**UDP Chat Room**

By

Anthony Tejada

Montclair State University

CSIT 340-02, Fall 2018

Professor Dawei Li

December 13, 2018

Contents

[Introduction 1](#_Toc532149106)

[Design & Implementation 1-2](#_Toc532149107)

[Testing 2-4](#_Toc532149108)

[Conclusion 4](#_Toc532149109)

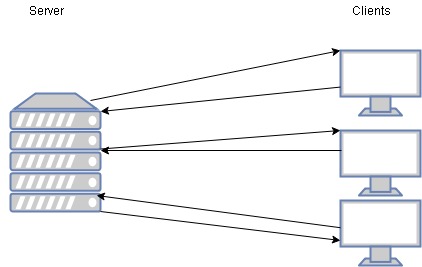
[Work 4](#_Toc532149110)

# Introduction

UDP is a transport layer protocol. This allows hosts to communicate with each other using the hosts` IP address. It is also considered unreliable because there are no mechanisms unlike TCP that allows for handling lost packets and other errors. In this project, there won’t be any need to implement packet error handling since small messages will be sent on the same network and the chances of experiencing any errors are really low. UDP also allows for multiple clients to send messages to a host server at the same time. This project will use multithreading on both server and the client files to allow the reliable processing of incoming messages coming back and forth from client to server. My main challenge will be using the threads since this is my first time using implementing one in a project. I will address this by reading multithreading documentation in the language I choose.

# Design & Implementation

The design of this project consist of one server. All clients will automatically send a “!join” message to the server in order to officially be appended to the server client list. The server here is acting as the brains of the whole system. All messages go through the server. A client cannot directly send another client a message. It has to be sent to the server first in order for it to be processed. This is important because any message sent from client to server needs to be broadcasted. The client will also be able to quit the chat by entering “!quit”. Once the message is sent, the server will remove the client from the client list and broadcast to all clients that the user has left. **Fig 1** below showcases the design of this chat room.



# 

**Fig 1: UDP chat room design**

Implementing this chat room was done using Python and the Pycharm IDE. A couple of included libraries were also imported. This included the socket, threading, random, sets and queue. The socket library was used to create the connection socket in which all communication will happen. The server had a fixed port of 5000, while all clients had a random port number between 5,001 and 10,000.The server was my own local host machine. Two functions were created to be utilized with the thread library in their respective files, listenforclient() and listenforserver(). Two threads were used to increase performance and reliability by starting a separate task to manage incoming data. The server thread was created by calling the thread() function and setting the target to the thread functions mentioned above. After it is created, the object needs to be started by the start() function in order to work properly. The server thread simply waits for incoming messages and adds the clients address and incoming message to the queue. The main thread then gets all client tuples from queue and processes them according to what the message contains. The client file thread waits for all messages from server and prints them. This simple implementation meets all requirements for the UDP chat server. Some pseudo code for the server is displayed below along with the thread implementation in **Fig 2.**

***While true***

***While queue not empty***

***Get client tuple, from queue***

***If address not in list, then add to list and broadcast client joined.***

***If message is quit, then remove client from client list and broadcast client left***

***If regular message, broadcast to all clients.***



**Fig 2: Implementation of the server thread in python.**

# Testing

Unit testing was conducted on each component of the server and client python files. This size of a project allowed me to make each requirement a component to test itself. After testing each component one by one, the files were deemed robust enough to finalize. **Table 1** shows all the test cases for the server. **Table 2** shows all the test cases for the client side. Pictures with corresponding test numbers are also below.

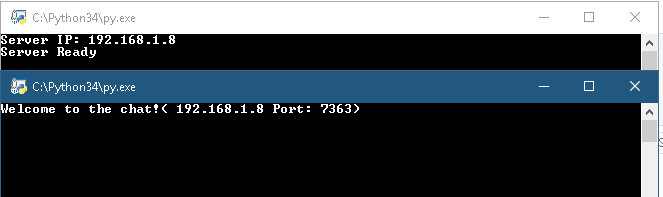
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test # | Requirement | Test Case | Expected Result | Result |
| 1 | Adding new client to list | Starting client and having server print the new client address | The server will print the clients address as soon as client starts. | Server printed the new clients connection address |
| 2 | Send welcome message to client | Start client and wait for message to arrive | Message will appear when client starts | Message arrived upon starting client |
| 3 | Broadcast when new client joins | Start a second client | First client will get message that new client joined | First client received new client join |
| 4 | Broadcast when client leaves | Enter “!quit” | All active clients receive client left message. | All active clients received the message of a certain client address leaving |
| 5 | Broadcast regular message to all active clients | Send message after starting client | All clients receive message with the unique identifier | All clients received sent message with the clients unique identifier |

**Table 1: Server test list that has test cases and the expected and actual outcome.**

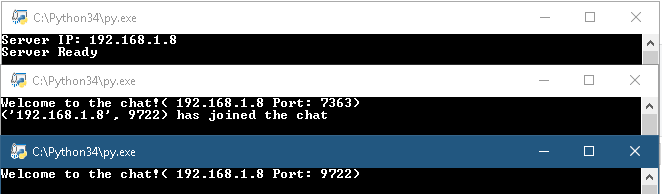
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test # | Requirement | Test Case | Expected Result | Result |
| 6 | Client auto joins chat server | Successfully send message after starting client | All other clients receive broadcast join message and the new clients message | Other clients received expected messages |
| 7 | Gracefully quit client | User types “!quit” | All clients get, client left broadcast message, and can continue to send and receive messages. | All clients got broadcast and continued to send messages. |
| 8 | Client can receive messages at the same time via threading | Send message and wait for a message from server(broadcast or client message) | Other clients will receive your message and server message will arrive. | Message was successfully sent and a broadcast message was received when client left. |

**Table 2: Client test list that has test cases and the expected and actual outcome.**

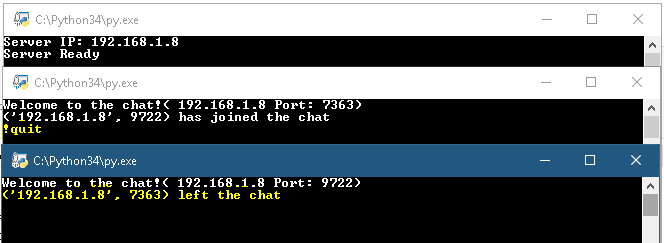
## Test case results



**Fig 3: Test #2 output**



**Fig 4: Test #3 output**



**Fig 5: Test #7 output**

# Conclusion

To conclude, a python UDP chat room was created. Both the server and client files had to be connected via sockets. Then the server was built to be the mediator of the chat room, handling all messages. The client was then built to connect to the server and be able to send and receive messages from it. This project proved to be difficult at first since my experience with threading was limited. Once I gathered the concept of threading and how it works in memory, I was able to implement a robust chat room. In addition to learning about threading, I also learned a lot about sockets and how they are used to open a lane of communication for devices on a network. I also sharpened my python skills working on this project and realized how easy programming on python can be with all the simple useful modules at your disposal. If you take a look at **Table 1** and **Table 2,** it shows all the requirements that were all successfully met.

# Work

All work was implemented by myself.