Jesse Woo

jw4202

CSEE4119 Programming Assignment 1

**Command line inputs:**

For server-side: ChatApp.py -s <port number>

For client-side: ChatApp.py -s <user name> <server IP> <server port> <client port>

**Runtime inputs:**

Server: none from user.

Client:

send <recipient name> <message>

dereg <user name>

reg <user name>

send\_all <message>

**Project Description:**

This is a simple chat app using UDP. Clients are able to register with a server and receive a table in the form of a text file that contains the user names of all other clients, their IP addresses and listening port numbers, and whether they are online or offline. Message are sent server-client and client-client using a header with two lines, followed by content data. The first line is an expect code, and the second is the name of the sender. The server maintains a dictionary with the names and info on each client, and with each request will update the dictionary and read it into an offline text file for offline storage. Server “broadcasts” messages and table information to all clients with brute force using a for loop, not a broadcast address. The server also maintains a text file for any client with offline messages containing the name of the sender, the message, and at timestamp. Both client and server use multithreading to simultaneously listen for requests and process inputs.

All requested features have been implemented. This includes client registration with server, peer to peer chatting, deregistration and registration, offline chat, and group chat. Acknowledgements on the client side are handled a flag that acts as a semaphore between threads. On the server side, server maintains a separate, global dictionary with all client names and whether an ack has been received. This obviates the need for sequence numbers. Both sides can handle invalid inputs such as an incorrect number of command line arguments or user inputs, or making a request using an incorrect user name.

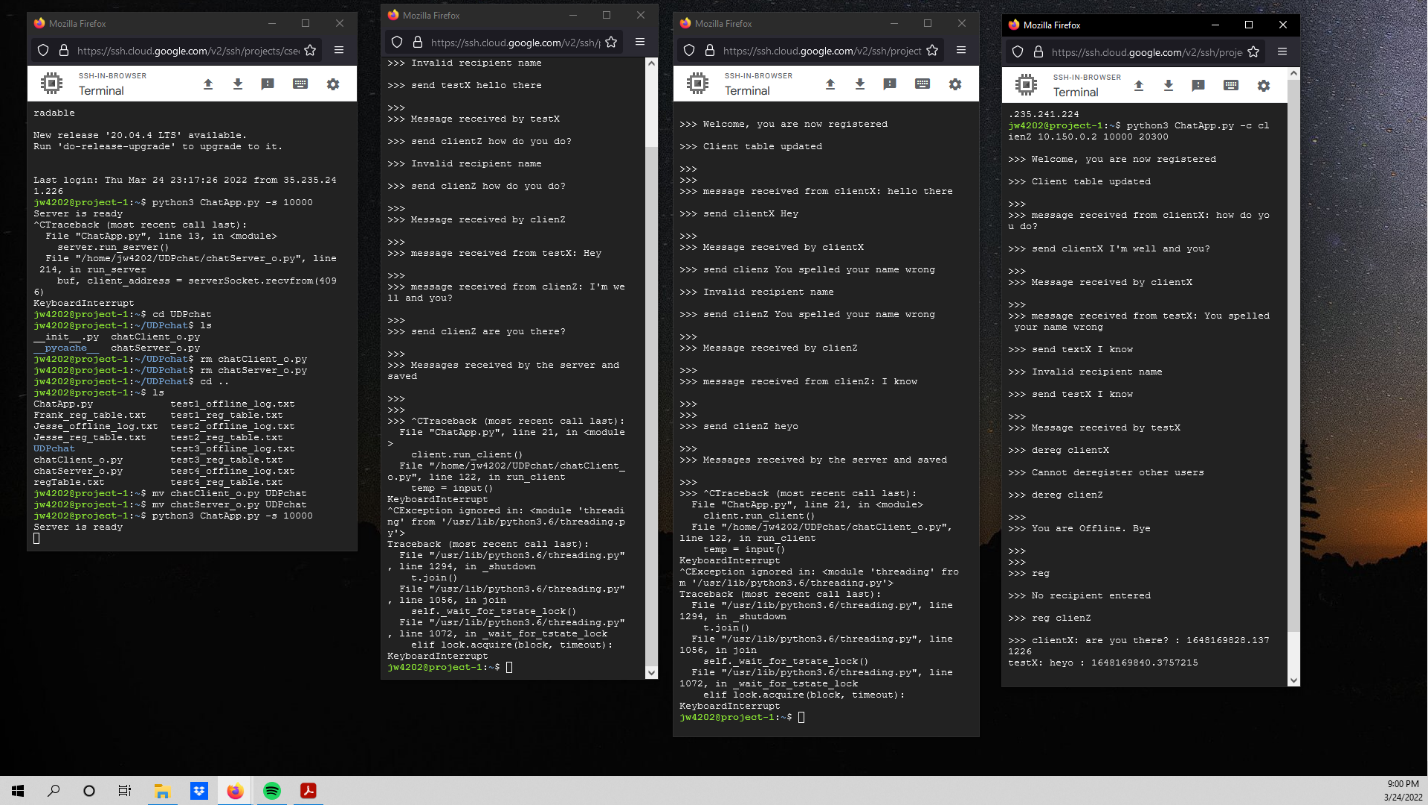
**Known Bugs:**

Because of the multithreading, the number of “>>>” lines can sometimes be a bit excessive. Also, the program can handle clients silently leaving by killing the process with ctrl-c, it will result in a ping from the server and the server updating the table. However on google cloud, if I silently leave by closing a client window, this does not appear to fully kill the process, so the client stays alive and is not properly toggled to offline. I can see in my bug testing that when the client or server send a message to a client that should be dead, they get an ack back and proceed as normal. I don’t know whether this is an issue with google cloud or my code. Again, killing the process with ctrl-c works as it should.

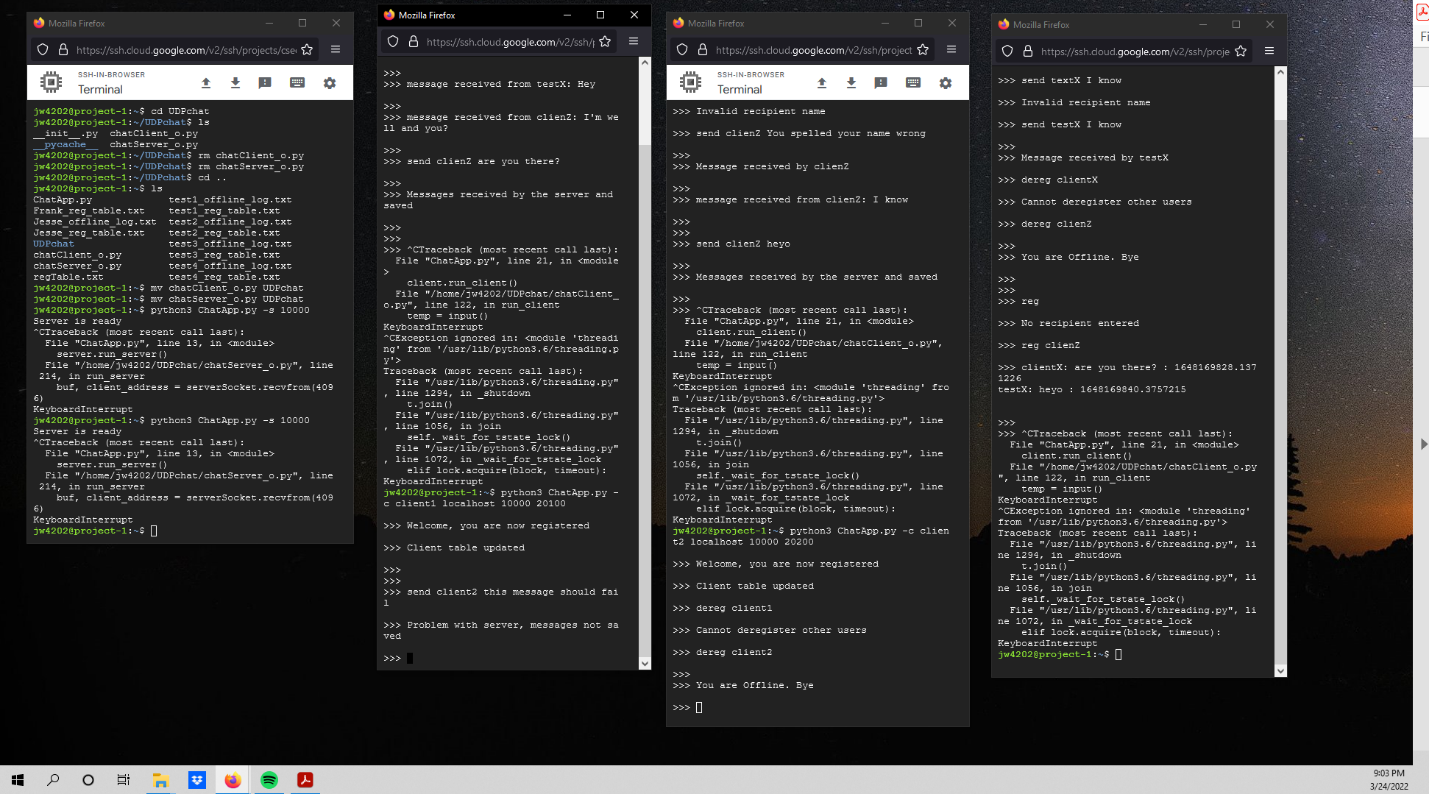
**Testing:**

Here are the outputs from the three test cases called for in the spec. They are included as separate files in the submission. I have also tested various invalid inputs and other scenarios.

Case 1



Case 2



Case 3

