David McCullers

Data Communications and Networking

Term Project Report

May 7, 2020

**Brief Overview: What is the Project?**

In this project I work to build a simple chatroom service using UDP Protocol Socket Programming in Python 3 in which several instances of a client program can connect to, send messages to, and receive messages from a hosting server program that services all requests made from and facilitates messages sent by client programs.

**What is UDP?**

UDP is a connectionless Protocol that provides no “direct” connection (hence connectionless) between server and client, instead the sender of information (either from the server or client) simply specifies the location (IP and port #) that it wants the data to be sent to, the location that it’s being sent from and then the Operating System of the machine delivers that data to a specific application. A real-world example that is similar to this concept is the mailing of a letter. An individual might write a letter [the data], store it in an envelope [UDP Protocol], write the mailing address on the envelope [location IP and Port], a return address [sending IP and Port], and then mail the letter without any direct connection to the person you’re sending it to. Much like when mailing a letter with the risk of it being lost in the mail, UDP Protocols aren’t always reliable in getting data to its destination - just like the real-life example, the sender of the data doesn’t know whether the data has reached the location it was sent to and the order that the data is received isn’t always as sent.

**Why UDP?: About the Project**

Between the protocols used for data transfer to be considered for this project: TCP and UDP, UDP offers quicker data transfer with less reliability and TCP offers a more reliable service at the cost of speed in transmitting and receiving data. The reason UDP was chosen over TCP to be used was due to the services implemented in the program (the Chatroom) a) not being greatly affected by a loss or drop of data being sent or received and b) the usability of the Chatroom being significantly benefited by a more instantaneous transmission of data - offering a closer to real-time environment for communication between active users. This project seeks to explore the use and functionality of the UDP Protocol in a manageable and expandable testing environment capable of observing the intricacies and nuanced inner workings of UDP that could be implemented in a more largely scaled and comprehensive model of the same program at the industry level. There are two major components that the project should include along with specified criteria established to assure a more well-rounded and complete product and are as follows:

**Main Components:**

-A simple client program that in running a user is able to transmit chat messages to the server - as well as receive messages from the server originally sent by other users

-A server program that takes in messages from all existing client programs and redistributes those received messages to other users connected to the server

**Project Requirement Criteria:**

- Server needs to:

• greet new users that connect to it

• announce to all other connected clients that a new user has connected to the server

• uniquely identify and display messages from different connected clients

-Client needs to

• automatically join the chatroom when the program is executed

be able to “leave” the chatroom by inputting a special message that

disconnects from the server

• have a multithreaded operation that allows for simultaneous sending data

to and receiving data from the server

**Challenging Aspects of Implementation:**

The biggest challenges in completing the project and its requirements are my lack of knowledge and experience with the technologies at hand and developing a schema for the workings of the project. As an opportunity for personal growth and development I have chosen Python - a language I’ve used only on a cursory level to minimal success - as my language for the assignment. Despite my unfamiliarity with it syntactically, core computer science ideas and motifs shone through in my learning of Python-specific capabilities and principles regardless of my lack of knowledge of the specific language and the syntactical dissimilarities with those programming languages I’m more comfortable with. In order to teach myself the tools and technology I would need, I used online resources from several outlets dedicated to teaching code to inept developers as well as read the material and resources provided in our textbook and Lecture Slides. With the sample base template of both a TCP and UDP server-client system implemented using Python in the slides along with the knowledge and ideas I gained from those outlets, I had a good foundation for creating the rest of the program that I could build on to. With that foundation I was able to design and finally implement each component required to create a functioning chatroom application.

**Design and Implementation (Server Side):**

To overcome the issue of a possible drop of data with UDP’s unreliable but high speed service, I decided to attach a header to each message sent from the client including the message’s length. Once the server receives the sent data - it can use the message length stated in the header to check whether the data received is the same amount as the data sent. This is done by having a constant variable [HEAD\_LEN] hardcoded in both the server and client that is always sent and received in each message that tells the server how many digits the message’s length can be parsed.

For example,

HEAD\_LEN = 4

The sent message: “Hello World!”

message\_length value: 12

Will be sent to the server with the following format: “12\_ \_Hello World!” where (\_) are trailing spaces in the message header

This way, even with the variable number of digits in message length the server can still obtain the message length’s value

However, this approach does make it such that any given message’s upper limit in character length is 10^HEAD\_LEN

This same idea was applied in creating a USR\_LEN such that the username’s length could be found and parsed more easily. Once the message, username, and message length have been parsed by the server, the client’s IP address is stored in a client\_list (dictionary) paired with their username so that their established connection can be broadcasted to all other connected clients and all messages sent from their address can be identified with relation to their username. The server then takes in all incoming messages and sends them to the clients in the list excluding the client that sent the message with the exclusion of the key phrases “!join”, “!quit”, and “!users” which tell the user to perform different actions outside of sending the received message. When a message with “!join” in it has been received, the server sends a message to the user that sent the message “Welcome to the server!” with their supplied username and to other users that the client has connected to the server. This operation should only be performed when the client is initially connecting - so messages containing “!join” are restricted to only be sent from the client when connecting. When a message with “!users” is received by the server, this tells the server to send a list of users currently connected back to only the sending client, and when “!quit” is received by the server the server sends to all users except the sender a message announcing the user’s disconnection from the server, the client is then sent a “!terminate” message (handled on the client side), and the client is then removed from the client list of the server.

**Design and Implementation (Client Side):**

The client operates with the use of multithreading - breaking up the different processes of sending data to and receiving data from the server. The threads have their own definitions of run() and are started immediately after a call to send the client’s username to the server to be stored in its client\_list. The sending thread takes input from the user of the client program and concatenates the input with the message and user headers containing information about the length of the client’s username and message, and then sends the whole concatenated message to the server to be processed and redistributed to other connected clients. Cases in which “!join” or “!terminate” are found in the user’s input are not sent to the server as they’d be interpreted by the server as the special cases discussed in the Server section. When “!terminate” is found in the input, the client sends the message to be processed by the server and be removed from its client list and the sending thread is terminated - no longer allowing the user to send messages to the server.

The receiving thread takes in data sent from the server - repeatedly every second or so using a Timer from the threading package - and attempts to parse the USR\_LEN and HEAD\_LEN that would be contained in a message being redistributed by the server from another user, if this fails it’s assumed that the message does not contain a USR\_LEN or HEAD\_LEN and is therefore a message originating from the server such as “Welcome to the server” or “[user] has disconnected”. In the case that the message received contains “!terminate” (Having been sent back from the server after the client sends a “!quit”), the thread is terminated and the client is no longer to send or receive any more data from the server, otherwise the processed message is then displayed for the client to be read.

**Requirements Testing (1):**

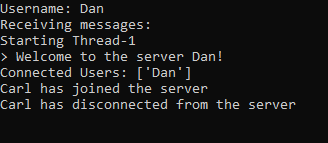
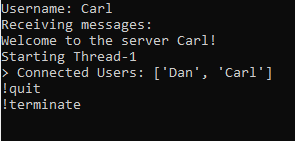
• The client automatically joins the chat room when the client program is run. This can be implemented by sending a special message to the ChatRoomServer, for example, “@\_@Join”, “$$$JOIN”, or “!Join”, etc.

• Whenever a new client joins the chat room: the ChatRoomServer sends a welcome message to the newly joined client. It also notifies (broadcasts to) all existing clients in the chat room that a new client joined. It will add this new client to the active clients set/list.

• Whenever an existing client leaves the chat room: the ChatRoomServer notifies (broadcasts to) all existing clients that a client left the chat room. It removes the leaving client from the active clients set/list. (It also needs to cooperate with the leaving client program, so that the leaving client program can gracefully terminate.)

• The notifications and messages involved above should bear unique identifier of the client, for example, the IP address and the port number used by the client.

• The client allows the user to input a special message to leave the chat room, for example, “@\_@Quit”, “$$$QUIT”, or “!Quit”, etc. When the user issues this message, make sure that the client program terminates gracefully, so as not to cause problems for the server, and not to affect the future rerunning of the client and running of new clients.

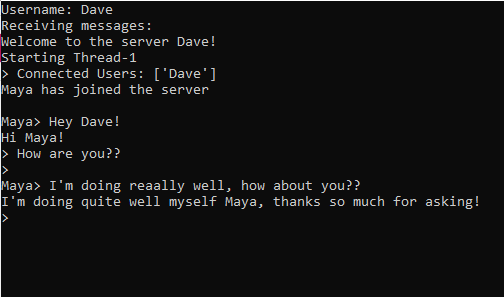


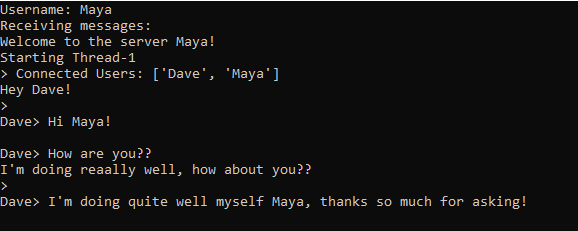
**Requirements Testing (2):**

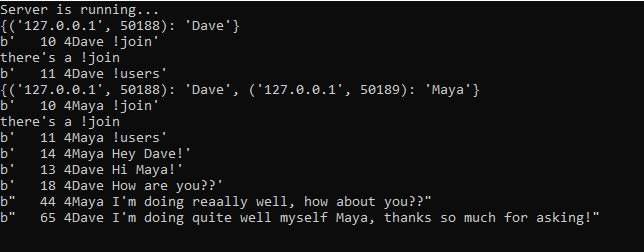
• Whenever an existing client sends a message to the chat room: the ChatRoomServer broadcasts the message to all other existing clients in the chat room.

• When the user types in some message, the client sends the message to the ChatRoomServer. The message needs to be broadcast to all other clients (excluding the sending client) in the chat room. This is done by the server program.

• When the local client is running and in the chat room, it should always wait for messages sent from the chat room server. For example, when another client in the chat room sends a message to the chat room, the ChatRoomServer will send the message to this local client, because the server does broadcasting.







**Note**: I made the conscious decision not to include an overly distinct identification of clients that connect to the server for a more typical user experience in Chatroom based clients or programs by representing their messages by their chosen username - a username that could be a duplicate of another already connected to the server - instead of using their more unambiguous IP address that they use to connect to the socket. One possibility to combine both of these aspects in future updates or implementation would be to invalidate the entry of usernames that are currently connected to the server.

**Note**: this testing also includes examples of my added feature of a !users list that is displayed to the client so that they can see what other users are online.

**Conclusion:**

Throughout the project I ran into many roadblocks and issues that I had to overcome in my lack of knowledge with the technology, as well as how certain aspects would affect the client’s experience in using the program. The difficulty of the project would vary from individual to individual as any one student may have more experience than another, but mostly from the fact that the amount of work or design plan that the individual uses to complete the project with the outlined criteria. As an overall, I found working on the project in this capacity in which I was actively learning - demonstrative of the various benefits Project Based Learning holds. This is a technique in academic development in which students learn about a concept or practice with aspects of that concept at their own pace in a project based format - allowing for individualized growth and experience. Having the freedom to implement or design in any way with the few constraints detailed in the project’s description also introduced an element of creativity that we may not have otherwise had; I feel my knowledge on the topic of Socket Programming has grown vastly in completing the project and I will remain to have useful skills where I chose to dedicate my research to in order to complete it.