

**TRIBHUVAN UNIVERSITY**

**INSTITUTE OF ENGINEERING THAPATHALI CAMPUS**

**A Lab Report**

**On**

**Lamport Clock Synchronization**

**(Distributed System Lab:3)**

**Submitted By:**

Sajjan Acharya

(THA076BCT038)

**Submitted To:**

Department of Electronics and Computer Engineering

Thapathali Campus Kathmandu, Nepal

July, 2023

# Theory

Lamport Clock Synchronization is a fundamental algorithm employed to establish a partial ordering of events within a distributed computer system. Its primary purpose is to infer the temporal relationships, or "happened-before" relationships, among events occurring across various processes, even in the absence of perfect synchronization. Proposed by the renowned computer scientist Leslie Lamport in 1978, this algorithm finds wide application in distributed systems to maintain the correct order of events.

The mechanism of Lamport timestamps involves assigning each event within a process a logical timestamp or Lamport timestamp. Unlike physical time, these timestamps serve the sole purpose of event ordering. The algorithm is built on three essential rules:

1. Local Logical Clocks: Each process maintains a local logical clock, which is essentially a software counter. This counter is incremented before any event takes place within that process.
2. Timestamps in Messages: When a process sends a message to another process, it attaches its current logical clock value (timestamp) to the message.
3. Receiving and Updating Clocks: Upon receiving a message, the recipient process updates its local logical clock to the greater of its current counter and the timestamp received in the message. Subsequently, the logical clock is incremented by 1 to mark the message as received.

Steps to Implement Lamport Clock Synchronization:

* Define the LamportClock Class: Create a class that represents each process and its associated logical clock. This class will include methods for incrementing the counter, sending messages (returning the current timestamp), and receiving messages (updating the logical clock).
* Initialization: Instantiate the LamportClock class for each process within the distributed system. Each instance will maintain its own local logical clock.
* Event Handling: Prior to any event occurring in a process, increment the logical clock of that process by calling the increment() method of the LamportClock class.
* Sending Messages: When a process sends a message to another process, invoke the send\_message() method to increment the logical clock and obtain the current timestamp. Include this timestamp in the message being sent.
* Receiving Messages: Upon receiving a message, extract the sender's timestamp from the message. Call the receive\_message(sender\_timestamp) method to update the local logical clock of the recipient process based on the sender's timestamp. Make sure to increment the logical clock by 1 after the message is considered received.
* Comparing Event Order: To establish the order of events, compare the timestamps of different events. If the Lamport timestamp of event A is less than the timestamp of event B, then event A occurred before event B. If the timestamps are equal, there is no causality relationship between the events.
* Leveraging Lamport Timestamps: Utilize Lamport Clock Synchronization in various distributed algorithms, such as resource synchronization, consensus protocols, and distributed mutual exclusion. These timestamps help maintain event ordering and aid in the correct execution of distributed system processes.

Implementing Lamport Clock Synchronization in practical distributed systems involves multiple processes communicating and exchanging messages. The logical clocks are continually updated based on the timestamps contained in the exchanged messages, thereby ensuring the accurate ordering of events across the entire distributed system.

# Code/Implementation

class LamportClock:

counter = 0 # Class-level counter shared across all instances

def send\_message(self):

LamportClock.counter += 1

return LamportClock.counter

@classmethod

def receive\_message(cls, sender\_timestamp):

cls.counter = max(cls.counter, sender\_timestamp) + 1

return cls.counter

# Function to simulate message exchange between processes

def simulate\_message\_exchange():

# Create two processes

process\_A = LamportClock()

process\_B = LamportClock()

# Process A sends a message to Process B

timestamp\_A = process\_A.send\_message()

print("Process A sends a message to Process B with timestamp:", timestamp\_A)

# Process B receives the message from Process A

timestamp\_B = process\_B.receive\_message(timestamp\_A)

print("Process B receives the message from Process A and updates its timestamp to:", timestamp\_B)

# Process B responds to Process A

timestamp\_B = process\_B.send\_message()

print("Process B sends a response message to Process A with timestamp:", timestamp\_B)

# Process A receives the response from Process B

timestamp\_A = process\_A.receive\_message(timestamp\_B)

print("Process A receives the response from Process B and updates its timestamp to:", timestamp\_A)

# Print the final timestamps of both processes after the message exchange

print("Final timestamp of Process A:", process\_A.counter)

print("Final timestamp of Process B:", process\_B.counter)

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

# Run the simulation of message exchange between processes

simulate\_message\_exchange()

The provided code demonstrates Lamport Clock Synchronization, a simple algorithm used in distributed systems to establish a partial ordering of events. Two processes, process\_A and process\_B, exchange messages, and each process maintains its logical clock represented by the counter variable. When a process sends a message, it increments its counter and includes the timestamp with the message. On receiving a message, a process updates its counter based on the received timestamp and increments it further. The algorithm ensures that events can be partially ordered by comparing the logical timestamps, enabling coordination and event ordering in a distributed system. The code simulates the message exchange between the processes and prints their final timestamps after the exchange to demonstrate how Lamport Clocks maintain event ordering.

# Results

Process A sends a message to Process B with timestamp: 1

Process B receives the message from Process A and updates its timestamp to: 2

Process B sends a response message to Process A with timestamp: 3

Process A receives the response from Process B and updates its timestamp to: 4

Final timestamp of Process A: 4

Final timestamp of Process B: 4

# Discussion

The logical timestamps generated by Lamport Clocks help maintain event causality, ensuring that events happening in different processes can be partially ordered based on their timestamps. This enables coordination and synchronization in distributed systems, which is crucial for applications requiring consistency and reliability. However, one potential drawback of Lamport Clocks is that they do not account for network delays or clock drifts, as they rely on logical increments instead of physical time. As a result, the partial ordering might not always reflect the actual causal relationship between events in certain scenarios. Additionally, in a large-scale distributed system, managing the synchronization of logical clocks across processes can become complex and potentially lead to issues like clock skew. Thus, while Lamport Clocks provide a basic mechanism for event ordering, more advanced clock synchronization techniques such as Vector Clocks or logical clock protocols like NTP should be considered to address these limitations and ensure accurate event ordering in real-world distributed systems.