Lab Exercise 3 – Using Dynamic Vector Clocks to Implement Group Messaging

CIS656 – Fall 2017

Due Date: November 9, 2017 (see below for description of deliverables.)
Grade Weight: 25 points (out of a total of 100 Lab points)

Objective

By completing this project, students will learn:

- How to send messaging using UDP datagrams
- How to implement dynamic vector clocks to maintain causal ordering of multicast messages among a dynamic group of peer processes.

Lab Description

Lamport's logical clock concept (see refs at end for original paper) have been extended in a variety of ways that have proven useful in constructing distributed systems. By using vector clocks instead of scalar clocks, we can determine causal relationships between events. Others have extended vector clocks to work with a dynamic set of processes instead a fixed set of known processes (see Landes paper).

In this lab exercise you are going to implement vector clocks for a dynamic group of chat processes. Clients will multicast messages via UDP datagrams sent serially. When using UDP datagrams to communicate over a wide area network, it is possible that packets will be lost or arrive at the destination in a different order than what they were originally sent.

In this application, every time a user enters a message, all current and future users that join the group will receive the message. On a fast local area network, it is likely that the UDP messages will arrive in the same order they were transmitted, so to simulate the fact that packets may arrive in an indeterminate order, you will be provided with a server process that will relay any sent messages to all other clients in the group. However, the server will insert random delays in transmission to ensure the packets will arrive in an indeterminate order. However, you can assume packets will NOT get lost. They will simply arrive out of order.

Assume the only "observable event" is the transmission and reception of a message to/from the group. Each process will receive every message sent by every process but itself. Each process p_i maintains its own private vector clock C_i and increments $C_i[i]$ each time it transmits a message. When a message m is received by p_j from p_i with the vector clock C_m piggy backed on it, the message can only be printed out when the following conditions are met:

$C_m[i] == C_j[i] + 1$ *(that is, m is the next message p_j expects from p_i)*
$C_m[k] <= C_j[k]$ for all k != i *(p_j has seen all messages seen by p_i before m)*

Hence, a client can simply buffer the messages in a queue ordered by vector timestamp and only remove and print the first message in the queue when the above condition is met. When a message is finally printed, p_j's local clock is updated by setting the kth value as follows:

$C_j[k] = \max(C_j[k], C_m[k])$ for all k.

To understand this approach better, look at the group conversation shown in Figure 1 below. Bob asks a question to the group, but the transmission to Alice takes a bit long to get there. In the meantime, Chuck's response is received almost immediately by Alice and Bob, as is Bob's subsequent response to Chuck's message. All of this is seen by Alice before she actually receives Bob's original transmission! Clearly the order in which we print the messages out to Alice is going to be different than the order in which she received them!

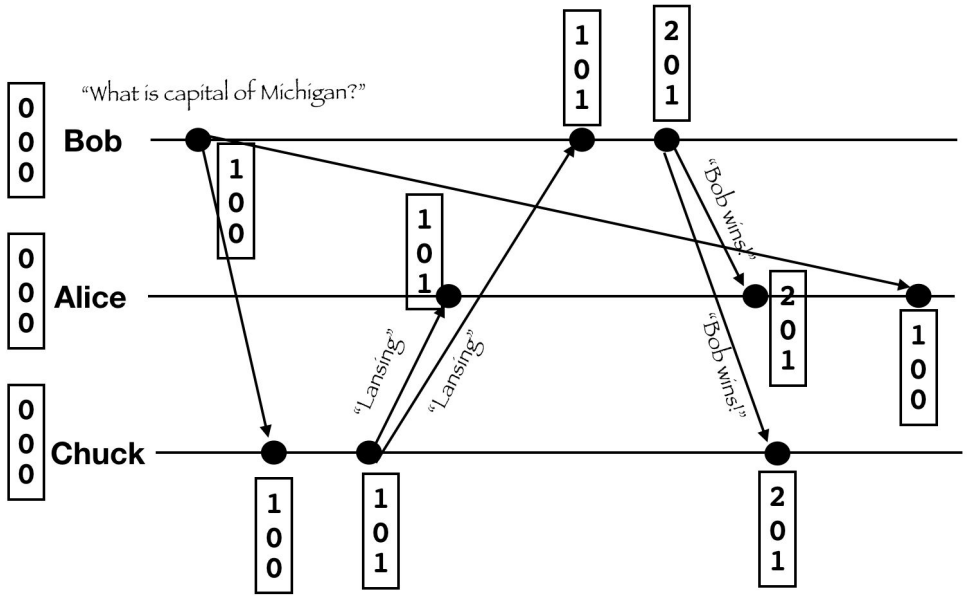

Figure 1. Scenario displaying the possible effects of packets arriving out of order in UDP transmissions.

Take a look at the table below. The second column indicates the order in which the actual messages were received by Alice, but the second column shows the order in which they get printed following the rules described above.

What is the capital of Michigan?	3	1

Lansing	1	2
Bob wins!	2	3

Implementation Details

You will be provided the following code artifacts:

- A runnable server relay process (in the form of a jar) that multicasts messages (via serially transmitted UDP datagrams) to all processes other than the message's original sender. The server will send packets to participating clients on the same IP / port it received packets from that client on.
- An abstract interface of a Clock you will implement as a VectorClock and a set of unit tests to help you more clearly understand and validate the desired functionality of your clock implementation.
- A MessageTypes class that defines the types of messages that will be exchanged.
- A Message class implementation that will encode / decode all messages in a given JSON format, and also provide methods for sending and receiving messages via UDP datagrams.
- A stubbed out MessageComparator and a stubbed out VectorClockComparator. You will provide an implementation for the latter, and then implement the former using the latter!
- A PriorityQueue implementation that can be used in conjunction with the MessageComparator to implement the ordered queue of messages weighting to be printed.

With these code assets you will implement the client group messaging process itself. On startup the client will prompt the user to enter a unique username. The client will then attempt to send a MessageTypes.REGISTER message with the "sender" field set to the proposed name. If that name is not taken, the server will immediately respond with a MessageTypes.ACK message. The pid field of the ack message will have a unique scalar process id that the process should use moving forward as its process id. If the name has been taken the server will send a MessageTypes.ERROR message, with the "message" field set to a printable error message. The client should exit immediately in this case.

After a slight delay, the server will follow the ACK message with a transmission of the message history (e.g. all messages of type CHAT_MSG transmitted by all members of the group to-date). However, the order in which these messages are received will not be the order in which they are sent! Any future messages sent to the group will also be forwarded to the client as well, though not necessarily in order.

A client can send a message the user enters on the console by simply ticking its local vector clock, and passing it along with the message to the server as a message of type CHAT_MSG. The server will automatically forward a copy of the message to every registered client.

Thread Model

It is recommended that your implementation has two threads – 1) a UI thread that will prompt the user to enter messages, and subsequently send them to the server, and 2) a message receive thread that will receive all messages and enforce the above rules to print them out.

The message receive thread will have logic that is something like this:

```
while (true) {
        msg = receive a message
        add msg to priority queue
        topMsg = peek (don't' remove!) at top message on queue
        while (topMsg != null) {
                if topMsg meets the print condition (see above description!)
                        print topMsg
                        remove topMsg from queue)
                        topMsg = peek at top message on queue
                } else {
                        topMsg  = null;  // we're still waiting for an earlier message
                }
        }
}
```

You may find it helpful to validate your implementation, by setting the message's `tag` attribute to the arrival order. Then when you print out messages you can print the tag and compare the order in which the messages originally arrive to the order you are actually printing them out.

When printing the message, at minimal you should print the name of the person who sent it followed by the message sent. For example:

```
Bob: Hi there everybody!
```

Implementing the Comparators

Recall that with vector clocks causality is preserved so that:

$a \rightarrow b \Leftrightarrow C(a) < C(b)$

Hence, you can implement your vector clock comparator so that it returns the following values:

$a \rightarrow b$: then return -1
$b \rightarrow a$: then return 1
otherwise: return 0

Remember that if neither a→b nor b→a are true then a and b are said to be *concurrent,* that is causally independent, so their ordering with respect to each other simply doesn't matter.

You can implement your MessageComparator by simply return the value generated by the VectorClockComparator when comparing the clocks associated with the two messages.

Implementing VectorClock
The only data structure you need in your vector clock implementation is a Map<String,Integer>. The key will be a string representation of a process's unique pid. If you read the unit tests you will see that they require the clock support situations where the clocks of various processes may not have all the pids of every participant. For example, a client process will only become known to other clients when it sends its first message. With the help of the reference below and the unit tests, you should be able to figure out how to handle the dynamic nature of the required vector clock.

Running the Server
The binary executable jar (server.jar) that is provided for you will be used by chat processes to multicast messages to each other. Assuming you have the JDK set up in your terminal window environment, you can start this process from the command line with the following command:

```
java –jar server.jar
```

The actual messages sent to and received from this process are in JSON. The interactions above described in a high level should actually be sufficient for you to implement our client, though the message parsing source code is provided and you could take a look at that to get a better handle on the details, if interested.

The server is hard-coded to expect incoming datagrams on port 8000. Make sure you kill any other process that might be bound to that port (there shouldn't be any!)

Deliverables

There are two deliverables required in order to receive full credit for this lab exercise.

Provide Demo to Instructor
Provide a demo of the working client to the instructor before or on the designated due date.

Electronically Submit the Source Code
Archive all the source code needed to build and execute your implementation in a zip file and submit it to Blackboard. The timestamp on your submission must be no later than the due date to receive credit. An URL to github repo is also acceptable.

References / Resources

You will want to consult the following references – all of which are posted on the class blackboard site for your convenience:

- Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558-565. [Online]
- Landes, Tobias. (2006). Dynamic Vector Clocks for Consistent Ordering of Events in Dynamic Distributed Applications.. 31-37. [Online]
- All About UDP Datagrams. Oracle Java Tutorials [Online]