Distributed Systems Assignment

Submitted by

Koyal Borbora

Roll Number - 210710307056
Batch - 2021-2025
Branch - Computer Science and Engineering
Date - 19.11.2024

Table of Contents

1 Question **ii**
1.1 Literature Review ii

2 Implementation in Python **iii**
2.1 Code : [1st Approach] iii
2.1.1 Sample Output: iv
2.2 Code : [2nd Approach] v
2.2.1 Sample Output: vi

1 Question

Implement Lamport's Logical Clock using Python Programming Language

1.1 Literature Review

A Lamport Logical Clock[**?**, **?**] 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 : [1st Approach]

```
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
```

2.2 Code : [2nd Approach]

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

    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"Process {self.process_id} Clock: {self.time}"


def simulate_event(processes):
    print("\nChoose an event to simulate:")
    print("1. Internal event")
    print("2. Send message")
    print("3. Receive message")
    choice = int(input("Enter your choice (1/2/3): "))

    if choice == 1:
        # Internal event
        process_id = int(input("Enter process ID (e.g., 1, 2, etc.): "))
        processes[process_id].increment()
        print(processes[process_id])

    elif choice == 2:
        # Send message
        sender_id = int(input("Enter sender process ID: "))
        receiver_id = int(input("Enter receiver process ID: "))
        sent_time = processes[sender_id].send_event()
        print(f"Process {sender_id} after sending: {processes[sender_id]}")

        # Simulate the receiver receiving the message
        processes[receiver_id].receive_event(sent_time)
        print(f"Process {receiver_id} after receiving: {processes[receiver_id]}")
```

```
    elif choice == 3:
        # Receive message
        receiver_id = int(input("Enter receiver process ID: "))
        received_time = int(input("Enter the timestamp received: "))
        processes[receiver_id].receive_event(received_time)
        print(processes[receiver_id])

    else:
        print("Invalid choice. Try again.")


if __name__ == "__main__":
    # Initialize processes
    num_processes = int(input("Enter the number of processes: "))
    processes = {i: LamportClock(i) for i in range(1, num_processes + 1)}

    print("\nInitialized processes:")
    for process in processes.values():
        print(process)

    # Simulate events
    while True:
        simulate_event(processes)
        continue_simulation = input("\nDo you want to simulate another event?
        (yes/no): ").lower()
        if continue_simulation != "yes":
            break
```

2.2.1 Sample Output:

```
Enter the number of processes: 2

Initialized processes:
Process 1 Clock: 0
Process 2 Clock: 0

Choose an event to simulate:
1. Internal event
2. Send message
3. Receive message
Enter your choice (1/2/3): 1
Enter process ID (e.g., 1, 2, etc.): 1
Process 1 Clock: 1

Do you want to simulate another event? (yes/no): yes
```

```
Choose an event to simulate:
1. Internal event
2. Send message
3. Receive message
Enter your choice (1/2/3): 2
Enter sender process ID: 2
Enter receiver process ID: 1
Process 2 after sending: Process 2 Clock: 1
Process 1 after receiving: Process 1 Clock: 2

Do you want to simulate another event? (yes/no): yes

Choose an event to simulate:
1. Internal event
2. Send message
3. Receive message
Enter your choice (1/2/3): 3
Enter receiver process ID: 1
Enter the timestamp received: 2
Process 1 Clock: 3

Do you want to simulate another event? (yes/no): yes

Choose an event to simulate:
1. Internal event
2. Send message
3. Receive message
Enter your choice (1/2/3): 3
Enter receiver process ID: 1
Enter the timestamp received: 1
Process 1 Clock: 4

Do you want to simulate another event? (yes/no): no
```