20CYS402 – Distributed Systems & Cloud Computing

Lab 2 - Simulating Clock Synchronization

Name: Chitra Harini **Date:** 18/06/2025

Roll no: CH.EN.U4CYS22010 **Lab - 1**

Github Link: 20CYS402-Distributed-Systems-Cloud-Computing/LAB2 at main · Harini-chitra/20CYS402-Distributed-Systems-Cloud-Computing

---

Objective

- **Simulate physical clock synchronization** using the Berkeley algorithm, where a time daemon polls several nodes for their local times, computes the average, and adjusts the clocks.
- **Simulate logical clock synchronization** using Lamport's algorithm to order distributed system events using logical clocks.

Question 2.1: Physical Clock Synchronization – Berkeley Algorithm

Program Code

```
def time_to_minutes(t):
    h, m = map(int, t.split(":"))
    return h * 60 + m
def minutes_to_time(m):
    h = m // 60
    m = m % 60
    return f"{h:02d}:{m:02d}"
daemon_time = "15:00"
node_times = ["15:10", "14:50", "15:25"]
all_times_min = [time_to_minutes(daemon_time)] + [time_to_minutes(t) for t in node_times]
avg_time_min = sum(all_times_min) // len(all_times_min)
avg_time_str = minutes_to_time(avg_time_min)
print(f"After Synchronization...\n")
print(f"Time Daemon : {avg_time_str}")
for i, t in enumerate(node_times, 1):
    node_min = time_to_minutes(t)
    correction = avg_time_min - node_min
    correction_sign = "+" if correction >= 0 else ""
    print(f"Node {i}: {avg_time_str} [Correction Value: {correction_sign}{correction}]")
```

Explanation

- Each computer's time is converted to minutes for easy calculation.

- The average time is computed, then each node's clock is adjusted by the difference between its initial time and the average.

Input/Output Example

Input:

Time Daemon : 15:00

Node 1: 15:10

Node 2: 14:50

Node 3: 15:25

Output:

After Synchronization...

Time Daemon : 15:05

Node 1: 15:05 [Correction Value: -10]

Node 2: 15:05 [Correction Value: +10]

Node 3: 15:05 [Correction Value: -25]

Screenshot

Conclusion

The Berkeley algorithm effectively synchronizes clocks by averaging the times of all nodes, ensuring a consistent physical clock across distributed systems.

Question 2.2: Logical Clock Synchronization – Lamport's Algorithm

Program Code

```
class Process:
    def __init__(self, pid):
        self.pid = pid
        self.clock = 0
    def event(self, desc):
        self.clock += 1
        print(f"Process {self.pid}: '{desc}' - Clock: {self.clock}")
    def send(self, receiver):
        self.clock += 1
        print(f"Process {self.pid}: 'send to {receiver.pid}' - Clock: {self.clock}")
        receiver.receive(self.clock)
    def receive(self, sender_clock):
        self.clock = max(self.clock, sender_clock) + 1
        print(f"Process {self.pid}: 'receive' - Updated Clock: {self.clock}")
p1 = Process(1)
p2 = Process(2)
p1.event("start")
p2.event("start")
p1.send(p2)
p2.event("local work")
p2.send(p1)
p1.event("end")
```

Explanation

- Each process has a logical clock starting at 0.
- A local event or send increments the clock by 1.
- On receiving, the process sets its clock to max(current, received) and increments by 1.
- The sequence printed shows how the logical clocks maintain causal order.

Input/Output Example

Events Simulated:

- **P1: start, send to P2, receive from P2, end**
- **P2: start, receive from P1, local event, send to P1**

Sample Output:

Process 1: 'start' - Clock: 1

Process 2: 'start' - Clock: 1

Process 1: 'send to 2' - Clock: 2

Process 2: 'receive' - Updated Clock: 3

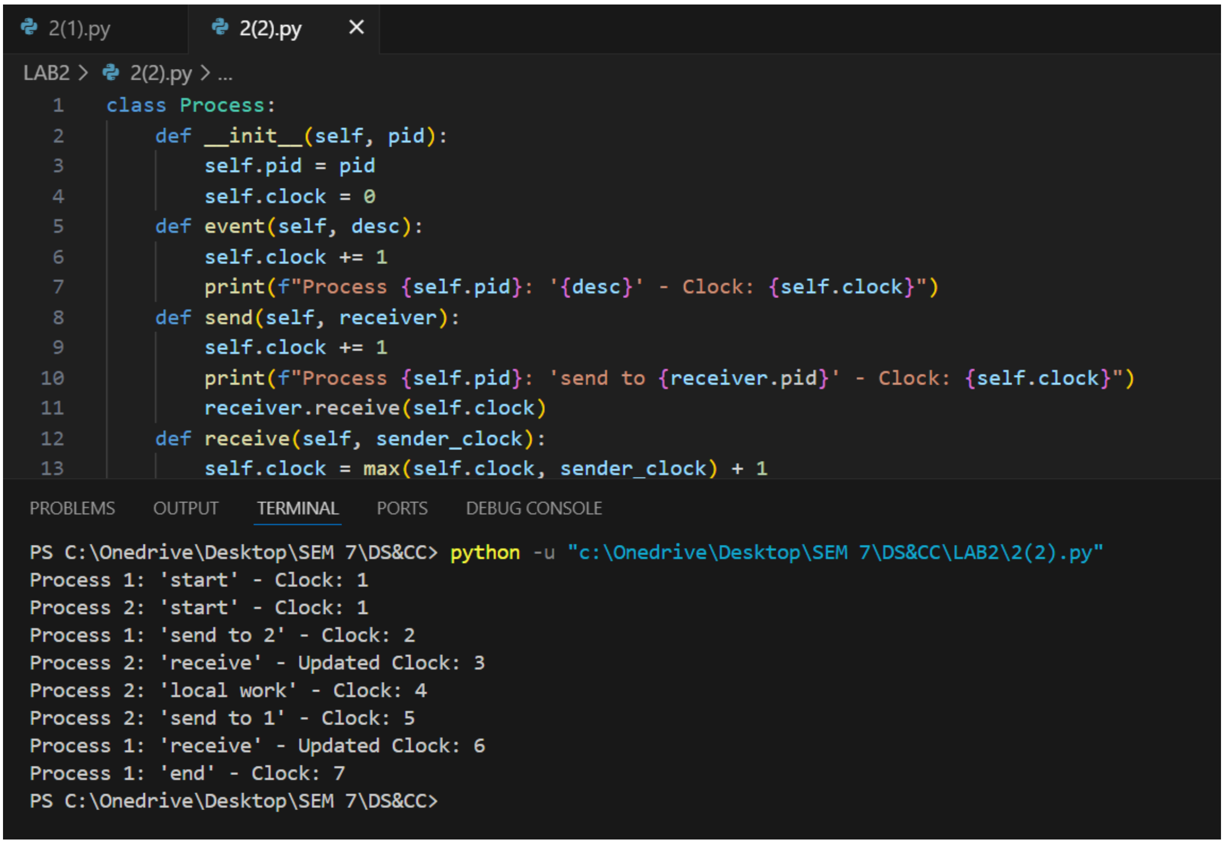
Process 2: 'local work' - Clock: 4

Process 2: 'send to 1' - Clock: 5

Process 1: 'receive' - Updated Clock: 6

Process 1: 'end' - Clock: 7

Screenshot



Conclusion

Lamport's logical clocks maintain the correct causal ordering of events in distributed processes, enabling reliable event sequencing without requiring synchronized physical time.