:~/coms7200/client$ ./ftclient localhost 5000
ftclient [-d] <server> <port> <filename>
rojesh@ubuntu:~/coms7200/client$ ./ftclient localhost 5000 h1
FILE TRANSFER COMPLETED
rojesh@ubuntu:~/coms7200/client$ ./ftclient localhost 5000 h1
0 h1
Connection could not be established as the file may exist or server is not ready
to accept the connection
rojesh@ubuntu:~/coms7200/client$

```

**Fig 8: Screenshot showing re-transmission of the same file h1 and error message related to it.**

Test case III: Establishing the connection between the client and the server with different port number.

Test Id	Summary	Test Results	Comments
T003	Assigning different port numbers to the client and server and running them.	Client tries to communicate with the server but the connection is failed	The connection fails as expected since the ports do not match and generates an error message.

```

Terminal
rojesh@ubuntu: ~/coms7200/server
rojesh@ubuntu:~/coms7200/server$ ./ftserver 5000
RECEIVED PACKED -> Seq Num: 1 CRC Num: 0 Buf Size: 474
Accepted PACKET SEQ 1
FTP TYPE: 1
FTP SIZE: 256
R TYPE: 4
R SIZE: 466
^C
rojesh@ubuntu:~/coms7200/server$ ./ftserver 5000

rojesh@ubuntu:~/coms7200/client
rojesh@ubuntu:~$ cd coms7200 ./client
rojesh@ubuntu:~/coms7200$ cd client
rojesh@ubuntu:~/coms7200/client$ ^C
rojesh@ubuntu:~/coms7200/client$ ./ftclient localhost 5000
ftclient [-d] <server> <port> <filename>
rojesh@ubuntu:~/coms7200/client$ ./ftclient localhost 5000 h1
FILE TRANSFER COMPLETED
rojesh@ubuntu:~/coms7200/client$ ./ftclient localhost 5000 h1
0 h1
Connection could not be established as the file may exist or server is not ready
to accept the connection
rojesh@ubuntu:~/coms7200/client$ ./ftclient localhost 5001 h1
Not received or timed out
Not received or timed out
Maximum retrials, could not connected to server
rojesh@ubuntu:~/coms7200/client$

```

**Fig 9: Screenshot displaying connection failure.**

#### Test Case IV: Implementation of the concurrency of the system.

Test ID	Summary	Result	Comment
T004	Multiple clients trying to communicate the server at the same time.	When more than one client tries to establish the connection with the server at the same time and tries to initiate the file transfer, the server creates the child process for each communicating client with a different port number.	The program demonstrates the concurrency.

```

Terminal
romaan@ubuntu: ~/Desktop/coms7200/client
Left Ptr: 0      Right Ptr: 0      Buffer Count: 5
Inx_number:0 1 2 3 4
Seq_number:102 103 104 105 106
Ack_number:97 98 99 100 101
Seg Number:0 1 2 3 4
Seg Size :474 474 474 474 104

Sender: TRANSMIT -> Seq Num: 102 CRC Num: 283825302 Buf Size: 474
Sender: TRANSMIT -> Seq Num: 103 CRC Num: 1966087850 Buf Size: 474
Sender: TRANSMIT -> Seq Num: 104 CRC Num: 2288731788 Buf Size: 474
Sender: TRANSMIT -> Seq Num: 105 CRC Num: 4030117963 Buf Size: 474
Sender: TRANSMIT -> Seq Num: 106 CRC Num: 3742740091 Buf Size: 104
Receiver Reading -> Seq Num: 102 CRC Num: 0 Buf Size: 0
Receiver Reading -> Seq Num: 103 CRC Num: 0 Buf Size: 0
Receiver Reading -> Seq Num: 104 CRC Num: 0 Buf Size: 0
Receiver Reading -> Seq Num: 105 CRC Num: 0 Buf Size: 0

g Number:0 0 0 0 0
g Size :0 0 0 0 0

Sender: AFTER PREPARE_TRANSMIT
Left Ptr: 0      Right Ptr: 0      Buffer Count: 5
Inx_number:0 1 2 3 4
Seq_number:102 103 104 105 106
Ack_number:97 98 99 100 101
Seg Number:0 1 2 3 4
Seg Size :474 474 474 474 104

Sender: TRANSMIT -> Seq Num: 102 CRC Num: 283825302 Buf Size: 474
Sender: TRANSMIT -> Seq Num: 103 CRC Num: 1966087850 Buf Size: 474
Sender: TRANSMIT -> Seq Num: 104 CRC Num: 2288731788 Buf Size: 474
Sender: TRANSMIT -> Seq Num: 105 CRC Num: 4030117963 Buf Size: 474
Sender: TRANSMIT -> Seq Num: 106 CRC Num: 3742740091 Buf Size: 104
Receiver Reading -> Seq Num: 102 CRC Num: 0 Buf Size: 0
Receiver Reading -> Seq Num: 103 CRC Num: 0 Buf Size: 0
Receiver Reading -> Seq Num: 104 CRC Num: 0 Buf Size: 0
Receiver Reading -> Seq Num: 105 CRC Num: 0 Buf Size: 0

romaan@ubuntu: ~/Desktop/coms7200/server
Accepted PACKET SEQ 102
BUFFER SIZE:474
GOT TO R_READ
ACK -> Seq Num: 102 CRC Num: 0 Buf Size: 0
RECEIVED PACKED -> Seq Num: 103 CRC Num: 1966087850 Buf Size: 474
Accepted PACKET SEQ 103
BUFFER SIZE:474
GOT TO R_READ
ACK -> Seq Num: 103 CRC Num: 0 Buf Size: 0
RECEIVED PACKED -> Seq Num: 104 CRC Num: 2288731788 Buf Size: 474
Accepted PACKET SEQ 104
BUFFER SIZE:474
GOT TO R_READ
ACK -> Seq Num: 104 CRC Num: 0 Buf Size: 0
RECEIVED PACKED -> Seq Num: 105 CRC Num: 4030117963 Buf Size: 474
Accepted PACKET SEQ 105
BUFFER SIZE:474
GOT TO R_READ
ACK -> Seq Num: 105 CRC Num: 0 Buf Size: 0
RECEIVED PACKED -> Seq Num: 106 CRC Num: 3742740091 Buf Size: 104
Accepted PACKET SEQ 106
BUFFER SIZE:104
GOT TO R_READ
ACK -> Seq Num: 106 CRC Num: 0 Buf Size: 0

```

**Fig 10: Multiple clients communicating with the single server.**



```

Terminal
rojesh@ubuntu: ~/coms7200/server
ACK -> Seq Num: 2 CRC Num: 964098303 Buf Size: 474
RECEIVED PACKED -> Seq Num: 16 CRC Num: 2339091483 Buf Size: 474
Accepted PACKET
ACK -> Seq Num: 16 CRC Num: 1167887989 Buf Size: 474
RECEIVED PACKED -> Seq Num: 17 CRC Num: 3941165864 Buf Size: 474
Accepted PACKET
ACK -> Seq Num: 17 CRC Num: 2566867956 Buf Size: 474
RECEIVED PACKED -> Seq Num: 18 CRC Num: 1462104936 Buf Size: 474
Accepted PACKET
ACK -> Seq Num: 18 CRC Num: 3844517720 Buf Size: 474
RECEIVED PACKED -> Seq Num: 19 CRC Num: 1933095433 Buf Size: 474
Accepted PACKET
ACK -> Seq Num: 19 CRC Num: 320277908 Buf Size: 474
RECEIVED PACKED -> Seq Num: 20 CRC Num: 2471632752 Buf Size: 474
Accepted PACKET

rojesh@ubuntu: ~/coms7200/client
Connected to new port: 48593
Sender: BEFORE PREPARE_TRANSMIT
Left Ptr: 0 Right Ptr: 0 Buffer Count: 0
Inx_number: 0 1 2 3 4
Seq_number: 0 0 0 0 0
Ack_number: 0 0 0 0 0
Seg Number: 0 0 0 0 0
Seg Size : 0 0 0 0 0
Sender: AFTER PREPARE_TRANSMIT
Left Ptr: 0 Right Ptr: 1 Buffer Count: 1
Inx_number: 0 1 2 3 4
Seq_number: 2 0 0 0 0
Ack_number: 0 0 0 0 0
Seg Number: 0 0 0 0 0
Seg Size : 75 0 0 0 0
Sender: TRANSMIT -> Seq/Ack Num: 2 CRC Num: 3138009671 Buf Size: 75
Receiver Reading -> Seq/Ack Num: 2 CRC Num: 964098303 Buf Size: 474
rojesh@ubuntu: ~/coms7200/client
rojesh@ubuntu:~/coms7200/client$ ./ftclient localhost 5000 h1
FILE TRANSFER COMPLETED
rojesh@ubuntu:~/coms7200/client$

```

**Fig 11: Screenshot demonstrating concurrency and showing the different port assigned for the client.**

Test case V: Implementation of the concurrency, both the clients trying to send the same file

Test ID	Summary	Result	Comment
T005	Multiple clients trying to communicate the server at the same time and trying to send the same file.	When more than one client tries to establish the connection with the server at the same time and tries to initiate the file transfer for the same file, the server creates the child process for each communicating client with a different port number. But since both the clients are sending the same file it terminates the connection with one to avoid the duplication.	The program demonstrates the concurrency and avoids duplication.

```

Terminal
rojesh@ubuntu: ~/coms7200/server
rojesh@ubuntu:~/coms7200/server$ rm h1 h2
rojesh@ubuntu:~/coms7200/server$ ./ftserver -d 5000
FT_SERVER initialized
RECEIVED PACKED -> Seq Num: 1 CRC Num: 4209931641 Buf Size: 474
Accepted PACKET
Newly bound socket:45826
First ACK with port number -> Seq Num: 1 CRC Num: 0 Buf Size: 474
RECEIVED PACKED -> Seq Num: 2 CRC Num: 3138009671 Buf Size: 75
Accepted PACKET
ACK -> Seq Num: 2 CRC Num: 964098303 Buf Size: 474
RECEIVED PACKED -> Seq Num: 1 CRC Num: 4209931641 Buf Size: 474
Accepted PACKET
Newly bound socket:55831
First ACK with port number -> Seq Num: 1 CRC Num: 0 Buf Size: 474

rojesh@ubuntu:~/coms7200/client
rojesh@ubuntu:~/coms7200/client$ ./ftclient localhost 5000 h2
Connection could not be established as the file may exist or server is not ready to accept the connection
rojesh@ubuntu:~/coms7200/client$

```

**Fig 12: Screenshot displaying the error message to avoid the duplication as both the clients are transferring the same file.**

#### Test Case VI: Testing network performance using Iperf

Test ID	Summary	Result	Comments
T006	<p>Testing the network performance between the client and the server.</p> <p>Using <b>Iperf</b> as a network testing tool.</p>	There is a good link between the client and the server with minimal or no data loss.	Demonstrated the network performance.

```

Terminal
rojesh@ubuntu: ~/coms7200/server
rojesh@ubuntu:~/coms7200/server$ iperf -s -u
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size: 208 KByte (default)
[ 3] local 127.0.0.1 port 5001 connected with 127.0.0.1 port 46313
[ ID] Interval      Transfer      Bandwidth
[ 3] 0.0-10.0 sec  4.77 MBytes  4.00 Mbits/sec  0.007 ms
0/ 3401 (0%)
[ 3] 0.0-10.0 sec  1 datagrams received out-of-order

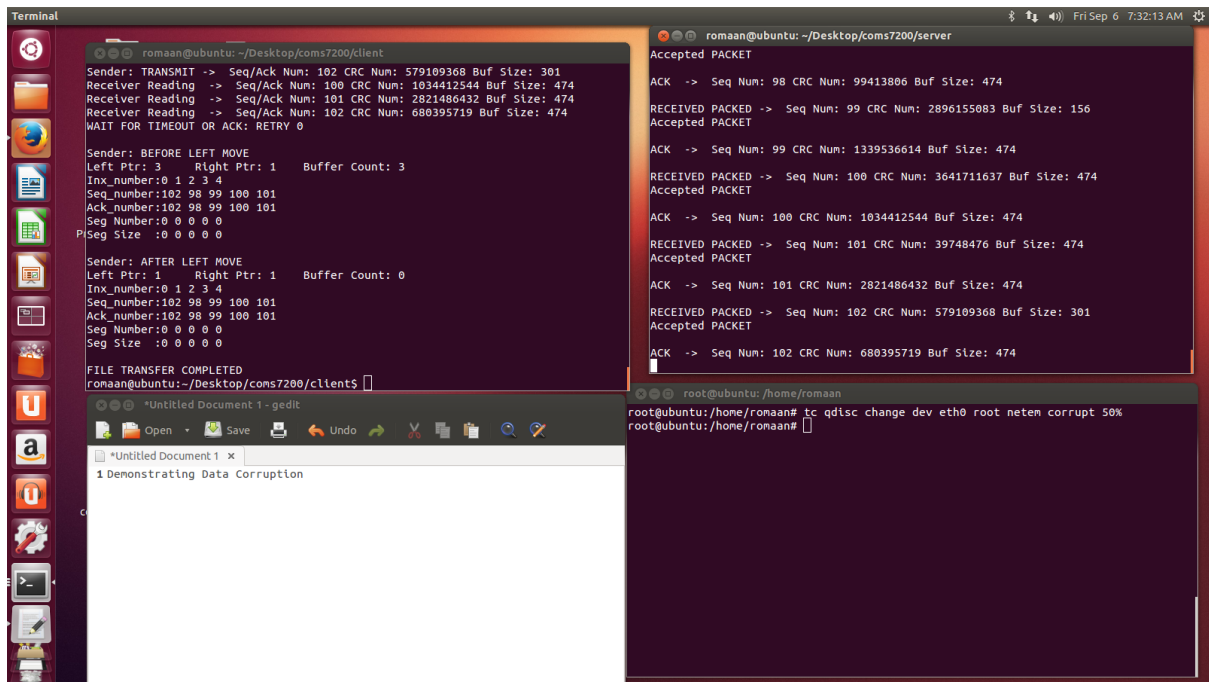
rojesh@ubuntu:~/coms7200/client
rojesh@ubuntu:~/coms7200/client$ iperf -c localhost -u -b 4m
Client connecting to localhost, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size: 208 KByte (default)
[ 3] local 127.0.0.1 port 46313 connected with 127.0.0.1 port 5001
[ ID] Interval      Transfer      Bandwidth
[ 3] 0.0-10.0 sec  4.77 MBytes  4.00 Mbits/sec
[ 3] Sent 3402 datagrams
[ 3] Server Report:
[ 3] 0.0-10.0 sec  4.77 MBytes  4.00 Mbits/sec  0.006 ms  0/ 3401 (0%)
[ 3] 0.0-10.0 sec  1 datagrams received out-of-order
rojesh@ubuntu:~/coms7200/client$

```

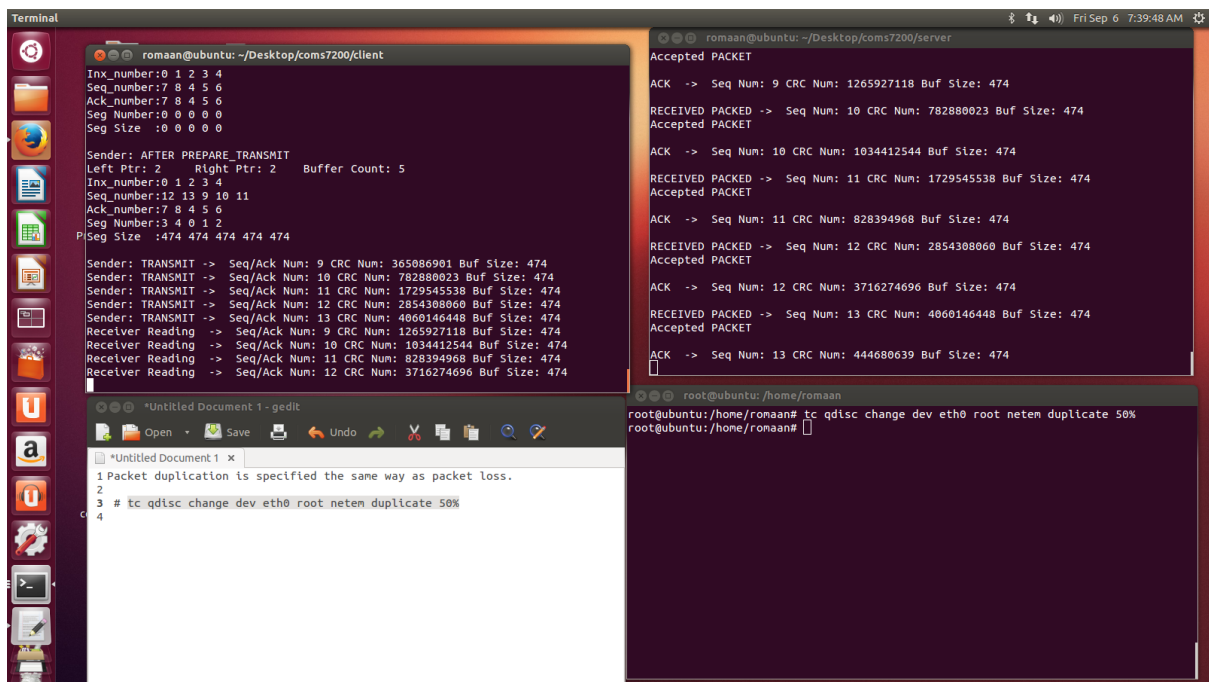
**Fig 13: Testing the network performance.**

### Test Case VII - X: Using Netem command to evaluate network reliability

Test ID	Summary	Results	Comment
T007	<p><b>Data Corruption</b></p> <p>Random noise can be emulated with the corrupt option of the netem function. This introduces a single bit error at a random offset in the packet.</p> <p>The command used to test this is as follows</p> <pre>tc qdisc change dev eth0 root netem corrupt 0.1%</pre>	<p>The expected result after running this command in the interface is the introduction of 10% data corruption in each packet</p> <p>Refer fig no. 14</p>	<p>Excepted data corruption was observed</p>
T008	<p><b>Duplication</b></p> <p>The following command line is executed to explore the duplication</p> <pre>tc qdisc change dev eth0 netem duplicate 0.5%</pre>	<p>Data duplication is expected while transferring.</p> <p>Refer fig no. 15</p>	<p>Expected data duplication was observed.</p> <p>Seq. no were repeating.</p>
T009	<p><b>Reordering</b></p> <p>Packet reordering occurs when packets traverse paths with differing delay. This can be tested with following command lines</p> <pre>tc qdisc change dev eth0 root netem gap 2 delay 10ms</pre>	<p>The network is expected send every 2<sup>nd</sup> packet immediately and delay every other packets by 10 ms.</p> <p>Refer fig no. 16</p>	<p>The program performed as expected.</p>
T0010	<p><b>Correctness</b></p> <p>Check sum command is used to detect the correctness</p> <pre>crc 32 &lt;filename&gt;</pre>	<p>The checksum computed on both server side and client side should match</p> <p>Refer fig no. 17</p>	<p>The checksum were same for both client and server.</p>



**Fig 14: Screenshot demonstrating data corruption**



**Fig 15: Screenshot displaying the demonstration of duplication**

```
Terminal
romaan@ubuntu: ~/Desktop/coms7200/client
Ack_number:12 13 9 10 11
Seq Number:0 0 0 0 0
Seg Size :0 0 0 0 0

Sender: BEFORE PREPARE_TRANSMIT
Left Ptr: 2   Right Ptr: 2   Buffer Count: 0
Inx_number:0 1 2 3 4
Seq_number:12 13 9 10 11
Ack_number:12 13 9 10 11
Seq Number:0 0 0 0 0
Seg Size :0 0 0 0 0

P Sender: AFTER PREPARE_TRANSMIT
Left Ptr: 2   Right Ptr: 4   Buffer Count: 2
Inx_number:0 1 2 3 4
Seq_number:12 13 14 15 11
Ack_number:12 13 9 10 11
Seq Number:0 0 5 6 0
Seg Size :0 0 474 156 0

Sender: TRANSMIT -> Seq/Ack Num: 14 CRC Num: 124803175 Buf Size: 474
Sender: TRANSMIT -> Seq/Ack Num: 15 CRC Num: 2477510741 Buf Size: 156
Receiver Reading -> Seq/Ack Num: 14 CRC Num: 157522135 Buf Size: 474

romaan@ubuntu: ~/Desktop/coms7200/server
Accepted PACKET
ACK -> Seq Num: 11 CRC Num: 828394968 Buf Size: 474
RECEIVED PACKED -> Seq Num: 12 CRC Num: 2854308060 Buf Size: 474
Accepted PACKET
ACK -> Seq Num: 12 CRC Num: 3716274696 Buf Size: 474
RECEIVED PACKED -> Seq Num: 13 CRC Num: 4060146448 Buf Size: 474
Accepted PACKET
ACK -> Seq Num: 13 CRC Num: 444680639 Buf Size: 474
RECEIVED PACKED -> Seq Num: 14 CRC Num: 124803175 Buf Size: 474
Accepted PACKET
ACK -> Seq Num: 14 CRC Num: 157522135 Buf Size: 474
RECEIVED PACKED -> Seq Num: 15 CRC Num: 2477510741 Buf Size: 156
Accepted PACKET
ACK -> Seq Num: 15 CRC Num: 1034412544 Buf Size: 474

root@ubuntu: /home/romaan
root@ubuntu:/home/romaan# tc qdisc change dev eth0 root netem delay 10ms reorder 25% 50%
root@ubuntu:/home/romaan#
```

**Fig 16: Screenshot demonstrating the reordering**

```
Terminal
rojesh@ubuntu: ~/coms7200/server
rojesh@ubuntu:~/coms7200/server$ clear 1
rojesh@ubuntu:~/coms7200/server$ rm h1 h2
rojesh@ubuntu:~/coms7200/server$ ./ftserver -d 5000
FT_SERVER initialized

RECEIVED PACKED -> Seq Num: 1 CRC Num: 4209931641 Buf Size: 474
Accepted PACKET
Newly bound socket:45826
First ACK with port number -> Seq Num: 1 CRC Num: 0 Buf Size: 474

RECEIVED PACKED -> Seq Num: 2 CRC Num: 3138009671 Buf Size: 75
Accepted PACKET
ACK -> Seq Num: 2 CRC Num: 964098303 Buf Size: 474

RECEIVED PACKED -> Seq Num: 1 CRC Num: 4209931641 Buf Size: 474
Accepted PACKET
Newly bound socket:55831
First ACK with port number -> Seq Num: 1 CRC Num: 0 Buf Size: 474
^C
rojesh@ubuntu:~/coms7200/server$ crc32 2
/usr/bin/crc32: No such file or directory
rojesh@ubuntu:~/coms7200/server$ crc32 h2
bf3b2348
rojesh@ubuntu:~/coms7200/server$

rojesh@ubuntu: ~/coms7200/client
Sender: AFTER PREPARE_TRANSMIT
Left Ptr: 0   Right Ptr: 1   Buffer Count: 1
Inx_number:0 1 2 3 4
Seq_number:2 0 0 0 0
Ack_number:0 0 0 0 0
Seq Number:0 0 0 0 0
Seg Size :75 0 0 0 0

Sender: TRANSMIT -> Seq/Ack Num: 2 CRC Num: 3138009671 Buf Size: 75
Receiver Reading -> Seq/Ack Num: 2 CRC Num: 964098303 Buf Size: 474
WAIT FOR TIMEOUT OR ACK: RETRY 0

Sender: BEFORE LEFT MOVE
Left Ptr: 0   Right Ptr: 1   Buffer Count: 1
Inx_number:0 1 2 3 4
Seq_number:2 0 0 0 0
Ack_number:2 0 0 0 0
Seq Number:0 0 0 0 0
Seg Size :0 0 0 0 0

Sender: AFTER LEFT MOVE
Left Ptr: 1   Right Ptr: 1   Buffer Count: 0
Inx_number:0 1 2 3 4
Seq_number:2 0 0 0 0
Ack_number:2 0 0 0 0
Seq Number:0 0 0 0 0
Seg Size :0 0 0 0 0

FILE TRANSFER COMPLETED
rojesh@ubuntu:~/coms7200/client$ crc32 h2
bf3b2348
rojesh@ubuntu:~/coms7200/client$ ^C
rojesh@ubuntu:~/coms7200/client$ ^C
rojesh@ubuntu:~/coms7200/client$
```

**Fig 17: Screenshot demonstrating the correctness of the program.**

## References

- [1] W. R. Stevens, UNIX Network Programming Networking APIs: Sockets and XTI, 2<sup>nd</sup> ed. Prentice hall Inc. 1998.
- [2] Tech\_ICT, (Undated), "Circular Queue", [Online] Viewed on : 4th sept 2013  
Available on: [Reference: [http://www.teach-ict.com/as\\_as\\_computing/ocr/H447/F453/3\\_3\\_5/data\\_structures/miniweb/pg13.htm](http://www.teach-ict.com/as_as_computing/ocr/H447/F453/3_3_5/data_structures/miniweb/pg13.htm)]
- [3] G.Fiedler, (Oct, 2008), "Reliability and Flow Control", [Online], viewed on 4<sup>th</sup> sep 2013  
Available on: <http://gafferongames.com/networking-for-game-programmers/reliability-and-flow-control/>