**ICSI 500 - Project 3**

Jacob Clouse

Due December 2nd, 2023

Project 3 Writeup

|  |
| --- |
| **NOTE:**  Please watch my YouTube video demo before grading -> <https://youtu.be/trIiaS1dKrQ> |

# Table of Contents:

[Table of Contents: 1](#_Toc109059800)

[High-level Data Flow Diagrams 2](#_Toc1677889201)

[List of Routines and their Brief Descriptions 6](#_Toc1570071622)

[Implementation Details: 9](#_Toc987318754)

[Test Documentation 11](#_Toc1706571754)

[Testing Outputs: 12](#_Toc329072916)

[User Documentation – Sources 13](#_Toc112490706)

[How To Run My Program: 15](#_Toc1118039262)

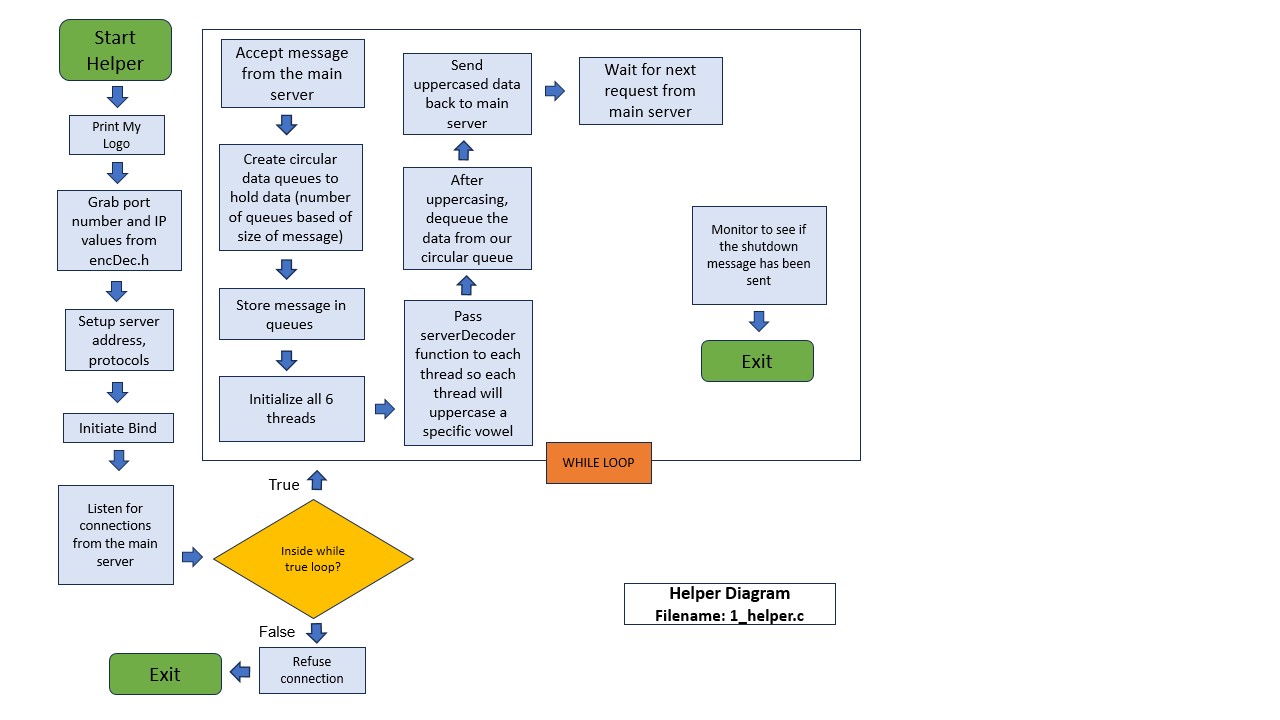
[Describe parameter (if any): 20](#_Toc900613282)

-

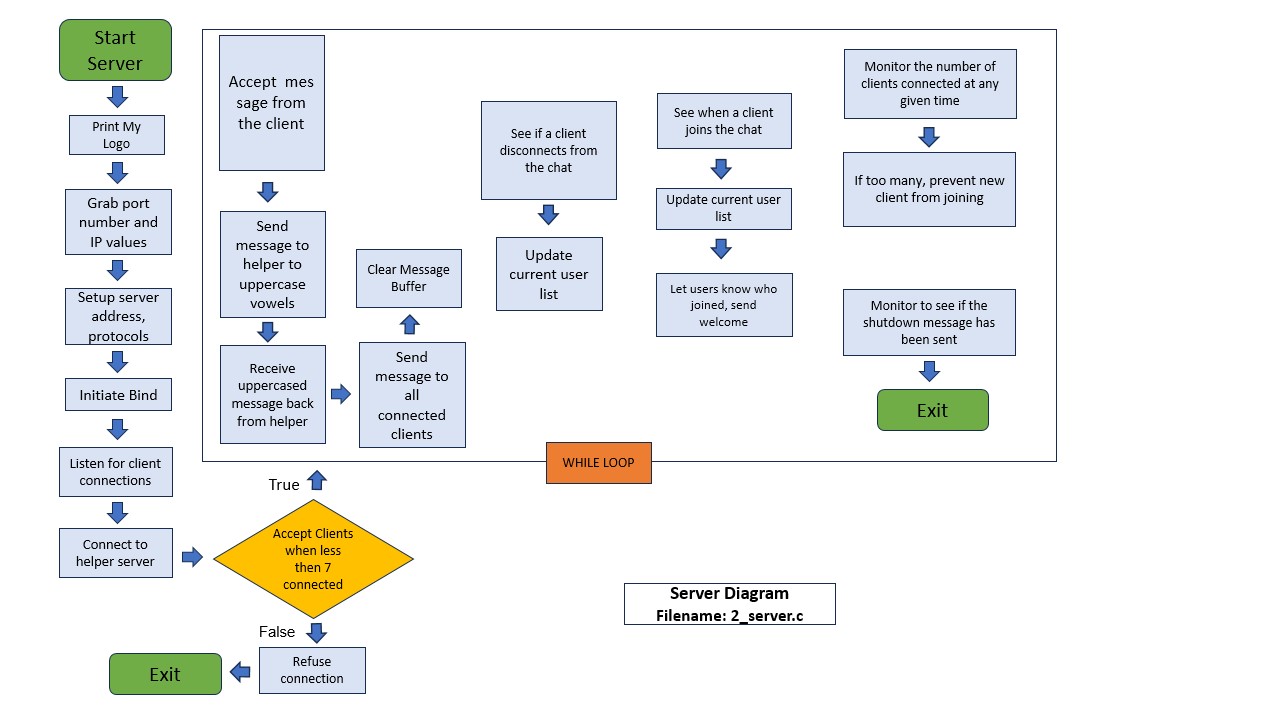
# High-level Data Flow Diagrams

NOTE: Since we built project 3 based off project 2, the diagrams for 3\_client.c and 2\_server.c will be very similar to their counterparts from project 2.

**Diagram for 1\_helper.c (Helper / Uppercase Program):**

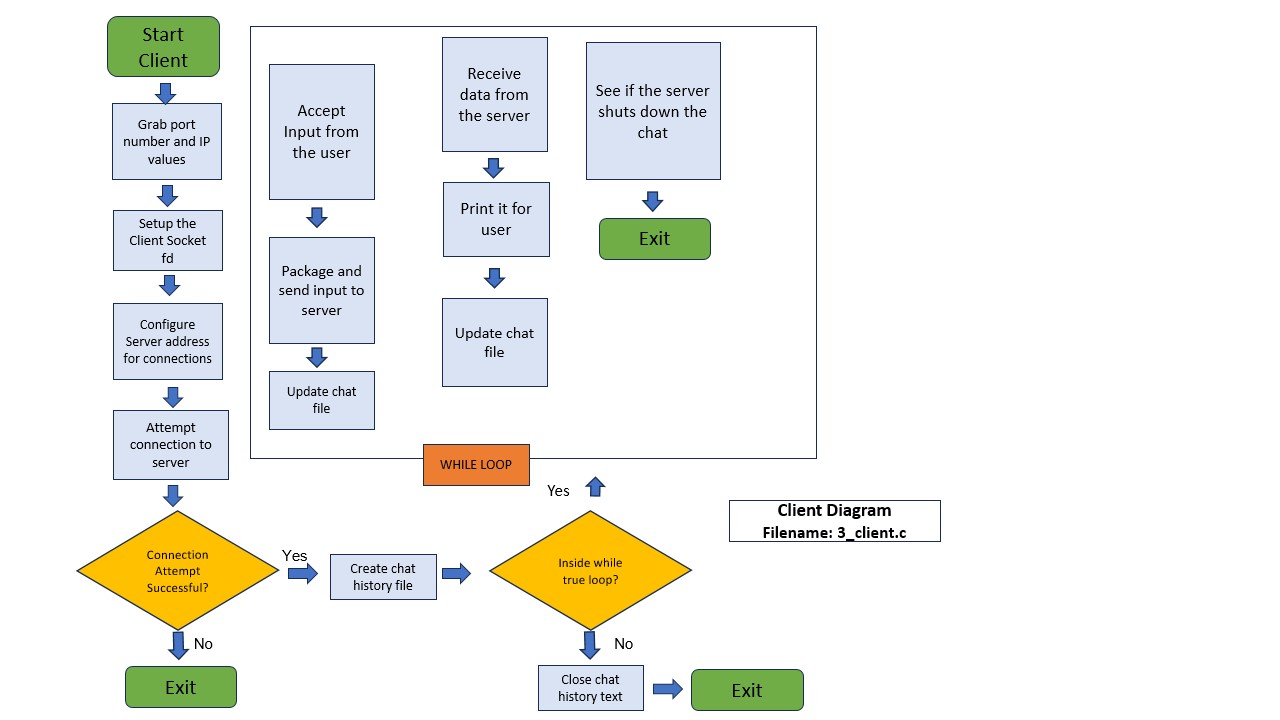
  
This is the only new program, it was specifically created to wait for input from the main server, queue that data, uppercase it with threading, then dequeue the data and return it to the original server.

**Diagram for 2\_server.c (Server Program):**



This program was originally created for project two and was modified to send the messages it receives from clients to the 1\_helper program to be uppercased. It will wait until it receives an answer from the helper, then it will send out a message back to the connected clients. Some changes were made internally, but this will be discussed further in the Implementation Details section.

**Diagram for 3\_client.c (Client Program):**



Like the server program, the client program was created for project 2 and was changed very little (if at all) during the course of project 3. Its main function is to take input from the user, send them to the main server, receive a message from the server and display them. It also creates a chat text file to the course of conversation that is displayed in the buffer.

List of Routines and their Brief Descriptions:   
NOTE: As stated before, some of these functions are the same as the ones I created for project 2.

**Helper Functions (1\_helper.c):**

**void \*serverDecoder(void \*arg)** - We pass this function to our threads so we can go through, select the case that matches our thread ID and then uppercase the selected vowel. It creates a temporary queue to modify the values in our original queue, them pushes them back onto the original queue before destroying the temporary queue. We pass in the queue via a frame and it works directly from our queue data values.

**void \*serverEncoder(int server\_socket, int dataBytesIncoming)** - This is the function that sends the uppercased data back to the main server on thread 6. It takes in the socket information and uppercased data and then pushes it back through the socket.

**void setupTheQueue(structForQueueBoi \*inputQueue, int maximumSizeBoi)** - This function initializes our queue data structure so we can use it to subsequently store the data we receive from the main server. It takes a queue data structure name and the maximum size of said queue as inputs.

**bool addToTheQueue(structForQueueBoi \*queueStructureToUse, char dataToAppend)** - This function allows us to add a character item to the back of our queue and will adjust the head and tail values of the queue appropriately. Before doing anything, it ensures that the queue is not full, returning false if the queue is full (if the queue is not full, it will return true). As arguments, it accepts the specific queue to append to and the data to append. This is used by the helper function to prepare the data received from the main server so it can be passed into the serverDecoder function to be uppercased. It also is used within the serverDecoder function to create a temporary queue so we can alter the encoded data values.

**char removeFromTheQueue(structForQueueBoi \*queueToDecriment)** - This function is used to remove the first item in specified queue and return it as a character. Before doing anything, it makes sure that the queue is not empty so it doesn’t error out and will return a null character if so. As argumensts, it takes the specific queue data structure you want to modify and it will remove the first character (queues are FIFO). This is used within the serverDecoder function to remove data from the original and temporary queues so we can uppercase the vowels. It is also used to consolidate all the uppercased data before it is returned to the main server.   
  
**struct StructForQueueBoi** – This is the structure used to organize our circular queue data structure (keeps track of head, tail, number of elements and size).

**struct** **TheadsForAIEOU** – This is the structure used to nest our queues in when we pass them to threads, basically attaches a thread id so we can use them in serverDecoder.

**Main Server Functions (2\_server.c):**

**void talkWithHelperServer(char \*dataSentToHelper, char \*returnedDataFromHelper)** - This function was specifically created to facilitate communication between the 2\_server.c and the helper node. It takes data that the main server receives from clients (taken as an argument) so it can forward it onto the helper to be uppercased, then it receives the uppercased data back.

**void \*clientThreadSplitFunction(void \*arg)** - This function uses threading to consolidate and handle all the interactions between the client and server. It will intake a client id, make sure it is still connected, send out client updates and inform when other user join. It calls many of the subsequent functions and takes client socket id and thread ids as arguments. It also facilitates the communcation between the main server and the helper node by calling the talkWithHelperServer function.

**void outputConnectedClients(int selectedClient)** - A function that is used by the server to send out a list of all currently connected clients to any new clients joining the network. It takes a client id so it can send out a message to the client via the socket connection.

**void sendPublicMessage(char \*inputData, int selectedClient, char \*senderName)** - A function the server uses to send a chat message to all clients, not just a 1 on 1 chat. This loops through and sends out data client by client with client ids.

**void newClientJoinedPush(char \*senderName)** - This function will push a message to all current clients announcing when a new client has joined and what their name is. Takes the target name of the sender as an argument.

**struct organizedClientData** – is a structure we use to send client messages between the client and server. Basically takes socket id and name/data of client.

**Client Functions (3\_client.c):**

**char \*createFileName(const char \*username)** - This is the function we use to create a chat log text file to keep track of user conversations. Uses date and time to create a unique chat file.

**void writeMessageToFile(FILE \*outputFile, const char \*message)** - A function that is used to immediately write an incoming message to the chat history file with fflush (flush the buffer).

**General Functions (encDec.c):**

**void myLogo()** - This prints out a custom logo that I have made to the terminal.

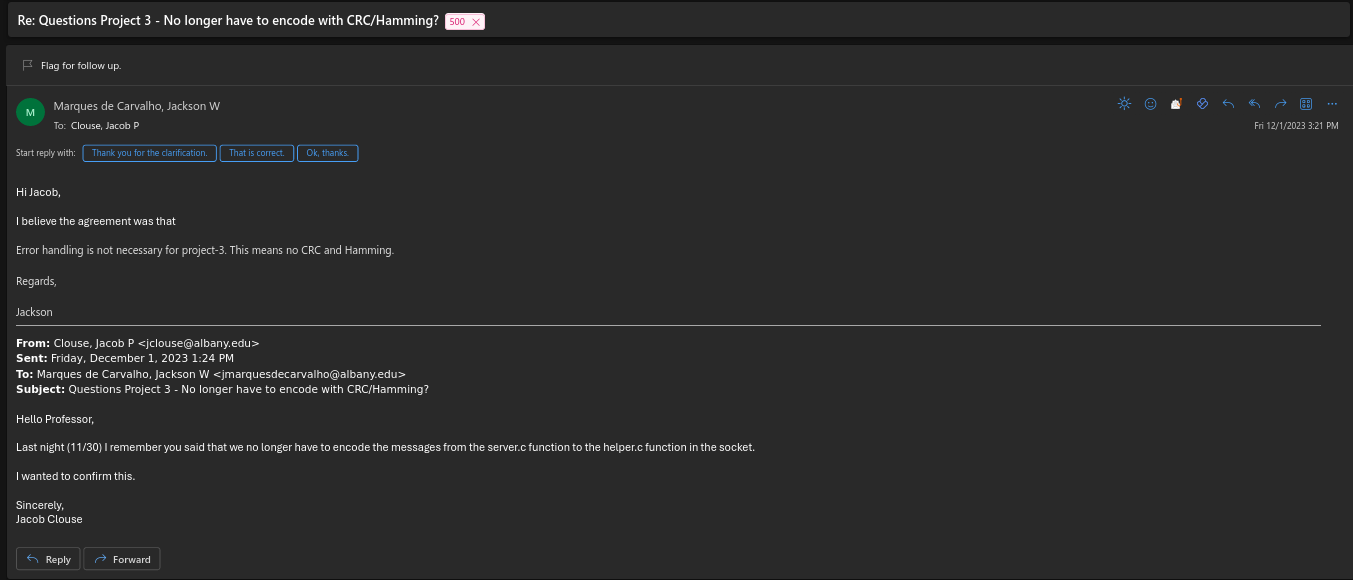
**void serverConnected()** - This function runs on the helper and prints when the main function connects to the helper program.

**void connectionTerminated()** - This function runs on the helper and will print out when the main server terminates its connection to the helper program.  
  
  
**void serverConnectedToHelper()** - This function runs on the main server function and will print when the main server establishes connection with the helper program.

## Implementation Details:

With Project 3, the main goal was the implementation of the helper node function to receive data from the main server so it could uppercase the and send it back. The core of the client / server chat remains unchanged from project 2 as it was not the focus of this project, and I will not be diving deeply into the code that I did not directly write for this project.

**IMPORTANT:** As of 11/30/2023, Professor Jackson removed the requirement for us to encode / decode the uppercased data with error handling, here is the email confirming this:



**Helper Node Function (1\_helper.c):**

The meat of this project was creating the helper node function to receive data from the main server and uppercase it. I basically was creating another server program and modeled much of the addressing, binding and listening functionality based on the main server code. In my failed, initial attempt, I created a threading function to directly read the data from the input buffer before creating threads to iterate over the buffer, uppercase the vowels based on the thread id (the thread id would be used as the case id) and then take that output. However, this ignored the requirement for us to queue the data we received from the main server in a circular queue to perform multi-threaded vowel modification.

On my second attempt to write this, I scrapped most of my code and wrote the queuing functions. This was supplemented by heavy research on the best methods to queue and dequeue elements within c, and a watched a fantastic video on the subject by Jenny’s Lectures and another video by Jacob Sorber and was able to better understand how to develop my queuing and dequeuing functions (source links are in my code in line with functions). I tested my functions by themselves first to make sure data was being correctly pushed into my queue and then returned from it before I touched threading again. When initializing my queues, I made sure to define the size of the individual queues (in the instructions the size is specified as 5) and then we take the buffer and divide it by our queue size. We then raise it to the nearest whole number and we know how many queues we will need to hold all of the incoming data from the buffer. Any extra space will be filled with spaces as to not leave incomplete queues.

I was then able to salvage some of my previous threading code and parts of my serverDecoder function (specifically the switch cases with the thread ids as the case identifiers) so I could parrallelize how queues were processed. The main addition was the use of another, temporary queue in my serverDecoder function that would serve as a temporary holding structure for the modified queue data. I then was able to create my threads, initialize them, pass the serverDecoder function to each of them, and then join the threads in the main section of my helper node to achieve my uppercased string.

The uppercased data is then sent back to the main server function with serverEncoder function (as specified in the instructions), and the helper node is made ready to repeat the process and accept more input to uppercase.

**Main Server Changes (2\_server.c):**

I initially changed the server function to use threads instead of switch statements. I did this to familiarize myself with threads and how they worked more so I could plan how I would proceed with the helper node function. It is functionally the same and many of the same sub-routines / functions are the same, but it services the client / server chat with threads now.

As it stands, this server is run via a script providing the port and IP, it connects to the helper function and waits for message from the client to send out to be uppercased, then receives them and forwards them to all connected clients. It retains the ability to give clients notice when new users join the chat and send new users a list of current clients.

**Client Recap (3\_client.c):**

The client function was not changed functionally, the only real change was moving the header declarations into the encDec.h header file, adding comments and code clean up. It retains its ability to connect to the main server socket, create a message text log file with the data/time and username, send message information to the server, receive incoming texts and write all messages it sends/receives to the log file.

## Test Documentation

**How I tested my code:** I tested my code by taking features and running them in isolation before adding them to the main program. For instance, I created a standalone c file to create circular queues, add data to them, remove data from them, destroy them and free the memory that they used to make sure that they worked. I have had issues in past projects were I would write code directly in the main function that wouldn’t work and I would have issues with troubleshooting to find what was causing the issue. This way I was able to verify my approach was correct before I took the plunge.   
  
I did a similar test with threading; I originally had the first attempt’s threading code so I confirmed that the threading implementation worked as intended and didn’t cause any unexpected behavior.

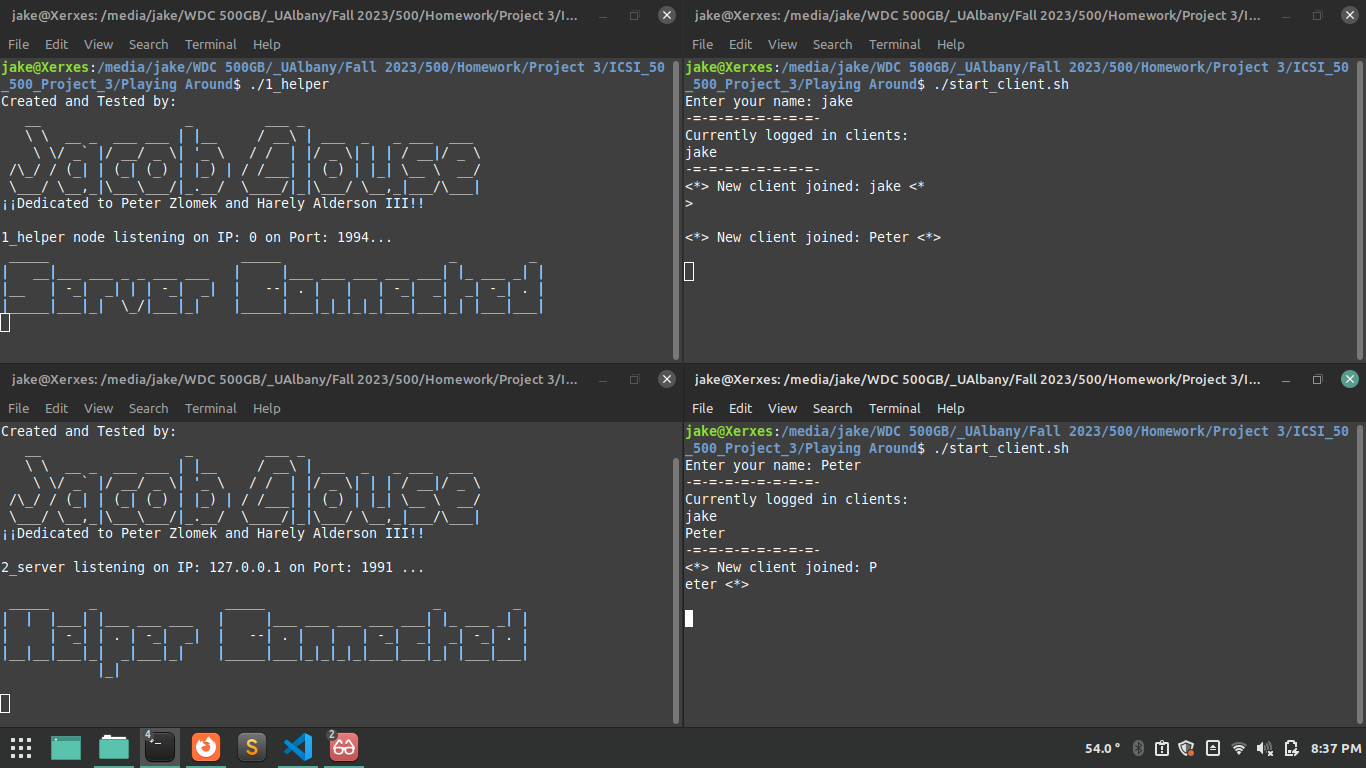
Once both parts were working, I first added queues (with print statements) into the main helper function and then compiled all three programs. I ran my helper, my server and two clients to test and see if the data was being received by the helper and divided into queues.

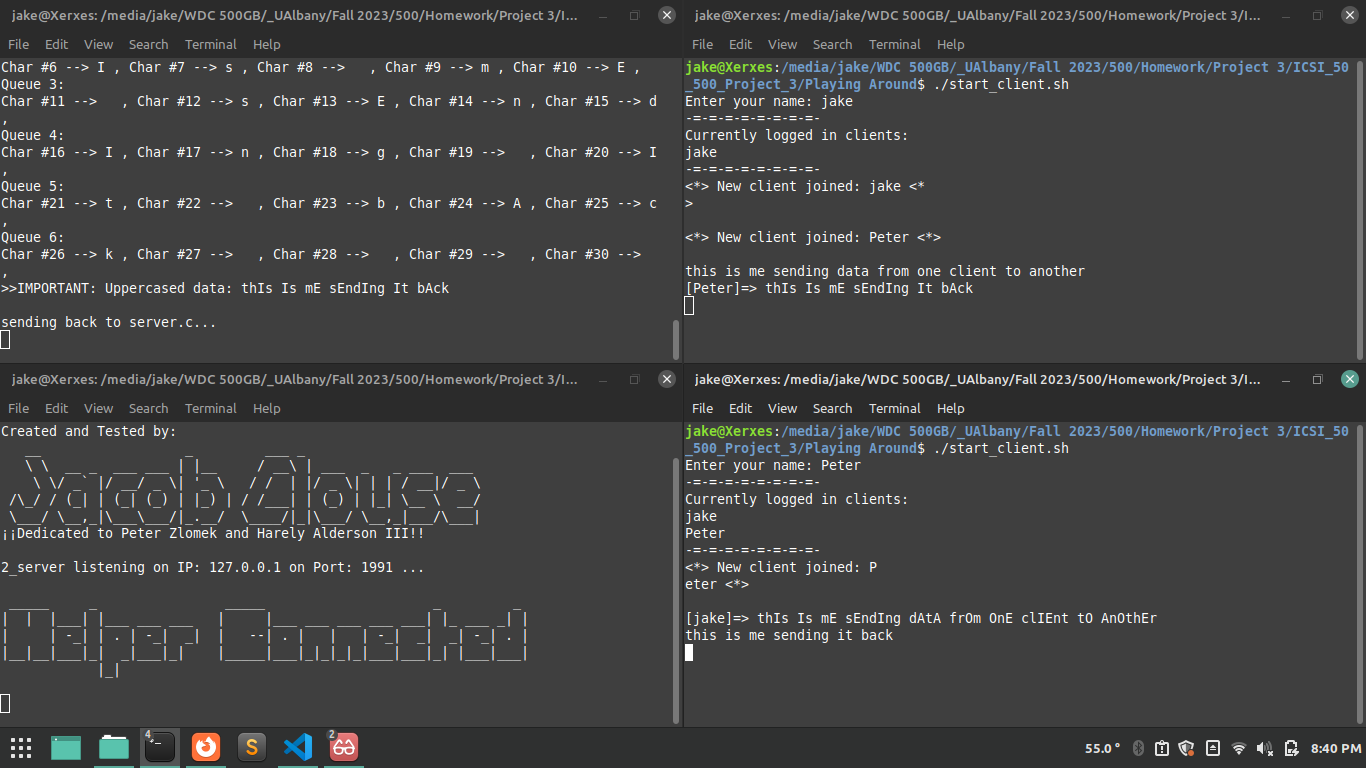
After this, I added the threading logic to the helper function and then checked to see if the data was being uppercased correctly. I did end up having some issues where threads were not joining and the data wasn’t printing out and initially thought is was due to the threading. It was an error I made in the queuing logic (I was passing in the wrong number of data variable and that was causing it to get stuck), which was quickly corrected.

After this, I was able to confirm the uppercase functionality was correctly functioning by creating a few sample chats between multiple clients. Data was being received, printed and written just as expected.

## Testing Outputs:

Here is the initial setup for testing (immediately after running the helper, the server and two clients):

  
  
Here is an example conversation between two clients (note how the helper is printing out the data in the queues):



Testing would be me expanding on this and adding more clients to the testing suite.

## User Documentation – Sources

Here are the sources that I used directly in this project:

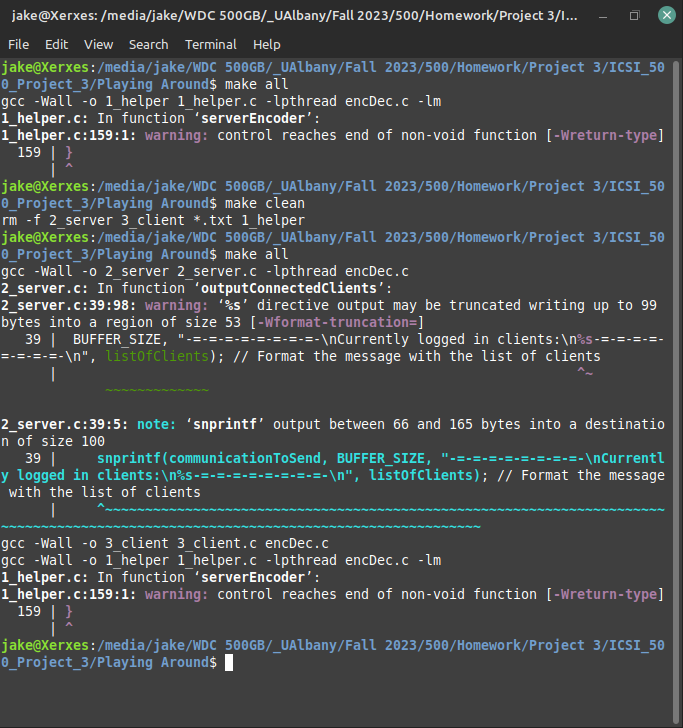
|  |
| --- |
| - Convert Uppercase to Lowercase in C: https://www.javatpoint.com/convert-uppercase-to-lowercase-in-c  - Working and creating threads in C but am running into a problem with the argument type: https://stackoverflow.com/questions/71230931/working-and-creating-threads-in-c-but-am-running-into-a-problem-with-the-argumen  - Java Multithreading Uppercase String: https://stackoverflow.com/questions/53089643/java-multithreading-uppercase-string  - Switch Statements: https://www.tutorialspoint.com/cprogramming/switch\_statement\_in\_c.htm  - Mutex lock for Linux Thread Synchronization: https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/  - Jenny's Lectures - Circ. Queue: https://youtu.be/dn01XST9-bI  - Introduction and Array Implementation of Queue: https://www.geeksforgeeks.org/introduction-and-array-implementation-of-queue/  - Creating a Queue in C: https://www.digitalocean.com/community/tutorials/queue-in-c  - Implementing a queue in C: https://www.youtube.com/watch?v=Ra6p-Bmajlw  - Multithreading in C: https://www.geeksforgeeks.org/multithreading-in-c/ |

Here are sources that I used in the previous project that this project is based on:

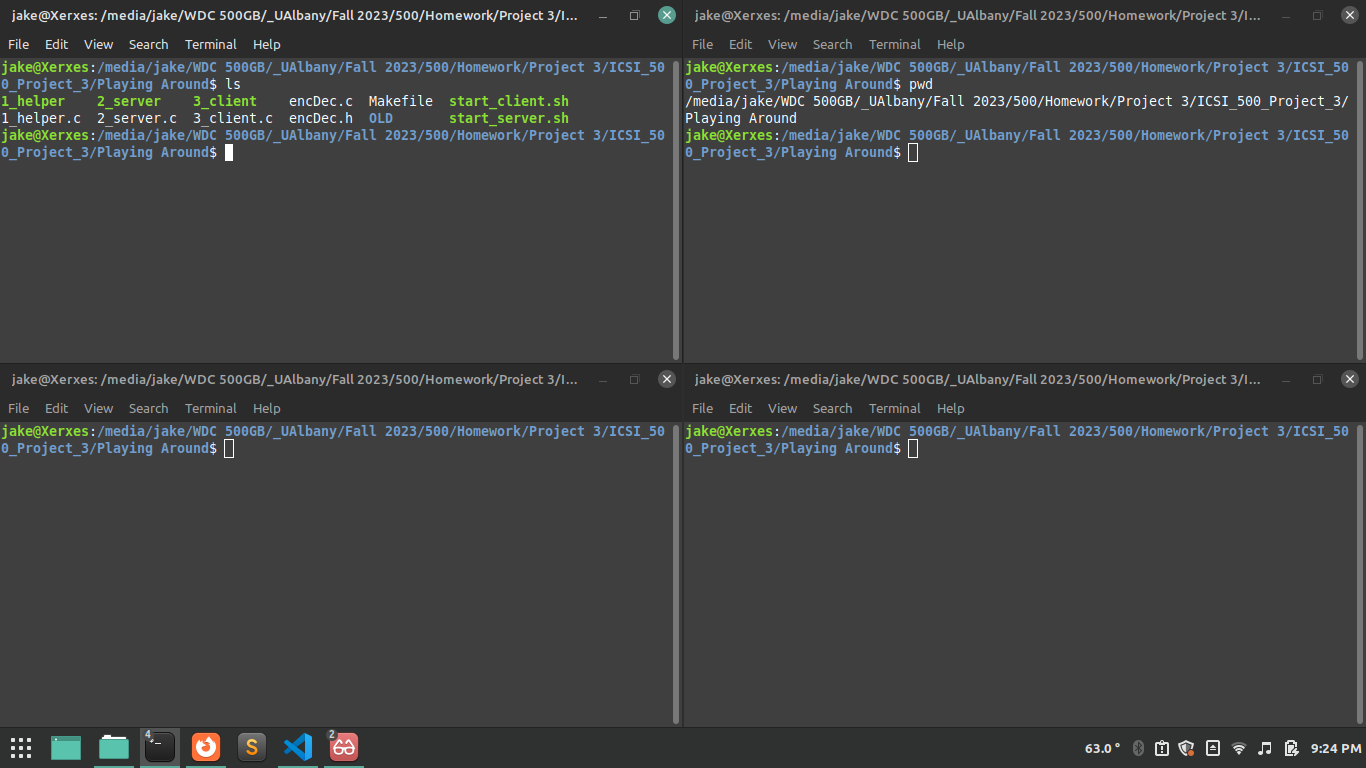
|  |
| --- |
| - Sockets in C: https://www.geeksforgeeks.org/socket-programming-cc/#  - CRC Explained: https://www.youtube.com/watch?v=izG7qT0EpBw  - Hammering Explained: https://www.youtube.com/watch?v=1A\_NcXxdoCc  - CRC encoding (Nesco Academy): https://youtu.be/A9g6rTMblz4  - CRC decoding (Nesco Academy): https://youtu.be/wQGwfBS3gpk  - Program 9 : C program to implement Cyclic Redundancy Check(CRC): https://youtu.be/5sHAx-WcES0  - crc32 -- Compute CRC-32 Checksum: https://refspecs.linuxbase.org/LSB\_3.0.0/LSB-Core-generic/LSB-Core-  generic/zlib-crc32-1.html  - Sending a structure through a socket in c: https://stackoverflow.com/questions/47664715/sending-a-structure-  through-a-socket-in-c  - Convert to binary: https://www.c-programming-simple-steps.com/text-to-binary.html  - Hamming code g4g: https://www.geeksforgeeks.org/hamming-code-implementation-in-c-cpp/#  - CRC walkthrough: https://youtu.be/5sHAx-WcES0?si=Q1e8ZW1-b2zEx2bz  - Hamming Code 1 (Nesco Academy): https://www.youtube.com/watch?v=WdmGSWrcMvM  - Hamming Code 2 (Nesco Academy): https://www.youtube.com/watch?v=1A\_NcXxdoCc  - Hamming Code 3 (Nesco Academy): https://www.youtube.com/watch?v=wbH2VxzmoZk  - geeks 4 geeks python Hamming code: https://www.geeksforgeeks.org/hamming-code-implementation-in-python/#  - Text to binary: https://www.c-programming-simple-steps.com/text-to-binary.html  - Binary to text: https://cboard.cprogramming.com/c-programming/52090-binary-string.html  - Sockets in C Youtube: https://www.youtube.com/watch?v=b\_TUtu3PemQ  - Sockets in C playlist: https://www.youtube.com/watch?v=\_lQ-  3S4fJ0U&list=PLPyaR5G9aNDvs6TtdpLcVO43\_jvxp4emI  - Get current time in c: https://stackoverflow.com/questions/5141960/get-the-current-time-in-c  - Flush the buffer in c: https://www.geeksforgeeks.org/use-fflushstdin-c/ |

## How To Run My Program:

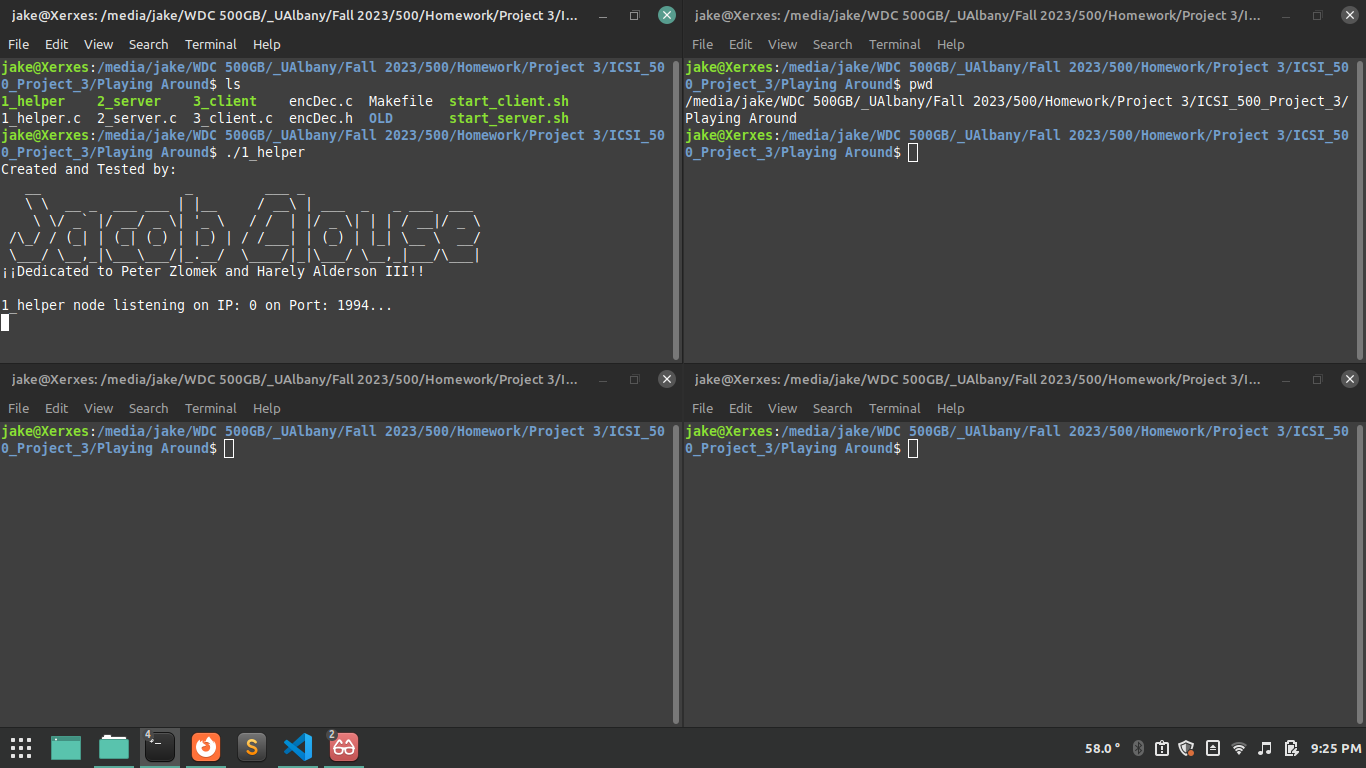
1) First, we need to compile the code. For my 1\_helper.c, 2\_server.c and 3\_client.c. You can do this by opening the terminal in the directory with the c files in them and typing **make all** to create all 3 binaries:



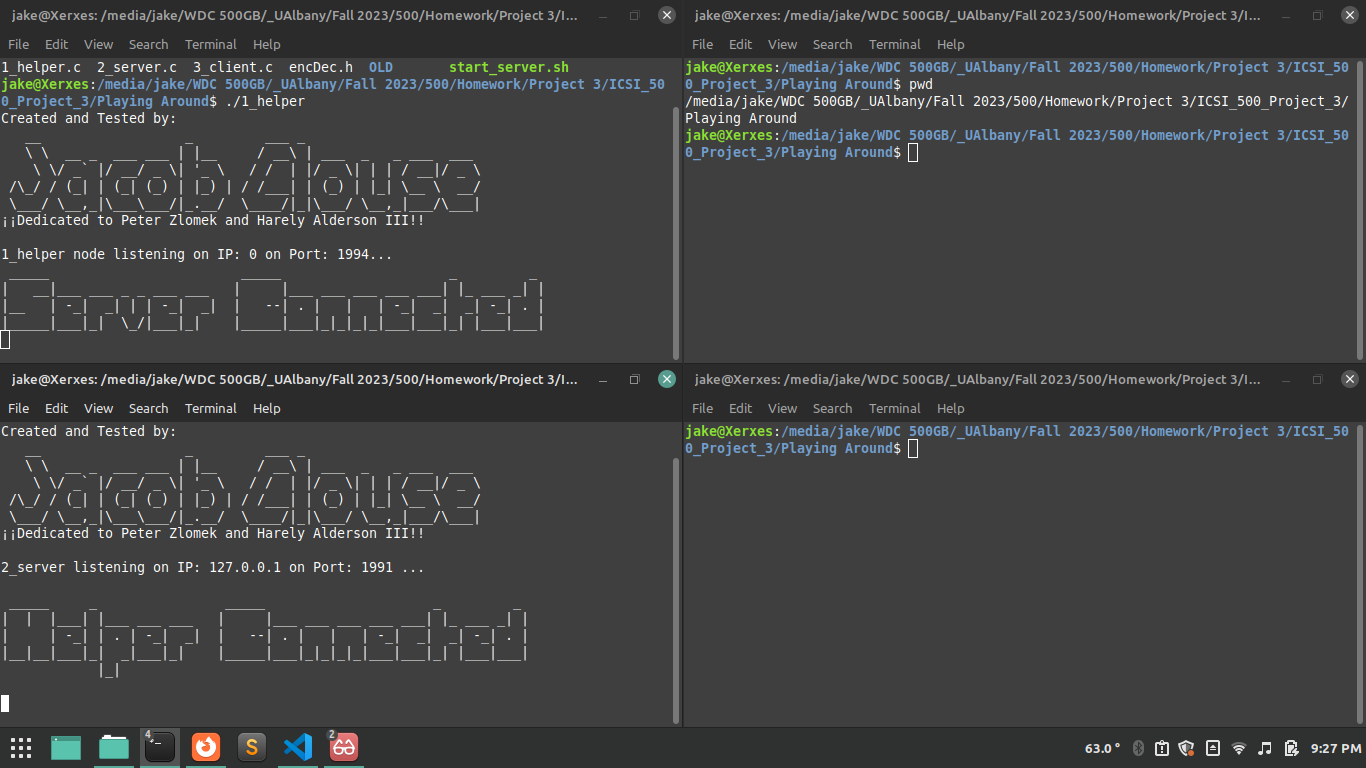
2) If you want to have 2 clients, you will need to open 3 additional terminal windows in the same directory as your first one:



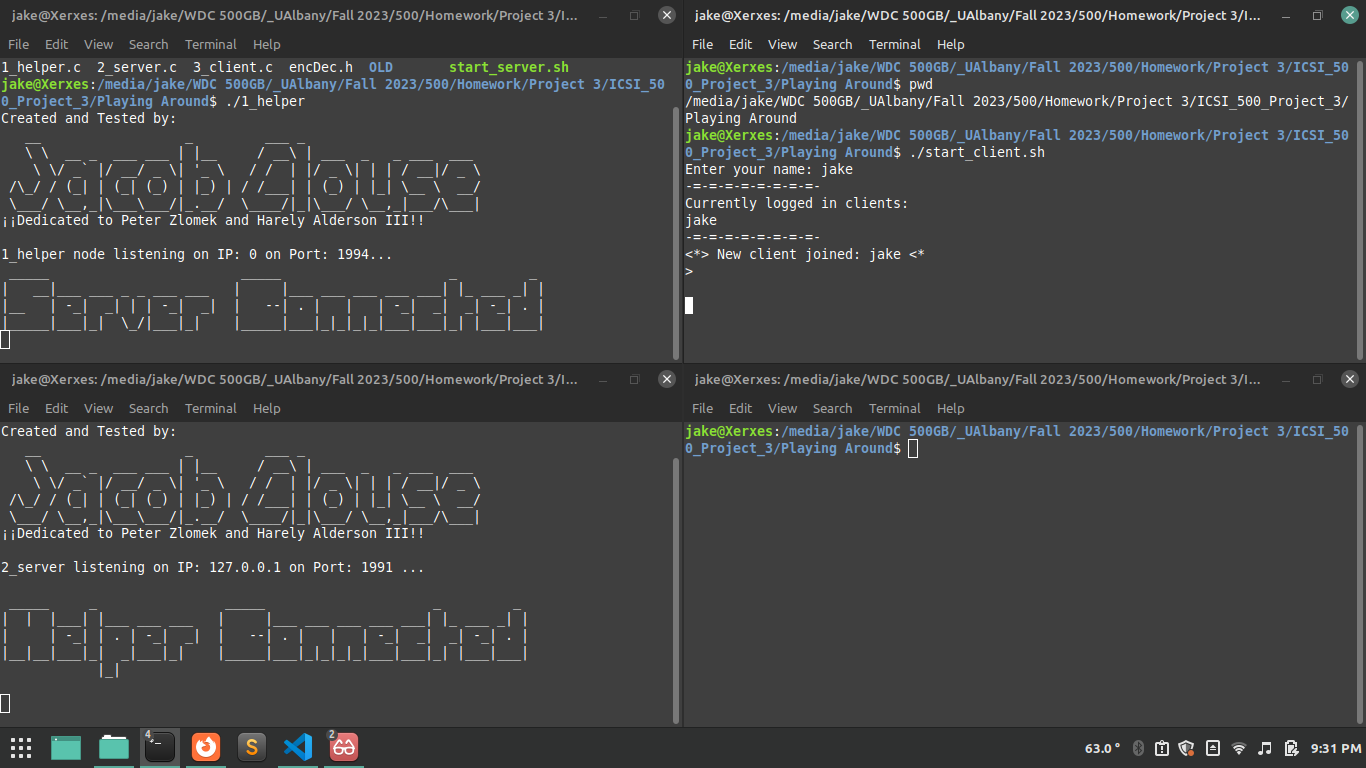
3) In the first window, you need to type **./1\_helper** to start up the helper node server:



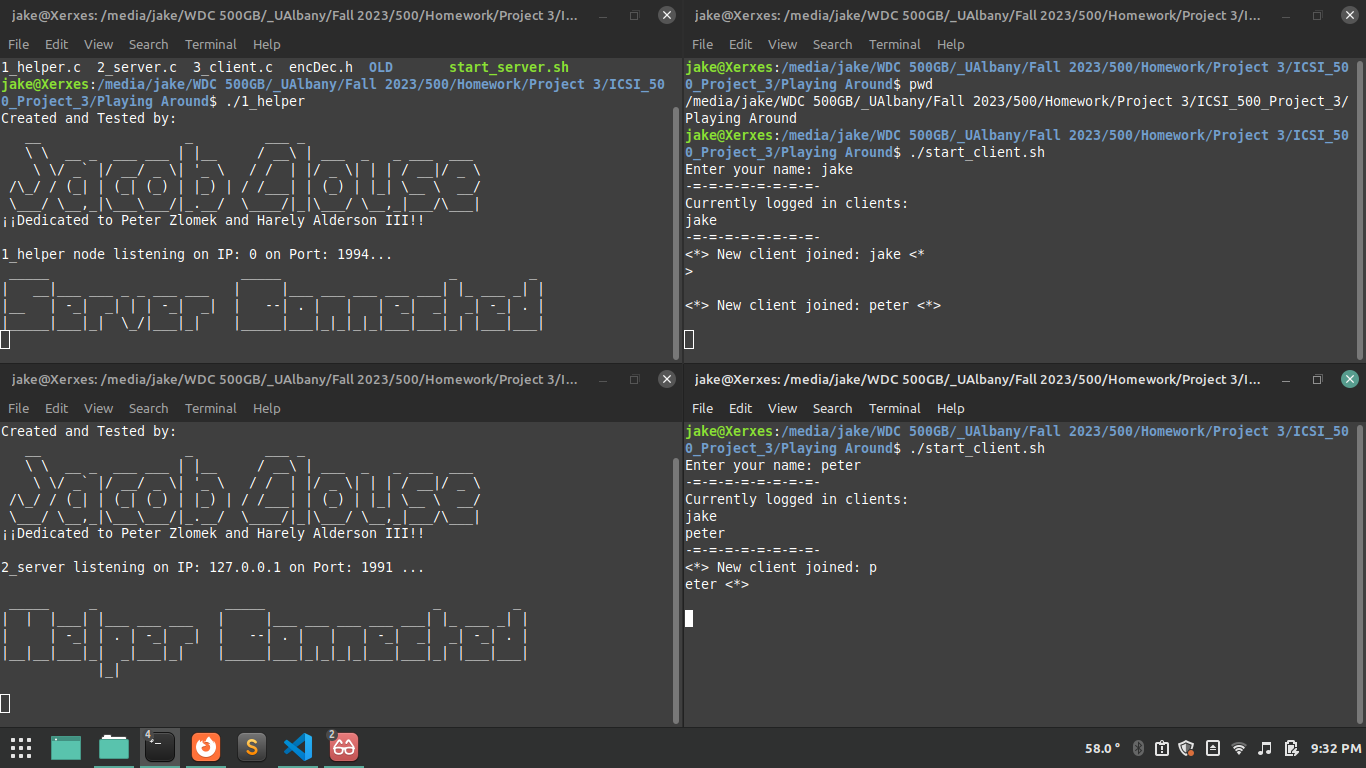
4) In the next window, you need to use type in **./start\_server** to execute the shell script to start the main server function (if you haven’t done so before, use **chmod +x start\_server** to make the script executable):



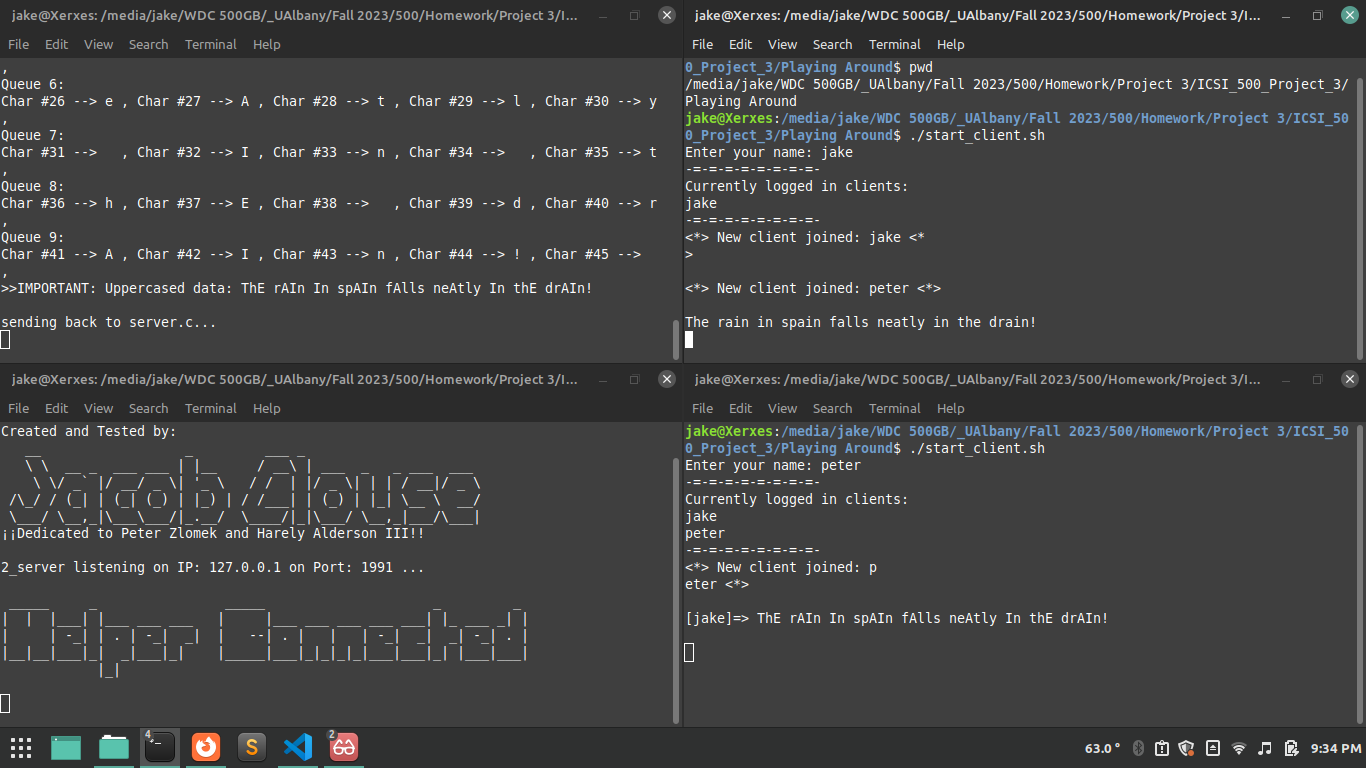
5) To start the first client application, type in **./start\_client** to excute the shell script to run the client application (again, if you haven’t already you will need to use **chmod +x ./start\_client** to make it executable). When prompted, enter in your name:



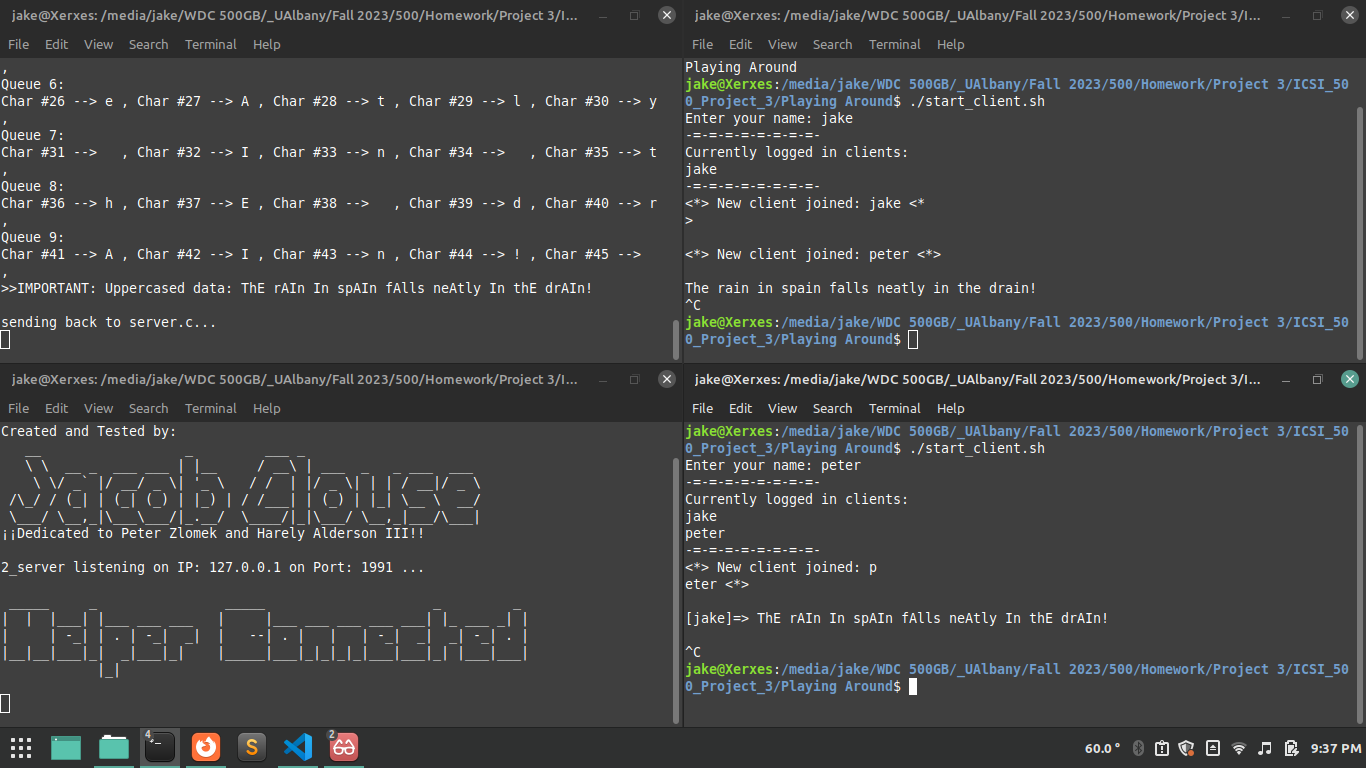
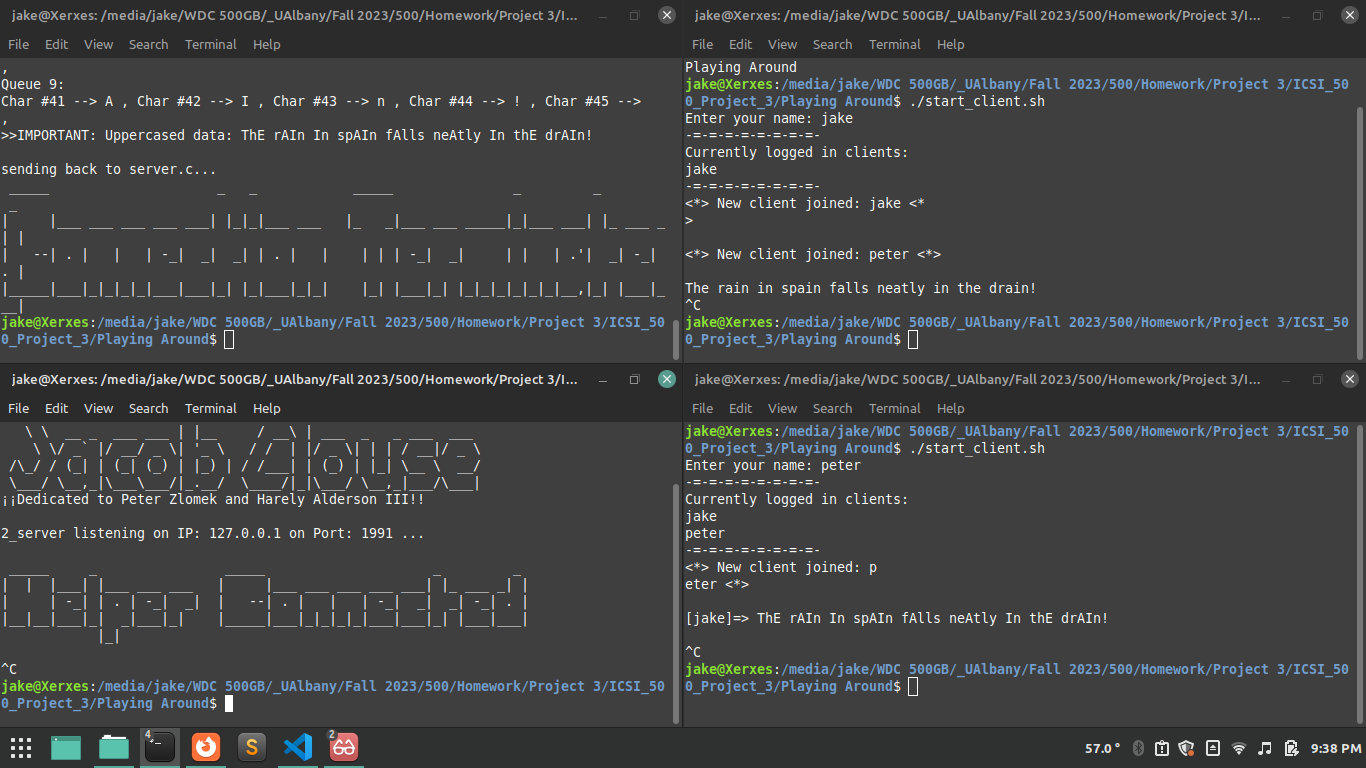
6) Repeat step 5 in the next window, enter in a different name:



7) Now you can talk between both clients and see the helper function convert the vowels to uppercase in real time! Here is an example:



How to Stop my Program:

1. First, press Control+C in both the windows where the client’s are running, this will cause them to exit:   
     
   2) Then press Control+C in the window where the server is running (it will automatically send a message to the helper to exit as well):  
     
   And you should be done, the program is no longer running.

## Describe parameter (if any):

There are no parameters for running this code besides having a working copy of gcc to compile with, a good IDE, and a Linux distribution!

NOTE: I didn’t use any files for texting, just typed in what came to mind when testing out the chat system!