Assignment 2
COMP515 Fall 2019
Distributed Computing Systems
Erhan Tezcan 0070881
14.02.2020

1. Task

We are asked to write a chat program, where each client is a physically separate client with their own IP address. Their identification is done via concatenating their IP and the actively listened port, for example a client with IP `56.122.25.35` and port `8080` would have a "username" as: `56.122.25.35/8080`. In this chat program, each clients are aware of the other client's addresses on startup. The actual chat room will start once connection among each and every client is established. The messaging is done via RPC and causally ordered delivery of the messages are implemented.

2. Implementation

We split the program into several parts to explain it in a better and more coherent way.

- **Launching the Client**
  Each client has a text file in their active directory, which contains the ip and port of the other clients. From this file, they also know about their own ip and port. This is done via entering a number from the command line as an argument, for example: `go run ./peer.go 1` means that the first ip and port in the text file is this client. So the client will ignore that line, and read the other lines.
  After this, the client starts a goroutine for the messenger service, and also opens the port and listens to incoming connections. This goroutine is explained in the next item.
- **Messaging Routine**
  First, the client needs to make sure it has established a connection with all other clients. So to do that, it dials the clients it had just read from the text file, and in a cyclic fashion keeps dialing until all of them are connected. When a dialing gets no response, the client waits for a small amount of time to reduce the number of dials to a reasonable amount.
  Once all connections are established, each client is in the messaging mode, where every input from the user to the client's terminal willl be registered as a message and be multicasted.
- **Message**
  A message is composed of 4 items:
  (1) *Timestamp* `[]int`
  This is the vector clock timestamp that is piggybacking the message.

(2) *Si* `int`
This is the sender's index on the timestamp, which the reciever will use to check the causal order conditions by comparing it's own vector clock and the timestamp.
(3) *Oid* `string`
This is the id of the sender, which acts as the username.
(4) *Transcript* `string`
This is the message text itself.

2.1. **Sending of a Message.** When a client is sending a message, it first increments its own vector timestamp. It is also important to note that this is the only occurance of the increment of the timestamp, which will be the assumption for the causally ordered delivery. After the increment happens, the message is constructed, where the vector clock, sender index on the vector clock, sender id and the message itself makes up the message construct. Then in a for loop we send the message to each client. A perhaps small optimization here could be creating a goroutine for each message, thus making this feel more like a concurrent multicast rather than a series of unicasts, but still I thought it would be an unnecessary overhead of routine creation for a fine grained task like this.

2.2. **Receipt and Delivery of a Message.** When a message is recieved we print the reciever vector clock upon recieving, the timestamp attached, the sender id and the message itself in a single line. This also helps us explicitly see if the ordering is correct and the conditions are met.

2.2.1. *Causally Ordered Delivery.* We have implemented the causally ordered delivery using the methods that we were taught in the class. First, we assume that the vector clock increment only occurs during the sending of the message, by the sender itself, at the sender. Normally, we would check the incoming message, compare the vector clocks, and then if the conditions are met deliver the message, and then check the buffer for possible deliveries. If the condition was not met, we would add the message to the buffer. I have implemented this slightly differently, to reduce code size. In my code, when a mesage is recieved, it is immediately put in the buffer. Then, the program checks if there is a deliverable message in the buffer. When a delivery happens, the program checks the buffer again. This will go on, until no deliveries are made.
Let's explain these conditions in a more explicit way now: Suppose we recieve a message $m : \{ts, si, oid, txt\}$ where ts is the timestamp, si is the sender index, oid is the sender id, txt is the transcript. Also on the clientside, vc is the vector clock. Here is the algorithm:

(1) The message is added to buffer
(2) We enter the delivery loop. For every message in the buffer,

(a) Check if it is deliverable. To do this, we must meet the two conditions:

$$ts[si] = vc[si] + 1$$
$$ts[i] \leq vc[i], \forall i \neq si,$$

These two conditions basically mean that the reciever was expeting m as the next message from the sender, and the reciever has not seen any messages that were not seen by the sender when it sent m.

(b) If conditions are met, the message is delivered, and upon delivery the reciever will update it's vector clock as below:

$$vc[i] \leftarrow \max(vc[i], ts[i]), \forall i$$

(c) If conditions are not met, the message is kept in the buffer.

(3) If there was a delivery in the previous step, that step will be repeated. If not, the message recieving is completed, and the remote procedure will return.

2.3. **Special Messages.** In my implementation, for several debug purposes and ease of use, I have added 4 special messages.

- **__EXIT__**: When a client sends this message, after the multicast it will terminate. Each reciever will also respond to the remote procedure call, but each reciever, right before the return the call, will start a goroutine. This goroutine will wait one second and then terminate the reciever client. In other words, when a client sends **__EXIT__** message every client will gracefully terminate.
- **__DELAY__**: When a client multicasts this message, each reciever will immediately put it to the buffer and will not deliver it. This is done to test the causal ordering.
- **__VEC__**: When a client recieves this message for the input, it will not multicast it. Instead, it will print its vector clock to the console.
- **__BUF__**: When a client recieves this message for the input, it will not multicast it. Instead, it will print the contents of the buffer. If a client has recieved the **__DELAY__** message, we would expect to see that message here if it is not delivered yet. This also applies to any other message that was not delivered due to causal ordering.

3. Demonstration

In this section, we will show some cases of the program in action. The results here are taken directly from the 5 AWS virtual machines. It is important to note that, when we are launching the clients, say in this order:

- `go run ./peer.go 1`
- `go run ./peer.go 2`
- `go run ./peer.go 3`
- `go run ./peer.go 4`
- `go run ./peer.go 5`

Then the order of the ip and ports in the `peers.txt` file should also respect the machines that are running these clients. For example, for the order above, the first line in the text file should be the ip and port of the peer that runs with command line argument `1`. Carelessness regarding this point will cause in wrong behaviour of the program, where the client will connect to itself, mistaking it with another client.

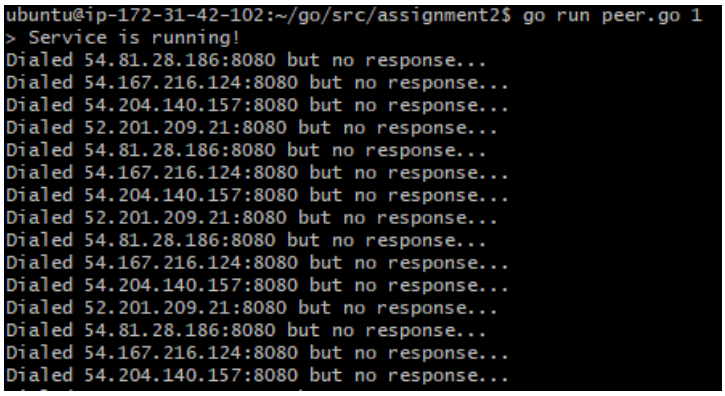

FIGURE 1. Launching of the first client.

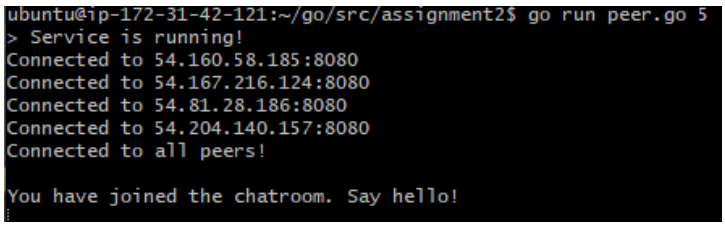

FIGURE 2. Launching of the last client.

We launch the client in this order: $1, 2, 3, 4, 5$. As you can see, the first client, when launched, is the only one online so it will keep dialling other clients but will not get a response (figure 1). When the last client is launched, it will be able to connect to every client that it will be looking for, therefore we see the result on figure 2.

On figure 3 we see that first all clients are connected, then client 5 (right) says *hey there!* and then *whatsup?*. We see the delivery on the client 1 (left), and then this client responds *all good thank you!*. Another client also joins the conversation by saying *im also good thanks*, and we see the delivery of this message on both sides. Lastly, client 5 responds with *cool!*. Notice that we explicitly show the vector clock and message time stamp on the delivery of a message.

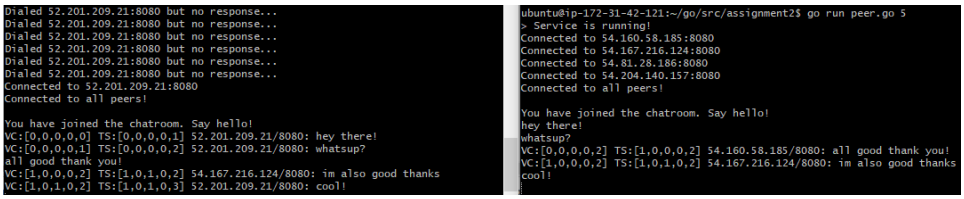

FIGURE 3. Conversation of 3 clients.

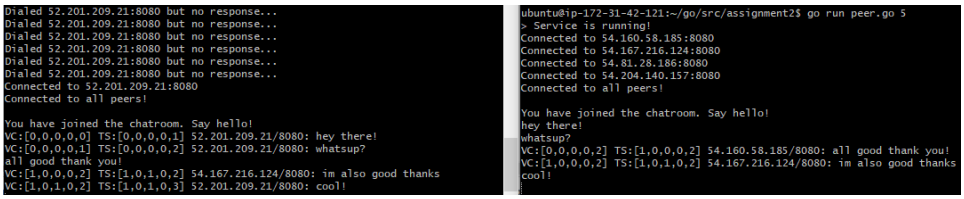

FIGURE 4. Causally ordered delivery.

On figure 4 we see how the causal ordering works. This is actually a continuation of figure 3. We see that at client 1 (left) side, **__VEC__** prints the vector clock, **__BUF__** shows the buffer, which is empty at first time. Then, client 5 (right) sends a **__DELAY__** message, which is recieved but not delivered by the clients. We can see this by checking the buffer again. After that, client 5 sends a message: this is after the delay and client 1, upon recieving this, first delivers the delayed message from the buffer and then the actual message, thus showing the causally ordered delivery.

FIGURE 5. **__EXIT__** message. On the left, we see the client that sends the message. Notice how it exits immediately. On the right, we see a client that recieves that message.

Thus we conclude that we have implemented every task given to us in the assignment. Functionalities will be demonstrated in further detail during the actual one-to-one demonstrations.