TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

THAPATHALI CAMPUS

A Lab Report Of Lamport Clock Synchronization Using Java

Submitted By:

Anuj Rayamajhi (THA076BCT007)

Submitted To:

Department of Electronics and Computer Engineering

Thapathali Campus

Kathmandu, Nepal

July, 2023

1. Introduction

The lab work "Lamport Clock Synchronization using Java" explores the concept of Lamport's logical clocks, which are a distributed algorithm for ordering events in a distributed system. In this lab, we aim to implement Lamport's Clock Synchronization algorithm in Java to simulate how multiple processes in a distributed system can coordinate their clocks to maintain a consistent view of time. The lab will demonstrate how processes exchange events and update their local clocks based on the exchanged timestamps.

2. Theory

Lamport's Logical Clock is a distributed algorithm developed by Leslie Lamport to order events in a distributed system. In a distributed system, where multiple processes are running independently on different nodes, it is challenging to establish a global notion of time. Lamport's Clock provides a way to assign logical timestamps to events across processes, allowing for a partial ordering of events in the distributed system.

The algorithm uses logical clocks represented by integers. Each process maintains its own logical clock. The logical clock value increases monotonically with time, and any two events on the same process have different logical timestamps. When an event occurs, the process increments its logical clock before processing the event.

When a process sends a message to another process, it includes its current logical clock value in the message. The receiving process updates its own logical clock to be the maximum of its current value and the received logical clock value + 1. This ensures that the receiving process's clock is always ahead of the sender's clock, and the events are correctly ordered based on the Lamport timestamps.

Example:

Consider a simple scenario with two processes, P1 and P2, in a distributed system. Let's assume that the processes are independent and do not share a global clock. The events in each process are represented as E1, E2, E3, etc.

1. Process P1:

- *E1 occurs with a logical timestamp of 1. (P1's clock = 1)*
- *E2 occurs with a logical timestamp of 3. (P1's clock = 3)*
- *E3 occurs with a logical timestamp of 4. (P1's clock = 4)*

2. Process P2:

- *E4 occurs with a logical timestamp of 2. (P2's clock = 2)*
- *E5 occurs with a logical timestamp of 5. (P2's clock = 5)*

Now, let's assume that P1 sends a message to P2 after event E3 with its current logical clock value of 4.

Message: "Hi P2, this is P1. My current clock value is 4."

P2 receives the message and updates its logical clock based on the received timestamp. The new logical clock value for P2 will be *max (5, 4+1) = 5*.

Now, event E6 occurs on P2 with a logical timestamp of 6. (P2's clock = 6)

Based on the logical timestamps, we can see that E4 (on P2) comes before E1, E2, and E3 (on P1), and E5 (on P2) comes before E6 (on P2). However, we cannot determine the relationship between E1, E2, E3 (on P1) and E4, E5, E6 (on P2) since they are not related by direct message exchange. The Lamport Logical Clock algorithm provides a partial ordering of events that can be used to reason about causality in a distributed system without requiring a global clock.

3. Code

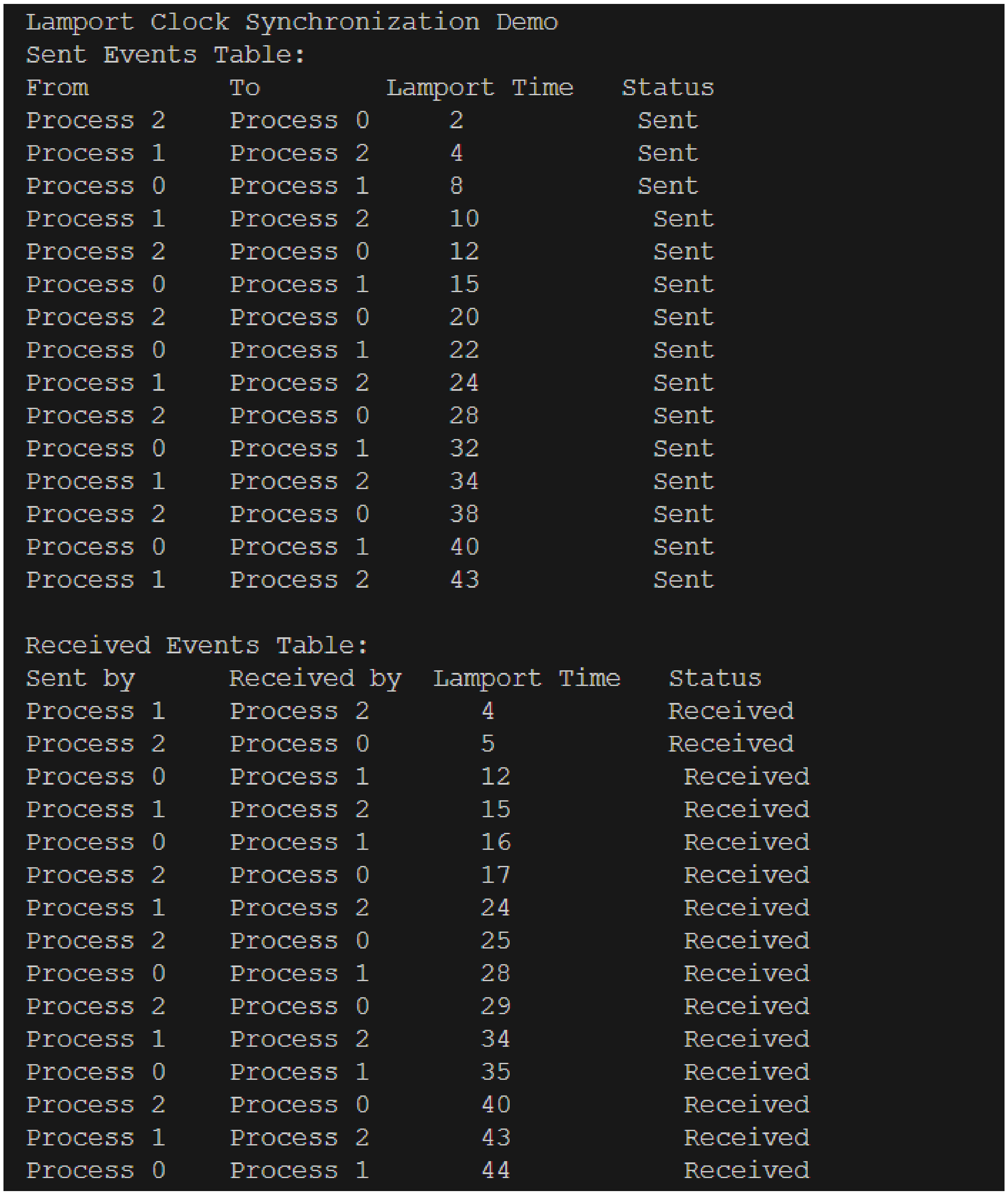
The Java implementation of Lamport's Clock Synchronization consists of several classes. The **“LamportClock”** class represents the logical clock and provides methods to increment the clock value, get the current clock value, and update the clock based on received timestamps. The Event class is a simple data structure to hold information about events exchanged between processes, including the sender, receiver, and the corresponding Lamport timestamp.

The **“LamportProcess”** class simulates a process in the distributed system. Each process runs in a separate thread and executes a loop where it sends events to other processes and receives events from them. Upon sending an event, it updates its own logical clock and sends the current timestamp along with the event. Upon receiving an event, it updates its logical clock based on the received timestamp. The main class **“LamportClockSyncDemo”** initializes the processes, starts their threads, and waits for all threads to finish. It then prints the results in tabular form, showing the sent and received events along with their respective Lamport timestamps.

Code Below:

```java
import java.util.concurrent.atomic.AtomicInteger;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;


class LamportClock {
    private AtomicInteger clock = new AtomicInteger(0);

    public void sendEvent() {
        // Increment the clock value before sending an event
        clock.incrementAndGet();
    }

    public int getClockValue() {
        return clock.get();
    }

    public void updateClock(int receivedClockValue) {
        // Update the clock value by taking the maximum of received value
and current value + 1
        clock.set(Math.max(receivedClockValue, clock.get()) + 1);
    }
}

class Event {
    private int from;
    private int to;
    private int lamportTime;
    public Event(int from, int to, int lamportTime) {
        this.from = from;
        this.to = to;
        this.lamportTime = lamportTime;
    }

    public int getFrom() {
        return from;
    }

    public int getTo() {
        return to;
    }

    public int getLamportTime() {
        return lamportTime;
    }
}

class LamportProcess implements Runnable {
    private int processId;
    private LamportClock clock;
    private LamportProcess[] processes;
    private List<Event> sentEvents = new ArrayList<>();
    private List<Event> receivedEvents = new ArrayList<>();

    public LamportProcess(int processId, LamportClock clock,
LamportProcess[] processes) {
        this.processId = processId;
        this.clock = clock;
        this.processes = processes;
    }

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) { // Assuming 5 events for
demonstration purposes
            // Simulate some local computation before sending an event
            try {
                Thread.sleep((int) (Math.random() * 1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            // Send event
            clock.sendEvent();
            int receiverId = (processId + 1) % processes.length;
            processes[receiverId].receiveEvent(clock.getClockValue());
            sentEvents.add(new Event(processId, receiverId,
clock.getClockValue()));

            // Simulate some local computation before receiving an event
            try {
                Thread.sleep((int) (Math.random() * 1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            // Receive event
            int receivedClockValue = clock.getClockValue(); // Simulate
receiving the timestamp from another process
            clock.updateClock(receivedClockValue);
            receivedEvents.add(new Event(processId,receiverId,
receivedClockValue));
        }
    }

    public void receiveEvent(int receivedClockValue) {
        clock.updateClock(receivedClockValue);
    }

    public List<Event> getSentEvents() {
        return sentEvents;
    }

    public List<Event> getReceivedEvents() {
        return receivedEvents;
    }
}

public class LamportClockSyncDemo {
    public static void main(String[] args) {
        LamportClock clock = new LamportClock();
        int numProcesses = 3; // You can adjust the number of processes
for your demo

        LamportProcess[] processes = new LamportProcess[numProcesses];
        Thread[] processThreads = new Thread[numProcesses];
        System.out.println("Lamport Clock Synchronization Demo");

        // Initialize and start the processes
        for (int i = 0; i < numProcesses; i++) {
            processes[i] = new LamportProcess(i, clock, processes);
            processThreads[i] = new Thread(processes[i]);
            processThreads[i].start();
        }

        // Wait for all processes to finish
        for (int i = 0; i < numProcesses; i++) {
            try {
                processThreads[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        System.out.println("Sent Events Table:");
        System.out.println("From         To        Lamport
Time   Status");
        List<Event> allSentEvents = new ArrayList<>();
        for (LamportProcess process : processes) {
            allSentEvents.addAll(process.getSentEvents());
        }
        Collections.sort(allSentEvents,
Comparator.comparingInt(Event::getLamportTime));
        for (Event event : allSentEvents) {
            System.out.printf("Process %d    Process
%d     %d           Sent%n", event.getFrom(), event.getTo(),
event.getLamportTime());
        }

        System.out.println("\nReceived Events Table:");
        System.out.println("Sent by      Received by  Lamport
Time   Status");
        List<Event> allReceivedEvents = new ArrayList<>();
        for (LamportProcess process : processes) {
            allReceivedEvents.addAll(process.getReceivedEvents());
        }
        Collections.sort(allReceivedEvents,
Comparator.comparingInt(Event::getLamportTime));
        for (Event event : allReceivedEvents) {
            System.out.printf("Process %d    Process
%d       %d           Received%n", event.getFrom(), event.getTo(),
event.getLamportTime());
        }
    }
}
```

4. Output



```
Lamport Clock Synchronization Demo
Sent Events Table:
From         To        Lamport Time   Status
Process 2    Process 0     2           Sent
Process 1    Process 2     4           Sent
Process 0    Process 1     8           Sent
Process 1    Process 2     10           Sent
Process 2    Process 0     12           Sent
Process 0    Process 1     15           Sent
Process 2    Process 0     20           Sent
Process 0    Process 1     22           Sent
Process 1    Process 2     24           Sent
Process 2    Process 0     28           Sent
Process 0    Process 1     32           Sent
Process 1    Process 2     34           Sent
Process 2    Process 0     38           Sent
Process 0    Process 1     40           Sent
Process 1    Process 2     43           Sent

Received Events Table:
Sent by      Received by  Lamport Time   Status
Process 1    Process 2       4           Received
Process 2    Process 0       5           Received
Process 0    Process 1       12           Received
Process 1    Process 2       15           Received
Process 0    Process 1       16           Received
Process 2    Process 0       17           Received
Process 1    Process 2       24           Received
Process 2    Process 0       25           Received
Process 0    Process 1       28           Received
Process 2    Process 0       29           Received
Process 1    Process 2       34           Received
Process 0    Process 1       35           Received
Process 2    Process 0       40           Received
Process 1    Process 2       43           Received
Process 0    Process 1       44           Received
```

5. Conclusion

Lamport's Clock Synchronization algorithm is a fundamental concept in distributed systems to achieve logical ordering of events across multiple processes. In this lab, we successfully implemented Lamport's Clock Synchronization in Java, demonstrating how processes interact and coordinate their logical clocks to ensure event ordering consistency. By exchanging timestamps and updating their clocks accordingly, processes achieve synchronization, despite the lack of a global clock. Understanding and implementing such clock synchronization algorithms are crucial for building robust and coordinated distributed systems.