TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

THAPATHALI CAMPUS

A Lab Report

Of

Distributed System

On

Lamport Clock Synchronization

Submitted By:

Santosh Pandey (THA076BCT041)

Submitted To:

Department of Electronics and Computer Engineering

Thapathali Campus

Kathmandu, Nepal

28th July, 2023

Theory:

The Lamport timestamp technique is a straightforward logical clock algorithm employed in distributed computer systems for event ordering. In scenarios where nodes or processes are not perfectly synchronized, this method offers a cost-effective solution to establish partial event sequencing. Furthermore, it serves as a foundation for the more advanced vector clock method and is named after its creator, Leslie Lamport.

In distributed algorithms, event ordering plays a crucial role in tasks like resource synchronization. Let's consider a system with two processes and a disk, where processes communicate with each other and the disk to request access. The disk grants access based on the order of incoming messages. For instance, if process A requests write access to the disk and then sends a read instruction to process B, and process B receives the message and sends its own read request to the disk, the disk can determine the message arrival order by analyzing the sequence of moves from A to B while following the path of the message from sending to receiving.

A Lamport clock is a valuable tool for establishing a partial ordering of events between processes. According to the clock consistency condition, if event "a" precedes event "b" ($a \rightarrow b$), then the logical clock value of "a" is less than that of "b" ($C(a) < C(b)$).

This one-way relation provides a partial causal ordering based solely on the Lamport clock. Conversely, if the logical clock value of "a" is not less than that of "b" ($C(a) \not\prec C(b)$), then event "a" cannot have happened before event "b" ($a \not\rightarrow b$). In other words, if $C(a) \geq C(b)$, event "a" cannot be causally related to event "b."

In a distributed system, to achieve a total ordering of events using Lamport timestamps, one can introduce an arbitrary mechanism, such as the process ID, to resolve ties. However, it's essential to recognize that this total ordering is artificial and cannot reliably indicate a causal relationship between events.

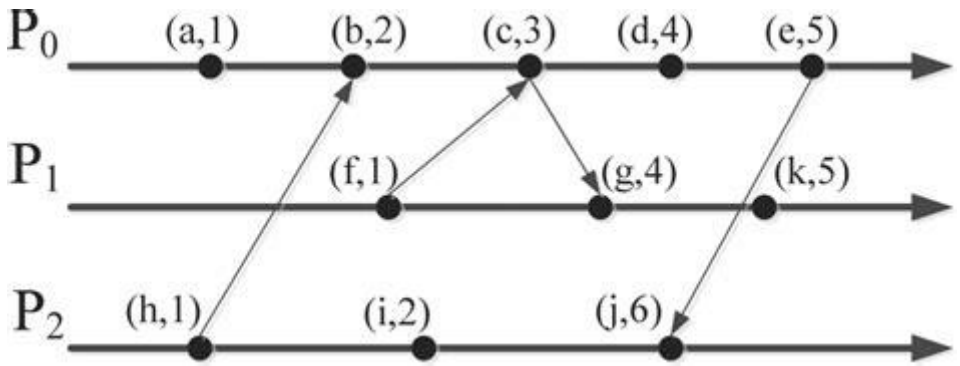

Figure 0.1Lampert's Clock Synchronization

Algorithm:

1. Prior to each local event, such as a message-sending event, the process increments its counter.
2. When a process sends a message to another entity, it includes its current counter value along with the message after completing step 1.
3. Upon receiving a message from another process, the recipient updates its own counter if required. The counter is set to the greater value between its current counter and the timestamp present in the received message. Subsequently, the counter is incremented by 1, marking the message as received

Implementation:

Lampert.py

```
# define Lampert clock
class LamportClock:
    # initialize clock
    def __init__(self):
        self.clock = 0
    # increment clock
    def increment(self):
        self.clock += 1
    # update clock
    def update(self, received_time):
        self.clock = max(self.clock, received_time) + 1
    # get current clock
    def get_timestamp(self):
        return self.clock
```

```
# define Process to use Lampert clock
class Process:
    # Initialize process with name
    def __init__(self, name):
        self.name = name
        self.clock = LamportClock()
    # Send message to another Process
    def send_message(self, receiver, message):
        print("\tBefore Synchronization:")
        print(f"\t{self.name}'s Lamport Clock: {self.clock.get_timestamp()}")
        print(f"\t{receiver.name}'s Lamport Clock:
{receiver.clock.get_timestamp()}")
        receiver.clock.update(self.clock.get_timestamp())
        self.clock.increment()
        print("\tAfter Synchronization:")
        print(f"\tMessage: {message}")
        print(f"\t{self.name}'s Lamport Clock: {self.clock.get_timestamp()}")
        print(f"\t{receiver.name}'s Lamport Clock:
{receiver.clock.get_timestamp()}\n")

if __name__ == "__main__":
    # Create two processes
    process1 = Process("Sam")
    process2 = Process("Harry")

    # Initial clock values before the exchange
    print("Initial Lamport Clock values")
    print(f"Timestamp of Process 1: {process1.clock.get_timestamp()}")
    print(f"Timestamp of Process 2: {process2.clock.get_timestamp()}\n")

    # Simulate some message exchanges
    process1.send_message(process2, "Hello from Sam to Harry")
    process2.send_message(process1, "Hello from Harry to Sam")
    process2.clock.increment()
    process2.clock.increment()
    process1.send_message(process2, "How are you?")
    process2.send_message(process1, "I'm doing fine, thanks!")

    # Final clock values after the exchanges
    print("Final Lamport Clock Values:")
    print(f"Timestamp of Process 1: {process1.clock.get_timestamp()}")
    print(f"Timestamp of Process 2: {process2.clock.get_timestamp()}")
```

Output:

```
Lab3>python Lampert.py
Initial Lamport Clock values
Timestamp of Process 1: 0
Timestamp of Process 2: 0

        Before Synchronization:
        Sam's Lamport Clock: 0
        Harry's Lamport Clock: 0
        After Synchronization:
        Message: Hello from Sam to Harry
        Sam's Lamport Clock: 1
        Harry's Lamport Clock: 1

        Before Synchronization:
        Harry's Lamport Clock: 1
        Sam's Lamport Clock: 1
        After Synchronization:
        Message: Hello from Harry to Sam
        Harry's Lamport Clock: 2
        Sam's Lamport Clock: 2

        Before Synchronization:
        Sam's Lamport Clock: 2
        Harry's Lamport Clock: 4
        After Synchronization:
        Message: How are you?
        Sam's Lamport Clock: 3
        Harry's Lamport Clock: 5

        Before Synchronization:
        Harry's Lamport Clock: 5
        Sam's Lamport Clock: 3
        After Synchronization:
        Message: I'm doing fine, thanks!
        Harry's Lamport Clock: 6
        Sam's Lamport Clock: 6

Final Lamport Clock Values:
Timestamp of Process 1: 6
Timestamp of Process 2: 6
```

Conclusion:

In this lab report, we delved into the fundamental principles and practical implementation of Lamport clock synchronization, a widely adopted algorithm in distributed systems for establishing a partial event ordering. The simplicity and effectiveness of the Lamport clock allow processes to maintain causality and ensure consistency in distributed environments.

Throughout the lab, we introduced the concept of the Lamport clock, which serves as a logical timekeeping mechanism for individual processes. Each process increments its clock upon event occurrence. Additionally, when a process sends a message to another, it updates its clock to ensure the message's timestamp reflects the causal relationship between events.