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

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 user ***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

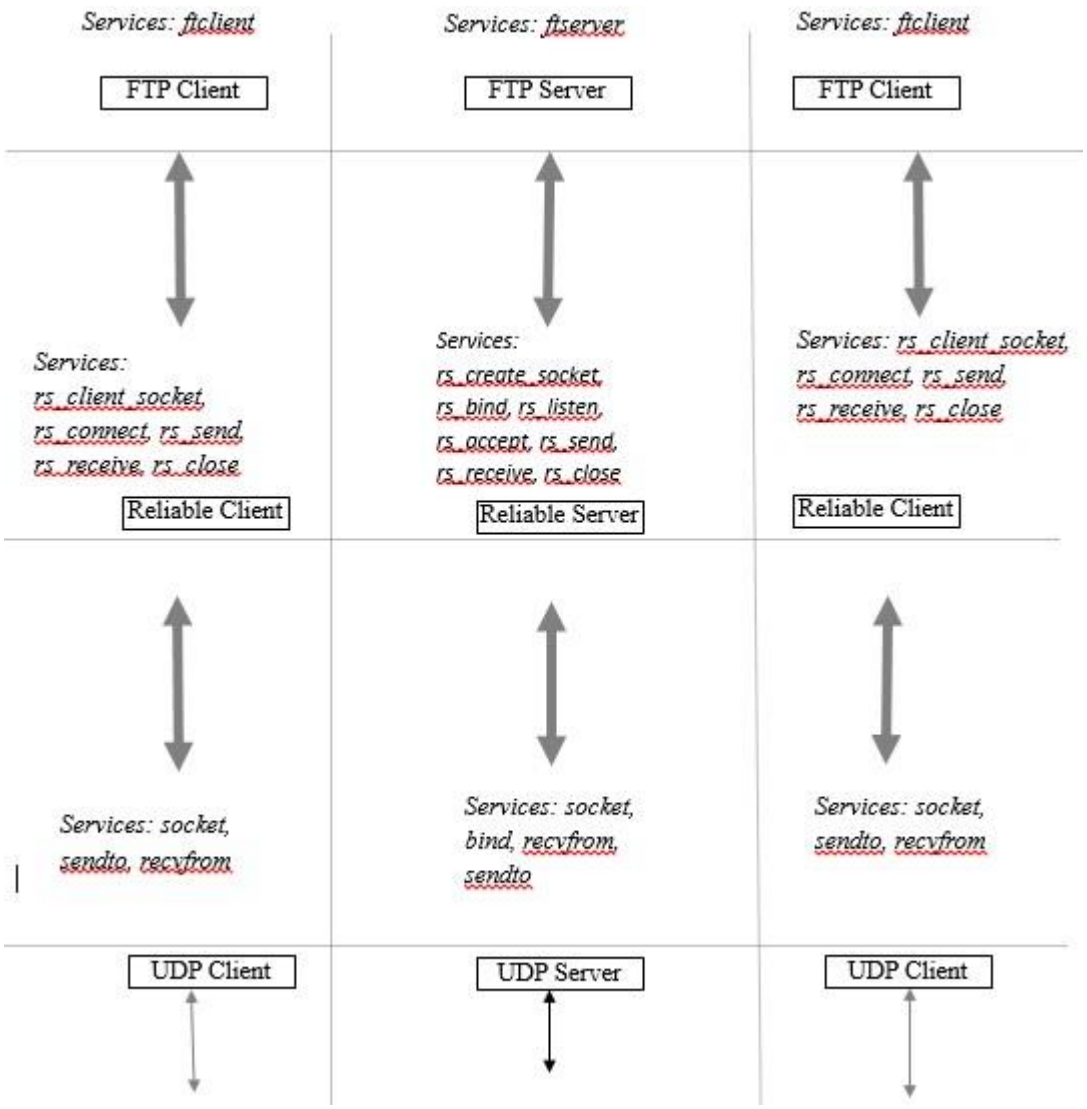

Fig 1: Three Layered Architecture for client server communication

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.

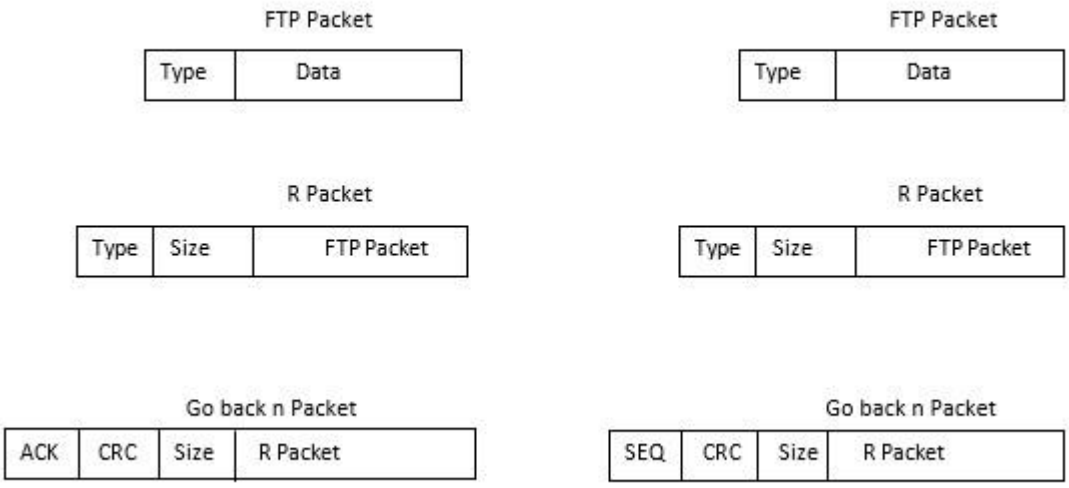

Fig 2: Packet format in different layers.

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>*

Ftsever

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

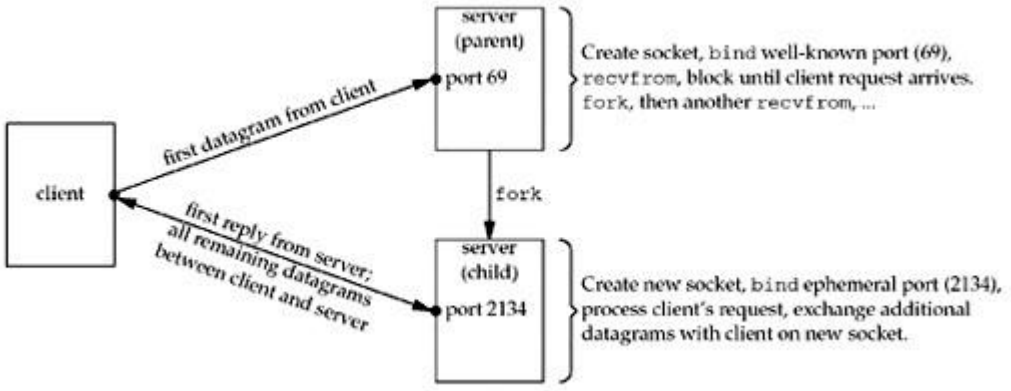

Fig 3: Process involved in stand-alone concurrent UDP server [1]

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.

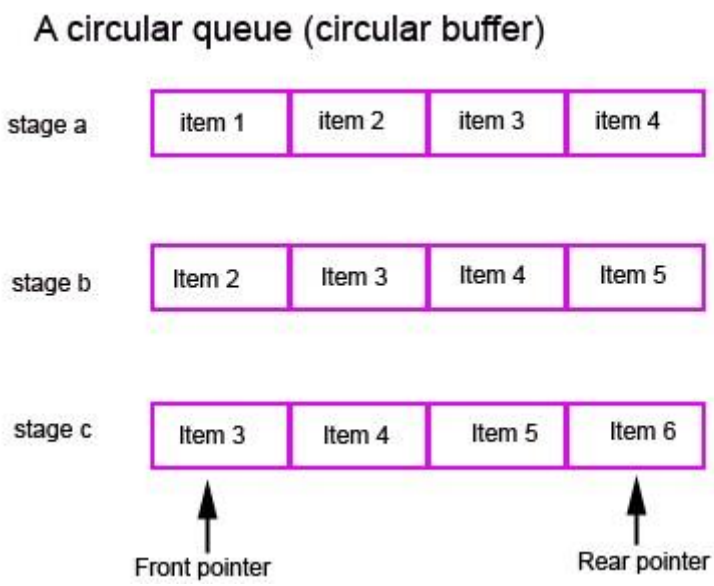

Fig 4: Circular Queue Buffer [2]

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.	When the server code is run, it stays idle and waits for the client to initiate to establish a connection On executing the client side, it tries to establish the connection with the server first. On successful connection, file is transferred to the server which in turn send an acknowledgement.	The codes run normally and provides the output as expected.

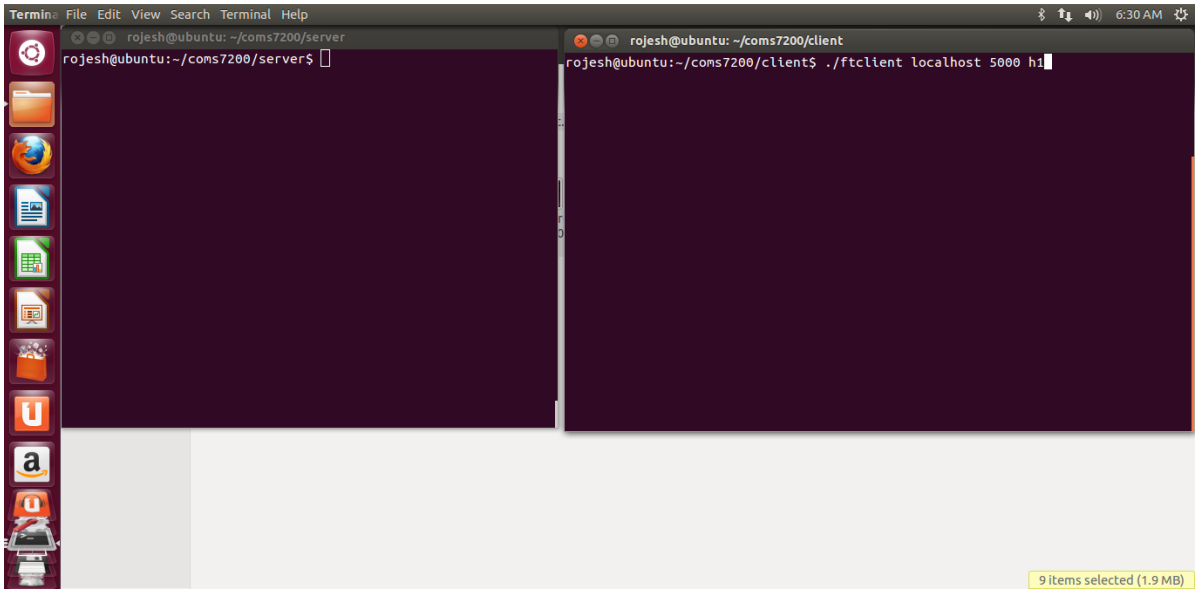

Fig 5: Screenshot showing server and client running in separately. Server is in idle state and client trying to communicate with server using port no. 5000

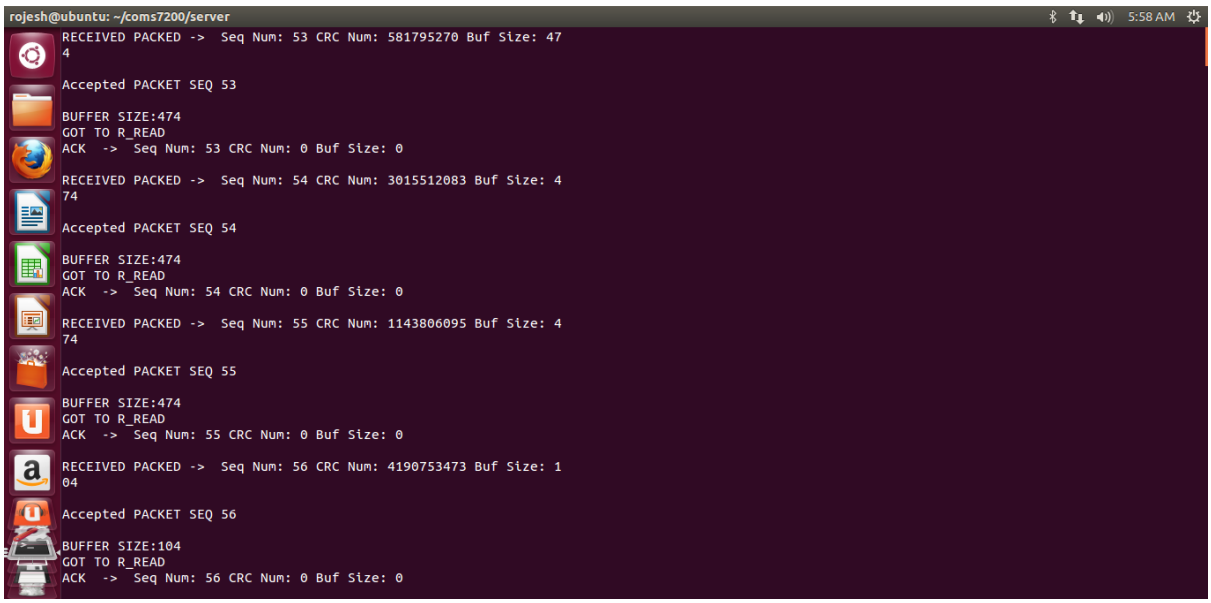

Fig 6: Screen shot of server receiving file from client.

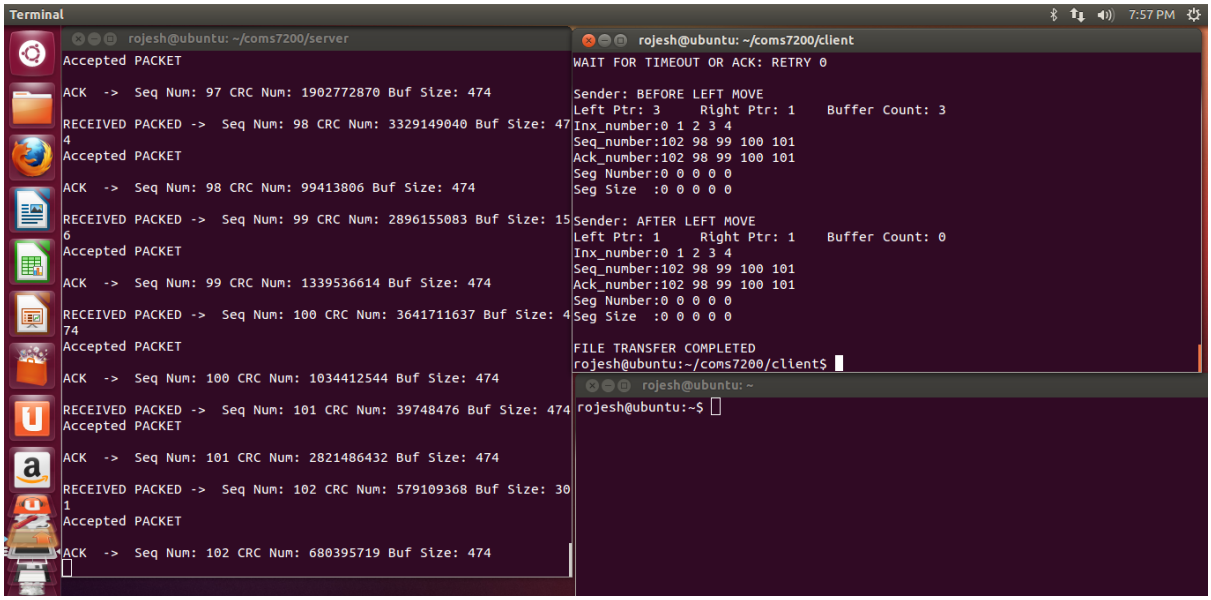

Fig 7: Screenshot showing the successful transfer of file from the client to serve

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.

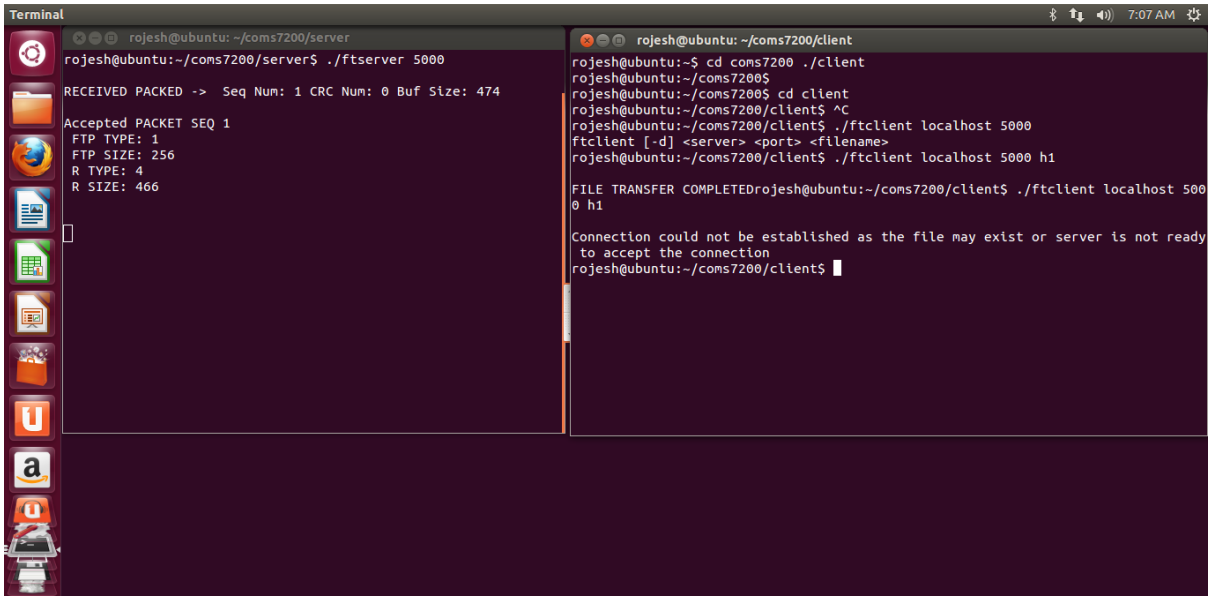

Fig 8: Screenshot showing re-transmission of the same file h1 and error message related to it.

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.

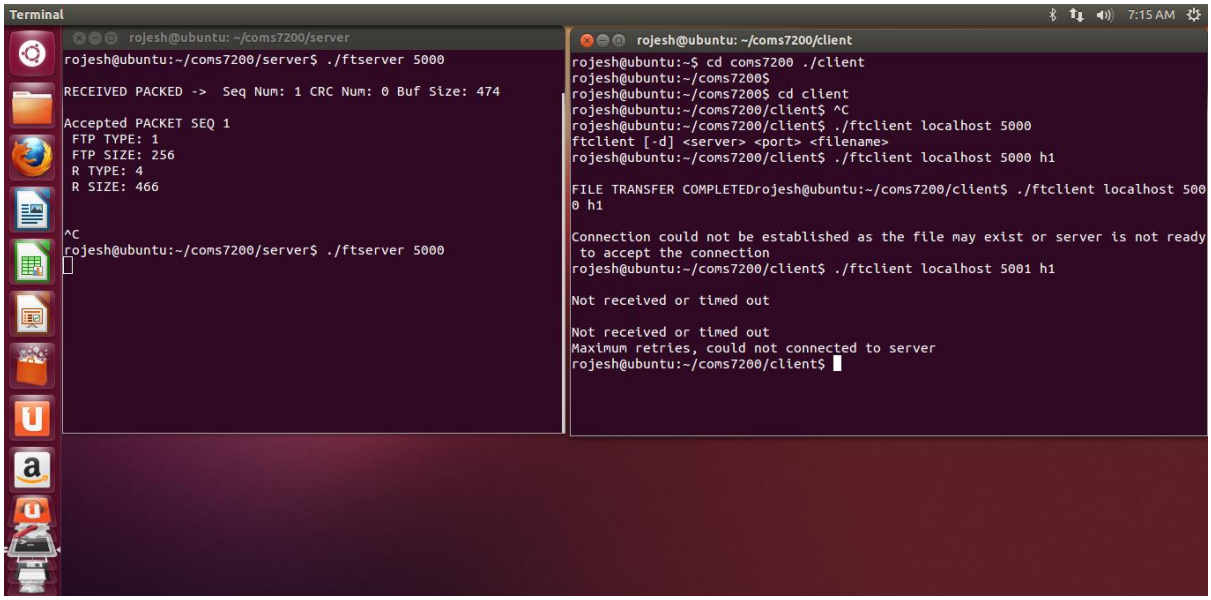

Fig 9: Screenshot displaying connection failure.

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.

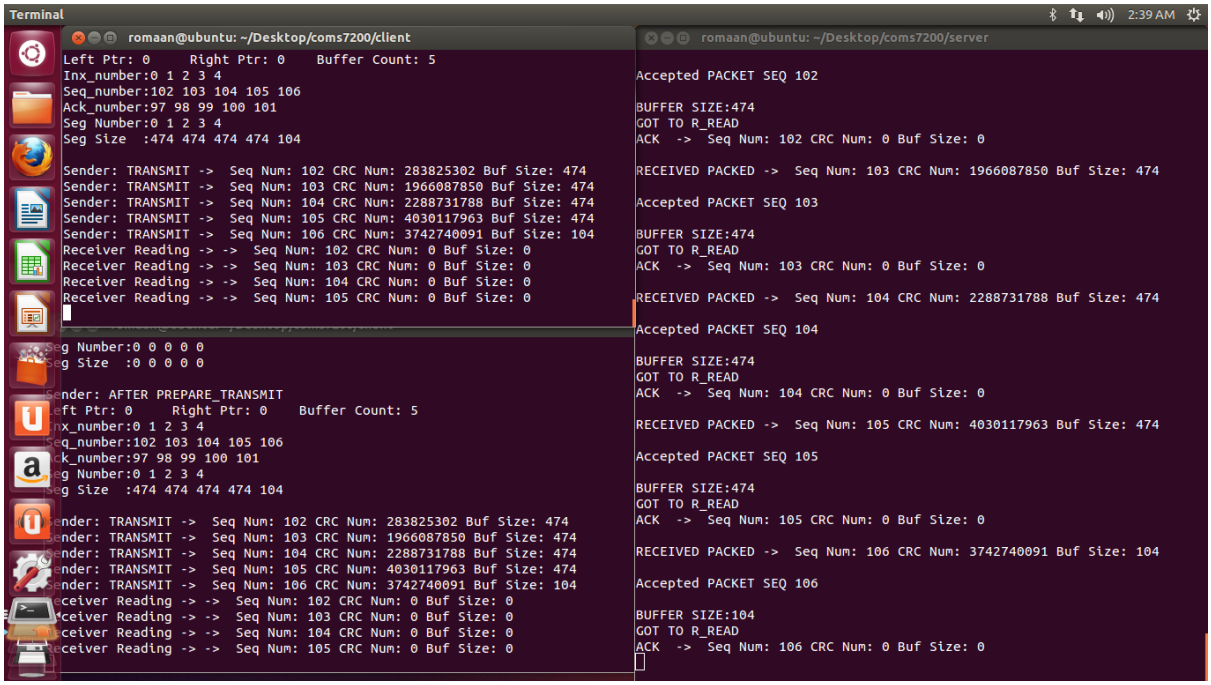

Fig 10: Multiple clients communicating with the single server.

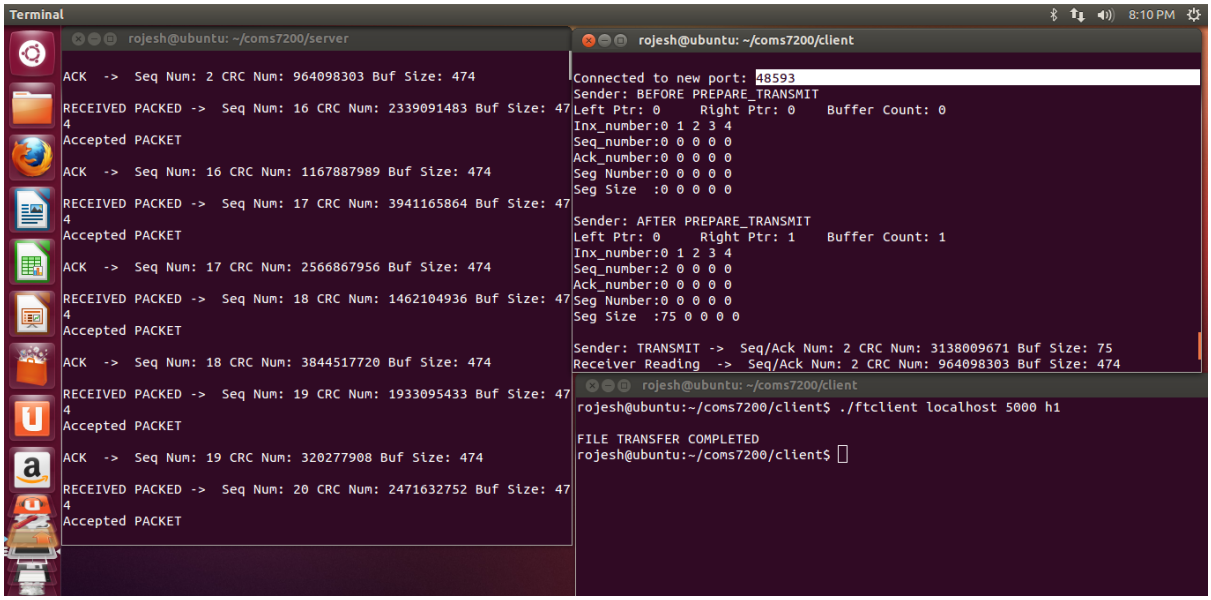

Fig 11: Screenshot demonstrating concurrency and showing the different port assigned for the client.

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.

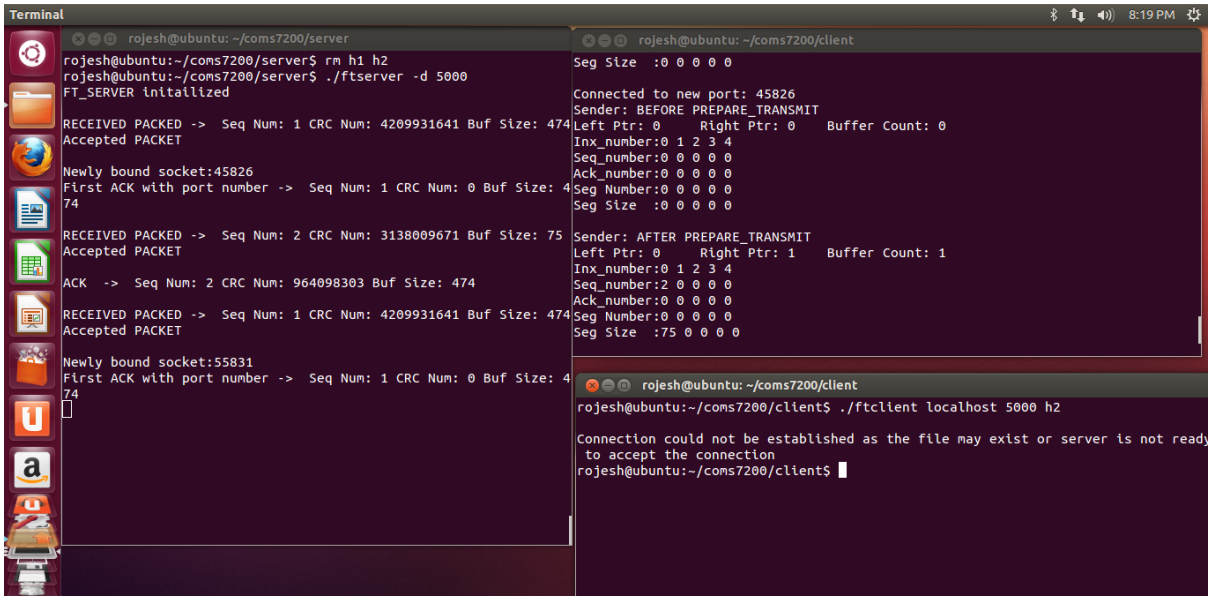

Fig 12: Screenshot displaying the error message to avoid the duplication as both the clients are transferring the same file.

Test Case VI: Testing network performance using Iperf

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

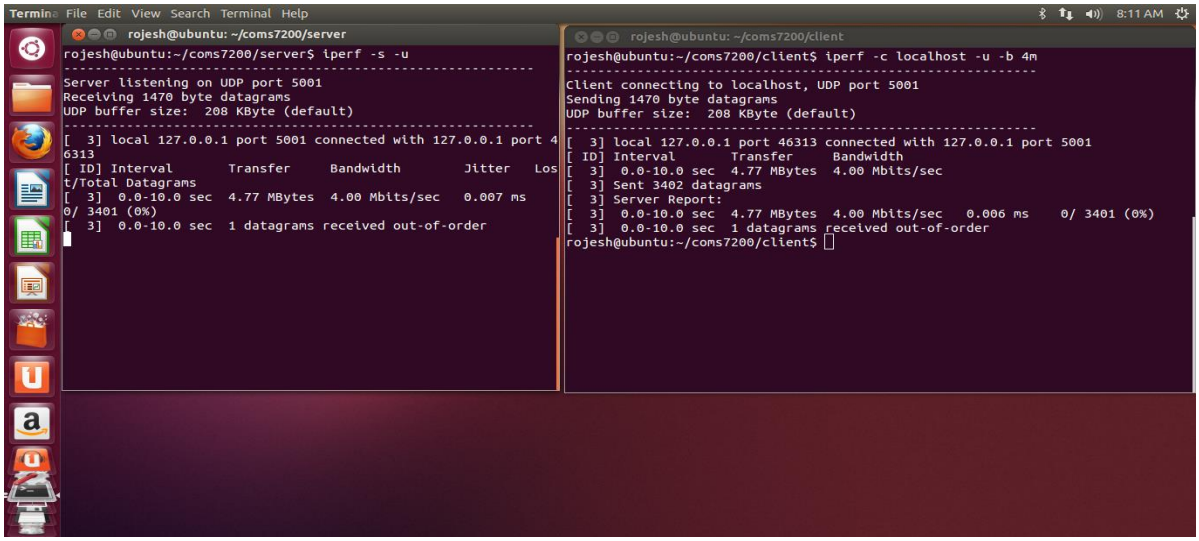

Fig 13: Testing the network performance.

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

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

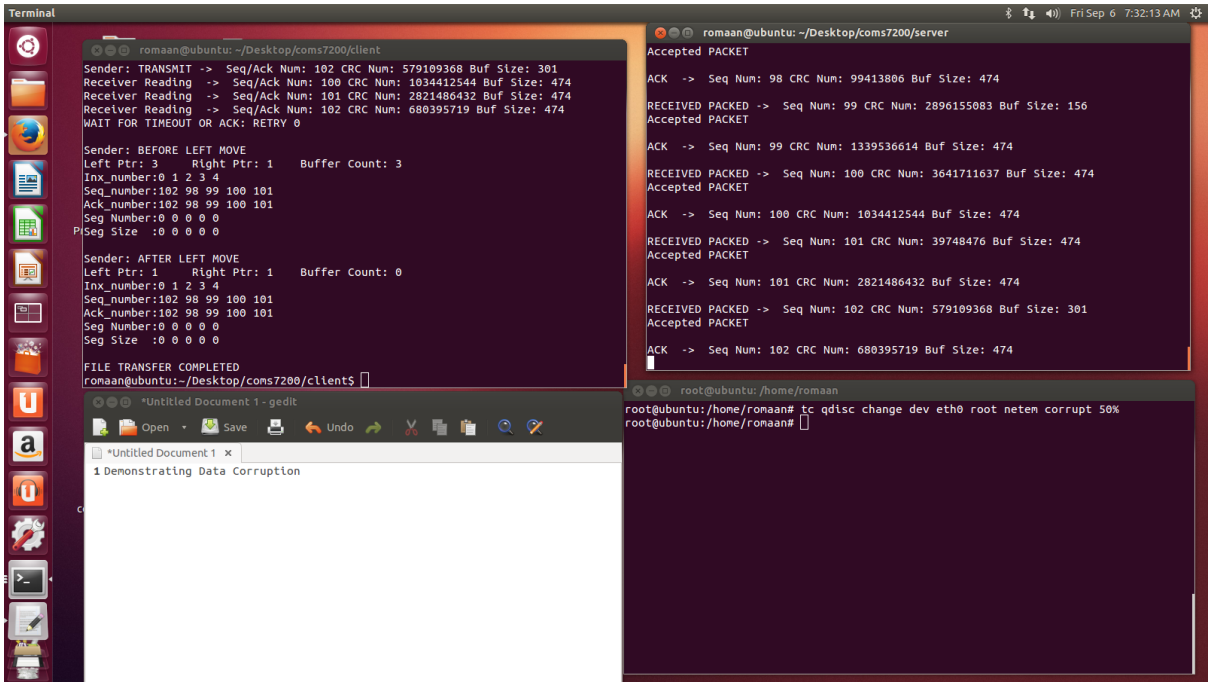

Fig 14: Screenshot demonstrating data corruption

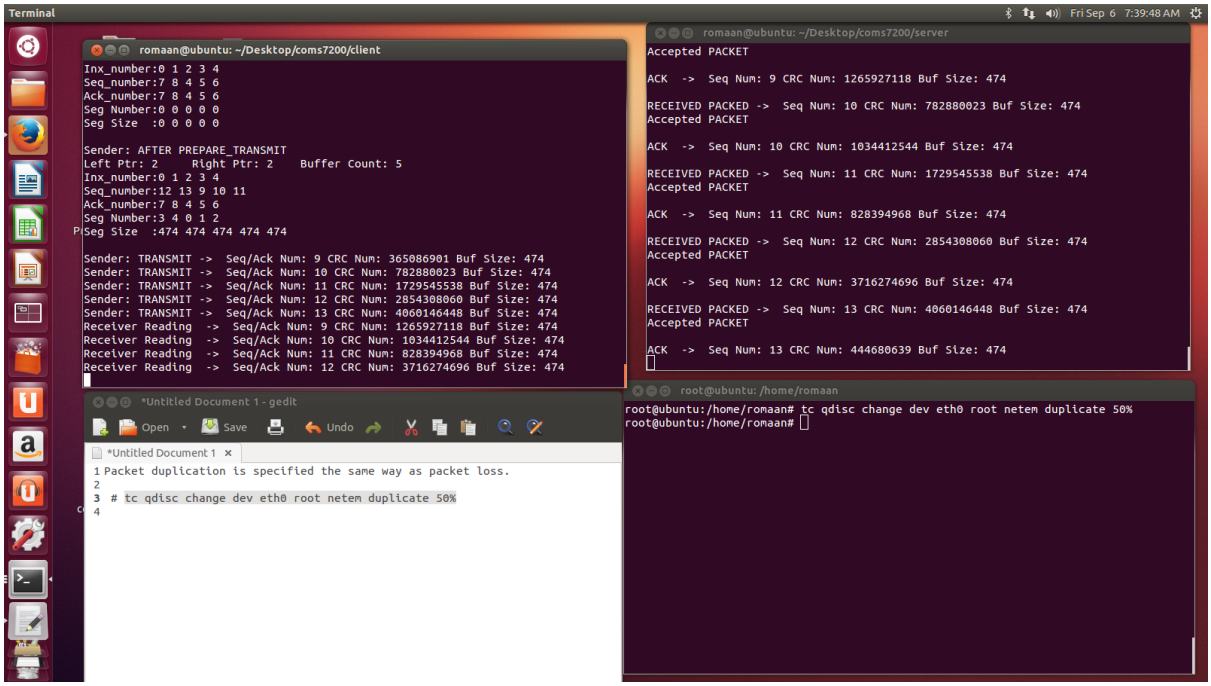

Fig 15: Screenshot displaying the demonstration of duplication

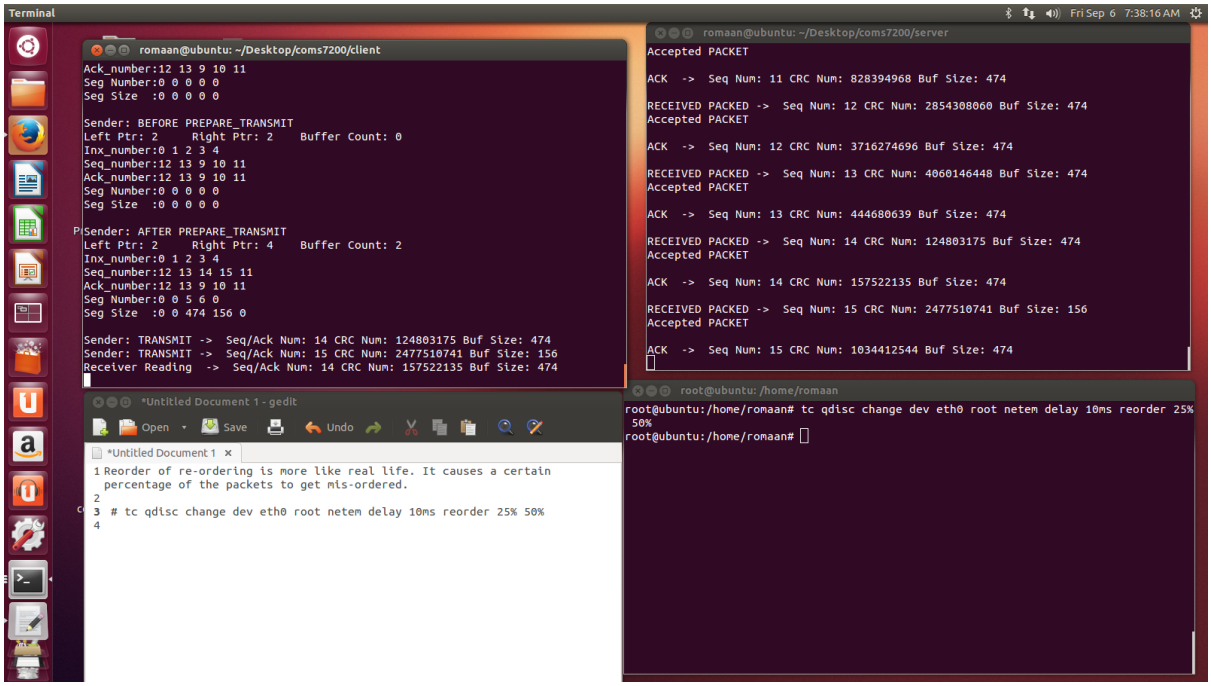

Fig 16: Screenshot demonstrating the reordering

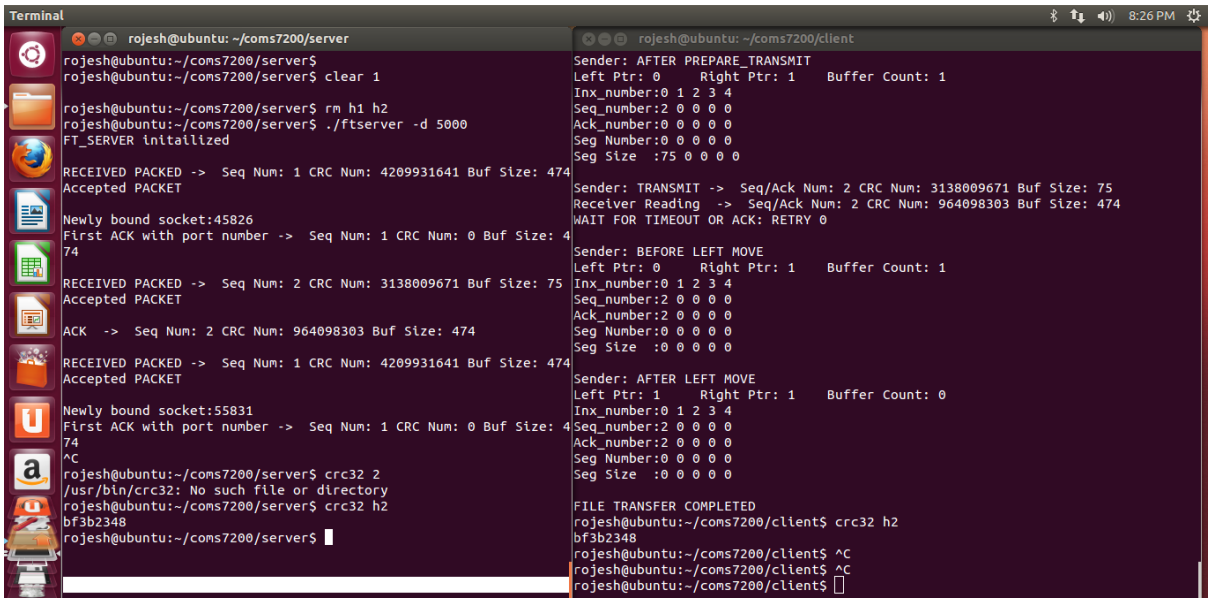

Fig 17: Screenshot demonstrating the correctness of the program.

References

[1] W. R. Stevens, UNIX Network Programming Networking APIs: Sockets and XTI, 2nd 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

[3] G.Fiedler, (Oct, 2008), “Reliability and Flow Control”,[Online], viewed on 4th sep 2013

Available on: http://gafferongames.com/networking-for-game-programmers/reliability-and-flow-control/