CMPSC 176B Project Checkpoint #B: Large Scale Chat Server with File Transfer

Andrew Doan and Richard Boone

Project Description:

We are implementing a chat server for multiple clients to connect to and actively chat with each other. We would like to host multiple chatrooms on one server which clients can connect to individually. Upon connection, clients will be able to send encrypted text messages to other clients within the chatroom. They will also be able to share files with other clients who are in the chatroom. Each separate chat server instance will be able to host 10 individual clients, 5 separate chat channels, and store up to 100 unique usernames. At this time, no changes have occured in the specification of our project.

Original Timeline:

Milestone	Deadline
Initial server and single client programs with text echo	1/25/2018
Server and single client programs with file transfer echo	2/1/2018
Single chatroom support for multiple clients	2/8/2018
Single chatroom support for multiple clients with encryption	2/15/2018
Multiple chatroom support (no encryption, file transfer)	2/22/2018
Compile report B	2/24/2018
Multiple chatroom support with encryption and usernames	3/1/2018
Multiple chatroom support with file transfer and encryption	3/8/2018
Final Report:	3/10/2018
Stretch Goal:	Simple GUI/web page

Progress

As shown above, our original timeline showed us completing the entire project with a command line interface by the final report. However, we were able to complete the entire project as well as our stretch goal, and add a GUI for the client-side connections.

Implementation

Server Side:

On the server side, our server program listens from three different ports. Most interaction occurs through the first port, which acts as the chatroom and initial server. If the client wants to send a file to the chatroom, they are redirected to another server on the second port, where they upload the file to the server. If the client wants to download a file which another user has sent to the chatroom they are redirected to the third port, where they download the file.

The server we use uses a multithreaded approach to networking with multiple chatrooms. On startup of the server, we spawn two extra threads, one for the file receiving server and one for the file sending server. On each of these threads, and also on the main thread, our server listens for new connections. When a new connection comes in, the server spawns a new thread for the client according to which port the connection is coming over. If the connection occurs on the chatroom port, the incoming connection is asked for a username and initially assigned to the first chatroom. The server assumes all incoming traffic is encrypted. If the server receives unencrypted traffic, it will read it as garbage and close both the socket and the thread.

Although we planned to have the server support up to 10 clients and up to 5 chatrooms, we built the server to be expandable to many more clients. We have tested the server up to more than 20 clients with no issues, so it is effectively infinitely expandable, although at some point it would be limited by the computing and network power of its host computer.

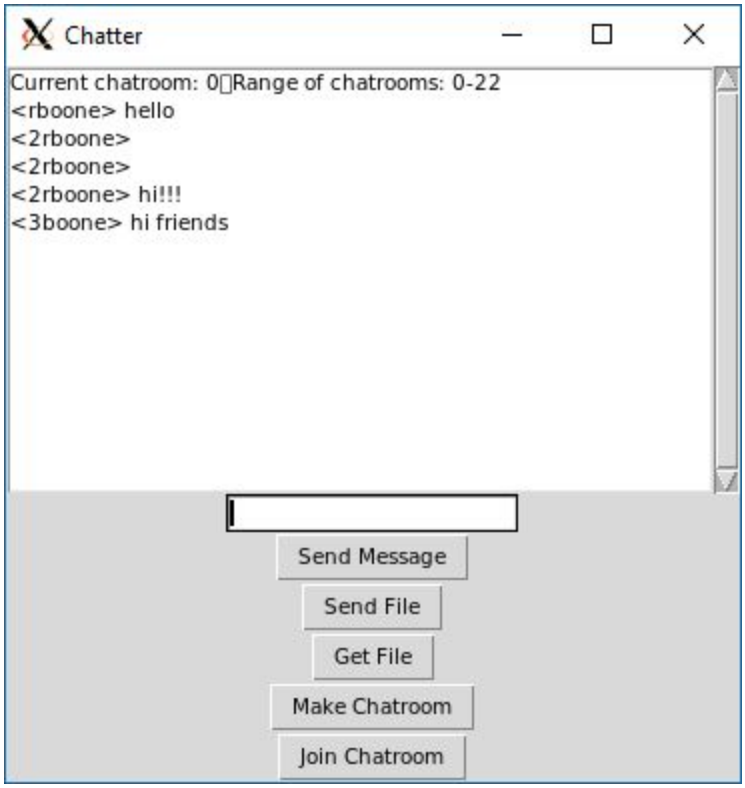
Client Side:

The client program is started by giving it the IP address and port number of the server's chat server, as well as the encryption key and a username for the given client. The same AES-128 encryption key is needed on both the client and the server. The client connects to the server's chat room, and is assigned to the first chatroom. If the client wants to send a message to the chatroom, the client types the message in the message box and clicks send message. If the client wants to send a file to the chatroom, they type in the file name and click the "Send File" button. If the file input is valid, the client sends the command to the server. The server will send the client the port number of the file receiving server. The client will connect and upload the file, then close the connection. When the file is sent, the server announces to all clients that a file has been uploaded with its respective filename. If the client wants to receive a file, they will type the file name into the prompt and click the "Get File" button. If the file is unavailable, the server will tell the client so. If it is, the server will send the client the port number for the file sending server. The client will then connect to the file sending server and download the file.

GUI:

The GUI was implemented client side using tkinter. The client is able to see all of the chats that have occurred in rooms they have been since they joined the server, and are able to scroll to the sides to see long messages, and up and down to see previous messages. The GUI also has buttons for each of the

functions implemented: send message, send file, get file, make chatroom, and join a chatroom. The text entry box is used to type out the message to send, file to send, file to get, or chatroom number to join. A sample of our GUI is displayed in the image below.



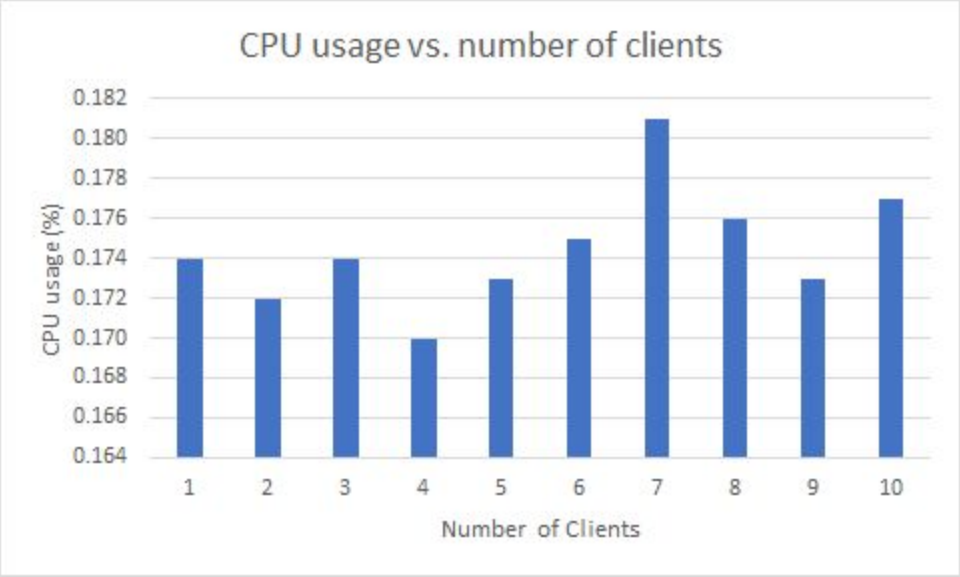
Performance analysis

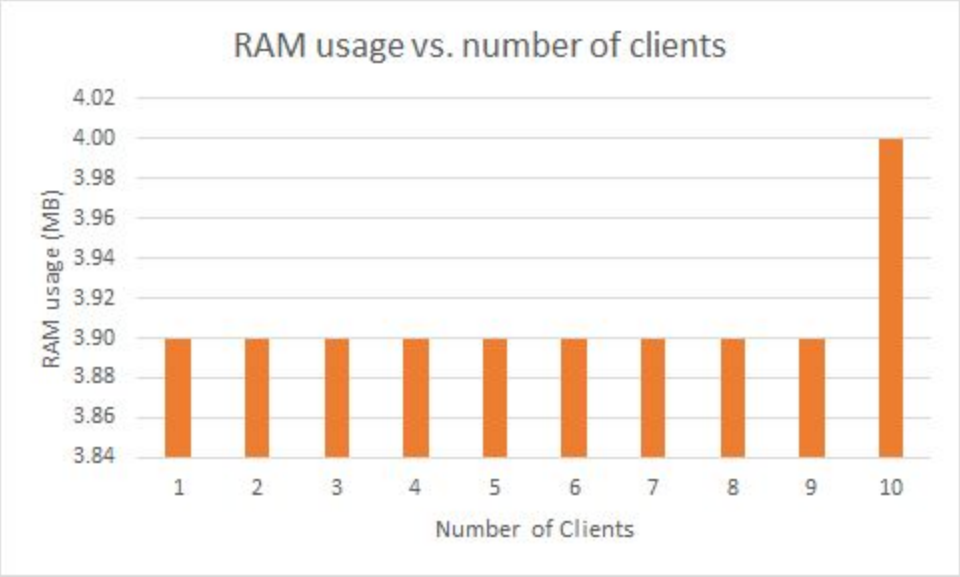
To evaluate the validity of our program for a larger scale, we ran a performance analysis on both the client and server sides. On the server side, we measured CPU and RAM usage of the server-side program with a varying number of clients. This generated the following table.

Client number	CPU usage	RAM
1	17.40%	3.9MB
2	17.20%	3.9MB
3	17.40%	3.9MB
4	17.00%	3.9MB
5	17.30%	3.9MB
6	17.50%	3.9MB
7	18.10%	3.9MB
8	17.60%	3.9MB

9	17.30%	3.9MB
10	17.70%	4.0MB

The above table produces the following graphs for CPU usage and RAM usage. As you can see, there is almost no difference in CPU usage or in RAM usage. This is a good sign, indicating that our server does not take on a lot of extra load for each client. This would generally indicate that the server could scale very well into large numbers of clients.





On the client side, we also did an evaluation of the CPU and RAM usage. We found that each client tends to use about 28.3 MB of RAM individually. It should be noted that when multiple clients were running on the same computer, extra RAM usage per client was lowered to approximately 8.4MB of RAM, with most of the additional RAM used by the python backend, thus indicating that high graphics overlap allows for many clients on a single computer. We were unable to get client CPU usage to go above 0.0%, even during file transfer, indicating that our CPU usage is very minimal.

Finally, we did an analysis of performance during file transfer. As you can see in the table below, there is effectively no change in CPU usage or in RAM usage during file transfer. This again, shows high scalability of our project to many clients and multiple file transfers.

	CPU usage	RAM Usage
Client	0.0%	28.4MB
Server	17.8%	3.9MB

Conclusions

We were able to make a chat server with a client-server architecture which allowed us to have multiple clients with chatrooms, encryption, usernames, and a GUI. Because of its low increases in CPU and RAM usage when adding clients and running file transfer, our server is highly scalable to a large number of clients with minimal issues.