ECE470
Project 2 – Part 4
John Koontz

Command Line Chatrooms

Command Line Chat Rooms is a lightweight application for communicating with other users across a network. Upon start up a user is connected to the lobby where they can create an account to have a personal display name, sign into an existing account, list all active chatrooms, join an active chat room, or create and join their own. Once inside a chat room a user will see the previous 10 chats within the room. The can then send a new chat to be sent to everyone in the room. When a user is done with a chatroom, they can leave by entering the quit command and return to the lobby.

Goals

- User account creation
- Login and Logout
- Chatroom creation.
- Listing of active chat rooms.
- Joining a chatroom
- Allow for multiple chat rooms to be active at a given time.
- Multiple users in a single chat room at a time.
- Messages sent in a room are sent to all other users in that room.
- Minimal latency
- Simple interface that allows for an enjoyable chat experience
- Host on cloud server

Assumptions

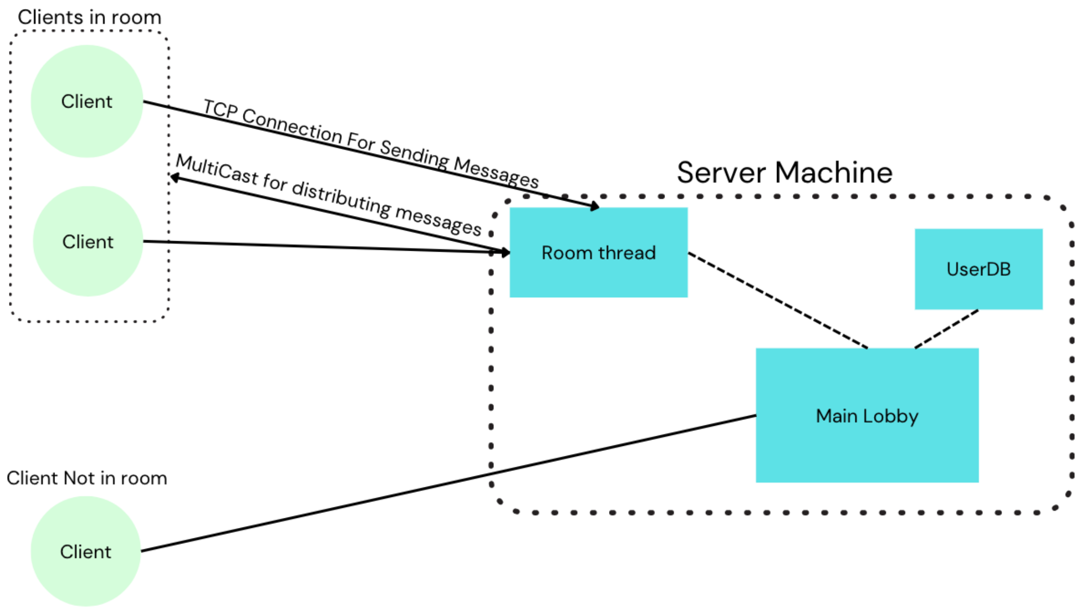
- Chats will arrive in the correct order.

Constraints

- Power of the machine running the server may affect latency.

Architecture

The application consists of a server and a client. The server facilitates user management, room creation and deletion, and the routing of chat messages. The client serves as an interface for interacting with the server and communicating in chat rooms. Clients will connect to the chat server main lobby via TCP. Once they join a room the client will then connect to a room thread via TCP and join an associated multicast group for receiving messages from other clients in the room.



Design

Models:

Command line Chatrooms relies on three main data models. User represents a user, containing a username and password. Chat represents a chat message and contains the creator of the chat, the room that chat was sent in, contents of the chat, and the date and time it was sent. Room represents a chat room and contains the name of the room, a short description, a password, and an owner referencing the creator of the room.

User

Name *string*	**Password** string

Chat

Creator (references username) *string*	**Room** *int*	**Message** *string*	**Datetime** *string*

Room

Name *string*	**Description** *string*	**Password** *string*	**Owner (references username)** *string*

Client:

The client relies on one main module called *ClientOperations*. This module is initialized, and its *run* method is called by a simple program *Client* which is the program executed by a user to start the interface. *ClientOperations* contains the following properties:

TCPcomm – a TCP communication object that takes a TCP socket and facilitates connection, message sending and receiving, and connection closure.

MCsoc – a Multicast socket value.
MCcomm- a UDP Multicast communication object that takes a Multicast socket and facilitates connection, message sending and receiving, and connection closure.
currentUser- a user object for the current user.
currentRoom- a room name of the current room.
prevChats- a list of the past 10 chats sent within the current room.

ClientOperations contains the following methods:

login- sends a login request to the server.
logout- sends a logout request to the server.
createUser- sends a create user request to the server.
join- sends a join request to the server and waits for a response with the TCP and Multicast connection information. Once the connection information is received it then disconnects the client from the lobby and reconnects to the chatrooms TCP and Multicast. Once connected it waits to receive a TCP message with the previous 10 chats within the room and reads them into *prevChats.*
list- sends a list request to the server, receives a response with all active rooms, and prints them out.
sendChat- sends a chat request to a chat server.
addPrevChat- adds a chat to *prevChats* removing the oldest chat if there is more than 10 in the list.
printChat- prints all previous chats and a prompt for the user to input a new chat or quit leave the room.
return_to_lobby- disconnects TCP and Multicast connections to a chat server and reconnects to the lobby via TCP.
chatUI- a interface that accepts input from the user for writing a message or leaving the room. Loops until the user enters the quit command. Uses select to get data from *stdin* or from the Multicast connection. If input comes from the Multicast connection the chat is added to *prevChats* and *printChat* is called. If input comes from *stdin* the loop is ended if it is the quit command or the input is made into a chat message that is sent to the chat server over the TCP connection. Once the loop is broken prevChats is reset and *return_to_lobby* is called.
run- an interface that takes a command from the user and calls the proper methods.

Server:

The server relies on two main modules called *LobbyOperations* and *ChatOperations*. The *LobbyOperations* module is initialized, and its *run* method is called by a simple program *LobbyServer* which is the program executed by a user to start the lobby server. *ChatOperations* module is initialized, and its *run* method is called by a simple program *ChatServer* which is the program executed by *LobbyOperations* to start a chat server. *LobbyOperations* contains the following properties:

server- a TCP server.
inputs- list of inputs used by select.
outputs- list of outputs used by select.

Comm- a TCP communication object that takes a TCP socket and facilitates connection, message sending and receiving, and connection closure.
currentUsers- list of current users.
activeRooms- list of active rooms and their connection information.
UserDB- a UserDB object that is used to read and write users to a file that serves as the database.

LobbyOperations contains the following methods:

login- looks up a username in the database and compares the passwords. If the passwords match the user is added to *currentUsers*. Sends a success or error message to the client.
logout- removes a user from *currentUsers.*
join- checks if the requested room is in activeRooms. If it is, the rooms connection information is sent to the client. Otherwise, connection information is generated, and new thread is started with the target of *ChatServer* and with arguments of the new connection information. The connection information is then sent to the client.
list- a list response message is sent to the client containing all the rooms in *activeRooms.*
createUser- *UserDBs newuser* method is called with the requested username and password. This adds a user to the database.
run- a main loop that uses select to receive requests from any of the connected clients and routes the request to the correct operation.

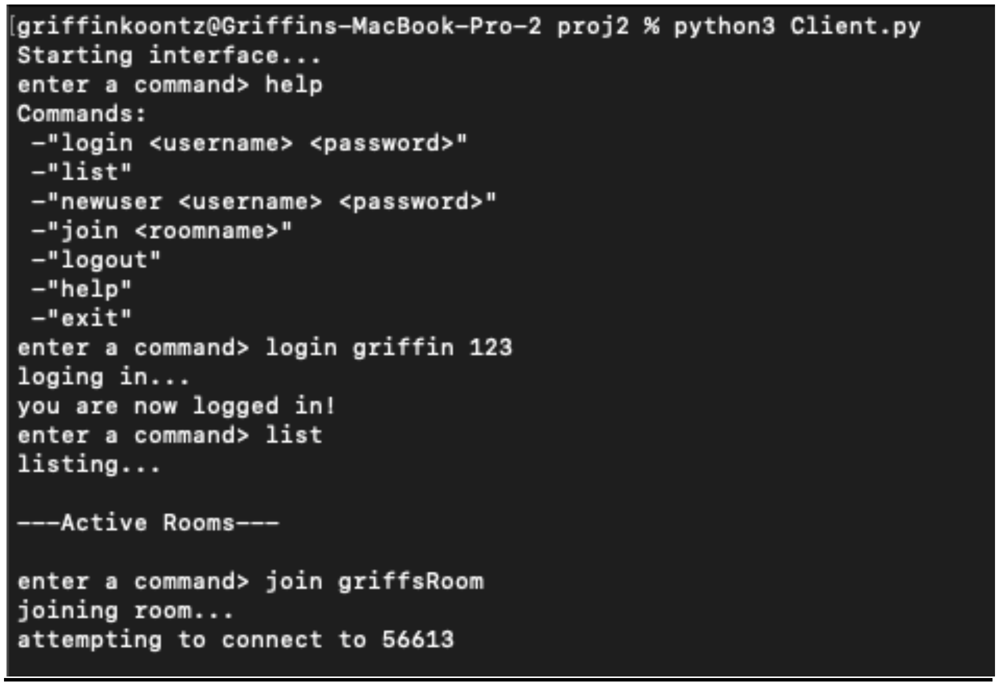
ChatOperations contains the following properties:

TCPserver- a TCP server.
inputs- list of inputs used by select.
outputs- list of outputs used by select.
TCPComm- a TCP communication object that takes a TCP socket and facilitates connection, message sending and receiving, and connection closure.
MCComm a UDP Multicast communication object that takes a Multicast socket and facilitates connection, message sending and receiving, and connection closure.
prevChats- a list of the past 10 chats sent within the room.

ChatOperations contains the following methods:

addPrevChat- adds a chat to *prevChats* removing the oldest chat if there is more than 10 in the list.
sendPrevChats- sends a TCP message to the client containing the chats in *prevChats*.
broadcastChat- sends a Multicast chat message.
receivedChat- adds a received chat to prevChats and then sends it to the room with *broadcastChat*.
run- a main loop that uses select to receive requests from any of the connected clients and routes the request to the correct operation. If there are no connections, the loop terminates, and all sockets are closed.

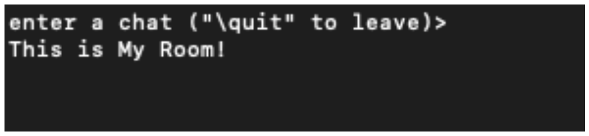
<u>**Sample Output**</u>



```
[griffinkoontz@Griffins-MacBook-Pro-2 proj2 % python3 Client.py
Starting interface...
enter a command> help
Commands:
 -"login <username> <password>"
 -"list"
 -"newuser <username> <password>"
 -"join <roomname>"
 -"logout"
 -"help"
 -"exit"
enter a command> login griffin 123
loging in...
you are now logged in!
enter a command> list
listing...

---Active Rooms---

enter a command> join griffsRoom
joining room...
attempting to connect to 56613
```

Figure 1: a new client is started

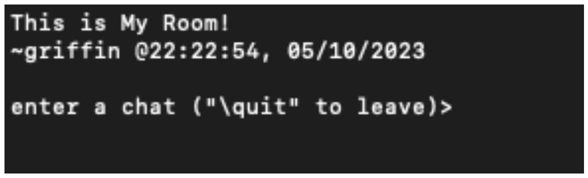
A new client has started, and the user has logged in. Active rooms are then listed and show no currently active rooms. The user then requests to join a room that does not yet exist, that means a new one will be created.



```
enter a chat ("\quit" to leave)>
This is My Room!
```

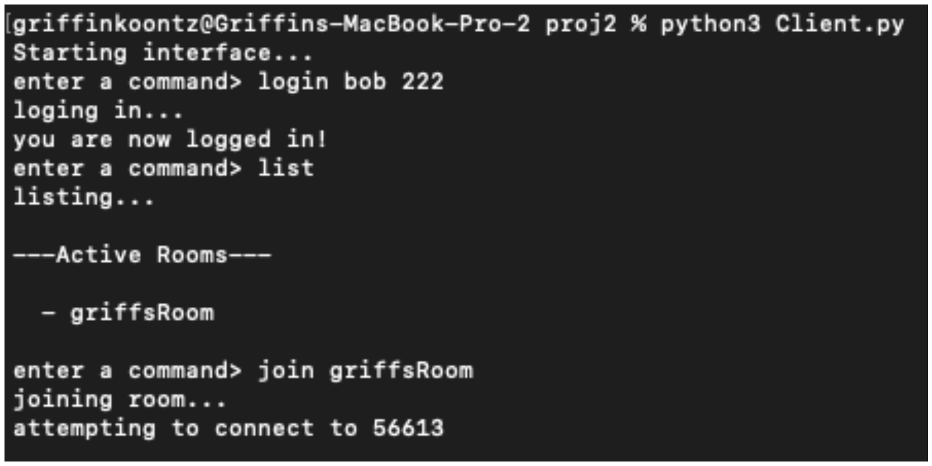
Figure 2: in a brand-new room

The room is then entered. There are no previous chats, but the user is about to send one.



```
This is My Room!
~griffin @22:22:54, 05/10/2023

enter a chat ("\quit" to leave)>
```

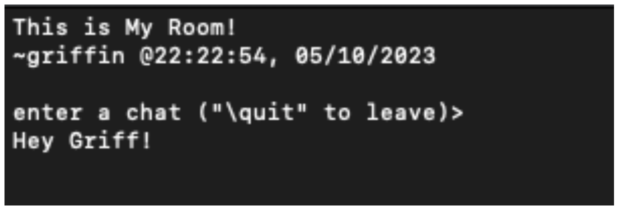
Figure 3: the chat was sent



```
[griffinkoontz@Griffins-MacBook-Pro-2 proj2 % python3 Client.py
Starting interface...
enter a command> login bob 222
loging in...
you are now logged in!
enter a command> list
listing...

---Active Rooms---

  - griffsRoom

enter a command> join griffsRoom
joining room...
attempting to connect to 56613
```

Figure 4: another client comes online

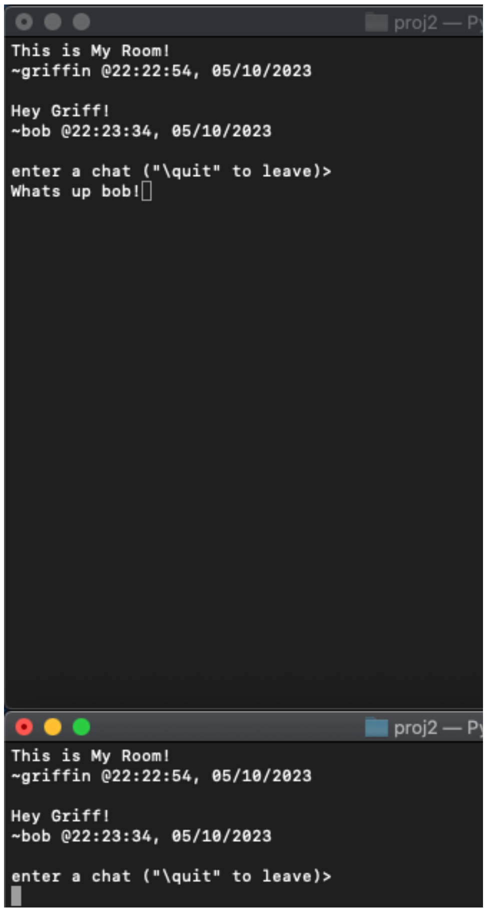
Now another client is started. The user logs in and lists the active rooms. The room started by the other user is seen and the user requests to join it.



```
This is My Room!
~griffin @22:22:54, 05/10/2023

enter a chat ("\quit" to leave)>
Hey Griff!
```

Figure 5: the room is joined

The room is joined and the previous message from 'griffin' can be seen. The user then is about to send their own message.

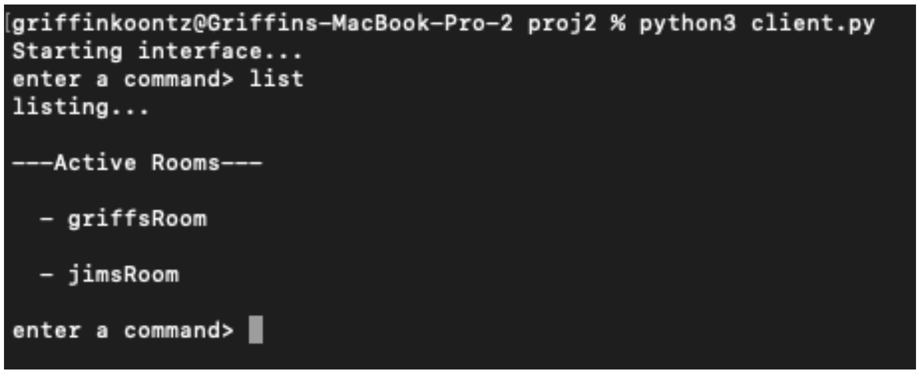


```
This is My Room!
~griffin @22:22:54, 05/10/2023

Hey Griff!
~bob @22:23:34, 05/10/2023

enter a chat ("\quit" to leave)>
Whats up bob!
```

```
This is My Room!
~griffin @22:22:54, 05/10/2023

Hey Griff!
~bob @22:23:34, 05/10/2023

enter a chat ("\quit" to leave)>
```

Figure 6: the everyone receives the chat

The chat is received by all users.



```
[griffinkoontz@Griffins-MacBook-Pro-2 proj2 % python3 client.py
 Starting interface...
 enter a command> list
 listing...

 ---Active Rooms---

   - griffsRoom

   - jimsRoom

 enter a command>
```

Figure 7: two active rooms

Here a list shows multiple active rooms.

<u>Conclusion</u>

In conclusion, Command Line Chatrooms successfully serves its purpose of allowing for application for communicating with other users across a network. Users can create and join rooms and send chats to others in the room. The project did not meet the goal to be hosted on a permanent cloud server, however it is at the point where it could be done with minimal changes. To further expand on this project, I would add functionality to store a room name and previous chats in a database. This would allow for permanent rooms that could be rejoined even if it is not currently active. Other functionality that could come along with this improvement would be to allow private rooms that require a password to enter. Despite the missed goal and potential future improvements, the Command Line Chatrooms application has met most of its goals and reaches its core functionality.

Appendix

Client:

```
import socket
from ClientOperations import ClientOperations
import time

if __name__ == "__main__":
    # create the socket
    # defaults family=AF_INET, type=SOCK_STREAM, proto=0, filno=None
    commsoc = socket.socket()

    # connect to localhost:5000
    commsoc.connect(("localhost",50000))

    # run the application protocol
    print('Starting interface...')
    Operations = ClientOperations(commsoc)
    Operations.run()

    # close the comm socket
    commsoc.close()
```

ClientOperations:

```
from Comm import Comm
from MultiComm import MultiComm
from models.Message import Message
from models.User import User
from models.Room import Room
from models.Chat import Chat
import models.Payloads as Payloads
import socket
import select
import sys
import os
import struct

class ClientOperations(object):

```

```python
    def __init__(self, s : socket):
        self.TCPcomm = Comm(s)
        self.MCsoc = socket.socket()
        self.MCcomm = MultiComm()
        self._currentUser = User()
        self._currentRoom = ''
        self._prevChats = []

    def _return_to_lobby(self):
        self.TCPcomm.close()
        self.MCcomm.close()

        s = socket.socket()
        # connect to localhost:5000
        s.connect(("localhost",50000))
        self.TCPcomm = Comm(s)

    def login(self, user : User):
        request = Message()
        request.setType('LGIN')
        request.setPayload(Payloads.LGIN(user))
        self.TCPcomm.sendMessage(request)
        response = self.TCPcomm.recvMessage()
        if response.getType() == 'GOOD':
            self._currentUser = user
            print(response.getPayload()['message'])
            return 1
        elif response.getType() == 'ERRO':
            print(response.getPayload()['message'])
            return -1
        else:
            raise Exception('Unexpected Response')

    def logout(self, user : User):
        request = Message()
        request.setType('LOUT')
        request.setPayload(Payloads.LOUT(user))
        self.TCPcomm.sendMessage(request)
        response = self.TCPcomm.recvMessage()
        if response.getType() == 'GOOD':
            self._currentUser = User()
            print(response.getPayload()['message'])
            return 1
        elif response.getType() == 'ERRO':
            print(response.getPayload()['message'])
            return -1
        else:
            raise Exception('Unexpected Response')
    def createUser(self, user : User):
        request = Message()
        request.setType('CUSR')
        request.setPayload(Payloads.CUSR(user))
        self.TCPcomm.sendMessage(request)
        response = self.TCPcomm.recvMessage()
        if response.getType() == 'GOOD':
            print(response.getPayload()['message'])
            return 1
        elif response.getType() == 'ERRO':
            print(response.getPayload()['message'])
            return -1
        else:
            raise Exception('Unexpected Response')

    def join(self, room : Room):
        request = Message()
        request.setType('JOIN')
        request.setPayload(Payloads.JOIN(room))
        self.TCPcomm.sendMessage(request)
        response = self.TCPcomm.recvMessage()
        if response.getType() == 'CONN':
            port = response.getPayload()['TCPport']
            # disconect from lobby server
            self.TCPcomm.close()
            # connect to room TCPserver
            commsoc = socket.socket()
            print(f'attempting to connect to {port}')
            commsoc.connect(("localhost", port))
            self.TCPcomm = Comm(commsoc)

            # connect to the room Multicast
            multicast_group = response.getPayload()['GRP']
            multicast_port = response.getPayload()['MCport']

            self.MCsoc = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
            # Set the SO_REUSEPORT option
            self.MCsoc.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEPORT, 1)
            # Bind the socket to the server address
            self.MCsoc.bind(('', multicast_port))
            # Tell the operating system to add the socket to the multicast group
            group = socket.inet_aton(multicast_group)
            mreq = struct.pack('4sL', group, socket.INADDR_ANY)
            self.MCsoc.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mreq)
            # initialize MCcomm
            self.MCcomm = MultiComm(self.MCsoc, (multicast_group, multicast_port))
            # set the current room
            self._currentRoom = room.get_name()
            # receive the prevChats
            msg = self.TCPcomm.recvMessage()
            if msg.getType() == 'PREV':
                chats = msg.getPayload()['chats']
                self._prevChats = []
                for chat in chats:
                    c = Chat()
                    c.from_dict(chat)
                    self._prevChats.append(c)
            elif msg.getType() == 'ERRO':
                print(response.getPayload()['message'])
                return -1
            else:
                raise Exception('Unexpected Response')
            return 1
        elif response.getType() == 'ERRO':
            print(response.getPayload()['message'])
            return -1
        else:
            raise Exception('Unexpected Response')

    def list(self):
        request = Message()
        request.setType('LTRQ')
        request.setPayload(Payloads.LTRQ())
        self.TCPcomm.sendMessage(request)
        response = self.TCPcomm.recvMessage()
        if response.getType() == 'LTRS':
            rooms = response.getPayload()['rooms']
            print('\n---Active Rooms---\n')
            for room in rooms: # print the room names
                print(f'  - {room}\n')
            return 1
        elif response.getType() == 'ERRO':
            print(response.getPayload()['message'])
            return -1
        else:
            raise Exception('Unexpected Response')


    def sendChat(self, chat):
        c = Chat(self._currentUser.get_name(), self._currentRoom, chat)
        request = Message()
        request.setType('CHAT')
        request.setPayload(Payloads.CHAT(c))
        self.TCPcomm.sendMessage(request)
        response = self.TCPcomm.recvMessage()
        if response.getType() == 'GOOD':
            return 1
        elif response.getType() == 'ERRO':
            print(response.getPayload()['message'])
            return -1
        else:
            raise Exception('Unexpected Response')

    def addPrevChat(self, chat : Chat):
        if len(self._prevChats) < 10:
            self._prevChats.append(chat)
        else:
            self._prevChats.pop(0)
            self._prevChats.append(chat)

    def printChat(self):
        os.system('clear')
        for chat in self._prevChats:
            print(f'{chat.get_contents()}\n~{chat.get_creator()}
@{chat.get_datetime()}\n')
        print('enter a chat ("\quit" to leave)> ')

    def chatUI(self):
        loop = True
        self.printChat()
        while loop:
            inputs = [sys.stdin, self.MCsoc]
            outputs = []
            excepts = []
            ready_inputs, ready_outputs, ready_excepts = select.select(inputs,
outputs, excepts, 1)

            for ready_input in ready_inputs:
                if ready_input == sys.stdin:
                    chat = input()
                    if chat == '\quit':  # Check for '\quit'
                        print('Quitting')
                        loop = False # stop the loop
                        break
                    self.sendChat(chat)
                elif ready_input == self.MCsoc:
                    msg = self.MCcomm.recvMessage()
                    chat = Chat()
                    chat.from_dict(msg.getPayload())
                    self.addPrevChat(chat)
                    self.printChat()
        self._prevChats = []
        os.system('clear')
        self._return_to_lobby()

    def run(self):
        while True:
            command = input("enter a command> ")
            args = command.split()
            if len(args) < 1:
                continue
            elif args[0] == "login":
                if len(args) < 3:
                    print('missing arguments. -"login <username> <password>"')
                else:
                    print("loging in...")
                    self.login(User(args[1], args[2]))

            elif args[0] == "list":
                print("listing...")
                self.list()

            elif args[0] == "newuser":
                print("creating user...")
                if len(args) < 3:
                    print('missing arguments. -"newuser <name> <password>"')
                else:
                    u = User(args[1], args[2])
                    self.createUser(u)

            elif args[0] == "logout":
                print("logging out...")
                self.logout(self._currentUser)

            elif args[0] == "join":
                if len(args) < 2:
                    print('missing arguments.  -"join <roomname>"')
                else:
                    print('joining room...')
                    room = Room(args[1])
                    self.join(room)
                    self.chatUI()

            elif args[0] == "help":
                print('Commands:\n -"login <username> <password>"\n -"list"\n -
"newuser <username> <password>"\n -"join <roomname>"\n -"logout"\n -"help"\n -"exit"')

            elif args[0] == 'exit':
                print('exiting...')
                break
            else:
                print('Invalad command. -"help" for list off commands.')
```

LobbyServer:

```
import socket
from Comm import Comm
from LobbyOperations import LobbyOperations

if __name__ == "__main__":
    # create the server socket
    #  defaults family=AF_INET, type=SOCK_STREAM, proto=0, filno=None
    serversoc = socket.socket()

    # set blocking 0
    serversoc.setblocking(0)

    # bind to local host:5000
    serversoc.bind(("localhost",50000))

    # make passive with backlog=5
    serversoc.listen(5)

    # wait for incoming connections
    while True:
        print("Listening on ", 50000)

        # run ops
        lobbyOps = LobbyOperations(serversoc)
        lobbyOps.run()

    # close the server socket
    serversoc.close()
```

LobbyOperations:

```
from Comm import Comm
from models.Message import Message
from models.Room import Room
from models.User import User
import models.Payloads as Payloads
import socket
from UserDB import UserDB
import select, socket, sys
from utils import get_free_tcp_port
import threading
from ChatServer import ChatServer
import time
```

```
class LobbyOperations(object):

    def __init__(self, server):
        self._server = server
        self._inputs = [server]
        self._outputs = []
        self._Comm = Comm()
        self._currentUsers = []
        self._activeRooms = {}
        self._UserDB = UserDB()

        self.free_multicasts = [['224.3.29.71', 10000], ['224.3.29.81', 10010],
['224.3.29.81', 10020],  ['224.3.29.91', 10030],  ['224.3.29.91', 10040]]

        self._route = { 'LGIN': self.login,
                        'LOUT': self.logout,
                        'JOIN': self.join,
                        'LTRQ': self.list,
                        'CUSR': self.createUser}

    def _getRequest(self):
        req = self._Comm.recvMessage()
        if req == False:
            return False, False
        reqcmd = req.getType()
        reqPayload = req.getPayload()

        return reqcmd,reqPayload

    def _putResponseGood(self, message : str):
        resp = Message()
        resp.setType('GOOD')
        resp.setPayload(Payloads.GOOD(message))
        self._Comm.sendMessage(resp)

    def _putResponseError(self, message : str):
        resp = Message()
        resp.setType('ERRO')
        resp.setPayload(Payloads.ERRO(message))
        self._Comm.sendMessage(resp)

    def _updateActiveRooms(self):
        keys_to_remove = []
        for key in self._activeRooms.keys():
            room = self._activeRooms[key]
            if not room['thread'].is_alive():
                self.free_multicasts.append([room['GRP'], room['MCport']])
                keys_to_remove.append(key)
        for key in keys_to_remove:
            del self._activeRooms[key]

    def login(self, payload : dict):
        #compare passwords & login user
        user = User(payload['username'], payload['password'], self._Comm.get_sock())
        if self._UserDB.search(user.get_name(), user.get_password()):
            self._currentUsers.append(user)
            self._putResponseGood('you are now logged in!')
        else:
            self._putResponseError('invalad username or password.')

    def logout(self, payload : dict):
        #logout user
        for user in self._currentUsers:
            if user.get_name() == payload['username']:
                self._currentUsers.remove(user)
                self._putResponseGood('you are now logged out!')
                return
        self._putResponseError('Could not logout.')

    def join(self, payload : dict):
        self._updateActiveRooms()
        # check if chat room is active
        if payload['room'] in self._activeRooms.keys():
            # is active
            print('sending connection info for active room')
            conns = self._activeRooms[payload['room']]
            resp = Message()
            resp.setType('CONN')
            resp.setPayload(Payloads.CONN(conns['TCPport'], conns['GRP'],
conns['MCport']))
            self._Comm.sendMessage(resp)
        else:
            # start thread with new port and multicast
            TCPport = get_free_tcp_port()
            multicast = self.free_multicasts.pop()
            t = threading.Thread(name=payload['room'], target=ChatServer,
args=(TCPport, multicast))
            t.start()

            time.sleep(1)
            #send to client
            resp = Message()
            resp.setType('CONN')
            resp.setPayload(Payloads.CONN(TCPport, multicast[0], multicast[1]))
            self._Comm.sendMessage(resp)
            # add room to active rooms
            self._activeRooms[payload['room']] = {'TCPport' : TCPport, 'GRP' :
multicast[0], 'MCport': multicast[1], 'thread': t}
            print('added new active room')

    def list(self, payload : dict):
        self._updateActiveRooms()
        print('sending list')
        rooms = [str(key) for key in self._activeRooms.keys()]
        resp = Message()
        resp.setType('LTRS')
        resp.setPayload(Payloads.LTRS(rooms))
        self._Comm.sendMessage(resp)

    def createUser(self, payload : dict):
        self._UserDB.newUser(payload['username'], payload['password'])
        self._putResponseGood(f'user created')

    def run(self):
        while self._inputs:
            readable, writable, exceptional = select.select(
                self._inputs, self._outputs, self._inputs)

            for s in readable:
                if s is self._server:
                    connection, client_address = s.accept()
                    connection.setblocking(0)
                    self._inputs.append(connection)
                    print(f'new connection {client_address}')
                else:
                    self._Comm = Comm(s)
                    cmd, payload = self._getRequest()
                    if cmd: #cmd and payload
                        self._route[cmd](payload)
                    else:
                        self._inputs.remove(s)
                        s.close()
                        print(f'Closed connection to {s}')

            for s in exceptional:
                self._inputs.remove(s)
                if s in self._outputs:
                    self._outputs.remove(s)
                s.close()
                print(f'Closed connection to {s}')
```

ChatServer:

```python
import socket
import struct
from Comm import Comm
from MultiComm import MultiComm
from ChatOperations import ChatOperations

def ChatServer(port, multicast):
    print('New Chat Server Thread Started...')
    #  create the TCP socket
    #  defaults family=AF_INET, type=SOCK_STREAM, proto=0, filno=None
    TCPsoc = socket.socket()

    # set blocking 0
    TCPsoc.setblocking(0)

    # bind to local host:5000
    TCPsoc.bind(("localhost",port))

    # make passive with backlog=5
    TCPsoc.listen(5)

    # create multicast socket
    multicast_group = multicast[0]
    server_address = ('', multicast[1])

    # Create the socket
    MCsock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    # Set the time-to-live (TTL) for the socket
    ttl = struct.pack('b', 1)
    MCsock.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, ttl)


    MCComm = MultiComm(MCsock, (multicast_group, server_address[1]))

    # wait for incoming connections
    print("Chat server listening on ", port)

    # run ops
    chatOps = ChatOperations(TCPsoc, MCComm)
    chatOps.run()

    # close the server socket
    TCPsoc.close()
```

ChatOperations:

```python
from Comm import Comm
from MultiComm import MultiComm
from models.Message import Message
from models.Room import Room
from models.User import User
from models.Chat import Chat
import models.Payloads as Payloads
import socket
from UserDB import UserDB
import select, socket, sys

class ChatOperations(object):

    def __init__(self, TCPserver : socket, Multicast : MultiComm):
        # TCP
        self._TCPserver = TCPserver
        self._inputs = [TCPserver]
        self._outputs = []
        self._TCPComm = Comm()

        # Multicast
        self._MCComm = Multicast

        self._prevChats = []

        self._route = {'CHAT':self.received_chat}

    def _getRequest(self):
        req = self._TCPComm.recvMessage()
        if req == False:
            return False, False
        reqcmd = req.getType()
        reqPayload = req.getPayload()
        return reqcmd,reqPayload

    def _putResponseGood(self, message : str):
        resp = Message()
        resp.setType('GOOD')
        resp.setPayload(Payloads.GOOD(message))
        self._TCPComm.sendMessage(resp)

    def _putResponseError(self, message : str):
        resp = Message()
        resp.setType('ERRO')
        resp.setPayload(Payloads.ERRO(message))
        self._TCPComm.sendMessage(resp)

    def addPrevChat(self, chat : Chat):
        if len(self._prevChats) < 10:
            self._prevChats.append(chat)
        else:
            self._prevChats.pop(0)
            self._prevChats.append(chat)

    def sendPrevChats(self):
        msg = Message()
        msg.setType('PREV')
        chats = []
        for chat in self._prevChats:
            chats.append(chat.to_dict())
        msg.setPayload(Payloads.PREV(chats))
        self._TCPComm.sendMessage(msg)

    def brodcast_chat(self, chat : dict):
        msg = Message()
        msg.setType('CHAT')
        msg.setPayload(chat)
        self._MCComm.sendMessage(msg)

    def received_chat(self, payload : dict):
        chat = Chat()
        chat.from_dict(payload)
        self.addPrevChat(chat)

        contents =  payload['contents']
        self._putResponseGood(f'got your chat: "{contents}"')
        self.brodcast_chat(payload)

    def run(self):
        connections = 0
        while self._inputs:
            readable, writable, exceptional = select.select(
                self._inputs, self._outputs, self._inputs)

            for s in readable:
                if s is self._TCPserver:
                    connection, client_address = s.accept()
                    connection.setblocking(0)
                    self._inputs.append(connection)
                    print(f'new chat connection {client_address}')
                    connections += 1
                    # send prev chats
                    self._TCPComm = Comm(connection)
                    self.sendPrevChats()

                else:
                    self._TCPComm = Comm(s)
                    cmd, payload = self._getRequest()
                    if cmd:
                        self._route[cmd](payload)
                    else:
                        connections -= 1
                        self._inputs.remove(s)
                        s.close()
                        print(f'Closed connection to {s}')

            for s in exceptional:
                self._inputs.remove(s)
                if s in self._outputs:
                    self._outputs.remove(s)
                s.close()
                print(f'Closed connection to {s}')

            if connections == 0:
                break

        print('chat server shutting down')
        self._TCPComm.close()
        self._MCComm.close()
```

Comm:

```
import socket
from models.Message import Message

class Comm(object):
    '''
    classdocs
    '''
    BUFSIZE = 8196

    def __init__(self, s : socket = -1):
        '''
        Constructor
        '''
        self._sock = s

    def get_sock(self):
        return self._sock

    def _loopRecv(self, size: int):
        data = bytearray(b" "*size)
        mv = memoryview(data)
        while size:
            rsize = self._sock.recv_into(mv,size)
            if rsize == 0:
                return False
            mv = mv[rsize:]
            size -= rsize
        return data

    def sendMessage(self, m: Message):
        data = m.marshal()
        self._sock.sendall(data)

    def recvMessage(self) -> Message:
        try:
            m = Message()
            mtype = self._loopRecv(4)
            size = self._loopRecv(4)
            data = self._loopRecv(int(size.decode('utf-8')))
            params = b''.join([mtype,size,data])
            m.unmarshal(params)
        except:
            return False
        else:
            return m

    def close(self):
        self._sock.close()
```

MultiComm:

```
import socket
from models.Message import Message

class MultiComm(object):
    '''
    classdocs
    '''
    BUFSIZE = 8196

    def __init__(self, s : socket = -1, group = ()):
        '''
        Constructor
        '''
        self._sock = s
        self._group = group

    def sendMessage(self, m: Message):
        data = m.marshal()
        self._sock.sendto(data, self._group)

    def recvMessage(self) -> Message:
        try:
            m = Message()
            data, address = self._sock.recvfrom(1024)
            if not data:
                return False
            mtype = data[:4]
            size = data[4:8]
            data = data[8:len(data)]
            params = b''.join([mtype,size,data])
            m.unmarshal(params)
        except Exception:
            raise Exception('bad getMessage')
        else:
            return m

    def close(self):
        self._sock.close()
```

UserDB:

```
import csv
import sys

class UserDB(object):
    def __init__(self) -> None:
        self.users = {}
        file = open('users.csv', 'r')
        reader = csv.reader(file)
        for row in reader:
            self.addUser(row[0], row[1])
        file.close()

    def addUser(self, username, password):
        self.users[username] = password

    def newUser(self, username, password):
        self.addUser(username, password)
        file = open('users.csv', 'a', newline='')
        writer = csv.writer(file)
        writer.writerow([username, password])
        file.close()

    def search(self, username , password):
        try:
            p = self.users[username]
            if p == password: return True
            else: return False
        except:
            return False
```

Chat:

```
from datetime import datetime

class Chat(object):

    def __init__(self, creator = '', room = '', contents = ''):
        self._creator = creator
        self._room = room
        self._contents = contents
        self._datetime = datetime.now().strftime("%H:%M:%S, %m/%d/%Y")

    def set_creator(self, creator):
        self._creator = creator

    def get_creator(self):
        return self._creator

    def set_room(self, room):
        self._room = room

    def get_room(self):
        return self._room

    def set_contents(self, contents):
        self._contents = contents

    def get_contents(self):
        return self._contents

    def set_datetime(self):
        self._datetime = datetime.now().strftime("%m/%d/%Y, %H:%M:%S")

    def get_datetime(self):
        return self._datetime

    def from_dict(self, chat : dict):
        self._creator = chat['creator']
        self._room = chat['room']
        self._contents = chat['contents']
        self._datetime = chat['datetime']

    def to_dict(self):
        return {'creator' : self._creator, 'room' : self._room, 'contents' :
self._contents, 'datetime': self._datetime}
```

Message

```
from enum import Enum
import json
```

```python
class Message(object):
    '''
    classdocs
    '''
    # Constants
    CMDS = Enum('CMDS', {'LGIN': 'LGIN', 'LOUT': 'LOUT', 'CUSR': 'CUSR',
'ROOM':'ROOM', 'JOIN':'JOIN',
                'CONN':'CONN', 'CHAT':'CHAT', 'DELR': 'DELR', 'GOOD': 'GOOD', 'ERRO':
'ERRO','LTRQ':'LTRQ', 'LTRS':'LTRS', 'PREV':'PREV'})

    def __init__(self):
        '''
        Constructor
        '''
        self._cmd = Message.CMDS['GOOD']
        self._payload = dict() # dictionary

    def __str__(self):
        return f'Type = {self.getType()}, Payload = {self.getPayload()}'

    def reset(self):
        self._cmd = Message.CMDS['GOOD']
        self._payload = dict()

    def setType(self, mtype: str):
        self._cmd = Message.CMDS[mtype]

    def getType(self) -> str:
        return self._cmd.value

    def setPayload(self, d: dict):
        self._payload = d

    def getPayload(self) -> dict:
        return self._payload

    def marshal(self) -> str:
        size = len(json.dumps(self._payload))
        header = '{}{:04}'.format(self._cmd.value, size)
        return b''.join([header.encode('utf-8'),
json.dumps(self._payload).encode('utf-8')])

    def unmarshal(self, value: bytes):
        self.reset()
        if value:
            self._cmd = Message.CMDS[value[0:4].decode('utf-8')]
            self._payload = json.loads(value[8:].decode('utf-8'))
```

Room:

```python
from models.User import User

class Room(object):
    def __init__(self, name : str, description = '', owner = '', password = '', ):
        self._name = name
        self._description = description
        self._owner = owner
        self._password = password

    def set_name(self, name):
        self._name = name

    def get_name(self):
        return self._name

    def set_description(self, description):
        self._description = description

    def get_description(self):
        return self._description

    def set_owner(self, owner : User):
        self._owner = owner.get_name()

    def get_owner(self):
        return self._owner

    def set_password(self, password):
        self._password = password

    def get_password(self):
        return self._password
```

User:

```python

class User(object):
    def __init__(self, name = '', password = '', s = None):
        self._name = name
        self._password = password
        self._sock = s

    def set_name(self, name):
        self._name = name

    def get_name(self):
        return self._name
    def set_password(self, password):
        self._password = password

    def get_password(self):
        return self._password

    def set_sock(self, s):
        self._sock = s

    def get_sock(self):
        return self._sock
```

Payloads:

```
from models.User import User
from models.Room import Room
from models.Chat import Chat

def LGIN(user : User):
    return dict(username = user.get_name(), password = user.get_password())

def LOUT(user : User):
    return dict(username = user.get_name())

def CUSR(user : User):
    return dict(username = user.get_name(), password=user.get_password())

def ROOM(room : Room):
    return dict(name=room.get_name())

def JOIN(room : Room):
    return dict(room=room.get_name())

def GOOD(msg):
    return dict(message=msg)

def ERRO(msg):
    return dict(message=msg)

def CONN(tcp, group, mcast):
    return dict(TCPport=tcp, GRP=group, MCport=mcast)

def LTRQ():
    return dict()

def LTRS(rooms : list):
    return dict(rooms=rooms)

def DELR(roomname, user):
    return dict(name=roomname, owner=user)

def CHAT(chat : Chat):
    return dict(creator=chat.get_creator(), room=chat.get_room(),
contents=chat.get_contents(), datetime=chat.get_datetime())

def PREV(chats : list):
    return dict(chats=chats)
```