Western Science

ASSIGNMENT 02

TCP & UDP CHATROOM

Due Date:
October 25, 2024 23:55:59

21d 9h 6m 10s left

ACADEMIC DISHONESTY

Assignments will be run through a similarity checking software to check for code that looks very similar to that of other students. Sharing or copying code in any way is considered plagiarism (Academic dishonesty) and may result in a mark of 0 on the assignment and/or reported to the Dean's Office. Plagiarism is a serious offence. Work is to be done **individually.**

If you want to store a PDF version of this assignment, press Ctrl+p *on Windows or* Command+p *on Mac, the print window will appear. Then, select* Save As PDF *from the* **Destination** *dropdown. Then click* Save

This file was last modified on 2024-09-19 8:06 pm ET

UPDATES and CHANGES

The following are the updates/changes made to this assignment AFTER it was posted:

No changes have been made!

Anouncements and Important Notes

The following are the announcements and important notes for this assignment AFTER it was posted:

No announcements have been made!

1 INTRODUCTION

Imagine you are back in the early days of networking, in the year 1980. Meet Alexander Trevor, a pioneering engineer who is working on a cutting-edge project, implementing the first commercial multi-user chat program over the internet. Your task is to assist Trevor in bringing this vision to life using the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) that you have learned about in class. Fortunately, Trevor has already outlined the basic design for both the server and client sides for each protocol. This foundation will make it easier for you to implement the chatroom functionalities.

Your goal is to create two chatroom programs: one using the TCP and the other using the UDP. The chatroom should allow multiple clients to connect to a server and communicate with each other in real-time.

Objectives:

- Understand and apply TCP and UDP protocols in socket programming.
- Develop multi-threaded server and client applications.
- Handle multiple clients in a chatroom environment.
- Implement message broadcasting and client management.

2 Important Notes

1. TCP Chatroom has two classes: `ServerTCP` and `ClientTCP`, and UDP Chatroom has two classes: `ServerUDP` and `ClientUDP`. You are required to implement all four classes in one Python file named **chatroom.py**, and submit it on Gradescope.
2. You are required to adhere to the class structure, method descriptions, and naming conventions for both methods and instance variables as outlined in the sections below.
3. Feel free to implement your own logic of the methods as long as they fulfill the requirements specified in the assignment.
4. You can add additional methods or instance variables to the class if needed for your implementation.
5. when developing the methods, it is efficient to use try-except blocks to handle exceptions or errors that may occur in your code.
6. when setting or connecting a socket, the IP address should be set to to your local machine's IP address. you can use say `addr = socket.gethostbyname(socket.gethostname())` to get your local machine's IP address.
7. some classes may have two or more methods that need to run simultaneously. To handle this, you can use the `threading` module in Python to create threads for each method.
8. You should be aware that the socket methods `accept()` and `recv()` are blocking methods, meaning they will wait until they receive a connection or message and block the program's execution until then. To handle this, you can use the `select` module in Python to monitor sockets with timeout. This will allow you to check if there is any data to be received before calling a blocking method.
9. (Optional) When displaying the messages in the terminal and displaying the input prompt, you can use the `sys.stdout.write()` method to display messages without a newline character and `sys.stdout.flush()` to flush the output buffer.

3 TCP Chatroom Implementation

ServerTCP Class

This class implements a chatroom server using the TCP protocol. It coordinates several methods to manage client connections and message broadcasting. When the server is started with the `run()` method, it continuously listens for incoming connections. Upon a new connection, the `accept_client()` method checks if the client's name is unique, adds the client to the clients dictionary, and broadcasts their entrance to others using the `broadcast()` method. The server then uses the `handle_client()` method simultaneously to listen for messages from the

connected client, broadcasting them to other clients or closing the connection if the client exits. If the server needs to shut down, the `shutdown()` method is invoked to send a shutdown message to all clients, close their sockets, and stop the server. The following is a detailed description of its methods:

__init__(self, server_port)

Instance variables:

- **`server_port`**: The port number on which the server will listen for incoming messages.
- **`server_socket`**: The socket object used by the server to send and receive messages.
- **`clients`**: A dictionary to store the client addresses and names.
- **`run_event`**: A threading event to control the server's running state.
- **`handle_event`**: A threading event to control the server's message handling state.

This method Initializes the server socket, binds it to the local address and the given server port `server_port`, and sets the socket to listen for incoming connections. Initializes the `clients` dictionary to store the client addresses and names, and the `run_event` and `handle_event` threading events to control the server's running and message handling states.

accept_client(self)

This method handles a new client connection if available. It receives the client's message, which contains their name. If the name is already in the `clients` dictionary, the method sends a **'Name already taken'** message to the client and returns `False`. Otherwise, it performs the following actions:

- Sends a **'Welcome'** message to the client.
- Adds the client socket as a key and the name as a value in the `clients` dictionary.
- Broadcasts a message indicating that the user has joined.
- Returns `True` if the client's name is unique.

close_client(self, client_socket)

This method closes the connection for a given client socket by removing the user from the `clients` dictionary and closing the socket. It returns `True` if the client was successfully removed, otherwise `False`.

broadcast(self, client_socket_sent, message)

This method determines the appropriate broadcast message based on the provided `message` parameter. If `message` is **'join'**, the broadcast message will be **'User {x} joined'**. If it's **'exit'**, the broadcast message will be **'User {x} left'**. For any other message, the broadcast message format will be **'{x}: {message}'**, where `x` is the client's name, and `message` is the content sent by the client. This broadcast message is then sent to all connected clients except for the one who sent it, represented by `client_socket_sent`.

shutdown(self)

This method shuts down the server by sending a **'server-shutdown'** message to all connected clients. It then proceeds to close all client sockets, sets the `run_event` and `handle_event` to stop the server's operation, and finally closes the server socket.

get_clients_number(self)

This method returns the number of currently connected clients by simply returning the length of the `clients` dictionary.

handle_client(self, client_socket)

This method continuously listens for messages the given client socket `client_socket` until the `handle_event` is set. The received message is decoded and broadcasted to all other clients using the `broadcast()` method. The method also handles breaking the loop and closing the client's connection if the received message is **'exit'**.

run(self)

This method starts the server and begins listening for incoming connections by invoking the `accept_client` method. Upon accepting a new connection, the server handles it while continuing to listen for other incoming connections simultaneously. The server will keep running in this way until it receives a `KeyboardInterrupt` or the `run_event` is set, at which point the `shutdown` method is invoked to stop the server.

ClientTCP Class

This class implements a chatroom client using the TCP protocol. It provides methods to connect to the server, send messages, and receive messages from the server. The client can send messages to the server using the `send()` method and receive messages from the server using the `receive()` method. The client uses the `run()` method to handle the main loop for message sending and receiving. The following is a detailed description of its methods:

__init__(self, client_name, server_port)

Instance variables:

- **`server_addr`**: The server's address to which the client will connect.
- **`client_socket`**: The socket object used by the client to send and receive messages.
- **`server_port`**: The port number on which the server is listening for incoming connections.
- **`client_name`**: The name of the client connecting to the server.
- **`exit_run`**: A threading event to control the main loop's running state.
- **`exit_receive`**: A threading event to control the message receiving loop's state.

This method initializes a TCP socket as the client's socket `client_socket`, sets the server address `server_addr` to the local machine's IP, intializes the client's name, and server port to the given `client_name` and `server_port` parameters respectively. It also sets up threading events to control the client's operation and message receiving loop.

connect_server(self)

This method attempts to connect the client to the server using the its socket `client_socket` and the server's address and port. then, it sends the client's name to the server and waits for a response. If the response contains **'Welcome'**, it indicates that the client has successfully joined the chatroom. The method returns `True` if the connection is successful; otherwise, it returns `False`.

send(self, text)

This method sends the given `text` to the server as a message. The message is encoded and then transmitted through the client's socket `client_socket`.

receive(self)

This method continuously listens for messages from the server until the `exit_receive` event is set. In the loop, the client receives a message from the server and decodes it. If the message is **'server-shutdown'**, which indicates that the server is shutting down, the method sets the `exit_run` and `exit_receive` events to stop the client's operation and exits the loop. Otherwise, the message is displayed to the user.

run(self)

This method is the main loop for the client. It first connects to the server using the `connect_server()` method. If successful, connection is established, and the method start to handle receiving messages from the server using the `receive()` method simultaneously with handling user input in the main loop. In this loop, the user can input the text to be sent as a message to the server using the `send()` method. this loop continues until the `exit_run` event is set or the user types **'exit'** or a `KeyboardInterrupt` is detected, at which point the client sends **'exit'** message to the server to notify it of the client's leaving, sets the `exit_receive` event to stop the client's receiving operation and exits the loop.

4 UDP Chatroom Implementation

This section describes the implementation of a chatroom using the UDP protocol. The chatroom consists of a server that manages client connections and broadcasts messages, and a client that connects to the server to send and receive messages.

ServerUDP Class

The `ServerUDP` class implements a chatroom server using the UDP protocol. It coordinates several methods to manage client connections and message broadcasting. When the server is started with the `run()` method, it continuously listens for incoming messages. Upon receiving a message, the server checks if it is a new client

connection or a message from an existing client. If it is a new connection, the `accept_client()` method is called to handle the new client. If it is a message from an existing client, the server broadcasts the message to all other clients using the `broadcast()` method. The server continues to run until a `KeyboardInterrupt` is detected or an error occurs, at which point the `shutdown()` method is invoked to stop the server. The following is a detailed description of its methods:

__init__(self, server_port)

Instance variables:

- **`server_port`**: The port number on which the server listens for incoming messages.
- **`server_socket`**: The UDP socket used by the server to send and receive messages.
- **`clients`**: A dictionary that stores client addresses and their corresponding names.
- **`messages`**: A list that stores messages to be broadcasted to clients.

This method initializes the server socket, binds it to the local address and the specified server port `server_port`, and sets up the `clients` dictionary to keep track of connected clients. The `messages` list is used to manage messages that need to be broadcasted. When storing elements in the `messages` list, each element is a tuple containing the client's address who sent the message and the message content.

accept_client(self, client_addr, message)

This method handles a new client connection. The client's name is extracted from the received `message`. If the name is already in use, the method sends a **'Name already taken'** message to the client and returns `False`. Otherwise, it performs the following actions:

- Sends a **'Welcome'** message to the client.
- Adds the client's address `client_addr` as a key and the name as a value in the `clients` dictionary.
- Appends a tuple containing the client's address `client_addr` and **'User {x} joined'** to the `messages` list, where `x` is the client's name.
- calls the `broadcast()` method to broadcast the message to all connected clients and then returns `True`.

close_client(self, client_addr)

This method removes a client from the chatroom by deleting their entry from the `clients` dictionary and appending a tuple containing the client's address `client_addr` and **'User {x} left'**, where `x` is the client's name, to the `messages` list. It then calls the `broadcast()` method to broadcast this message to all connected clients. The method returns `True` if the client was successfully removed, otherwise `False`.

broadcast(self)

This method broadcasts the most recent message in the `messages` list to all connected clients in `clients` except for the client who sent the message. The message is sent to each client by encoding it and transmitting it by the server socket `server_socket`.

shutdown(self)

This method shuts down the server by sending a **'server-shutdown'** message to all connected clients and then closing their connections (`close_client()` method can be called for this). It then closes the server socket to stop listening for incoming messages.

get_clients_number(self)

This method returns the number of currently connected clients by returning the length of the `clients` dictionary.

run(self)

This method starts the UDP chatroom server and continuously listen for incoming messages. If a client sends a message **'join'**, the `accept_client()` method is called to add the client to the chatroom. If a client sends a message **'exit'**, the `close_client()` method is called to remove the client. For any other message, it is appended to the server's message list if the client is already in the chatroom. The server then broadcasts the message to all connected clients using the `broadcast()` method. The server continues to run until a `KeyboardInterrupt` is detected or an error occurs, at which point the `shutdown()` method is invoked to stop the server.

ClientUDP Class

The `ClientUDP` class implements a chatroom client using the UDP protocol. The client can connect to the server using the `connect_server()` method, send messages using the `send()` method, and receive messages using the `receive()` method. The client's main loop is managed by the `run()` method, which handles sending and receiving messages. The following is a detailed description of its methods:

__init__(self, client_name, server_port)

Instance variables:

- **`server_addr`**: The server's address to which the client will connect. In this case, the server address is set to the local machine's IP.
- **`client_socket`**: The UDP socket used by the client to send and receive messages.
- **`server_port`**: The port number on which the server is listening for incoming connections.
- **`client_name`**: The name of the client connecting to the server.
- **`exit_run`**: A threading event to control the main loop's running state.
- **`exit_receive`**: A threading event to control the message receiving loop's state.

This method initializes the client's UDP socket `client_socket`, sets the server address `server_addr` to the local machine's IP, and initializes the client's name and server port based on the provided `client_name` and `server_port` parameters. It also sets up threading events to manage the client's operation and message receiving.

connect_server(self)

This method attempts to connect the client to the server by calling the `send()` method with a message **'join'**. The method waits for a response from the server. If the response is **'Welcome'**, the connection is considered successful, and the method returns `True`. Otherwise, it returns `False`.

send(self, text)

This method sends the given `text` to the server. The message's format that is sent to the server is **'{`client_name`}:{`text`}'**, where `client_name` is the client's name and `text` is the content to be sent. The message is encoded and transmitted through the client's socket `client_socket`.

receive(self)

This method continuously listens for messages from the server until the `exit_receive` event is set. In the loop, the client receives a message from the server and decodes it. If the message is **'server-shutdown'**, which indicates that the server is shutting down, the method sets the `exit_run` and `exit_receive` events to stop the client's operation and exits the loop. Otherwise, the message is displayed to the user.

run(self)

Same as the `run` method in the `ClientTCP`, this method is the main loop for the client. It first connects to the server using the `connect_server()` method. If successful, connection is established, and the method start to handle receiving messages from the server using the `receive()` method simultaneously with handling user input in the main loop. In this loop, the user can input the text to be sent as a message to the server using the `send()` method. this loop continues until the `exit_run` event is set or the user types **'exit'** or a `KeyboardInterrupt` is detected, at which point the client sends **'exit'** message to the server to notify it of the client's leaving, sets the `exit_receive` event to stop the client's receiving operation and exits the loop.

5 Attachments

1 The **chatroom.py** file should contain the following code structure:

```
class ServerTCP:
    def __init__(self, server_port):
        pass
    def accept_client(self):
        pass
    def close_client(self, client_socket):
        pass
    def broadcast(self, client_socket_sent, message):
        pass
    def shutdown(self):
        pass
    def get_clients_number(self):
        pass
    def handle_client(self, client_socket):
        pass
    def run(self):
        pass


class ClientTCP:
    def __init__(self, client_name, server_port):
        pass
    def connect_server(self):
        pass
    def send(self, text):
        pass
    def receive(self):
        pass
    def run(self):
        pass


class ServerUDP:
    def __init__(self, server_port):
        pass
    def accept_client(self, client_addr, message):
        pass
    def close_client(self, client_addr):
        pass
    def broadcast(self):
        pass
    def shutdown(self):
        pass
```

```
    def get_clients_number(self):
        pass
    def run(self):
        pass


class ClientUDP:
    def __init__(self, client_name, server_port):
        pass
    def connect_server(self):
        pass
    def send(self, text):
        pass
    def receive(self):
        pass
    def run(self):
        pass
```

2 To test your `chatroom.py` file, create two python files named `server.py` and `client.py` to run the server and client respectively. The server file should import the `ServerTCP` or `ServerUDP` class from the `chatroom.py` file, while the client file should import the `ClientTCP` or `ClientUDP` class from the `chatroom.py` file. You can then run the server and client files in separate terminal windows to test the chatroom functionality.

Example:

`server.py` file:

```
# to run in terminal: python server.py
from chatroom import ServerTCP
server = ServerTCP(12345)
server.run()
```

`client.py` file:

```
# to run in terminal: python client.py --name
from chatroom import ClientTCP
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--name', '-n', type=str, help='Client name')
args = parser.parse_args()
client = ClientTCP(args.name, 12345)
client.run()
```

- Submission

REMEMBER!

- You have 4 coupons (for the entire semester) that will be automatically applied when you submit late.
- It is the student's responsibility to ensure the work was submitted and posted in GradeScope.
- Any assignment not submitted correctly will not be graded.
- Submissions through the email will not be accepted under any circumstances.
- Please check this page back whenever an announcement is posted regarding this assignment.
- Marks will be deducted if you submit anything other than the required Python files.
- Submit the assignment on time. Late submissions will penalty unless you have enough coupons to be applied (automatically).
- You may re-submit your code as many times as you like. Gradescope uses your last submission for grading by default. There are no penalties for re-submitting. However, re-submissions that come in after the due date will be considered late.
- Again, assignments will be run through similarity checking software to check for code similarity. Sharing or copying code in any way is considered plagiarism (Academic dishonesty) and may result in a mark of 0 on the assignment and/or be reported to the Dean's Office. Plagiarism is a serious offense. Work is to be done individually.

1. You must submit one file only to the Assignment submission page on Gradescope. The required file is as follows:

   chatroom.py

2. Make sure that you get the confirmation email after submission.
3. The submission will be using Gradescope Assignment-02 page .