COMP 4981 - Chat Program

Assignment #3 User Manual

April 8, 2020

Team Members

Patrick Wong

Mikhaela Layon

Table of Contents

Introduction

ChatProgram is a simple client/server application. Users can send and receive messages to each other through a Terminal interface. TCP is the protocol used to handle the connections in the application.

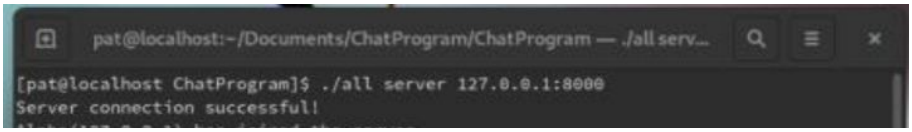
The purpose of the server is to handle connected sockets. The server maintains a list of connected sockets, receives incoming messages and sends these messages according to the list. The client establishes a connection to the server, sends messages to the server, and reads continuously for responses from the server.

Select() and system calls are used in both the client and server side. On the server side, both types of calls are used to service and multiplex incoming messages. On the client side, these calls are used to prompt the use for input and handling the “save conversation” command in a non-blocking manner.

Instructions

Server

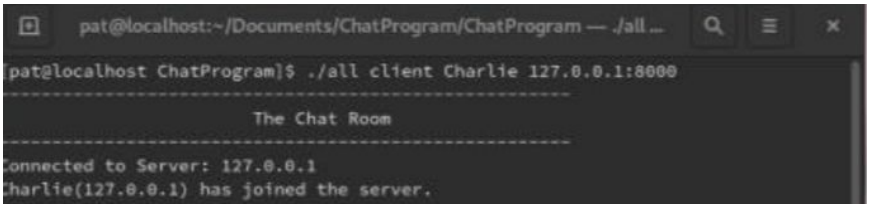
1. Go to the directory that contains the source files.
2. Open the Terminal in the directory
3. The usage pattern to run the server is as follows:
   ./all server <ip_address>:<port>
4. The example below runs the server at host 127.0.0.1, port 8000. The server can now listen for incoming connections.
5. The server can support up to 8 client connections.



Client

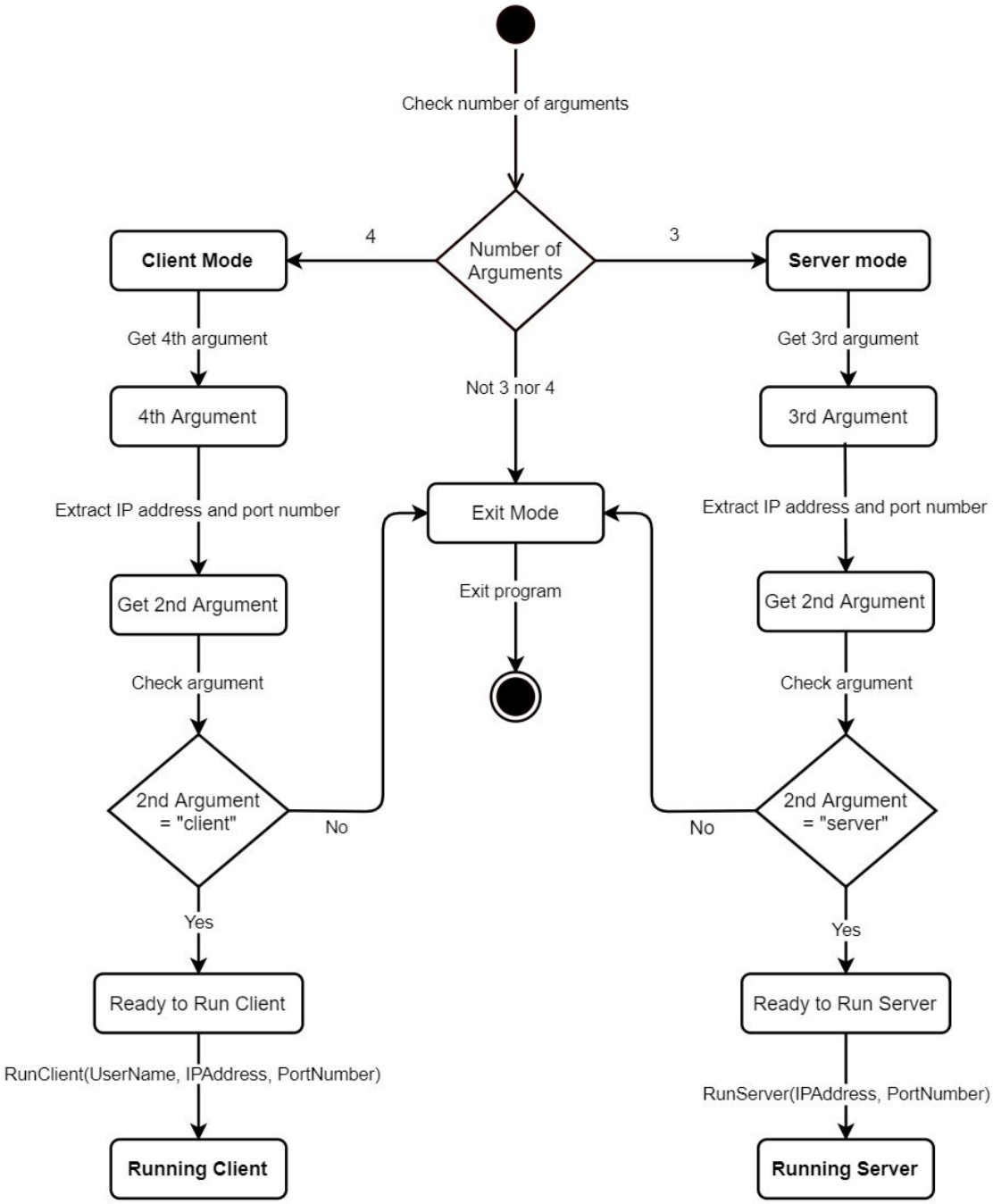
1. Go to the directory that contains the source files.
2. Open the Terminal in the directory
3. Make sure that the server is running.
4. The usage pattern to run the server is as follows:

   /all client <username> <ip_address>:<port>
5. The example below runs the client at host 127.0.0.1, port 8000. The username of the client is Charlie. Charlie now can send messages to the server.



State Diagram

Application



Application State Overview

Note that the first argument is the name of the executable.

Client Mode
In this state, the program will check if the IP address and the port number are valid before running the client. The third argument will be used as the username/alias of the client, and the argument does not require validation.

Server mode
In this state, the program will check if the IP address and the port number are valid before running the server.

4th Argument (Client Mode side)
The 4th argument is a string that contains the IP address and the port number delimited by a colon (:) character. For example, “127.0.0.1:8000” is a valid 4th argument. The program will extract the IP address and port number from the string.

2nd Argument (Client Mode side)
The program checks if the argument is “client.” If this is true, then the client is ready to behave as a client process. Otherwise, the program exits.

Ready to Run Client
The program is ready to run as a client. The username, IP address and port number are supplied to run this process.

3rd Argument (Server side)
The 3rd argument is a string that contains the IP address and the port number delimited by a colon (:) character. For example, “127.0.0.1:8000” is a valid 4th argument. The program will extract the IP address and port number from the string.

2nd Argument (Server side)
The program checks if the argument is “server.” If this is true, then the client is ready to behave as a server process. Otherwise, the program exits.

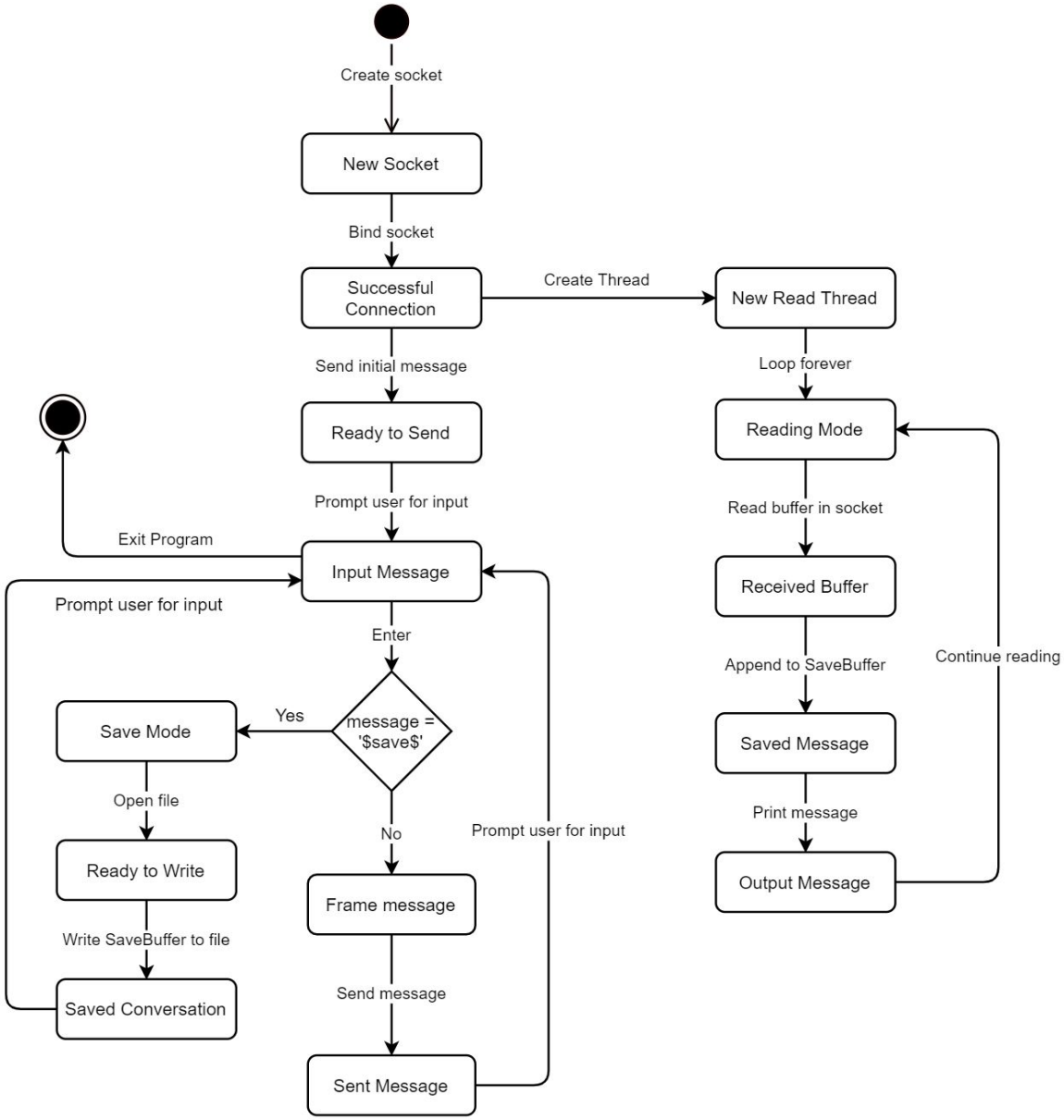
Ready to Run Server
The program is ready to run as a client. The IP address and port number are supplied to run this process.

Running Client
Given a valid IP address, a valid port number, and a username, the program will run as a client.

Running Server
Given a valid IP address, and a valid port number, the program will run as a server.

Exit Mode
If there is an error when verifying the context of the application, then the program terminates.

Running Client



Running Client State Overview

New Socket
The client creates the socket descriptor.

Successful Connection
In this state, the socket is bound to the IP address and port via TCP. The socket can read/write incoming data.

New Read Thread
A new thread is created to read continuously for data in the socket.

Reading Mode
The client reads continuously for data in the socket.

Received Buffer
The client in this state has the contents of the buffer from the socket.

Saved Message
The SaveBuffer contains all incoming messages. Whenever a message is read from the socket, the client will append the message to the SaveBuffer.

Output Message
The incoming message is displayed to the Terminal, so that the user can see what other clients are talking about.

Ready to Send
After the client sends an initial message to the server, then the client is ready to send additional messages to the server to engage with the conversation.

Input Message
The client is prompted to send a message.

Save Mode
When the client enters a message, then the program will check if the message is *$save$*. If this is true, then the program will enter a save mode. In this mode, the program will open a file and write the contents of SaveBuffer to the file. The file name will be called *chatlog.txt*.

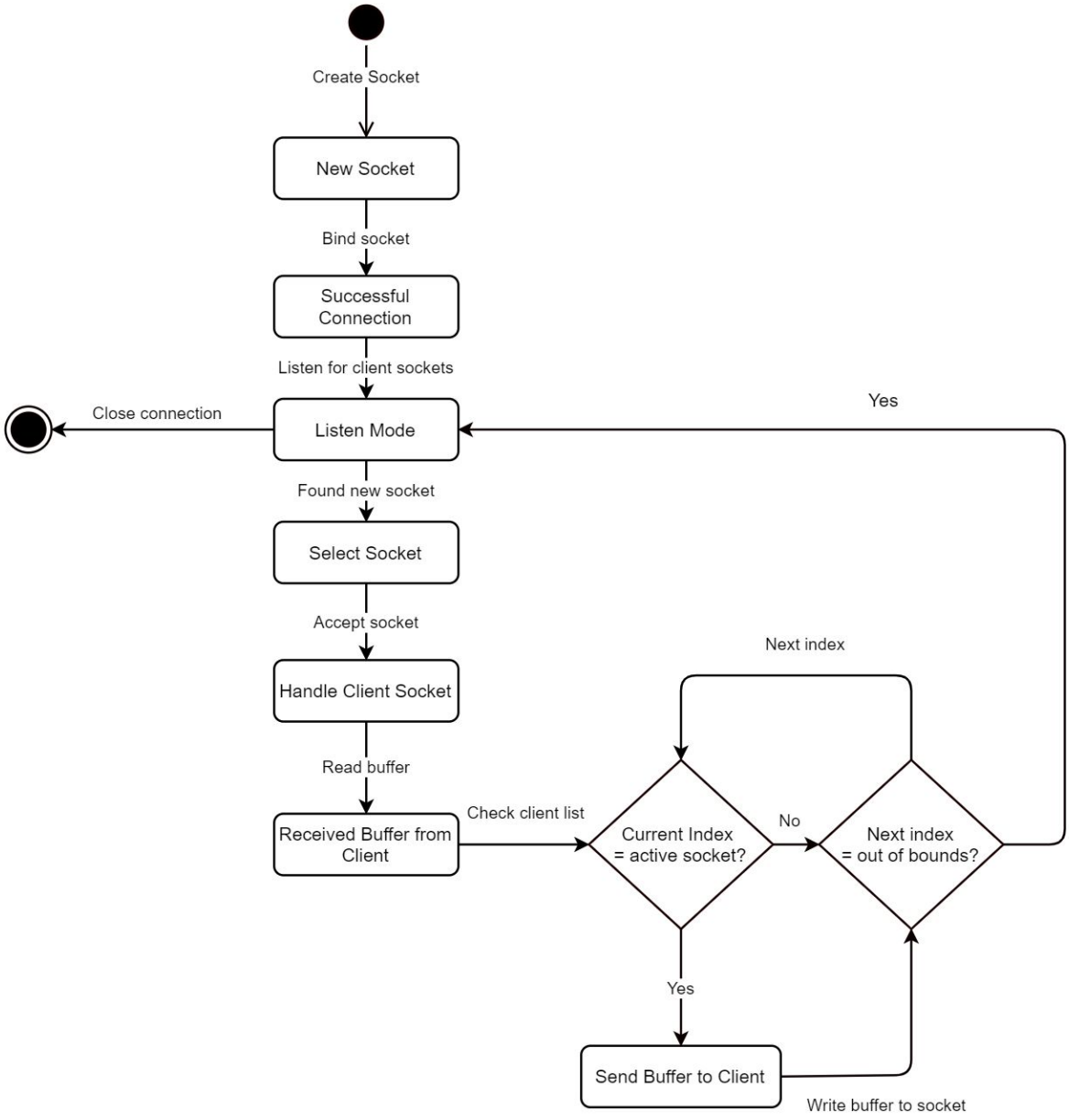
Ready to Write
Once the chatlog.txt file is opened, then the program will write the contents of SaveBuffer to the file.

Saved Conversation
The file, chatlog.txt, contains the contents of the saved conversation developed by the read thread. The program will prompt the user to enter the next message.

Frame Message
The program will decorate the message to include the username of the client.

Sent Message
The client will send the framed message to the server.

Running Server



Running Server State Overview

New Socket
The server creates the socket descriptor.

Successful connection
In this state, the socket is bound to the ip address and port via TCP. The socket can listen for incoming connections.

Listen mode
The server will listen for incoming client connections.

Select socket
Using the multiplex I/O model, the server keeps a list of client sockets. If a client socket has data in the buffer, then the server is notified and will service that particular client.

Handle Client Socket
The server will attempt to read and consume the buffer from the selected client socket.

Received Buffer from client
Once the server receives the buffer from the client, then the server will send the same message to each connected client. To establish that, the server will iterate the list, checking if the slot is an active connection. If the slot is an active connection, then the server will send the message to that client. Once the server runs through the list of clients, then the server will listen for the next incoming active socket connection.

Pseudocode

Application

Main

```
If argument count is 3
     Tokenize third argument
     Assign IP Address to first token
     Assign Port number to second token
     Get second argument
     If second argument is not “server”
          Print error message to Terminal
          Exit program
     RunServer(IP Address, Port Number)

Else if argument count is 4
          Assign Username to third argument
          Tokenize fourth argument
          Assign IP Address to first token
          Assign Port number to second token
          Get second argument
          If second argument is not “client”
               Print error message to Terminal
               Exit program
          RunClient(Username, IP Address, Port Number)

Else
     Print error message to Terminal
     Exit program
```

Running Client

Client

```
Create new socket
Fill socket structure with IP address and Port number
Connect to server socket
Create SaveBuffer
Run ReadThread
Create initial message
Send initial message to server socket

Loop forever
     Prompt user for message
     If message == '$save$'
          Open chatlog.txt file
          Write SaveBuffer to file
          Print save message to Terminal
          Close file
     Else
          Include Username to message
          Send message to server socket
```

ReadThread

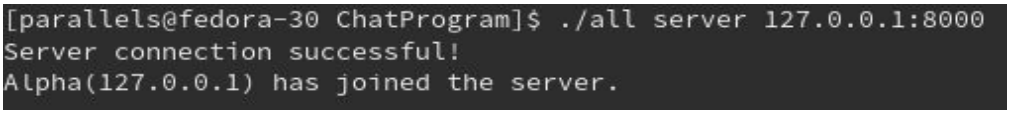
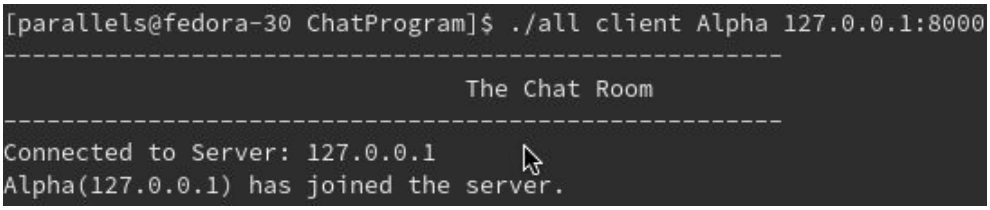
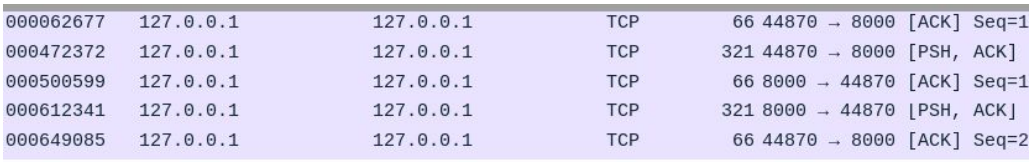
```
Loop forever
     If there is buffer from socket
          Receive buffer
     Append buffer to SaveBuffer
     Print buffer to Terminal
```

Running Server

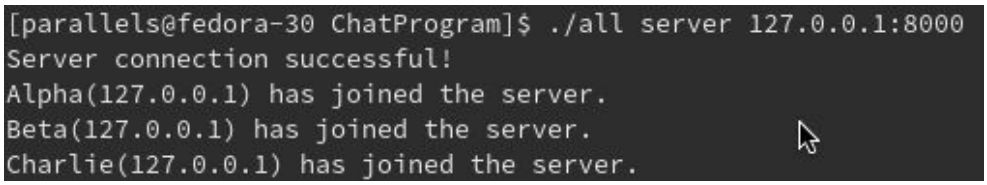
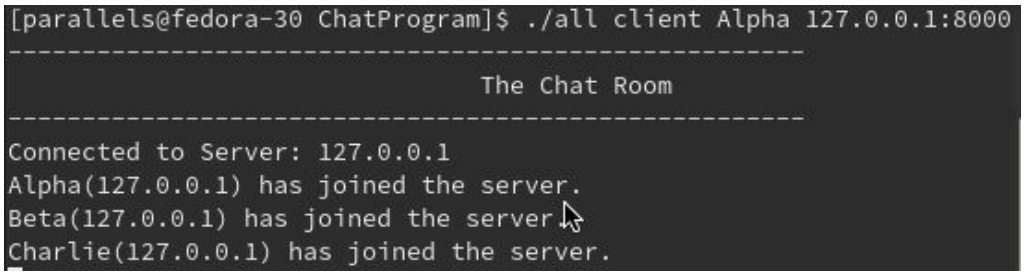
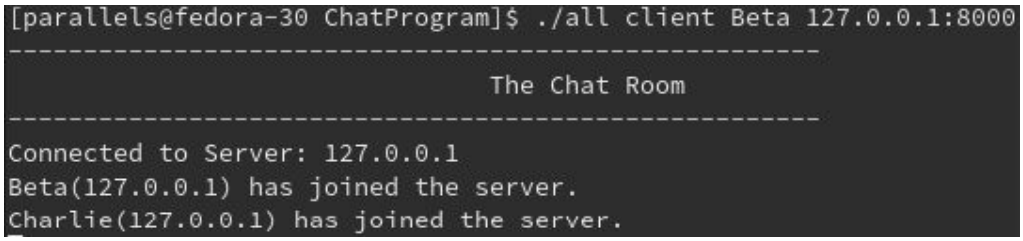
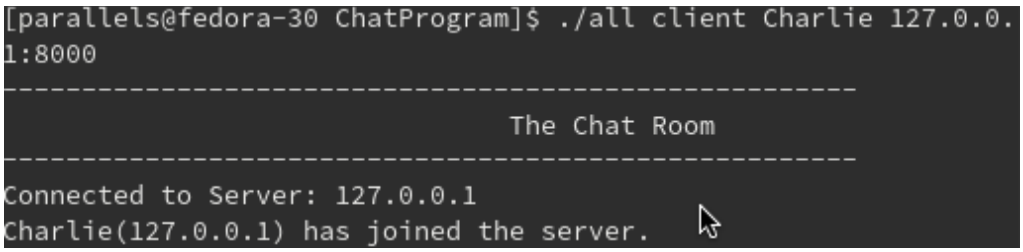
Server

```
Create socket
Fill socket structure with IP address and Port number
Bind socket structure to socket
Listen for connections
Initialize client list
Loop forever
     Select active socket
     Loop through client list
          If slot is empty
               Set active socket to slot
     Loop through active sockets
          If socket has data
               Accept socket
               Read buffer from socket
          Print buffer to Terminal
          Write buffer to all active sockets
```

Test Cases

Test Case #	1
Scenario	Server handling a single client
Procedure	● Run server ● Run client
Expectation	The server terminal should be able to show the client connected to it. Subsequently, the client terminal should be able to show which server it is connected to.
Screenshot	Server  Client  Wireshark (3 way handshake) 
Result	Success messages on both client and server terminals. Wireshark displays 3 way hand shake occuring.
Passed (Y/N)?	Y

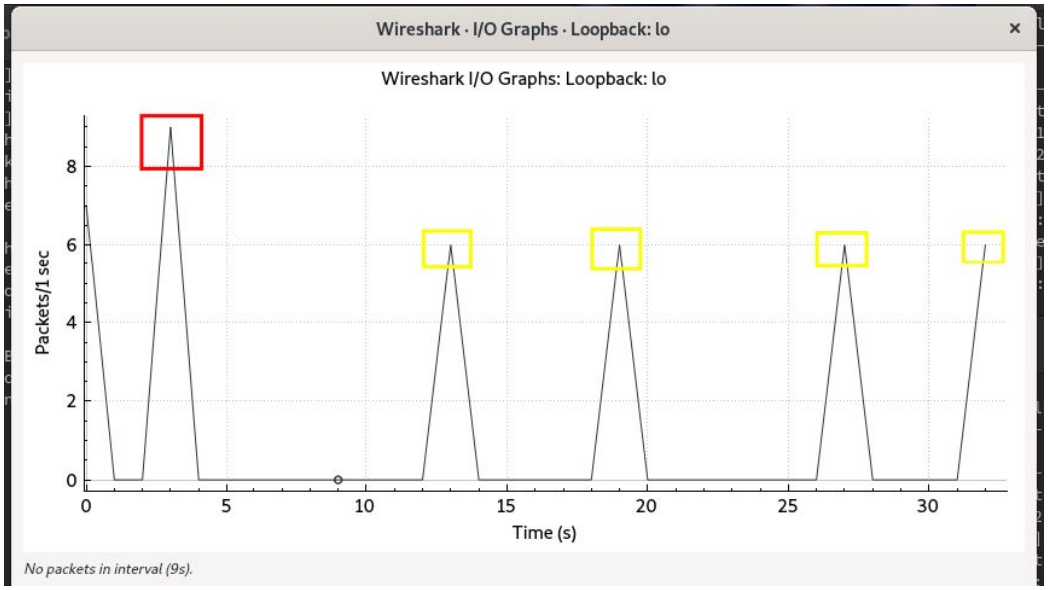
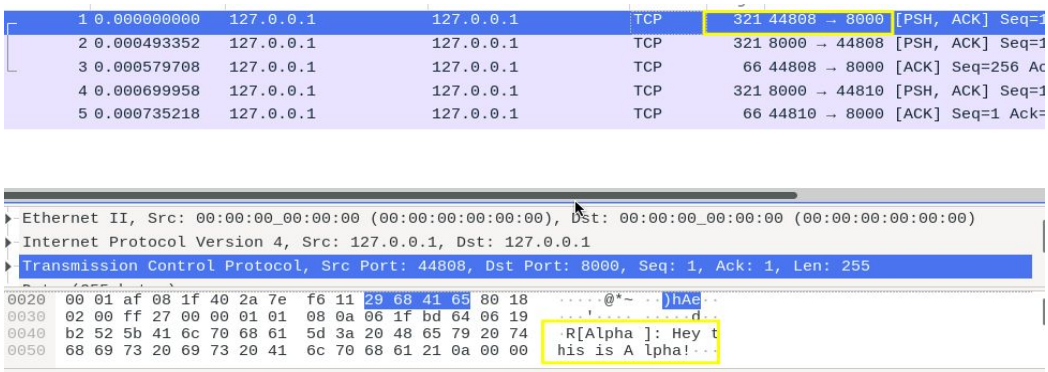
Test Case #	2
Scenario	Server handling multiple clients
Procedure	● Run server

	● Run client 1 ● Run client 2 ● Run client 3
Expectation	The server terminal should be able to show all clients connected to it. Subsequently, the client terminals should be able to show which server it is connected to, as well as the clients connected to the same server.
Screenshot	Server  Client 1  Client 2  Client 3 

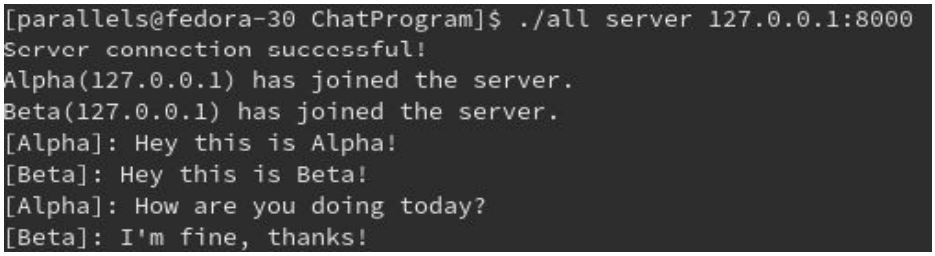
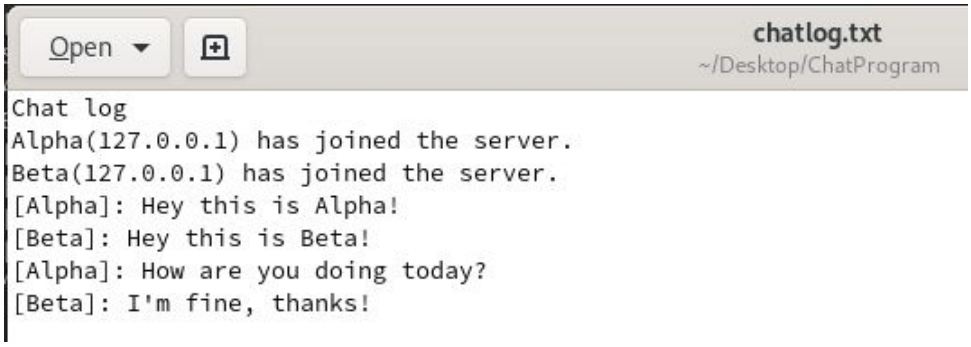
Result	Success messages on both client and server terminals.
Passed (Y/N)?	Y

Test Case #	3
Scenario	Server broadcasting to all clients
Procedure	• Type any message on a client • Press enter
Expectation	The server terminal should be able to show the message sent by a client to the rest of the clients connected to it.

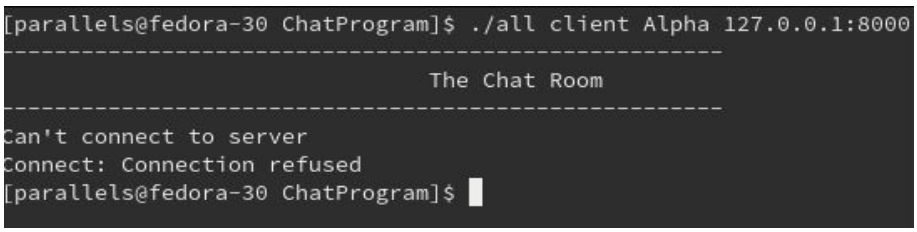
Screenshot	Server
	`[parallels@fedora-30 ChatProgram]$ ./all server 127.0.0.1:8000`
	`Server connection successful!`
	`Alpha(127.0.0.1) has joined the server.`
	`Beta(127.0.0.1) has joined the server.`
	`[Alpha]: Hey this is Alpha!`
	
	Client 1 (message sender)
	`Connected to Server: 127.0.0.1`
	`Alpha(127.0.0.1) has joined the server.`
	`Beta(127.0.0.1) has joined the server.`
	`Hey this is Alpha!`
	`[Alpha]: Hey this is Alpha!`
	
	Client 2
	`Connected to Server: 127.0.0.1`
	`Beta(127.0.0.1) has joined the server.`
	`[Alpha]: Hey this is Alpha!`
	
	Wireshark (I/O Graph)

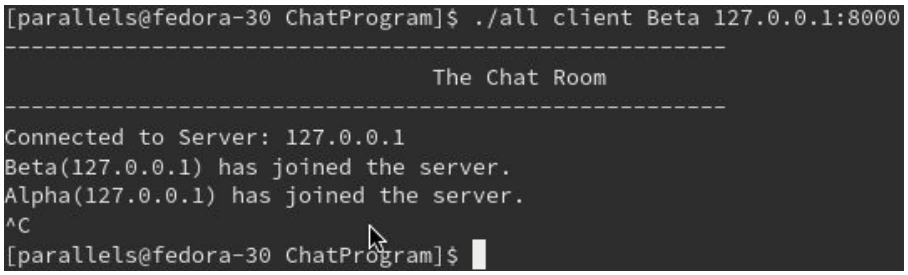
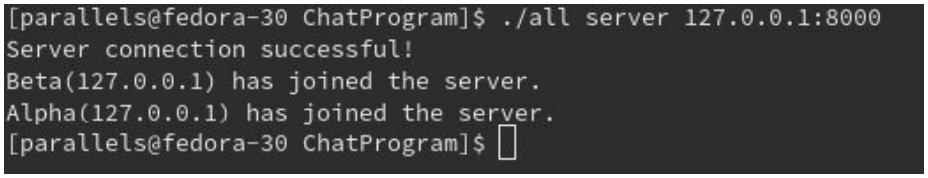
	 In this screenshot, the yellow peaks are ACKs received for sending messages between clients, while the red peak is the ACK received for establishing a connection between server and clients Wireshark packet details: 
Result	Client receives echoed message. Server displays message received. Wireshark displays the message being received by the server.
Passed (Y/N)?	Y

Test Case #	4
Scenario	Saving chat log into a file
Procedure	• Type $save$ on a client terminal • Open chat log from project directory

Expectation	The client terminal should be able to show a visual confirmation that the chat log has been saved. The chat log is then contained on a text file in the project directory.
Screenshot	Server  Client 1  Text File 
Result	Success message from client, and a file called chatlog.txt containing the chat log is created.
Passed (Y/N)?	Y

Test Case #	5
Scenario	Client unable to find server
Procedure	• Run client with unknown server address
Expectation	The client terminal should be able to show an error message when an entered server address can't be found.

Screenshot	
Result	Program ends automatically
Passed (Y/N)?	Y

Test Case #	6
Scenario	Client disconnects after connecting to server
Procedure	• Run server • Run client 1 • Run client 2 • Ending client 1
Expectation	The server terminal should be terminated when a client has disconnected to the server.
Screenshot	Client  Server 
Result	The server terminal ends automatically
Passed (Y/N)?	Y