Tianen Chen
tc2841
3/1/2017

UDPCHAT README

How to run the program:
First, it is important to know that all commands and nicknames are case-sensitive. The program is run using Python 2.7.6. Additionally, the server must be started before starting any of the clients.

- Start the server by calling "python UdpChatServer.py <SERVERPORT>" where <SERVERPORT> is any allowed port number (e.g. 3000)
- Start each client by calling "python UdpChat.py <NICKNAME> <SERVERIP> <SERVERPORT> <USERPORT>"
    - Here, <NICKNAME> is any string without spaces. <SERVERIP> should just be 127.0.0.1 since we are always running on the same machine and google cloud does not allow us to bind to its external IP. We assume that every client knows the <SERVERPORT> value and uses the same server port as called above. Lastly, <USERPORT> can be any available port on the machine.
- Send messages via "send <TARGETUSER> <MESSAGE>"
    - Here, <TARGETUSER> will be any user displayed on the online list. If a user is not registered in the chat client. The program will return an error. The <MESSAGE> can be any string that doesn't include the pipe symbol (|). This is because I have structured my entire data structure sent to and from sockets around the pipe symbol.
- Deregister using the command "dereg <NICKNAME>"
    - Users can only deregister themselves
- Re-register by using the command "reg <NICKNAME>"
    - Users can only re-register themselves.
- Quit the program in any user or server by hitting ctrl+c (Keyboard Interrupt).

Documentation and method:

The program operates using 2 main sets of code: a client and a server. The client uses 2 threads to continually send and receive messages. The client takes raw input from the user and sends it out whenever a command occurs. Additionally, it sends out information about itself when registration or deregistration or errors occur. The server's primary purpose is to listen to messages from each client, respond to errors, and broadcast the lists of people who are registered and people who are online to every client. Essentially, the server "handles information."

The transfer of direct messages when both clients are online is done client-to-client via UDP connection. Otherwise, all messages get sent through the server. The server makes sure to update every client based on the information given to it. The primary data structure in my program is split up using pipe symbols. For example, when receiving a direct message, the client program receives: [KEY|FROMNAME|FROMIP|FROMPORT|MESSAGE]. The KEY is basically a command that tells the client that this should be read as a direct message. The

FROMNAME tells us who it's from, and the FROMIP and FROMPORT is the information about who it's from for our socket.. The MESSAGE is just a string. The sending of strings to and from sockets is the crux of my program. As far as I know, in python, all messages must be encoded as a string. Thus, it was important for me to find a suitable data structure that encompassed all necessary operations while still being a string.

The algorithms in this program are pretty simple. Assuming we are not handling massive amounts of clients (+50 clients), the program operates cleanly by just iterating over every list and performing checks within each subroutine. An important feature of this program is acknowledgements. There are two scenarios specified in the assignment where acknowledgements come into play. First, whenever a client-to-client message is received, an ACK must be sent. That ACK must be read before 500 msecs or else the message is timed out and the sender instead sends the message to the server. Within my program, I accomplished this using the python time module and a while loop. Secondly, whenever a client deregisters, an ACK must be received from the server. If the client does not receive the ACK from the server within 6 tries, each of 500 msecs long, the program closes. Once again, we implement the python time module and a while loop. However, in order to ensure that it goes through 6 times, we incorporate a "count" within our while loop that only goes up to 6 before killing the program.

Deregistration occurs in two forms. The first form is a deregistration using the "dereg" command. The dereg command eliminates the user from the users currently online. However, the user can simply re-register using the "reg" command. The second form is using a keyboard interrupt (ctrl+c). The user gets deregistered but cannot log back in using the "reg" command. They must start the program back up from the command line using the correct port and IP.

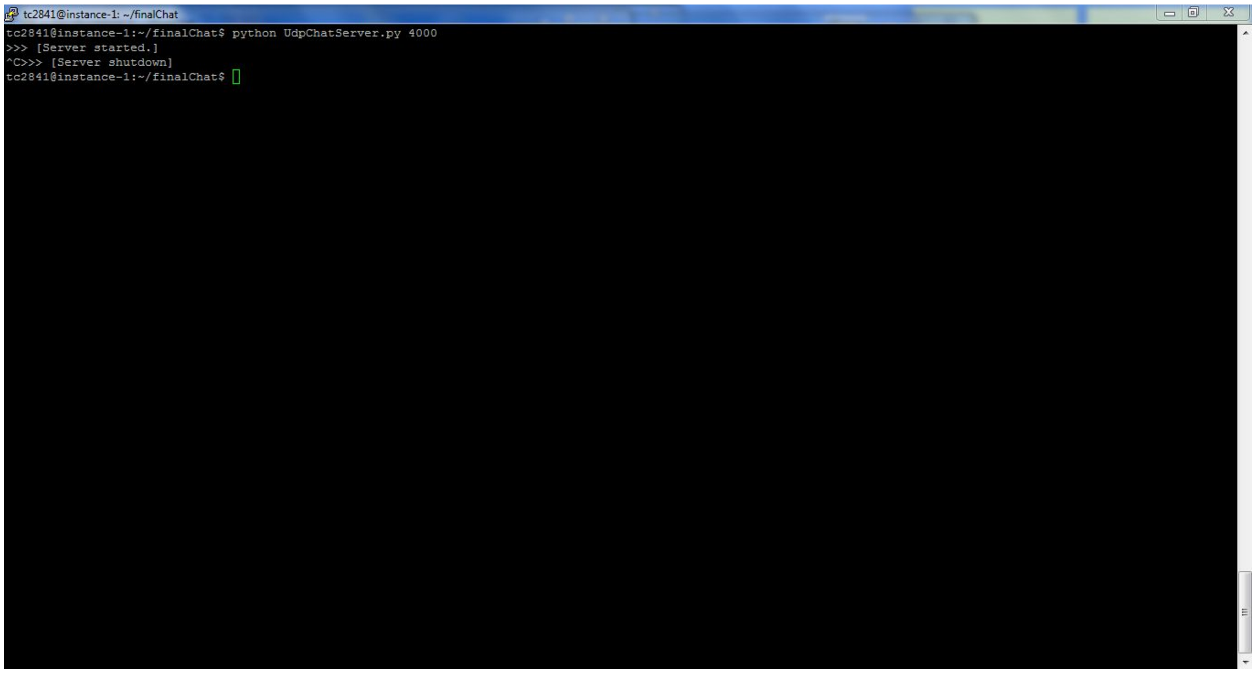
Known bugs/flaws:

- The first flaw of this program is that we assume the clients all know the server IP and port. Since we are using the localhost as an IP, this is fine. Also, the assignment specifies that all users know the server IP and port.
- When a user is de-registered, the online table still updates for them. I left it in because like facebook chat, when a user turns his chat offline, he can still see who's online and offline
- Everything is case sensitive.
- Cannot use the pipe symbol because of the way the data structure is assembled.
- Everything uses simple linear search.
- You don't know if you connected to a server with a random port. When connecting to a random server port that isn't running UdpChatServer.py, the user has no idea if it connected or not except for the fact that the online list never displays.
- When signing on with a nickname already in use, the person who messed up signing on must follow the prompts on screen to re-sign up.
- Unlike exactly test case 2 on the assignment, my program automatically exits every client connected to the server if the said server is closed. Test case 2 has the client try to send messages before realizing the server is down. In a sense, my code is more efficient because all clients "catch" that the server is closed almost immediately.

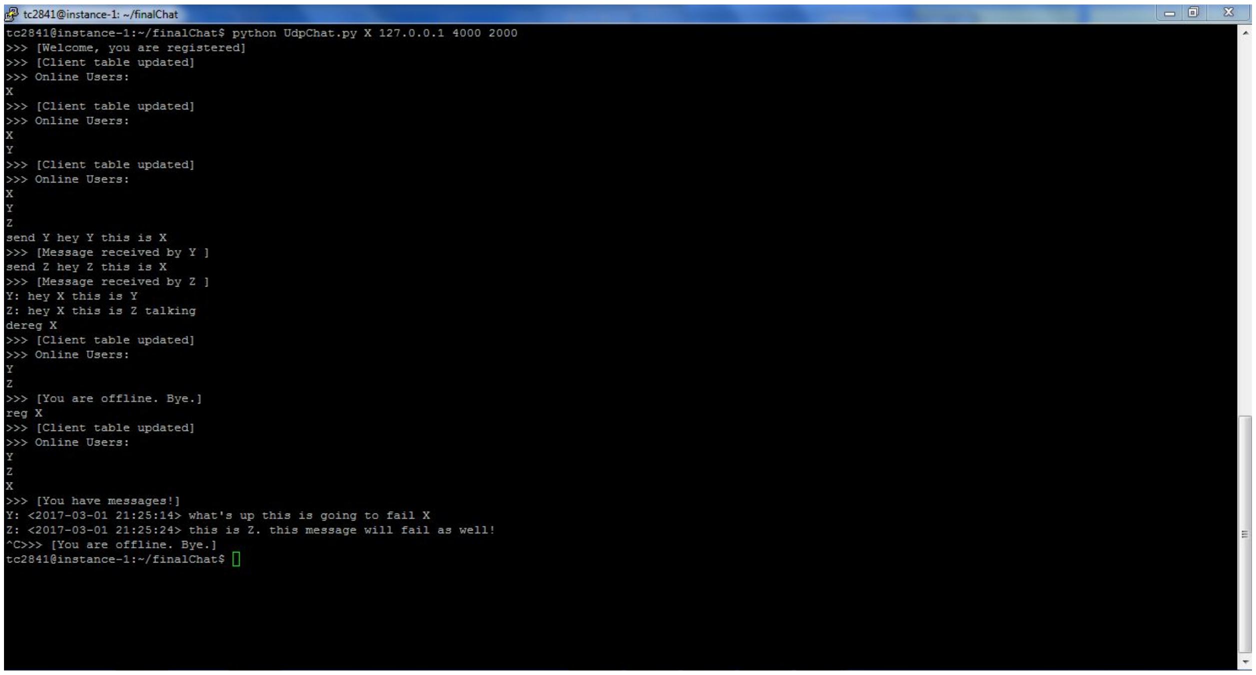
- I haven't tried all forms of typing in nonsense to the command line. More than likely, some form of nonsense will crash my program.

SCREENSHOTS OF TEST CASES

TESTCASE 1: (IN ORDER: SERVER, X, Y, Z)

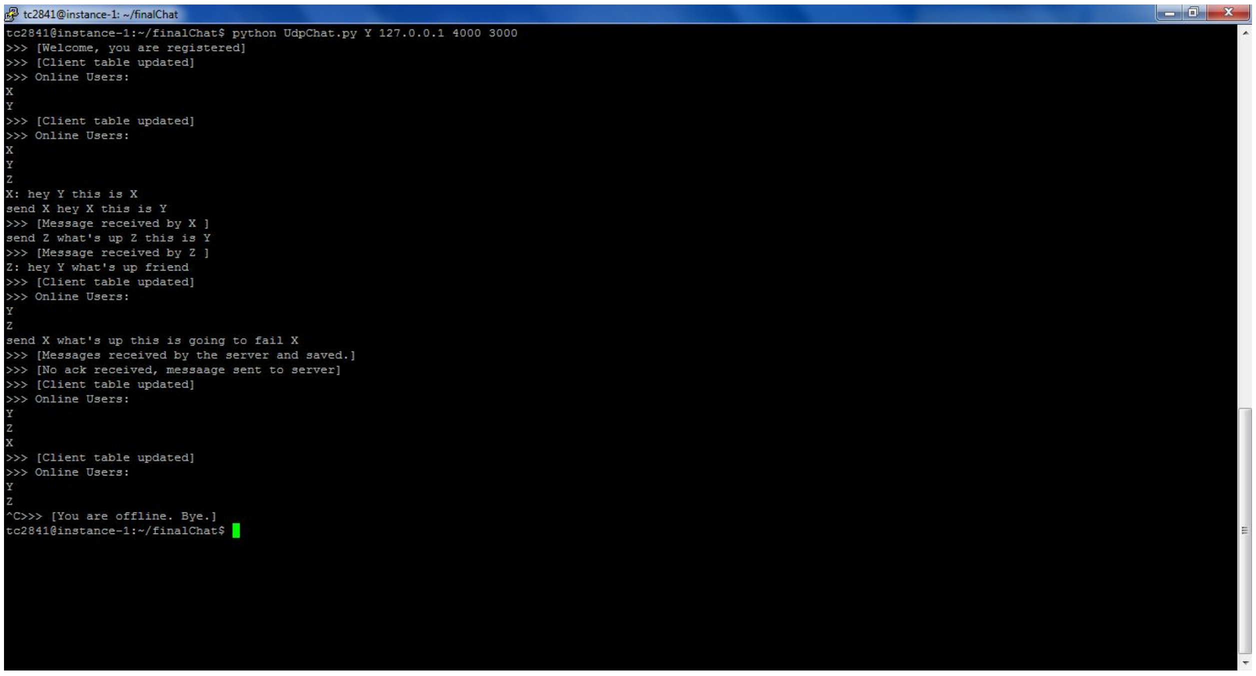


```
tc2841@instance-1: ~/finalChat
tc2841@instance-1:~/finalChat$ python UdpChatServer.py 4000
>>> [Server started.]
^C>>> [Server shutdown]
tc2841@instance-1:~/finalChat$
```



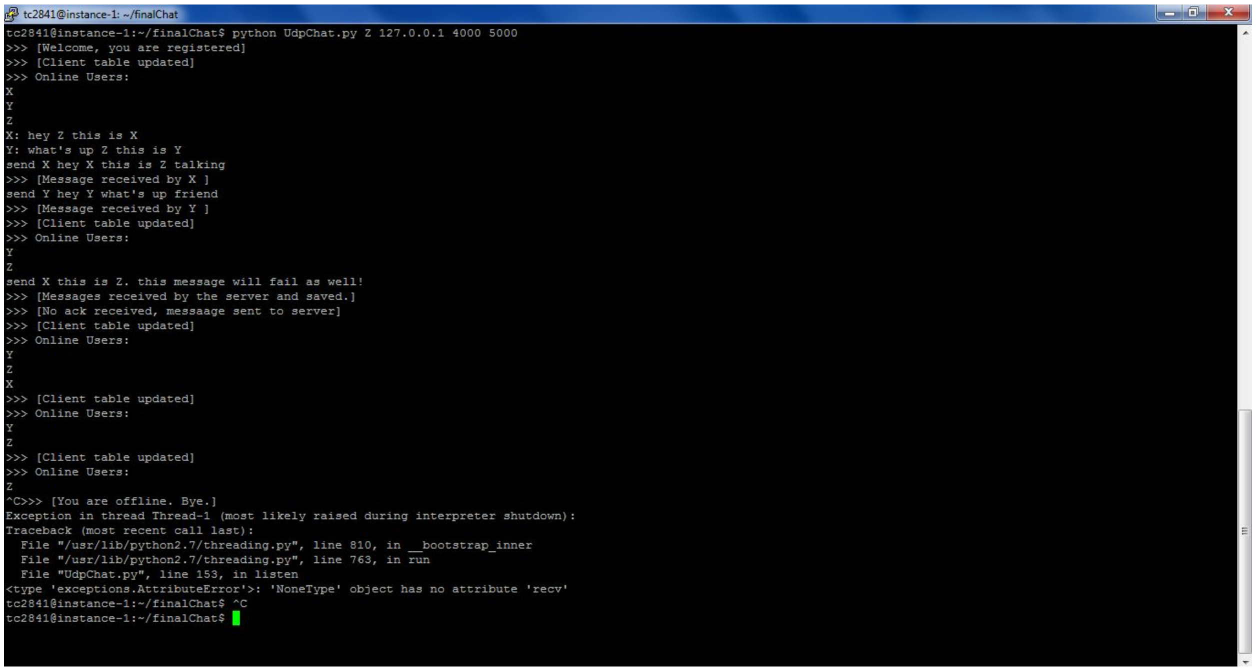
```
tc2841@instance-1: ~/finalChat
tc2841@instance-1:~/finalChat$ python UdpChat.py X 127.0.0.1 4000 2000
>>> [Welcome, you are registered]
>>> [Client table updated]
>>> Online Users:
X
>>> [Client table updated]
>>> Online Users:
X
Y
>>> [Client table updated]
>>> Online Users:
X
Y
Z
send Y hey Y this is X
>>> [Message received by Y ]
send Z hey Z this is X
>>> [Message received by Z ]
Y: hey X this is Y
Z: hey X this is Z talking
dereg X
>>> [Client table updated]
>>> Online Users:
Y
Z
>>> [You are offline. Bye.]
reg X
>>> [Client table updated]
>>> Online Users:
Y
Z
X
>>> [You have messages!]
Y: <2017-03-01 21:25:14> what's up this is going to fail X
Z: <2017-03-01 21:25:24> this is Z. this message will fail as well!
^C>>> [You are offline. Bye.]
tc2841@instance-1:~/finalChat$
```

Tianen Chen
tc2841
3/1/2017



tc2841@instance-1: ~/finalChat

```
tc2841@instance-1:~/finalChat$ python UdpChat.py Y 127.0.0.1 4000 3000
>>> [Welcome, you are registered]
>>> [Client table updated]
>>> Online Users:
X
Y
>>> [Client table updated]
>>> Online Users:
X
Y
Z
X: hey Y this is X
send X hey X this is Y
>>> [Message received by X ]
send Z what's up Z this is Y
>>> [Message received by Z ]
Z: hey Y what's up friend
>>> [Client table updated]
>>> Online Users:
Y
Z
send X what's up this is going to fail X
>>> [Messages received by the server and saved.]
>>> [No ack received, messaage sent to server]
>>> [Client table updated]
>>> Online Users:
Y
Z
X
>>> [Client table updated]
>>> Online Users:
Y
Z
^C>>> [You are offline. Bye.]
tc2841@instance-1:~/finalChat$ 
```



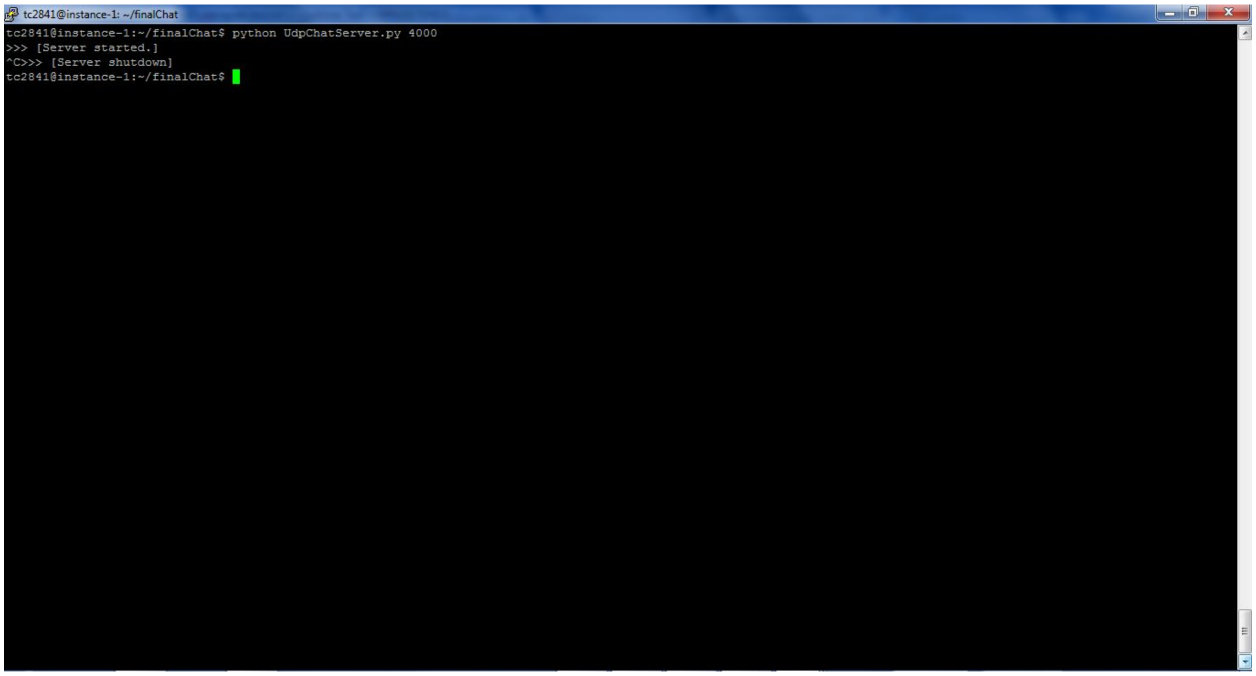
tc2841@instance-1: ~/finalChat

```
tc2841@instance-1:~/finalChat$ python UdpChat.py Z 127.0.0.1 4000 5000
>>> [Welcome, you are registered]
>>> [Client table updated]
>>> Online Users:
X
Y
Z
X: hey Z this is X
Y: what's up Z this is Y
send X hey X this is Z talking
>>> [Message received by X ]
send Y hey Y what's up friend
>>> [Message received by Y ]
>>> [Client table updated]
>>> Online Users:
Y
Z
send X this is Z. this message will fail as well!
>>> [Messages received by the server and saved.]
>>> [No ack received, messaage sent to server]
>>> [Client table updated]
>>> Online Users:
Y
Z
X
>>> [Client table updated]
>>> Online Users:
Y
Z
>>> [Client table updated]
>>> Online Users:
Z
^C>>> [You are offline. Bye.]
Exception in thread Thread-1 (most likely raised during interpreter shutdown):
Traceback (most recent call last):
  File "/usr/lib/python2.7/threading.py", line 810, in __bootstrap_inner
  File "/usr/lib/python2.7/threading.py", line 763, in run
  File "UdpChat.py", line 153, in listen
<type 'exceptions.AttributeError'>: 'NoneType' object has no attribute 'recv'
tc2841@instance-1:~/finalChat$ ^C
tc2841@instance-1:~/finalChat$ 
```

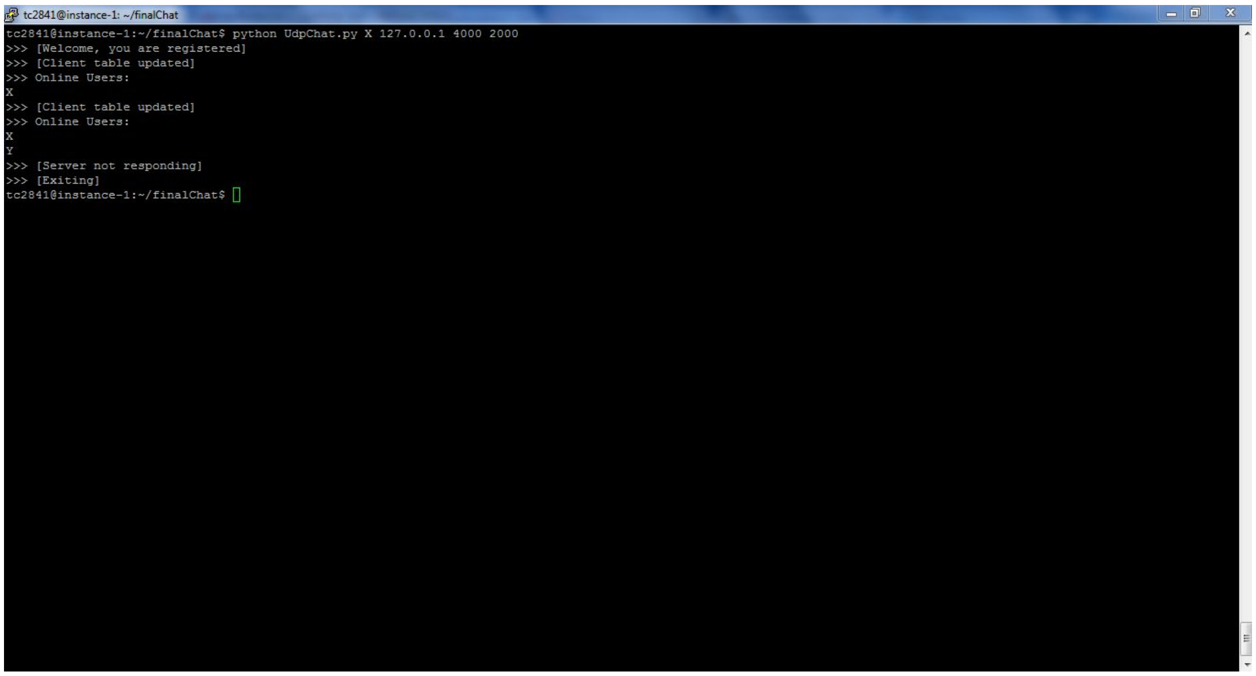
Tianen Chen
tc2841
3/1/2017

TEST CASE 2: (IN ORDER: SERVER, X, Y)

Tianen Chen
tc2841
3/1/2017

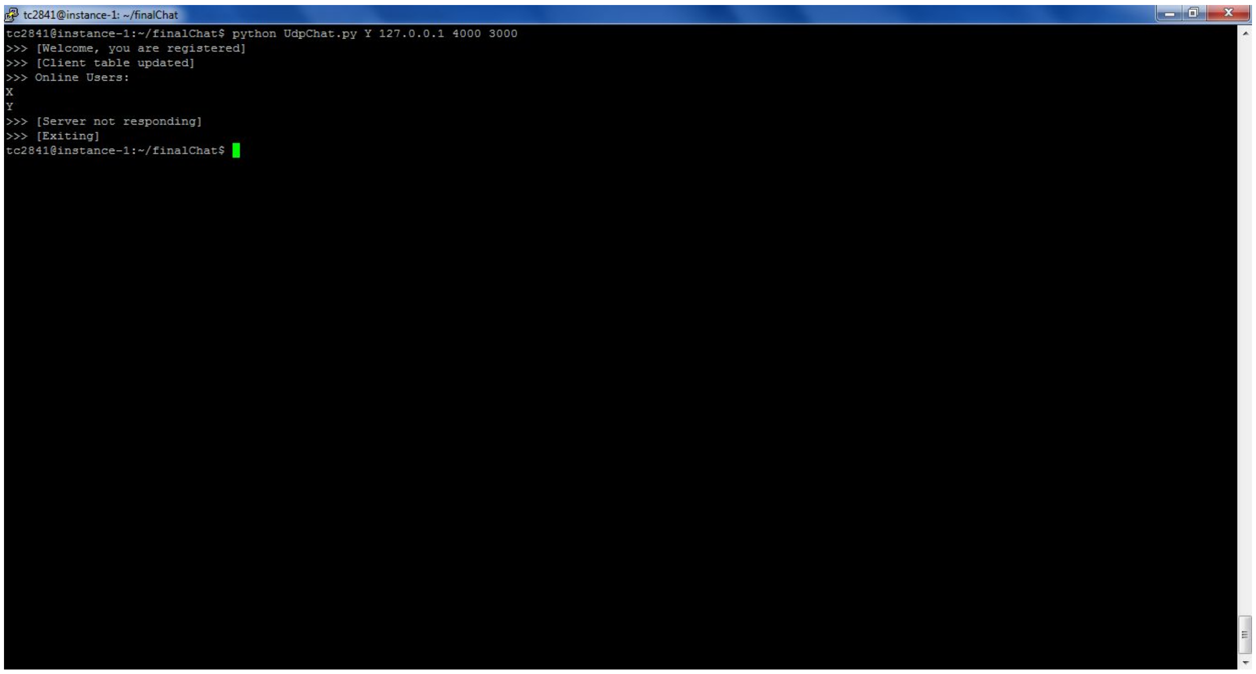


```
tc2841@instance-1:~/finalChat$ python UdpChatServer.py 4000
>>> [Server started.]
^C>>> [Server shutdown]
tc2841@instance-1:~/finalChat$
```



```
tc2841@instance-1:~/finalChat$ python UdpChat.py X 127.0.0.1 4000 2000
>>> [Welcome, you are registered]
>>> [Client table updated]
>>> Online Users:
X
>>> [Client table updated]
>>> Online Users:
X
Y
>>> [Server not responding]
>>> [Exiting]
tc2841@instance-1:~/finalChat$
```

Tianen Chen
tc2841
3/1/2017



tc2841@instance-1: ~/finalChat

```
tc2841@instance-1:~/finalChat$ python UdpChat.py Y 127.0.0.1 4000 3000
>>> [Welcome, you are registered]
>>> [Client table updated]
>>> Online Users:
X
Y
>>> [Server not responding]
>>> [Exiting]
tc2841@instance-1:~/finalChat$
```