INTERNET RELAY CHAT APP

Based on RFC 2812 & 2813

MARCH 30, 2024

BAHA' QUDIESAT

Project report

Table of contents

INTRODUCTION

OBJECTIVE:

Development of IRC (internet relay chat) application following the rules in RFC 2812.

Hope to gain:

- Knowledge in creating TCP-based multi-threaded apps in c++.

- Concurrent interactions between clients-server using TCP-based connection and their architecture.
- Debugging: debugging is a big issue when you have multiple threads working on the same time.
- Handling concurrent commands in a way that does not produce race conditions, as I have learned to use locks in c++.

PROJECT DESIGN

Server implementation and code

```
int main()
{
    setNumericReplies();
    try
    {
        ChatServer server;
        // server.start();
        thread serverthread(st, ref(server));
        std::string input;
        //runAllTests();
        while (std::getline(std::cin, input) and input != "quit")
        {
            //runAllTests();
        }
        server.stop();
    }
    catch (const std::exception &e)
    {
        std::cerr << "Error: " << e.what() << std::endl;
    }

    return 0;
}
```

At first, we create an instance of the class chat server which will act as the server, we will send a reference to it to the st function in a new thread. Which is responsible for invoking the start function of the server. Which is responsible for starting the server and the listening for the client commands in that thread.

When the server receives a connection request, it creates a Client object as each connection will handle only and exactly one client. This object is managed it this thread, and it executes the handle client method. The handle client method is responsible for receiving the commands from the client, extracting the command, and invoking the corresponding method to handle the client command.

For Example: if the received command from client is JOIN, the JOINCOMMAND() method is invoked. And so on for other supported commands.

As mentioned above, each client is managed in its own thread, so we can handle multiple clients connected at the same time. Each new client connected to the server we will create a new thread to handle its requests. And we map each client to a port so we make sure each client's communication is independent and concurrent with other clients.

Client implementation and code

```
int client_socket = socket(AF_INET, SOCK_STREAM, 0);
if (client_socket == -1) throw runtime_error("DEBUG: ERR_FAILD_TO_CREATE_SOCK");
//connect to serv
sockaddr_in server_address{};
server_address.sin_family = AF_INET;
server_address.sin_port = htons(PORT);
server_address.sin_addr.s_addr = inet_addr(server_ip.c_str());
if (connect(client_socket, (sockaddr*)&server_address, sizeof(server_address)) == -1)
    cerr << "DEBUG: ERR_FAILD_TO_CONNECT_TO_SERVER" << strerror(errno) << endl;
cout << "DEBUG: conected to server" << server_ip << " on port" << PORT << endl;
//threads for sending receiveing MSGS
thread receive_thread(receiveMessage, client_socket);
thread send_thread(sendMessage, client_socket);
//wait for threads to finish
receive_thread.join();
send_thread.join();
```

The main function of the client will establish connection to the server, then 2 threads created, one thread for receiving from server and the other one is to send to the server. The reason for that is if the server may send commands to client in the future we don't want the send message to wait while the receive message completely handles any server request.

NICK COMMAND

Usage: NICK <nickname>

As described in RFC, NICK command is used to give user a nickname or change the existing one.

Note: must be the first command and the user can't use any command except after adding a nickname for the first time.

Implemented error codes:

ERR_NONICKNAMEGIVEN:

ERR_ERRONEUSNICKNAME:

ERR_NICKNAMEINUSE:

How to handle the nick command:

```
381  void Client::NickCommand(vector<string> params)
382  {
383      if (params.size() == 1)
384      {
385          sendToClient(431);
386      }
387      else if (params.size() > 2)
388      {
389          sendToClient(432);
390      }
391      else
392      {
393          if (!ClientManager::isNickNameAvialble(params[1]))
394              sendToClient(432);
395          else if (!ClientManager::isValidNickName(params[1]))
396              sendToClient(433);
397          else
398          {
399              this->nickName = params[1];
400          }
401      }
```

First we will check if the number of parameters is exactly 2, if it is not then we will produce the corresponding error code. Then we will check that the nickname is available (no other user with the same nickname) it is specified in the RFC page 11 that nickname must be unique.

EXTRA PART: Then we will check if the nickname follows the rules of names. The RFC does not specify the rules of naming but there are some rules specific to channels so excluding those and empty strings the nickname can be anything. Also nick name should be less than 10 in length as specified in the RFC document.

```
bool ClientManager::isValidNickName(string nickName)
{
    return nickName.length() <= 9 && nickName[0] != '&' && nickName[0] != '!' && nickName[0] != '+' && nickName[0] != '#';
}
```

USER COMMAND

Usage: USER <username> * * :<realname>

The USER command must be the second command the client must perform and is used after choosing the nickname to specify the username, hostname and realname of a new user.

The Mode is not required for this project but must be given in the command for using it in Project 2.

The user can only register one time.

Implemented error codes:

ERR_NEEDMOREPARAMS: when some of the parameters are missing.

ERR_ALREADYREGISTRED: if the user has already registered.

After choosing the nickname and registering the user, the client can use the functionalities of the chat app. We will split the real name based on the ":" character since realname may contains spaces.

```
void Client::UserCommand(vector<string> params, string message)
{
    if (registered)
    {
        sendToClient(462);
    }
    if (params.size() < 5)
    {
        this->sendToClient(461);
    }
    else if (nickName.empty())
    {
        this->sendToClient(431);
    }
    else if (!userName.empty())
    {
        this->sendToClient(462);
    }
    else if (params[2].size() > 1)
    {
        sendToClient(472); // ewview
    }
    else if (params[4][0] != ':')
    {
        sendToClient(472); // review
    }
    else
    {
        this->userName = params[1];
        this->mode = params[2][0];
        this->realName = split(message, ':')[1]; // may contains space!
        registered = true;
        string paramsMassge = getWelcomeMessageParam();
        this->sendToClient(1, paramsMassge);
```

TIME COMMAND

The time command is used to query local time from the server.

There will be no target parameter, as it is not required for this project. (there is only one server).

```
string Client::getCurrentTime()
{
    time_t rawtime;
    struct tm *timeinfo;
    time(&rawtime);
    timeinfo = localtime(&rawtime);
    return ("Current local time and date: %s", asctime(timeinfo));
}
```

JOIN COMMAND:

The JOIN command will add the client to a channel specified in the arguments, keys are not used anywhere so we will exclude them from the parameter list. So the JOIN command will have only either channel names to join, or the special parameter “0” that leaves from all the channels. In the special parameter “0”, we will use the PART command by looping across the channels that the client present it.

In the following screenshot, this is how we handle the JOIN command.

```
oid Client::joinCommand(vector<string> params)
 // JOIN #foo,&bar fubar
   if (params.size() == 1)
   {
       sendToClient(461);
   }
   else if (params.size() == 2 && params[1][0] == '0')
   {
       channelManager.removeClientFromAllChannels(nickName);
       sendToClient("You just left all Channels!");
   }
   else
   {
       sendToClient(channelManager.joinClientToChannels(split(params[1], ','), nickName));
   }
```

If the second parameter is not zero, the join Command function will invoke the join client to channels function.

```
clients_mutex.lock();
if (!ClientManager::isNickNameAvialble(params[1]))
    sendToClient(433);
else if (!ClientManager::isValidNickName(params[1]))
    sendToClient(432, params);
else
{
    this->nickName = params[1];
}
clients_mutex.unlock();
```

```
public:
    string joinClientToChannels(vector<string> channels, string nickName)
    {
        string res = "";
        for (string channel : channels)
            res += joinClientToChannel(channel, nickName);
        return res;
    }
    string joinClientToChannel(string channelName, string nickName)
    {
        if (!isAvalidChannelName(channelName))
            return channelName + "   " + numericReplies[403];
        if (channels.find(channelName) == channels.end())
        {
            channels.insert(make_pair(channelName, Channel(channelName)));
        }
        Client *clientPtr = getClientWithNickName(nickName);
        Client client = *clientPtr;
        return channels[channelName].addUser(client);
    }
```

Extra part: the users should know of any new users listening to the channel, as in the current description of the RFC the program will not inform the users of any changes of added uses and who their messages gets sent to. And it is not good that the user have to check names list each time he wants to send a message.

```
string addUser(Client client)
{
    channelMembers.insert(std::make_pair(client.nickName, client));
    sendTextToAllUsersException(client.nickName + " has joind the chennel", this->name);
    return channelInfo();
```

Instead, in my implementation, all channel clients will get a message which user gets add to which channel.

PART COMMAND

The client must be able to leave a channel using the PART command and channel name, he can also give a message before leaving the channel. The following code will show you how I handled the join command. The implemented error messages are ERR_NEEDMOREPARAMS, ERR_NOSUCHCHANNEL, ERR_NOTONCHANNEL.

```
void Client::partCommand(string message)
{ // PART #oz-ops,&group5 :I lost
    vector<string> params = split(message, ' ');
    if (params.size() == 1)
        sendToClient(461);
    else
    {
        vector<string> p = split(message, ':');
        if (p.size() <= 2)
            sendToClient(461);
        vector<string> chen = split(p[1], ',');
        sendToClient(channelManager.partFrom(nickName, chen))
    }
```

```
string partFrom(string userNickName, vector<string> channelsName, string partMessage = "left the channel.")
{
    string res = "";
    for (size_t i = 0; i < channelsName.size(); i++)
    {
        auto it = channels.find(channelsName[i]);
        if (it == channels.end())
            res.append(channelsName[i] + " " + numericReplies[403]);
        else
            res += it->second.part(userNickName, partMessage);
    }
    return res;
```

```
// Function to remove a user from the channel
string part(string userNickName, string partMessage = "left the channel.")
{
    auto it = channelMembers.find(userNickName);
    if (it == channelMembers.end())
    {
        return numericReplies[442];
    }
    channelMembers.erase(userNickName);
    for (const auto &pair : channelMembers)
    {
        Client c = pair.second;
        c.sendToClient(userNickName + " " + partMessage);
    }
    return "You just left the " + name + " Channel";
}
```

Extra part:

Users will be able to see who left which channel, and the client who is leaving the channel must get a confirmation message that he left the channel (shown in second screenshot). And the user gets a confirmation message of leaving the channel in the part function (screenshot 3).

OTHER COMMANDS:

Design for how to handle commands are the same for the rest of the commands so I will not include the code for them inside the report. Except for the quit command. I implemented it inside the client code.

QUIT COMMAND

```
if (message == "QUIT")
{
    cout << "SIGNING OUT..." << endl;
    close(c_socket); // Close the socket
    exit(0); // Exit the program
}
```

Error codes implemented:

- User command: 001 RPL_WELCOME, 002 RPL_YOURHOST, 003 RPL_CREATED and 004 RPL_MYINFO upon successful registration of the client.
- NICK command: ERR_NONICKNAMEGIVEN, ERR_ERRONEUSNICKNAME: nickname should not start with reserved characters that has meaning. Ex: '&' , '#' …, ERR_NICKNAMEINUSE.
- List command:322 RPL_LIST, 323 RPL_LISTEND.
- TOPIC command: 331 RPL_NOTOPIC, 332 RPL_TOPIC.
- Names command: 353 RPL_NAMREPLY, 366 RPL_ENDOFNAMES.
- Time command: 391 RPL_TIME, ERR_NOSUCHSERVER, ERR_NEEDMOREPARAMS.
- PRIVMSG: 401 ERR_NOSUCHNICK, 407 ERR_TOOMANYTARGETS, 412 ERR_NOTEXTTOSEND
- JOIN: ERR_NEEDMOREPARAMS, ERR_NOSUCHCHANNEL, RPL_TOPIC, ERR_TOOMANYCHANNELS
- PART: ERR_NEEDMOREPARAMS, ERR_NOSUCHCHANNEL, ERR_NOTONCHANNEL
- LIST: RPL_LIST, RPL_LISTEND, ERR_NOSUCHSERVER
- NAMES: RPL_NAMREPLY, RPL_ENDOFNAMES, ERR_NOSUCHSERVER
- QUIT: no reply.
- Other: ERR_UNKNOWNCOMMAND, ERR_NOSUCHSERVER : when the parameter is given.

Sample run:

To run the code please follow the instructions in the readme.txt file.

I will run the code on 2 different machines
server IP:

```
oqudeis1@remote00:~/Desktop/p1/v5$ hostname -I
128.226.114.200
oqudeis1@remote00:~/Desktop/p1/v5$
```

Client IP:

```
oqudeis1@remote06:~/Desktop/p1/v5$ make
g++ -pthread client.cpp -o client
oqudeis1@remote06:~/Desktop/p1/v5$ hostname -I
128.226.114.206
oqudeis1@remote06:~/Desktop/p1/v5$ ./client.o
bash: ./client.o: No such file or directory
oqudeis1@remote06:~/Desktop/p1/v5$ ./client
NICK BAHA
USER BAG * * :BAHA QUDIESAT
001 :welcome to Calculus IRC BAHA!BAG@remote00
002 Your host is Arabella, running version 1.0
003 This server was created 2024-03-29 16:57:52
004 Arabella 1.0 * *
```

In the above screenshot I also run NICK and USER commands.

Now I will join 2 channels channel #A and channel #B.

```
JOIN #A,#B
Join confirmed
331 #A : no topic is set
353    RPL_NAMREPLY
@BAHA
 #A :End of NAMES list.Join confirmed
331 #B : no topic is set
353    RPL_NAMREPLY
@BAHA
 #B :End of NAMES list.
```

Now I will change channel #A topic and list the clients inside the channel.

```
TOPIC #A :CATS LOVERS
BAHA set channel topic to CATS LOVERS
LIST #A
353    RPL_NAMREPLY
@BAHA
 #A :End of NAMES list.
```

I will create another 2 users, SAFA and BARA, and I will join channel A from user SAFA.

```
bqudeis1@remote06:~/Desktop/p1/v5$ ./client
NICK BARA
USER BARA * * :BARA IEAD
001 :welcome to Calculus IRC BARA!BARA@remote00
002 Your host is Arabella, running version 1.0
003 This server was created 2024-03-29 16:57:52
004 Arabella 1.0 * *
```

```
bqudeis1@remote06:~/Desktop/p1/v5$ ./client
NICK SAFA
USER SA * * :SAFA AHMAD
001 :welcome to Calculus IRC SAFA!SA@remote00
002 Your host is Arabella, running version 1.0
003 This server was created 2024-03-29 16:57:52
004 Arabella 1.0 * *
JOIN #A
Join confirmed
332 #A: CATS LOVERS
353    RPL_NAMREPLY
@BAHA
@SAFA
 #A :End of NAMES list.
```

Now, send a message to channel #A. The message should reach user SAFA and not reach user BARA.

First screenshot from user baha, and the next From user Bara and Safa:

```
SAFA has joind the chennel. from SAFA
PRIVMSG #A :WELCOME SAFA TO THE CAT LOVERS CHANNEL
```

```
NICK BARA
USER BARA * * :BARA IEAD
001 :welcome to Calculus IRC BARA!BARA@remote00
002 Your host is Arabella, running version 1.0
003 This server was created 2024-03-29 16:57:52
004 Arabella 1.0 * *
```

```
NICK SAFA
USER SA * * :SAFA AHMAD
001 :welcome to Calculus IRC SAFA!SA@remote00
002 Your host is Arabella, running version 1.0
003 This server was created 2024-03-29 16:57:52
004 Arabella 1.0 * *
JOIN #A
Join confirmed
332 #A: CATS LOVERS
353    RPL_NAMREPLY
@BAHA
@SAFA
 #A :End of NAMES list.
:WELCOME SAFA TO THE CAT LOVERS CHANNEL from BAHA
```

As you can see, the message reached only the clients that have joined the channel.

Now lets try the part command from Safa.

```
:WELCOME SAFA TO THE CAT LOVERS C
PART #A
You just left the #A Channel
```

Let's use list command to list the current clients of channel #A from client Baha.

```
SAFA left the channel.
LIST #A
353    RPL_NAMREPLY
@BAHA
 #A :End of NAMES list.
```

Let's try PRIVMSG to BARA client from BAHA client.

```
PRIVMSG BARA :hi Bara, would you like to come to channel #A??
```

Screenshot form bara user:

```
03 This server was created 2024-03-29 16:57:52
04 Arabella 1.0 * *
hi Bara, would you like to come to channel #A??
```

Now let's try the time command:

```
TIME Arabella
Arabella :Fri Mar 29 18:00:16 2024
```

Now lets try to register user baha again.

```
USER BAHA * * :BA
462    :Unauthorized command (already registered)
```

Lets try to change nickname baha to Safa (which is already taken), then lets try nick with no arguments and with space.

```
NICK SAFA
433    :Nickname is already in use
```

```
NICK
431    :No nickname given
NICK
431    :No nickname given
```

Now let's set, clear and view the topic for channel #A.

```
 #A :End of NAMES list.
TOPIC #A :CAT LOVERS
BAHA set channel topic to CAT LOVERS
TOPIC #A :
BAHA clear channel topic.
TOPIC #A
331 #A : no topic is set
```

trying join with special parameter zero:

```
JOIN 0
You just left all Channels!
```

```
QUIT
SIGNING OUT...
bqudeis1@remote06:~/Desktop/p1/v5$
```

Issues encountered:

1) multiple communication between multiple clients and the server, and it is fully solved.
2) The names command did not work as expected so I deleted it.
3) There was a bug in "join 0" but I fixed it, and it seems to work fine now.
4) The documentation of RFC is very hard to understand and very unclear.
5) There is a bug in the List command as in the screenshots above now is fixed.

I have done lots of error handling in my code: some mentions.

```
server_socket = socket(AF_INET, SOCK_STREAM, 0);
if (server_socket == -1)
{
    throw runtime_error("couldnot create socket");
}
```

```
if (bind(server_socket, (sockaddr *)&address, sizeof(address)) == -1)
{
    throw runtime_error("couldnot bind");
}
```

```
if (listen(server_socket, SOMAXCONN) == -1)
{
    throw runtime_error("error cant listen");
}
```

```
if (client_socket == -1)
{
    throw runtime_error("could not accept connection request");
}
```

Handling multiple registrations in the same time

```
clients_mutex.lock();
if (!ClientManager::isNickNameAvialble(params[1]))
    sendToClient(433);
else if (!ClientManager::isValidNickName(params[1]))
    sendToClient(432, params);
else
{
    this->nickName = params[1];
}
clients_mutex.unlock();
```

Done by: Baha Qudiesat.

Email: bqudies1@binghamton.edu

Project two report

Changes done to commands:

There have been some changes from project 1 in order for project 2 to work. Some are specified in RFC, some are specified by the professor, and some are specified by me, those which are specified by me I will mention them.

Pass command (RFC 2812): we need to implement the pass command as specified in the RFC.
for our project, the client registration process will be different than project 1. The registration process will be as follows:

- PASS <PASSWORD>
- NICK < NICKNAME >
- USER <USERNAME> * * :<REALNAME>

Commands that take server's nickname as parameter:

Such as: TIME, SERVER.
we will change the server nickname parameter to server IP address.

NICK command can be used for login.

You can login to your account after logging out (either by closing the terminal or by quit command) using the nick command.

- NICK < NICKNAME > : <PASSWORD>

PASS <password> <server IP> will be used to initiate the password for the connection between servers.

SERVER <SERVER IP>: the server command will be used to initiate the connection between 2 servers.

In my implementation it will produce server message as specified in the RFC, the nicknames known in the server, as well as NJOIN message with the list of channels for that server and the list of users for each channel. That is if the connection established successfully.

Also the server will get a reply back with known nicknames to the second server (and their mapping). And The Njoin message.

Join message: the join command will be sent from client to server only, but for global channels (not required but I did it if it can give me extra points) we will need the other servers to know where these channels are or at least know the channel names and who is listening on those channels. In my implementation I chose the second way to implement it with list of channels and their users updated for each server.

SQUIT: breaks the link between 2 servers. When a client is removed as the result of a SQUIT message, the server SHOULD add the nickname to the list of temporarily unavailable nicknames to prevent future nickname collisions if we reconnect the servers "specified in the RFC page 13".

QUIT message is not required. We don’t have to implement how to handle messages for away users.

Errors codes handled

- Pass command: ERR_NEEDMOREPARAMS ERR_ALREADYREGISTRED: password can’t change.
- User command: 001 RPL_WELCOME, 002 RPL_YOURHOST, 003 RPL_CREATED and 004 RPL_MYINFO upon successful registration of the client.
- NICK command: ERR_NONICKNAMEGIVEN, ERR_ERRONEUSNICKNAME: nickname should not start with reserved characters that has meaning. Ex: ‘&’ , ‘#’ …, ERR_NICKNAMEINUSE.
- List command:322 RPL_LIST, 323 RPL_LISTEND.
- TOPIC command: 331 RPL_NOTOPIC, 332 RPL_TOPIC.
- Names command: 353 RPL_NAMREPLY, 366 RPL_ENDOFNAMES.
- Time command: 391 RPL_TIME, ERR_NOSUCHSERVER, ERR_NEEDMOREPARAMS.
- PRIVMSG: 401 ERR_NOSUCHNICK, 407 ERR_TOOMANYTARGETS, 412 ERR_NOTEXTTOSEND
- JOIN: ERR_NEEDMOREPARAMS, ERR_NOSUCHCHANNEL, RPL_TOPIC, ERR_TOOMANYCHANNELS
- PART: ERR_NEEDMOREPARAMS, ERR_NOSUCHCHANNEL, ERR_NOTONCHANNEL
- LIST: RPL_LIST, RPL_LISTEND, ERR_NOSUCHSERVER
- NAMES: RPL_NAMREPLY, RPL_ENDOFNAMES, ERR_NOSUCHSERVER
- QUIT: no reply.
- Other: ERR_UNKNOWNCOMMAND, ERR_NOSUCHSERVER : when the parameter is given.

Testing most cases

I will try to include all test cases for this project. The sample run will be in the next section of the code.

For the figures bellow:

- Rhombus (diamonds) will reassemble the servers.
- Squares will reassemble the channels.
- Circles will be the users

Registering using pass->nick->user

```
PASS a
NICK cat
USER userName * * :Zyara
001 :welcome to Calculus IRC cat!userName@ibrahim-HP-Pavilion-Laptop-15t-eg200
002 Your host is server1, running version 1.0
003 This server was created 2024-04-28 19:47:40
004 server1 1.0 * *
```

As you can see, user have successfully registered using the three commands.

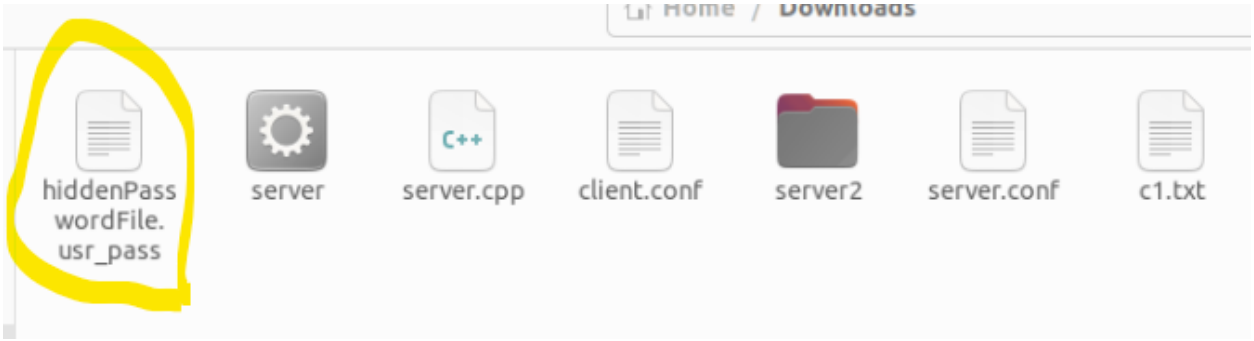
Now we will use the quit command to sign out and exit:

```
QUIT
SIGNING OUT...
```

Now we will use the nick command to sign in again:

```
NICK cat :a
welcome back!
```

How nickname and password stored in the .usr_pass file:

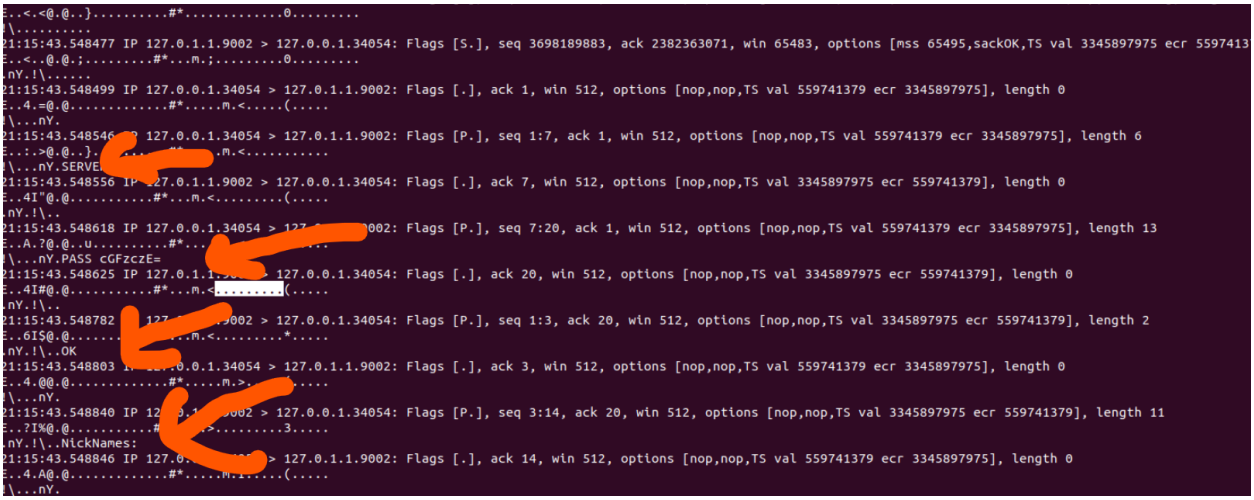


When password = pass. The password after encoding is as follows:

```
1 cat cGFzcw==
```

To connect 2 servers. First run the 2 servers then use pass server combination. (before that use tcpdump to capture the messages between servers)

```
PASS pass1
SERVER 127.0.1.1
```



As you can see, the server will send the pass message plus the server message. Then server 1 will send the nicknames message and njoin message. After server 2 receive these messages. It will also receive an ok message representing the end of the list of messages from server 1. Then it is server 2 turn to send his nicknames to server 1 plus the njoin message.

Logistics:

Runtime arguments through config files: done. Working 100%

Report + readme: both done.

Makefile: done

Coding style: done for entire project, plus used google formatting tool.

Basic implementation:

P1 fully applied + client pass command: done

Server commands: all done.

Channel operations: (njoin,join) : njoin done, join is not done

Authentication and password management are done

Reply codes implementation.

Concurrent connection registration: done.

Concurrent channel operation: not done for server communication, done for the same server only.

Privmsg between clients in different servers: not done.

Group communication through channels: not done.

RFC grammar: followed RFC grammar for only implemented commands.

Correct use of reply codes + server response format: done for most of commands as specified above.

Error handling: the project is buggy as whole. Some functionality are unimplemented.