Queensland University of Technology

Process Management and Distributed Computing

Systems Programming
CAB403

Group 84

Kevin Duong
N9934731

David Thai
N9994653

Semester 2 2019
Due Date
27th October 2019

Statement of Completeness

A statement of completeness indicating the tasks attempted, any deviations from the assignment specification, and problems and deficiencies in the solution.

Task 1

Criteria	Status
Server command line parameter – configurable port & default port	Complete
Server exits gracefully upon receiving SIGNAL (ctrl + c)	Complete
Client command line parameters	Complete
Implementation of SUB <channelid>	Complete
Implementation of CHANNELS	Complete
Implementation of UNSUB <channelid>	Complete
Implementation of SEND <channelid> <message>	Complete
Implementation of BYE	Complete
Implementation of NEXT <channelid> and NEXT	Complete
Implementation of LIVEFEED <channelid> and LIVEFEED	Complete
Description of data structures in the report	Complete

Task 2

Criteria	Status
Multi-process implementation	Incomplete
Process synchronization	Incomplete
Description of how the critical-section problem is handled in the report	Incomplete

Task 3

Criteria	Status
Thread usage	Incomplete
Thread synchronization	Incomplete
Description of how the threads are created and managed in the report	Incomplete

Team Members & Contribution

Information about the team, including student names and student numbers, and statement of each student's contributions to the assignment.

Team Information

Group Number 84	
Team Member	**Student ID**
David Thai	N9994653
Kevin Duong	N9934731

Individual Contributions

Task 1

Criteria	Contribution
Server command line parameter – configurable port & default port	Kevin Duong
Server exits gracefully upon receiving SIGNAL (ctrl + c)	Kevin Duong
Client command line parameters	Kevin Duong
Implementation of SUB <channelid>	David Thai
Implementation of CHANNELS	David Thai
Implementation of UNSUB <channelid>	David Thai
Implementation of SEND <channelid> <message>	David Thai
Implementation of BYE	Kevin Duong
Implementation of NEXT <channelid> and NEXT	David Thai
Implementation of LIVEFEED <channelid> and LIVEFEED	David Thai

Task 2

Criteria	Contribution
Multi-process implementation	
Process synchronization	

Task 3

Criteria	Contribution
Thread usage	
Thread synchronization	

Report

Criteria	Contribution
Description of data structures in the report	Kevin Duong
Description of how the critical-section problem is handled in the report	
Description of how the threads are created and managed in the report	
Instructions on how to compile and run the program	Kevin Duong

Data Structure (Page 3)

Description of the implemented data structure(s).

Creating and Managing Fork

Description of how the forking is created and managed in Task 2, if applicable.

Critical Problem/s

Description of how the critical-section problem is handled in Task 2, if applicable.

Creating and Managing Threads

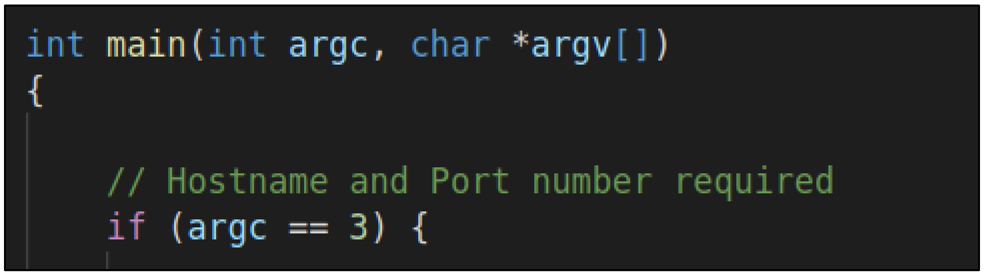
Description of how the threading is created and managed in Task 3, if applicable.

Data Structure

Client / Server Command Configuration

Client Line Parameters

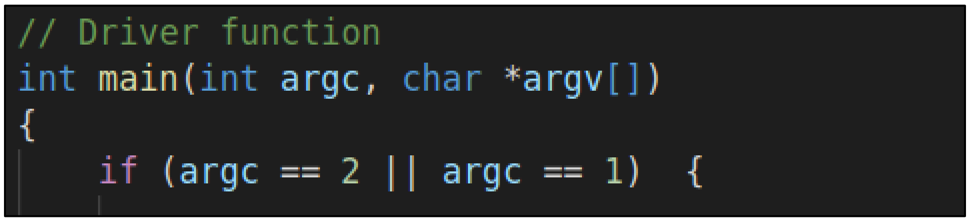
The main function in the client code contains 'argv' and 'argc' which are arguments passed in C. The client requires the server's hostname (IP address) and the port number, adding two arguments to the line which makes argc the value of 3. If the requirements are met, then the client can progress and connect to the server.



```c
int main(int argc, char *argv[])
{

    // Hostname and Port number required
    if (argc == 3) {
```

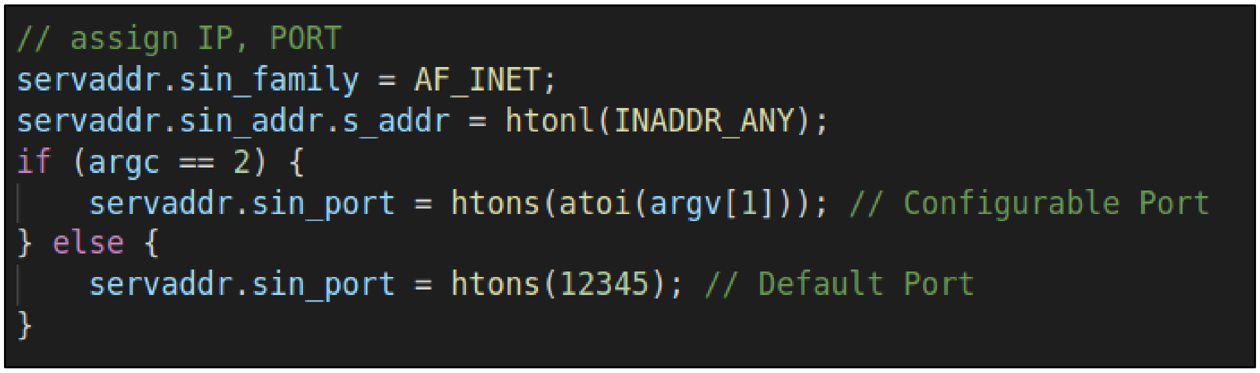
Server Line Parameters

In the main function for server, the choice of using a default port number or a custom port number can be seen with the conditional statement's logical operators. With both argument numbers being acceptable, the command line for example can be written as *'./server 3000'* or *'./server'.*



```c
// Driver function
int main(int argc, char *argv[])
{
    if (argc == 2 || argc == 1)  {
```

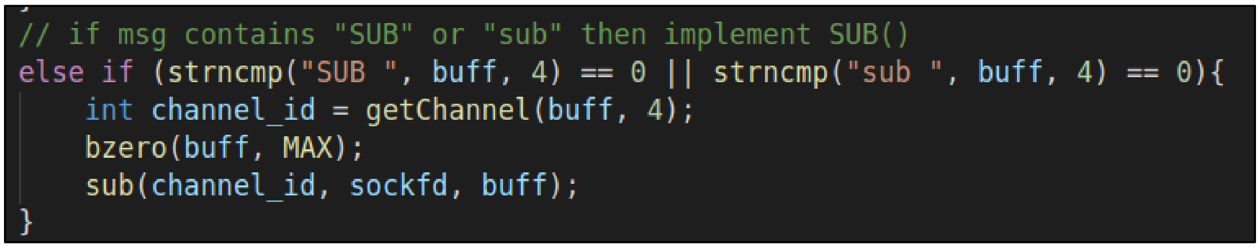
As to how the ports are configured, another conditional statement is established upon configuring the port number. If the server is executed with no parameters, a default port number of '12345' is assigned to the server, else the argument is used and converted to an integer.



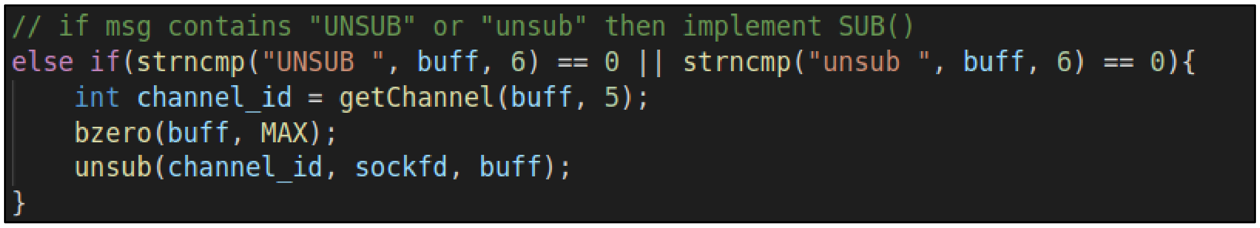
```c
// assign IP, PORT
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
if (argc == 2) {
    servaddr.sin_port = htons(atoi(argv[1])); // Configurable Port
} else {
    servaddr.sin_port = htons(12345); // Default Port
}
```

Implementation of SUB / UNSUB Channels

The user is allowed to subscribe to a channel by inserting the line ‘sub’ along with a channel ID. Likewise, this applies to unsubscribing by sending ‘unsub’ and a channel ID. Inserting both kind of strings will pass the integer to their designated functions that process the task.



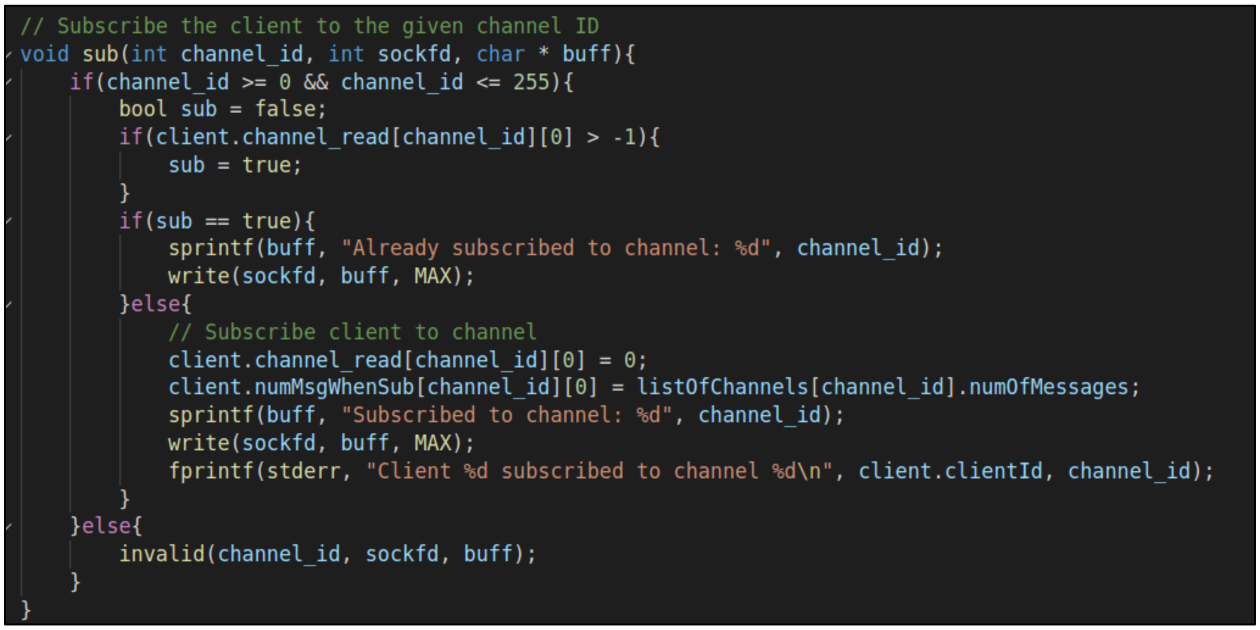
```
// if msg contains "SUB" or "sub" then implement SUB()
else if (strncmp("SUB ", buff, 4) == 0 || strncmp("sub ", buff, 4) == 0){
    int channel_id = getChannel(buff, 4);
    bzero(buff, MAX);
    sub(channel_id, sockfd, buff);
}
```



```
// if msg contains "UNSUB" or "unsub" then implement SUB()
else if(strncmp("UNSUB ", buff, 6) == 0 || strncmp("unsub ", buff, 6) == 0){
    int channel_id = getChannel(buff, 5);
    bzero(buff, MAX);
    unsub(channel_id, sockfd, buff);
}
```

Subscribing a Channel: SUB <*channelid*>

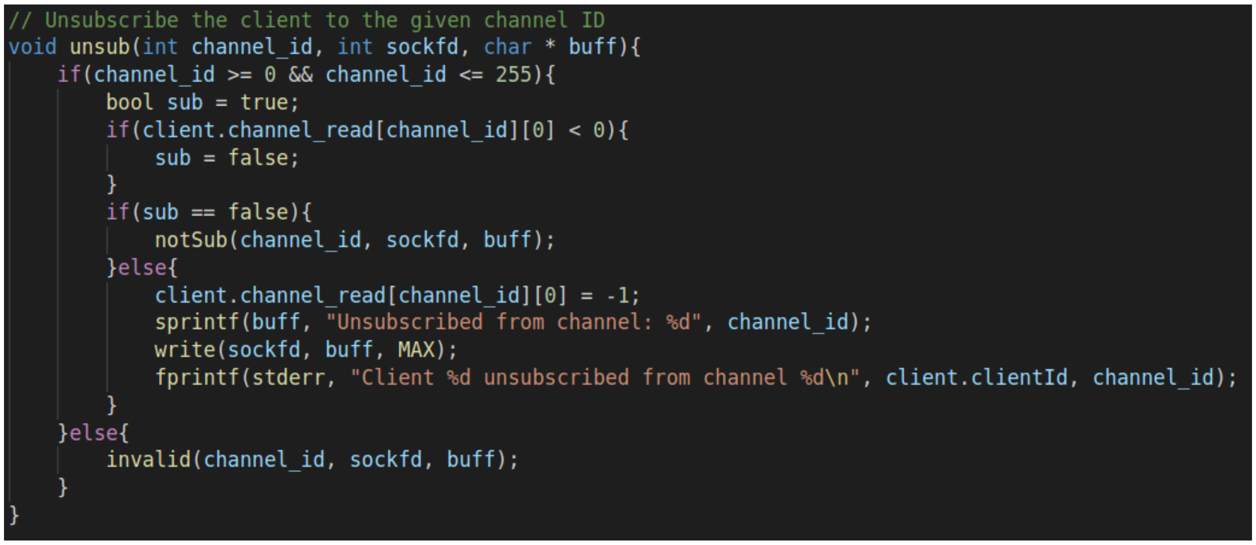
The process of subscribing a channel can be seen below in the code snippet. It takes the arguments of the channel ID, the client’s socket and the current buffer. Choosing an ID is determined by the limited range from 0 to 255; if the ID selected is not within the range, then the function will call another function that handles invalid channel IDs. If the user has chosen to subscribe the same channel ID, then the function will tell the user that it already has done so, indicated by the Boolean condition of sub == true. For a normal process, the function will store the channel ID to the client and return messages to both the user and server indicating the subscription is successful.



```
// Subscribe the client to the given channel ID
void sub(int channel_id, int sockfd, char * buff){
    if(channel_id >= 0 && channel_id <= 255){
        bool sub = false;
        if(client.channel_read[channel_id][0] > -1){
            sub = true;
        }
        if(sub == true){
            sprintf(buff, "Already subscribed to channel: %d", channel_id);
            write(sockfd, buff, MAX);
        }else{
            // Subscribe client to channel
            client.channel_read[channel_id][0] = 0;
            client.numMsgWhenSub[channel_id][0] = listOfChannels[channel_id].numOfMessages;
            sprintf(buff, "Subscribed to channel: %d", channel_id);
            write(sockfd, buff, MAX);
            fprintf(stderr, "Client %d subscribed to channel %d\n", client.clientId, channel_id);
        }
    }else{
        invalid(channel_id, sockfd, buff);
    }
}
```

Unsubscribing a Channel: UNSUB <*channelid*>

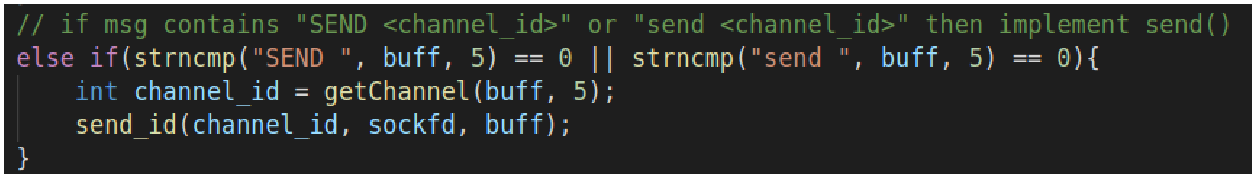
When unsubscribing a channel ID, it checks if the client has subscribed to the channel or not. Not subscribing the channel yet unsubscribing so will call the function ‘notSub’ to tell that the user hasn’t subscribed in the first place. Otherwise, the client’s channel ID number will be removed by changing the value to -1 and pushing the message to the client and server saying the user has unsubscribed to a channel.



```c
// Unsubscribe the client to the given channel ID
void unsub(int channel_id, int sockfd, char * buff){
    if(channel_id >= 0 && channel_id <= 255){
        bool sub = true;
        if(client.channel_read[channel_id][0] < 0){
            sub = false;
        }
        if(sub == false){
            notSub(channel_id, sockfd, buff);
        }else{
            client.channel_read[channel_id][0] = -1;
            sprintf(buff, "Unsubscribed from channel: %d", channel_id);
            write(sockfd, buff, MAX);
            fprintf(stderr, "Client %d unsubscribed from channel %d\n", client.clientId, channel_id);
        }
    }else{
        invalid(channel_id, sockfd, buff);
    }
}
```

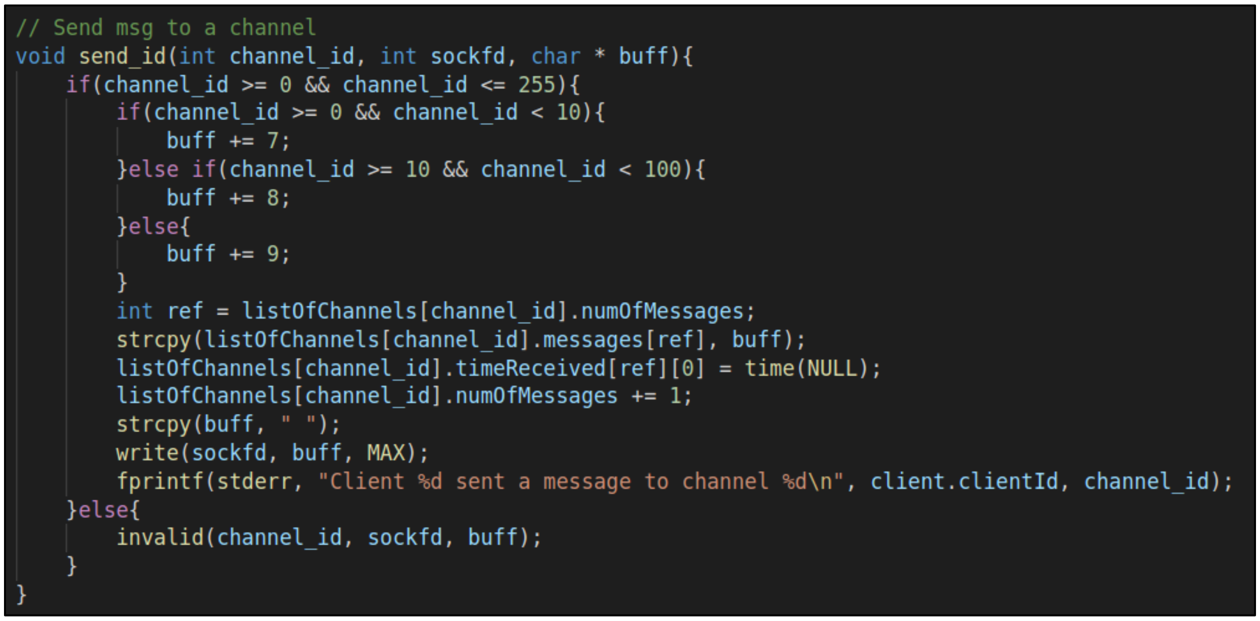
Implementation of SEND

To send a message to a channel, a couple of parameters are needed to enter on the line such as the channel ID and the message string. These are called with the ‘send’ command.



```c
// if msg contains "SEND <channel_id>" or "send <channel_id>" then implement send()
else if(strncmp("SEND ", buff, 5) == 0 || strncmp("send ", buff, 5) == 0){
    int channel_id = getChannel(buff, 5);
    send_id(channel_id, sockfd, buff);
}
```

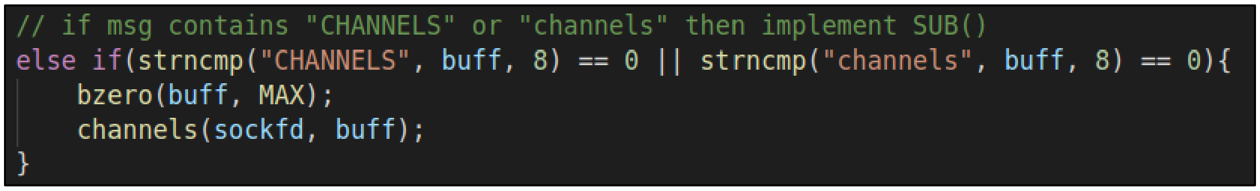
Within the send function, the channel ID is first verified to be able to send to the right channel. Buffer handling is done so to stabilize the message being sent, based on the integer size of the channel seen in the conditional statements of the channel ID number ranges. Overall, the message buffer is relayed to the designated channel and a successful message will be sent to the client.



```c
// Send msg to a channel
void send_id(int channel_id, int sockfd, char * buff){
    if(channel_id >= 0 && channel_id <= 255){
        if(channel_id >= 0 && channel_id < 10){
            buff += 7;
        }else if(channel_id >= 10 && channel_id < 100){
            buff += 8;
        }else{
            buff += 9;
        }
        int ref = listOfChannels[channel_id].numOfMessages;
        strcpy(listOfChannels[channel_id].messages[ref], buff);
        listOfChannels[channel_id].timeReceived[ref][0] = time(NULL);
        listOfChannels[channel_id].numOfMessages += 1;
        strcpy(buff, " ");
        write(sockfd, buff, MAX);
        fprintf(stderr, "Client %d sent a message to channel %d\n", client.clientId, channel_id);
    }else{
        invalid(channel_id, sockfd, buff);
    }
}
```

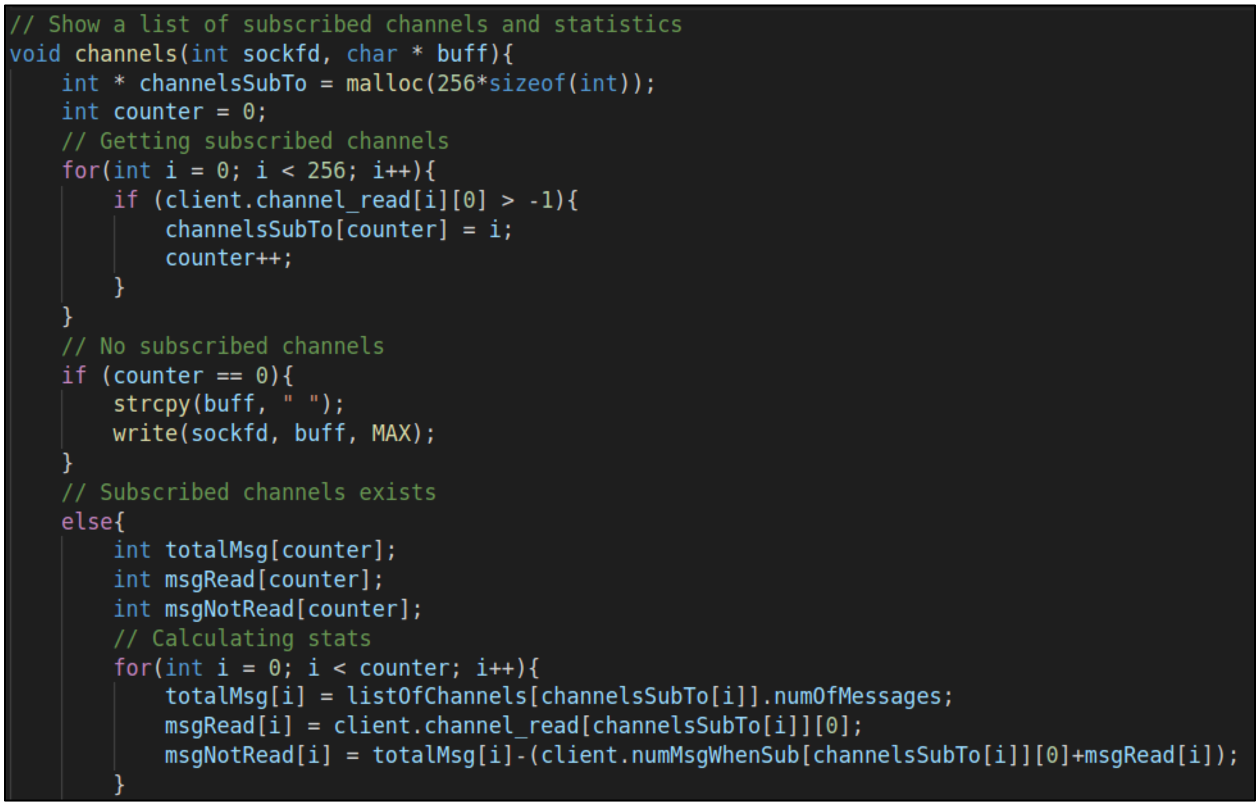
Implementation of CHANNELS

If the user has subscribed to a bunch of channels, they can view these channels altogether to see how many messages have been sent to the channels, how many they have seen, and how many they haven't seen. This can be done by entering the line 'channels'.

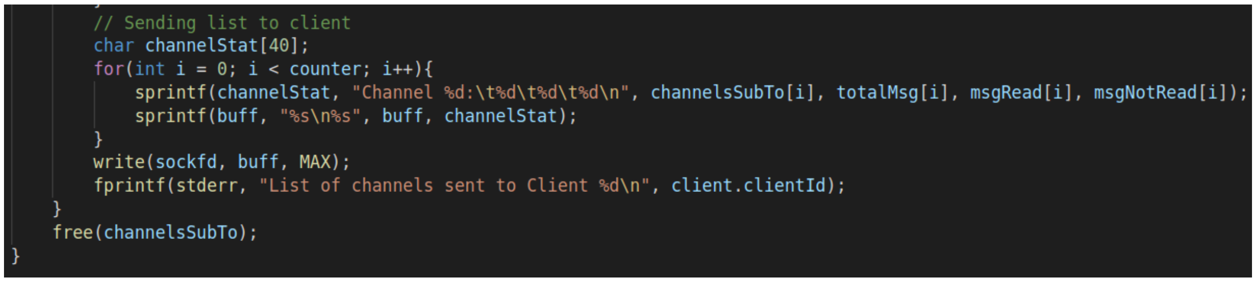


```c
// if msg contains "CHANNELS" or "channels" then implement SUB()
else if(strncmp("CHANNELS", buff, 8) == 0 || strncmp("channels", buff, 8) == 0){
    bzero(buff, MAX);
    channels(sockfd, buff);
}
```

The function 'channels' will show a list of subscribed channels and statistics as intended. If the user hasn't subscribed to any channels, then this function will output nothing to the terminal. Aside from this, it fetches all the channel IDs that have been subscribed by the user and sends them to retrieve their messages. It is done so by running a *for loop* to tabulate a row of channels and their message statistics in numbers, for each column represents the total messages stored in the channel, messages read, and messages not read. Then a message is relayed to the server saying the list of channels has been sent to the client.



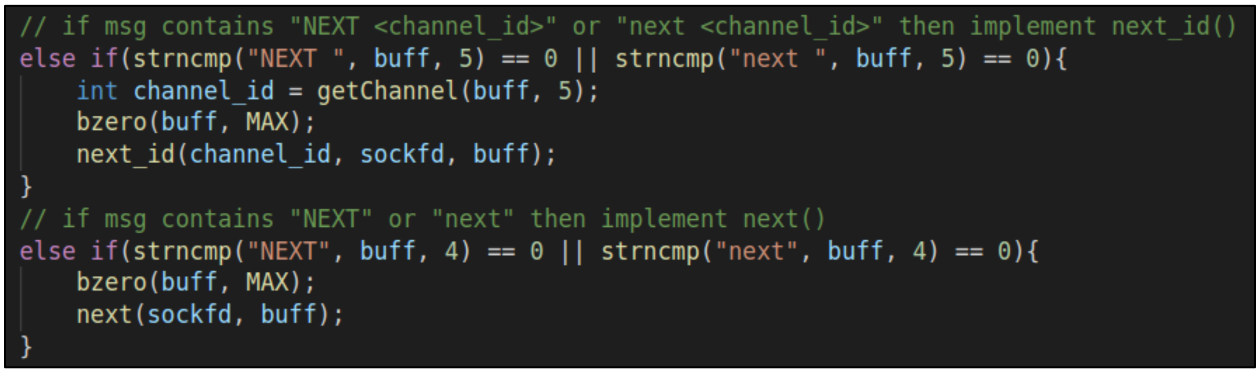
```c
// Show a list of subscribed channels and statistics
void channels(int sockfd, char * buff){
    int * channelsSubTo = malloc(256*sizeof(int));
    int counter = 0;
    // Getting subscribed channels
    for(int i = 0; i < 256; i++){
        if (client.channel_read[i][0] > -1){
            channelsSubTo[counter] = i;
            counter++;
        }
    }
    // No subscribed channels
    if (counter == 0){
        strcpy(buff, " ");
        write(sockfd, buff, MAX);
    }
    // Subscribed channels exists
    else{
        int totalMsg[counter];
        int msgRead[counter];
        int msgNotRead[counter];
        // Calculating stats
        for(int i = 0; i < counter; i++){
            totalMsg[i] = listOfChannels[channelsSubTo[i]].numOfMessages;
            msgRead[i] = client.channel_read[channelsSubTo[i]][0];
            msgNotRead[i] = totalMsg[i]-(client.numMsgWhenSub[channelsSubTo[i]][0]+msgRead[i]);
        }
```



```c
        // Sending list to client
        char channelStat[40];
        for(int i = 0; i < counter; i++){
            sprintf(channelStat, "Channel %d:\t%d\t%d\t%d\n", channelsSubTo[i], totalMsg[i], msgRead[i], msgNotRead[i]);
            sprintf(buff, "%s\n%s", buff, channelStat);
        }
        write(sockfd, buff, MAX);
        fprintf(stderr, "List of channels sent to Client %d\n", client.clientId);
    }
    free(channelsSubTo);
}
```

Implementation of NEXT

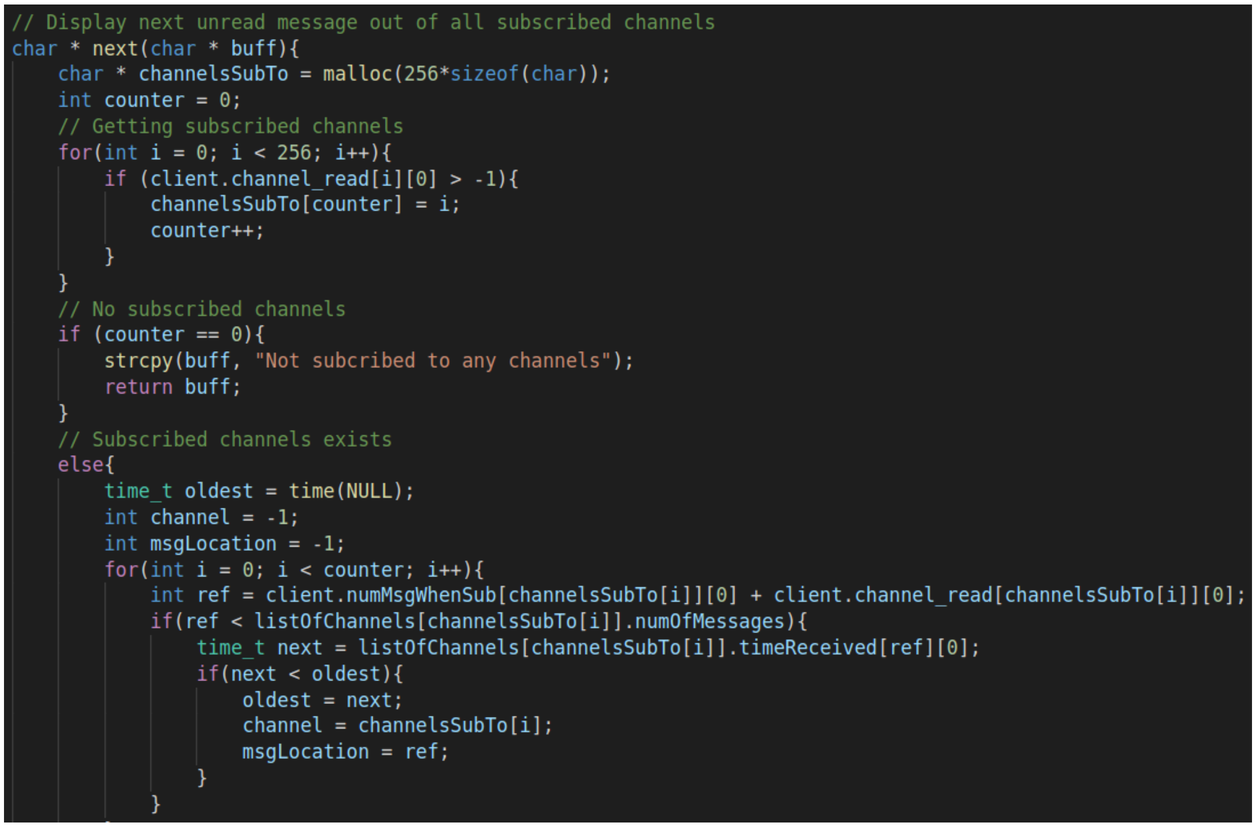
The idea of using 'Next' would allow the user to retrieve an unread message in any channel they've subscribed to. This can be used in two ways: retrieving the messages sent in order using only the line 'Next' or reading unread messages in a specific channel by appending the channel ID to the line after 'Next'.



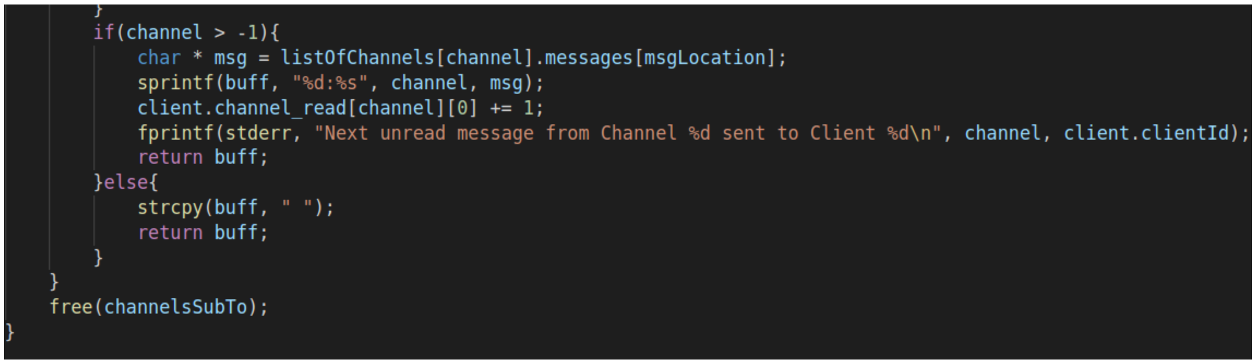
```c
// if msg contains "NEXT <channel_id>" or "next <channel_id>" then implement next_id()
else if(strncmp("NEXT ", buff, 5) == 0 || strncmp("next ", buff, 5) == 0){
    int channel_id = getChannel(buff, 5);
    bzero(buff, MAX);
    next_id(channel_id, sockfd, buff);
}
// if msg contains "NEXT" or "next" then implement next()
else if(strncmp("NEXT", buff, 4) == 0 || strncmp("next", buff, 4) == 0){
    bzero(buff, MAX);
    next(sockfd, buff);
}
```

Read messages: NEXT

If the user only uses ‘Next’, then it’ll be calling the function ‘next_id’ to process the task. This function will look for every subscribed channel to find the oldest message available, and if the user hasn’t subscribed to any channel, then the process ends here. As for a normal process, the oldest message will be sent to the client just below the ‘next’ command, and a message will be sent to the server saying the client has received their unread message.



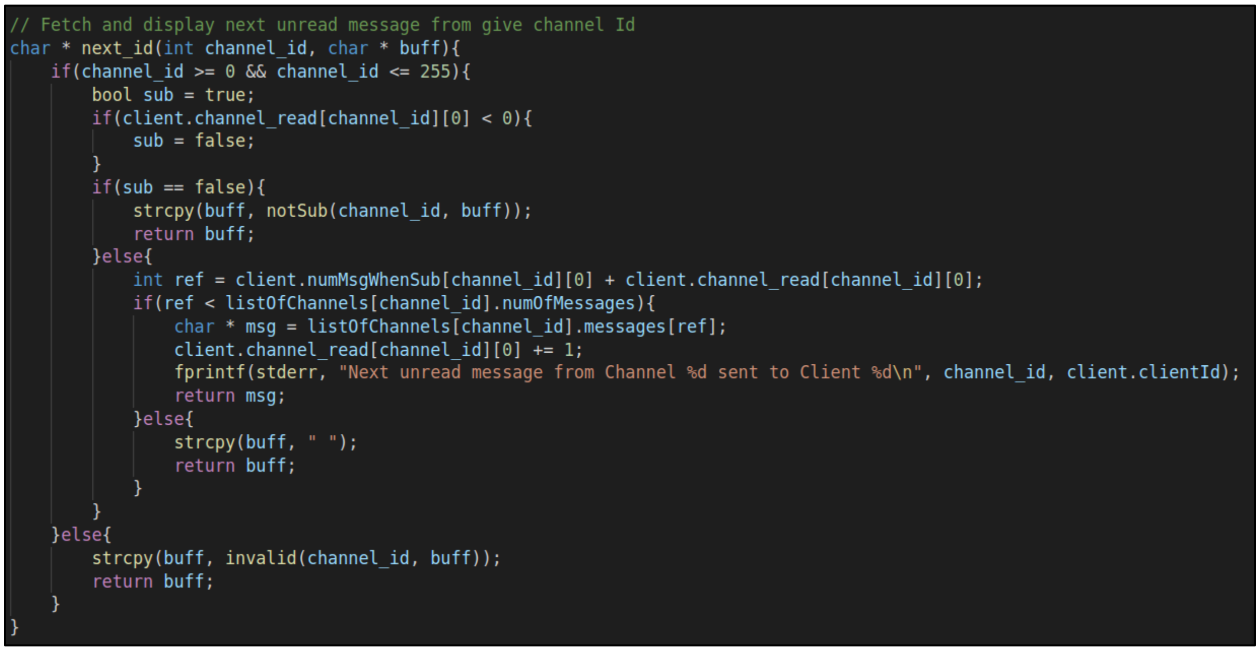
```
// Display next unread message out of all subscribed channels
char * next(char * buff){
    char * channelsSubTo = malloc(256*sizeof(char));
    int counter = 0;
    // Getting subscribed channels
    for(int i = 0; i < 256; i++){
        if (client.channel_read[i][0] > -1){
            channelsSubTo[counter] = i;
            counter++;
        }
    }
    // No subscribed channels
    if (counter == 0){
        strcpy(buff, "Not subcribed to any channels");
        return buff;
    }
    // Subscribed channels exists
    else{
        time_t oldest = time(NULL);
        int channel = -1;
        int msgLocation = -1;
        for(int i = 0; i < counter; i++){
            int ref = client.numMsgWhenSub[channelsSubTo[i]][0] + client.channel_read[channelsSubTo[i]][0];
            if(ref < listOfChannels[channelsSubTo[i]].numOfMessages){
                time_t next = listOfChannels[channelsSubTo[i]].timeReceived[ref][0];
                if(next < oldest){
                    oldest = next;
                    channel = channelsSubTo[i];
                    msgLocation = ref;
                }
            }
```



```
        }
        if(channel > -1){
            char * msg = listOfChannels[channel].messages[msgLocation];
            sprintf(buff, "%d:%s", channel, msg);
            client.channel_read[channel][0] += 1;
            fprintf(stderr, "Next unread message from Channel %d sent to Client %d\n", channel, client.clientId);
            return buff;
        }else{
            strcpy(buff, " ");
            return buff;
        }
    }
    free(channelsSubTo);
}
```

Read channel messages: NEXT <*channelid*>

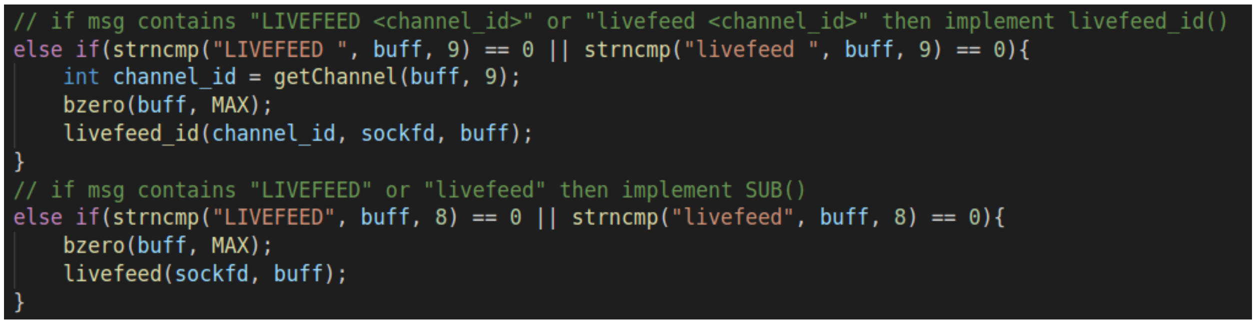
Using the command line of 'Next' followed by a channel number will call the 'next_id' function to complete the task. For this case, the user will retrieve a message from a specific channel which is verified first to find out if the channel exists. Then, the message is extracted from the channel and sent away to the client. It will also ensure that the message has been read by the client so that it can use 'next' again to read another message that is unread. A successful operation will send a message to the server indicating that the message from the specific channel has been sent to the client.



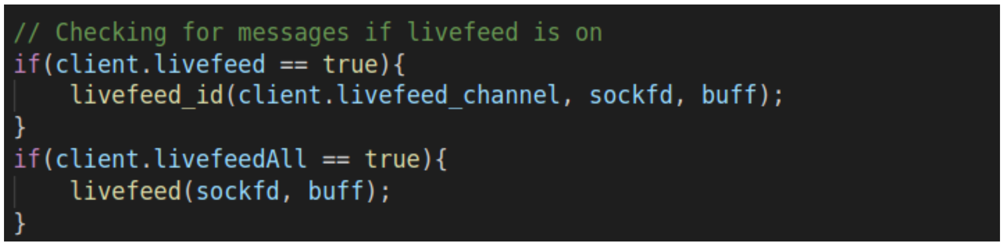
```c
// Fetch and display next unread message from give channel Id
char * next_id(int channel_id, char * buff){
    if(channel_id >= 0 && channel_id <= 255){
        bool sub = true;
        if(client.channel_read[channel_id][0] < 0){
            sub = false;
        }
        if(sub == false){
            strcpy(buff, notSub(channel_id, buff));
            return buff;
        }else{
            int ref = client.numMsgWhenSub[channel_id][0] + client.channel_read[channel_id][0];
            if(ref < listOfChannels[channel_id].numOfMessages){
                char * msg = listOfChannels[channel_id].messages[ref];
                client.channel_read[channel_id][0] += 1;
                fprintf(stderr, "Next unread message from Channel %d sent to Client %d\n", channel_id, client.clientId);
                return msg;
            }else{
                strcpy(buff, " ");
                return buff;
            }
        }
    }else{
        strcpy(buff, invalid(channel_id, buff));
        return buff;
    }
}
```

Implementation of LIVEFEED

Entering 'livefeed' as a command to the client will bring up a continuous stream of messages from all channels. To livestream a specific channel, the user must attach an ID number to the 'livefeed' command.



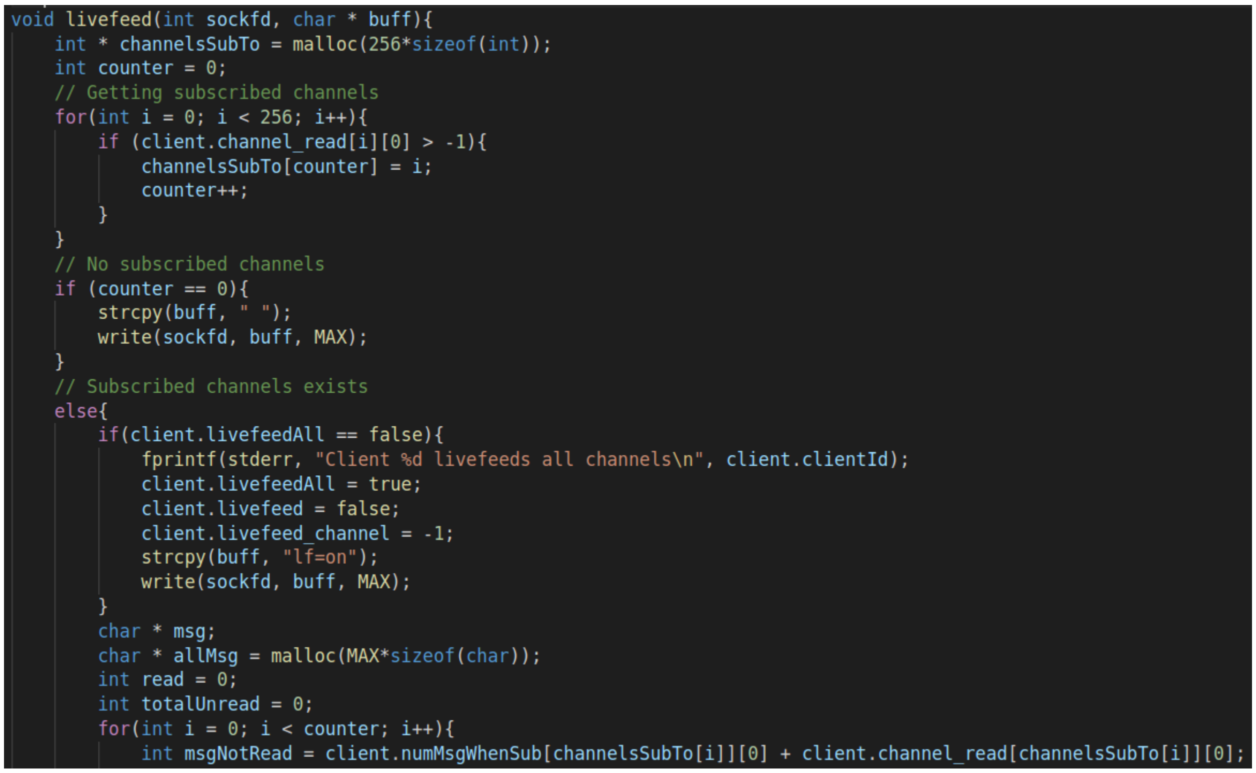
```c
// if msg contains "LIVEFEED <channel_id>" or "livefeed <channel_id>" then implement livefeed_id()
else if(strncmp("LIVEFEED ", buff, 9) == 0 || strncmp("livefeed ", buff, 9) == 0){
    int channel_id = getChannel(buff, 9);
    bzero(buff, MAX);
    livefeed_id(channel_id, sockfd, buff);
}
// if msg contains "LIVEFEED" or "livefeed" then implement SUB()
else if(strncmp("LIVEFEED", buff, 8) == 0 || strncmp("livefeed", buff, 8) == 0){
    bzero(buff, MAX);
    livefeed(sockfd, buff);
}
```



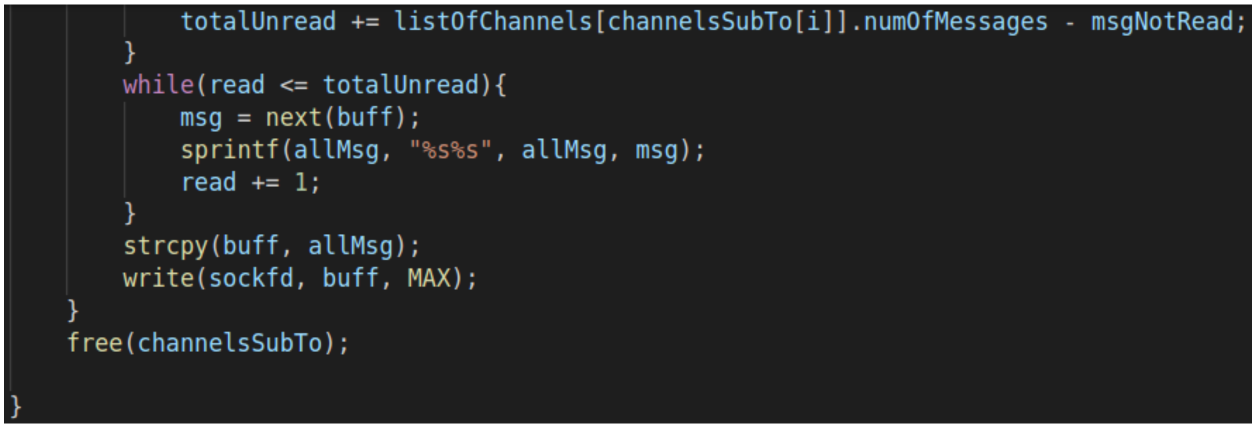
```c
// Checking for messages if livefeed is on
if(client.livefeed == true){
    livefeed_id(client.livefeed_channel, sockfd, buff);
}
if(client.livefeedAll == true){
    livefeed(sockfd, buff);
}
```

Livestream messages: Livefeed

The ‘livefeed’ function handles all subscribed channels unread messages and incoming messages. It checks if any channels are subscribed first from the client before retrieving the messages, otherwise it ends.



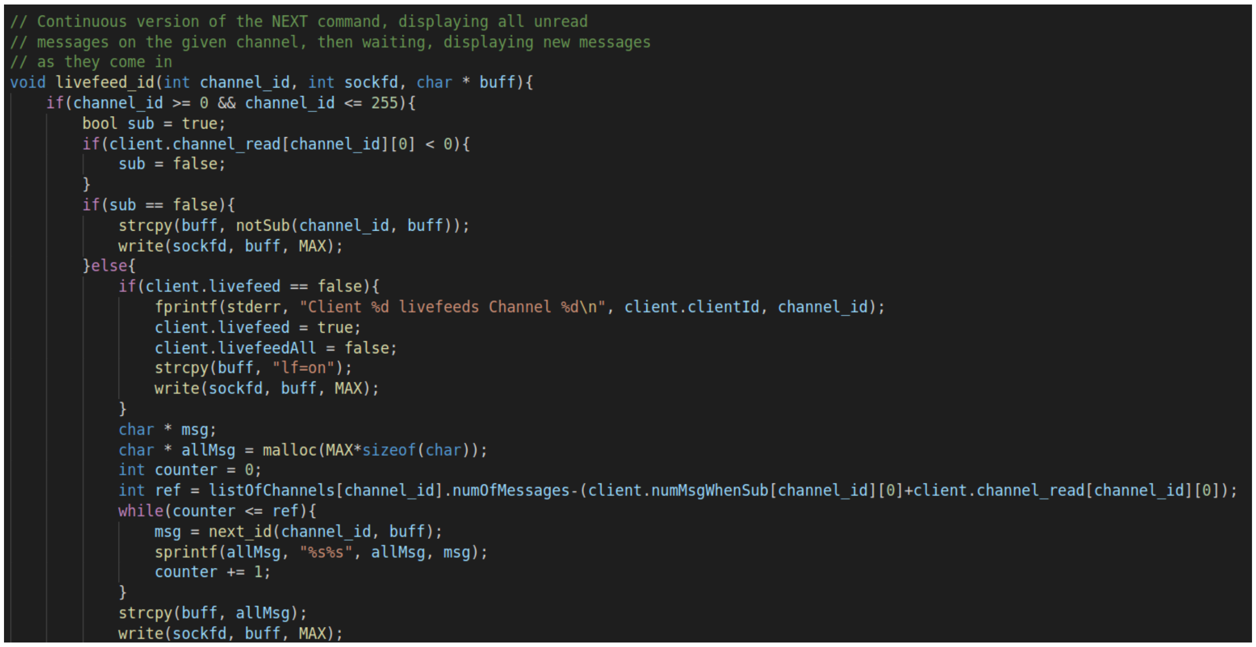
```c
void livefeed(int sockfd, char * buff){
    int * channelsSubTo = malloc(256*sizeof(int));
    int counter = 0;
    // Getting subscribed channels
    for(int i = 0; i < 256; i++){
        if (client.channel_read[i][0] > -1){
            channelsSubTo[counter] = i;
            counter++;
        }
    }
    // No subscribed channels
    if (counter == 0){
        strcpy(buff, " ");
        write(sockfd, buff, MAX);
    }
    // Subscribed channels exists
    else{
        if(client.livefeedAll == false){
            fprintf(stderr, "Client %d livefeeds all channels\n", client.clientId);
            client.livefeedAll = true;
            client.livefeed = false;
            client.livefeed_channel = -1;
            strcpy(buff, "lf=on");
            write(sockfd, buff, MAX);
        }
        char * msg;
        char * allMsg = malloc(MAX*sizeof(char));
        int read = 0;
        int totalUnread = 0;
        for(int i = 0; i < counter; i++){
            int msgNotRead = client.numMsgWhenSub[channelsSubTo[i]][0] + client.channel_read[channelsSubTo[i]][0];
```



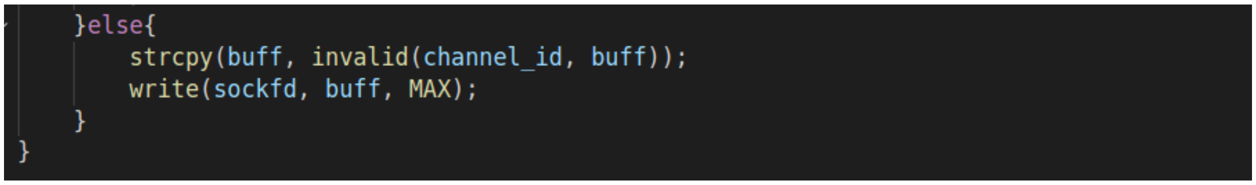
```c
            totalUnread += listOfChannels[channelsSubTo[i]].numOfMessages - msgNotRead;
        }
        while(read <= totalUnread){
            msg = next(buff);
            sprintf(allMsg, "%s%s", allMsg, msg);
            read += 1;
        }
        strcpy(buff, allMsg);
        write(sockfd, buff, MAX);
    }
    free(channelsSubTo);

}
```

Livestream messages: Livefeed <*channelid*>

If the livefeed is called with a channel ID in mind, the function checks if the channel exists and whether the client has subscribed or not. Messages sent to the chosen channel will be automatically sent to the client for the duration of the lifefeed.

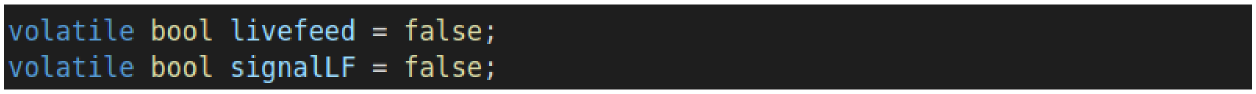


```c
// Continuous version of the NEXT command, displaying all unread
// messages on the given channel, then waiting, displaying new messages
// as they come in
void livefeed_id(int channel_id, int sockfd, char * buff){
    if(channel_id >= 0 && channel_id <= 255){
        bool sub = true;
        if(client.channel_read[channel_id][0] < 0){
            sub = false;
        }
        if(sub == false){
            strcpy(buff, notSub(channel_id, buff));
            write(sockfd, buff, MAX);
        }else{
            if(client.livefeed == false){
                fprintf(stderr, "Client %d livefeeds Channel %d\n", client.clientId, channel_id);
                client.livefeed = true;
                client.livefeedAll = false;
                strcpy(buff, "lf=on");
                write(sockfd, buff, MAX);
            }
            char * msg;
            char * allMsg = malloc(MAX*sizeof(char));
            int counter = 0;
            int ref = listOfChannels[channel_id].numOfMessages-(client.numMsgWhenSub[channel_id][0]+client.channel_read[channel_id][0]);
            while(counter <= ref){
                msg = next_id(channel_id, buff);
                sprintf(allMsg, "%s%s", allMsg, msg);
                counter += 1;
            }
            strcpy(buff, allMsg);
            write(sockfd, buff, MAX);
```

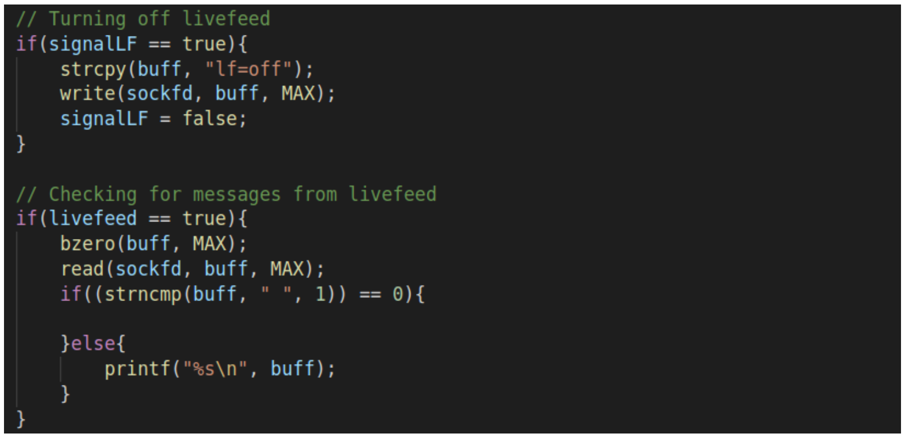


```c
    }else{
        strcpy(buff, invalid(channel_id, buff));
        write(sockfd, buff, MAX);
    }
}
```

In the client, volatile bools are used to determine whether livefeed is turned off or checking messages is a go if the livefeed is on.



```c
volatile bool livefeed = false;
volatile bool signalLF = false;
```



```c
// Turning off livefeed
if(signalLF == true){
    strcpy(buff, "lf=off");
    write(sockfd, buff, MAX);
    signalLF = false;
}

// Checking for messages from livefeed
if(livefeed == true){
    bzero(buff, MAX);
    read(sockfd, buff, MAX);
    if((strncmp(buff, " ", 1)) == 0){

    }else{
        printf("%s\n", buff);
    }
}
```

Signal Handling and BYE

Cases are made for using *'ctrl + c'* in the program, where it is allowed or not allowed to be used in both the server and client program. The program will terminate when using the shortcut key in the server, which will recognize the activity and prompt the message: 'Server exiting…'. This is using the signal function in the server's main function to add the inputs of SIGINT and a function to display the message and terminate the server.

```
// Server exits gracefully upon receiving SIGNAL (ctrl + c)
void server_exit() {

    printf("\nServer exiting...\n");
    exit(1);
}
```

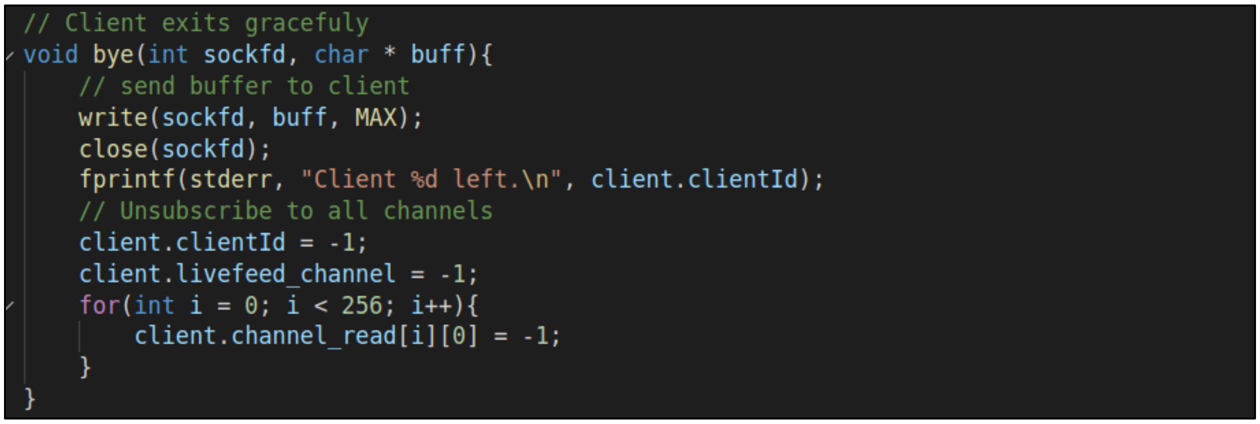
```
signal(SIGINT, server_exit);
```

In the client, the user will not be able to leave immediately by pressing *'ctrl + c'*. Doing so will ask the user to use the command line 'BYE' or 'bye' unless livefeed is turned on. The signal rejects the user from terminating the program.

```
void handler(int num) {
    if(livefeed == true){
        livefeed = false;
        printf("\nEnding Livefeed.\n");
        signalLF = true;
    }else{
        write(STDOUT_FILENO, "\nPlease enter 'BYE' or 'bye' to exit.\n", 39);
    }
}
```

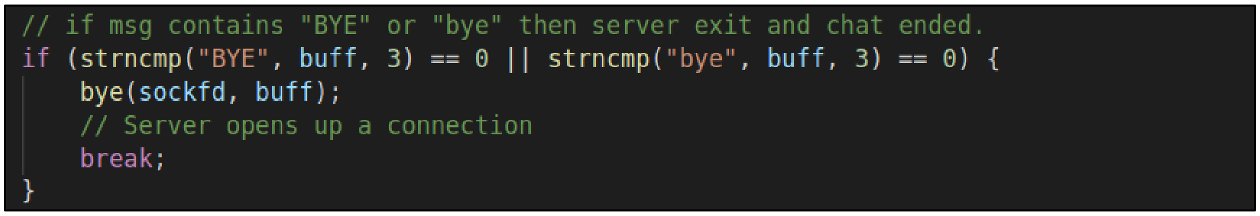
```
signal(SIGINT, handler);
```

An alternative way of exiting the program as the client is to enter the line 'bye', which terminates the client program as a result. First, it unsubscribes all the client ID's channels in the server, meaning that when the client enters again with the same ID, it will no longer have its subscriptions as before.

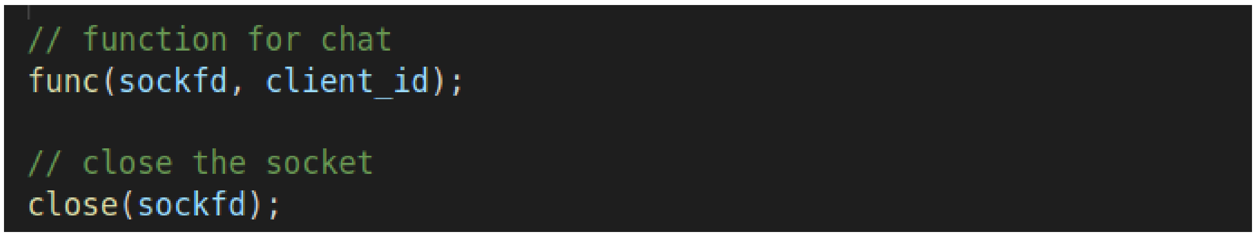


```c
// Client exits gracefuly
void bye(int sockfd, char * buff){
    // send buffer to client
    write(sockfd, buff, MAX);
    close(sockfd);
    fprintf(stderr, "Client %d left.\n", client.clientId);
    // Unsubscribe to all channels
    client.clientId = -1;
    client.livefeed_channel = -1;
    for(int i = 0; i < 256; i++){
        client.channel_read[i][0] = -1;
    }
}
```

Then, the loop function to connect to the server will break within the client script and its socket closed, with the same applying to the server.

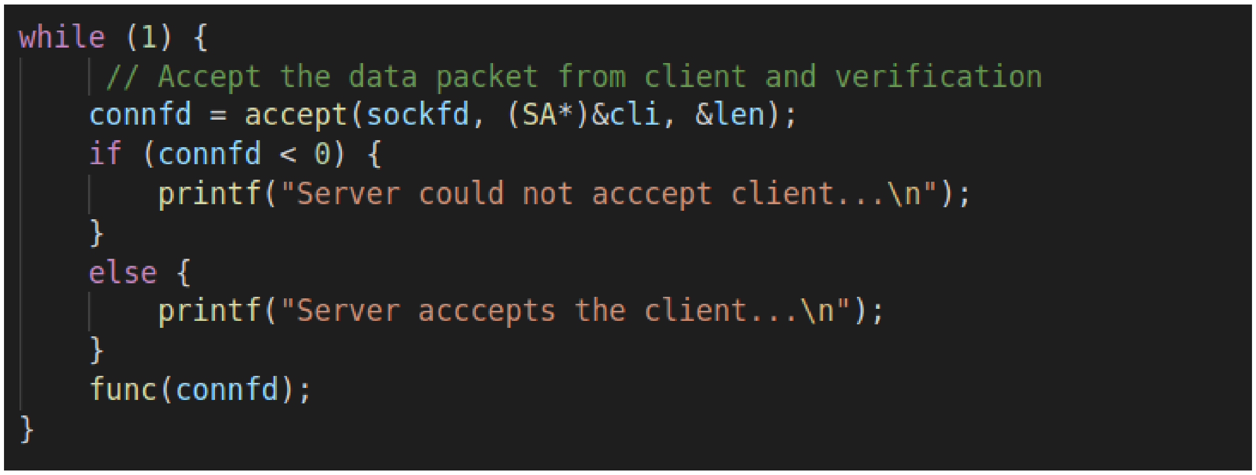


```c
// if msg contains "BYE" or "bye" then server exit and chat ended.
if (strncmp("BYE", buff, 3) == 0 || strncmp("bye", buff, 3) == 0) {
    bye(sockfd, buff);
    // Server opens up a connection
    break;
}
```



```c
// function for chat
func(sockfd, client_id);

// close the socket
close(sockfd);
```

The server on the other hand will remain active, listening to a new client after breaking the loop in the connection.



```c
while (1) {
    // Accept the data packet from client and verification
    connfd = accept(sockfd, (SA*)&cli, &len);
    if (connfd < 0) {
        printf("Server could not acccept client...\n");
    }
    else {
        printf("Server acccepts the client...\n");
    }
    func(connfd);
}
```

Compiling and Running the Program

Specification Requirements

The development of the client/server system was created in a Linux operated environment using Ubuntu, hence, it is recommended to acknowledge the requirements to be able to compile and run the program.

Operating System

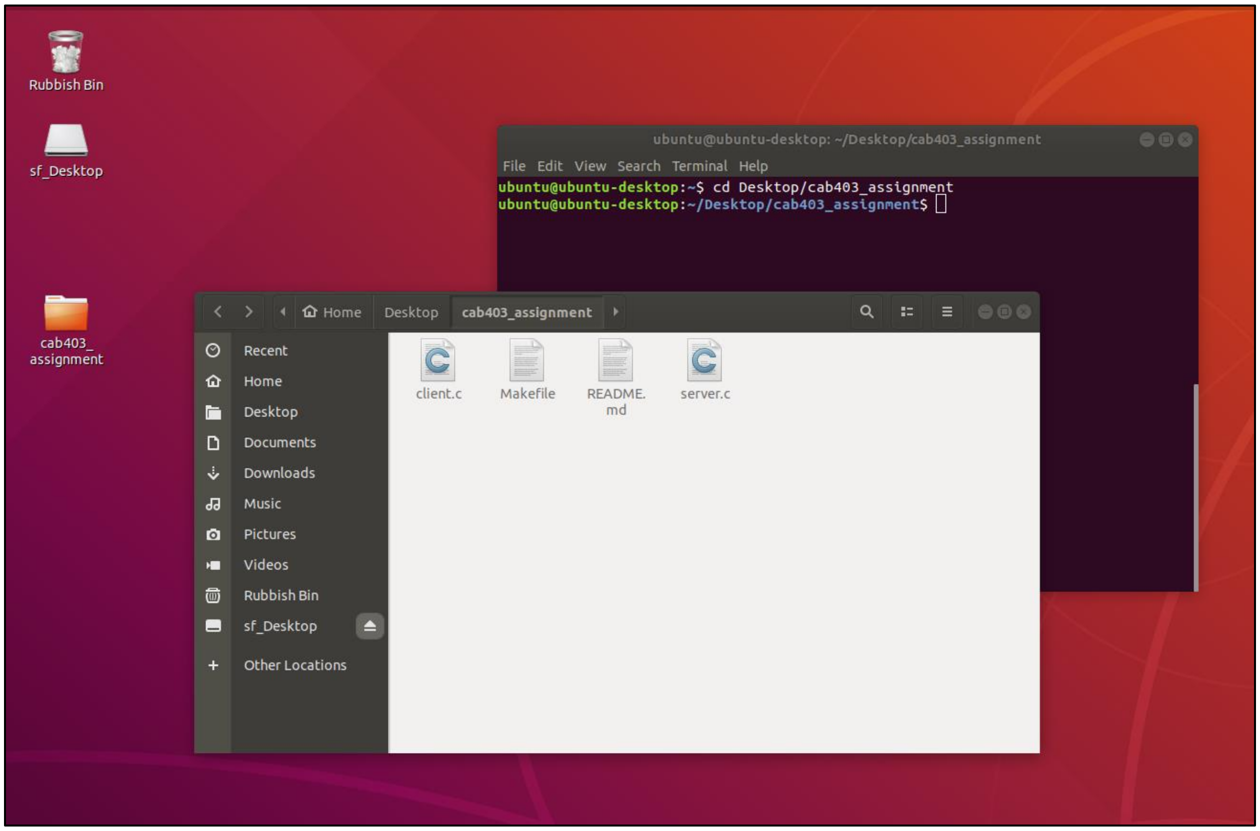
Ubuntu 18.04.3 LTS
https://ubuntu.com/download/desktop

Virtual Machine

VirtualBox
https://www.virtualbox.org/

Compiling the Program

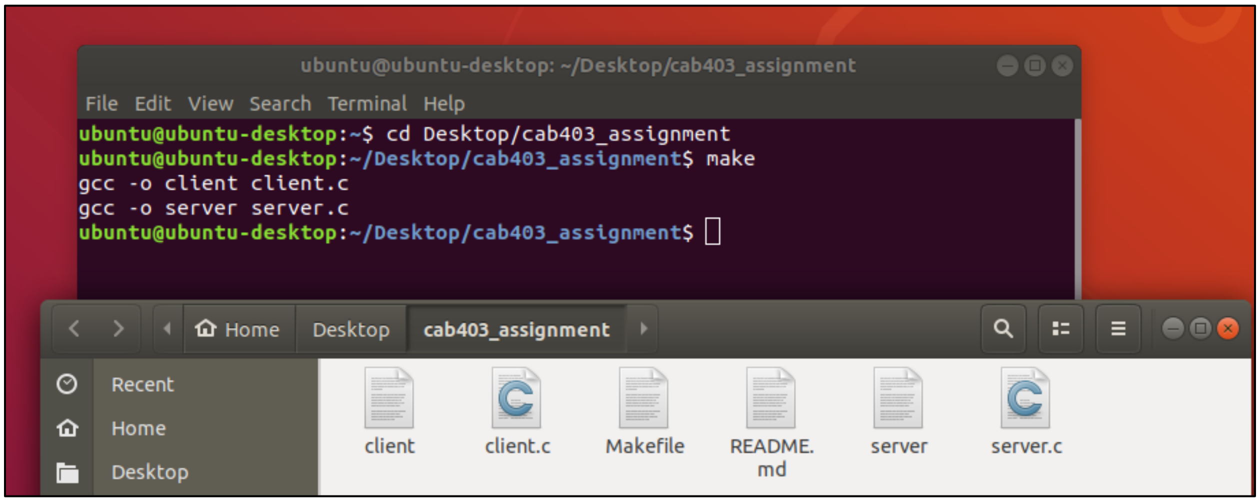
A Makefile is included in the program's files which is used to compile the files into executables in order to run the application. Using the terminal in Ubuntu, locate the Makefile by changing the directory to its location where it is placed.



```
ubuntu@ubuntu-desktop:~$ cd Desktop/cab403_assignment
ubuntu@ubuntu-desktop:~/Desktop/cab403_assignment$ 
```

File directory command to search the target folder.

When the file directory matches the Makefile's location, simply enter the 'make' command to compile the scripts inside. This Makefile will compile the files of 'client.c' and 'server.c' to allow the final files to be executed in the terminal.



```
ubuntu@ubuntu-desktop:~$ cd Desktop/cab403_assignment
ubuntu@ubuntu-desktop:~/Desktop/cab403_assignment$ make
gcc -o client client.c
gcc -o server server.c
ubuntu@ubuntu-desktop:~/Desktop/cab403_assignment$
```

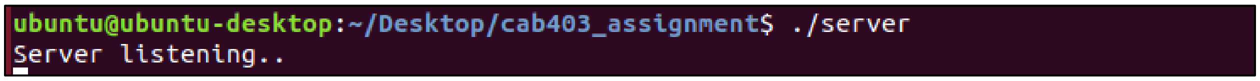
Compile the files by entering 'make'.

The compiler will release two new files with the same names as their C file scripts. The terminal is now able to access these files to demonstrate the client/server program.

Using the Program

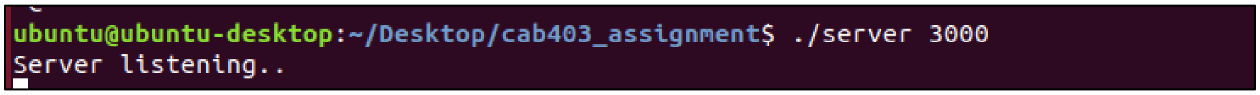
Setting up the environment

It is important to start the server before using the client. Run the server in the terminal by typing './server'. If the user wants to create a custom port, then they must append their port number to the terminal as an argument; for example './server 3000'.



```
ubuntu@ubuntu-desktop:~/Desktop/cab403_assignment$ ./server
Server listening..
```

Running the server via terminal (Default Port).

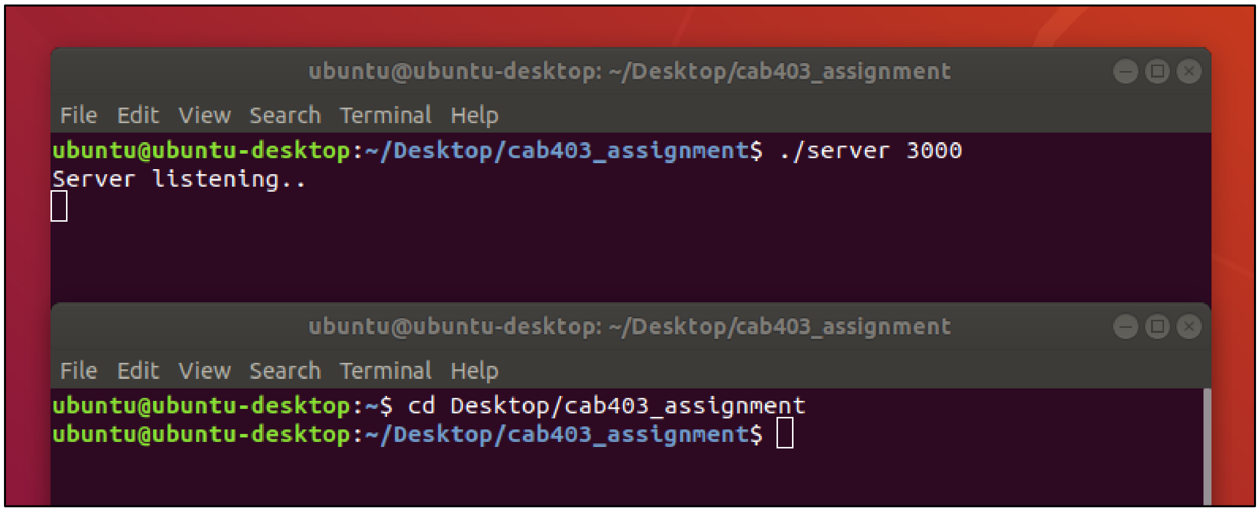


```
ubuntu@ubuntu-desktop:~/Desktop/cab403_assignment$ ./server 3000
Server listening..
```

Running the server via terminal (Custom Port).

As the server is listening to any client reaching out to it, the user can connect to the server by executing the client file. Assuming the server is running on the same computer, the user must open a new terminal while the server is on:

1. Right click on the terminal app and select 'New Terminal'.
2. Locate the client file by changing the file directory to its location.

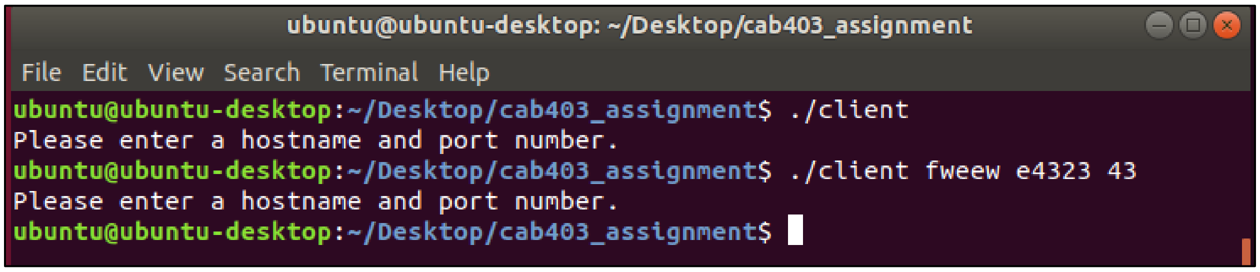


```
ubuntu@ubuntu-desktop:~/Desktop/cab403_assignment$ ./server 3000
Server listening..
```

```
ubuntu@ubuntu-desktop:~$ cd Desktop/cab403_assignment
ubuntu@ubuntu-desktop:~/Desktop/cab403_assignment$
```

Two terminals (client and server) opened at the same time.

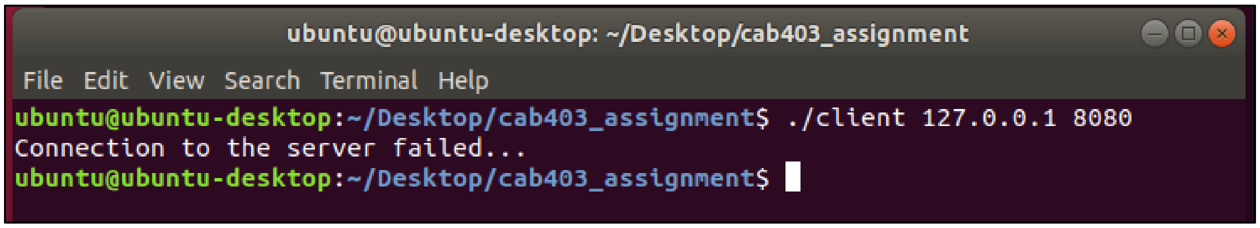
The client can be executed by entering two arguments required to connect to the server: the hostname (IP address) and the port number. Failing to enter the specific requirements will cause the client to reject the claim and ask for the following arguments. This happens when the user either runs the program by itself (empty arguments) or having more than 2 arguments.



```
ubuntu@ubuntu-desktop:~/Desktop/cab403_assignment$ ./client
Please enter a hostname and port number.
ubuntu@ubuntu-desktop:~/Desktop/cab403_assignment$ ./client fweew e4323 43
Please enter a hostname and port number.
ubuntu@ubuntu-desktop:~/Desktop/cab403_assignment$
```

Client requesting a hostname and port number.

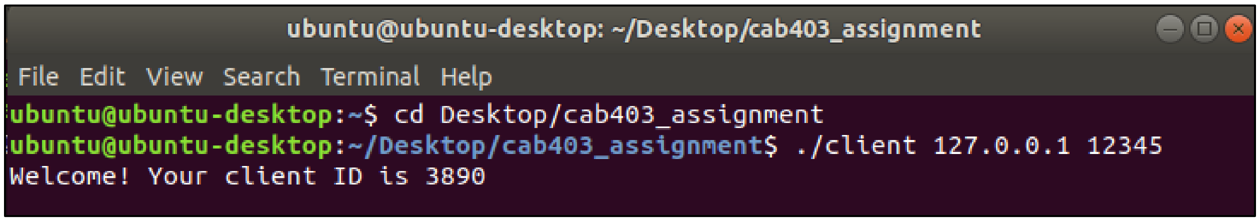
It is also the case when the user has entered the wrong port number to connect to the server. The client will prompt a message to the terminal saying, 'Connection to the server failed…'.



```
ubuntu@ubuntu-desktop:~/Desktop/cab403_assignment$ ./client 127.0.0.1 8080
Connection to the server failed...
ubuntu@ubuntu-desktop:~/Desktop/cab403_assignment$
```

Invalid port number entered in the argument.

A successful connection will display the client ID's number. If the server is running a default port, then the user must enter the port number '12345' to be able to connect.

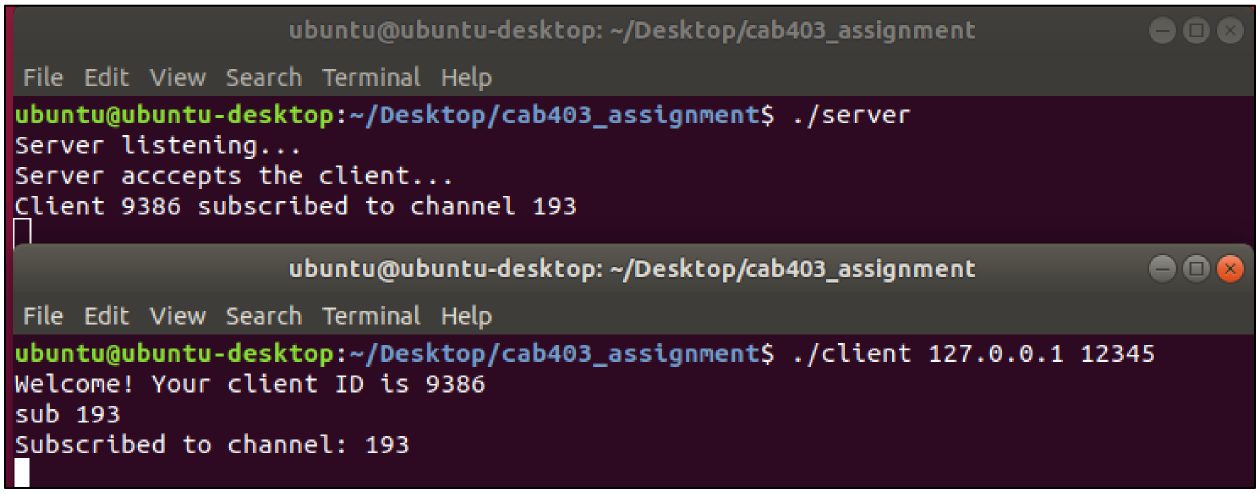


```
ubuntu@ubuntu-desktop:~$ cd Desktop/cab403_assignment
ubuntu@ubuntu-desktop:~/Desktop/cab403_assignment$ ./client 127.0.0.1 12345
Welcome! Your client ID is 3890
```

Client greeted with an ID number.

Channels

The program is essentially a message board application where users can leave messages to any channel they choose to join. The server will constantly track and log feedback to the terminal from the client's actions. Users can subscribe to a channel's ID which ranges from an integer of 0-255. To do this, simply enter the line 'sub' and an integer.

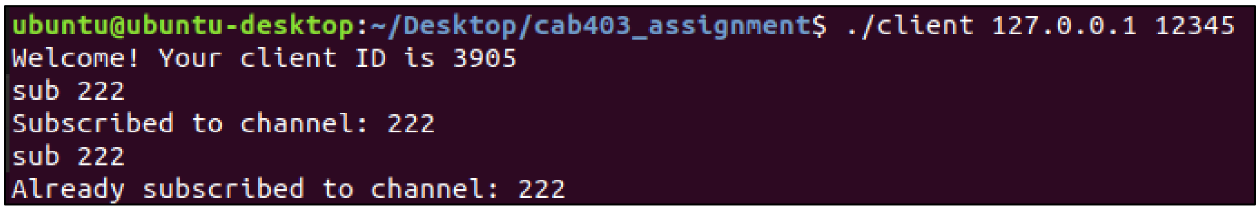


```
ubuntu@ubuntu-desktop:~/Desktop/cab403_assignment$ ./server
Server listening...
Server acccepts the client...
Client 9386 subscribed to channel 193
```

```
ubuntu@ubuntu-desktop:~/Desktop/cab403_assignment$ ./client 127.0.0.1 12345
Welcome! Your client ID is 9386
sub 193
Subscribed to channel: 193
```

Client subscribes to channel 193.

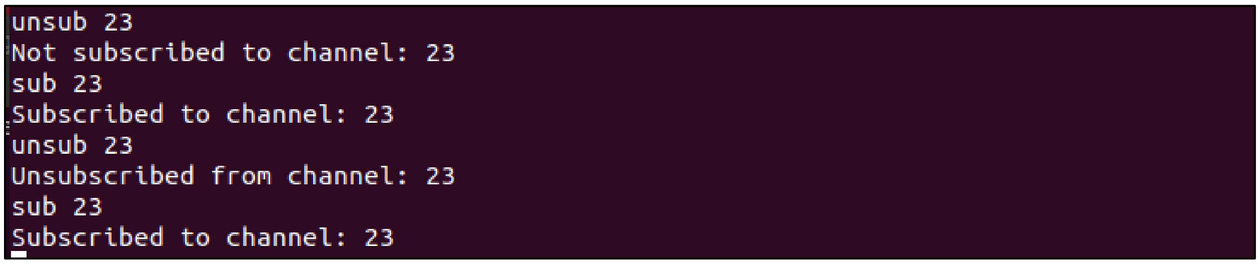
If the user subscribes to the same channel again, then the server will notify the client saying they've already subscribed to the channel.



```
ubuntu@ubuntu-desktop:~/Desktop/cab403_assignment$ ./client 127.0.0.1 12345
Welcome! Your client ID is 3905
sub 222
Subscribed to channel: 222
sub 222
Already subscribed to channel: 222
```

Client already subscribed to channel 222.

The user can unsubscribe to a channel that they've subscribed to by entering the 'unsub' command with the channel's integer value. If the user unsubscribes to an unknown channel, then the terminal will tell the user that they have not subscribed so in the first place. Once removed from the channel, the user can rejoin the channel by subscribing again.



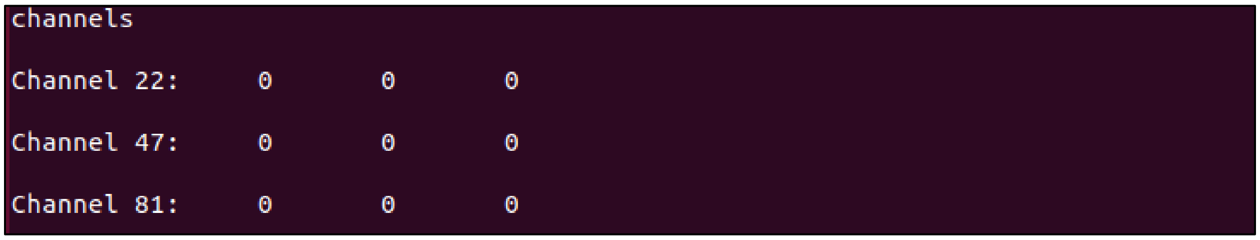
```
unsub 23
Not subscribed to channel: 23
sub 23
Subscribed to channel: 23
unsub 23
Unsubscribed from channel: 23
sub 23
Subscribed to channel: 23
```

Unsubscribing/Subscribing a channel.

In the client program, a list of subscribed channels can be shown with the statistics of (columns):

1. All messages sent to that channel
2. Messages that are marked as read by the client
3. Messages are ready but have not been seen yet

These are placed in columns for each channel in the subscription list. To view the list, enter "channels". If the user hasn't subscribed to any channel, then this feature will not be available.

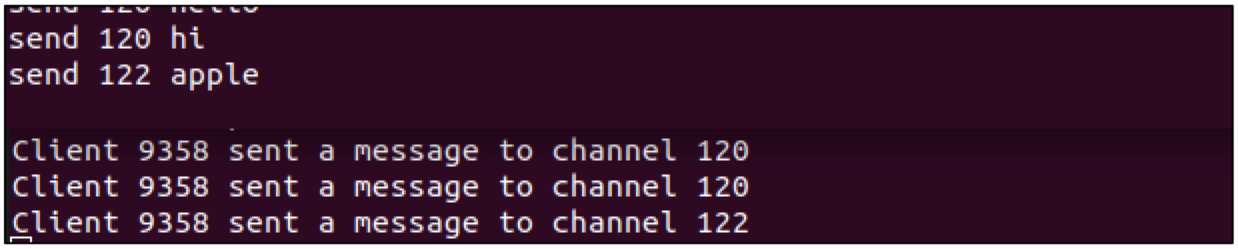


```
channels

Channel 22:     0       0       0

Channel 47:     0       0       0

Channel 81:     0       0       0
```

Subscribed channels and their statistics.

Send

Sending a message to a channel can be done so by entering the command line 'send', channel ID number, and a message separated by spaces. Maximum message length of 1024 bytes is allowed, meaning longer messages will have to be divided in segments. It is invalid to send messages beyond the channel range.

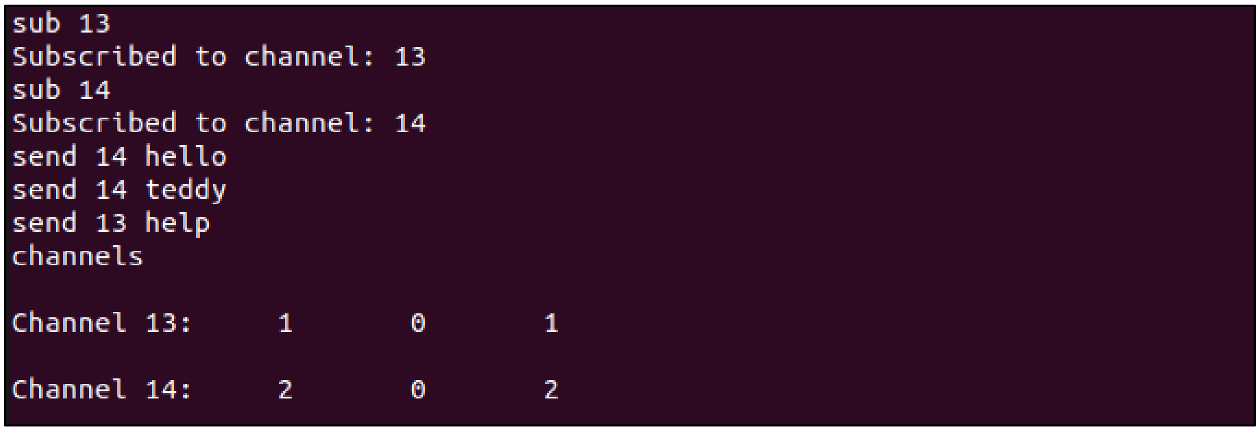


```
send 120 hi
send 122 apple

Client 9358 sent a message to channel 120
Client 9358 sent a message to channel 120
Client 9358 sent a message to channel 122
```

Sending messages to channels 120 and 122.

Next

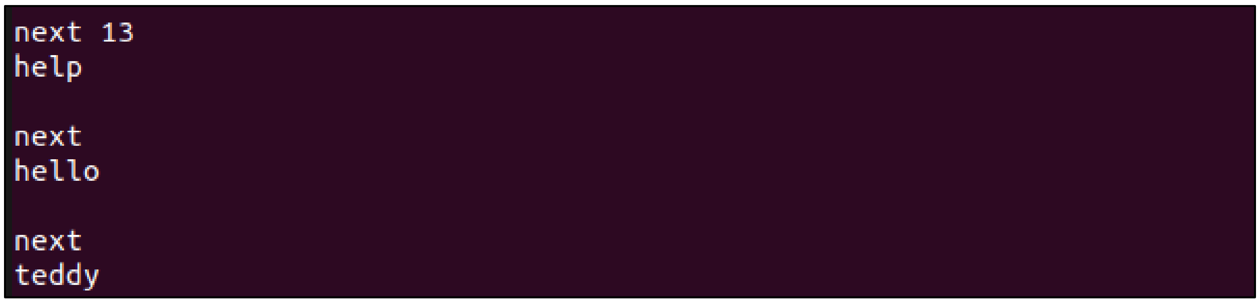
The following channels have messages that are not seen yet:



```
sub 13
Subscribed to channel: 13
sub 14
Subscribed to channel: 14
send 14 hello
send 14 teddy
send 13 help
channels

Channel 13:     1       0       1

Channel 14:     2       0       2
```

All messages sent to the channels and have not been seen yet (third column).

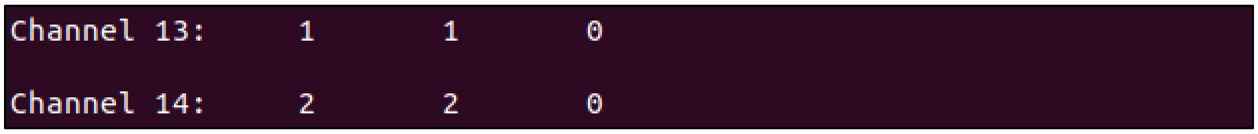
Users who subscribed to a channel can find out what messages they haven't read by entering the line 'next' for a most recent message at any channel, or specifically a message from a channel by typing 'next' appended with the channel ID. In this case, the user who've sent the messages to the channel can view them again.



```
next 13
help

next
hello

next
teddy
```

Reading the messages in the channels.

As a result, the messages in the channel have been marked as read.

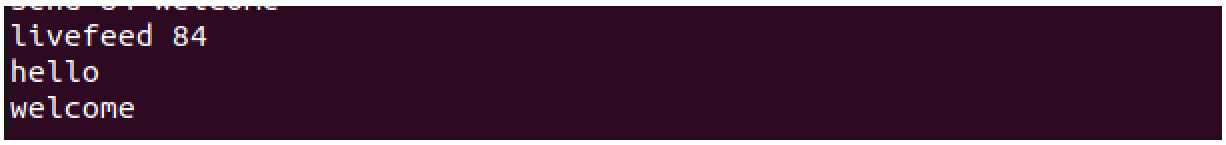


```
Channel 13:     1       1       0

Channel 14:     2       2       0
```

Messages that have been read by the client (second column).

Livefeed

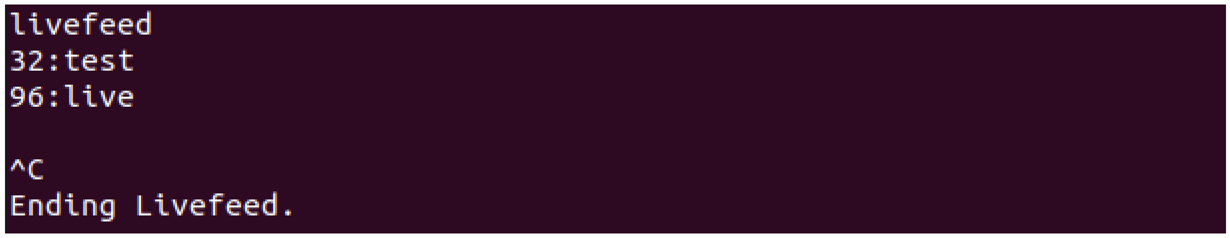
Retrieves all unread messages that the user has subscribed to and any messages sent through via livestream. By entering ‘livefeed’ and a channel ID that contains messages unknown to the user, they can view all the messages in that channel rather than using ‘next’ for one message at a time.



```
livefeed 84
hello
welcome
```

Using livefeed <channelid> to retrieve all messages in the channel continuously.

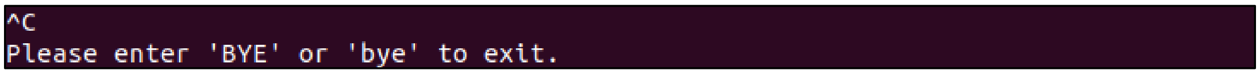
The user is able to retrieve messages continuously until they cancel the livefeed using ‘ctrl + c’. Livefeed by itself can retrieve all different messages from any channel that has been unread, allowing the user to read all messages from all subscribed channels. To know which message belongs to which, the channel ID is attached to the message with a colon ‘:’.



```
livefeed
32:test
96:live

^C
Ending Livefeed.
```

Bye

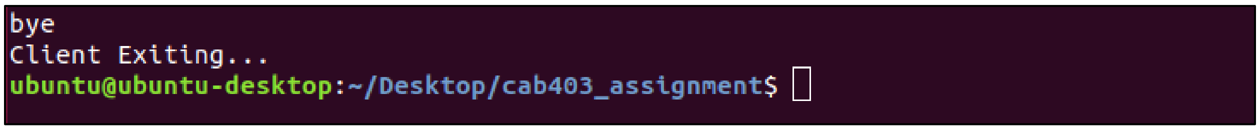
The user is unable exit by *‘ctrl + c’*, for a signal handler is applied to keep the server stable. Instead, the program will advise the user to gracefully leave the server by using the command line ‘exit’.



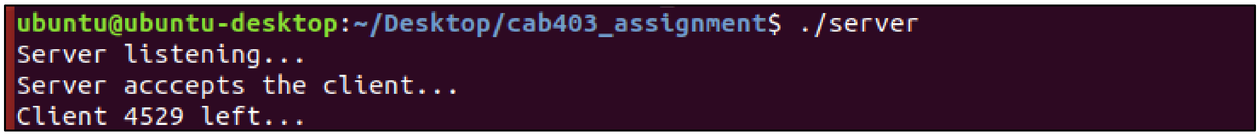
```
^C
Please enter 'BYE' or 'bye' to exit.
```

How to exit the server.

When the user leaves the server properly, the client program will exit to the terminal. The server is notified that the client has left and will still remain open for any connection. This will also unsubscribe all of the client ID’s channel.



```
bye
Client Exiting...
ubuntu@ubuntu-desktop:~/Desktop/cab403_assignment$
```

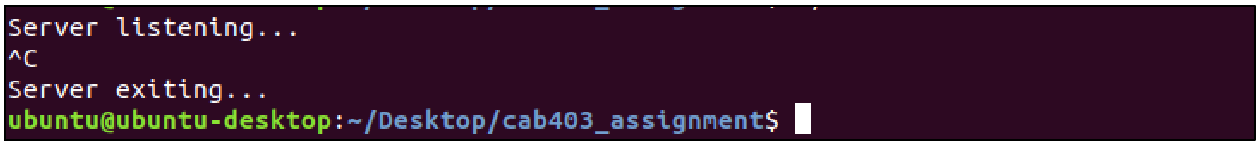


```
ubuntu@ubuntu-desktop:~/Desktop/cab403_assignment$ ./server
Server listening...
Server acccepts the client...
Client 4529 left...
```

Client/server feedback using the ‘bye’ command.

Ending the program

To end the server, press *'ctrl + c'* which will prompt a message in the terminal saying the server is exiting. If the client was connected to the server, the user in the client terminal must press enter a couple of times to exit the program.



```
Server listening...
^C
Server exiting...
ubuntu@ubuntu-desktop:~/Desktop/cab403_assignment$ 
```

Server exiting gracefully.