Peer-to-Peer Messaging

COMP 3203 Project Report

Katherine Beltran - 100939080

Abdul Bin Asif Niazi - 100917191

Victoria Gray - 100936938

Nicholas Rivard - 100650469

December 7, 2015

Table of Contents

Section 1: Introduction

1.1 Context

In today's world, where communication is increasingly done over the internet, the protection of personal data is of the utmost importance. Unfortunately, using the common client-server architecture, the transmission of a message, whether it is a text, an email, a photo, etc, relies on a central server to receive it and then transmit it towards the destination. The server can then store any data that is sent through it, regardless of a sender's preferences. This has, at times, led to disastrous outcomes, such as hackers breaking into servers and stealing information or corporations tracking their customers' personal information in violation of privacy laws. Another issue is that the data flowing through such a server can potentially be intercepted en masse if the server is compromised.

1.2 Problem Statement

In this project, we are circumventing the problem of insecure centralized messaging systems. Our motivation for this is to mitigate the bulk collection of data by potentially insecure servers, like those at Sony, Facebook and Target. While such centralized servers are ideal targets for those intent on stealing information, there is no one target containing much data in a decentralized peer to peer (P2P) system. Furthermore, breaking into one node does not necessarily reduce the cost of breaking into any other node on the system. This architecture therefore compartmentalizes data. This means that in order to obtain the data of many users, a perpetrator would have to independently break into many nodes instead of just one. Therefore, in principle, a decentralized P2P system replaces one obvious high-value target with many low-value targets, making it that much more expensive to compromise the system in the large.

1.3 Result

At the end of this project, we were successful in constructing a first generation P2P messaging application with one community server collecting and distributing the contact information of the currently connected nodes. The community server is used as a meeting place for clients to find out who is online and ready to talk. Multiple clients may connect to this server and see who is currently connected. Clients can then use this information to connect to one another directly in a private two-way conversations. Our architecture is similar to a private P2P network using a Direct Connect (DC) Protocol, but to send text messages instead of files. We have implemented our interface using the TCP protocol to ensure that all sent messages are received by the clients they are sent to.

1.4 Outline

The following sections of this report are structured as follows: Section 2 presents the background information relevant to our project; Section 3 describes in detail how we have come to obtain our result, as well as the steps we chose to achieve our goal; Section 4 is where we evaluate our result; finally, we present our conclusions in Section 5.

Section 2: Background

P2P communication was first popularized by Napster in 1999[1]. Napster was a music file sharing application with a set of central servers that linked together users who had files with users who were requesting those files. Napster is just one of the first generation of P2P systems. Kazaa and Gnutella are examples of second generation P2P systems; these were able to operate without any central servers. Eliminating the central vulnerability allowed users to connect remotely to each other. Freenet is one software we used for inspiration to build our P2P system. The community server is password protected and gives a list of all the currently active users. Users can use this information to connect to each other, but each node is also password-protected and requires authentication to connect. Many know the very famous P2P system called BitTorrent, however BitTorrent is known as a special case for P2P systems. Theoretically, BitTorrent is a first generation protocol, but does not create a network in the traditional sense. It uses trackers and has decentralized servers.

Section 3: Result

During our investigations for this project, we made use of many online resources that discuss P2P networks. Thorough research both aided us in writing this report and informed our choices about first, how to transfer messages directly between client nodes and second, how to implement a centralized server capable of simultaneously handling multiple clients.

Ultimately, we have created a first generation P2P messaging application with one centralized server to mediate node discovery and the use of an alternative architecture to client-server. Multiple clients can connect to our server and are able to see which users are currently online and available to have a conversation. A user can then enter another user's IP address and port number into their client program and establish a direct and private connection to have a conversation.

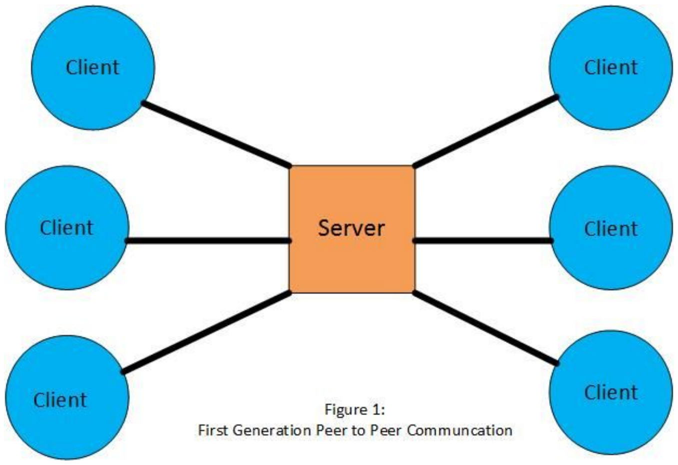


Figure 1:
First Generation Peer to Peer Communcation

Figure 1 shows six clients connected to the central server. Each client node will have two pieces of software running on their machine: one server program (*nodeServe*r) to listen for incoming connections and one client program (*client)* used to connect to the community server and to establish a connection to another user's *nodeServer*.

The community server will keep a list of all active connections. This list will contain the username, IP address and listening port of each user's *nodeServer*. A connection is established with the community server once a client connects and provides the correct password. Connected clients can access the list of active users and can then make connections to them using the *client* program. The *client* program will connect to another user's *nodeServer* and the two nodes will thus be able to communicate with each other. Each pair of nodes involved in a conversation will then be disconnected from the community server so others do not try to connect to them.

Figure 2 provides a graphical representation of how our system works. In step 1, multiple clients may connect to the community server given that they have obtained its IP and port number. Once connected, they are prompted to enter a password to access the server's database. When access is granted, the user is asked to enter their username and the port number that the *nodeServer* is listening on. Step 2 shows a user (user1) obtaining the list of active clients connected to the community server. Once another user1 selects another user to connect to, using the same client program and the IP and port number given from the community server, the user can now establish a private chat connection. Step 3 demonstrates user1 trying to establish a private chat between themself and user2. User1's *client* program will be attempting to connect to User2's *nodeServer* program. During step 4, both nodes in the private chat will be disconnected from the community server and will have a private chat established. Figures 3 explains, in detail, how a user will connect to the community server, run both client programs as well as the *nodeServer*.

Figure 2:
Abstract Concept of our
P2P

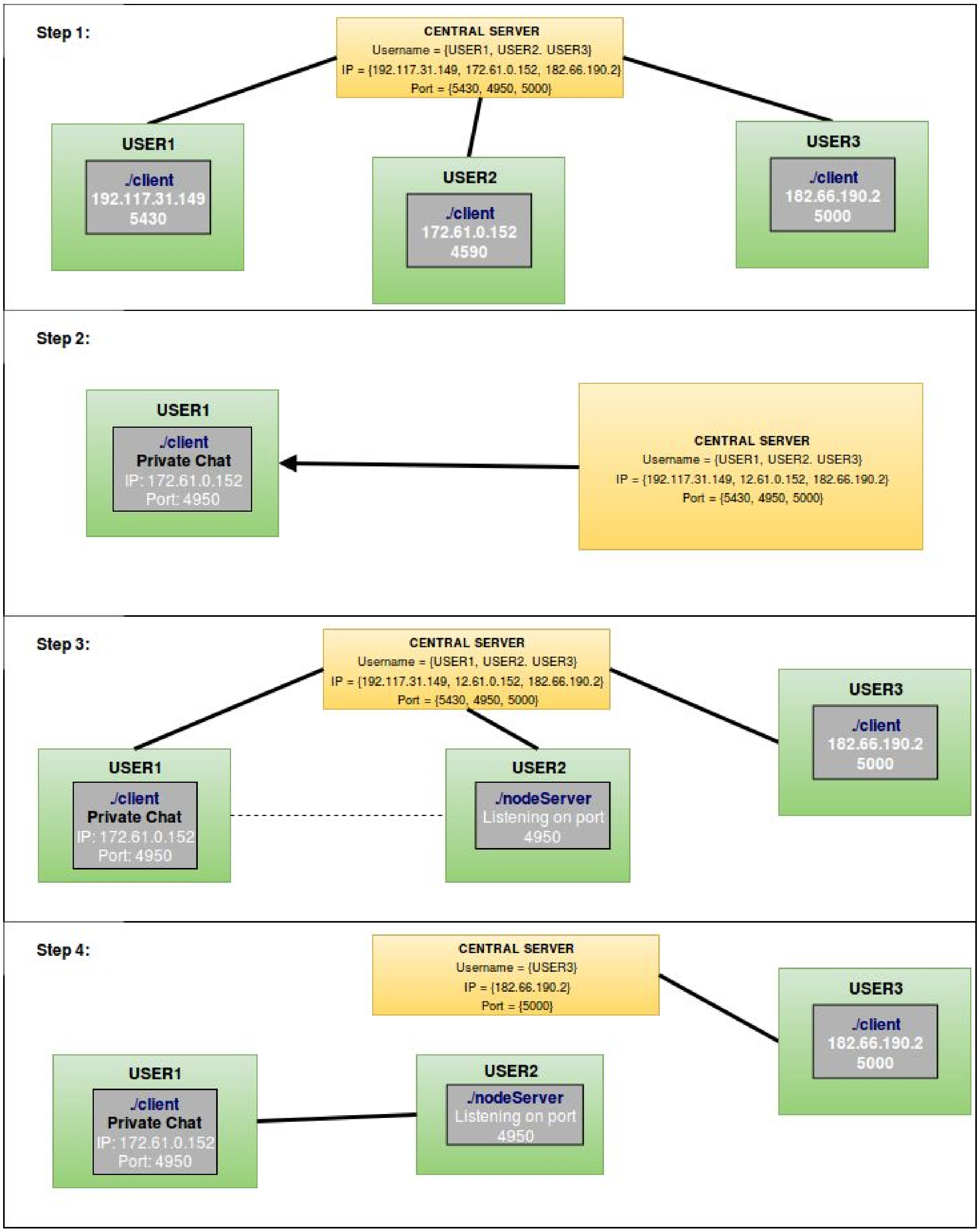


Figure 3: Explanation of Task

We have created a peer to peer application used for Linux machines. The following information provides a detailed explanation of how to connect and start our peer to peer chat.

Explanation of tasks for Community Server

****One person must have the community server running before anyone can connect. Each person that wishes to connect to the community server must know the IP address and port number of the machine where the community server is running.

1) Run **./client** choosing option 1, with the IP address of the machine where the community server is running. You will be prompted to enter the community server password.
2) Once you have been approved, enter a username and a port number - which your **nodeServer** is listening on
3) Enter any key to get the list of users with their IPs and port numbers that are connected to the community server
4) Choose from the displayed list the user you wish to connect to

Explanation of tasks for the Private Chat

<u>nodeServer</u>

1) Enter the port number in which you would like to listen for connection
   Note: this is the same port number that you gave the community server

<u>client</u>

****You will need the IP address and port number given from the community server

1) Run **./client** choosing option 2
2) Enter the IP address and port number of the user you wish to connect to (provided by the community server)
3) Enter the user specific password to start the chat
   Note: this password must first be established from the users before a connection can be made.

Our overall goal after developing a multi-client P2P messaging application was to create a second generation messaging application with no centralized server (Figure 4). Each node on the system would, in principle, be a server and client itself and be able to broadcast messages to each node. Each node would have the ability to send a message to one specific node or to broadcast a message to everyone that they are connected to. In order for two nodes to connect, the client node would need some sort of authentication, such as a passphrase. The server node would then have the authority to accept or deny the connection.

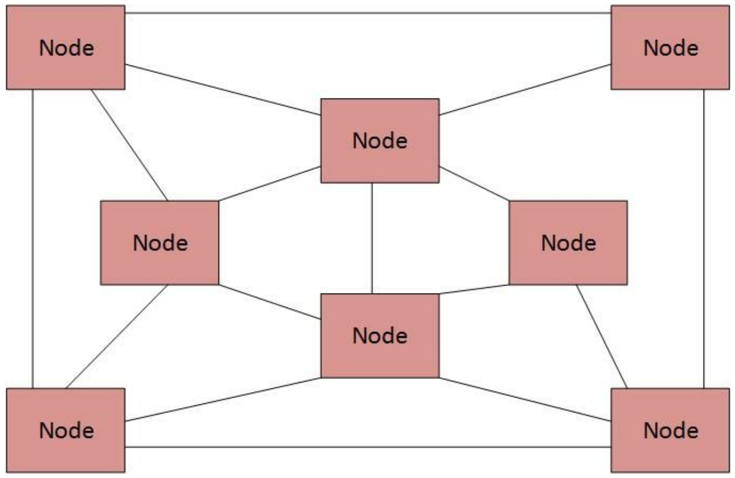
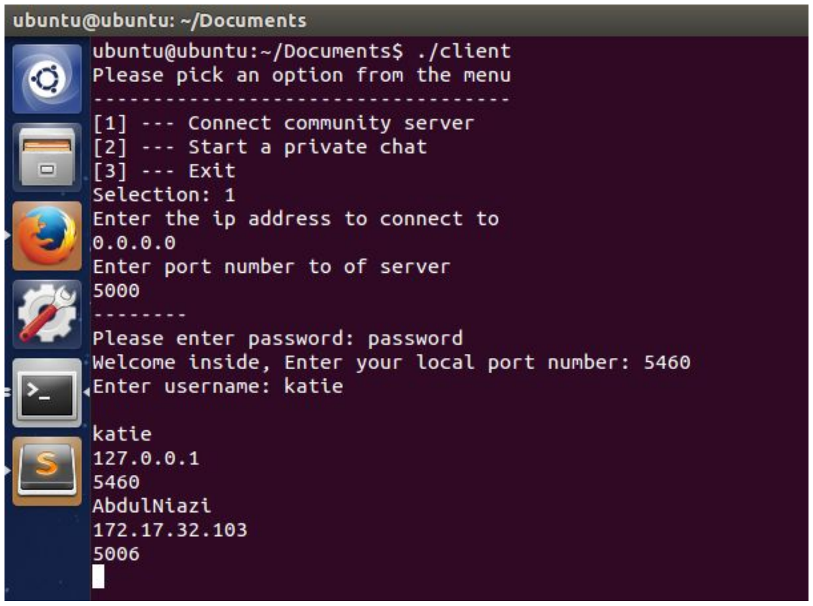


Figure 4:
Second Generation Peer to Peer

3.1 *Features*

Our P2P application has many features whether analyzing the *nodeServer program*, the community server program, or the *client* program. Figure 5 demonstrates how a user may connect to the community server. First, in order to connect to the community server, each client must provide the correct password to access the list of users connected to the community server. Once the correct password has been entered, the user is prompted for their username and port number that their *nodeServer* will be listening on. After entering any key, the list of users with the respective IP addresses and port numbers will be displayed on the screen. The user now may choose from the list who they would like to start a private chat with.

Figure 5: Client program used to connect to the Community Server with one active user

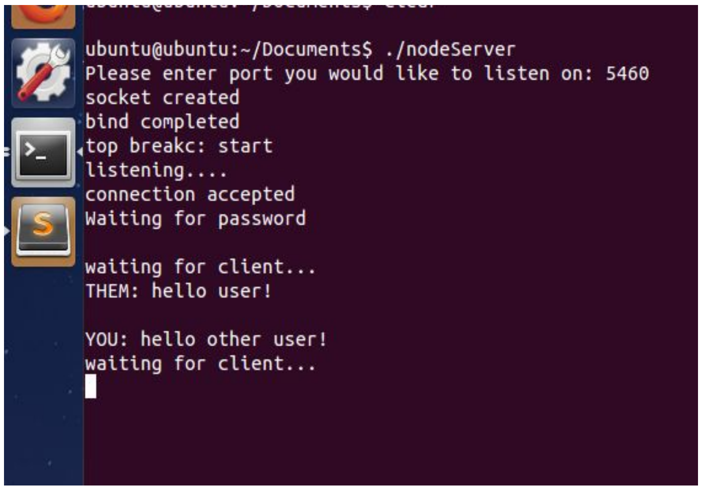


```
ubuntu@ubuntu: ~/Documents
ubuntu@ubuntu:~/Documents$ ./client
Please pick an option from the menu
-----------------------------------
[1] --- Connect community server
[2] --- Start a private chat
[3] --- Exit
Selection: 1
Enter the ip address to connect to
0.0.0.0
Enter port number to of server
5000
--------
Please enter password: password
Welcome inside, Enter your local port number: 5460
Enter username: katie

katie
127.0.0.1
5460
AbdulNiazi
172.17.32.103
5006
```

After a user has obtained all the necessary information to perform a private chat, they may enter the IP address and port number into private chat option in the *client* program (Figure 6). That specific user must then acquire the password of the user they are trying to converse with. After the correct password is confirmed, both user may start their private chat. Our private peer to peer application runs in a linear form; a user may only send a message after it has received a message, the very first message is sent from the user requesting a chat.

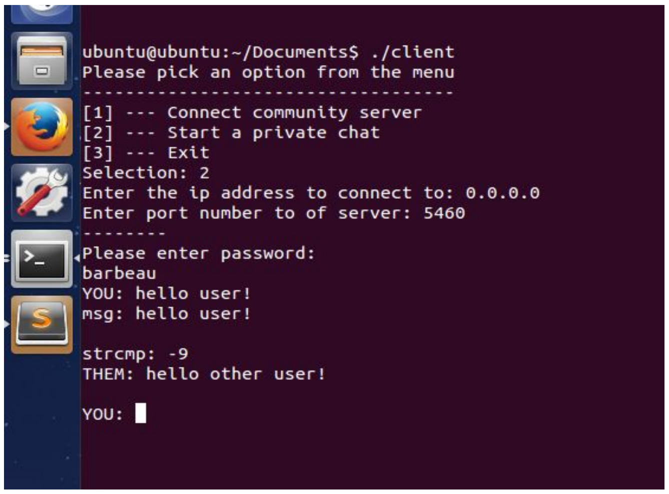
Figure 6: nodeServer and client program used to start a private chat



```
ubuntu@ubuntu:~/Documents$ ./nodeServer
Please enter port you would like to listen on: 5460
socket created
bind completed
top breakc: start
listening....
connection accepted
Waiting for password

waiting for client...
THEM: hello user!

YOU: hello other user!
waiting for client...
```



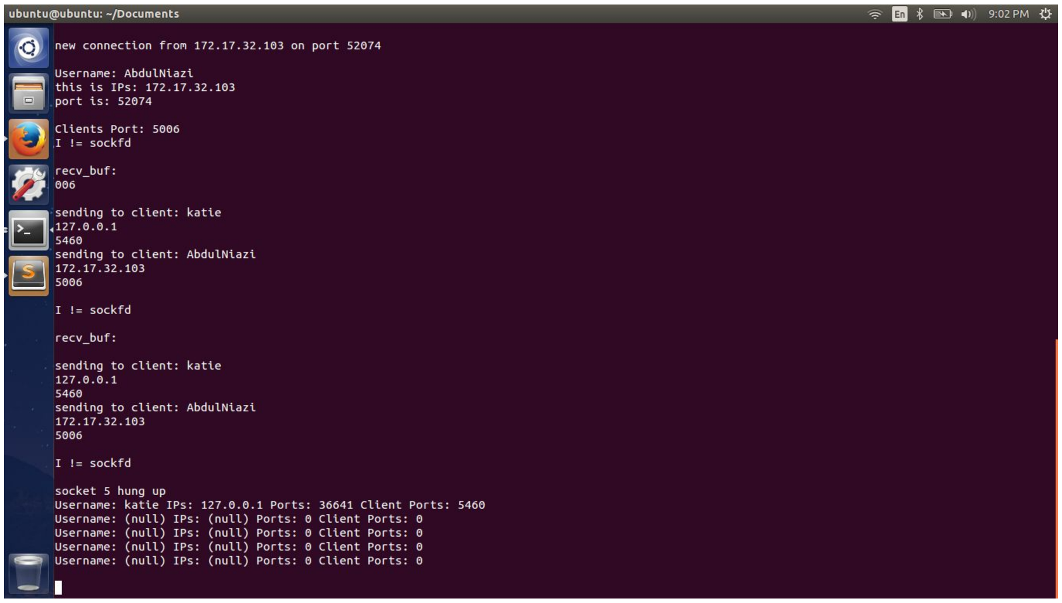
```
ubuntu@ubuntu:~/Documents$ ./client
Please pick an option from the menu
-----------------------------------
[1] --- Connect community server
[2] --- Start a private chat
[3] --- Exit
Selection: 2
Enter the ip address to connect to: 0.0.0.0
Enter port number to of server: 5460
--------
Please enter password:
barbeau
YOU: hello user!
msg: hello user!

strcmp: -9
THEM: hello other user!

YOU:
```

Some of the other features that we have incorporated into our application is when a client disconnects from the community server, their username and respective IP and port number are then removed from the community server database (Figure 7). This will generally occur when a private chat has been established by both users (Step 4, Figure 2). If either user of the private chat quits their private chat, they may reconnect to the community server to obtain new users that they wish to chat with. They will then be prompted to enter the correct community server password, as well as username and port number that their *nodeServer* will be listening on. The community server does not remember the username, IP address and port number that was previously entered into the server's database as user data is not stored.

Figure 7: Community Server updating the list of users once they disconnect (User AbduNiazi has disconnected and removed from the server's database)



```
ubuntu@ubuntu: ~/Documents                                    9:02 PM

new connection from 172.17.32.103 on port 52074

Username: AbdulNiazi
this is IPs: 172.17.32.103
port is: 52074

Clients Port: 5006
I != sockfd

recv_buf:
006

sending to client: katie
127.0.0.1
5460
sending to client: AbdulNiazi
172.17.32.103
5006

I != sockfd

recv_buf:

sending to client: katie
127.0.0.1
5460
sending to client: AbdulNiazi
172.17.32.103
5006

I != sockfd

socket 5 hung up
Username: katie IPs: 127.0.0.1 Ports: 36641 Client Ports: 5460
Username: (null) IPs: (null) Ports: 0 Client Ports: 0
Username: (null) IPs: (null) Ports: 0 Client Ports: 0
Username: (null) IPs: (null) Ports: 0 Client Ports: 0
Username: (null) IPs: (null) Ports: 0 Client Ports: 0
```

Our program makes use of encryption to protect the privacy of all traffic, including passwords, contact information and user messages. This is accomplished using an XOR cipher on all buffers before they are sent over the network. To decrypt all messages, the same XOR function is used on all received buffers. This works because applying XOR twice to a buffer (with the same key both times) results in the original contents of the buffer being output.

Figure 8: Code snippet of how our program encrypts and decrypts packets

```c
 void encrypt(char* buf){
    char temp[BUFSIZE];
    int i;
    for(i = 0; i < BUFSIZE; ++i){
         temp[i] = buf[i] ^ 'A';
    }
    memset(buf, '\0', BUFSIZE);
    strcpy(buf, temp);
}
```

3.2 *Problems and Solutions*

Throughout the creation of our P2P messaging application, we encountered many different problems. One of the first problems was that the private chat could send and receive messages, however it would not allow the sending or receiving of white spaces. This problem was resolved with the use of *fgets* rather than *scanf* to read what the user was sending to the another user.

The second problem that we faced was how the community server and clients were receiving and sending data. In some cases, the clients were receiving messages from the community server like “*Welcome Inside*”, but when the client had to send which port its *nodeServer* was listening on, the server was not receiving the information to store in the array of listening ports. In order to resolve this problem, we had added multiple print statements to see where the connection was not been synchronized. In the end, we had established that all our send and receive functions must have the same buffer size.

One of the last problems that we encountered is that the community server does not fully display the list of users connected to it. When the community server is running on one machine (for example, node1), if node1 is running *client*, node1’s client will be the only client that can see the full list of clients connected to the community server. All other clients can see their username, IP and ports and then a large white space is printed which is essentially the other users connected. However, instead of sending the list of users connected the community server only sends white spaces. To fix this problem, we received advice to write all of the information that we are sending into one buffer. Once all the information needed was written to the buffer, we would then send that buffer.

<u>Section 4: Evaluation</u>

A pre and a post-trial questionnaires (Appendix A and B respectively), were used to evaluate the performance, usability and overall appeal to a target group. The pre-trial questionnaire (Appendix A) was used to gather background information about users to establish a testing group. The post-trial questionnaire (Appendix B) was used to improve our current working P2P messaging application.

4.1 Pre-Trial Questionnaire

In order to evaluate the best targeted group for our P2P messaging application, first, we distributed our pre-trial questionnaire throughout various locations on the Carleton campus to students. Once we had received all necessary information, we began our process to establishing our targeted market. Based on the pre-trial questionnaire we learned that majority of post-secondary students do indeed use messaging applications such as Facebook messenger, Skype and Whatsapp. They are primarily used for communicating between peers for the purpose of group work and assignments. Therefore, we have targeted our study around post-secondary students in all faculties.

4.2 Usability

Based on the information provided from the post-trial questionnaire, we learned that majority of Computer Science students found our application easy and straight-forward to use, compared to non Computer Science students. Due to the fact that we did not have enough time to construct a user friendly GUI for this application, non Computer Science students found that the use of a terminal was hard to grasp, since for many of them this was the first time they had seen this type of interface. However, Computer Science students found the use of the terminal chat system easy to use with the aid of the instructions provided by Figure 3 (pg 6). They had noted that without the instructions to obtain the central server's IP and port number, connecting to the central server would have been difficult. Overall, the use of our P2P application was simple and straight-forward for those with a background of using terminals and command prompts.

4.3 Performance

Figure 9: Run Time of User Tasks

Tasks	*Time to Complete Task*
Connect to Community Server	0h 0m 0s 73ms
User entries (password, username and listening port)	0h 0m 6s 52ms
Obtain list of active users	0h 0m 0s 20ms
Error password (Community Server)	0h 0m 0s 11ms
Start private chat (including entering IP, port number, password and first message)	0h 0m 8s 52ms
Error password (after entering IP and port number)	0h 0m 1s 17ms
Message Sent/Receive	0h 0m 1s 11ms

Figure 9 provides a detailed run time of all the user tasks, including error possibilities. Based on the information given from the table, we learned that our P2P application runs in a similar time to the messaging app Whatsapp. We logged the performance times of Whatsapp given from *Learner Everyday Blog*[1] with our own logged performance times in order to obtain this conclusion.

4.4 Overall Evaluation and Improvements

Considering the information provided by the post-trial questionnaire (Appendix B), our P2P messaging application received an average rating 2.8 out of 5. Majority of users found that the application was straight-forward and simple, as long as they had used a terminal-like chat system before. Figure 10 lists some of the comments and suggestions our users have provided for future improvements.

[1] This blog logged the performance of the most used user of a group, however we modified it to log the performance of a sent message. *What I Learned Today(n.d)[3].*

Figure 10: Suggestion, Comments and Improvements from the Post-Trial Questionnaire

- did not like having to go through a central server to find user’s to chat with
- too many things you are asked to enter
- liked how we could choose a username instead of being known as a number
- preferred a user friend GUI
- liked getting updates on who was connected to the server and seeing the messages
- having a password to access the server made the application feel more secure
- would like to be able to send images and files across the network as well

Section 5: Conclusion

Throughout the course as our project evolved, so did our understanding of network principles. Surrounded by the vast network of networking possibilities, our focus centralized around the task of transferring data between nodes and across networks. We have succeeded in creating a Linux application in which clients with the application can connect to a secure authenticated network through a server and then communicate with other clients on the same network. We have achieved this by programming a client/server program in C and then modifying it to accept multiple clients. Those clients may then attempt to connect to the network, however they must know the password requirements before their IP addresses are added to the network and before they can send messages across. This project has allowed for our greater understanding of network communication and led us to explore and accomplish P2P messaging tasks.

5.1 Future Works

In the future we intend to incorporate SSDP host discovery, which uses UDP multicast on port 1900. This would allow nodes to locate each other and then communicate with each other, without the need for a centralized server to mediate contact information transfer. This capability would bring our program one step closer to being ad-hoc. We built a separate prototype SSDP ping system, but had difficulty testing it and never managed to successfully incorporate it into our overall program. The main problem we encountered in testing SSDP was that both our program (and also commercial SSDP programs) could not properly detect UPnP programs running on the networks. We hypothesize that we were not running the UPnP programs properly.

Contribution of Team Members

Katherine Beltran: All diagrams and figures found in the report, Section 2, 3.1, 3.2 & 4, the merge of the community client and private chat client and disconnection of the community client from the community server.
Abdul Bin Asif Niazi: Wrote the communityserver program and helped write the nodeServer & client programs. Wrote and edited different parts of the report.
Victoria Gray: Report sections 1.1, 1.2, 1.3, 3, 4.4, 5 and combining and editing of the report. The disconnection of the community client from the server.
Nicholas Rivard: Editing of all report sections, writing of encryption and SSDP text in report, client and nodeServer code, aspects of program design, encryption and (incomplete) SSDP.

Appendix A

Pre-Trial Questionnaire

1. Do you use chat applications on your computer? (ex. Facebook Messenger, Skype)
   Please circle only ***one:***

   Yes No

   List any **if you answered yes**:

2. Do you think that your conversations should be kept private?
   Please circle only ***one:***

   Yes No

3. How well does your current messaging application work?

Not Very Well	1	2	3	4	Execellent

4. In the box below, please state some changes that you wish your current messaging application would have.

5. Would you like to participate in our Peer to Peer Application test?
   Please circle only ***one:***

   Yes No

Appendix B

Post-Trial Questionnaire

1. How would you rate the application overall?

Not Very Well	1	2	3	4	Execellent

2. Was using the application simple and straightforward?
   *Please circle only **one:***

   Yes No

3. What did you like about the application?

4. What did you dislike?

5. What are some improvements?

6. Would you recommend this application to someone else?
   *Please circle only **one:***

   Yes No

References

1. Peer to Peer File Sharing. (n.d.). Retrieved December 6, 2015, from https://en.wikipedia.org/wiki/Peer-to-peer_file_sharing#History
2. Peterson, L., & Davie, B. (2003). *Computer networks: A systems approach* (3rd ed.). Amsterdam: Morgan Kaufmann.
3. What I Learned Today. (n.d.). Retrieved December 6, 2015, from http://learnereveryday.blogspot.ca/2015/04/whatsapp-group-chat-data-analysis.html