**Project Report**

On

**Reliable Communication over Unreliable Channel**

*Submitted by*

**Bhumit Patel L20382154**



**Dr. Lawrence Osborne**

**Department of Computer Science**

**Lamar University**

**Beaumont, TX**

**Table of Contents**

[1. Introduction 1](#_Toc436255201)

[i. PURPOSE OF THE PROJECT 1](#_Toc436255202)

[ii. PRINCIPLE OF COMPUTER SCIENCE IS BEING STUDIED: 1](#_Toc436255203)

[iii. HOW THE PROJECT IS DESIGNED 1](#_Toc436255204)

[iv. CONTENT OF OTHER SECTIONS 1](#_Toc436255205)

[2. Background 2](#_Toc436255206)

[Transport Layer: 2](#_Toc436255207)

[a. Primary Responsibilities of Transport layer Protocols: 2](#_Toc436255208)

[b. Transport Layer Reliability: 2](#_Toc436255209)

[c. Transmission Control Protocol (TCP) 2](#_Toc436255210)

[d. User Data-gram Protocol (UDP) 3](#_Toc436255211)

[e. TCP or UDP 3](#_Toc436255212)

[3. Software design: 3](#_Toc436255213)

[i. Requirement Analysis 3](#_Toc436255214)

[ii. Design 6](#_Toc436255215)

[iii. Challenges and Problems 9](#_Toc436255216)

[a. Thread 9](#_Toc436255217)

[b. Mutual Exclusion 10](#_Toc436255218)

[c. Data Structure Creation and Pointers 10](#_Toc436255219)

[4. Implementation 11](#_Toc436255220)

[i. rsocket.h 11](#_Toc436255221)

[ii. rsocket.c 11](#_Toc436255222)

[5. Tests 13](#_Toc436255223)

[6. Results: 16](#_Toc436255224)

[7. Conclusion 18](#_Toc436255225)

**Table of Figures**

[Figure 1: r\_Socket Design 6](#_Toc436255226)

[Figure 2 : r\_bind design 7](#_Toc436255227)

[Figure 3 : r\_sendTo design 7](#_Toc436255228)

[Figure 4 : r\_recvFrom Design 8](#_Toc436255229)

[Figure 5 : r\_close design 8](#_Toc436255230)

[Figure 6 : Workflow between sender and receiver 12](#_Toc436255231)

[Figure 7: Result graph with different probabilities 15](#_Toc436255232)

[Figure 8 : Complete view of user1 and user2 16](#_Toc436255233)

[Figure 9 : User1 Terminal 16](#_Toc436255234)

[Figure 10 : User2 Terminal 17](#_Toc436255235)

**Table of Tables**

[Table 1 : Requirements 5](#_Toc436255236)

[Table 2 : Threads Table 9](#_Toc436255237)

[Table 3 : Result tables with average retransmission 14](#_Toc436255238)

# Introduction

## PURPOSE OF THE PROJECT

The purpose of this project is to build and support for reliable communication over an unreliable link. We need to develop a program that will reliably transfer data from one machine to another machine. We need to use UDP as the underlying network mechanism and implement TCP functionality. We need to implement our own socket type, called BRP (*Basic Reliable Protocol*) socket that will guarantee that any message sent using a BRP socket is always delivered to the receiver.

## **PRINCIPLE OF COMPUTER SCIENCE IS BEING STUDIED:**

Parallel thread communication in between two threads R and S, thread R used to receive messages from the UDP socket and thread S handles the timeouts and retransmissions, providing reliable data transfer, using C programming, creating library functions in .h file and include in .c files.

## HOW THE PROJECT IS DESIGNED

We have designed this project by different user defined functions like:

•    r\_socket – used for creating sockets

•    r\_bind – used for binding the socket with given port and IP address

•    r\_sendto – used for sending messages between the two users

•    r\_recvfrom – used for receiving messages from other user

•    r\_close –  used for closing the socket

## CONTENT OF OTHER SECTIONS

* Background
* Software Design
* Implementation
* Tests and data collected
* Results of Tests
* Brief Conclusion

# Background

## Transport Layer:

The Transport layer is the link between the Application layer and the lower layer that are responsible for network transmission. This layer accepts data from different conversations and passes it down to the lower layers as manageable pieces that can be eventually multiplexed over the media.

The Transport Layer is responsible for establishing a temporary communication session between two applications and delivering data them. TCP/IP uses two protocols to achieve this:

Transmission Control Protocol (TCP)

User Data-gram Protocol (UDP)

### Primary Responsibilities of Transport layer Protocols:

1. Tracking the individual communication between applications on the source and destination hosts.

2. Segmenting data for manageability and reassembling segmented data into streams of application data at the destination.

3. Identifying the proper application for each communication stream.

### Transport Layer Reliability:

Different applications have different transport reliability requirements TCP/IP provides two transport layer protocols, TCP and UDP

### Transmission Control Protocol (TCP)

* Provides reliable delivery ensuring that all of the data arrives at the destination.
* Uses acknowledged delivery and other processes to ensure delivery
* Makes larger demands on the network – more overhead (slower)

### User Data-gram Protocol (UDP)

* Provides just the basic functions for delivery – no reliability
* Less overhead

### TCP or UDP

* There is a trade-off between the value of reliability and the burden it places on the network.
* Application developers choose the transport protocol based on the requirements of their application.

# Software design:

List modules the program has, the functions and the sequence graph shows the dependency of the module, flow chart or other diagram, challenges and how you resolved them.

## Requirement Analysis

According to the project description document, we first tried to list all the components in our project as below:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Category | Usage | Description |
| r\_socket() | lib 🡪 function | open and UDP socket  with BRP | create 2 threads R and S |
| R: handle all messages received from the UDP socket |
| S: handle the timeouts and retransmissions |
| r\_bind() | lib 🡪 function | bind the socket with some address-port | port number in user1.c should be 50000+2\*first 3 digits  port number in user2.c should be 50000+2\*first 3 digits +1 |

|  |  |  |  |
| --- | --- | --- | --- |
| r\_sendto() | lib 🡪 function | send the message with other headers to the receiver | add a message sequence number and message type at the beginning of the message |
| store the message, sequence no. and destination address port in the unacknowledged-message table before sending the message |
| each entry in the table, there’s a time field that is filled up initially with the time of first sending the message |
| r\_recvfrom() | lib 🡪 function | receive data message and ack message from the sender | look up the received-message table to see if any message is received:  yes 🡪 return the first message and delete it from the table  no 🡪 block the call 🡪 sleep() then see again if the message is received, return to the user only when a message is available |
| call dropMessage(float p) to simulate random packet loss |

|  |  |  |  |
| --- | --- | --- | --- |
| r\_close() | lib 🡪 function | close the socket | kill all the threads |
| free all memory associated with the socket |
| if there’s any data in the received\_message table 🡪 discard |
| combined message | lib/.h 🡪 predefined data structure | store the message itself with more information | sequence #  data/ack  content |
| received-message table | lib/.h 🡪  predefined data structure | store the message entry information | message  size  address port  latest sent time  \*the maximum size of this table will not be more than 50 at any time |
| unacknowledged-message table | lib/.h 🡪 predefined data structure | store the unacknowledged message | same structure as above  \*the maximum size of this table will not be more than 50 at any time |
| socket entry table | lib/.h 🡪 predefined data structure | store the socket information as threads (multi-sockets created) | assume that the maximum number of BRP sockets that can be opened is 25 |
| dropMessage() | lib 🡪 predefined function | create results for packet loss simulating | random p < defined p, reutrns 1 🡪 considered as lost |
| random p >= defined p, returns 0 🡪 normal recieved |

Table 1 : Requirements

## Design

The invocation/ pointing relationship between each basic function/structure based on our analysis is represented as in the graphs below:

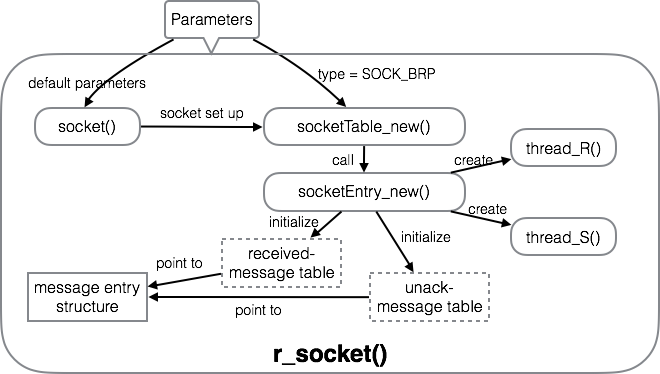


Figure 1: r\_Socket Design

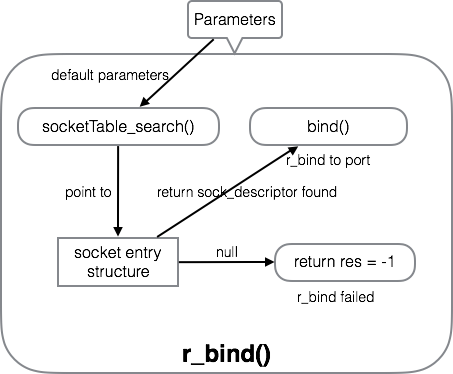


Figure 2 : r\_bind design

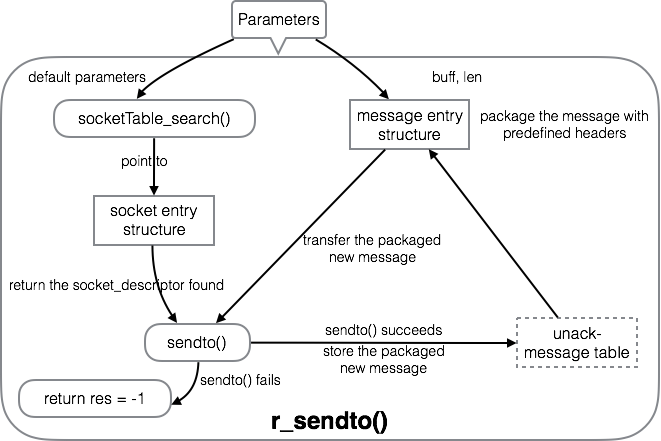


Figure 3 : r\_sendTo design

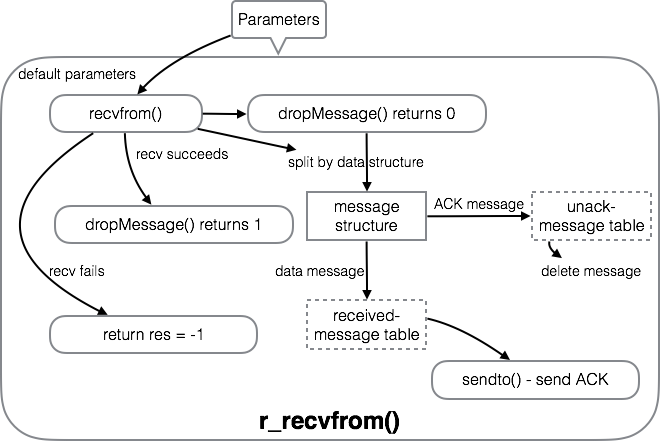


Figure 4 : r\_recvFrom Design

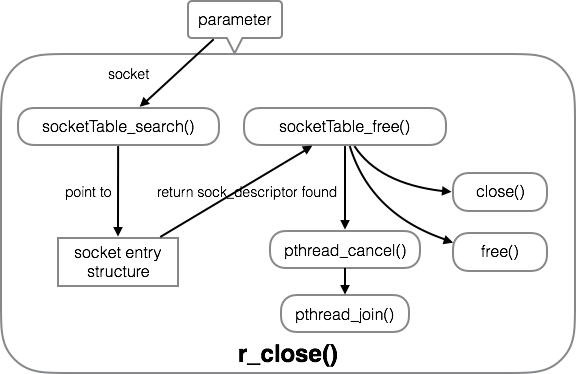


Figure 5 : r\_close design

## Challenges and Problems

### Thread

As described in the project document “Thread R handles all messages received from the UDP socket, and thread S, handles the timeouts and retransmissions.”, threadR&threadS have their separate functions during message transmissions, and both of them should be created in the r\_socket() function. If we just simply pthread() a new thread to deal with timeout or received messages, it will be difficult to clarify each thread’s function and specification. Then we tried to list the specification of each thread:

|  |  |
| --- | --- |
| Thread | Function Description |
| R | handles all messages received from the UDP socket |
| when it receives a message, if  a data message 🡪 stores it in the **received-message table**, sends an ACK message to the sender  an ACK message in response to a previously sent message 🡪 updates the **unacknowledged-message table** 🡪 take out the message for which the ACK has arrived  the maximum size of these tables will not be more than 50 at any time |
| S | handles the timeouts and retransmissions |
| sleeps for some time (T), wakes up periodically |
| On waking up, scans the unacknowledged-message table (for all messages) to see if any of the messages timeout period (set to 2T) is over (current time – entry time). If  yes 🡪 retransmits the message and resets the time  no 🡪 no action |

Table 2 : Threads Table

Then tried to exchange the “start” parameter part with predefined thread main() in the command:

pthread\_create

(pthread\_t \**thread*, const pthread\_attr\_t \**attr*, void \*(\**start*)(void \*), void \**arg*)

*thread*

Is the location where the ID of the newly created thread should be stored, or NULL if the thread ID is not required.

*attr*

Is the thread attribute object specifying the attributes for the thread that is being created. If *attr* is NULL, the thread is created with default attributes.

*start*

Is the main function for the thread; the thread begins executing user code at this address.

*arg*

Is the argument passed to *start*.

### Mutual Exclusion

Multi-thread requires proper mutual exclusion. We tried to search mutual exclusion lock using example and format from some C related website:

Initialize the POSIX thread mutex 🡪

Lock the POSIX thread mutex during operation on tables 🡪

Unlock the POSIX thread mutex after completion of operating 🡪

Destroy the POSIX thread mutex when r\_close(socket)

We have to admit that we just tried to embedded the mutex and related statements into our rsocket.c, and make sure that there’s no bug while compiling the code. But there’s no specified testing cases design for testing the performance of it till the report submission due.

### Data Structure Creation and Pointers

At the beginning of our design, there’re the message struct, 2 message table struct(received message entry and unacknowledged message entry) and socket table struct. During the group discussion about the pre-defined functions, we found the message entry struct for each table can be combined into one and we need more level for store the parallel sockets and the corresponding socket entry. More complicated the structure is, more pointer mass and bugs we created.

# Implementation

## rsocket.h

The static variables and structures were clarified in rsocket.h which includes but not limited to:

SOCK\_BRP 🡪 BRP socket type

BRP\_MAX 🡪 maximum number of sockets can be opened

MSG\_MAX\_COUNT 🡪 maximum table size

T 🡪 timeout checking interval

P 🡪 packet loss simulation probability

gettimesec() function

message type enum

message struct, message entry struct, socket entry struct

r\_socket(), r\_bind(), r\_sendto(), r\_recvfrom(), r\_close()

## rsocket.c

Defined the main 5 predefined functions:

r\_socket(), r\_bind(), r\_sendto(), r\_recvfrom(), r\_close()

the internal logic and function invocation was represented in 3.2 design.

The functions to defined thread function of thread R and thread S are:

void\* \_R(void\* param)

void\* \_S(void\* param)

The function to simulate the packet loss is defined by function:

int dropMessage(float prob)

According to the entry and table creation functions, there’re also corresponding release and free functions such as:

*free*\_msg\_table(\_msg\_entry\_t\* msge)

*sockfd\_entry\_free(\_sockfd\_entry\_t\* sfde)*

The procedure for which sender and receiver invoke the lib functions can be described in the graph:

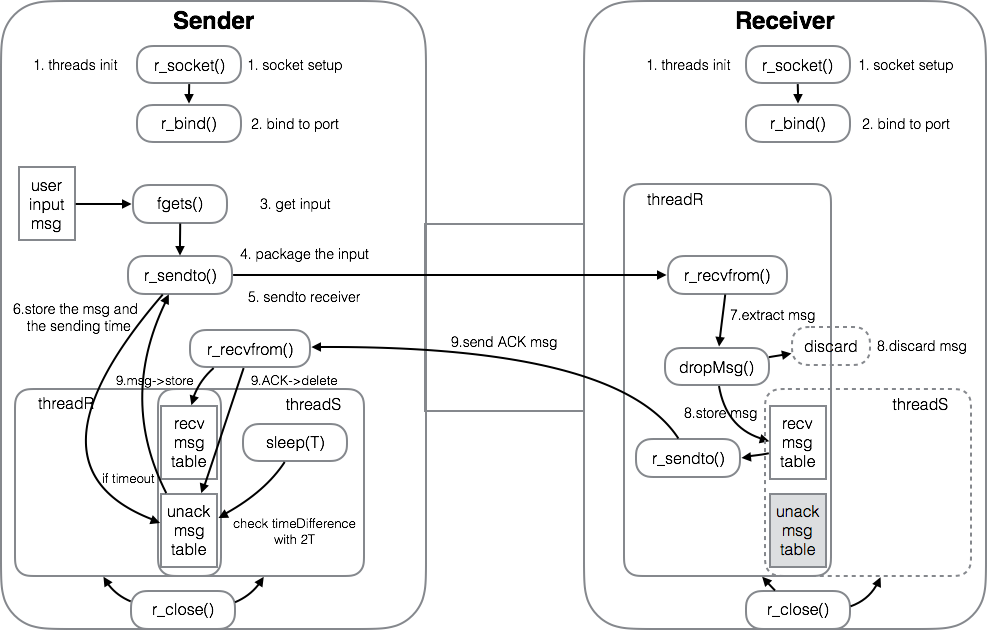


Figure 6 : Workflow between sender and receiver

# Tests

The string considered while testing: **1234567890abcdefghijklmnopqrstuvwxyz**

Length of the string is: 36

The tests are as shown below with different probabilities of dropMessage().

|  |  |  |  |
| --- | --- | --- | --- |
| Probability 0.05 | | | |
| Send | Receive | Total | Average |
| 36 | 39 | 38.2 | 1.061 |
| 36 | 39 |
| 36 | 37 |
| 36 | 37 |
| 36 | 39 |

|  |  |  |  |
| --- | --- | --- | --- |
| Probability 0.10 | | | |
| Send | Receive | Total | Average |
| 36 | 39 | 40 | 1.111 |
| 36 | 37 |
| 36 | 40 |
| 36 | 39 |
| 36 | 45 |

|  |  |  |  |
| --- | --- | --- | --- |
| Probability 0.15 | | | |
| Send | Receive | Total | Average |
| 36 | 48 | 45 | 1.25 |
| 36 | 47 |
| 36 | 42 |
| 36 | 46 |
| 36 | 42 |

|  |  |  |  |
| --- | --- | --- | --- |
| Probability 0.20 | | | |
| Send | Receive | Total | Average |
| 36 | 39 | 43.4 | 1.205 |
| 36 | 45 |  |  |
| 36 | 45 |  |  |
| 36 | 42 |  |  |
| 36 | 46 |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Probability 0.25 | | | |
| Send | Receive | Total | Average |
| 36 | 50 | 46.8 | 1.3 |
| 36 | 49 |
| 36 | 45 |
| 36 | 46 |
| 36 | 44 |

|  |  |  |  |
| --- | --- | --- | --- |
| Probability 0.30 | | | |
| Send | Receive | Total | Average |
| 36 | 54 | 52.6 | 1.461 |
| 36 | 54 |
| 36 | 51 |
| 36 | 58 |
| 36 | 46 |

|  |  |  |  |
| --- | --- | --- | --- |
| Probability 0.35 | | | |
| Send | Receive | Total | Average |
| 36 | 52 | 56.6 | 1.572 |
| 36 | 52 |
| 36 | 59 |
| 36 | 63 |
| 36 | 57 |

|  |  |  |  |
| --- | --- | --- | --- |
| Probability 0.40 | | | |
| Send | Receive | Total | Average |
| 36 | 57 | 59 | 1.638 |
| 36 | 56 |
| 36 | 55 |
| 36 | 69 |
| 36 | 58 |

|  |  |  |  |
| --- | --- | --- | --- |
| Probability 0.45 | | | |
| Send | Receive | Total | Average |
| 36 | 57 | 64.2 | 1.783 |
| 36 | 68 |
| 36 | 60 |
| 36 | 73 |
| 36 | 63 |

|  |  |  |  |
| --- | --- | --- | --- |
| Probability 0.50 | | | |
| Send | Receive | Total | Average |
| 36 | 79 | 78 | 2.166 |
| 36 | 81 |
| 36 | 73 |
| 36 | 78 |
| 36 | 79 |

Table 3 : Result tables with average retransmission

The graph shown below is related change in the retransmission of data with respect to change in dropMessage() probability.

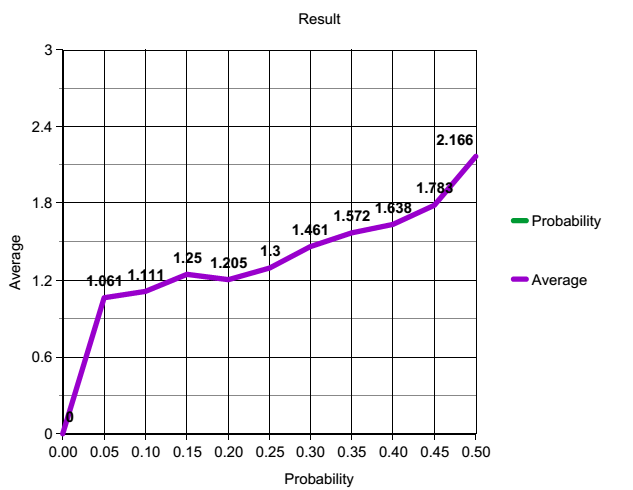


Figure 7: Result graph with different probabilities

# Results:

The program executed successfully by using **rsocket.h, user1** and **user2**. The below shown us the screenshot with running program.

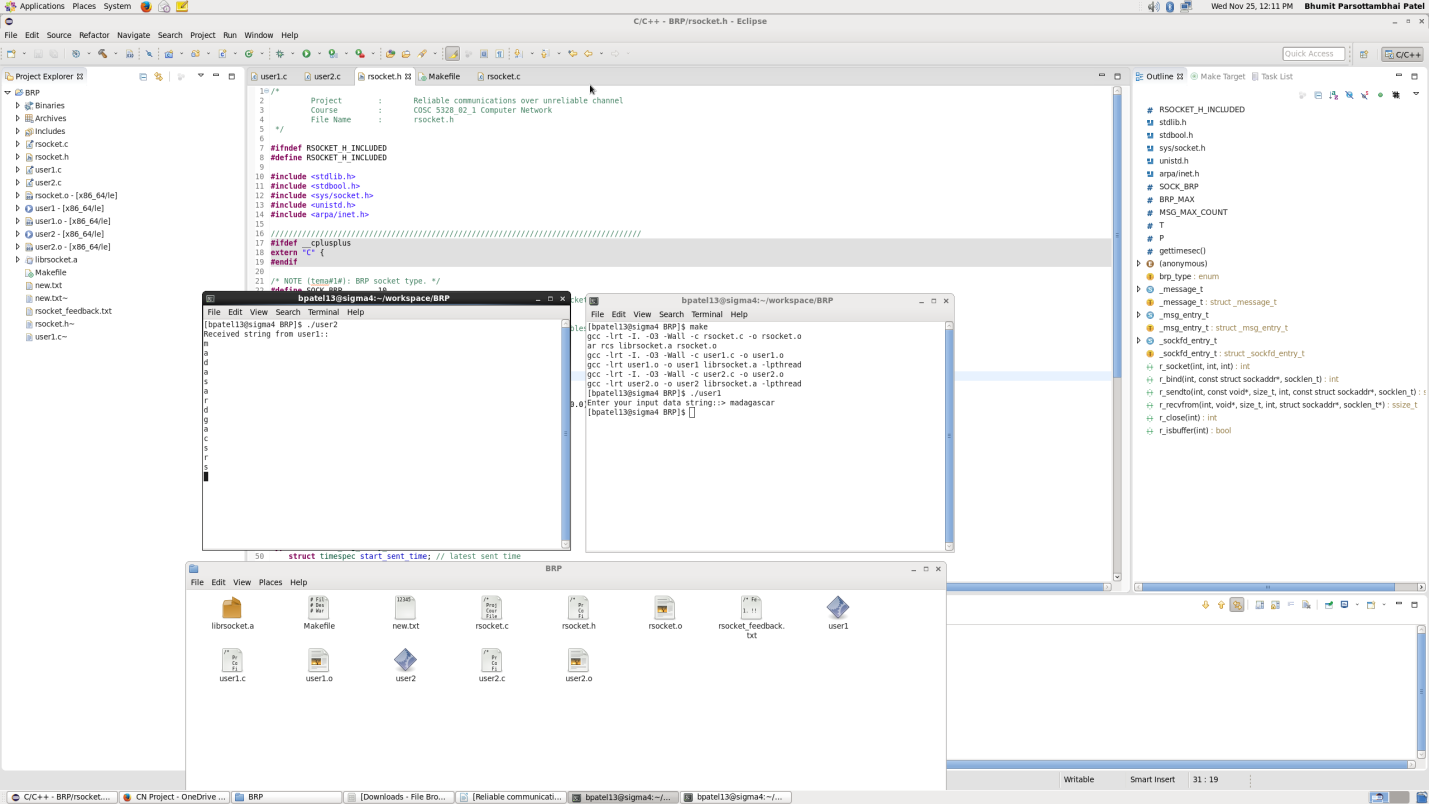


Figure 8 : Complete view of user1 and user2

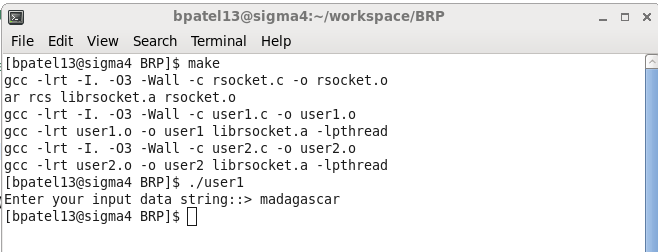


Figure 9 : User1 Terminal

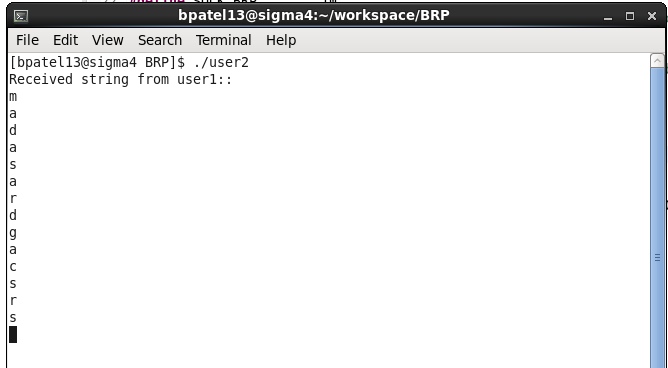


Figure 10 : User2 Terminal

# Conclusion

The main concern of the project is to provide reliability over unreliable channel. Everything is about creating a library with methods like r\_socket(), r\_bind(), r\_sendTo(), r\_recvFrom() and r\_close(), where in the methods can be helpful to provide reliability over unreliable channel. The project deals with the network programming to provide the required features. While programming on this, a library has been created with five methods. And threads like R and S are created to send and receive the data simultaneously between the communicating hosts.