TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

THAPATHALI CAMPUS

A Lab Report of Distributed Systems on Implementation of Lamport Clock

Submitted by:

Dipesh Bartaula

[THA076BCT015]

Submitted to:

Department of Electronics and Computer Engineering

Thapathali Campus

Kathmandu, Nepal

July 28, 2023

TITLE: TO IMPLEMENT LAMPORT CLOCK

THEORY:

Lamport clock, named after its creator Leslie Lamport, is a simple and widely used algorithm for ordering events in a distributed system. It allows multiple processes running on different machines to agree on the order of events, even though they may not share a global clock or have synchronized clocks.

In a distributed system, multiple processes or nodes may execute concurrently and interact with each other by exchanging messages. Lamport's logical clock algorithm assigns a logical timestamp to each event in the system to order them. These timestamps are not real-time values but are used to maintain a consistent order of events regardless of physical time differences between different processes.

To synchronize logical clocks, Lamport defined a relation called happens-before. The expression A→B is read "A happens before B" and means that all processes agree that first event 'A' occurs, then afterward, event 'B' occurs. The happens-before relation can be observed directly in two situations:

i. If A and B are events in the same process, and A occurs before B, then A→B is true.
ii. If A is the event of a message being sent by one process, and B is the event of the message being received by another process, then A→B is also true. A message cannot be received before it is sent, or even at the same time it is sent, since it takes a finite, nonzero amount of time to arrive.

Each process P_i keeps its own logical clock L_i which is used to apply Lamport timestamp to events.

The ordering of events depends on two situations:

i. If two events within the same process occurred, they occurred in the order in which that process was observed.
ii. Whenever a message is sent between processes, the event of sending a message occurs before the event of receiving the message.

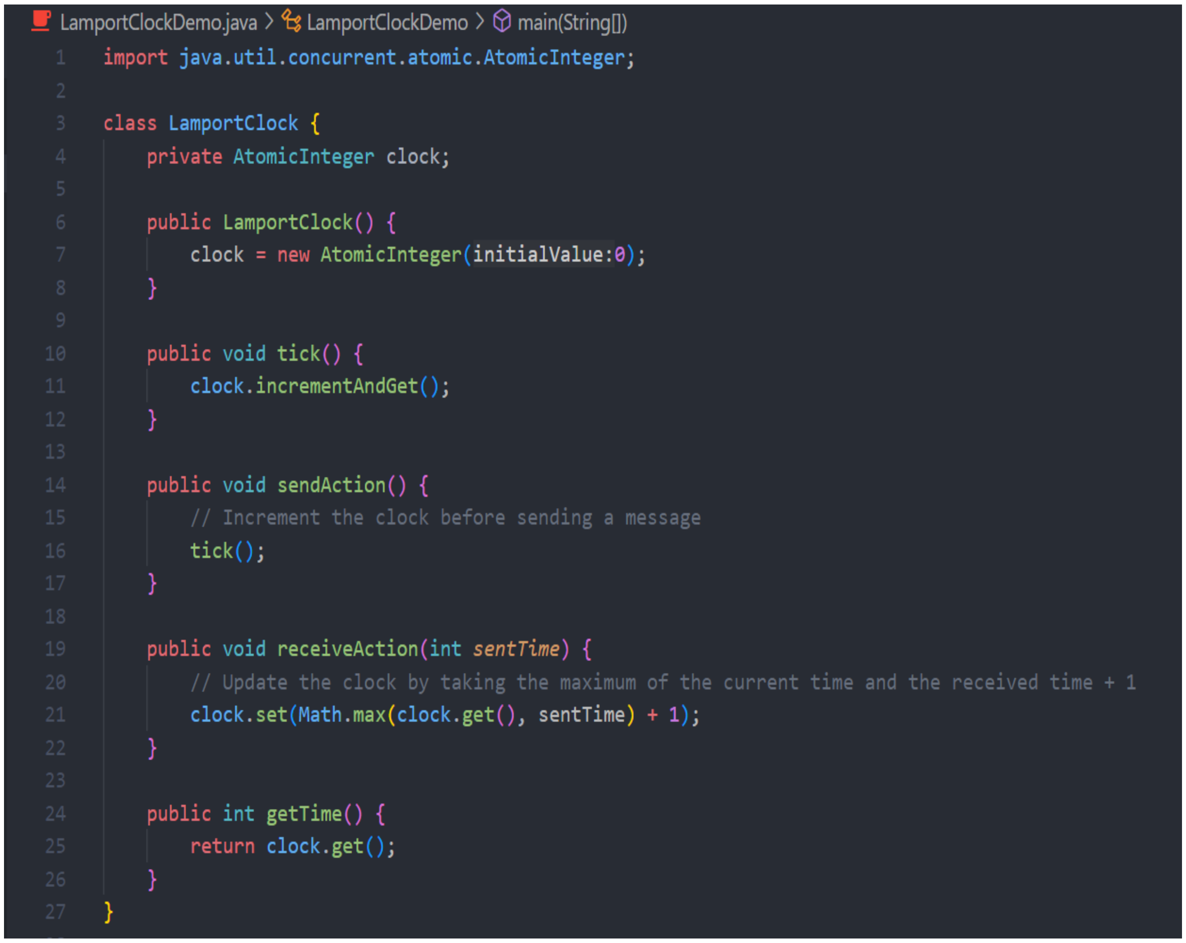
The algorithm is particularly useful in distributed systems, where processes interact with each other, and there is no central authority to provide a global time reference.

ALGORITHM:

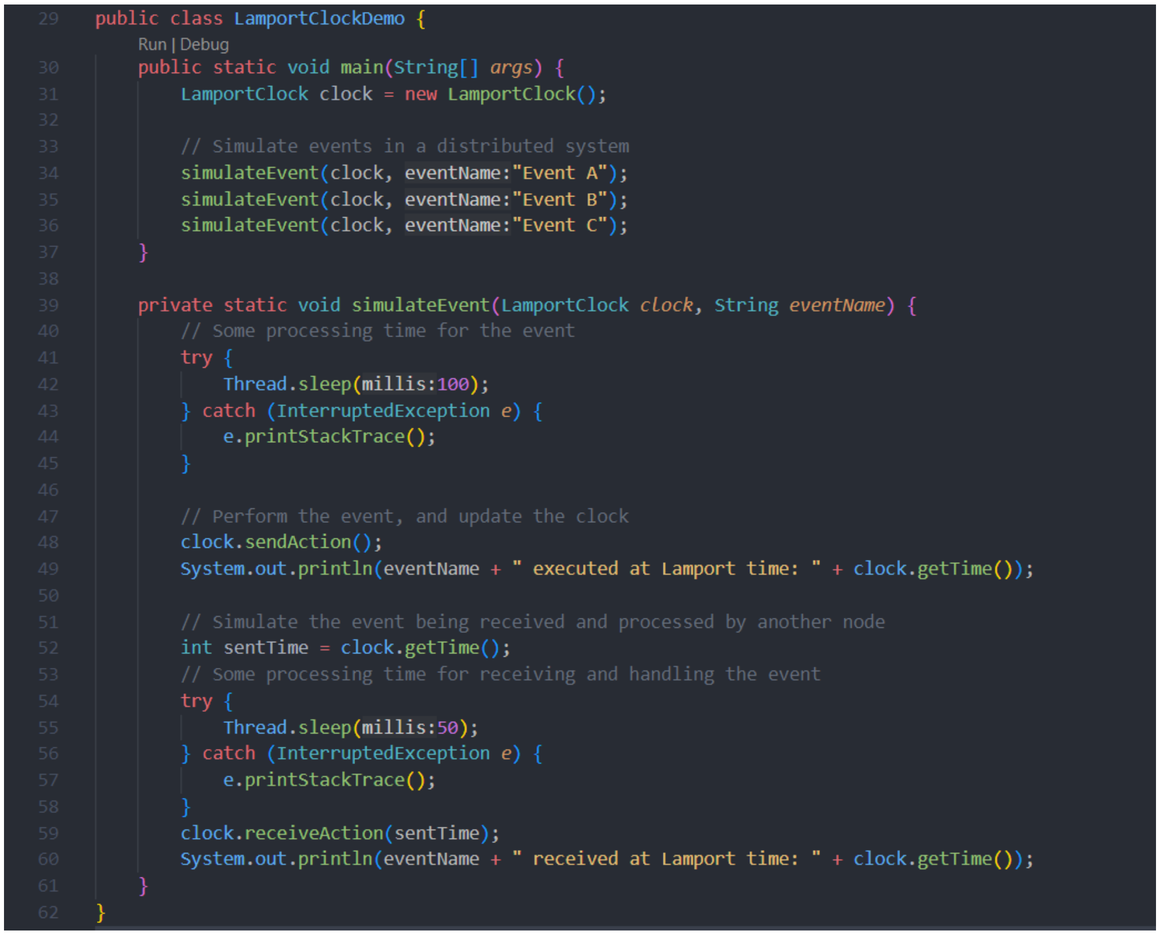
- Event Timestamping: Each time a process performs an internal event (e.g., a computation step), or sends or receives a message, it increments its logical clock by one to timestamp the event.

- Message Timestamping: When a process sends a message to another process, it includes its current logical clock value in the message.
- Message Reception: When a process receives a message, it updates its logical clock as follows: It takes the maximum of its current logical clock value and the timestamp received in the message, and then increments it by one.
- Event Ordering: The Lamport timestamps enable the processes to determine the relative ordering of events. If event A happened before event B (A→B), then the Lamport timestamp of A will be less than the Lamport timestamp of B.
- Ties: If two events have the same Lamport timestamp, an arbitrary tie-breaking rule can be used to resolve the ordering between them.

CODE:



```java
import java.util.concurrent.atomic.AtomicInteger;

class LamportClock {
    private AtomicInteger clock;

    public LamportClock() {
        clock = new AtomicInteger(initialValue:0);
    }

    public void tick() {
        clock.incrementAndGet();
    }

    public void sendAction() {
        // Increment the clock before sending a message
        tick();
    }

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

    public int getTime() {
        return clock.get();
    }
}
```

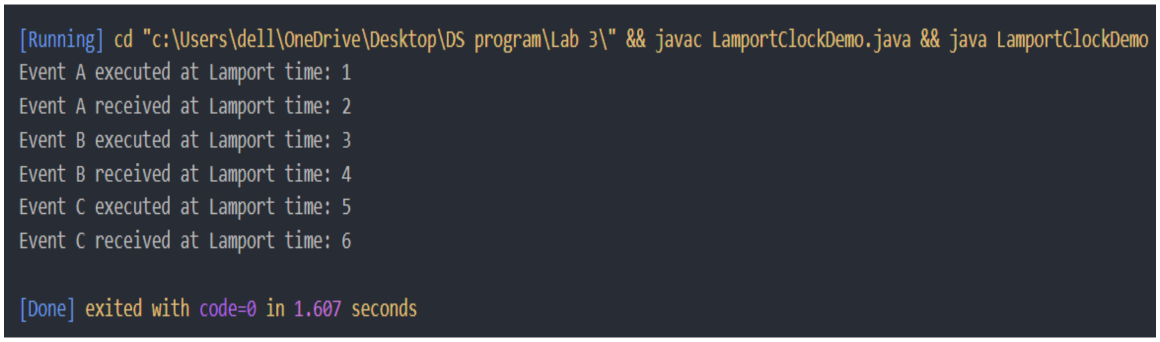


```java
public class LamportClockDemo {
    Run | Debug
    public static void main(String[] args) {
        LamportClock clock = new LamportClock();

        // Simulate events in a distributed system
        simulateEvent(clock, eventName:"Event A");
        simulateEvent(clock, eventName:"Event B");
        simulateEvent(clock, eventName:"Event C");
    }

    private static void simulateEvent(LamportClock clock, String eventName) {
        // Some processing time for the event
        try {
            Thread.sleep(millis:100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Perform the event, and update the clock
        clock.sendAction();
        System.out.println(eventName + " executed at Lamport time: " + clock.getTime());

        // Simulate the event being received and processed by another node
        int sentTime = clock.getTime();
        // Some processing time for receiving and handling the event
        try {
            Thread.sleep(millis:50);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        clock.receiveAction(sentTime);
        System.out.println(eventName + " received at Lamport time: " + clock.getTime());
    }
}
```

In this implementation, the `LamportClock` class has methods for tick, `sendAction`, and `receiveAction`, which simulate events in the distributed system. The `getTime` method returns the current logical time of the Lamport clock.

The `LamportClockDemo` class demonstrates the usage of the Lamport clock. It creates a `LamportClock` object and simulates three events (A, B, and C) in a distributed system. The events are processed, and the Lamport time is printed at the execution and receiving of each event.

OUTPUT:



```
[Running] cd "c:\Users\dell\OneDrive\Desktop\DS program\Lab 3\" && javac LamportClockDemo.java && java LamportClockDemo
Event A executed at Lamport time: 1
Event A received at Lamport time: 2
Event B executed at Lamport time: 3
Event B received at Lamport time: 4
Event C executed at Lamport time: 5
Event C received at Lamport time: 6

[Done] exited with code=0 in 1.607 seconds
```

In the given output, three events (A, B, and C) were executed and processed in a distributed system using Lamport's logical clock. Each event was simulated to have some processing time before being executed and received.

The output shows the Lamport time (logical time) at which each event was executed and when it was received by another node:

- Event A was executed at Lamport time 1 and received at Lamport time 2.
- Event B was executed at Lamport time 3 and received at Lamport time 4.
- Event C was executed at Lamport time 5 and received at Lamport time 6.

Lamport's clock ensures that events are correctly ordered based on their Lamport timestamps, which helps establish a consistent global ordering of events in a distributed system.

DISCUSSION:

The provided program implements Lamport's logical clock, a fundamental algorithm used in distributed systems to order events. In this implementation, Lamport's clock is represented by a simple `LamportClock` class with methods for tick, `sendAction`, `receiveAction`, and `getTime`. Additionally, a `LamportClockDemo` class demonstrates the usage of the Lamport clock by simulating three events (A, B, and C) in a distributed system.

Lamport's logical clock works based on the idea of causality: if an event A causally influences another event B, then the logical time of A must be less than the logical time of B. The program correctly captures this causality by incrementing the clock before sending a message and updating the clock when receiving a message based on the maximum of the current time and the received time + 1.

CONCLUSION:

In conclusion, the provided Java program demonstrates the implementation of Lamport's logical clock, a fundamental algorithm used in distributed systems for event ordering. By incrementing the clock before sending messages and updating it upon receiving messages, Lamport's clock ensures the correct partial ordering of events based on causality. The simulated output showcases the successful event ordering of A, B, and C. However, it's important to note that this implementation is a simplified example and does not consider real-world complexities in distributed systems. Nonetheless, Lamport's logical clock serves as a valuable tool for maintaining event causality in distributed environments.