|  |  |
| --- | --- |
|  | **DEPARTMENT OF COMPUTER ENGINEERING** |

Experiment No. 05

|  |  |
| --- | --- |
| Semester | B.E. Semester VIII – Computer Engineering |
| Subject | Distributed Computing Lab |
| Subject Professor In-charge | Dr. Umesh Kulkarni |
| Assisting Professor | Prof. Prakash Parmar |
| Academic Year | 2024-25 |

|  |  |
| --- | --- |
| Student Name | Deep Salunkhe |
| Roll Number | 21102A0014 |

**Title:** Multiple Nodes and Local Clocks in Distributed Computing.

**Explanation:  
  
1. Introduction**

In distributed computing, there is no global clock to synchronize events across multiple nodes. Instead, each node maintains its **local clock**, which may drift due to differences in processing speed and network latency. This lab demonstrates a **random message-passing system** where nodes communicate asynchronously, assigning timestamps based on their local clocks.

**2. Objective**

The goal of this experiment is to:

* Simulate multiple nodes with independent local clocks.
* Demonstrate message passing between nodes at random intervals.
* Show how timestamps are assigned and updated using **logical clocks**.
* Observe the effect of clock drift and synchronization through communication.

**3. Concepts and Theory**

**3.1 Distributed Systems and Local Clocks**

A distributed system consists of multiple independent nodes that communicate over a network. Each node maintains its **own clock**, which may not be synchronized with others. Since there is no shared global clock, timekeeping in distributed systems is challenging.

**3.2 Logical Clocks**

Logical clocks are mechanisms that help maintain event ordering in distributed systems. The **happened-before relation (→)** states:

* If an event **A** occurs before **B** in the same node, then **A → B**.
* If a node sends a message with timestamp **T**, the receiver updates its clock to **max(local\_clock, T) + 1**.

This ensures **causal consistency** among events.

**4. Code Explanation**

**4.1 Node Class**

Each node in the system:

* Maintains a localClock initialized to the system time.
* Sends messages at random intervals (1-4 seconds).
* Assigns a timestamp to each message before sending it.
* Receives messages and updates its local clock based on the received timestamp using the **logical clock update rule**.

**Clock Synchronization Rule**

When a node receives a message with timestamp **T**, it updates its clock as:

localClock=max(localClock,T)+1localClock = max(localClock, T) + 1localClock=max(localClock,T)+1

**4.2 Message Passing Mechanism**

Nodes randomly choose another node to send a message to. The receiver updates its clock based on the received timestamp. This simulates asynchronous communication in a distributed system.

**4.3 Distributed System Setup**

* A fixed number of nodes (5 in this case) are created.
* Each node runs as a separate thread, simulating concurrent execution.
* The system runs for **20 seconds**, allowing multiple message exchanges.
* The execution is managed using **Java’s ExecutorService**.

**5. Observations and Results**

After running the program, the console output shows:

1. **Message passing between nodes** with timestamps.
2. **Clock drift** due to independent updates at each node.
3. **Synchronization through message exchange**, ensuring timestamps remain consistent.

**6. Conclusion**

This experiment illustrates:

* The **importance of logical clocks** in maintaining event order in a distributed system.
* How **message passing helps synchronize** local clocks.
* The **effect of random delays** in communication.
* A **real-world simulation** of asynchronous distributed computing.

In real-world distributed systems, advanced clock synchronization methods like **NTP (Network Time Protocol)** or **Lamport Timestamps** can be used to improve accuracy.

**Code:**

**import** **java.util.Random**;

**import** **java.util.concurrent.ExecutorService**;

**import** **java.util.concurrent.Executors**;

**import** **java.util.concurrent.TimeUnit**;

**class** Node **implements** Runnable {

**private** **final** **int** id;

**private** **long** localClock;

**private** **final** **Random** random **=** **new** Random();

**public** Node(**int** id) {

**this**.id **=** id;

**this**.localClock **=** System.currentTimeMillis(); // Initialize with system time

    }

**private** **void** sendMessage(**Node** receiver) {

**long** timestamp **=** localClock;

        System.out.println("Node " **+** id **+** " sent a message to Node " **+** receiver.id **+** " at " **+** timestamp);

        receiver.receiveMessage(**this**, timestamp);

    }

**private** **void** receiveMessage(**Node** sender, **long** timestamp) {

        System.out.println("Node " **+** id **+** " received a message from Node " **+** sender.id **+** " with timestamp " **+** timestamp);

        synchronizeClock(timestamp);

    }

**private** **void** synchronizeClock(**long** receivedTimestamp) {

        localClock **=** Math.max(localClock, receivedTimestamp) **+** 1;

        System.out.println("Node " **+** id **+** " updated its local clock to " **+** localClock);

    }

    @**Override**

**public** **void** run() {

**try** {

**while** (**true**) {

                Thread.sleep(random.nextInt(3000) **+** 1000); // Random interval between 1-4 sec

                localClock **+=** random.nextInt(10); // Simulate local clock drift

**Node** receiver **=** DistributedSystem.getRandomNode(id);

**if** (receiver **!=** **null**) {

                    sendMessage(receiver);

                }

            }

        } **catch** (**InterruptedException** e) {

            Thread.currentThread().interrupt();

        }

    }

}

**public** **class** DistributedSystem {

**private** **static** **final** **int** NODE\_COUNT **=** 5;

**private** **static** **final** **Node**[] nodes **=** **new** **Node**[NODE\_COUNT];

**private** **static** **final** **Random** random **=** **new** Random();

**public** **static** **void** main(**String**[] args) {

**ExecutorService** executor **=** Executors.newFixedThreadPool(NODE\_COUNT);

**for** (**int** i **=** 0; i **<** NODE\_COUNT; i**++**) {

            nodes[i] **=** **new** Node(i);

            executor.execute(nodes[i]);

        }

**try** {

            Thread.sleep(20000); // Run for 20 seconds

        } **catch** (**InterruptedException** e) {

            Thread.currentThread().interrupt();

        }

        executor.shutdownNow();

    }

**public** **static** **Node** getRandomNode(**int** excludeId) {

**while** (**true**) {

**int** randomIndex **=** random.nextInt(NODE\_COUNT);

**if** (randomIndex **!=** excludeId) {

**return** nodes[randomIndex];

            }

        }

    }

}

**Output:**





