Experiment No.4
Implement Clock Synchronization algorithms
Date of Performance: 15/03/24
Date of Submission: 18/03/24

Aim: To implement Clock Synchronization algorithms

Objective:- To implement Clock Synchronization algorithms

Theory:

Lamport invented a simple mechanism by which the happened-before ordering can be captured numerically. A Lamport logical clock is an incrementing software counter maintained in each process.

It follows some simple rules:

1. A process increments its counter before each event in that process;
2. When a process sends a message, it includes its counter value with the message;
3. On receiving a message, the receiver process sets its counter to be the maximum of the message counter and its own counter incremented, before it considers the message received.

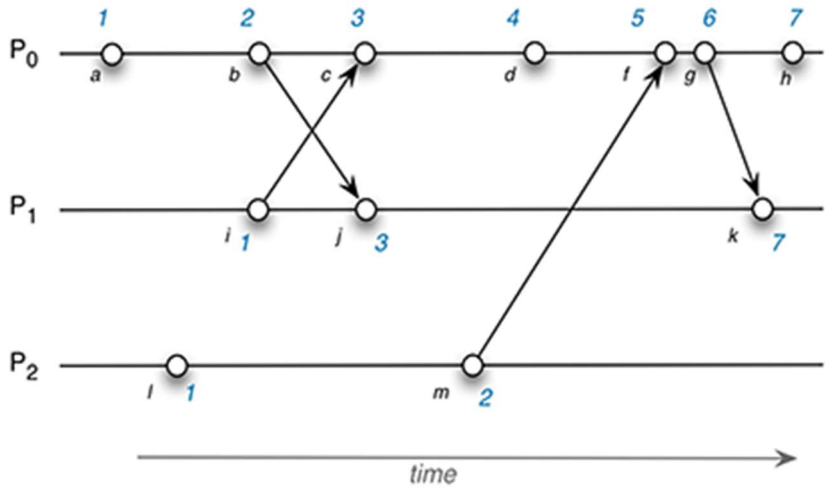
Conceptually, this logical clock can be thought of as a clock that only has meaning in relation to messages moving between processes. When a process receives a message, it resynchronizes its logical clock with that sender.

Each process maintains a single Lamport timestamp counter. Each event on the process is tagged with a timestamp from this counter. The counter is incremented before the event timestamp is assigned. If a process has four events, *a, b, c, d*, they would get Lamport timestamps of *1, 2, 3, 4*.

If an event is the sending of a message then the timestamp is sent along with the message. If an event is the receipt of a message then the the algorithm instructs you to compare the current value of the process' timestamp counter (which was just incremented before this event) with the timestamp in the received message. If the timestamp of the received message is greater than that of the current system, the system timestamp is updated with that of the timestamp in the received message plus one. This ensures that the timestamp of the received event and all further timestamps will be greater than that of the timestamp of sending the message as well as all previous messages.

In the figure below, event *k* in process P_1 is the receipt of the message sent by event *g* in P_0. If event *k* was just a normal local event, the P_1 would assign it a timestamp of 4. However, since the received timestamp is 6, which is greater than 4, the timestamp counter is set to 6+1, or 7. Event *k* gets the timestamp of 7. A local event after *k* would get a timestamp of 8

With Lamport timestamps, we're assured that two causally-related events will have timestamps that reflect the order of events. For example, event *c* happened before event *k* in the Lamport causal sense: the chain of causal events is $c \rightarrow d$, $d \rightarrow f$, $f \rightarrow g$, and $g \rightarrow k$. Since the *happened-before* relationship is transitive, we know that $c \rightarrow k$ (*c* happened before *k*). The Lamport timestamps reflect this. The timestamp for *c* (3) is less than the timestamp for *k*(7). However, just by looking at timestamps we cannot conclude that there is a causal happened-before relation. For instance, because the timestamp for *l* (1) is less than the timestamp for *j* (3) does not mean that *l* happened before *j*. Those events happen to be concurrent but we cannot discern that by looking at Lamport timestamps. We need need to employ a different technique to be able to make that determination. That technique is the use of*vector timestamps*.



Lamport Clock Assignments

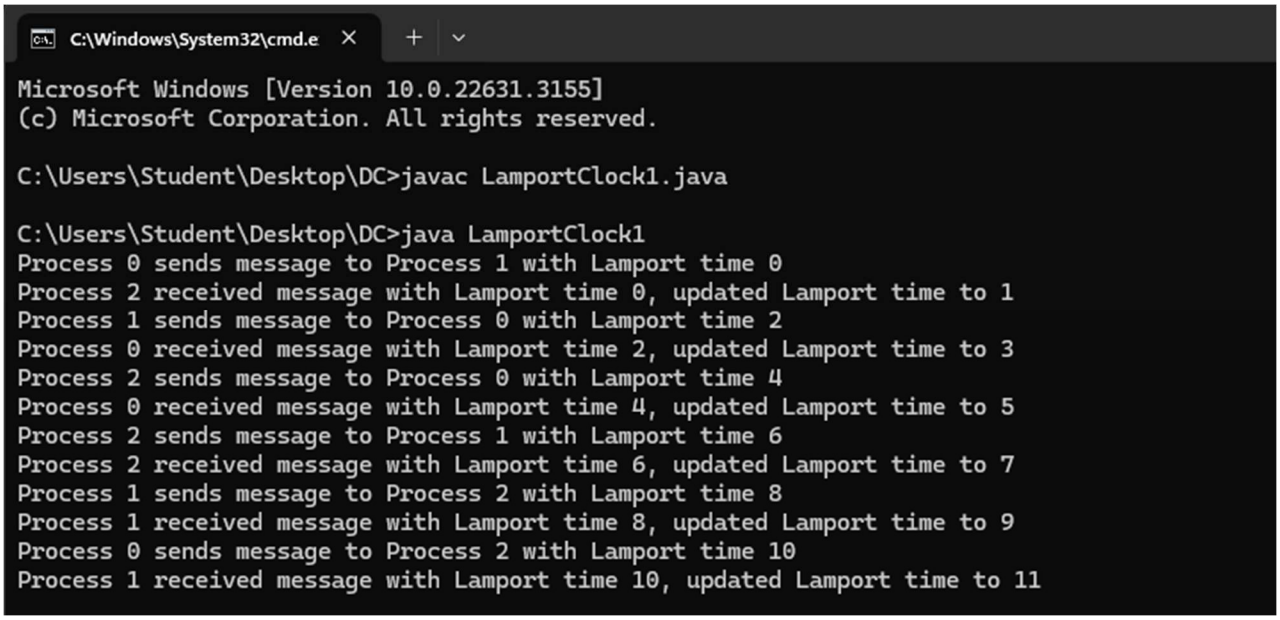
Code :

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

class LamportClock {
   private int time = 0;

   public synchronized int getTime() {
      return time;
```

```
    }

    public synchronized void setTime(int time) {
        if (time > this.time) {
            this.time = time;
        }
    }

    public synchronized void tick() {
        time++;
    }
}

class Process extends Thread {
    private int id;
    private LamportClock lamportClock;
    private List<Process> processes;

    public Process(int id, LamportClock lamportClock, List<Process> processes) {
        this.id = id;
        this.lamportClock = lamportClock;
        this.processes = processes;
    }

    public void sendMessage(int receiverId) {
        Process receiver = processes.get(receiverId);
        synchronized (lamportClock) {
            int sendTime = lamportClock.getTime();
            System.out.println("Process " + id + " sends message to Process " + receiverId + "
with Lamport time " + sendTime);
            receiver.receiveMessage(sendTime);
            lamportClock.tick();
        }
    }

    public void receiveMessage(int sendTime) {
        synchronized (lamportClock) {
            lamportClock.setTime(Math.max(lamportClock.getTime(), sendTime) + 1);
            System.out.println("Process " + id + " received message with Lamport time " +
sendTime +
```

```
                ", updated Lamport time to " + lamportClock.getTime());
        }
    }

    @Override
    public void run() {
        for (int i = 0; i < processes.size(); i++) {
            if (i != id) {
                sendMessage(i);
                try {
                    Thread.sleep(100); // Simulating some processing time
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

public class LamportClock1 {
    public static void main(String[] args) {
        int numProcesses = 3;
        LamportClock clock = new LamportClock();
        List<Process> processes = new ArrayList<>();

        for (int i = 0; i < numProcesses; i++) {
            processes.add(new Process(i, clock, processes));
        }

        Collections.shuffle(processes); // Randomize the order of process execution

        for (Process process : processes) {
            process.start();
        }
    }
}
```

Output:



Conclusion: Implementing Clock Synchronization algorithms using Lamport's logical clocks is fundamental for maintaining causality and consistency in distributed systems. By tracking the ordering of events across distributed nodes, Lamport's logical clocks ensure that events occur in a globally consistent order, even in the absence of a global clock. Implementing these algorithms involves updating local clocks based on message exchanges and adjusting timestamps to reflect the causal relationship between events. However, while Lamport's logical clocks provide a simple and effective solution for clock synchronization, they may not address issues related to clock drift or network delays. Nonetheless, their implementation significantly enhances the reliability and coordination of distributed systems, fostering smoother communication and accurate event ordering.