Assignment No 4

Title: Implementation of Berkeley Algorithm for Clock Synchronization

Objectives:

- To understand the concept of clock synchronization in distributed systems.
- To learn the working principles of the Berkeley Algorithm.
- To implement a practical solution for synchronizing clocks across multiple nodes in a network.
- To simulate time adjustment through communication and coordination between a master and multiple slave nodes.

Problem Statement:

In a distributed computing environment, maintaining synchronized clocks among all participating systems is crucial for coordinated tasks, logging, scheduling, and consistency. Due to differences in system clocks, time drifts can occur, leading to discrepancies. The **Berkeley Algorithm** is a centralized clock synchronization technique that aims to average the time across participating nodes and adjust their clocks accordingly.

The task is to **implement the Berkeley Algorithm** where:

- One node acts as the master (time server).
- Other nodes act as slaves (clients).
- The master queries the time from each slave, calculates an average, and sends back the correction to each slave (and itself) based on the offset.
- All nodes adjust their clocks accordingly to synchronize time.

Outcomes:

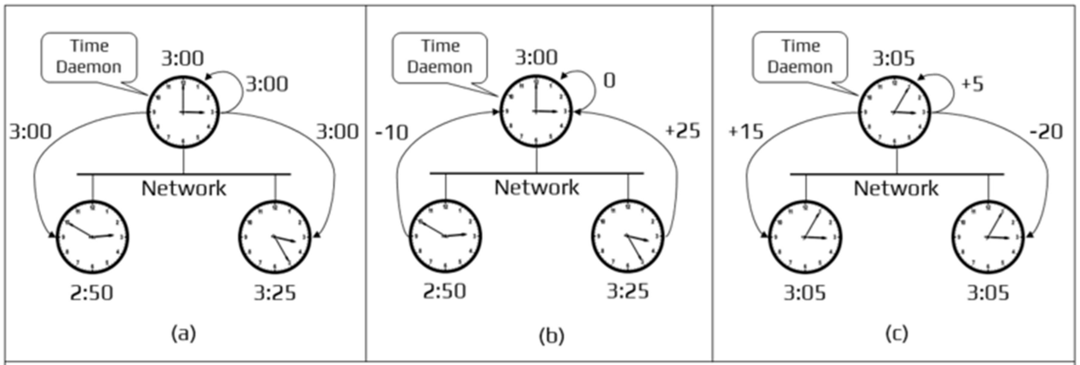
- Understanding of how clock synchronization works in distributed systems.
- Practical knowledge of implementing time-based coordination using client-server architecture.
- Experience in simulating time drift and correction via communication protocols.
- Insight into handling real-time systems where timing plays a critical role.

Theory:

The **Berkeley Algorithm**, proposed by Richard Nelson in 1989 at the University of California, Berkeley, is a **clock synchronization algorithm** used in distributed systems. Unlike algorithms such as NTP (Network Time Protocol), Berkeley Algorithm does not rely on external time sources but instead synchronizes the clocks of participating systems relative to each other.

Working of Berkeley Algorithm:

1. **Master Node Selection**: One node in the distributed system is chosen as the master (or coordinator). All others act as slaves.
2. **Polling**: The master polls each slave to request its current local time.
3. **Clock Reading and RTT Adjustment**: The master collects the time values from the slaves, noting the round-trip time (RTT) to estimate and correct for network delays.
4. **Average Time Calculation**: The master computes the average time, **excluding any faulty or significantly deviated clocks** to prevent outliers from skewing results.
5. **Offset Calculation**: The master calculates how much each node (including itself) needs to adjust its clock (i.e., offset = average time - current time).
6. **Dissemination of Offsets**: The master sends each slave its respective offset, not the actual time, allowing the slave to adjust its clock locally.
7. **Clock Adjustment**: Each node adjusts its clock based on the offset received.



(a) The time daemon asks all the other machines for their clock values.
(b) The machines answer.
(c) The time daemon tells everyone how to adjust their clock.

Key Characteristics:

- It is a **relative synchronization** technique—no node has the "true" time.
- Designed to **minimize drift** among clocks in a system.
- Works efficiently in **intranet environments** where nodes are trusted and communication delays are minimal.
- Avoids direct clock setting; instead, it sends offsets to be applied.

Advantages:

- Simple and efficient for small to medium-sized distributed systems.
- Reduces the impact of a single inaccurate clock by averaging.
- Avoids reliance on external time servers.

Disadvantages:

- Centralized approach – single point of failure (master).

- Not suitable for highly dynamic networks or systems with frequent node failure.
- Assumes relatively consistent network latency.

Software Requirements:

- **Programming Language:** Python / Java / C++
- **Libraries/Packages (for Python):**
    - `socket` – for network communication
    - `datetime` – for time operations
    - `threading` – for concurrent execution
    - `time` – for simulating delays and clock ticks
- **IDE:** VS Code, PyCharm, Eclipse, or any preferred environment
- **OS:** Windows / Linux / macOS

Hardware Requirements:

- **Processor:** Intel i3 or better
- **RAM:** 4 GB or more
- **Disk Space:** 100 MB
- **Connectivity:** Localhost or LAN (for real-time communication)
- **Devices:** Minimum 1 system (simulation) or multiple for networked testing

Conclusion: