SOFE4790U Distributed Systems (Fall 2018)

Lab#4

Implementing Logical Clocks

Objective

In this lab you will be experimenting with Lamport Logical Clocks, and Vector Clocks.

Note: The Lab tasks must be completed in the lab. If you don't show up for a lab, you will receive a zero and there is no need for you to submit a lab report.

Disclaimer: Parts of this lab have been adapted from (http://www.cs.wustl.edu/~kjg/CS333_SP97/logicaltime.html) and source code from the book (Concurrent and Distributed Computing in Java, by Vijay K. Garg)

Background

An event in a distributed system is either a local step, a send event, or a receive event. We often need to assign an order to the events, but processes in the system do not have synchronized clocks, and there is no global clock. One simple solution is to maintain logical clocks at the processes. Logical clocks do not tick like real time clocks, but instead keep track of the order of events that occur at each process and ensure that events are assigned consistent logical times according to the following happens before relation:

Event A happens before event B if and only if:

1) A and B occur at the same process (either internal local steps, send events, or receive events) and event A occurs before B at that process, or
2) A is a send event and B is a receive events (it may be received by multiple processes) for the same message.

The Lamport logical clock is an algorithm that assigns logical times to all the events in the system in a way that is consistent with the happens before relation, as follows:

- Each process keeps an integer, initially 0, that represents its internal logical clock.
- Whenever a process takes a local step, it increments its logical time by 1, and the incremented time is considered to be the time of the local event.
- Whenever a process sends a message (send event), it increments its logical time by 1, and sends that new time with the message. This time is considered to be the logical time of the send event.
- Whenever a process receives a message (receive event), it first compares its own logical clock time to the logical time sent with the message, and sets its own logical clock to be the maximum of the two times. Then, it increments its logical time by 1, and the incremented time is considered to be the time of the receive event.

Notice that the logical times of events at a given process always increase as the execution proceeds. Furthermore, notice that the logical send time is always less than the logical receive time. The logical times assigned to the events gives us a partial order on all events.

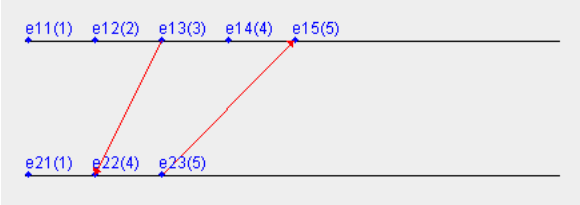
Task 1: Visualizing Lamport Logical Clocks (10 marks)

The sample Java application (Lamport.java), which can be downloaded from Blackboard is an example for visualizing Lamport Logical Clocks. Download the code, compile it – don't run it as yet. As input, the application expects the number of processes you want to simulate, the number of events at each of the processes, and the relationship between the events. Read below before you start entering input to the application).

As you enter the number of processes and events, note the following:
Event numbers at process 1 start at 11
Event numbers at process 2 start at 21
Even numbers at process 2 start at 31
…and so on.

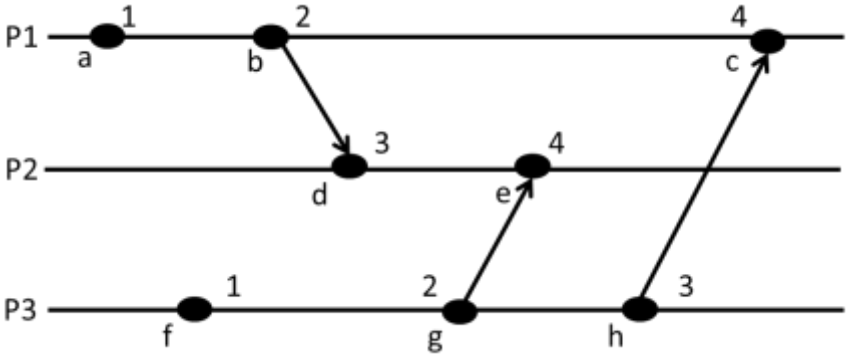
In the sample run below for example, you will notice that for process2 event2 (really 22) I have entered 13 for the relationship – this means that event 13 at process 1 is sending an action to event 22 at process 2. It is confusing but we need to make the best of this application and learn something from it.

Enter the number of processes: 2
Enter the number of events for process 1: 5
Enter the number of events for process 2: 3
Enter the relationship (0 for intra-process, or event number of other process:
For process 1...
For event1: 0
For event2: 0
For event3: 0
For event4: 0
For event5: 23
For process 2...
For event1: 0
For event2: 13
For event3: 0
1
2
3
4
5
1
4
5



Sample Run 1) Now, run the application (Lamport.java) with the above input and observe the output The application prints on the command line, the logical clock values for the various events, and also the above diagram. Are the values correct?

Sample Run 2) As another example, re-run the application for the following diagram, and check if it produces the correct Lamport timestamps (note that the Lamport timestamps on the following diagram are correct).



Demonstrate and discuss the results with the TA.

Task 2: Logical Timestamps (20 marks)

Consider the Java code LamportClock.java available in the Blackboard. Note that the given code implements all three types of relations (Local, Send & Receive) between the processes. Now modify the given code such that it can take the number of processes & dynamic number of events for each process and the relation between the processes (like Lamport.java) as input. At the end it should provide the logical timestamp for every event depending on the inputs. You may need to consider using Java threads to simulate processes. Remember your code should be able to simulate for any dynamic number of processes and dynamic number of events in each process. Now test your code using the diagram shown in sample run 2.

Hint: You can implement Task 2 using your own way of programming. A more complex approach is followed in the provided source code LamportProcess.java. If you prefer to use LamportProcess.java, there are 6 well-defined **TO DO** tasks for you to implement in that file in order to run the application successfully and produce the correct results.

Demonstrate and discuss the results with the TA. Note that TA may test your code with other diagram like the take-home exercise provided in the class.

Task 3: Vector Clocks (20 marks)

Vector clocks (as discussed in class) were developed to overcome the shortcoming of Lamport's clocks: the fact that from L(a) < L(b) we cannot conclude that a→b. Consider the Java code (VectorClock.java, available on Blackboard) that implements a vector clock. Now modify the code such that it does the same task as task 2. The difference is it should successfully implement Vector Clock timestamps instead of Lamport Clock. Your task to test the VectorClock class to simulate processes that take internal steps and send messages randomly. You should use Java threads to simulate processes. Test your code using the diagram shown in sample run 2.

Hint: You should be able to reuse some of the code you developed in Task 2 (either your own code or the modified LamportProcess.java).

Demonstrate and discuss the results with the TA. Note that TA may test your code with other diagrams like the take-home exercise provided in the class.

Lab Report (50 Marks):

Please submit your report and source code in one archive (.zip)

1. Submit the source code you have written for Tasks (2) and (3). (**10 marks**)
2. For Task (1), discuss the results of sample run (1) and (2). Did it show the expected output? If no then what should be the correct output? Why the visual presentation is important to understand the logical clock relationship? What did you learn from Task 1? Did you face any difficulty? If so mention that. (**10 marks**)
3. What did you learn from task 2? What challenges did you face and how did you solve that? Now, given two events a and b, if event a → b, then Lamport timestamp of a is less than Lamport timestamp of b, i.e. L(a) < L(b). However, the converse is not true (if L(a) < L(b) then we cannot infer that a→b). Discuss specific examples or events from your sample run for Task (2) to demonstrate both cases. (**15 marks**)
4. Describe any challenges you faced with task 3 and how you solved them. Discuss specific examples or events from your sample run for Task (3) to demonstrate that vector clocks do overcome the shortcoming of Lamport's clocks. (**15 marks**).

Submit your source code and lab report (in Word or PDF) on Blackboard by 11:59pm on Friday, November 9. No late submissions will be accepted no matter what is the reason.