Computers and Systems Engineering Dept.
CSE Program

Course Assessment Specification (CAS)

Programme Title : **Computer and Systems Engineering**
Coursework Title : **Programming Project**
Course Name : **Computer Networks**
Course Code : **CSE351S**
Level : **3**
ASU Credit Rating : **3 Credits**
Weighting : **20%**

Maximum mark available:

- 20 on software projects (1 project)

Lecturer : Prof. Ayman M. Bahaa-Eldin

Contact : If you have any issues with this coursework, you may contact your lecturer.

Contact details are Email: ayman.bahaa@eng.asu.edu.eg

Hand-out Date : As shown in submission matrix
Hand-in Date : As shown in submission matrix
Hand-in Method : Submission through LMS
Feedback Date : Your work will be marked and returned within two weeks.

Introduction

This coursework is itemized into several parts to get the 60 marks associated to it.
You must use the templates provided by the instructor to prepare your work.
All assignments and projects will be handed-in electronically, while quizzes and exams are written.

Learning Outcome to be assessed

2. Explain the use of socket programming in networking applications
4. Recognize the differences between computational and communication overheads
5. Evaluate network performance
10. Work and communicate effectively in a team

Marking Criteria

89% and above:

Your work must be of outstanding quality and fully meet the requirements of the coursework specification and learning outcomes stated. You must show independent thinking and apply this to your work showing originality and consideration of key issues. There must be evidence of wider reading on the subject. In addition, your proposed solution should:

- illustrate a professional ability of drafting construction details,
- express a deep understanding of the in-hand problem definition,
- and applying, masterly, the learned knowledge in the proposed solution.

76% - 89%:

Your work must be of good quality and meet the requirements of the coursework specification and learning outcomes stated. You must demonstrate some originality in your work and show this by applying new learning to the key issues of the coursework. There must be evidence of wider reading on the subject. In addition, your proposed solution should:

- illustrate a good ability of drafting construction details,
- express a very good understanding of the in-hand problem definition,
- and applying most of the learned knowledge, correctly, in the proposed solution.

67% - 76%:

Your work must be comprehensive and meet all the requirements stated by the coursework specification and learning outcomes. You must show a good understanding of the key concepts and be able to apply them to solve the problem set by the coursework. There must be enough depth to your work to provide evidence of wider reading. In addition, your proposed solution should:

- illustrate a moderate ability of drafting construction details,
- express a good understanding of the in-hand problem definition,
- and applying most of the learned knowledge, correctly, in the proposed solution.

60% - 67%:

Your work must be of a standard that meets the requirements stated by the coursework specification and learning outcomes. You must show a reasonable level of understanding of the key concepts and principles and you must have applied this knowledge to the coursework problem. There should be some evidence of wider reading. In addition, your proposed solution should:

- illustrate a fair ability of drafting construction details,
- express a fair understanding of the in-hand problem definition,
- and applying some of the learned knowledge, correctly, in the proposed solution.

Below 60%:

Your work is of poor quality and does not meet the requirements stated by the coursework specification and learning outcomes. There is a lack of understanding of key concepts and knowledge and no evidence of wider reading. In addition, your proposed solution would be:

- Illustrate an inability of drafting construction details,
- Failed to define the parameters, limitations, and offerings of the in-hand problem,
- Failed to correctly apply the learned knowledge for proposing a valid solution.

Academic Misconduct

The University defines Academic Misconduct as 'any case of deliberate, premeditated cheating, collusion, plagiarism or falsification of information, in an attempt to deceive and gain an unfair advantage in assessment'. This includes attempting to gain marks as part of a team without contributing. The department takes Academic Misconduct very seriously and any suspected cases will be investigated through the University's standard policy. If you are found guilty, you may be expelled from the University with no award.

It is your responsibility to ensure that you understand what constitutes Academic Misconduct and to ensure that you do not break the rules. If you are unclear about what is required, please ask.

Detail of the task

Extending the P2P Chat program

General Guidelines

- This project can be conducted in groups up to 4 students (the group may be 1 (individual), 2, 3 or 4 maximum)
- You need to deliver your work as a single PDF file with a prober software engineering technical documentation including requirements, design, test cases and implementation.
- The implementation section must have your code, screen shots of the working program and test logs for the set of test cases you have proposed.

Introduction

In this project, you will learn how P2P applications work and some of their basic functionalities. Your task is to extend the provided P2P chat program (found on the LMS and at the end of this document), adding a new feature which is ***Chat Rooms***

A chat room is a group chatting feature where a user can create a room with a **room_id**, add users to it.

Detailed Requirements

Registry Server

The server is to be extended keeps track of the chat rooms available, and the clients in each room. Each P2P client must regularly contact the room server to maintain the connection established initially (client must ping server every 20s to stay alive, or client will be disconnected).

Client

Each client is extended to acts as both a server (to other P2P clients for distributing messages along the network), and a client (receiver of messages from other P2P clients). Once a client is registered on the room at the server, it starts to look for a peer in the chat room it is in currently. When a message is sent by a user, it must be delivered to all the online participants in the same room.

The message should be marked with a text before displaying the message on the peer screen as "Message from [USER] at [room_id]: text of the message".

The user is prompted if it would like to send a message for a peer or to one of its rooms.

Hint

- Uses threading to manage multiple peers on each P2P client.
- Uses a concept called flooding to distribute messages along the network.

Code

Below you will find the code for the client (peer.py), server (registry.py) and database (db.py). You need to have MongoDb installed and running on the same host as the registry server.

What to Hand in

You will hand in a single PDF document containing your design of the system, complete code and screenshots at the peers confirming that you successfully implemented the required extension.

peer.py

```
'''
   ##  Implementation of peer
   ##  Each peer has a client and a server side that runs on different threads
   ##  150114822 - Eren Ulaş
'''

from socket import *
import threading
import time
import select
import logging

# Server side of peer
class PeerServer(threading.Thread):


   # Peer server initialization
   def __init__(self, username, peerServerPort):
      threading.Thread.__init__(self)
      # keeps the username of the peer
      self.username = username
      # tcp socket for peer server
      self.tcpServerSocket = socket(AF_INET, SOCK_STREAM)
      # port number of the peer server
      self.peerServerPort = peerServerPort
      # if 1, then user is already chatting with someone
      # if 0, then user is not chatting with anyone
      self.isChatRequested = 0
      # keeps the socket for the peer that is connected to this peer
      self.connectedPeerSocket = None
      # keeps the ip of the peer that is connected to this peer's server
      self.connectedPeerIP = None
      # keeps the port number of the peer that is connected to this peer's server
      self.connectedPeerPort = None
      # online status of the peer
      self.isOnline = True
      # keeps the username of the peer that this peer is chatting with
      self.chattingClientName = None


   # main method of the peer server thread
   def run(self):

      print("Peer server started...")
```

```
# gets the ip address of this peer
# first checks to get it for windows devices
# if the device that runs this application is not windows
# it checks to get it for macos devices
hostname=gethostname()
try:
   self.peerServerHostname=gethostbyname(hostname)
except gaierror:
   import netifaces as ni
   self.peerServerHostname = ni.ifaddresses('en0')[ni.AF_INET][0]['addr']

# ip address of this peer
#self.peerServerHostname = 'localhost'
# socket initializations for the server of the peer
self.tcpServerSocket.bind((self.peerServerHostname, self.peerServerPort))
self.tcpServerSocket.listen(4)
# inputs sockets that should be listened
inputs = [self.tcpServerSocket]
# server listens as long as there is a socket to listen in the inputs list and the user is online
while inputs and self.isOnline:
   # monitors for the incoming connections
   try:
      readable, writable, exceptional = select.select(inputs, [], [])
      # If a server waits to be connected enters here
      for s in readable:
         # if the socket that is receiving the connection is
         # the tcp socket of the peer's server, enters here
         if s is self.tcpServerSocket:
            # accepts the connection, and adds its connection socket to the inputs list
            # so that we can monitor that socket as well
            connected, addr = s.accept()
            connected.setblocking(0)
            inputs.append(connected)
            # if the user is not chatting, then the ip and the socket of
            # this peer is assigned to server variables
            if self.isChatRequested == 0:
               print(self.username + " is connected from " + str(addr))
               self.connectedPeerSocket = connected
               self.connectedPeerIP = addr[0]
         # if the socket that receives the data is the one that
         # is used to communicate with a connected peer, then enters here
         else:
            # message is received from connected peer
            messageReceived = s.recv(1024).decode()
```

```
                # logs the received message
                logging.info("Received from " + str(self.connectedPeerIP) + " -> " + str(messageReceived))
                # if message is a request message it means that this is the receiver side peer server
                # so evaluate the chat request
                if len(messageReceived) > 11 and messageReceived[:12] == "CHAT-REQUEST":
                    # text for proper input choices is printed however OK or REJECT is taken as input in main process of
the peer
                    # if the socket that we received the data belongs to the peer that we are chatting with,
                    # enters here
                    if s is self.connectedPeerSocket:
                        # parses the message
                        messageReceived = messageReceived.split()
                        # gets the port of the peer that sends the chat request message
                        self.connectedPeerPort = int(messageReceived[1])
                        # gets the username of the peer sends the chat request message
                        self.chattingClientName = messageReceived[2]
                        # prints prompt for the incoming chat request
                        print("Incoming chat request from " + self.chattingClientName + " >> ")
                        print("Enter OK to accept or REJECT to reject:  ")
                        # makes isChatRequested = 1 which means that peer is chatting with someone
                        self.isChatRequested = 1
                    # if the socket that we received the data does not belong to the peer that we are chatting with
                    # and if the user is already chatting with someone else(isChatRequested = 1), then enters here
                    elif s is not self.connectedPeerSocket and self.isChatRequested == 1:
                        # sends a busy message to the peer that sends a chat request when this peer is
                        # already chatting with someone else
                        message = "BUSY"
                        s.send(message.encode())
                        # remove the peer from the inputs list so that it will not monitor this socket
                        inputs.remove(s)
                # if an OK message is received then ischatrequested is made 1 and then next messages will be shown to
the peer of this server
                elif messageReceived == "OK":
                    self.isChatRequested = 1
                # if an REJECT message is received then ischatrequested is made 0 so that it can receive any other chat
requests
                elif messageReceived == "REJECT":
                    self.isChatRequested = 0
                    inputs.remove(s)
                # if a message is received, and if this is not a quit message ':q' and
                # if it is not an empty message, show this message to the user
                elif messageReceived[:2] != ":q" and len(messageReceived)!= 0:
                    print(self.chattingClientName + ": " + messageReceived)
                # if the message received is a quit message ':q',
                # makes ischatrequested 1 to receive new incoming request messages
```

```
                # removes the socket of the connected peer from the inputs list
                elif messageReceived[:2] == ":q":
                    self.isChatRequested = 0
                    inputs.clear()
                    inputs.append(self.tcpServerSocket)
                    # connected peer ended the chat
                    if len(messageReceived) == 2:
                        print("User you're chatting with ended the chat")
                        print("Press enter to quit the chat: ")
                # if the message is an empty one, then it means that the
                # connected user suddenly ended the chat(an error occurred)
                elif len(messageReceived) == 0:
                    self.isChatRequested = 0
                    inputs.clear()
                    inputs.append(self.tcpServerSocket)
                    print("User you're chatting with suddenly ended the chat")
                    print("Press enter to quit the chat: ")
            # handles the exceptions, and logs them
            except OSError as oErr:
                logging.error("OSError: {0}".format(oErr))
            except ValueError as vErr:
                logging.error("ValueError: {0}".format(vErr))


# Client side of peer
class PeerClient(threading.Thread):
    # variable initializations for the client side of the peer
    def __init__(self, ipToConnect, portToConnect, username, peerServer, responseReceived):
        threading.Thread.__init__(self)
        # keeps the ip address of the peer that this will connect
        self.ipToConnect = ipToConnect
        # keeps the username of the peer
        self.username = username
        # keeps the port number that this client should connect
        self.portToConnect = portToConnect
        # client side tcp socket initialization
        self.tcpClientSocket = socket(AF_INET, SOCK_STREAM)
        # keeps the server of this client
        self.peerServer = peerServer
        # keeps the phrase that is used when creating the client
        # if the client is created with a phrase, it means this one received the request
        # this phrase should be none if this is the client of the requester peer
        self.responseReceived = responseReceived
        # keeps if this client is ending the chat or not
        self.isEndingChat = False
```

```
    # main method of the peer client thread
    def run(self):
        print("Peer client started...")
        # connects to the server of other peer
        self.tcpClientSocket.connect((self.ipToConnect, self.portToConnect))
        # if the server of this peer is not connected by someone else and if this is the requester side peer client then enters
here
        if self.peerServer.isChatRequested == 0 and self.responseReceived is None:
            # composes a request message and this is sent to server and then this waits a response message from the server
this client connects
            requestMessage = "CHAT-REQUEST " + str(self.peerServer.peerServerPort)+ " " + self.username
            # logs the chat request sent to other peer
            logging.info("Send to " + self.ipToConnect + ":" + str(self.portToConnect) + " -> " + requestMessage)
            # sends the chat request
            self.tcpClientSocket.send(requestMessage.encode())
            print("Request message " + requestMessage + " is sent...")
            # received a response from the peer which the request message is sent to
            self.responseReceived = self.tcpClientSocket.recv(1024).decode()
            # logs the received message
            logging.info("Received from " + self.ipToConnect + ":" + str(self.portToConnect) + " -> " +
self.responseReceived)
            print("Response is " + self.responseReceived)
            # parses the response for the chat request
            self.responseReceived = self.responseReceived.split()
            # if response is ok then incoming messages will be evaluated as client messages and will be sent to the connected
server
            if self.responseReceived[0] == "OK":
                # changes the status of this client's server to chatting
                self.peerServer.isChatRequested = 1
                # sets the server variable with the username of the peer that this one is chatting
                self.peerServer.chattingClientName = self.responseReceived[1]
                # as long as the server status is chatting, this client can send messages
                while self.peerServer.isChatRequested == 1:
                    # message input prompt
                    messageSent = input(self.username + ": ")
                    # sends the message to the connected peer, and logs it
                    self.tcpClientSocket.send(messageSent.encode())
                    logging.info("Send to " + self.ipToConnect + ":" + str(self.portToConnect) + " -> " + messageSent)
                    # if the quit message is sent, then the server status is changed to not chatting
                    # and this is the side that is ending the chat
                    if messageSent == ":q":
                        self.peerServer.isChatRequested = 0
                        self.isEndingChat = True
```

```
                break
            # if peer is not chatting, checks if this is not the ending side
            if self.peerServer.isChatRequested == 0:
                if not self.isEndingChat:
                    # tries to send a quit message to the connected peer
                    # logs the message and handles the exception
                    try:
                        self.tcpClientSocket.send(":q ending-side".encode())
                        logging.info("Send to " + self.ipToConnect + ":" + str(self.portToConnect) + " -> :q")
                    except BrokenPipeError as bpErr:
                        logging.error("BrokenPipeError: {0}".format(bpErr))
                # closes the socket
                self.responseReceived = None
                self.tcpClientSocket.close()
        # if the request is rejected, then changes the server status, sends a reject message to the connected peer's server
        # logs the message and then the socket is closed
        elif self.responseReceived[0] == "REJECT":
            self.peerServer.isChatRequested = 0
            print("client of requester is closing...")
            self.tcpClientSocket.send("REJECT".encode())
            logging.info("Send to " + self.ipToConnect + ":" + str(self.portToConnect) + " -> REJECT")
            self.tcpClientSocket.close()
        # if a busy response is received, closes the socket
        elif self.responseReceived[0] == "BUSY":
            print("Receiver peer is busy")
            self.tcpClientSocket.close()
    # if the client is created with OK message it means that this is the client of receiver side peer
    # so it sends an OK message to the requesting side peer server that it connects and then waits for the user inputs.
    elif self.responseReceived == "OK":
        # server status is changed
        self.peerServer.isChatRequested = 1
        # ok response is sent to the requester side
        okMessage = "OK"
        self.tcpClientSocket.send(okMessage.encode())
        logging.info("Send to " + self.ipToConnect + ":" + str(self.portToConnect) + " -> " + okMessage)
        print("Client with OK message is created... and sending messages")
        # client can send messsages as long as the server status is chatting
        while self.peerServer.isChatRequested == 1:
            # input prompt for user to enter message
            messageSent = input(self.username + ": ")
            self.tcpClientSocket.send(messageSent.encode())
            logging.info("Send to " + self.ipToConnect + ":" + str(self.portToConnect) + " -> " + messageSent)
            # if a quit message is sent, server status is changed
            if messageSent == ":q":
                self.peerServer.isChatRequested = 0
```

```
                self.isEndingChat = True
                break
        # if server is not chatting, and if this is not the ending side
        # sends a quitting message to the server of the other peer
        # then closes the socket
        if self.peerServer.isChatRequested == 0:
            if not self.isEndingChat:
                self.tcpClientSocket.send(":q ending-side".encode())
                logging.info("Send to " + self.ipToConnect + ":" + str(self.portToConnect) + " -> :q")
            self.responseReceived = None
            self.tcpClientSocket.close()


# main process of the peer
class peerMain:

    # peer initializations
    def __init__(self):
        # ip address of the registry
        self.registryName = input("Enter IP address of registry: ")
        #self.registryName = 'localhost'
        # port number of the registry
        self.registryPort = 15600
        # tcp socket connection to registry
        self.tcpClientSocket = socket(AF_INET, SOCK_STREAM)
        self.tcpClientSocket.connect((self.registryName,self.registryPort))
        # initializes udp socket which is used to send hello messages
        self.udpClientSocket = socket(AF_INET, SOCK_DGRAM)
        # udp port of the registry
        self.registryUDPPort = 15500
        # login info of the peer
        self.loginCredentials = (None, None)
        # online status of the peer
        self.isOnline = False
        # server port number of this peer
        self.peerServerPort = None
        # server of this peer
        self.peerServer = None
        # client of this peer
        self.peerClient = None
        # timer initialization
        self.timer = None

        choice = "0"
        # log file initialization
```

```
logging.basicConfig(filename="peer.log", level=logging.INFO)
# as long as the user is not logged out, asks to select an option in the menu
while choice != "3":
    # menu selection prompt
    choice = input("Choose: \nCreate account: 1\nLogin: 2\nLogout: 3\nSearch: 4\nStart a chat: 5\n")
    # if choice is 1, creates an account with the username
    # and password entered by the user
    if choice is "1":
        username = input("username: ")
        password = input("password: ")

        self.createAccount(username, password)
    # if choice is 2 and user is not logged in, asks for the username
    # and the password to login
    elif choice is "2" and not self.isOnline:
        username = input("username: ")
        password = input("password: ")
        # asks for the port number for server's tcp socket
        peerServerPort = int(input("Enter a port number for peer server: "))

        status = self.login(username, password, peerServerPort)
        # is user logs in successfully, peer variables are set
        if status is 1:
            self.isOnline = True
            self.loginCredentials = (username, password)
            self.peerServerPort = peerServerPort
            # creates the server thread for this peer, and runs it
            self.peerServer = PeerServer(self.loginCredentials[0], self.peerServerPort)
            self.peerServer.start()
            # hello message is sent to registry
            self.sendHelloMessage()
    # if choice is 3 and user is logged in, then user is logged out
    # and peer variables are set, and server and client sockets are closed
    elif choice is "3" and self.isOnline:
        self.logout(1)
        self.isOnline = False
        self.loginCredentials = (None, None)
        self.peerServer.isOnline = False
        self.peerServer.tcpServerSocket.close()
        if self.peerClient is not None:
            self.peerClient.tcpClientSocket.close()
        print("Logged out successfully")
    # is peer is not logged in and exits the program
    elif choice is "3":
        self.logout(2)
```

```
        # if choice is 4 and user is online, then user is asked
        # for a username that is wanted to be searched
        elif choice is "4" and self.isOnline:
            username = input("Username to be searched: ")
            searchStatus = self.searchUser(username)
            # if user is found its ip address is shown to user
            if searchStatus is not None and searchStatus != 0:
                print("IP address of " + username + " is " + searchStatus)
        # if choice is 5 and user is online, then user is asked
        # to enter the username of the user that is wanted to be chatted
        elif choice is "5" and self.isOnline:
            username = input("Enter the username of user to start chat: ")
            searchStatus = self.searchUser(username)
            # if searched user is found, then its ip address and port number is retrieved
            # and a client thread is created
            # main process waits for the client thread to finish its chat
            if searchStatus is not None and searchStatus is not 0:
                searchStatus = searchStatus.split(":")
                self.peerClient = PeerClient(searchStatus[0], int(searchStatus[1]) , self.loginCredentials[0], self.peerServer,
None)
                self.peerClient.start()
                self.peerClient.join()
        # if this is the receiver side then it will get the prompt to accept an incoming request during the main loop
        # that's why response is evaluated in main process not the server thread even though the prompt is printed by
server
        # if the response is ok then a client is created for this peer with the OK message and that's why it will directly
        # sent an OK message to the requesting side peer server and waits for the user input
        # main process waits for the client thread to finish its chat
        elif choice == "OK" and self.isOnline:
            okMessage = "OK " + self.loginCredentials[0]
            logging.info("Send to " + self.peerServer.connectedPeerIP + " -> " + okMessage)
            self.peerServer.connectedPeerSocket.send(okMessage.encode())
            self.peerClient = PeerClient(self.peerServer.connectedPeerIP, self.peerServer.connectedPeerPort ,
self.loginCredentials[0], self.peerServer, "OK")
            self.peerClient.start()
            self.peerClient.join()
        # if user rejects the chat request then reject message is sent to the requester side
        elif choice == "REJECT" and self.isOnline:
            self.peerServer.connectedPeerSocket.send("REJECT".encode())
            self.peerServer.isChatRequested = 0
            logging.info("Send to " + self.peerServer.connectedPeerIP + " -> REJECT")
        # if choice is cancel timer for hello message is cancelled
        elif choice == "CANCEL":
            self.timer.cancel()
            break
```

```
    # if main process is not ended with cancel selection
    # socket of the client is closed
    if choice != "CANCEL":
        self.tcpClientSocket.close()

# account creation function
def createAccount(self, username, password):
    # join message to create an account is composed and sent to registry
    # if response is success then informs the user for account creation
    # if response is exist then informs the user for account existence
    message = "JOIN " + username + " " + password
    logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " + message)
    self.tcpClientSocket.send(message.encode())
    response = self.tcpClientSocket.recv(1024).decode()
    logging.info("Received from " + self.registryName + " -> " + response)
    if response == "join-success":
        print("Account created...")
    elif response == "join-exist":
        print("choose another username or login...")

# login function
def login(self, username, password, peerServerPort):
    # a login message is composed and sent to registry
    # an integer is returned according to each response
    message = "LOGIN " + username + " " + password + " " + str(peerServerPort)
    logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " + message)
    self.tcpClientSocket.send(message.encode())
    response = self.tcpClientSocket.recv(1024).decode()
    logging.info("Received from " + self.registryName + " -> " + response)
    if response == "login-success":
        print("Logged in successfully...")
        return 1
    elif response == "login-account-not-exist":
        print("Account does not exist...")
        return 0
    elif response == "login-online":
        print("Account is already online...")
        return 2
    elif response == "login-wrong-password":
        print("Wrong password...")
        return 3

# logout function
def logout(self, option):
    # a logout message is composed and sent to registry
```

```
        # timer is stopped
        if option == 1:
            message = "LOGOUT " + self.loginCredentials[0]
            self.timer.cancel()
        else:
            message = "LOGOUT"
        logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " + message)
        self.tcpClientSocket.send(message.encode())


    # function for searching an online user
    def searchUser(self, username):
        # a search message is composed and sent to registry
        # custom value is returned according to each response
        # to this search message
        message = "SEARCH " + username
        logging.info("Send to " + self.registryName + ":" + str(self.registryPort) + " -> " + message)
        self.tcpClientSocket.send(message.encode())
        response = self.tcpClientSocket.recv(1024).decode().split()
        logging.info("Received from " + self.registryName + " -> " + " ".join(response))
        if response[0] == "search-success":
            print(username + " is found successfully...")
            return response[1]
        elif response[0] == "search-user-not-online":
            print(username + " is not online...")
            return 0
        elif response[0] == "search-user-not-found":
            print(username + " is not found")
            return None

    # function for sending hello message
    # a timer thread is used to send hello messages to udp socket of registry
    def sendHelloMessage(self):
        message = "HELLO " + self.loginCredentials[0]
        logging.info("Send to " + self.registryName + ":" + str(self.registryUDPPort) + " -> " + message)
        self.udpClientSocket.sendto(message.encode(), (self.registryName, self.registryUDPPort))
        self.timer = threading.Timer(1, self.sendHelloMessage)
        self.timer.start()

# peer is started
main = peerMain()
```

Registry.py

```
'''
   ##  Implementation of registry
   ##  150114822 - Eren Ulaş
'''

from socket import *
import threading
import select
import logging
import db

# This class is used to process the peer messages sent to registry
# for each peer connected to registry, a new client thread is created
class ClientThread(threading.Thread):
   # initializations for client thread
   def __init__(self, ip, port, tcpClientSocket):
      threading.Thread.__init__(self)
      # ip of the connected peer
      self.ip = ip
      # port number of the connected peer
      self.port = port
      # socket of the peer
      self.tcpClientSocket = tcpClientSocket
      # username, online status and udp server initializations
      self.username = None
      self.isOnline = True
      self.udpServer = None
      print("New thread started for " + ip + ":" + str(port))

   # main of the thread
   def run(self):
      # locks for thread which will be used for thread synchronization
      self.lock = threading.Lock()
      print("Connection from: " + self.ip + ":" + str(port))
      print("IP Connected: " + self.ip)

      while True:
         try:
            # waits for incoming messages from peers
            message = self.tcpClientSocket.recv(1024).decode().split()
            logging.info("Received from " + self.ip + ":" + str(self.port) + " -> " + " ".join(message))
            #   JOIN    #
            if message[0] == "JOIN":
```

```
            # join-exist is sent to peer,
            # if an account with this username already exists
            if db.is_account_exist(message[1]):
                response = "join-exist"
                print("From-> " + self.ip + ":" + str(self.port) + " " + response)
                logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)
                self.tcpClientSocket.send(response.encode())
            # join-success is sent to peer,
            # if an account with this username is not exist, and the account is created
            else:
                db.register(message[1], message[2])
                response = "join-success"
                logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)
                self.tcpClientSocket.send(response.encode())
        #   LOGIN    #
        elif message[0] == "LOGIN":
            # login-account-not-exist is sent to peer,
            # if an account with the username does not exist
            if not db.is_account_exist(message[1]):
                response = "login-account-not-exist"
                logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)
                self.tcpClientSocket.send(response.encode())
            # login-online is sent to peer,
            # if an account with the username already online
            elif db.is_account_online(message[1]):
                response = "login-online"
                logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)
                self.tcpClientSocket.send(response.encode())
            # login-success is sent to peer,
            # if an account with the username exists and not online
            else:
                # retrieves the account's password, and checks if the one entered by the user is correct
                retrievedPass = db.get_password(message[1])
                # if password is correct, then peer's thread is added to threads list
                # peer is added to db with its username, port number, and ip address
                if retrievedPass == message[2]:
                    self.username = message[1]
                    self.lock.acquire()
                    try:
                        tcpThreads[self.username] = self
                    finally:
                        self.lock.release()

                    db.user_login(message[1], self.ip, message[3])
                    # login-success is sent to peer,
```

```
            # and a udp server thread is created for this peer, and thread is started
            # timer thread of the udp server is started
            response = "login-success"
            logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)
            self.tcpClientSocket.send(response.encode())
            self.udpServer = UDPServer(self.username, self.tcpClientSocket)
            self.udpServer.start()
            self.udpServer.timer.start()
        # if password not matches and then login-wrong-password response is sent
        else:
            response = "login-wrong-password"
            logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)
            self.tcpClientSocket.send(response.encode())
#   LOGOUT  #
elif message[0] == "LOGOUT":
    # if user is online,
    # removes the user from onlinePeers list
    # and removes the thread for this user from tcpThreads
    # socket is closed and timer thread of the udp for this
    # user is cancelled
    if len(message) > 1 and message[1] is not None and db.is_account_online(message[1]):
        db.user_logout(message[1])
        self.lock.acquire()
        try:
            if message[1] in tcpThreads:
                del tcpThreads[message[1]]
        finally:
            self.lock.release()
        print(self.ip + ":" + str(self.port) + " is logged out")
        self.tcpClientSocket.close()
        self.udpServer.timer.cancel()
        break
    else:
        self.tcpClientSocket.close()
        break
#   SEARCH  #
elif message[0] == "SEARCH":
    # checks if an account with the username exists
    if db.is_account_exist(message[1]):
        # checks if the account is online
        # and sends the related response to peer
        if db.is_account_online(message[1]):
            peer_info = db.get_peer_ip_port(message[1])
            response = "search-success " + peer_info[0] + ":" + peer_info[1]
            logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)
```

```
                    self.tcpClientSocket.send(response.encode())
                else:
                    response = "search-user-not-online"
                    logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)
                    self.tcpClientSocket.send(response.encode())
            # enters if username does not exist
            else:
                response = "search-user-not-found"
                logging.info("Send to " + self.ip + ":" + str(self.port) + " -> " + response)
                self.tcpClientSocket.send(response.encode())
        except OSError as oErr:
            logging.error("OSError: {0}".format(oErr))


    # function for resettin the timeout for the udp timer thread
    def resetTimeout(self):
        self.udpServer.resetTimer()


# implementation of the udp server thread for clients
class UDPServer(threading.Thread):


    # udp server thread initializations
    def __init__(self, username, clientSocket):
        threading.Thread.__init__(self)
        self.username = username
        # timer thread for the udp server is initialized
        self.timer = threading.Timer(3, self.waitHelloMessage)
        self.tcpClientSocket = clientSocket


    # if hello message is not received before timeout
    # then peer is disconnected
    def waitHelloMessage(self):
        if self.username is not None:
            db.user_logout(self.username)
            if self.username in tcpThreads:
                del tcpThreads[self.username]
        self.tcpClientSocket.close()
        print("Removed " + self.username + " from online peers")


    # resets the timer for udp server
    def resetTimer(self):
```

```
        self.timer.cancel()
        self.timer = threading.Timer(3, self.waitHelloMessage)
        self.timer.start()


# tcp and udp server port initializations
print("Registy started...")
port = 15600
portUDP = 15500

# db initialization
db = db.DB()

# gets the ip address of this peer
# first checks to get it for windows devices
# if the device that runs this application is not windows
# it checks to get it for macos devices
hostname=gethostname()
try:
    host=gethostbyname(hostname)
except gaierror:
    import netifaces as ni
    host = ni.ifaddresses('en0')[ni.AF_INET][0]['addr']


print("Registry IP address: " + host)
print("Registry port number: " + str(port))

# onlinePeers list for online account
onlinePeers = {}
# accounts list for accounts
accounts = {}
# tcpThreads list for online client's thread
tcpThreads = {}

#tcp and udp socket initializations
tcpSocket = socket(AF_INET, SOCK_STREAM)
udpSocket = socket(AF_INET, SOCK_DGRAM)
tcpSocket.bind((host,port))
udpSocket.bind((host,portUDP))
tcpSocket.listen(5)

# input sockets that are listened
inputs = [tcpSocket, udpSocket]
```

```
# log file initialization
logging.basicConfig(filename="registry.log", level=logging.INFO)

# as long as at least a socket exists to listen registry runs
while inputs:

    print("Listening for incoming connections...")
    # monitors for the incoming connections
    readable, writable, exceptional = select.select(inputs, [], [])
    for s in readable:
        # if the message received comes to the tcp socket
        # the connection is accepted and a thread is created for it, and that thread is started
        if s is tcpSocket:
            tcpClientSocket, addr = tcpSocket.accept()
            newThread = ClientThread(addr[0], addr[1], tcpClientSocket)
            newThread.start()
        # if the message received comes to the udp socket
        elif s is udpSocket:
            # received the incoming udp message and parses it
            message, clientAddress = s.recvfrom(1024)
            message = message.decode().split()
            # checks if it is a hello message
            if message[0] == "HELLO":
                # checks if the account that this hello message
                # is sent from is online
                if message[1] in tcpThreads:
                    # resets the timeout for that peer since the hello message is received
                    tcpThreads[message[1]].resetTimeout()
                    print("Hello is received from " + message[1])
                    logging.info("Received from " + clientAddress[0] + ":" + str(clientAddress[1]) + " -> " + " ".join(message))

# registry tcp socket is closed
tcpSocket.close()
```

db.py

```
from pymongo import MongoClient

# Includes database operations
class DB:


    # db initializations
    def __init__(self):
        self.client = MongoClient('mongodb://localhost:27017/')
        self.db = self.client['p2p-chat']


    # checks if an account with the username exists
    def is_account_exist(self, username):
        if self.db.accounts.find({'username': username}).count() > 0:
            return True
        else:
            return False


    # registers a user
    def register(self, username, password):
        account = {
            "username": username,
            "password": password
        }
        self.db.accounts.insert(account)


    # retrieves the password for a given username
    def get_password(self, username):
        return self.db.accounts.find_one({"username": username})["password"]


    # checks if an account with the username online
    def is_account_online(self, username):
        if self.db.online_peers.find({"username": username}).count() > 0:
            return True
        else:
            return False


    # logs in the user
    def user_login(self, username, ip, port):
```

```
    online_peer = {
        "username": username,
        "ip": ip,
        "port": port
    }
    self.db.online_peers.insert(online_peer)


# logs out the user
def user_logout(self, username):
    self.db.online_peers.remove({"username": username})


# retrieves the ip address and the port number of the username
def get_peer_ip_port(self, username):
    res = self.db.online_peers.find_one({"username": username})
    return (res["ip"], res["port"])
```