

**Vidyavardhini’s College of Engineering & Technology**

Department of Computer Engineering Academic Year : 2024-25

|  |  |  |  |
| --- | --- | --- | --- |
| **Class:** | **BE** | **Semester:** | **VIII** |
| **Course Code:** | **CSL801** | **Course Name:** | **Distributed Computing Lab** |

|  |  |
| --- | --- |
| **Name of Student:** | **Pratima Dinkar Bombe** |
| **Roll No. :** | **07** |
| **Division:** | **-** |
| **Experiment No.:** | **04** |
| **Title of Experiment:** | **Clock Synchronization Algorithm (Lamport’s Algorithm)** |
| **Date of Submission:** | **04/02/2025** |
| **Date of Correction:** | **11/02/2025** |

Evaluation

|  |  |  |
| --- | --- | --- |
| **Performance Indicator** | **Max. Marks** | **Marks Obtained** |
| Performance | 5 |  |
| Understanding | 5 |  |
| Journal work and timely submission | 10 |  |
| Total | 20 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Performance Indicator** | **Exceed Expectations (EE)** | **Meet Expectations (ME)** | **Below Expectations (BE)** |
| Performance | 4-5 | 2-3 | 1 |
| Understanding | 4-5 | 2-3 | 1 |
| Journal work and timely submission | 8-10 | 5-8 | 1-4 |

**Checked by**

**Name of Faculty : Ms. Swati Varma**

**Signature :**

**Date :**

EXPERIMENT 4

**Aim:** To implement Clock Synchronization algorithms

**Objective**:- Toimplement Clock Synchronization algorithms

**Theory:**

Lamport invented a simple mechanism by which the [happened-before](http://en.wikipedia.org/wiki/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:

* A process increments its counter before each event in that process;
* When a process sends a message, it includes its counter value with the message;
* 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.

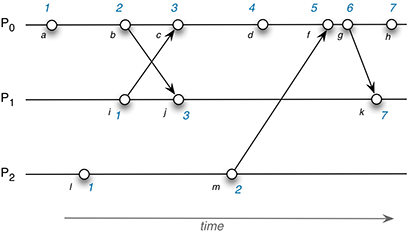
Conceptually, this logical clock can be thought of as a clock that only has meaning in relation to messages moving between processes. When a process receives a message, it resynchronizes its logical clock with that sender.

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 P1 is the receipt of the message sent by event *g* in P0. If event *k* was just a normal local event, the P1 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*→*d*, *d*→*f*, *f*→*g*, and *g*→*k*. Since the *happened-before* relationship is transitive, we know that *c*→*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 and output**:

class LamportClock:

    def \_\_init\_\_(self, process\_id):

        self.process\_id = process\_id

        self.clock = 0  # Initialize Lamport clock

    def send\_request(self):

        """Increment clock and return the timestamp with request"""

        self.clock += 1

        print(f"[Process {self.process\_id}] Sent request with timestamp {self.clock}")

        return self.clock

    def receive\_request(self, timestamp):

        """Update clock on receiving a request"""

        self.clock = max(self.clock, timestamp) + 1

        print(f"[Process {self.process\_id}] Received request (timestamp {timestamp}) → Updated clock to {self.clock}")

    def internal\_event(self):

        """Increment clock for an internal event"""

        self.clock += 1

        print(f"[Process {self.process\_id}] Internal event → Timestamp updated to {self.clock}")

# Simulating interactions between three processes

def simulate():

    process1 = LamportClock(1)

    process2 = LamportClock(2)

    process3 = LamportClock(3)

    print("\n--- Simulation Start ---\n")

    # Process 1: Internal event, then sends requests

    process1.internal\_event()

    timestamp1 = process1.send\_request()  # Sent to Process 2

    timestamp2 = process1.send\_request()  # Sent to Process 3

    # Process 2: Internal event, receives request from P1, sends request to P3

    process2.internal\_event()

    process2.receive\_request(timestamp1)

    timestamp3 = process2.send\_request()  # Sent to Process 3

    # Process 3: Internal event, receives requests from P1 & P2

    process3.internal\_event()

    process3.receive\_request(timestamp2)  # From P1

    process3.receive\_request(timestamp3)  # From P2

    # Process 1 & 2 receive responses from P3

    process1.receive\_request(process3.clock)

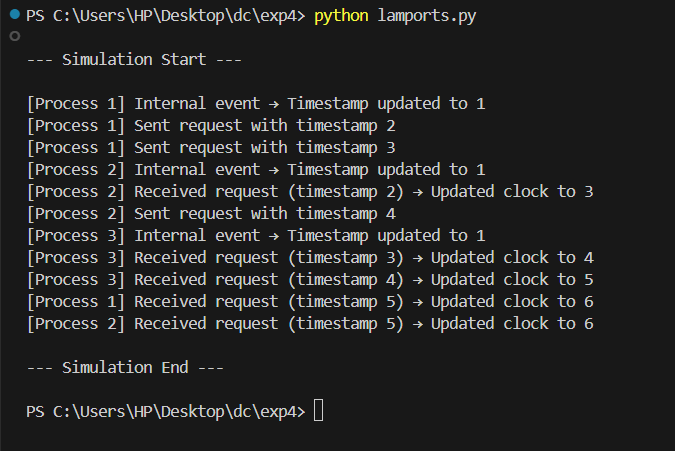
    process2.receive\_request(process3.clock)

    print("\n--- Simulation End ---\n")

# Run simulation

if \_\_name\_\_ == "\_\_main\_\_":

    simulate()



**Conclusion**: In conclusion, this implementation of Lamport's Logical Clocks effectively demonstrates how events in distributed systems can be ordered using logical timestamps. By following the incremental clock updates for internal events, message sending, and message receiving (using max(received\_timestamp, local\_clock) + 1), the simulation ensures that causality is maintained across multiple processes. This approach is crucial in distributed computing, where there is no global clock, and events must be synchronized logically to avoid inconsistencies.