COMP 415/515 – Distributed Computing Systems

Assignment-1

Due: February 24, 2019, 11:59 pm (Late submissions are not accepted.)

Submit your assignment deliverables to the folder: **F:\COURSES\GRADS\COMP515\HOMEWORK\A1**

Note: This is a group-oriented assignment, you are encouraged to form groups of 2, and we suggest a fair task distribution at the end of this assignment description. Working individually is allowed but not recommended.

Development of a Centralized Group Messenger on AWS EC2

This assignment is about the **centralized organization** as the system architecture in distributed platforms. It involves application layer software development using the **client/server model**, **TCP socket programming**, **threads, virtualization**, and **Lamport's distributed timestamp algorithm**. As the distributed platform, **Amazon Web Service Elastic Compute Cloud (AWS EC2)** would be used to deploy the centralized group messenger to be developed, and the results would be reported.

You are asked to design and implement operations of a group messenger service model. You need to use socket programming for the communication between the clients and the server. Upon connecting to the server all clients are directed to a single shared messenger room. Each client can write to the messenger room, as well as receive messages shared by other clients. On receiving a messenger message, the server (who keeps track of all the online clients) sends the received message to all connected (online) clients. This interaction resembles the group messaging in the messengers like WhatsApp.

Overview:

Figure 1 shows an overview of the system with 3 clients: USER1, USER2, and USER3, and the server: Messenger Server. You are asked to implement the *client* and *server* programs communication through TCP sockets. When a user runs the client program, she is asked for the IP address of the server to connect. Once a new user gets connected, the server asks for its username. The user needs to specify a username. The server should accept the client's username only if there is no other connected client to it with the same username. Once the server accepts the client's username, the client is directed to a messenger room where it can post a message to the messenger room and receive the posted messages by other users. For the sake of simplicity, we assume that a user does not receive any message while she is writing a message to the messenger room. The centralized design has two main components; a client program and a server program. The assignment is implemented in two parts; client-server interaction and timestamping.

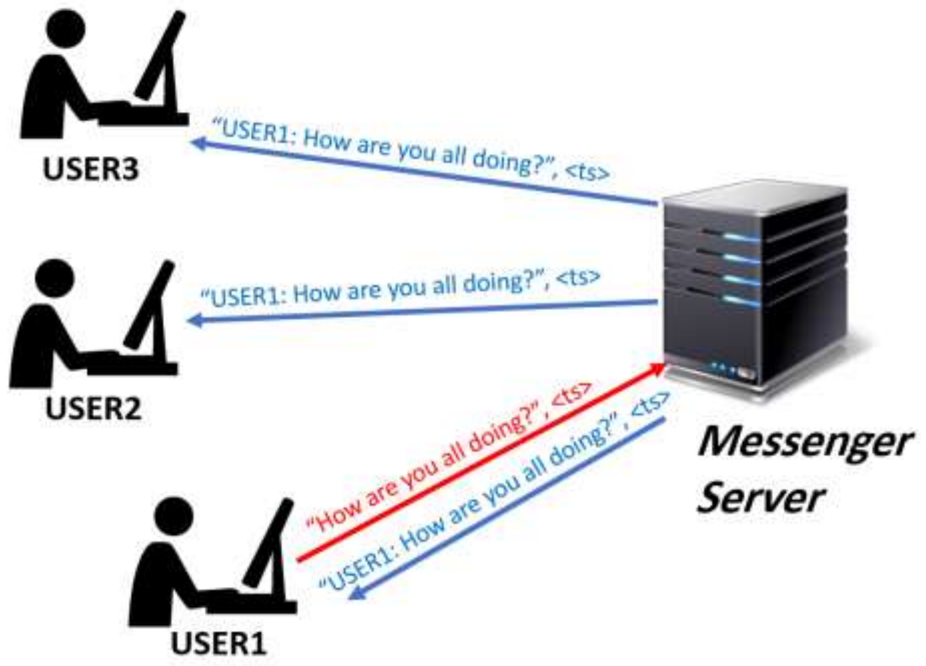
USER3
"USER1: How are you all doing?", <ts>
USER2
"USER1: How are you all doing?", <ts>
"How are you all doing?", <ts>
"USER1: How are you all doing?", <ts>
USER1
Messenger Server

Figure 1- The overview of the group messenger system. The username of each client is shown beneath the client's icon. The message sent by USER1 the messenger server is indicated in red. The blue messages are those that the messenger server sends back to all users upon receiving USER1's red message. <ts> shows the Lamport's timestamp associated with the messages, which is explained in Part2 of this description.

Part1- Client-Server interaction

The Messenger Server:

- Supports multiple **concurrent persistent TCP connections** from multiple clients. In this assignment, we dedicate port number 4444 to the Group Messenger protocol. Hence, both the client and server should interact with each other only via this port.
- Should keep a list of connected online clients as (IP, username) pairs.
- Should listen for three events:
  - Joining a new client
  - departing a connected online client
  - posting a new message by a connected online client
- **Joining a new client:** Server should listen for the incoming clients. Once a connection is established for a new client, the server should listen for the client's selected username. Upon receiving the client's username, the server should check the list of connected online clients against the existence of the selected username by the new client. If the new client's selected username has already been taken by another connected online

client, the server should send **reject** message and wait for receiving a new selected username from the client. Otherwise, it should send the **accept** message to the client, and update the list of connected online clients by adding the (username, IP) of the new client. The server should also send a join notification of the new client to all the connected clients (including the new one). In the example of Figure 1, when the client "USER1" joins the messenger room, the server should send the message *"USER1 joined the messenger room" to all the connected online clients*.

- **Departing a connected online client:** A client may leave the messenger room by sending a "*<username> quit"* message (e.g., "USER1 quit" in the example of Figure 1) to the server, or just terminating its connection. The latter should be detected by the server by checking the online status of the clients in the connected online clients' list. For example, this can be done by the server frequently expecting a heartbeat message from each active client. The heartbeat message is a simple message that is sent frequently (every 1 ms) from the client to the server to notify the server that the client is online and connected. Once the server realizes that a client departs the system, it should remove it from the list of connected online clients, and notify all the existing connected online clients by sending a message to them like *"USER1 left the messenger room"*. *Note that, exchanging heartbeat is one way to check the online status of a client. You may come up with another idea.*
- **Posting a new message:** Server should conduct persistent TCP connection with the connected online clients, and listen to them. Once a connected online client posts a message to the server, the server should recover its username (using the IP address of the connected client) from the list of connected online clients, and broadcast the message in the form of **<username>: <message>** to all the connected online clients (including the sender of the message). In the example of Figure 1, when the client with the username of "USER1" sends the message of "How are you all doing?" to the messenger server, the server should send a message in the format of "USER1: How are you all doing?" to all the connected clients including USER1 itself.

Client side:

- On startup, the client should ask for the IP address of the server from the user, and establish a persistent TCP connection to the server.
- Once the connection established, the client should ask for the username from the user and sends it to the server. The client either receives **accept** or **reject** from the server. The client should continue this procedure by asking another username from the user until it receives accept from the server. The client should show proper interactive messages to the user. For example, upon receiving a reject message from the server, the client should show something like "Your username has already been taken, enter a new one".

- Receiving an accept message from the server means that the client is redirected to the messenger room. From this point on, the client should listen to two events: reception of a message from the server, or reception of a message from the user.
- **Message reception from server event:** A message received from the server via TCP connection corresponds to one of the following events.
    - Posting a message by a user to the messenger room
    - The arrival of a new user to the messenger room
    - The departure of an existing user from the room

  The client program should display the received message to the user.

- **Message reception from user event:** A message received from the user via the input (i.e., keyboard) corresponds to the event of posting a message by the user to the messenger room. The client program should send the message to the server.
- The user can terminate its interaction to the server by either sending the "*<username> quit"* message (e.g., "USER1 quit") to the server or by closing its connection to the server (e.g., terminating the client process).

Part2- Lamport's distributed timestamp algorithm

In this part, you are asked to implement the Lamport's distributed timestamp algorithm at each process (i.e., client and server). Each process should hold its local counter. The local counter value of client processes are initialized randomly at the execution time (e.g. a value in the range 1 and 100). The local counter value of server is initialized to zero on its running time. Processes should display the initialized local counter values. The local counter value is updated as specified by the Lamport's timestamp algorithm assuming only two types of events: send of a message and receipt of a message. The processes should timestamp the messages they send and receive.

In the example of Figure 1, assume that the local Lamport's distributed timestamp of USER1 when it aims to send the "How are you all doing?" message to the server is 5. Thus, the client should send ("How are you all doing?", 5) to the server, which is then forwarded to all the connected online clients by the server. On receiving this message, the server, as well as clients, should extract the received timestamp value, and update their local counter accordingly. In this part of the assignment, clients should not only show the received message, but also the timestamp of the message, and the timestamp value assigned to the message reception event.

Part3- Deployment on the AWS EC2

- Create 5 virtual machines (VMs) in AWS where each VM should reside on a distinct AWS data center e.g., one in Virginia, the other one in Beijing, etc.
- Deploy and run the server on one VM, and one client process on each of the rest 4 machines.
- Test the correct execution of your system against the mentioned events (i.e., user join and leave as well as send and reception of messages). Take a screenshot of each scenario and insert it into your report (as specified later in this description).

Report

The assignment report should contain the step-by-step description of the following in such a way that anybody who reads the report could do the exact configuration without using any external references. Hence, you are strongly recommended to use screenshots and explain your answers by referring to the screenshots. In the report, you should provide captions, figure numbers, and referral to the figures as we did in this assignment description.

- How do you create your AWS VMs?
- How do you deploy your code on AWS?
- How does your Group Messenger system work on AWS with one server and 4 clients configuration? In specific, you should provide screenshots and description of user join and leave as well as send and reception of messages.
- You should explain the method by which the server identifies online status of connected clients (e.g., through heartbeat, etc.).

Deliverables:

You should submit your source code and assignment report (<u>in a single .rar or .zip file</u>). The name of the file should be <last name of group members>.

- Source Code: A .zip or (not .rar) file that contains your implementation in a single Eclipse, IntelliJ IDEA, or PyCharm. If you aim to implement your assignment in any IDE rather than the mentioned one, you should first consult with TA and get confirmation.

- The **report** is an <u>**important part of your assignment**</u>, which should be submitted as both a .pdf and Word file. The **report acts as proof of work to assert your contributions to this assignment.** Anybody who reads your report should be able to reproduce the parts we asked to document. If you need to put the code in your report, segment it as small as possible (i.e., just the parts you need to explain) and clarify each segment before heading to the next segment. For codes, you should be taking a **screenshot** instead of copy/pasting the direct code. Strictly avoid **replicating the code** as whole in the report, or leaving code **unexplained**.

Demonstration:

You are required to demonstrate the execution of your Group Messenger on AWS with the defined requirements. The demo sessions would be announced by the TA. Attending the demo session is required for your assignment to be graded. **All group members** must attend the demo session. **The on-time attendance of all group members is considered as a grading criterion.**

Important Notes:

- **Please read this assignment document carefully BEFORE starting your design and implementation. Take the report part seriously and write your report as accurate and complete as possible.**
- Your entire assignment should be implemented in C++, Java, or Python. Implementation in network-based languages (e.g., Ruby) is not allowed. For implementation in other languages, you should first consult with TA and get confirmation.
- In case you use some code parts from the Internet, you must provide the references in your report (such as web links) and explicitly define the parts that you used.
- You should **not** share your code or ideas with other assignment groups. Please be aware of the KU Statement on Academic Honesty.
- For the demonstration, you are supposed to bring your own laptop and run your program on your own laptop. Bring more laptops if you need more than one.
- Your entire code should be well-commented. If you are programming in Java, you are required to provide JavaDoc as well. If you are implementing in any other language, you are required to add the general description of each function, its inputs, and outputs.
  You may use the following reference to get ideas about writing a clear and neat JavaDoc.: https://www.oracle.com/technetwork/java/javase/documentation/index-137868.html
  Also, you may use the following reference to get ideas about commenting your codes properly:
  https://improvingsoftware.com/2011/06/27/5-best-practices-for-commenting-your-code/

Suggested task distribution:

We recommend you to work in a group of 2, and suggest the following task distribution.

Student 1: Client-Server interactions and Lamport's timestamp implementation.

Student 2: AWS deployment and report.

Please be informed that despite the task distribution, we expect all the group members to learn the project concepts as the entirety.

References:

- Socket Programming in Java [Link]
- Multithreading in Java [Link]
- AWS EC2 [Link]
- Centralized Organization (Section 2.3 of the textbook)
- Threads and Virtualization (Section 3.1 and 3.2 of the textbook)
- Client-server model (Section 3.3 and 3.4 of the textbook)
- Lamport’s distributed timestamp algorithm (Section 6.2 of the textbook)

Good Luck!