TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING THAPATHALI CAMPUS

A Lab Report

On

Lamport Clock Synchronization

(Distributed System Lab:3)

Submitted By:

Raj Kumar Dhakal

(THA076BCT033)

Submitted To:

Department of Electronics and Computer Engineering

Thapathali Campus Kathmandu, Nepal

July, 2023

Theory

Lamport Clock Synchronization is a crucial algorithm used to establish a partial order of events in distributed computer systems. Its main goal is to determine the temporal relationships, known as "happened-before" relationships, among events occurring across different processes, even when perfect synchronization is not possible. This algorithm was introduced by the renowned computer scientist Leslie Lamport in 1978 and has widespread applications in distributed systems to ensure the correct sequencing of events. The mechanism of Lamport timestamps involves assigning a logical timestamp or Lamport timestamp to each event within a process. These timestamps are used solely for the purpose of ordering events and are not tied to physical time. The Lamport Clock Synchronization algorithm relies on three fundamental rules;

1. Local Logical Clocks: Each process maintains its own local logical clock, which acts as a software counter. Before any event occurs within that process, its logical clock is incremented.

2. Timestamps in Messages: When a process sends a message to another process, it includes its current logical clock value (timestamp) along with the message.

3. Receiving and Updating Clocks: Upon receiving a message, the recipient process updates its local logical clock by taking the maximum value between its current counter and the timestamp received in the message. The logical clock is then incremented by 1 to indicate the receipt of the message.

Steps to implement Lamport Clock Synchronization in a distributed system:

1. Create a LamportClock class: Develop a class that represents each process and its associated logical clock. This class should have methods for incrementing the counter, sending messages (returning the current timestamp), and receiving messages (updating the logical clock).

2. Initialization: Instantiate the LamportClock class for each process in the distributed system, creating separate instances to maintain their local logical clocks.

3. Handling Events: Before any event occurs within a process, increase the logical clock of that process by calling the increment() method from the LamportClock class.

4. Sending Messages: When a process sends a message to another process, use the send_message() method to increment the logical clock and obtain the current timestamp. Include this timestamp in the message being sent.

5. 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. After considering the message received, increment the logical clock by 1.

6. 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.

7. Utilizing Lamport Timestamps: Leverage Lamport Clock Synchronization in various distributed algorithms, such as resource synchronization, consensus protocols, and distributed mutual exclusion. These timestamps help maintain event ordering and ensure the proper execution of distributed system processes.

By applying Lamport Clock Synchronization in practical distributed systems, multiple processes can communicate and exchange messages, and their logical clocks are continually updated based on the timestamps in the messages. This ensures the accurate ordering of events throughout the entire distributed system.

Code

```
class LamportClock:

    counter = 0

    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_X = LamportClock()
    process_Y = LamportClock()
    # Process X sends a message to Process Y
    timestamp_X = process_X.send_message()
    print("Process X sends a message to Process Y with timestamp:", timestamp_X)
    # Process Y receives the message from Process X
    timestamp_Y = process_Y.receive_message(timestamp_X)
print("Process Y receives the message from Process X and updates its timestamp to:", timestamp_Y)


    # Process Y responds to Process X
    timestamp_Y = process_Y.send_message()
```

```
    print("Process Y sends a response message to Process X with timestamp:", timestamp_Y)

    # Process X receives the response from Process Y

    timestamp_X = process_X.receive_message(timestamp_Y)

  print("Process X receives the response from Process Y and updates its timestamp to:",     t
        imestamp_X)

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

    print("Final timestamp of Process X:", process_X.counter)

    print("Final timestamp of Process Y:", process_Y.counter)

if __name__ == "__main__":

# Run the simulation of message exchange between processes     simulate_message_exchange()
```

Results

```
Process X sends a message to Process Y with timestamp: 1

Process Y receives the message from Process X and updates its timestamp to: 2

Process Y sends a response message to Process X with timestamp: 3

Process X receives the response from Process Y and updates its timestamp to: 4

Final timestamp of Process X: 4

Final timestamp of Process Y: 4
```

Discussion

The logical timestamps produced by Lamport Clocks serve a crucial role in maintaining the causal relationship among events in distributed systems. By using these timestamps, events

from different processes can be partially ordered, enabling coordination and synchronization, which is essential for applications needing consistency and reliability. However, one drawback of Lamport Clocks is their inability to consider network delays or clock drifts since they rely on logical increments rather than physical time. Consequently, the partial ordering might not always accurately reflect the actual causal relationship between events in certain situations. Furthermore, in large-scale distributed systems, managing logical clock synchronization across processes can become complex and lead to issues like clock skew. To address these limitations and ensure precise event ordering in real-world distributed systems, more advanced clock synchronization techniques like Vector Clocks or logical clock protocols such as NTP should be taken into account..