KOÇ ÜNİVERSİTESİ
COMPUTER ENGINEERING

COMP 415/515: Distributed Computing Systems

Assignment-3 (Optional)

Due: May 27, 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 Go RPC

This assignment is about **decentralized organization** as the system architecture in distributed platforms. It involves application layer software development in Go using the client/server model, threads (Goroutines), Go RPC and virtualization. Amazon Web Service Elastic Compute Cloud (AWS EC2) as the distributed platform would be used to deploy the decentralized group messenger, 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 should use **Go RPC** for communication among the peers (i.e., processes).
- 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 (group communication) it to all other processes.
- Upon receiving a multicast message, each of the recipient 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 obtaining 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” and the sender sequence number of the message (denoted as ts(m)) to all the other processes i.e., P2, P3, and P4.

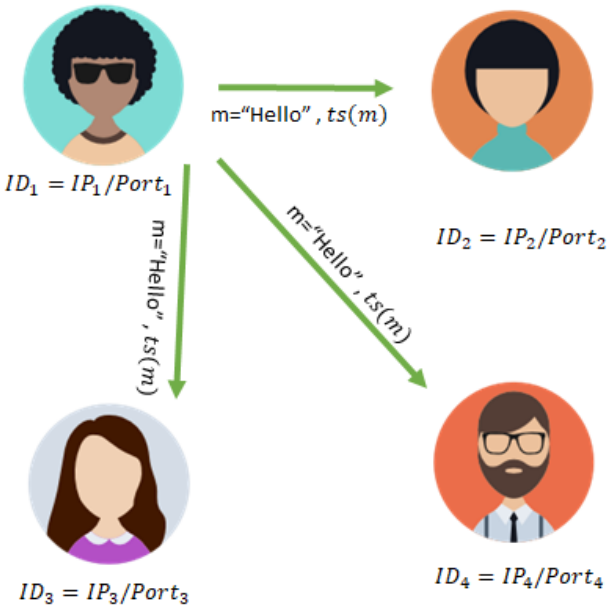

Figure 1- The overview of the distributed group messenger system. The username of each process is shown beneath the process's icon. The message multicast by P1 is indicated next to the green lines. ts(m) is the sender sequence number of the message.

Peer Process

Unique Identifier:

A Peer Process represents a participating peer in the messenger system. Each process in the system is identified using the combination of its IP address and port number. For example, a Peer 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.

Neighbors' list:

For a process, the neighbors' 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 "group.txt" and reside in 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 the 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 RPC 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 RPC 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.
- **SID:** Identifier of the original sender process.

Deployment on the AWS EC2

- Create 6 virtual machines (VMs) in AWS.
- Deploy and run your peer processes on the VMs.
- Test the correct execution of your system for message multicasting and delivery.

Deliverables:

You should submit your Go source codes and report describing the design of your implementation supported by screenshots of execution scenarios (in a single .zip file named <yourname-surname-ID>).

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.

KOÇ ÜNİVERSİTESİ
COMPUTER ENGINEERING

Important Notes:

- **Please read this assignment document carefully BEFORE starting your design and implementation.**
- Your entire assignment should be implemented in Go.
- Your code should be well-commented.
- 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.

Good Luck!