COMP3234B Computer and Communication Networks

Assignment 1: A Simple Instant Messaging Application (12%)

[Learning Outcome 4]

Due by: 23:59 Friday February 23, 2024

Total mark is 100.

I. OVERVIEW OF THE ASSIGNMENT

In this assignment, you will implement a simple instant messaging (IM) application using Python socket programming. In this application, multiple clients can connect to a server, get authorized, and then send private messages to another client or broadcast messages to all other clients via the server.

The IM application consists of two parts: the server program and the client program. The server program should always be running and use a welcome socket to wait for connection requests from clients. The client program establishes TCP connections with the server program. After a connection is set up, the client needs to send its user name and password to the server, and can further participate in the application after successful authentication. An authenticated user is able to query and display the names of concurrently online users, send private messages to one specific online user, and send broadcast messages to all the other online users. Each user may join and leave the system at any time.

II. DETAILED DESIGN OF THE APPLICATION

Figure 1 illustrates the key components of the server program and the client program, as well as their interactions. Detailed designs are given as follows.

Server basics:

The main method of the server program takes 2 parameters: (1) server's listening port; (2) full path of a .txt file, which contains user names and passwords for all users (clients) that may participate in the application.

Upon start, the server program listens at the specified port. Upon receipt of a TCP connection request from a client, the server creates a new connection socket, and a new thread that handles the socket delegated to this client. The main server thread then continues listening on the port.

Each line of the .txt file contains the user name and password of one user, in the following format:

username password

(username and password should be replaced by the actual user name and password.)

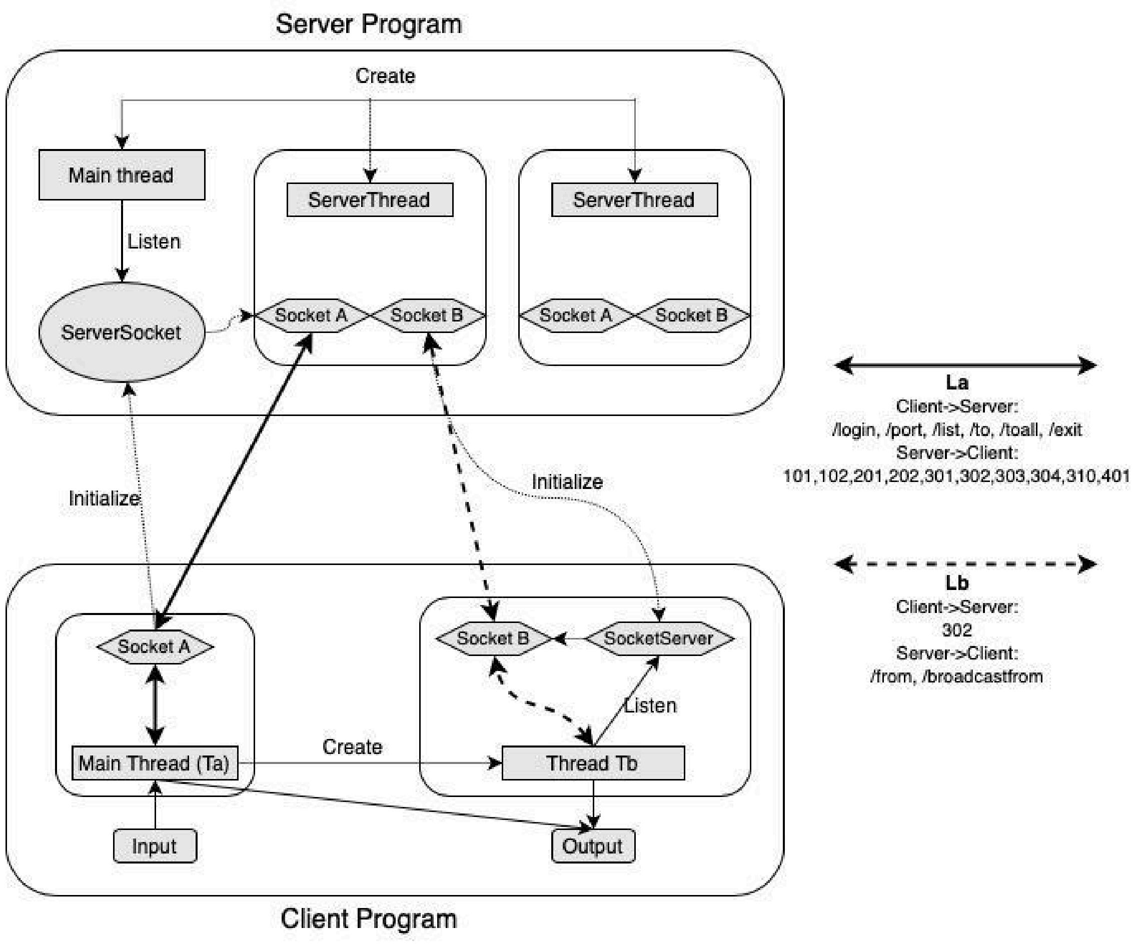

Figure 1

Client basics

The main method of the client program takes 2 parameters: (1) IP address of the server; (2) listening port of the server. Upon start, the client connects to the server at the IP and port by setting up a TCP connection. After the connection is successfully established, the client program goes to the stage of user authentication.

User authentication

The client prompts its user the following:

```
Please input your user name:
```

and then waits for the user to input the user name. After the user has input the user name, the client prompts:

```
Please input your password:
```

and then waits for the input of the password. After the user has input its password, the client sends the user name and password to the server, with a message in the following format:

/login username password\n

(Note: username and password should be replaced by the actual user name and password.)

If the authentication at the server is successful, the server sends the following message back to the client, and adds the user's name in the online user list:

101 Authentication successful\n

Otherwise, the server responds with the following message and the authentication procedure is repeated (i.e., the client prompts the user to enter user name and password again):

102 Authentication failed\n

When the authentication is successful, the main client thread and the server thread that handles the connection with this client, enter the next stage to setup a second TCP connection.

Establishing another TCP connection for chat message forwarding

So far the main client thread (referred to as thread `Ta` hereafter) has established one TCP connection with the server (called connection `La` hereafter). Using this connection, the client can send authentication messages to the server, as well as other command messages (to be discussed in "**Handling commands**" below). For example, it can send a chat message using the **/to** or **/toall** command to the server over connection `La`, and the server then forwards it to the target receiver(s) of the message. To receive chat messages from other users via the server, connection `La` cannot be used, as receiving and sending of chat messages may well occur at the same time. To resolve this issue, another TCP connection needs to be established between the server and the client, used for delivering incoming chat messages from the server to the client.

In particular, a new thread is created (called thread `Tb` hereafter) to handle the additional TCP connection (called connection `Lb` hereafter) with the server. To set up TCP connection `Lb`, the client side should create a ServerSocket instance, which listens to the incoming connection request from the server, while the server side should create a Socket instance and initiate connection `Lb` with the client. When TCP connection `Lb` is established, the server sends chat messages over the connection to the client, and waits for receipt of responses; the client receives chat messages from the server, displays the messages to its user, and responds to the server with the receipt status of the messages.

The ServerSocket instance at the client side should listen at a randomly selected available port. In order for server to initiate TCP connection `Lb`, the client should tell the server this port number. The main client thread `Ta` should send a message in the following format to the server via connection `La`:

/port port_number\n

(Note: port_number should be replaced by the actual port number)

Upon receipt of this message, the server creates a socket which connects to the client at the specified port (the server should know client's IP from the socket for connection `La`).

If connection `Lb` is successfully established, the server acknowledges the client by sending a message over connection `La`:

201 Build connection successful\n

Otherwise, it sends to the client a message:

202 Build connection failed\n

Upon receiving the failure message, the client program should display the message to the user, and then quit.

Handling commands

After connection `Lb` is successfully built, main client thread `Ta` waits for the input from the user, echoes it on its terminal, sends the command to server via connection `La`, waits for response from the server and echoes on its terminal. The commands the user can enter are as follows:

/list\n

/to target_username chat_content\n

/toall chat_content\n

/exit\n

(note that a user does not need to explicitly type \n following each command when typing the command in the client terminal, but just hits "Enter" or "Return" on the keyboard after finishing typing a command; the client program should append \n to each command when sending it to the server)

1) **/list**: Upon receiving a list command from a client, the server responds with a status code and the names of all online users in the following format:

301 username_of_user1 username_of_user2 … username_of_usern\n (Note: user names should be separated by a space)

2) **/to**: Upon receiving a to command from a client, the server first checks if the target receiver is online or not. If it is online, the server forwards the chat content to the target receiver over connection `Lb` between the server and the target receiver, with the following message:

/from chat_sender_username chat_content\n

If the target receiver receives the message successfully, it displays the chat content on its terminal, and replies with the following message:

302 Message receipt successful\n

Upon receiving the above message, the server sends a response to the chat sender with the following message:

303 Message delivery successful\n

If anything goes wrong within the above actions (either the target receiver is not online, or some exception is thrown when forwarding the message), the server replies to the chat sender with the message:

304 Message delivery failed\n

3) **/toall**: Upon receiving a toall command from the client, the server forwards the message to all the online users except the sender itself, using `Lb` connections between the server and each of the other online users, with the following message:

/broadcastfrom chat_sender_username chat_content\n

When each user receives the message, it displays the chat content on its terminal and responds to the server with a message:

302 Message receipt successful\n

After response messages have been received from all other online users, the server sends a message to the chat sender:

303 Message delivery successful\n

If anything goes wrong with the above actions, the server replies to the chat sender with the message:

304 Message delivery failed\n

4) **/exit**: Upon receiving an exit command from the client, the server responds with

310 Bye bye\n

It then cleans up resources allocated to connecting with this client, and updates its list of online users.

Upon receiving the response from the server, the client program cleans up its allocated resources and then quits.

Dealing with incorrect message format

If the server receives a command message in the wrong format, it simply responds with

401 Unrecognized message\n

Note: For better understanding of the interactions between a client and the server, please refer to example interactions given in the Appendix.

III. IMPLEMENTATION NOTES

1) Synchronization among multiple threads accessing the same data.

As you may have learned in an Operating System course, multiple threads may try to read/revise some shared data simultaneously, which may lead to errors if no synchronization is performed. In our application, multiple server threads may simultaneously want to send chat messages to the same target receiver via the same TCP connection `Lb` between the server and the receiver. To prevent errors, each server thread should acquire a lock before it can use the TCP connection to send and receive the respective response, and release the lock afterwards. Learn more about Python threading lock (https://docs.python.org/3/library/threading.html) to implement this lock.

2) You can assume that user names and passwords do not consist of space(s).

3) You can assume one user does not log in through two or more client programs simultaneously.

4) Your program should have sufficient robustness. In the networked environment, a TCP connection may be broken, or a remote end host may suddenly leave the system at any step of the dialog. Your program should be able to properly handle the exceptions thrown and print out error messages accordingly.

5) You should make sure your message formats, the formats of inputs and outputs, strictly comply to the stated design.

IV. PROGRAM TESTING

We will test your programs using normal interaction cases like those described in this handout, and with various abnormal cases, such as incorrect message formats, broken network connections, unexpected crash of the server or the client program, etc. Therefore, you should test your program in similar fashion during development.

V. GROUP OR INDIVIDUAL

You should work on this assignment individually, or in a group with at most two students.

You will get 10% bonus mark if you work on the assignment individually, i.e., your mark will be your original mark * 1.1, upper bounded by the full mark of 100.

VI. SUBMISSION

You should submit the following files in a zip file, named as a1-yourUID.zip (individual work) or a1-UID1-UID2.zip (group work; only one submission by one of the group members is needed for each group):

(1) client.py

(2) server.py

(3) A readme file to provide any additional information on the implementation and execution of your programs that you deem necessary for us to know.

Please submit the .zip file on Moodle:

(1) Login Moodle.

(2) Find “Assignments” in the left column and click “Assignment 1”.

(3) Click “Add submission”, browse your .zip file and save it. Done.

(4) You will receive an automatic confirmation email, if the submission was successful.

(5) You can “Edit submission” to your already submitted file, but ONLY before the deadline.

APPENDIX

Suppose at 00:00, Jacky launches his client program. At that time, Bob and Michael have already been online (their user names are “bob” and “michael” respectively). Jacky chats with them. After chatting, Jacky leaves the system. The following are the messages sent between the clients and the server. Note: (1) time is shown only to match chat messages sent from a sender with chat messages received by the receiver(s); (2) SERVER(x) and CLIENT(x) are shown to illustrate over which TCP connection the message is sent, where SERVER(x) means that the server sends the message to the client via link `Lx` (x=a or b) and CLIENT(x) means the client sends the message to the server via link `Lx` (x=a or b); (3) the note in ( ) at the end of some lines is to illustrate correspondence between respective messages.

Messages sent between Jacky and the server:

00:00 CLIENT(a): /login jacky 12345\n

00:01 SERVER(a): 101 Authentication successful\n

00:10 CLIENT(a): /port 30232\n

00:11 SERVER(a): 201 Build connection successful\n

```
00:20 CLIENT(a): /list\n
00:21 SERVER(a): 301 jacky bob michael\n
00:30 CLIENT(a): /to bob Hello! Nice to meet you!\n (00:31 Bob)
00:32 SERVER(a): 303 Message delivery successful\n
00:40 CLIENT(a): /to michael Hi, Michael!\n (00:41 Michael)
00:42 SERVER(a): 303 Message delivery successful\n
00:50 SERVER(b): /from michael Hi, Jacky!\n (00:49 Michael)
00:51 CLIENT(b): 302 Message receipt successful\n
01:00 CLIENT(a): /toall How about meeting at 3pm today?\n (01:01 Bob Michael)
01:02 SERVER(a): 303 Message delivery success\n
01:10 SERVER(b): /from bob Jacky, nice to meet you too!\n (01:09 Bob)
01:11 CLIENT(b): 302 Message receipt successful\n
01:30 SERVER(b): /broadcastfrom bob OK.\n (01:29 Bob)
01:31 CLIENT(b): 302 Message receipt successful\n
01:40 SERVER(b): /broadcastfrom michael No problem.\n (01:39 Michael)
01:41 CLIENT(b): 302 Message receipt successful\n
01:50 CLIENT(a): /exit\n
01:51 SERVER(a): 310 Bye bye\n
```

Messages sent between Bob and the server:

```
00:31 SERVER(b): /from jacky Hello! Nice to meet you!\n (00:30 Jacky)
00:32 CLIENT(b): 302 Message receipt successful\n
01:01 SERVER(b): /broadcastfrom jacky How about meeting at 3pm today?\n (01:00 Jacky)
01:02 CLIENT(b): 302 Message receipt successful\n
01:09 CLIENT(a): /to jacky Jacky, nice to meet you too!\n (01:10 Jacky)
01:11 SERVER(a): 303 Message delivery successful\n
01:29 CLIENT(a): /toall OK.\n (01:30 Jacky Michael)
01:31 SERVER(a): 303 Message delivery successful\n
01:40 SERVER(b): /broadcastfrom michael No problem.\n (01:39 Michael)
01:41 CLIENT(b): 302 Message receipt successful\n
```

Messages sent between Michael and the server:

```
00:41 SERVER(b): /from jacky Hi, Michael!\n (00:40 Jacky)
00:42 CLIENT(b): 302 Message receipt successful\n
00:49 CLIENT(a): /to jacky Hi, Jacky!\n (00:50 Jacky)
00:51 SERVER(a): 303 Message delivery successful\n
```

```
01:01 SERVER(b): /broadcastfrom jacky How about meeting at 3pm today?\n (01:00 Jacky)
01:02 CLIENT(b): 302 Message receipt successful\n
01:30 SERVER(b): /broadcastfrom bob OK.\n (01:29 Bob)
01:31 CLIENT(b): 302 Message receipt successful\n
01:39 CLIENT(a): /toall No problem.\n (01:40 Jacky Bob)
01:41 SERVER(a): 303 Message delivery successful\n
```