Cloud Computing Final Project

Name: INDROJIT DHE SHAON

Roll: 23CS06023

Program: M. Tech.

Branch: CSE

Course: Cloud Computing

Introduction:

Mutual exclusion is essential in distributed systems to avoid inconsistent or conflicting behavior caused by numerous processes or nodes accessing a common resource at the same time. This problem is solved by Lamport's Mutual Exclusion algorithm, which uses logical clocks to organize events and establish the sequence in which access requests are sent to the vital sector.

Overview:

The project intends to use three devices in a distributed system to achieve Lamport's Mutual Exclusion algorithm utilizing logical clocks. As a node in the system, every device will interact with other nodes using socket communication. Access to a file on any one of the three computers will be considered the crucial component.

Each device will have two threads in the implementation: one for transmitting event messages (particularly, requests for mutual exclusion) and producing local events, and another for listening to incoming messages. Correctly updating the logical clocks involves synchronizing the transmitter and recipient threads.

In order to do this, the project will -

1. To allow message passing, set up socket communication between the nodes.
2. To timestamp local events and inbound messages, implement logical clocks at each node.
3. To enforce mutual exclusion, limit the number of nodes that may access the vital area at once by using Lamport's method.
4. In order to precisely update the logical clocks and preserve the sequence of events, synchronize the sender and recipient threads.
5. To achieve mutual exclusion, employ file locking methods and simulate access to a file in the vital part.

A thorough description of the implementation, including the design decisions made, the difficulties encountered, and the synchronization

strategies employed to manage the sender and recipient threads, will also be included in the project.

Coding Analysis:

Full Code

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>
#include <vector>

// Mock function to simulate sending a message over sockets
void sendMessage(int recipient, int timestamp) {
    std::cout << "Sending message to device " << recipient << " with timestamp " << timestamp << std::endl;
    // Code for sending message over sockets
}

// Function for sender thread
```

```
void senderThread(int deviceID, std::vector<std::mutex>& clocks,
std::mutex& sendMutex) {
    int logicalClock = 0; // Logical clock for the current device


    while (true) {
        // Simulate local event
        std::this_thread::sleep_for(std::chrono::milliseconds(1000));
        logicalClock++;


        // Update the logical clock for the current device
        std::lock_guard<std::mutex> clockLock(clocks[deviceID]);
        logicalClock = std::max(logicalClock,
static_cast<int>(clocks[deviceID].locked()));
        clocks[deviceID].unlock();


        // Send mutual exclusion request to other devices
        sendMutex.lock();
        for (int i = 0; i < clocks.size(); ++i) {
            if (i != deviceID) {
                sendMessage(i, logicalClock);
            }
        }
        sendMutex.unlock();
```

```
    }
}

// Function for receiver thread
void receiverThread(int deviceID, std::vector<std::mutex>& clocks) {
    while (true) {
        // Receive message from other devices (mocked)
        int sender;
        int timestamp; // Received timestamp
        std::cin >> sender >> timestamp;

        // Update the logical clock for the current device
        std::lock_guard<std::mutex> clockLock(clocks[deviceID]);
        clocks[deviceID].lock();
        clocks[deviceID].unlock();

        // Update logical clock based on the received timestamp
        clocks[deviceID].lock();
        clocks[deviceID].unlock();
    }
}
```

```
int main() {
    // Number of devices
    const int numDevices = 3;

    // Mutexes for logical clocks of each device
    std::vector<std::mutex> clocks(numDevices);

    // Mutex for sending messages to avoid race conditions
    std::mutex sendMutex;

    // Create sender and receiver threads for each device
    std::vector<std::thread> senderThreads;
    std::vector<std::thread> receiverThreads;

    for (int i = 0; i < numDevices; ++i) {
        senderThreads.push_back(std::thread(senderThread, i,
std::ref(clocks), std::ref(sendMutex)));
        receiverThreads.push_back(std::thread(receiverThread, i,
std::ref(clocks)));
    }

    // Join sender and receiver threads
    for (int i = 0; i < numDevices; ++i) {
```

```
        senderThreads[i].join();
        receiverThreads[i].join();
    }

    return 0;
}
```

Justification:

This code offers a fundamental structure for putting transmitter and receiver threads on each device.

To improve this, you would need to include actual socket communication code, handle mutual exclusion requests correctly, and update the logical clocks appropriately.

Implementing the crucial section access logic based on Lamport's Mutual Exclusion algorithm is also required.

Include Necessary Headers:

The following headers are required for different code functionalities:

iostream: For tasks involving input and output.
thread: To facilitate multithreading.
Mutex: For primitives in synchronization such as mutexes.

chrono: For using time intervals.
vector: Used to build dynamic arrays.

Mock Send Message Function:

```
void sendMessage(int recipient, int timestamp) {
   std::cout << "Sending message to device " << recipient << " with timestamp " << timestamp << std::endl;
   // Code for sending message over sockets
}
```

The actual code that would deliver a message across sockets is stored in a placeholder function called this one. For the time being, it just outputs a message with the sender and the timestamp.

Sender Thread Function:

```
void senderThread(int deviceID, std::vector<std::mutex>& clocks, std::mutex& sendMutex) {
   int logicalClock = 0; // Logical clock for the current device

   while (true) {
      // Simulate local event
      std::this_thread::sleep_for(std::chrono::milliseconds(1000));
```

```
        logicalClock++;

        // Update the logical clock for the current device
        std::lock_guard<std::mutex> clockLock(clocks[deviceID]);
        logicalClock = std::max(logicalClock,
static_cast<int>(clocks[deviceID].locked()));
        clocks[deviceID].unlock();

        // Send mutual exclusion request to other devices
        sendMutex.lock();
        for (int i = 0; i < clocks.size(); ++i) {
            if (i != deviceID) {
                sendMessage(i, logicalClock);
            }
        }
        sendMutex.unlock();
    }
}
```

For every device, this function represents the sender thread. It mimics a local event that happens on a regular basis. The device's logical clock is increased. Sends requests for mutual exclusion to other devices using the value of the current logical clock.

Receiver Thread Function:

```
void receiverThread(int deviceID, std::vector<std::mutex>& clocks) {
    while (true) {
        // Receive message from other devices (mocked)
        int sender;
        int timestamp; // Received timestamp
        std::cin >> sender >> timestamp;

        // Update the logical clock for the current device
        std::lock_guard<std::mutex> clockLock(clocks[deviceID]);
        clocks[deviceID].lock();
        clocks[deviceID].unlock();

        // Update logical clock based on the received timestamp
        clocks[deviceID].lock();
        clocks[deviceID].unlock();
    }
}
```

For every device, this function represents the receiver thread. It simulates receiving messages from other devices and looks for a timestamp and sender ID. Depending on the timestamp of the received message, updates the device's logical clock.

Main Function:

```
int main() {
    // Number of devices
    const int numDevices = 3;

    // Mutexes for logical clocks of each device
    std::vector<std::mutex> clocks(numDevices);

    // Mutex for sending messages to avoid race conditions
    std::mutex sendMutex;

    // Create sender and receiver threads for each device
    std::vector<std::thread> senderThreads;
    std::vector<std::thread> receiverThreads;

    for (int i = 0; i < numDevices; ++i) {
```

```
        senderThreads.push_back(std::thread(senderThread, i,
std::ref(clocks), std::ref(sendMutex)));
        receiverThreads.push_back(std::thread(receiverThread, i,
std::ref(clocks)));
    }

    // Join sender and receiver threads
    for (int i = 0; i < numDevices; ++i) {
        senderThreads[i].join();
        receiverThreads[i].join();
    }

    return 0;
}
```

The necessary variables and data structures are initialized by the main function.
For every device, it generates sender and receiver threads. After that, it calls join() on each thread while waiting for all of them to complete execution.