电子科技大学

University of Electronic Science and Technology of China

Computer Network Programming

Project Report

Researcher Name: Tesfay Semere Gerezgiher

ID: 2017080601048

School: School of Information and Communication Engineering

Instructor: Professor (Associate) Bo Chen

Project type: Multi-client chatting, file sharing implementation with socket programming

Date: 29 Nov 2020

Table of Contents

1. Introduction

Sockets allow the communication between two endpoints, sockets on the same or different machines. In other words, it is a way to talk to other computers using standard Unix file descriptors. In Unix, every I/O action is performed by writing or reading a file descriptor. A file descriptor is just an integer associated with an open file such as:

- ✧ Text file
- ✧ Terminal
- ✧ Network connection.

Socket programming is a way to allow a user to use Berkeley socket API to implement the communication we've talked about. One can use socket API's to implement multi-client server communication using these API. We can use I/O multiplexing, multi-process, multi-threading and/or others to implement multi-client server communication.

In this project, I have used select system call and multi-threading. Since select system call allows to watch multiple file descriptors, I found it useful to implement:

- ✧ A server that monitors multiple clients in the server side.
- ✧ To allow the client to enter data from the keyboard while it is monitoring in coming data from other clients through the server in the client side.

Multi-threading is also to decrease the load from the main thread when a file is being shared. So when a file is to be shared, another thread is created: so, that thread can do the file sharing between the clients between which the file sharing is to be done (i.e. between the client who would like to share a file and the receiver of that file).

2. Project objectives

The main objective of this project is to let me have an idea about how information can be routed from and to the client in real world, to develop a skill in computer netw-orks socket programming using C programming. It also helps me:

- ✓ Become more familiar with socket programming.
- ✓ Gain hands-on experience with i/o multiplexing, select API, and multithreading.
- ✓ Gain more knowledge about Linus OS and C programming.
- ✓ Gain hands-on experience on debugging with gdb debugger on Ubuntu Linux distro terminal.

Thereafter, we get our final score.

3. Project Requirement

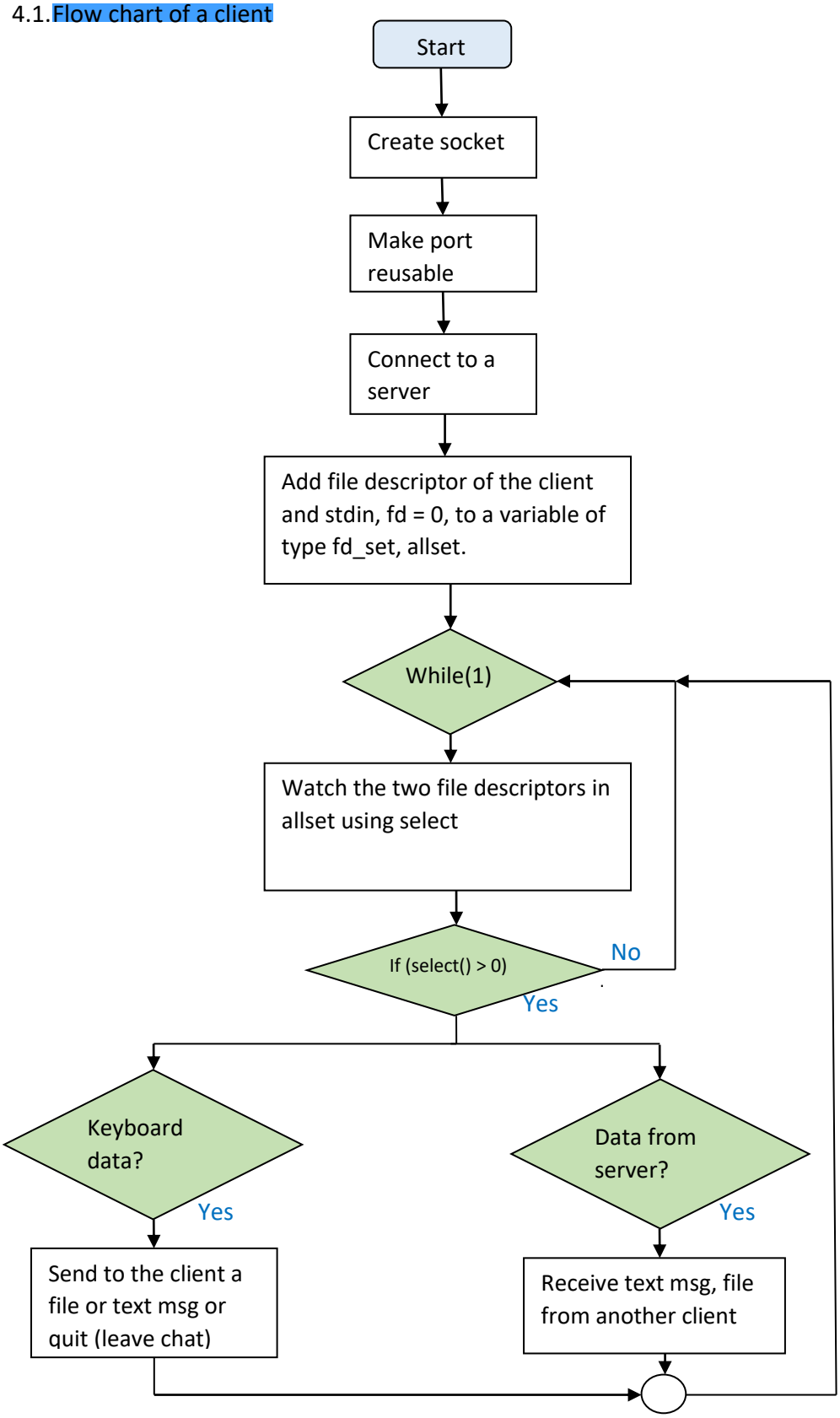
I am required to implement a server that provides a private chat between clients, public chat room and that supports file-sharing between the clients. With this project, the following tasks are implemented.

- ✓ A client can send a private message to an individual, single client.
- ✓ A client can send a public message to all the online clients in the chat room.
- ✓ The client can transfer file to another online client, who is still connected to the server.

In this project, clients do not communicate directly with other clients. Instead, all communication is routed through the server. i.e. I have implemented how information is routed from one client to another in real-world.

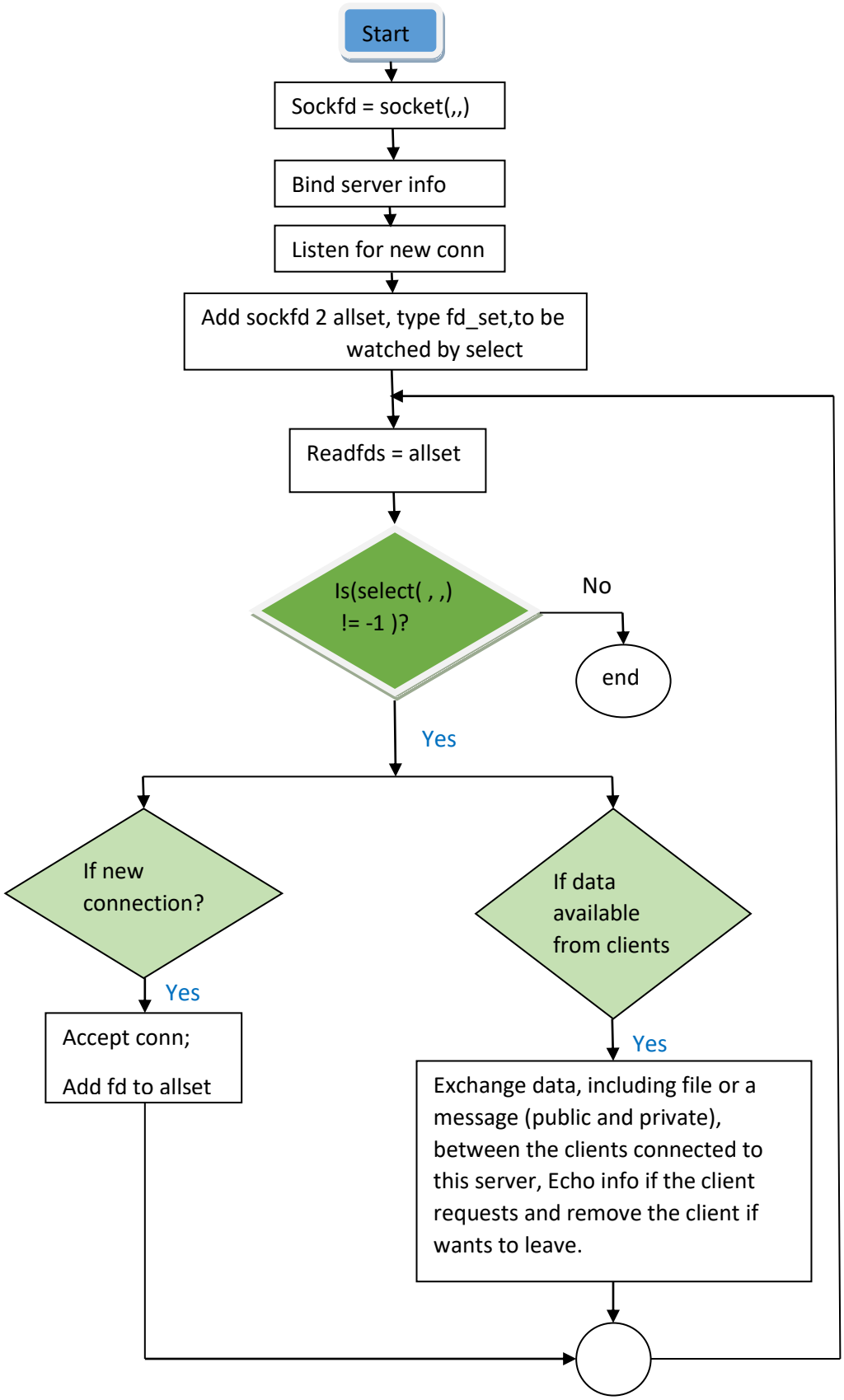
4. Flow-chart of a server code and a client code

4.1. Flow chart of a client



A flow chart that shows a client design.

4.2.Flow chart of a server



Working flow of my server

5. User defined structures and important function definitions

In this project, I have used several built-in and user-defined functions to make my code clearer and readable. Some of the functions I have used are listed and their usage and purpose are discussed meticulously.

5.1. User defined structures

```
typedef struct client{
    int connfd; //file descriptor of the client
    struct sockaddr_in addr;
    char *name; //name of the client
    char *data; //keep track of chat history
    int dlen; //keep track of data length
}Client;
```

Figure -1.1: user defined data structure for a client

These structures models a client whose file descriptor is 'connfd', name is stored in a memory address pointed to by a pointer 'name', chat history is stored in a memory address pointed to by a pointer 'data', and the length of string stored in a memory stored in 'data' is saved in 'dlen'. The information of the socket of the clients is stored in a structure addr.

```
typedef struct File_info{
    int r_fd; //receivers file descriptor
    int s_fd;
    char filename[20];
}File_info;
```

Figure -1.2: user defined data structure for File_info

File_info is a structure that models file sharing between the sender, whose file descriptor is 's_fd', to a receiver, whose file descriptor is 'r_fd', and the name of the file to be shared is 'filename'.

```
/*data structure for registered users*/
typedef struct {
    char s_ip[20]; //sender ip
    int s_port;  //sender port
    char *offline_msg;
    char r_ip[20]; //receiver ip
    int r_port; //receiver port
    char flag[2];
}Reg_users;
```

Figure -1.3: user defined data structure for registered User.

Reg_users models a user or client that has been registered or connected to a currently running server, the client may be signed out.

Some of the members of this structure are commented as shown in the figure above.

- ✧ Offline_msg: is used to store a message sent to a client, who is offline (i.e. has left the connection to a server for a moment), whose ip and port is r_ip and r_port respectively from a sender whose ip and port is s_ip and s_port respectively.

5.2. Berkeley APIs used and getaddrinfo

✓ Getaddrinfo():

Syntax: int getaddrinfo(const char *node, const char *service,

const struct addrinfo *hints,

struct addrinfo **res);

where:

- ✧ Node: refers to a port at which the socket listens on.
- ✧ Service: refers to a service or hostname, IP address of the socket.
- ✧ Hints: The hints argument points to an addrinfo structure that specifies criteria for selecting the socket address structures returned in the list pointed to by ***res***.
- ✧ Res: a linked list of data structures that contains a set of possible socket addresses.

✓ Socket(): creates an endpoint for communication and returns a file descriptor that refers to that endpoint.

Syntax: int socket(int domain, int type, int protocol)

Where,

- Domain, or address family, is a communication domain in which the socket should be created. e.g. AF_INET(IPv4), AF_INET6(IPv6)
- Type – is type of service. It is set to SOCK_STREAM for TCP.
- Protocol – indicate a specific protocol to use in supporting the socket operations.

E.g. sockfd = socket(AF_INET, SOCK_STREAM, 0);

✓ Bind(): assigns the address specified by addr to the socket referred to by the file descriptor, sockfd.

Syntax: int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)

Where,

- sockfd is a file descriptor returned by socket API.
- Addr is a structure that contains information of the server.
- Addrlen is the size of the structure that contains information for the server.

E.g. bind(sockfd, (struct sockaddr*)&server, sizeof(server))

✓ Connect(): is a socket API used by the client to establish a connection with the server.

Syntax: int connect(int sockfd, const struct sockaddr *addr,

socklen_t addrlen);

E.g. connect(sockfd, (struct sockaddr*)&server, sizeof(server))

✓ Listen():a system call that marks the socket referred to by sockfd as a passive socket.

Syntax: int listen(int sockfd, int backlog)

Where, backlog specifies the maximum number of incoming connections that can be added to the queue.

e.g. listen(sockfd, 5);

✓ Accept(): is a socket system call that extracts the first incoming connection request on the queue and returns a new file descriptor that refers to that node or socket on success.

Syntax: int accept(int sockfd, const struct sockaddr *addr, socklen_t * addrlen)

- Where, addr is a structure of type struct sockaddr_in that is set up to hold information of the client and, then, is casted to struct sockaddr* data type.

E.g. clifd = accept(sockfd, (struct sockaddr*)&clientaddr, &sin_size)

Where, sin_size = sizeof(struct sockaddr_in)

✓ Send(): system call used to transmit data to another socket, either server or client.

Syntax: int send(int sockfd, const void *buf, size_t len, int flags);

Where

- Sockfd is a file descriptor of the client: returned by accept system call in the server side.
- Buf is an array of characters to hold the data to be sent to the other node.
- Len is length of the message to be sent.

E.g. send(clifd,buffer,strlen(buffer),0)

✓ Recv(): system call used to receive data from another socket end, either from the server or the client.

Syntax: int recv(int sockfd, const void*buf, size_t len, int flags);

Where,

- Sockfd is a file descriptor of the client: returned by accept system call in the server side.
- len is the length of the buffer. Buf is explained in the send() API above.

E.g. recv(clifd,buffer,strlen(buffer),0)

✓ close(): closes any open descriptor.

Syntax: in close(int fd); // fd is a file descriptor to be closed.

E.g. close(sockfd)

✓ Select() API:

Syntax: int select(int nfds, fd_set *readfds, fd_set *writefds,

fd_set *exceptfds, struct timeval *timeout);

where,

- Nfds: the maximum number of descriptors ready. Select() can monitor only file descriptors numbers that are less than FD_SET_SIZE.
- The file descriptors listed in readfds will be watched to see if data is available for reading.
- The file descriptors listed in writefds will be watched to see if space is available for writing.
- The file descriptors listed in exceptfds will be watched for exceptional conditions.
- Timeout: defines how long the kernel, select to be specific, should wait to return. It is a pointer to a structure of type struct timeval.

  Syntax: struct timeval{

  long tv_sec;

  long tv_usec;

  }

In our experiment I set timeout to NULL, which means the select API will wait forever.

Return value: select returns the number of file descriptors that are ready, contained in the three file descriptor sets, on success, zero when timeout and -1 on failure.

✓ void FD_CLR(int fd, fd_set *set); //removes a file descriptor fd from set

✓ int FD_ISSET(int fd, fd_set *set); //checks if fd is present in set.

✓ void FD_SET(int fd, fd_set *set); //adds a file descriptor fd to set

✓ void FD_ZERO(fd_set *set); //clears a set

5.3. Important function definitions

I have used the following functions to implement important tasks and functions of the server.

a. `void echo_message(int connfd, const char *msg)`

this function receives connfd, file descriptor of the client which the server wants to send a message to, and msg, a pointer to a string that contains the information to be sent.

b. `void send_online_clients(int connfd, Client *client)`

sends to the clients that are still connected to the server, hence online clients. Receives to arguments.

- ✧ Connfd: file descriptor of the client to which the online clients is to be send to.
- ✧ Client: a pointer that points to the address of the first client structure.

c. `void send_reg_clients(int connfd, Reg_users *user)`

Sends a set of clients that has been registered or connected to a server. The arguments are:

- ✧ Connfd: file descriptor of the client to which the registered clients are to be send to, hence the clients that had been connected but now left are included.
- ✧ User: a pointer that points to the address of the first user of type Reg_users.

d. `void public_message(Client *cli, const char *buf,int except)`

This function sends a public message from a client whose file descriptor is except a message stored in a memory pointed to by a pointer **buf** to the online clients where the address of the first client is stored in a pointer cli.

e. `int is_regs(char *ip, char *prt, Reg_users *usr)`

This function checks if the client whose IP and port is pointed to by a pointer 'ip' and 'prt' respectively has been registered, connected to the server, from a list of registered users or clients stored in an array of structures where the address of the first array is pointed to by a pointer 'usr'.

Return value: the index of the client in the array if the client is registered, -1 is returned if the client has never been to a client before.

f. `int is_online(char *ip, char *prt, Client *cli)`

'is_online' is a function used to see if the client with IP address and port number is stored in a memory pointed to by 'ip' and 'prt' respectively is still connected to the currently running server. a list of clients is stored in an array of structures of type Client where the address of the first client, at index 0, is pointed to by a pointer 'cli'.

Return value: the index of the client in the array if the client is registered, -1 is returned if the client has never been to a client before.

6. Project Result and Analysis

This project, the server to be specific, is expected to implement multi-client chat room, where each client can send private message to an individual and can also send a public message to all the clients that are connected to the server at the moment. It is also required to support file sharing between these clients.

6.1. Client Architecture

The client is designed to allow incoming data from the server and incoming data from the keyboard at the same time using select API. The following steps has been performed to create a client.

- Create client socket.
- Connect to the server.
- Add file descriptor of the client socket to readfds so that select can watch it for an incoming data from the server.
- Add file descriptor of the sdtin(fd = 0) to readfds so that select API can watch it for an incoming data from the keyboard.
- Select API watches the file descriptors in readfds if data is available. When a data is available, select API returns.
- Exchange files, including image and text file, exchange text messages, private and public, with the other parties connected to the server.
- Close the socket when the client types the command '/q' and leave the chat room.

6.2. Commands a client can use

When a client wants to get a special service from the server, it has to type forward slash followed by a service type. A client can send a public message just by typing a message from the keyboard, stdin, but has to type a '/help' command to get the set of commands that are allowed to the client.

- ✓ A client can see the set of clients that are online by typing '/ls'.

- ✓ A client can see the set of clients that are registered, have been connected, to the server by typing '/als'.
- ✓ A client can leave the chat, close the connection with the server, by typing '/q'.
- ✓ A client can rename itself by typing a command '/name' followed by its new name.
- ✓ A client can send a private message by typing a command '/msg' followed by the receivers IP and, then, receivers port, which in turn is followed by the actual message, hence <private msg>.
- ✓ A client can share a file with another client by providing the receivers IP and port, followed by the name of the file (hence <filename>), preceded by the command '/file'.

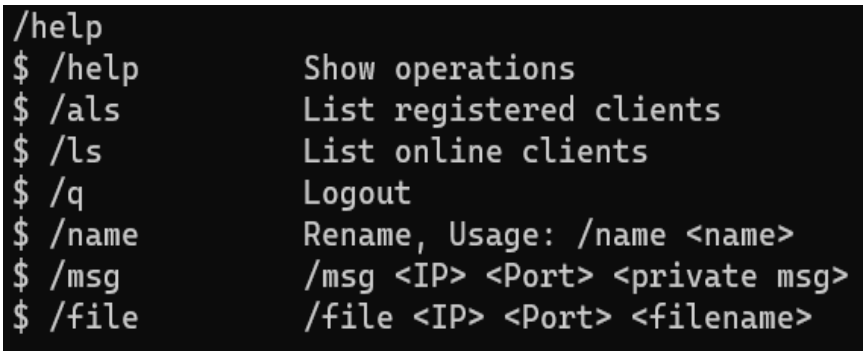


Figure -2.01: a set of commands that can be used by a client connected to a running server.

The client needs to provide the receivers IP address and port number when sending a private message or sharing a file to the client in another end point. The client can see the set of clients that he can send a message or file to (should be online) by typing a command '/ls'. Thereafter, the server feeds back each clients IP address and port to the requesting client; then, it is displayed on the screen of the client who made the request.

6.3. Result and analysis

Before you see the results of this project, let me provide you with an appropriate idea of the capability of the server.

- ✓ The server supports both IPv4 and IPv6 (this is made possible through the use of getaddrinfo).
- ✓ A client can send a private message to another client as many as he/she likes.
- ✓ A client can receive any message, private and public, without blocking for incoming or outgoing message.
- ✓ A client can send a public message to all other clients.
- ✓ A client can share files with other online clients.

I have provided you with the following screen-shots as a prove that I have implemented all the required tasks and optional tasks, such as file sharing, public message between clients, etc…

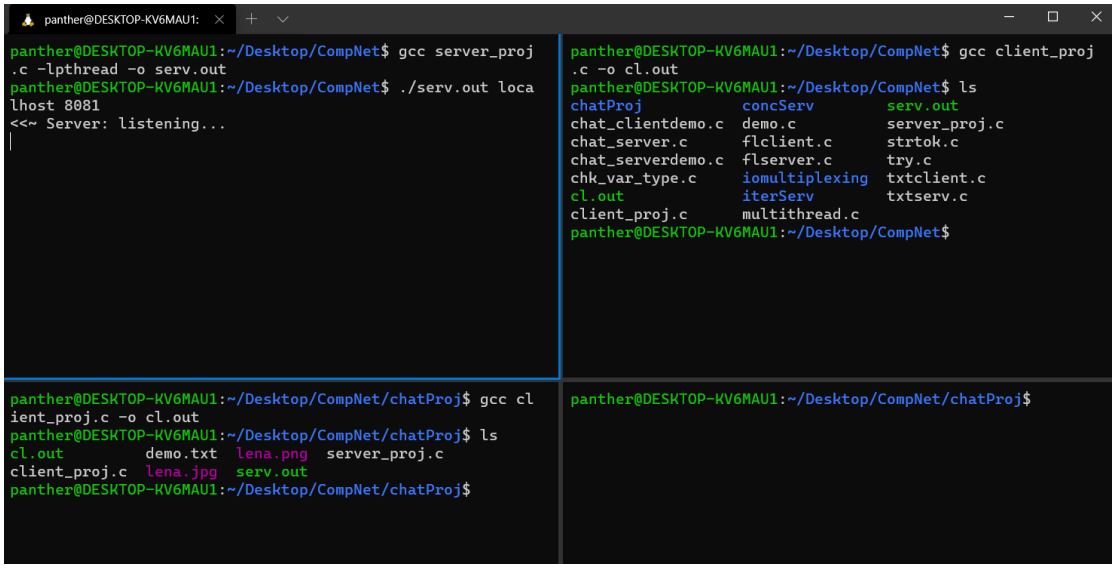


Figure -2.02: the code of the client and the server is compiled successfully. Now the server is listening for an incoming connection request. The server is listening on IP: localhost and port: 8081.

Note: the files are saved in the directory '/CompNet/chatProj' as you can see in the bottom left side. While the 'CompNet' directory has no file as you can see in the top right side. I will try to share a file to the client whose executable is in the 'CompNet' directory.

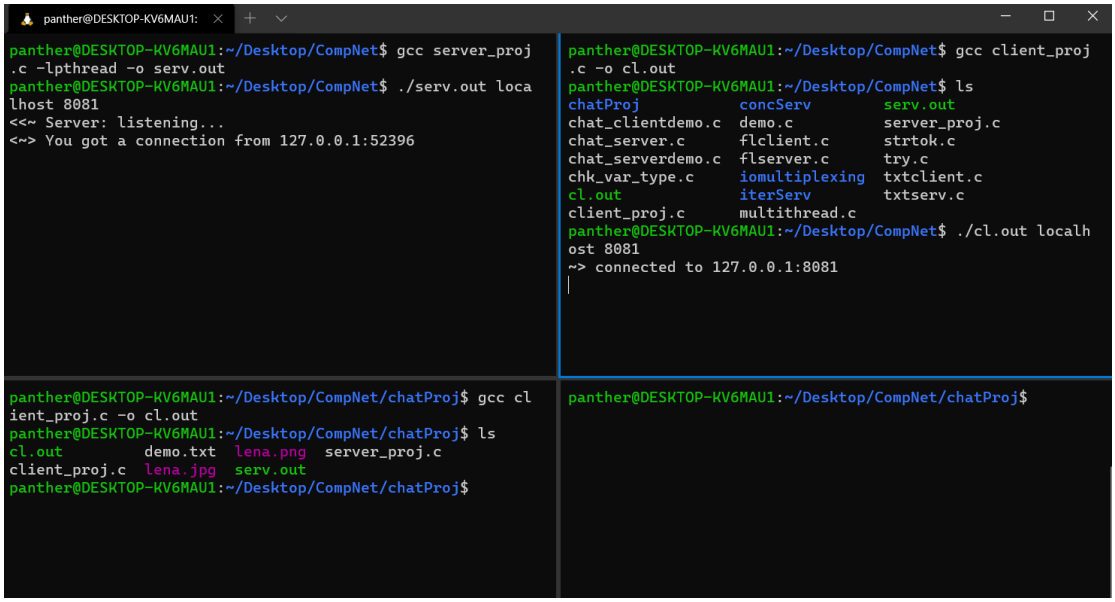


Figure -2.03: the server has accepted an incoming connection request from a client at IP: localhost and port: 52396.

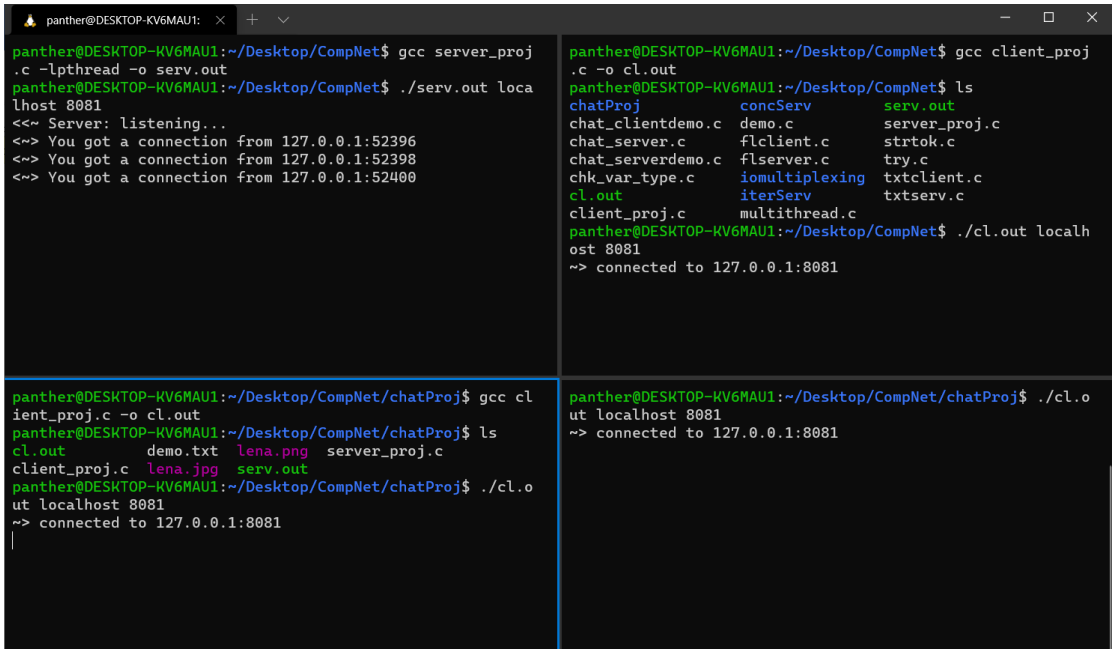


Figure -2.04: two more client are now connected to a running server listening at IP: 127.0.0.1 and port: 8081. There corresponding IP and port number is shown in the server, top left side of the figure above.

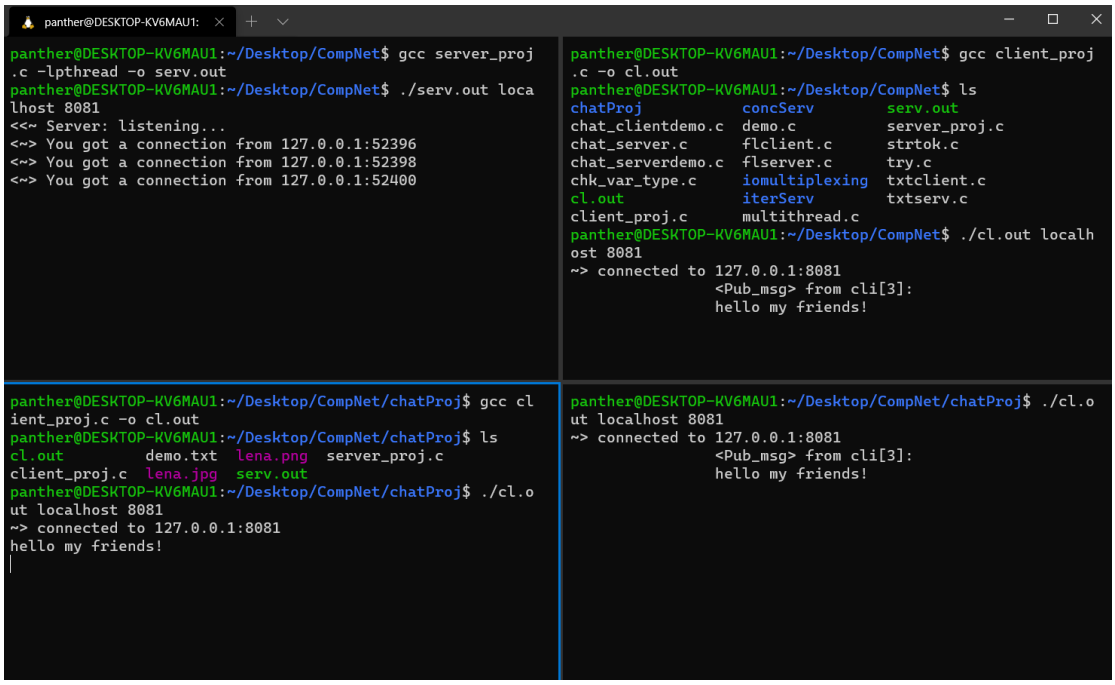


Figure -2.05: the client at the bottom-left side of the picture above sends a public message ‘hello my friends!’ to all the online clients. The other clients, on the right half of the picture, have received the text message from the client on the bottom-left side. You can see that <pub_msg> indicates the message received is a public message. Since the name of the client hasn’t been changed, the default name (which is the file descriptor of the socket that corresponds to that client is used) is displayed within the square bracket.

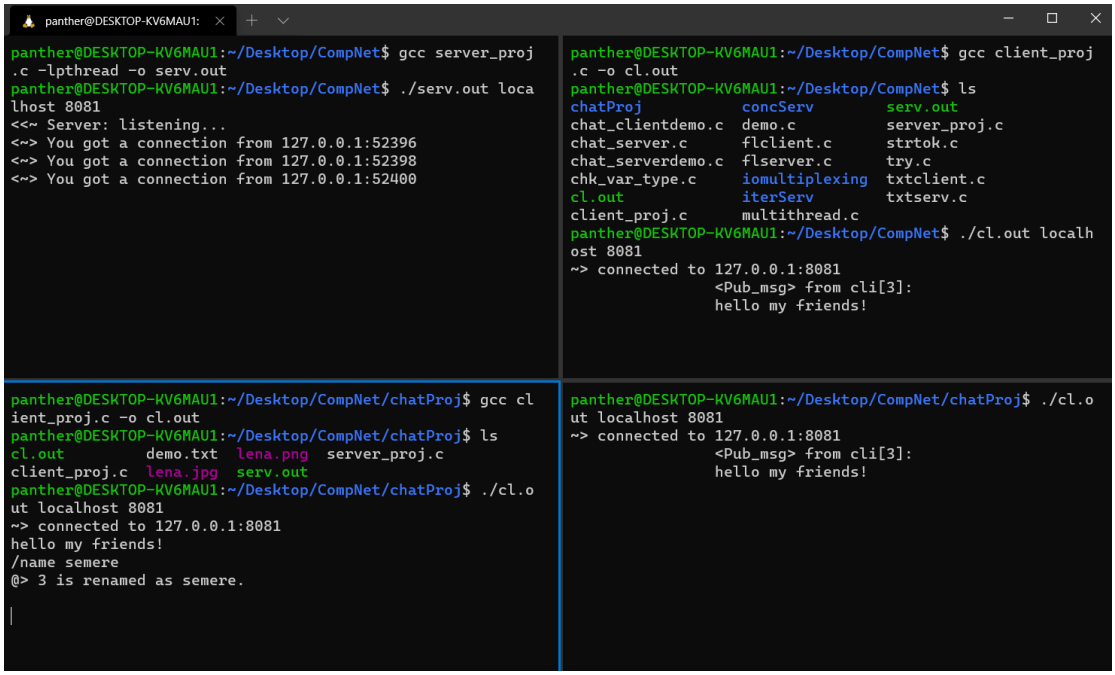


Figure -2.06: the client on the bottom-left side has renamed itself as semere.

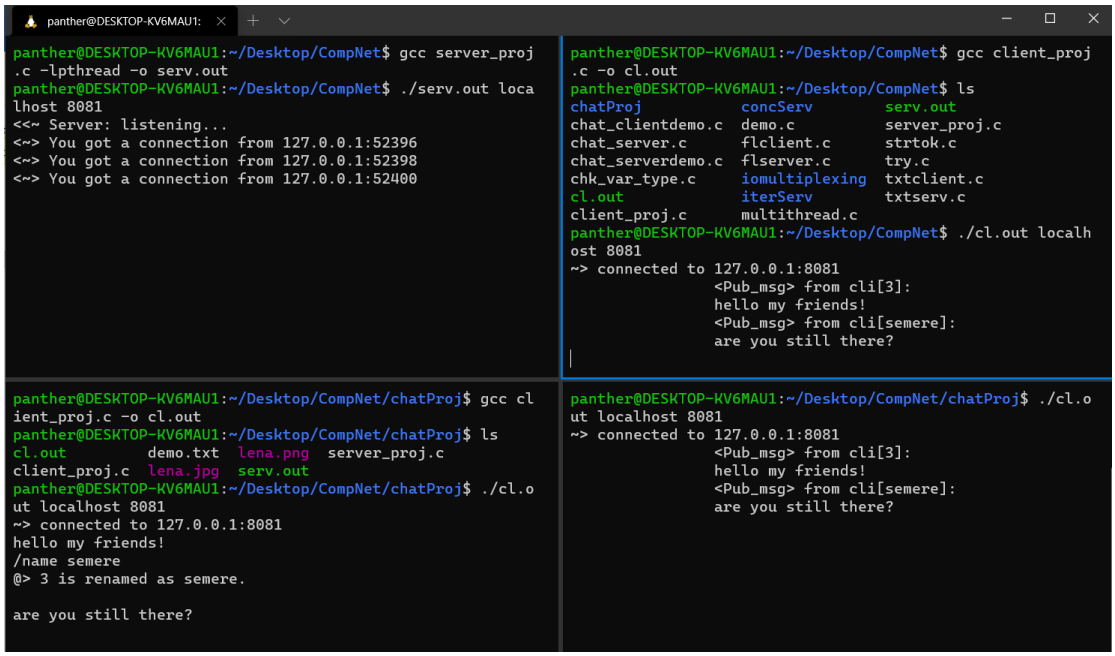


Figure -2.07: the client, name as semere, has sent a public message ‘are you still there?’ to the other clients connected to the server.

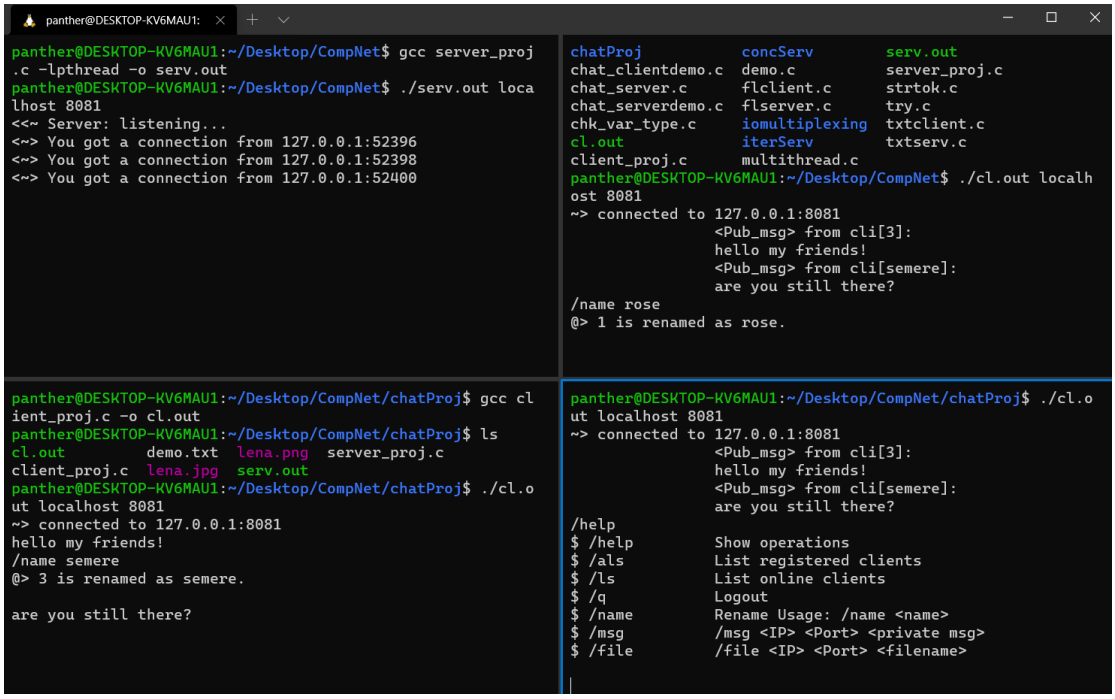


Figure -2.08: the second client, on the top-right side of the figure above, has renamed itself as rose.

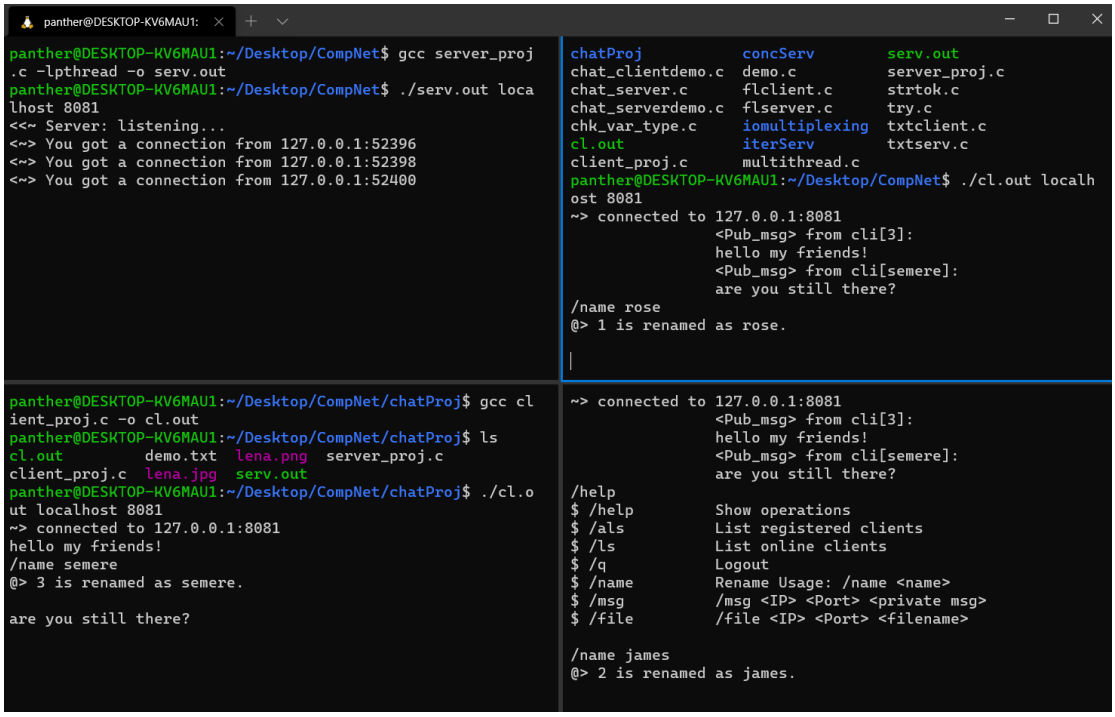


Figure -2.09: the third client, on the bottom-right side of the figure above, has renamed itself as james. This client has asked for help, by typing the command '/help', before it has renamed itself.

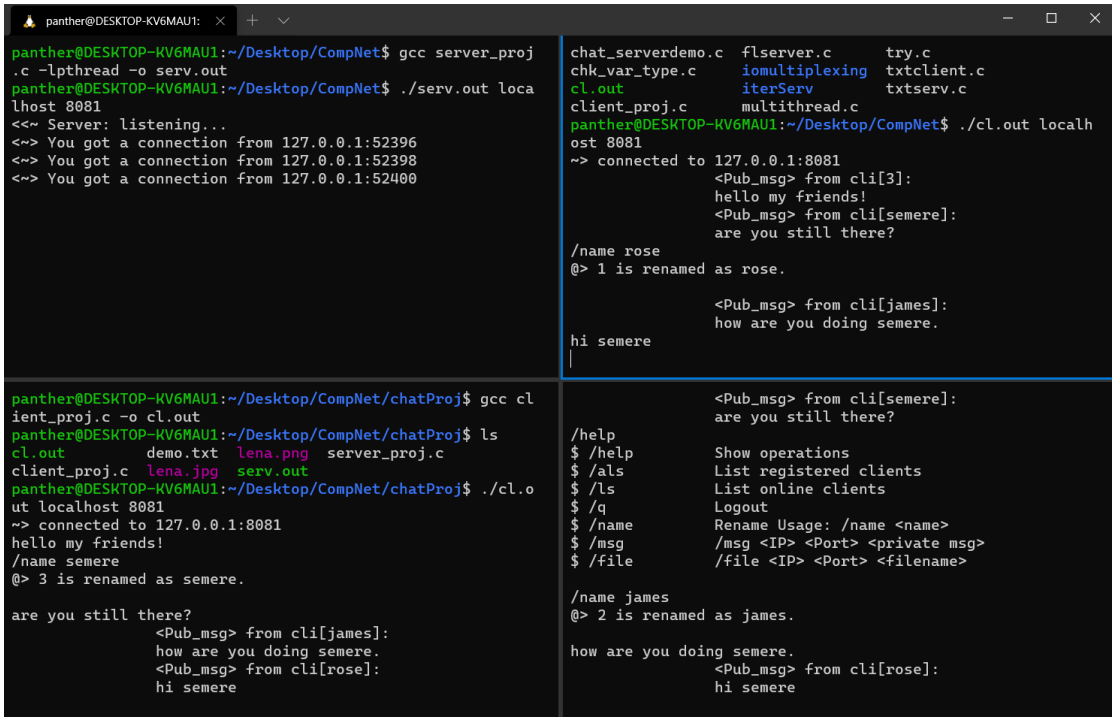


Figure -2.10: the clients are chatting with each other, by sending a public message to one another.

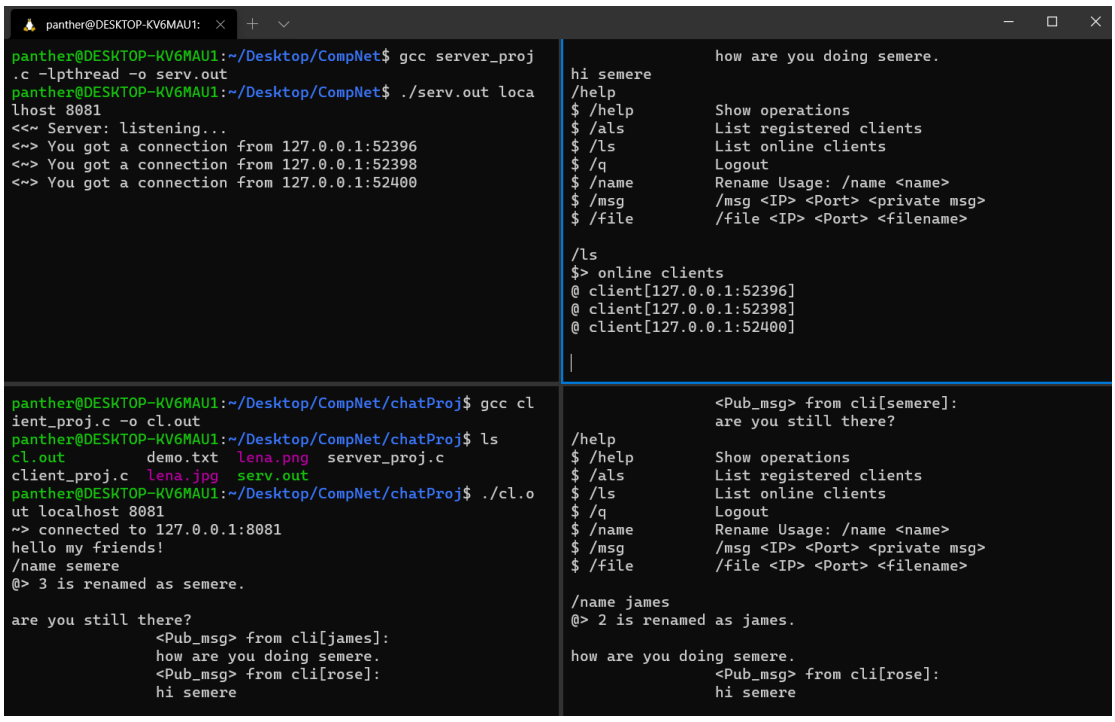


Figure -2.11: the client on the top-right side, whose name is rose, has asked for help and typed a command '**/ls**' to see the clients that are online. As you can see the list of clients that are online are listed and their corresponding IP and port is displayed.

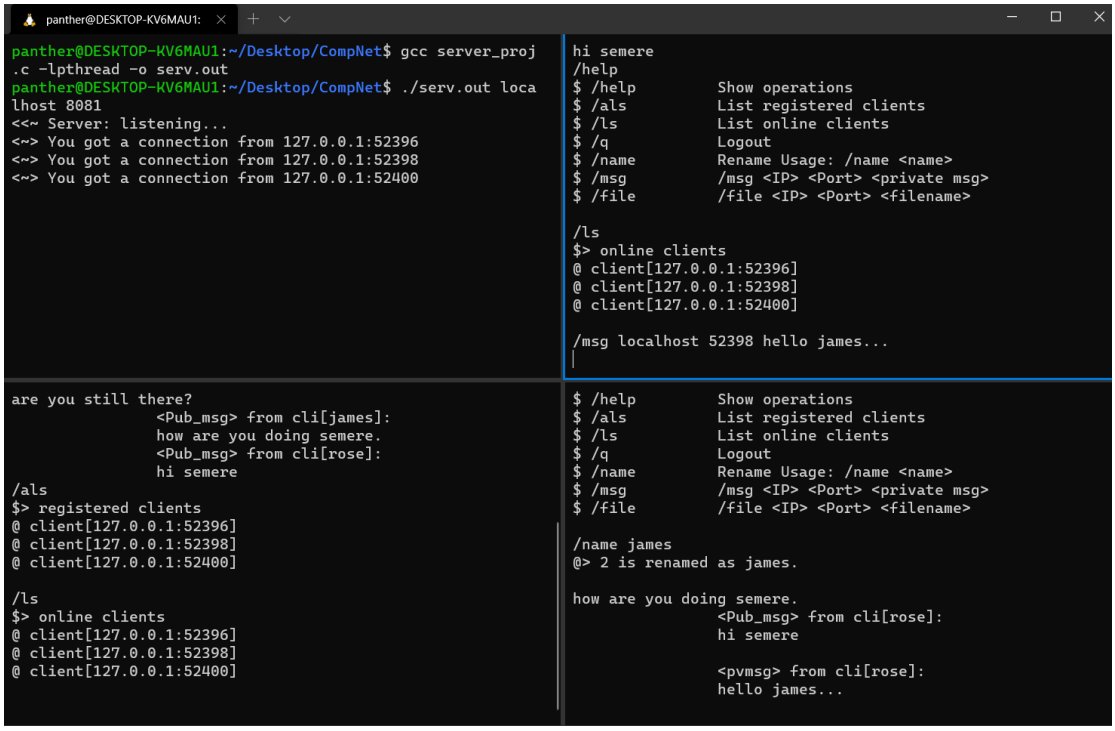


Figure -2.12: the client on the bottom left, named as semere, has typed '/als' and '/ls' to see the clients that are registered and to see the clients that are online respectively. Rose, the client on the top-right side has sent a private message to james, the client on the bottom right side. Hence, <pvmsg> indicates a private message.

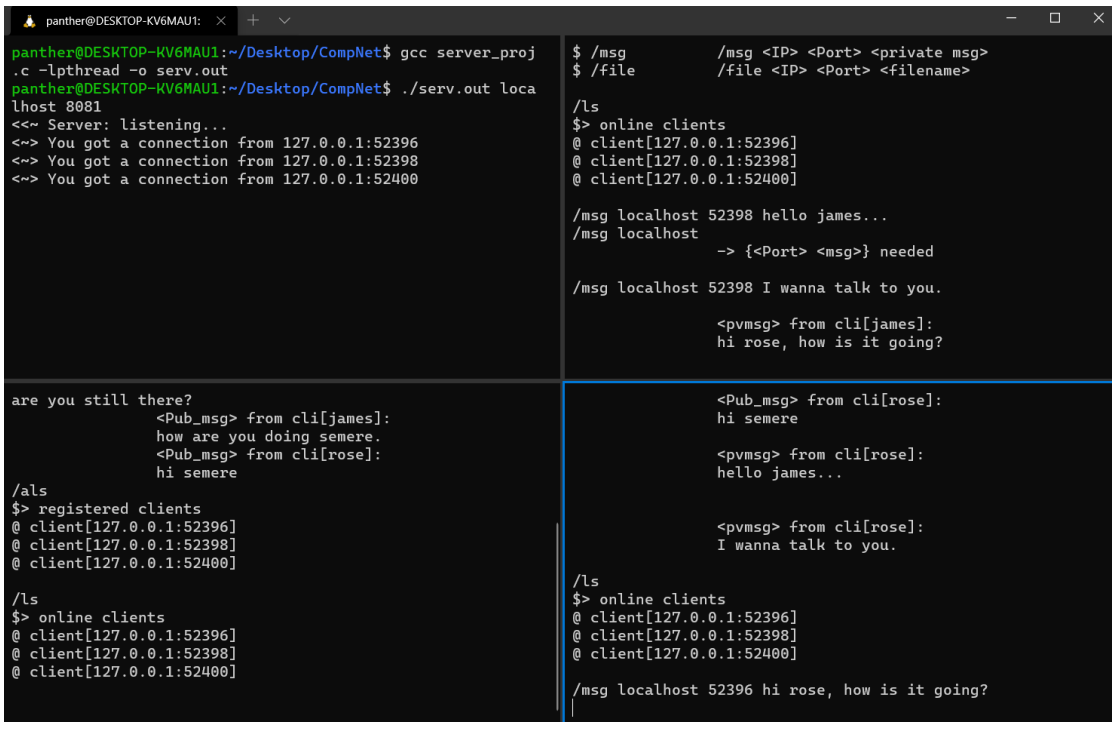


Figure -2.13: james, client on the bottom right side, has seen the clients that are online by typing the command '/ls' and replied a private message to rose, the client on the top right side. Hence, <pvmsg> indicates a private message.

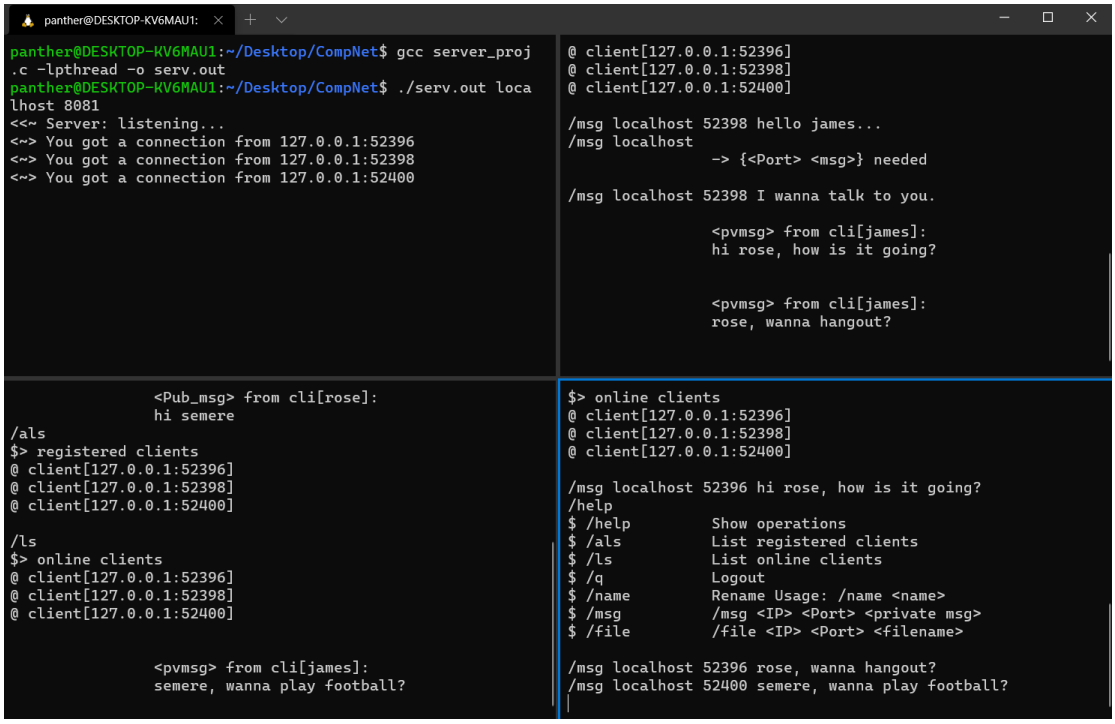


Figure -2.14: the third client, namely james, has sent two private messages: one to semere and one to rose.

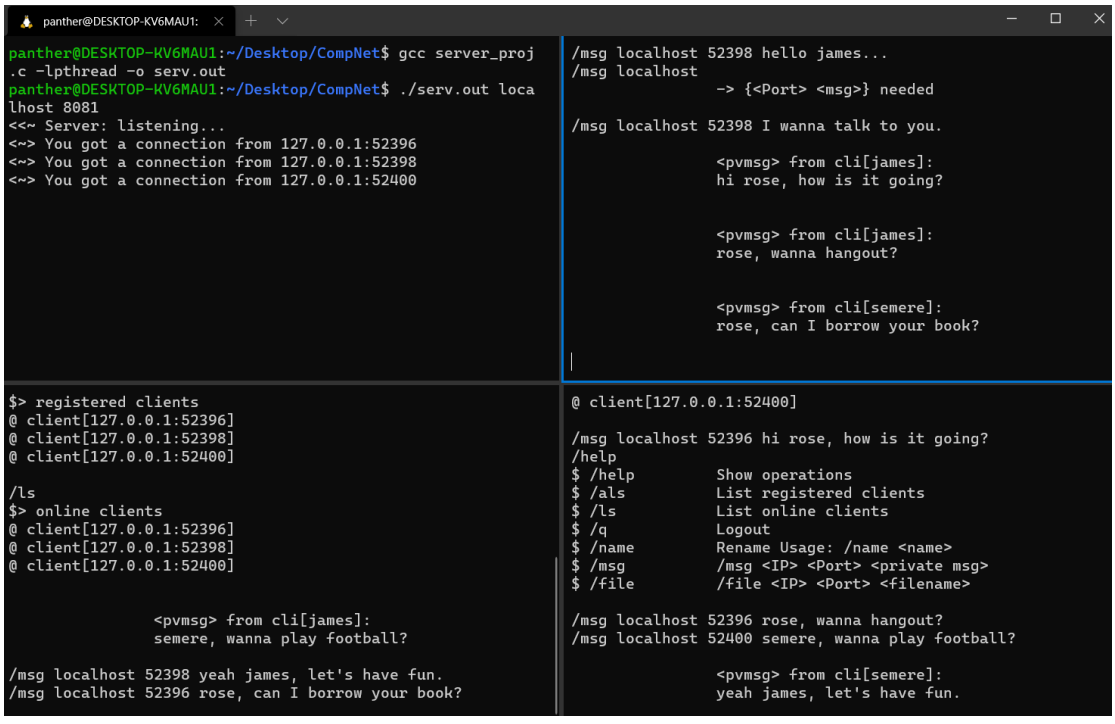


Figure -2.15: the second client, namely semere(on the bottom left side of the figure above), has sent two private messages: one to james and one to rose.

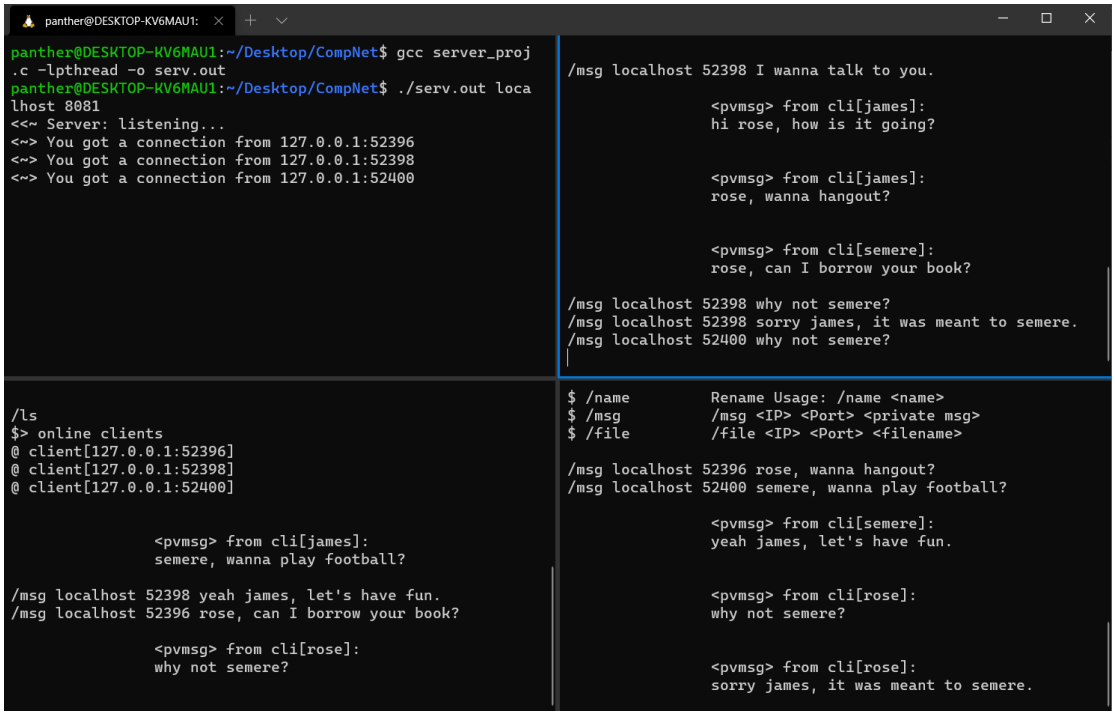


Figure -2.16: the client, on the top right sub-window, has sent two private messages to james (the client shown on the bottom-right sub window above) and one private message to semere (shown on the bottom-left sub window of the figure above).

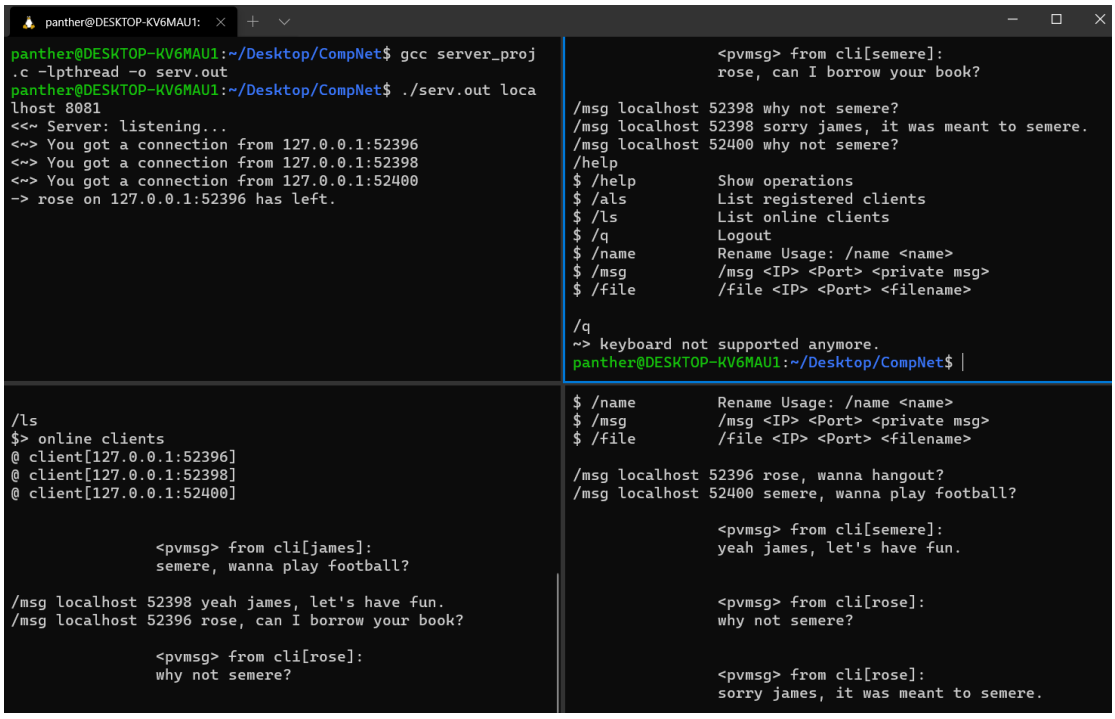


Figure -2.17: rose (shown in the top-right sub window with port:52396) has left the chat room by typing ‘/q’.

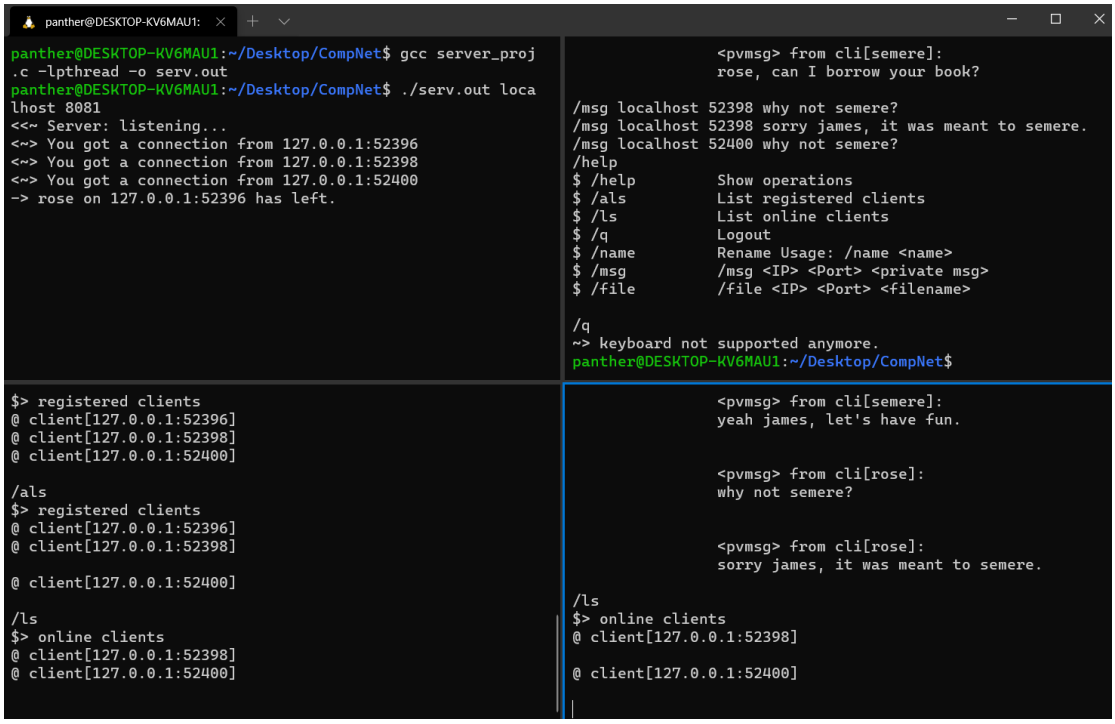


Figure -2.18: since there are only two clients left online, two clients are listed when a client types the command '/ls'. And since the number of clients who has been connected to the server is 3, typing '/als' displays those three clients who have registered. This is indeed the case what we can see in the bottom-left and bottom-right sub-windows of the figure above.

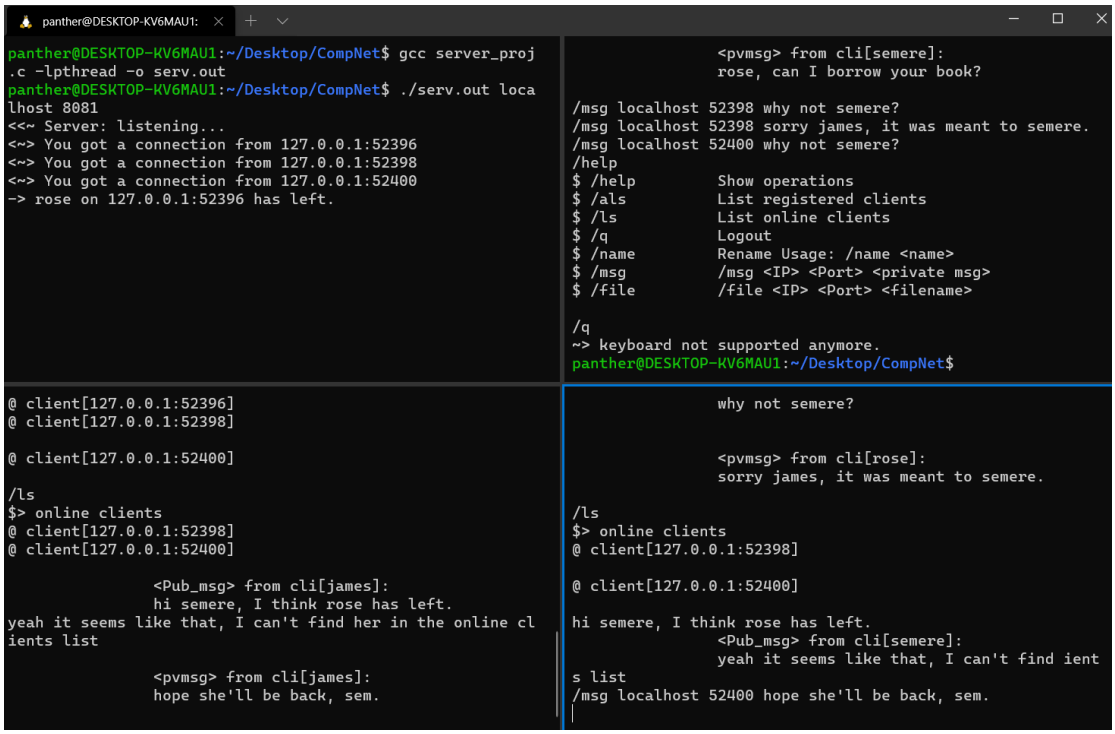


Figure -2.19: the two clients that are online keep chatting to each other. Since only two online clients are left they can send a private and public message with out fearing some one else would see their message.

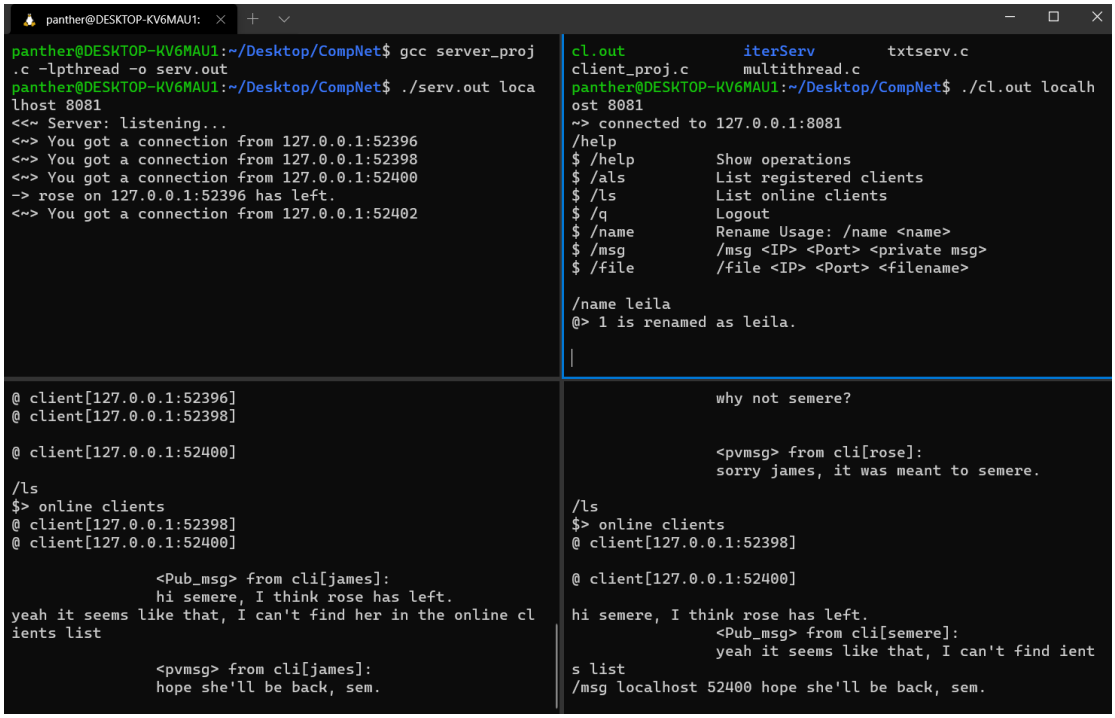


Figure -2.20: a new client, on the top-right sub-window, has registered to the server and renamed itself as 'leila'. Now, the server has four registered clients and three online clients.

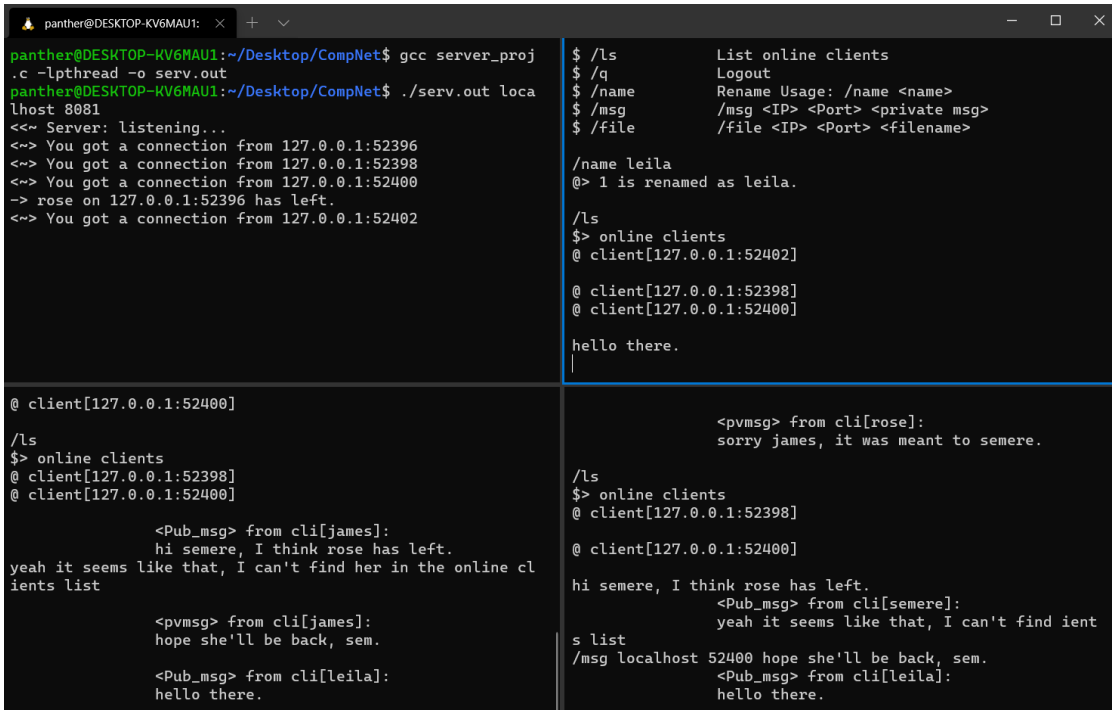


Figure -2.21: the new client, whose name is 'leila' has sent a public message. Before 'leila' sent the public message, she had typed the command '/ls' to see the list of online clients. This message can be seen on both the other two online clients.

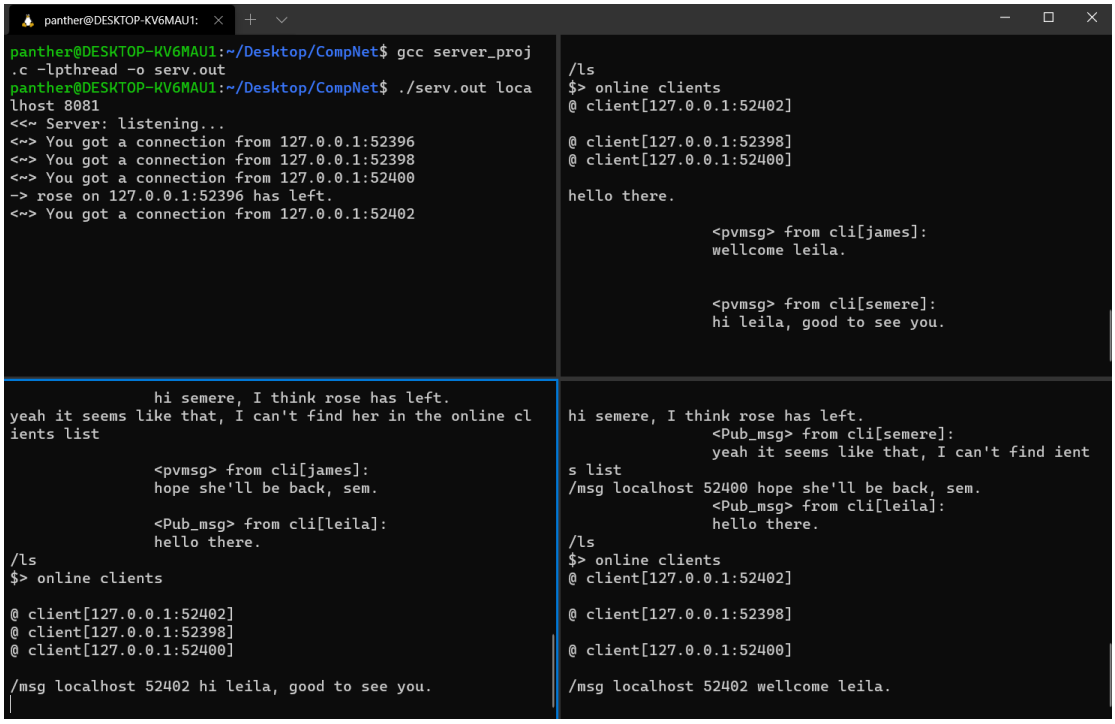


Figure -2.22: Leila, see the top-right sub-window, has received two private messages from the other two clients, from james and semere, on the bottom half of the window.

6.3.1. Sharing Files

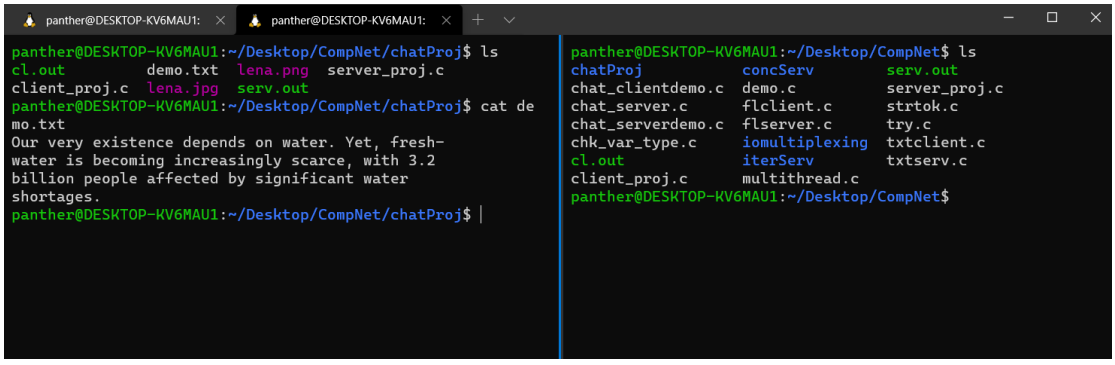


Figure -2.23: as we can see the '/CompNet/chatProj' directory, see the left half of the window above, has one text file ('demo.txt') and two image files ('lena.png' and 'lena.jpg'). but the 'CompNet' directory has no file. I will try to transfer a file from a client whose executable file is on the left-half to another client whose executable file is on the right half of the window (i.e. 'CompNet' directory).

As you can see it on the left-half of the sub-window above, the content of 'demo.txt' is displayed on the screen.

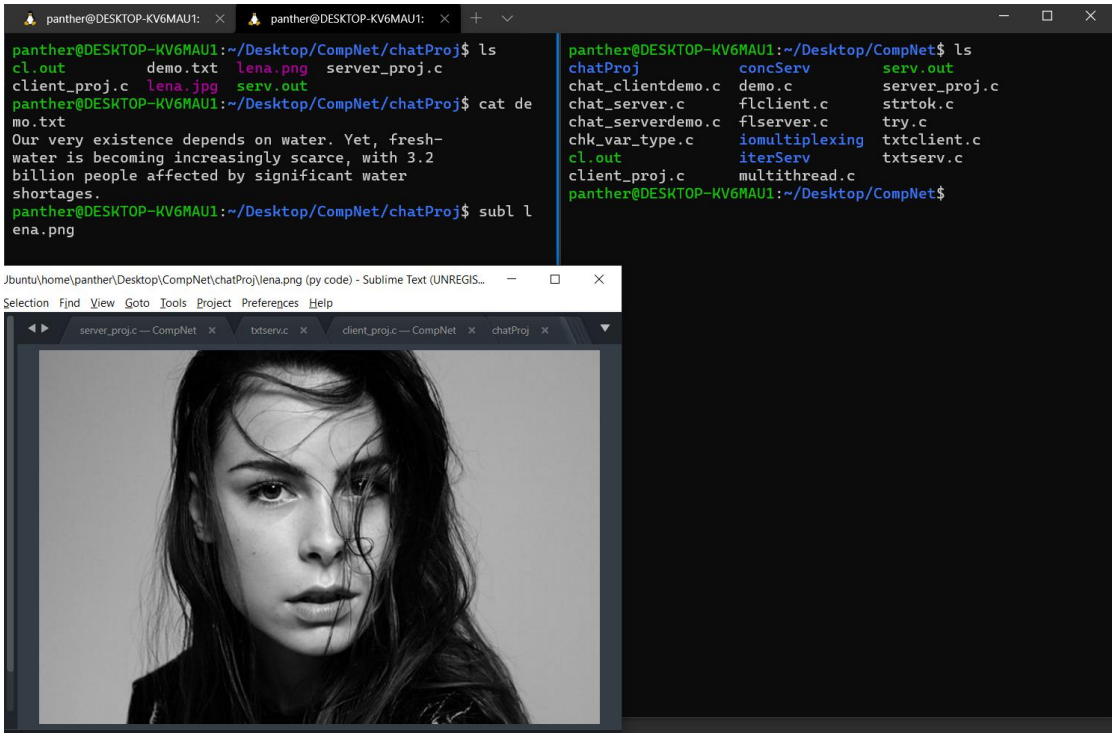


Figure -2.24: the image saved as 'lena.png' is displayed using sublime text editor. See the directory of the file to make sure 'lena.png' is indeed in the same directory (i.e. in 'CompNet/chatProj' directory).

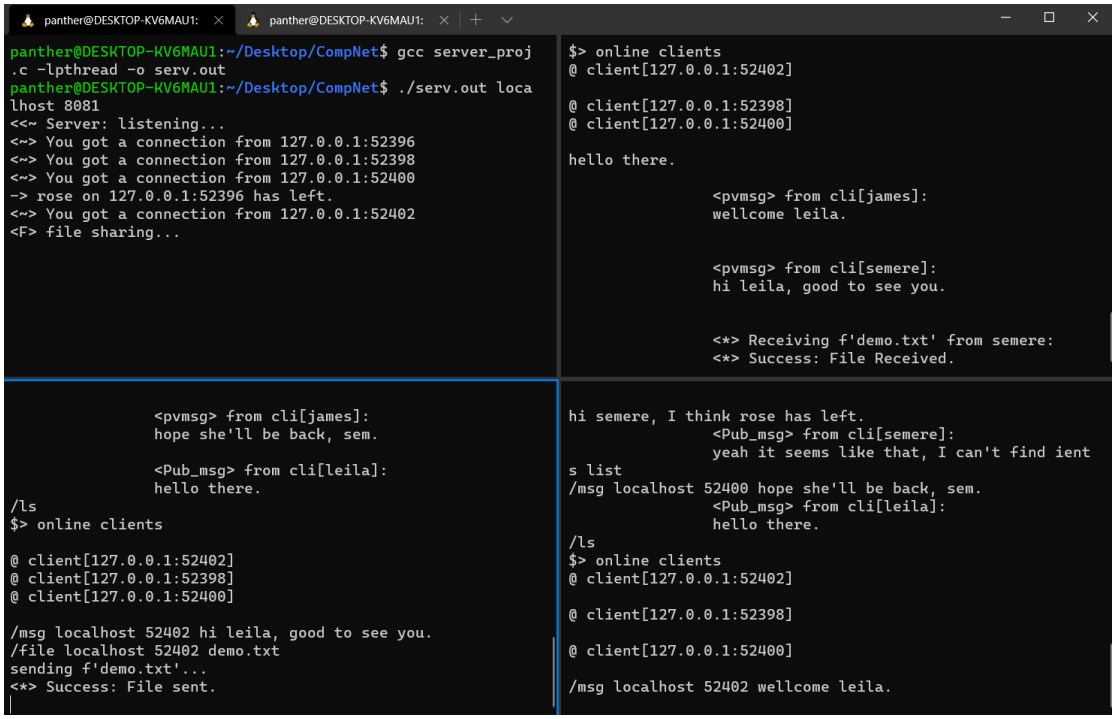


Figure -2.25: the client ('semere'), on the bottom-left sub-window and on the '/CompNet/chatProj' directory, has sent a text file to the client ('leila'), on the top-right sub-window and in the 'CompNet' directory, where the text file has not been before.

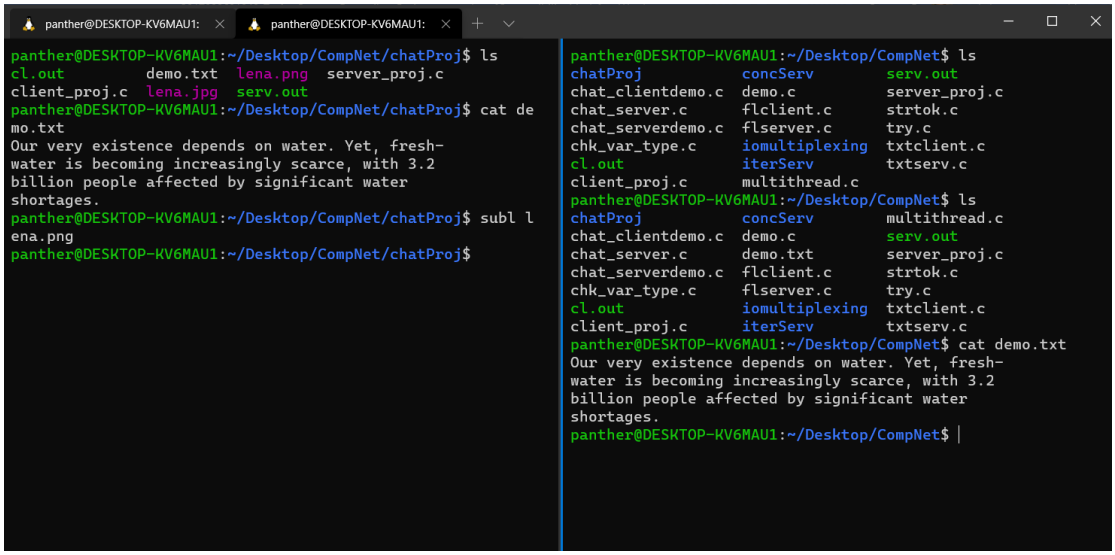


Figure -2.26: the text file ('demo.txt') has indeed been received by 'leila', whose executable file was in the 'CompNet' directory. See the left-half of the window to make sure if the text file is received.

The received file 'demo.txt' is, now, displayed on the screen from the 'CompNet' directory.

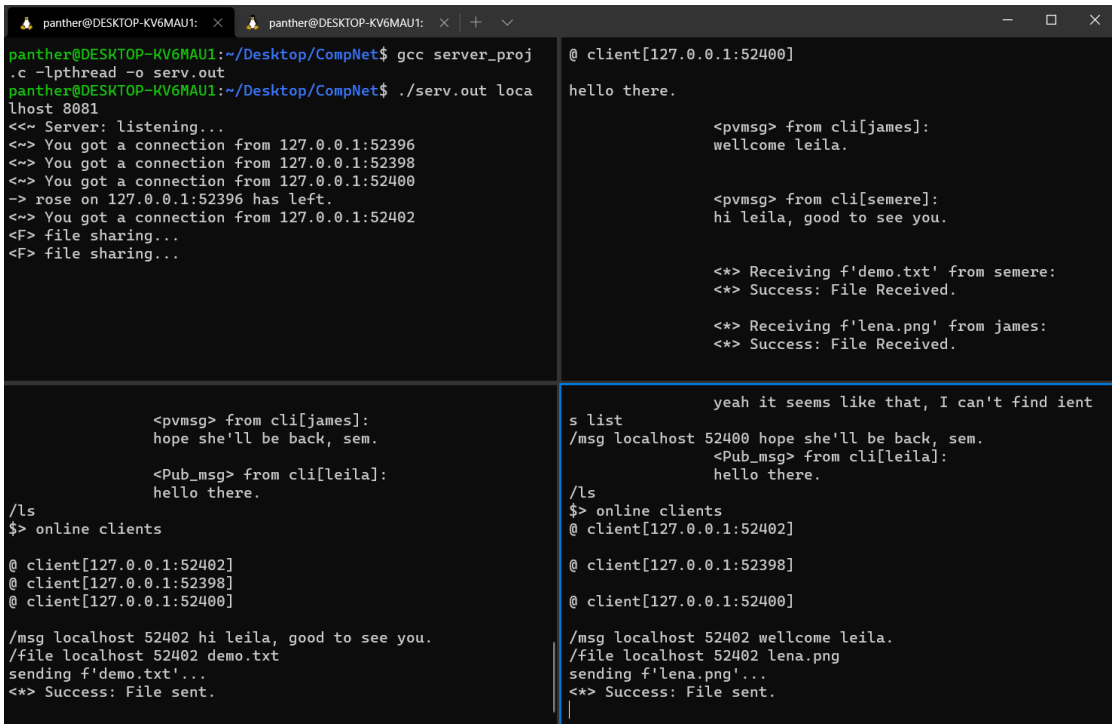


Figure -2.27: the client('james'), on the bottom-right sub-window, has sent an image file 'lena.png' to 'leila'(on the top-right sub-window), the client to whom the text file was sent to.

Note: an image has not been saved in the 'CompNet' directory, where leila's executable file is stored in.

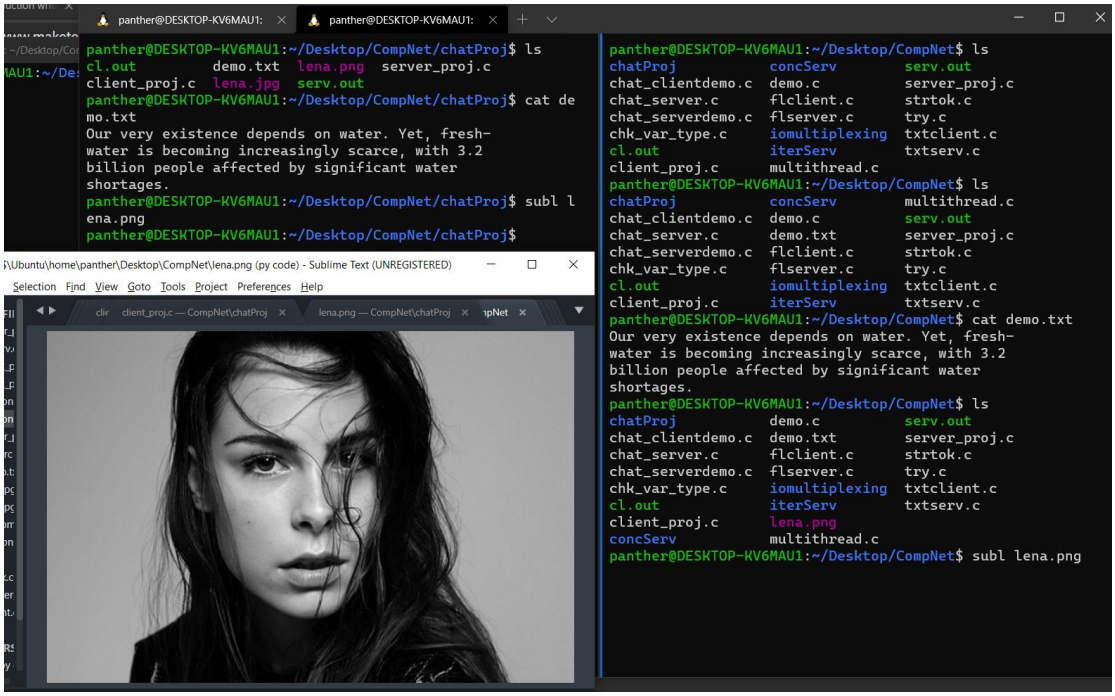


Figure -2.28: see at the list of files, in the 'CompNet' directory, on the left-bottom of the window. The image 'lena.png' has been received by Leila from james. Now the image is displayed on the bottom-left side of the window. See the directory of the displayed image to make sure the displayed image is indeed the one received from james, on the 'CompNet/chatProj' directory.

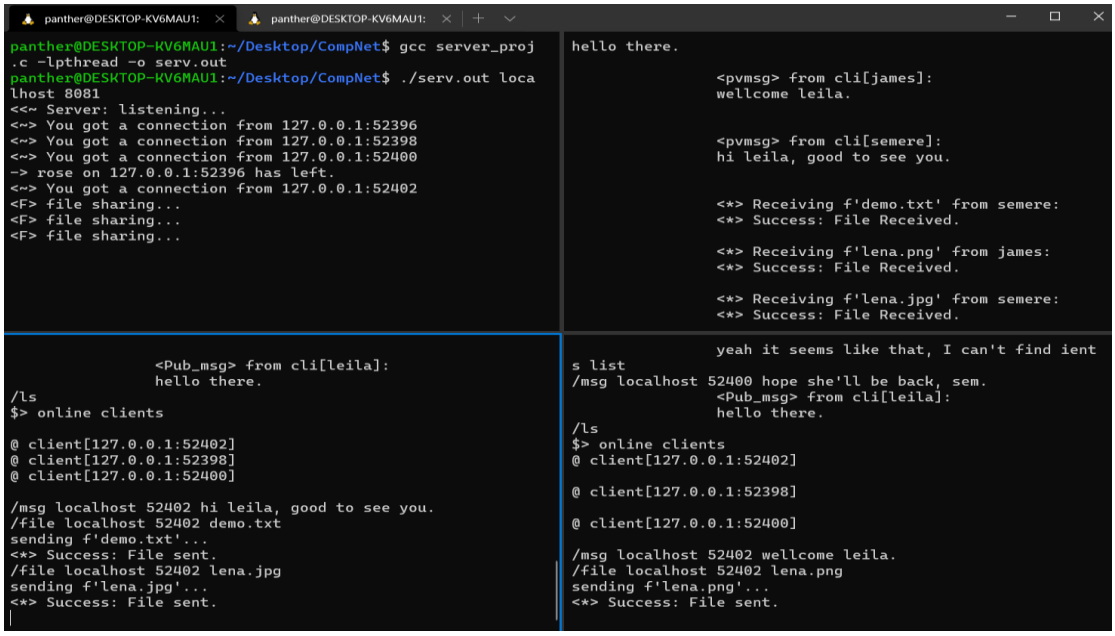


Figure -2.29: another the client('semere'), on the bottom-leftt sub-window, has sent an image file 'lena.jpg' to 'leila'(on the top-right sub-window), the client to whom the text file and image was sent to before.

Note: 'lena.jpg' has never been saved in the 'CompNet' directory, where leila's executable file is stored in.

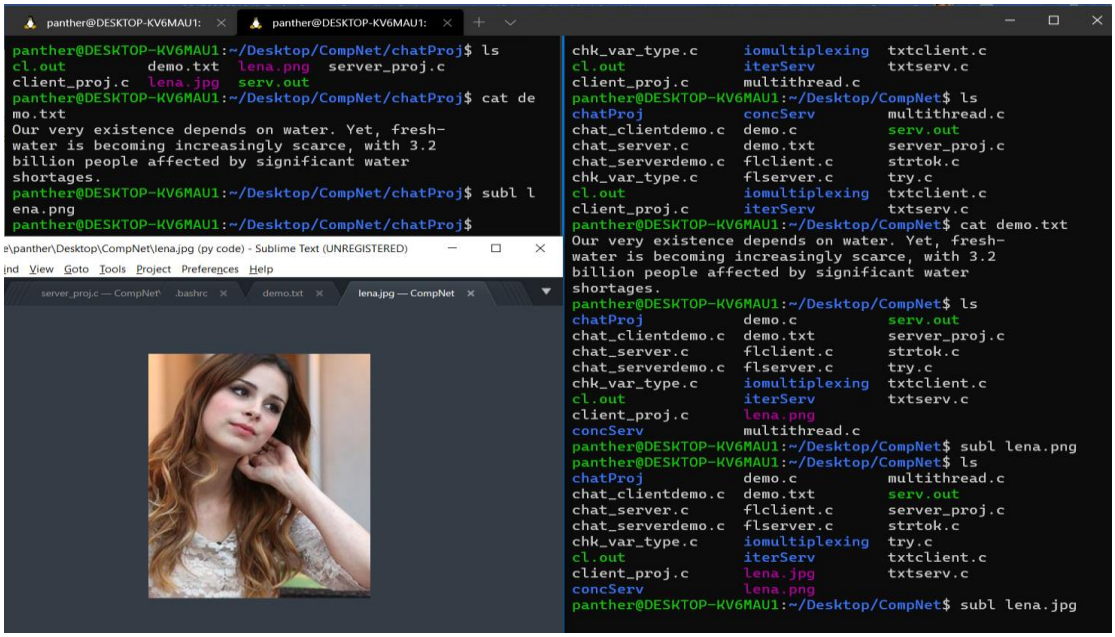


Figure -2.28: see at the list of files, in the 'CompNet' directory, on the left-bottom of the window. The image 'lena.jpg' has indeed been received by Leila from semere. Now the image is displayed on the bottom-left side of the window. See the directory of the displayed image to make sure the displayed image is indeed the one received from semere, on the 'CompNet/chatProj' directory.

Now, there are two image files and one text file in the '/CompNet' directory.

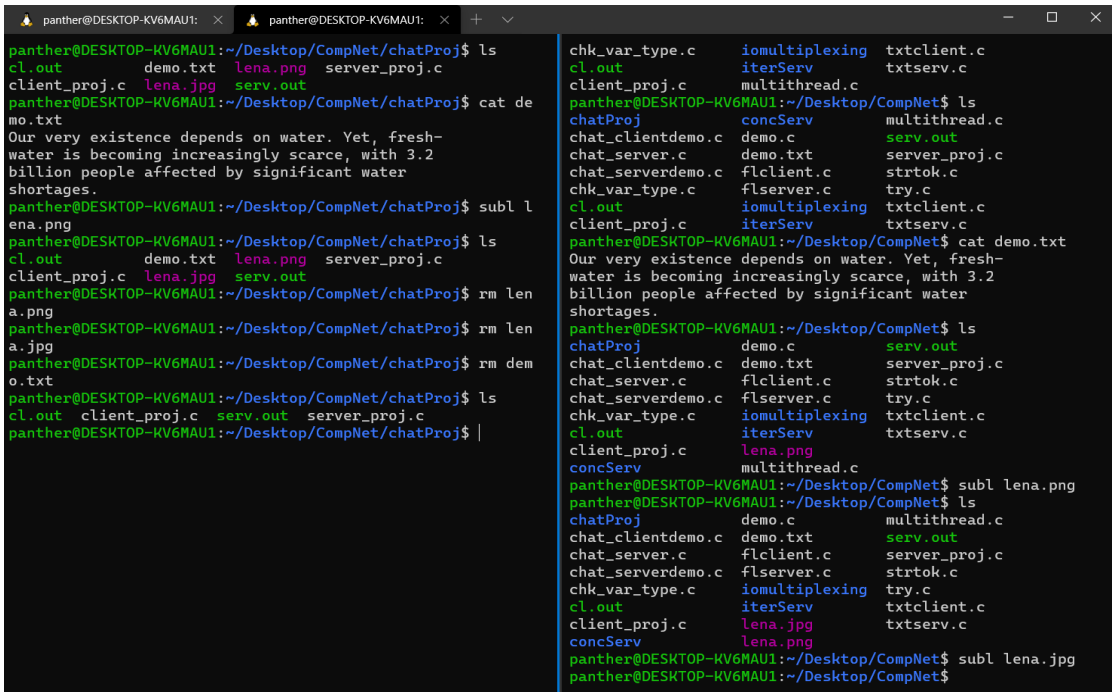


Figure -2.29: see the left-half of the window above. I have now removed all the image and text file from the '/CompNet/chatProj' directory so that I can send to that directory from 'CompNet' directory, the one I have so far been transferring the files to or where Leila has been receiving the files at.

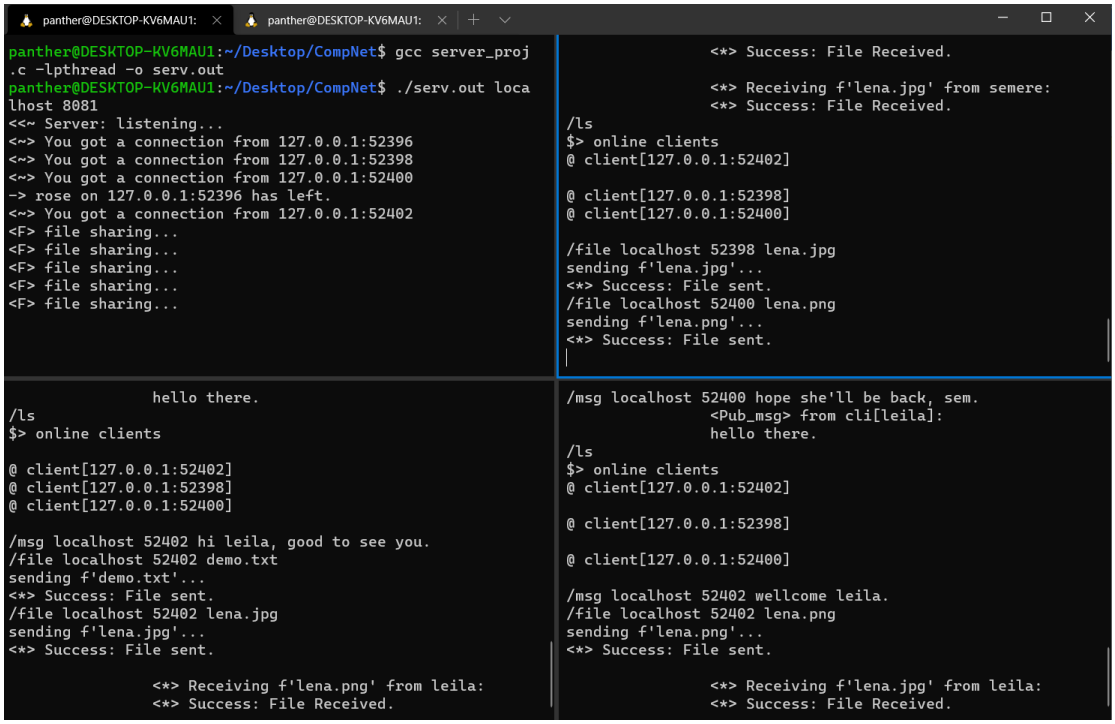


Figure -2.30: the client('leila'), on the top-right sub-window above, has sent the two image files to one to each of the other two clients (i.e. 'lena.jpg' to 'james' (see at the bottom-right sub-window) and 'lena.png' to 'semere'(see at the bottom-left sub-window)).

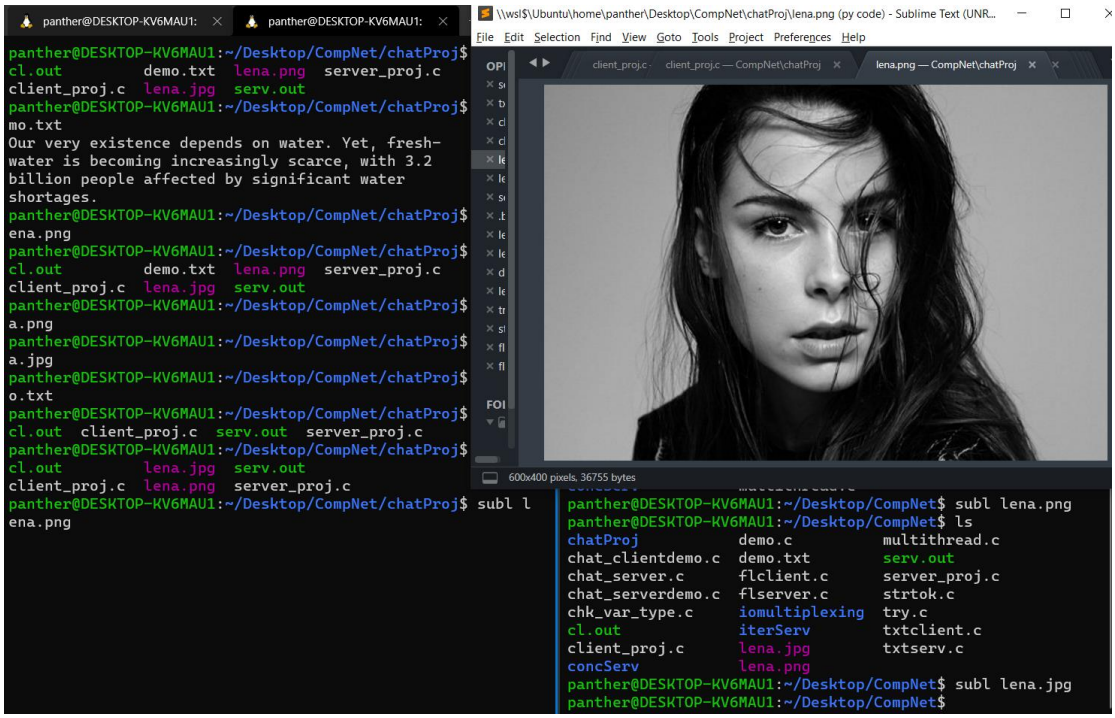


Figure -2.31: see the left-half of the window above. the two images have been received by each client; and since the two receivers are on the same directory, the images are stored in the same directory.

'lena.png' is displayed using sublime text editor as shown above.

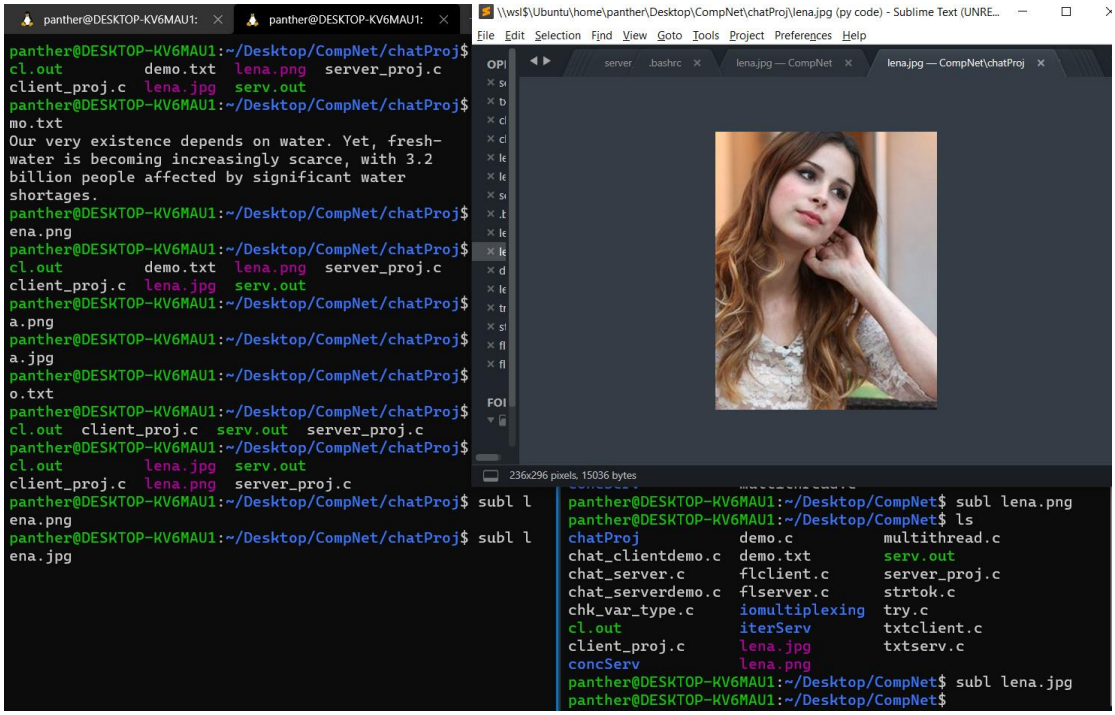


Figure -2.32: the second received image ‘lena.jpg’ is displayed using sublime text editor as shown above.

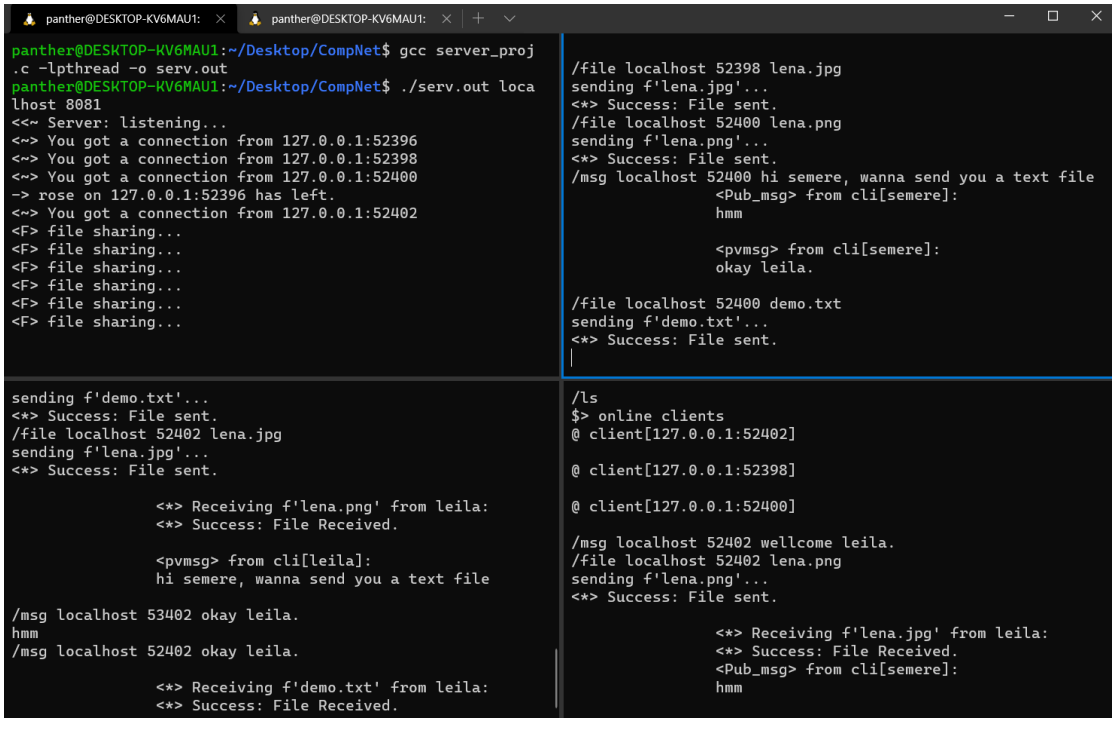


Figure -2.33: the client(‘semere’), on the bottom-left, has sent one public and one private, to the client on port:52402 or ‘leila’, messages. ‘leila’, see the top-right sub-window, then, has sent ‘demo.txt’ to ‘semere’ (see the bottom-left sub-window).

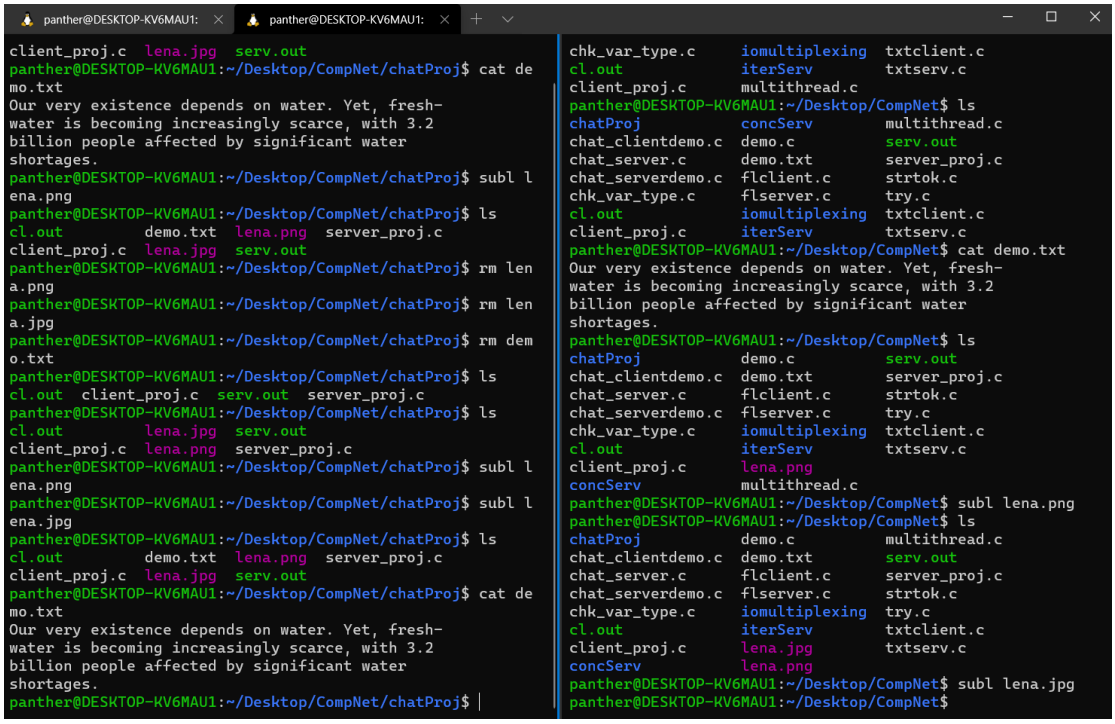


Figure -2.34: see the left-half of the window above. the text file 'demo.txt' received from 'leila' is now displayed on the window.

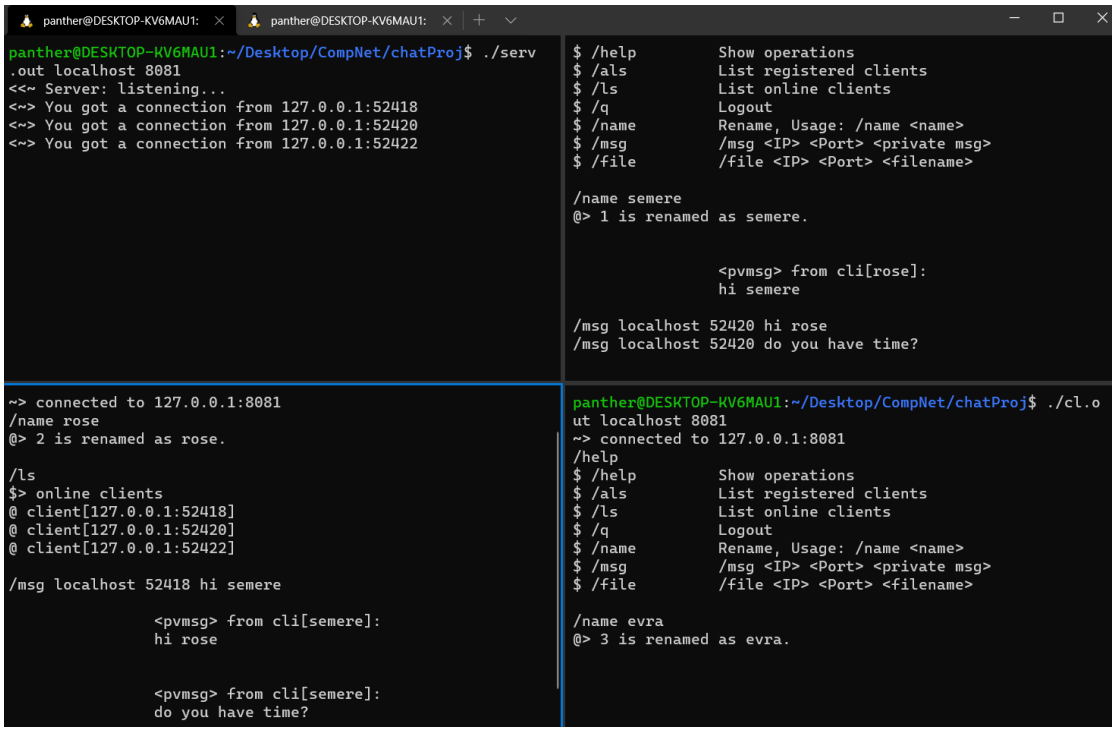


Figure -2.35: another three clients has registered. The two clients, on the left-half of the window, has asked for help (typing the command '/help') and then renamed them selves as 'semere' and 'evra'. The client on the bottom-left side has renamed itself as 'rose', then, seen the list of the online clients, hence '/ls', and send a private message to 'semere' on port 52418. thereafter '**semere**' sends two private messages to rose (on port 52420).

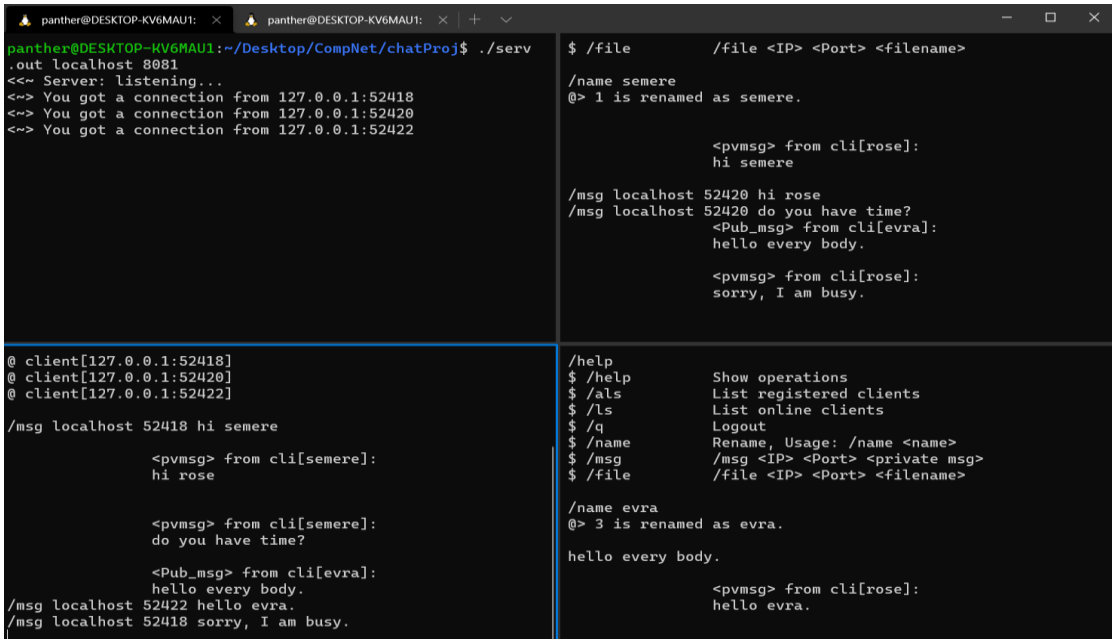


Figure -2.36: 'evra' (see at the bottom-right side) has sent a public message 'hello every body.' to 'rose' and 'semere'. 'rose' (see at the bottom-left side) has sent two private messages one to 'semere' and one to 'evra' (see at the bottom-right side).

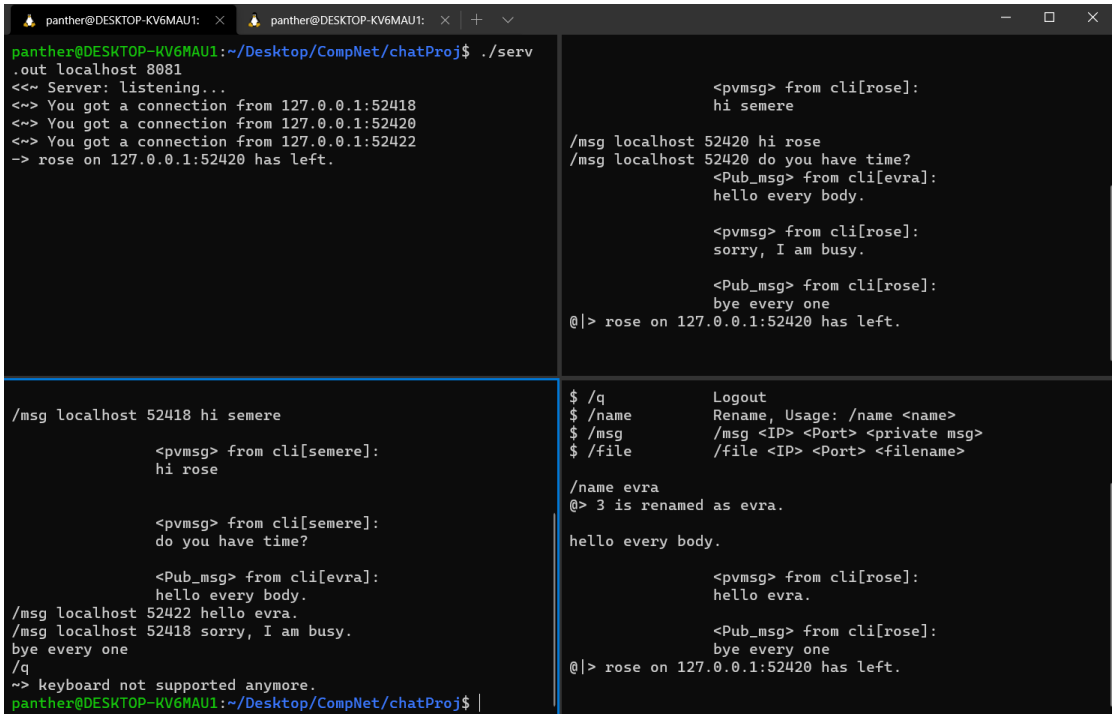


Figure -2.37: 'rose' (see at the bottom-left side) has quit the chat by typing the command '/q'. the other online clients have received a notification that 'rose' has left. i.e. the notification '@|> rose on 127.0.0.1:52420' has left is sent to the other parties as can be seen in the right-half of the window.

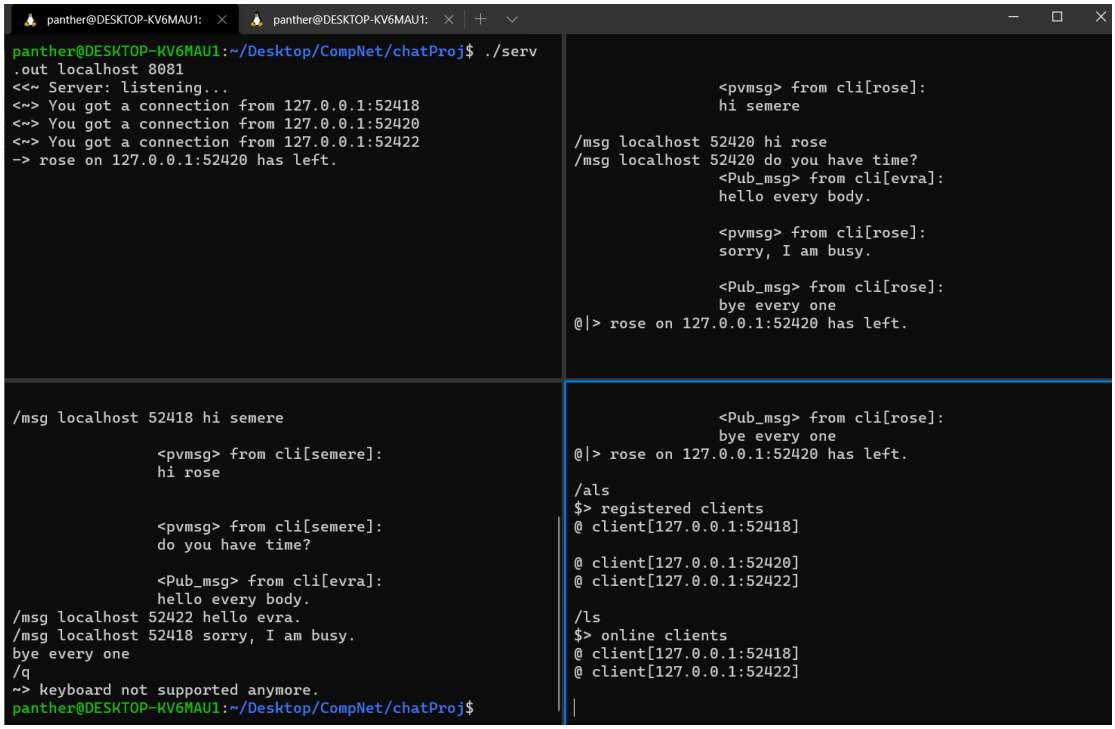


Figure -2.38: 'evra' (see at the bottom-right sub-window) has checked the registered clients and the online clients by typing '/als' and '/ls' respectively. Since the third client, whose name is 'rose', has left the group chat, only two clients are online as shown in the bottom-right sub-window above.

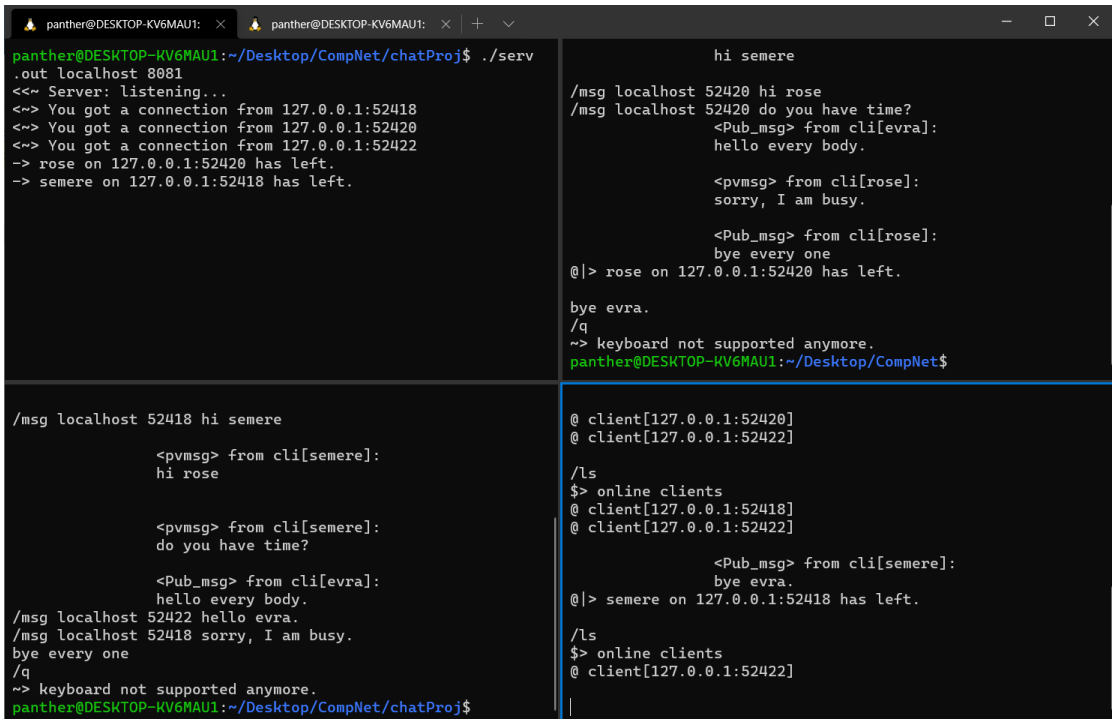


Figure -2.39: the client ('semere' on the top-right side) has sent a message 'bye evra' to the other client and, then, left the group. The last client, left online, has received a notification from the server and, then, typed '/ls' to see the list of online clients. Thereafter, we can see that only one client is left.

7. Summary

The server I have implemented supports **IPv4** and **IPv6** connection since the socket of the server is created using **getaddrinfo**: therefore, clients can use either **IPv4** or **IPv6** to connect to the server. ***the network host, server, is multihomed, accessible over multiple protocols.*** I have implemented private message transfer between two clients, public message transfer or public chat room, file sharing, including text file and image file sharing, between two clients. The Usage of the code has been made clear by providing certain commands to a user: once the client types or sends '/help' to the server, the server provides the client with a bunch of commands the client can use. The way how a text, string, is printed in the standard output, screen, is also made to look more beautiful and readable by an average user.

The client can receive multiple messages, private and public, from different or same clients. It can also send text messages, private and public, share files to different clients or the same client. The use of select on the server side and the client side prevents the server and client from blocking the program: the select API monitors the file descriptors that it is provided with to watch, then returns when a status of one or more file descriptors is changed.

8. Conclusion and Experience

8.1. Conclusion

In this project, a real-world chat room, that supports private and public message transferring and file sharing is implemented using socket programming in C. my server supports both IPv4 and IPv6 internet connections.

select API is used to monitor the status of multiple file descriptors in both end points as follows:

- ✧ Server side: select watches file descriptors of the client for reading new data arrivals and file descriptor of the server for reading new incoming request from a client.
- ✧ Client side: select watches stdin, fd = 0, file descriptor to see if a data is available from a key board and watches the file descriptor of the server to see for an incoming data.

8.2. Experience

I have gotten an irreplaciable experiences, such as:

- I am, now, used to computer network programming. Now I have an idea of how to implement real world chat application using socket programming.
- I have improved my skill on c programming.
- I know well how to debug using gdb debugger in terminal.
- I am familiar with ubuntu Linux distro.
- I have a basic idea of how operating system works and I am now more familiar to the two IP protocols (i.e., IPv4 and IPv6).

THANK YOU