COMP 415/515 – Distributed Computing Systems

Assignment-2

Due: March 15, 2020, 11:59 pm (Late submissions are not accepted.)

This is an individual assignment. You are not allowed to share your codes with each other.

Decentralized Group Messenger with

Causally Ordered Multicasting and GoRPC

This assignment is about the **decentralized organization** as the system architecture in distributed platforms. It involves distributed software development using the concepts of **unstructured peer-to-peer (P2P) systems**, **Remote Method Invocation (RMI) with Go RPC**, **threads, virtualization**, and **Causally ordered multicasting using Vector Clocks**. As the distributed platform, **Amazon Web Service Elastic Compute Cloud (AWS EC2)** would be used to deploy the decentralized P2P group messenger to be developed, and the results shall be reported.

You are asked to design and implement operations of a group messenger service model, which does not rely on a centralized server. Essentials:

- You need to use RMI for communication among peers (i.e., processes). Each process should support multiple **concurrent RMI** for other processes.
- As we describe later, every process should know the IP address and port number of all the other processes, which enables it to communicate them directly using RMI.
- Upon execution and startup, each process would be able to send a message to the other processes where all the messages are assumed to appear in a single messenger room.
- Each process can write to the messenger room, as well as receive messages shared by other processes.
- A process would send a message by **multicasting** it to all other processes. The multicast should be done in a causally ordered manner using vector clocks (as studied in class), which enables the processes to provide causally ordered messages for their users.
- Upon receiving a multicast message, each of the recipients processes prints the message. The multicast operations done by the participating peers result in the flooding of the messages to all group members, which eventually results in all the participating peers to obtain a copy of each message.
- It is assumed that the multicast group size is static.

Overview:

Figure 1 shows an overview of the system with 4 processes: P1, P2, P3, and P4. Each process has a unique username (i.e., identifier) that is the combination of its IP address and port number. P1 wants to say “Hello” to other processes. As such, he sends his message “Hello” together with his vector timestamp denoted by ts(m) to all the other processes i.e., P2, P3, and P4.

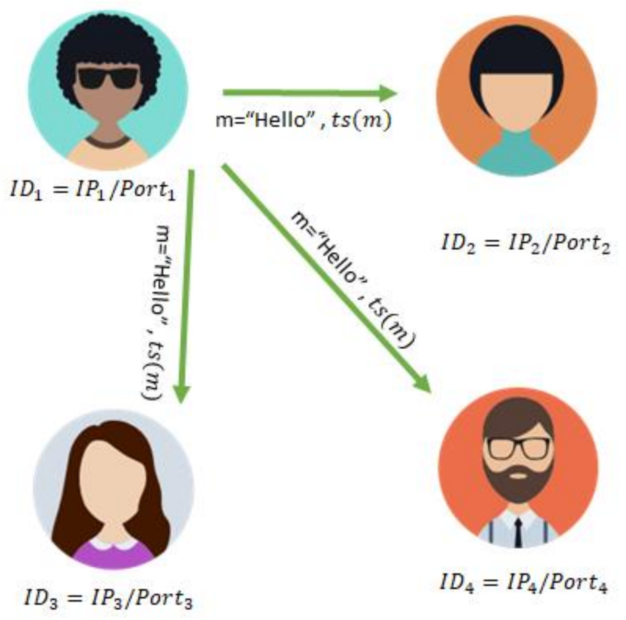

Figure 1- The overview of the distributed group messenger system. The username of each process is shown beneath the process’s icon. The message multicast by P1 is indicated next to the green lines. ts(m) shows the vector timestamp associated with the message, which is explained in Part 2 of this description.

Part1- Group Messenger Process (35 points)

Unique Identifier:

A Group Messenger Process represents a participating peer in the system. Each process in the system is identified using the combination of its IP address and port number. For example, a Group Messenger Process that is executed on a machine with IP address of 1.2.3.4 and the port number of 1111 has the identifier of 1.2.3.4/1111. The identifier of a process resembles its corresponding user's username. In this way, each process has a unique identifier (i.e., username) without the need to contact any centralized server and asking for the uniqueness of its identifier.

Neighbours' list:

For a process, the neighbours' list is the list of other processes in the group. You should provide a text file containing the IP/port of every process in the system. The name of the file should be "peers.txt" and reside on the same directory as the process code. Each line of the text file should solely represent one process. The text file is the same for each process. Upon startup, each process loads the text file, and parses it into the IP/port of other process. A process should simply discard the entry of the text file that represents itself by comparing the entries against its own IP and port number. Using the text file, every process knows the number of all the processes in the system.

Posting a message:

(Note: The messaging can only start after all the processes in the input file are started.)

Send: Once a user enters a new message, the process should multicast it to its neighbors. We call the process that performs a multicast as the **sender process**. For this sake, the sender process should send the message to all of its neighbors. Sending a message to each neighbor is done by invoking an RMI call to the ***messagePost*** method of the neighbor's process. We call the neighbors of the sender process that are the receivers of the multicast as **receiver processes**.

Receive: A process receives a message through an RMI call (from the sender) to its *messagePost method.* The *messagePost* method should only receive an object of type *message* and print the content of the message. You are free to follow your implementation of the *message* object, however, it should contain the following fields:

- **Transcript:** The text that the user of the sender process has entered.
- **OID:** Identifier of the original sender process.

Part2- Vector Clocks and Causally Ordered Multicasting (45 points)

In this part, you are asked to implement Causally Ordered Multicasting using Vector Clocks. Each process should hold its own local vector clock. The vector clock of each process is initially all zeros. Processes should display the initialized local vector clock at the startup. The processes should timestamp the messages they initiate using their vector clock. For this part to be accomplished, you should add a vector timestamp to your implemented message structure. Upon the reception of a message and its vector timestamp, the receiving process updates its own vector clock. Messages should be printed at each peer considering the causality. In other words, a process should only print a receipt message to the user if all the messages that casually precede it have been received and already delivered to the user. A correct implementation is the one that all the messages are shown to the user in a causally ordered manner. You should implement a buffering mechanism to keep the messages that are received but not delivered to the user because they do not casually succeed the previous ones. Your process should print the content of the buffer every time a message is added to it or getting dropped out of it.

Note: To provide a consistent way of indexing the processes in the vector clock, you may consider their index as it appears in the input file (i.e., peers.txt). As long as all the peers are given the same file they are having a consistent view towards each other's indices in the vector clock.

Report (20 Points)

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.

- Create 5 virtual machines (VMs) in AWS.
- Deploy and run your processes into the VMs.
- Test the correct execution of your system for message multicasting and delivery. Take a screenshot of each scenario and insert it into your report.
- How did you enable the RMI functionality over ***messagePost?***
- How did you implement the message data structure?
- How did you implement the causally ordered multicasting using vector clocks?
- You should demonstrate the cases where delivery of a message is postponed due to its vector timestamp, and put screenshots of the content of the process buffer accordingly in the report. Hint: You can introduce delays by thread sleep.

Deliverables:

You should submit your Go source codes and assignment report (in a single .zip file) on the Blackboard CMS. The name of the file should be (lastname_KU_id.zip).

- The **report** is an **important part of your assignment**, 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 Decentralized 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.

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 **Go "net/rpc" package**.
- 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 others. Please be aware of the KU Statement on Academic Honesty.
- For the demonstration, you are supposed to bring your own laptop to run your program.
- Your entire code should be well-commented.

References:

- Unstructured peer-to-peer systems (Section 2.3 of the textbook)
- Threads and Virtualization (Section 3.1 and 3.2 of the textbook)
- Vector Clocks and Causally Ordered Multicasting (Section 6.2 of the textbook)

Good Luck!