Distributed Systems Assignment

Name-Darpan Jyoti Mahanta
Roll Number - 210710007014
Batch - 2021-2025
Branch - Computer Science and Engineering

1 Question

Implement Lamport's Logical Clock using Python Programming Language

A Lamport Logical Clock[1, 2] is a mechanism used in distributed systems to provide a partial ordering of events based on their logical relationships, rather than relying on synchronized physical clocks. It was introduced by Leslie Lamport in 1978 and is widely used to maintain consistency in distributed systems.

Algorithm:

Happened before relation ($\rightarrow$): $a \rightarrow b$, means 'a' happened before 'b'.

Logical Clock: The criteria for the logical clocks are:

C1 : $C_i(a) < C_i(b)$, where C_i is the logical clock. If 'a' happened before 'b', then the time of 'a' will be less than 'b' in a particular process.

C2 : $C_i(a) < C_j(b)$, where the clock value of $C_i(a)$ is less than $C_j(b)$.

Reference:

- Process: P_i
- Event: E_{ij}, where i is the process number and j is the j-th event in the i-th process.
- tm: vector timestamp for message m.
- C_i: vector clock associated with process P_i, the j-th element is $C_i[j]$ and contains P_i's latest value for the current time in process P_j.
- d: drift time, generally $d = 1$.

Implementation Rules (IR):

IR1 : If $a \rightarrow b$ ['a' happened before 'b' within the same process], then $C_i(b) = C_i(a) + d$.

IR2 : $C_j = \max(C_j, tm + d)$ [If there are more processes, then tm = value of $C_i(a)$, C_j = max value between C_j and $tm + d$].

2 Implementation in Python

2.1 Code :-

```
class LamportClock:
    def __init__(self):
        self.time = 0

    def increment(self):
        """Increment the logical clock."""
        self.time += 1

    def send_event(self):
        """Simulate sending a message by incrementing the clock
        and returning its value."""
        self.increment()
        return self.time

    def receive_event(self, received_time):
        """
        Update the clock on receiving a message.
        Logical clock is set to the maximum of its current value
        and the received timestamp, then incremented.
        """
        self.time = max(self.time, received_time) + 1

    def __str__(self):
        return f"Logical Clock: {self.time}"


# Simulating two processes exchanging messages
if __name__ == "__main__":
    # Initialize clocks for two processes
    process1 = LamportClock()
    process2 = LamportClock()

    # Events in process1
    print("Process 1 performs an event.")
    process1.increment()
    print(process1)

    # Process 1 sends a message to Process 2
    print("\nProcess 1 sends a message to Process 2.")
    sent_time = process1.send_event()
    print(f"Process 1 clock after sending: {process1}")

    # Process 2 receives the message
    print("\nProcess 2 receives the message from Process 1.")
    process2.receive_event(sent_time)
    print(f"Process 2 clock after receiving: {process2}")

    # Process 2 performs an event
```

```
print("\nProcess 2 performs another event.")
process2.increment()
print(process2)

# Process 2 sends a message back to Process 1
print("\nProcess 2 sends a message to Process 1.")
sent_time = process2.send_event()
print(f"Process 2 clock after sending: {process2}")

# Process 1 receives the message
print("\nProcess 1 receives the message from Process 2.")
process1.receive_event(sent_time)
print(f"Process 1 clock after receiving: {process1}")
```

2.1.1 Sample Output:

Process 1 performs an event.
Logical Clock: 1

Process 1 sends a message to Process 2.
Process 1 clock after sending: Logical Clock: 2

Process 2 receives the message from Process 1.
Process 2 clock after receiving: Logical Clock: 3

Process 2 performs another event.
Logical Clock: 4

Process 2 sends a message to Process 1.
Process 2 clock after sending: Logical Clock: 5

Process 1 receives the message from Process 2.
Process 1 clock after receiving: Logical Clock: 6

3 Conclusion

Lamport's logical clock provides a simple yet effective mechanism for ordering events in distributed systems where no global clock exists. It ensures a causal relationship by incrementing the local clock for each event and synchronizing clocks during message passing. The implementation ensures that if event A causally influences event B, then the timestamp of A will be less than B, preserving causal order.

However, Lamport clocks cannot capture concurrent events, as they assign a single value per event, even if unrelated. Advanced logical clocks like vector clocks address this limitation. Despite its simplicity, Lamport's clock is foundational for understanding and designing distributed systems where event ordering is crucial, such as in distributed databases or synchronization protocols.

4o