Bilkent University

CS421 Programming Assignment 1

Emre Uncu

22003884

In NetChat Client application, Python's socket library is used for communication over TCP/IP. Communication with the server is done via HTTP protocol. Threading is used for simultaneous message receiving and command processing. The client is started in the main block seen below. Initially, IP and port information were taken as input while running the NetChatClient.py file. However, since this method was not feasible, it was later automated.

```
if __name__ == "__main__":
    username = sys.argv[1]
    # ip = input("Enter your IP address: ")
    # port = int(input("Enter your port number: "))
    client = NetChatClient(username)
    client.start()
```

Thanks to the thrading function at the beginning of the start() method, the message listening process is run in parallel in the background. If this did not happen, the program could only receive messages or only process commands. No commands would be entered while waiting for a message, and similarly, no messages could be received while entering a command.

```
    def start(self):
        self.register()
        self.get_user_list()
        threading.Thread(target=self.listen_for_messages, daemon=True).start()
        while True:
            command = input("Waiting for a command (to exit write EXIT): ")
```

The first function in the start() method, namely register(), is used to define a new user for the server. After the connection is established with the help of socket, a POST request is created for the registration process and an UPDATE line is added at the end. The resulting request was sent with sendall(). Then the response message was received and split for status code information. The 200, or OK, message indicates a successful registration process.

```
    def register(self):
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((self.server_host, self.server_port))

        data = f"UPDATE {self.username}@{self.ip}:{self.port}"

        request = "POST / HTTP/1.1\r\n"
        request += f"Host: {self.server_host}\r\n"
        request += "Connection: close\r\n"
        request += f"Content-Length: {len(data)}\r\n"
        request += "\r\n"
        request += data

        s.sendall(request.encode())

        resps = []
        while True:
            resp = s.recv(4096)
            if not resp:
                break
            resps.append(resp)

        s.close()
        response = b''.join(resps).decode()

        headers, _ = response.split("\r\n\r\n", 1)
        status_line = headers.split("\r\n")[0]
        status_code = int(status_line.split()[1])

        if status_code == 200:
            print(">>User registered in server!")
        else:
            print(">>Registration failed!")
```

After successful registration to the system, before accepting the command, the current user list was obtained from the server. For this, a GET request was created as follows. After sending the request as in the request() function, the response received was examined and the userlist in the content was obtained along with the OK message.

```
    def get_user_list(self):
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((self.server_host, self.server_port))

        request = "GET /userlist.txt HTTP/1.1\r\n"
        request += f"Host: {self.server_host}\r\n"
        request += "Connection: close\r\n"
        request += "\r\n"
```

After the client is registered with the server and the current user list is obtained, the command statements are as in the example below. Here, different operations are performed depending on the command contained in the input header.

```
            command = input("Waiting for a command (to exit write EXIT): ")
            if command == "EXIT":
                break
            elif command.startswith("LIST"):  # LIST
                parts = command.split(" ")
                if len(parts) == 1:
                    self.get_user_list()
                    print(f">>[{', '.join(self.known_users.keys())}]")
                else:
                    print(">>Invalid LIST command!")
            elif command.startswith("SEND_MULTI"): # SEND_MULTI [user1,user2,...]
"message"
```

The send_message required for the SEND and SEND_MULTI commands is as follows. First, it is checked whether the recipient is among the known persons. Then, the necessary information about the recipient is obtained from the known persons and the message content is sent again with the help of sendall(). The information of the sent message is stored because it is required in the READ command.

```
    def send_message(self, recipient, message):
        if recipient not in self.known_users:
            self.get_user_info(recipient)
        if recipient in self.known_users:
            ip, port = self.known_users[recipient].split(":")
            try:
                s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                s.connect((ip, int(port)))
                s.sendall(f"{self.username}: {message}".encode())
                s.close()

                if recipient not in self.message_history:
                    self.message_history[recipient] = []
                self.message_history[recipient].append(f"{message.split('"')[1]}
- user")
            except Exception as e:
                print(f">>Failed to send message: {e}")
        else:
            print(">>User not found!")
```

If the recipient information does not exist during the SEND command, get_user_info() below is called. This method is very similar to get_user_info(), the only difference is that it does not retrieve the IP and port information of each user from the list of users.

```
    def get_user_info(self, username):
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((self.server_host, self.server_port))
        request = f"GET /userlist.txt?username={username} HTTP/1.1\r\n"
        request += f"Host: {self.server_host}\r\n"
        request += "Connection: close\r\n"
        request += "\r\n"
```

The most difficult part of the assignment, delete(), examines the messages in history from the last to the first, and then updates the message as Deleted if it contains "- user", that is, if it is a sent message. The difficult part of the command, in order to delete the message from both clients, requires sending a signal indicating that the message has been deleted. When this command is applied at the beginning, "message deleted" is printed, but after the assignment is examined, the message history is printed instead, as in the READ command.

```
    def delete_message(self, recipient):
        if recipient in self.message_history and self.message_history[recipient]:
            messages = self.message_history[recipient]
            for i in range(len(messages)-1, -1, -1):
                if messages[i].endswith("- user") and not messages[i] == "Deleted
- user":
                    messages[i] = "Deleted - user"
                    if recipient in self.known_users:
                        try:
                            ip, port = self.known_users[recipient].split(":")
                            s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                            s.connect((ip, int(port)))
                            delete_signal = f"DELETE_MESSAGE:{self.username}"
                            s.sendall(delete_signal.encode())
                            s.close()
                            # print(">>Message deleted!")
                            if recipient in self.message_history:
                                for msg in self.message_history[recipient]:
                                    print(f">>{msg}")
                            else:
                                print(">>No messages found!")
```

The signal sent with the DELETE command and all other messages can be captured with listen_for_messages(). This method, used with threading, works continuously. If the received content contains the DELETE signal, the update process performed by the sender as "Deleted" is also performed by the receiver. In other cases, the received message is stored so that the READ command can use it. The message received information is printed and the user's command is waited for again.

```
    def listen_for_messages(self):
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.bind((self.ip, self.port))
        s.listen()
        while True:
            conn, addr = s.accept()
            data = conn.recv(1024)
            if data:
                message = data.decode()
                if message.startswith("DELETE_MESSAGE:"):
                    sender = message.split(":")[1]
                    if sender in self.message_history:
                        messages = self.message_history[sender]
                        for i in range(len(messages)-1, -1, -1):
                            if messages[i].endswith(f"- {sender}") and not
messages[i] == f"Deleted - {sender}":
                                messages[i] = f"Deleted - {sender}"
                                break
                else:
                    sender = message.split(":")[0]
                    if sender not in self.message_history:
                        self.message_history[sender] = []
                    self.message_history[sender].append(f"{message.split('"')[1]}
- {sender}")
                    print(f"\n\r\033[K>>You have a message from {sender}",
end="")
                    print("\nWaiting for a command (to exit write EXIT): ",
end="")
            conn.close()
        s.close()
```

As an example, SEND and DELETE commands are given below.

```
            elif command.startswith("SEND"):  # SEND user "message"
                parts = command.split(" ")
                if len(parts) >= 3:
                    recipient = parts[1]
                    message = " ".join(parts[2:])
                    self.send_message(recipient, message)
                    print(">>Message sent!")
                else:
                    print(">>Invalid SEND command!")
            elif command.startswith("DELETE"):  # DELETE user
                parts = command.split(" ")
                if len(parts) == 2:
                    user = parts[1]
                    self.delete_message(user)
                else:
                    print(">>Invalid DELETE command!")
```