CISC 5597 Distributed Systems

Lab 1: Networking — Multiple users chatting

Name: Swoichha Adhikari

Overview:

This report outlines the code development and workflow of a multithreaded client-server messaging application system developed using Python's socket programming. The server can accept multiple clients and assign each of them a unique random 6-digit integer ID from the range of 100000 - 999999. It also accepts different commands like list, forward, history, and exit from clients and will handle them accordingly. Once a client is connected to a server, clients can send and receive messages between each other. The code also handles errors for scenarios like sending a message from a client to itself, trying to use incorrect commands, using an invalid client ID to perform some invalid actions, etc.

Methodology:

Technologies Used in the codebase:

1. Python 3.x
2. Socket programming
3. Multithreading
4. Regular expressions for command parsing

System Architecture:

The application consists of two main components: the server and the client. The server-side code is written in the file ***server.py*** and the client-side code is written in the file named ***client.py.***

- **Client-Server Model:**

  Multiple clients can connect to the server, which then manages the communication between them. Client can use following command which are supported by the application and are coded in the *server.py* file:

  i. list: Displays list of all active client's IDs.
  ii. history <client_id>: Fetches the message history with the specified client.
  iii. forward <client_id>: <message>: Forwards a message to the specified client.
  iv. exit: Disconnects the client.

- **Multithreading:** The server uses multithreading to handle multiple clients simultaneously and handle the commands that they send to the server and to other clients.
- **Forwarding Message and History:** Clients can send messages to other clients using the command of the format "***forward <client_id>: <message>"*** and retrieve the chat history between different clients using the command of "***history<client_id>"***. There is a validation check added for both commands. A client cannot send messages or check history using their own id (assigned by the client during connect and can be checked using command "list")
- **Command Parsing with Regex:** All the commands like list, exit, forward, and history are case-insensitive, and this has been done using regular expressions.
- Color coded log: The logs/message displayed to client and server are color coded.
    I. Red: for error message
    II. Green: for successful events
    III. Yellow: for input
    IV. Blue: for server related log

Logic and Workflow

1. Client-Side Logic
    a. Establish Connection: The client first connects to the server (which should be up and running) on a specified IP and port.
    b. Receive Client ID: Upon successful connection, the server will assign a unique client_id of length 6 to the client.
    c. Send/Receive Messages: The client listens for messages from the server in a separate thread while sending commands or messages in the main loop to the server so that it can send messages to other clients.
    d. Exit Handling: The client can disconnect the connection to the server using comman "exit".

2. Server-Side Logic
    a. Manage Clients: Once the server is up and running it then waits for server to be connected. When a server is connected it then assigns a random client_id to each client and stores the connection.
    b. Command Handling: The server processes commands like list, exit, history, and forward. Each command is parsed using a regular expressions to handle case-insensitivity and other conditions (validating client_id, command format, etc.).
        i. List and Exit: Client can check the active available clients using command list and can end the connection to the server using command "exit".
        ii. Message Forwarding: When a client sends a message using the forward command, the server checks the validity of the command syntax and the target client. If the command is in the correct format, then it forwards the

message if the target is valid else, it will show some error message to the client.

iii. Chat History: Whenever a client sends a message to another client using the forward command, the server stores the messages. The chat history between clients is sorted based on timestamps that can be retrieved using the history command.

c. Error Handling: The server ensures clients cannot send messages to themselves or request their own history. It also validates the correctness of the command and the targeted client id.

Code Walkthrough

Client-Side

- **Connection to the server:** The client establishes a socket connection to the server at the given IP and port (set it up based on your server IP). After a successful connection, the server will send a randomly generated client id which will be stored in "client_id".

```
is_running = True
ip_port = ('127.0.0.1', 9999)

s = socket.socket()
s.connect(ip_port)

client_id = s.recv(1024).decode()
```

- **Receiving Messages:** To handle incoming messages from the server a thread is created. This allows the client to receive messages from the server while typing new commands asynchronously.

```
def message_check(s):
    """Thread that listens for incoming messages from the server"""
    global is_running
    while is_running:
        try:
            server_reply = s.recv(1024).decode()
            if server_reply:
                print(f"\n{server_reply}")  # Only display the server's reply once
                print('\033[93minput msg/command:\033[0m', end='', flush=True)  # Display the prompt after server response

            else:
                break
        except Exception as e:
            if is_running:
                print(f"\033[91m\nAn error occurred while receiving a message: {e}\033[0m")
            break
```

- **Sending commands to the server**: The client reads incoming messages from the other client/server and can send the message or command to the server.

```
while is_running:
    inp = input('\033[93minput msg/command:\033[0m').strip()  # First prompt for input
    if not inp:
        continue

    s.sendall(inp.encode())  # Send message to server

    if inp.lower() == "exit":  # Handle exit case (case insensitive)
        print("Goodbye!!!")
        is_running = False
        break

s.close()
```

Server Side:

- **Client connection management:** Each new connected client is assigned a random integer as client_id which is also sent to the client side. The client_id is stored in the client's dictionary. All the commands and function will use this client_id.

```
def link_handler(link, client):
    client_id = int(random.randint(100000, 999999))
    print('\033[92m\nserver start to receiving msg from [%s:%s]....\033[0m' % (client[0], client[1]))
    link.sendall(f'{client_id}'.encode())
    clients[client_id] = link
    available_commands = "Use any from these available commands: list; history <client_id>; forward <client_id>: <message>; exit;"
```

- **List command:** This section handles the listing of all active client. It provides the list of all active client's id.
- **Exit command:** This section handles the ending of the connection. Using the "exit" command, the client can end the connection with the server.

```
  if re.search(r"(?i)exit", client_data):
      print('\033[92m\ncommunication end with\033[0m [%s:%s]...' % (client[0], client[1]))
      break

  elif re.search(r"(?i)list", client_data):
      """Returns all the active client IDs."""
      sk_clients = list(clients.keys())
      final_list = []
      for sk_client in sk_clients:
          if client_id == sk_client:
              final_list.append(str(sk_client)+ "(your id)")
          else:
              final_list.append(str(sk_client))
      link.sendall(f"\033[92m\nActive Clients ID:\033[0m {final_list}".encode())
```

- **Forward Command**: This section handles forwarding messages between clients through the server. Multiple validation has been added such as checking if the target ID is the same as the sender (to prevent sending messages to oneself), ensures the target is a valid client, and forwards the message if all conditions are met.

```
elif re.search(r"(?i)forward\s\d+:\s*.+$", client_data):
    '''
    Implemented regex to handle the format
    msg format: forward target_ID: message_content
    '''
    input_split = client_data.split(':')
    target_id = int(input_split[0].split()[-1])
    message = input_split[1]
    messages.setdefault(client_id ,{}).setdefault(target_id, []).append([message,datetime.datetime.now()])

    if target_id:
        # Check if the target_id is the same as the client_id
        if target_id == client_id:
            link.sendall('\033[91m\nError: Cannot send message to yourself\033[0m'.encode())
        # Check if target_id is in clients.keys() (valid client)
        elif target_id in clients:
            clients[target_id].sendall(f"{client_id}: {message}".encode())
            link.sendall(f'\033[92m\nMessage forwarded to client {target_id}\033[0m'.encode())
        # If target_id is not in clients, it's an invalid client
        else:
            link.sendall('\033[91m\nError: Invalid Targeted ID\033[0m'.encode())
    else:
        link.sendall('\033[91m\nSomething went wrong. Cannot find the Target ID. Try again !!!\033[0m'.encode())
```

- **History Command:** This section retrieves and returns the chat history between the client and the target client based on message timestamps. Like the forwarding command validation has been added to check if the target ID is the same as the sender id.

```
elif re.search(r"(?i)history\s\d+", client_data):
    '''
    Returns chatting history between the requested client
    and the client with the ID listed
    #Format
    history target_client_ID
    '''
    target_id = int(client_data.split()[1])
    chat_hist = chat_history(client_id,target_id)
    if target_id == client_id:
        link.sendall('\033[91m\nError: Cannot check hsitory message to yourself\033[0m'.encode())
        # Check if target_id is in clients.keys() (valid client)
    else:
        if chat_hist:
            msg = []
            for chat in chat_hist:
                if client_id == chat[0]:
                    status="(me)"
                else:
                    status=""
                msg.append(f"{str(chat[0])+status} : {chat[1]}")
            link.sendall('\n'.join(msg).encode())
        else:
            link.sendall(f'\033[91m\nNo history found with client ID {target_id}\033[0m\n'.encode())
```

Key Challenges and Solutions:

1. Command Parsing with Regex:
    - Problem: I wanted to make the commands case-insensitive (forward, FORWARD, etc.).
    - Solution: Regular expressions were used with the (?i) flag to handle case-insensitive matching. This allowed flexibility in command input.
2. Message Forwarding:

- Problem: I wanted to have a validation so that the clients would not be able to send messages to themselves.
- Solution: I added logic to check using if-condition such that if target_id is equal to client_id then return an error that if they match.

3. Chat History:
    - Problem: Messages needed to be retrieved in timestamp order between clients.
    - Solution: To fix this I have stored messages in a dictionary and sorted them based on their timestamp for each request. So that when history command is used then the messages will be shown based on the timestamp.
4. Validating Client IDs:
    - Problem: As a client should not be able to send messages or check history with itself, I needed to add a validate that a target client exists before sending messages or retrieving history.
    - Solution: Used target_id in clients to check if a target client is connected.

How to Execute:

First change the IP and port number in each file.

1. Start the Server:
    - Run the server-side script: python3 server.py.
    - The server will start listening for client connections on IP:port.
2. Start the Client:
    - Run the client-side script: python3 client.py.
    - The client will connect to the server and be assigned a unique client ID.
3. Available Commands:

    All the commands are case in-sensative.

    - list: Displays all active client IDs.
    - forward <client_id>: <message>: Sends a message to another client.
    - history <client_id>: Retrieves the chat history between the client and another client.
    - exit: Disconnects from the server.