Department of Computer Engineering

Academic Term : January - May 2024

Class: B.E. (Computer) Semester VIII Division: B

Subject Name: Distributed Systems CSC 801

Name	**Riya Rajesh Patil**
Roll No.	**9282**
Experiment No.	**6**
Experiment Title	**To implement Mutual Exclusion/Clock synchronization algorithms.**

Aim - To implement Mutual Exclusion/Clock synchronization algorithms.

main.py

```
import threading
import time
import random

class Clock:
    def __init__(self):
        self.time = 0
        self.lock = threading.Lock()

    def get_time(self):
        with self.lock:
            return self.time

    def set_time(self, new_time):
```

```
        with self.lock:
            self.time = new_time

class Node(threading.Thread):
    def __init__(self, node_id, clock):
        super().__init__()
        self.node_id = node_id
        self.clock = clock

    def run(self):
        iterations = 10  # Set the number of iterations
        for _ in range(iterations):
            # Simulate some computation
            self.do_some_work()

            # Update the clock
            with self.clock.lock:
                current_clock = self.clock.time + 1
                self.clock.time = current_clock

            # Print the clock of this node
            print("Node", self.node_id, "- Clock:", current_clock)

            # Sleep for some time before repeating
            time.sleep(1)

    def do_some_work(self):
        # Simulate some computation
        time.sleep(random.random())
```

```
def main():
    # Create a clock instance
    clock = Clock()

    # Create and start multiple nodes
    for i in range(2):
        node = Node(i, clock)
        node.start()

if __name__ == "__main__":
    main()
```

```
(base) riyapatil@Riyas-MacBook-Air dc_6 % python main.py
Node 0 - Clock: 1
Node 1 - Clock: 2
Node 0 - Clock: 3
Node 1 - Clock: 4
Node 1 - Clock: 5
Node 0 - Clock: 6
Node 1 - Clock: 7
Node 0 - Clock: 8
Node 1 - Clock: 9
Node 0 - Clock: 10
Node 1 - Clock: 11
Node 0 - Clock: 12
Node 1 - Clock: 13
Node 0 - Clock: 14
Node 1 - Clock: 15
Node 0 - Clock: 16
Node 1 - Clock: 17
Node 1 - Clock: 18
Node 0 - Clock: 19
Node 0 - Clock: 20
(base) riyapatil@Riyas-MacBook-Air dc_6 %
```

Conclusion -

The Python code demonstrates a simplified simulation of a distributed system with two nodes. Each node maintains its local clock and periodically updates it while performing simulated computation. Threading and synchronization mechanisms ensure concurrent execution and accurate clock updates across nodes. Two nodes cannot be executed at the same time.

Postlab -

1. **Difference between logical & physical clock synchronisation.**

   1. Nature of Time:
   - Logical clocks: Measure events' ordering or causality, focusing on the sequence of events rather than real-world time.
   - Physical clocks: Measure real-world time, providing a continuous and accurate representation of time passing.

   2. Granularity:
   - Logical clocks: Typically discrete and event-driven, advancing based on events or message exchanges.
   - Physical clocks: Continuous and time-driven, advancing continuously based on the passage of time.

   3. Accuracy:
   - Logical clocks: Have no direct correlation with real-world time and prioritize consistency in event ordering over accuracy.
   - Physical clocks: Have a direct correlation with real-world time and aim to provide accurate timestamps reflecting real-world time.

   4. Dependency:
   - Logical clocks: Operate independently of physical clocks and focus on logical relationships between events.
   - Physical clocks: Depend on physical mechanisms such as oscillators and crystal oscillators to maintain synchronization with real-world time.

   5. Drift Correction:
   - Logical clocks: Typically do not include mechanisms for drift correction, as they prioritize event ordering over time accuracy.
   - Physical clocks: Often incorporate mechanisms for drift correction to account for variations in clock speeds and maintain accuracy over time.

6. Examples:
   - Logical clocks: Examples include Lamport logical clocks, Vector clocks, and other logical clock algorithms used in distributed systems.
   - Physical clocks: Examples include Network Time Protocol (NTP), Precision Time Protocol (PTP), and other protocols used to synchronize physical clocks across networks and systems.

2. What is UTC?

Coordinated Universal Time (UTC) is the primary time standard by which the world regulates clocks and timekeeping. It serves as a reference time scale used to synchronize time across various regions and nations, providing a globally consistent measure of time. UTC is based on atomic time, derived from a network of atomic clocks located around the world, ensuring high precision and accuracy. Unlike local time standards that may vary depending on geographic location or daylight saving time adjustments, UTC remains constant and does not observe any changes related to seasonal or regional factors. It is widely used in various fields, including telecommunications, aviation, computing, and scientific research, where precise and standardized timekeeping is essential. UTC is expressed as a 24-hour time format, with hours, minutes, and seconds, and is commonly denoted with a "+/-" offset from the Greenwich Mean Time (GMT) meridian, making it accessible and understandable globally. As a universally accepted time standard, UTC plays a crucial role in facilitating global communication, coordination, and synchronization across diverse time zones and systems.

3. What are the advantages & disadvantages of centralization (server)?

Advantages of Centralization (Server):

1. Resource Consolidation: Centralizing resources such as data storage, computing power, and network infrastructure on a server allows for efficient resource utilization and management. This consolidation can lead to cost savings and simplified maintenance.

2. Enhanced Security: Centralizing data and services on a server enables easier implementation of security measures such as firewalls, encryption, and access controls. It's often easier to enforce security policies and monitor activities in a centralized environment, reducing the risk of unauthorized access and data breaches.

3. Improved Data Integrity: Centralized storage of data on a server reduces the risk of data inconsistencies or duplication that can occur in distributed environments.

Centralization can ensure data integrity and consistency by providing a single source of truth for all users and applications.

4. Streamlined Maintenance: Centralized server environments simplify system maintenance tasks such as software updates, patches, and backups. Administrators can apply changes and updates centrally, reducing the effort and time required for maintenance across multiple systems.

5. Scalability and Performance: Centralized architectures can be scaled more easily by adding resources to the central server. This scalability allows organizations to adapt to changing workloads and user demands without the need for complex distributed configurations.

Disadvantages of Centralization (Server):

1. Single Point of Failure: A centralized server represents a single point of failure for the entire system. If the server experiences downtime or malfunctions, it can disrupt access to services and data for all users, leading to significant downtime and productivity losses.

2. Network Dependency: Centralized architectures rely heavily on network connectivity. If there are network issues or latency problems, it can impact the performance and availability of services accessed through the central server.

3. Bottleneck: In highly centralized environments, the server can become a bottleneck, particularly during peak usage periods. Heavy loads on the server can result in degraded performance and slow response times for users.

4. Limited Flexibility: Centralized architectures may lack the flexibility to accommodate diverse user needs and preferences. Changes or customizations to services often require updates to the central server, which can be time-consuming and complex.