

# Distributed Systems Lab



Distributed Systems Lab,  
Department of Computer Science,  
School of Technology,  
Pandit Deendayal Energy University

By:

Tirth Shah,

22BCP230

Under the guidance of:

Dr. Shakti Mishra, Department of Computer Science, School of  
Technology, Pandit Deendayal Energy University

# Certificate



This is to certify that **Tirth Shah**, student of **G6-Div3 CSE'26** with  
enrolment number **22BCP230** has satisfactorily completed his work  
in **Distributed Systems Lab** under the guidance of **Dr. Shakti Mishra**.

---

Lab Instructor

---

Head of the department

# Practical-1

## **Aim:**

The objective of this assignment is to design and implement a simple client-server application in Java, where the server echoes the client's messages. Additionally, both the server and the client should be able to exchange messages. When the client types `exit()`, both the client and server should stop gracefully.

## **Theory:**

Client-server architecture is a model in which multiple clients connect to a central server. The server typically handles requests and responses, acting as the central communication hub. In this particular implementation, we focus on a basic form of communication where the server simply echoes messages back to the client and can also send messages to the client.

## **Client-Server Communication:**

1. **Client:** Initiates communication by connecting to the server using a socket. It sends and receives messages through input/output streams.
2. **Server:** Listens for incoming connections using a `ServerSocket`. Once a client is connected, the server can send and receive messages through input/output streams.

## **Graceful Shutdown:**

A graceful shutdown ensures that all resources (sockets, input/output streams) are closed properly, avoiding potential data loss or corruption. In this implementation, the client sends a special command (`exit()`) to the server, signaling both parties to terminate their connections and exit.

## **Functions & Modules Used:**

### **Server Code (EchoServer.java):**

- **Modules/Classes:**
  - ServerSocket: Listens for incoming client connections.
  - Socket: Represents the connection between the server and a specific client.
  - PrintWriter: Used to send messages to the client.
  - BufferedReader: Used to receive messages from the client.
  - Thread: For handling server-side message sending and listening concurrently.
- **Functions:**
  - start(int port): Starts the server on the specified port, waits for client connections, and handles communication.
  - stop(): Gracefully shuts down the server, closing all sockets and streams.

### **Client Code (EchoClient.java):**

- **Modules/Classes:**
  - Socket: Represents the connection between the client and the server.
  - PrintWriter: Used to send messages to the server.
  - BufferedReader: Used to receive messages from the server and get user input from the console.
  - Thread: For handling client-side message listening concurrently.
- **Functions:**
  - startConnection(String ip, int port): Connects to the server at the specified IP address and port.
  - stopConnection(): Gracefully shuts down the client, closing all sockets and streams.

## **Analysis:**

The implementation consists of a basic client-server application where the server echoes messages received from the client and can also send messages to the client. The main

challenge was to ensure that both the client and the server shut down gracefully when the `exit()` command is issued by the client.

### Communication Flow:

1. The client initiates a connection to the server and receives a confirmation message indicating successful connection.
2. The client can send messages to the server, which the server echoes back.
3. The server can also send messages to the client.
4. Upon entering `exit()`, the client sends this command to the server, prompting both the client and the server to terminate their connections and exit.

### Concurrency Considerations:

- Separate threads were used on both the client and the server to handle concurrent reading and writing of messages, ensuring that the server can send messages to the client even while the client is typing a message, and vice versa.

### Test Case and Output:

1. Client first will talk to server and show echoing abilities.
2. Server will type a message not received from client showing the second ability.
3. Client will show the `exit()` capability.

```
● (base) tirthshah@Tirths-MacBook-Pro Lect1 %  
Server started on port 6666  
Client connected: /127.0.0.1  
Client: Hi Server  
Client: Hi Sir  
hello client not from echo  
hell again not from echo  
Client requested to exit. Shutting down...
```

```
● (base) tirthshah@Tirths-MacBook-Pro Lect1 % cd "/Use  
Connected to the server successfully!  
Client: Hi Server  
  
Echo: Hi Server  
Client: Hi Sir  
  
Echo: Hi Sir  
  
Server: hello client not from echo  
  
Server: hell again not from echo  
Client: exit()
```

**Conclusion:**

The assignment successfully implements a simple client-server communication system in Java. The server can echo messages from the client and also send messages back to the client. Additionally, a graceful shutdown is achieved when the client sends the `exit()` command, ensuring that both the client and the server close their connections and terminate without errors.

The use of multithreading allows for concurrent message handling, providing a seamless communication experience between the client and the server. The solution meets the specified requirements, making it a solid foundation for more complex client-server applications.

## Practical-2

### Aim:

The aim of this project is to design and implement a simple chat server and client application in Java that supports basic functionalities like broadcasting messages to all connected clients, sending direct messages between clients, listing active users, and handling client disconnections after inactivity.

### Theory:

A chat server is a system designed to manage and facilitate communication between multiple clients. It involves two main components: the server, which listens for incoming connections and manages the communication between clients, and the clients, which connect to the server and send/receive messages.

The core components of this chat application include:

1. **Socket Programming:** Sockets provide the communication mechanism between two computers using TCP (Transmission Control Protocol). In this project, the `ServerSocket` class is used by the server to listen for incoming connections, and the `Socket` class is used by the client to establish a connection to the server.
2. **Multithreading:** To handle multiple clients simultaneously, the server uses multithreading. Each client is managed by a separate thread, ensuring that the server can process multiple requests concurrently without blocking other clients.
3. **Timeout Handling:** To manage inactive clients, a timeout mechanism is implemented. If a client remains inactive for a specified period, the server disconnects the client to free up resources.
4. **Message Broadcasting and Direct Messaging:** The server can broadcast messages to all clients or relay direct messages between specific clients based on their unique call signs.

## Function & Modules Used:

### 1. ChatServer Class:

- **ServerSocket:** Used to listen for incoming client connections on a specified port.
- **ArrayList<ClientHandler> clients:** Maintains a list of all connected clients.
- **HashMap<String, ClientHandler> clientMap:** Maps client call signs to their corresponding ClientHandler objects for easy retrieval during direct messaging.
- **ExecutorService pool:** Manages a pool of threads to handle client connections and tasks efficiently.

#### Methods:

- **run():** Starts the server and listens for incoming connections.
- **broadcast(String message):** Sends a message to all connected clients.
- **shutServer():** Shuts down the server and releases resources.
- **log(String message):** Logs server events with a timestamp.

### 2. ClientHandler Class (Nested inside ChatServer):

- **Socket client:** Represents the connection between the server and a client.
- **BufferedReader in, PrintWriter out:** Used for reading messages from and sending messages to the client.
- **TimeoutHandler timeoutHandler:** Manages client inactivity by implementing a timeout mechanism.

#### Methods:

- **run():** Handles client communication, including reading client messages and responding appropriately.
- **sendMessageToClient(String targetCallSign, String message):** Sends a direct message to a specific client based on their call sign.



- `shutClient()`: Closes the client connection and releases resources.
- `sendMessage(String message)`: Sends a message to the connected client.
- `getCallSign()`: Retrieves the client's call sign.

### 3. **ChatClient Class:**

- **Socket clientSocket**: Establishes a connection to the server.
- **PrintWriter out, BufferedReader in**: Used for sending messages to and receiving messages from the server.
- **String callSign**: Represents the client's unique identifier in the chat system.
- **Thread serverListenerThread**: Listens for messages from the server in a separate thread.

#### **Methods:**

- `startConnection(String ip, int port)`: Establishes a connection to the server.
- `sendMessage(String msg)`: Sends a message to the server.
- `stopConnection()`: Closes the connection to the server.
- `startServerListener()`: Listens for incoming messages from the server and displays them to the client.
- `run()`: Manages client interaction, including setting up the connection, handling user input, and managing the connection lifecycle.

#### **Analysis:**

The chat server and client application were tested under various conditions to evaluate its performance and functionality. The following key aspects were analyzed:

1. **Concurrency Handling**: The server successfully managed multiple client connections simultaneously without any noticeable delay or resource contention, demonstrating the effectiveness of the multithreading approach.
2. **Timeout Mechanism**: The timeout handler effectively disconnected inactive clients after 1 minute of inactivity, ensuring that server resources were not wasted on idle connections.

3. **Message Broadcasting and Direct Messaging:** Both broadcasting and direct messaging functionalities worked as expected. Clients could easily send messages to all users or target specific users using the appropriate commands.
4. **Error Handling:** The server handled various exceptions gracefully, including client disconnections, network failures, and invalid commands, maintaining overall system stability.

### Test Case and Output:

1. Rishabh was demonstrating how it will get disconnected from the chat after inactivity.

```
○ (base) tirthshah@Tirths-MacBook-Pro Lect2 % cd "/Users/tirthshah/Documents/PDEU-Sem-5/Lab/Distributed Systems/Lect2/" &
Connected to server at localhost:4221
Enter Your Call Sign: Rishabh
Type your messages (type '@exit' to quit,
                        '@list' to list all the available persons to chat,
                        '@broadcast' to send a message to all available persons,
                        '@CallSign' replace CallSign to the call sign of the person you want to do 1-1 chat):

User Rishabh has joined the chat!!
User Rudra has joined the chat!!
User Tirth has joined the chat!!
Tirth: Rishabh you will be the sacrifice to show timeout
Tirth: I will exit
User Tirth has left the chat!!
You have been inactive for 1 minute. Disconnecting... Exit and reconnect to chat again.
```

2. Tirth will show broadcast and list facilities and will show exit facility showing that it will not disturb server. It will also show how he chatted with Rudra

```

● (base) tirthshah@Tirths-MacBook-Pro Lect2 % cd "/Users/tirthshah/Documents/PDEU-Sem-5/Lab/Distributed Systems/Lect2
Connected to server at localhost:4221
Enter Your Call Sign: Tirth
Type your messages (type '@exit' to quit,
                        '@list' to list all the available persons to chat,
                        '@broadcast' to send a message to all available persons,
                        '@CallSign' replace CallSign to the call sign of the person you want to do 1-1 chat):

```

User Tirth has joined the chat!!

Tirth: @list

Users in chat: [Tirth, Rishabh, Rudra]

Tirth: @broadcast Rishabh you will be the sacrifice to show timeout

Tirth: Rishabh you will be the sacrifice to show timeout

Tirth: @Rudra HI Rudra


Rudra: Hi

Tirth: @broadcast I will exit

Tirth: I will exit

Tirth: @exit

User Tirth has left the chat!!

Tirth:  .....

```

(base) tirthshah@Tirths-MacBook-Pro Lect2 % cd "/Users/tirthshah/Documents/PDEU-Sem-5/Lab/Dis
[2024-08-30 22:31:59] Server started on port 4221
[2024-08-30 22:32:02] Client connected: /127.0.0.1
[2024-08-30 22:32:04] Broadcast: User Rishabh has joined the chat!!
[2024-08-30 22:32:07] Client connected: /127.0.0.1
[2024-08-30 22:32:09] Broadcast: User Rudra has joined the chat!!
[2024-08-30 22:32:13] Client connected: /127.0.0.1
[2024-08-30 22:32:15] Broadcast: User Tirth has joined the chat!!
[2024-08-30 22:32:49] User Tirth has left the chat!!
[2024-08-30 22:33:04] User Rishabh has been inactive for 1 minute. Disconnecting...
.....

```

Connected to server at localhost:4221

Enter Your Call Sign: Rudra

Type your messages (type '@exit' to quit,  
 '@list' to list all the available persons to chat,  
 '@broadcast' to send a message to all available persons,  
 '@CallSign' replace CallSign to the call sign of the person you want to do 1-1 chat):

User Rudra has joined the chat!!

User Tirth has joined the chat!!

Tirth: Rishabh you will be the sacrifice to show timeout

Tirth: HI Rudra

Rudra: @Tirth Hi

Tirth: I will exit

User Tirth has left the chat!!

Rudra: @list

Users in chat: [Rishabh, Rudra]

You have been inactive for 1 minute. Disconnecting... Exit and reconnect to chat again.

## Conclusion:

The project demonstrates the practical application of several key concepts in Java networking and concurrency.

The chat server is capable of handling real-time communication between multiple clients, offering both broadcast and private messaging functionalities. The inclusion of a timeout mechanism for inactive clients enhances the server's resource management and ensures that it remains responsive and available for active users.

This implementation could be further extended to include features such as secure communication, adding support for sending multimedia files, persistent user sessions, or integration with a database for storing chat history.

## Practical-3

### **Aim:**

The primary goal of this assignment is to simulate a distributed system where multiple processes communicate through message passing.

The focus is on implementing Lamport's Logical Clock algorithm to order events across processes, thereby ensuring consistent event ordering in a distributed environment lacking a global clock.

### **Theory:**

#### **Distributed Systems and Event Ordering:**

In distributed systems, processes operate independently and communicate by exchanging messages. Since these processes do not share a global clock, determining the order of events across different processes is challenging.

Without a proper mechanism, it becomes difficult to resolve conflicts, manage dependencies, or synchronize actions across the system.

#### **Lamport's Logical Clock:**

Leslie Lamport introduced the concept of Logical Clocks to address the problem of ordering events in distributed systems. Each process maintains its own logical clock, which is a simple integer value. The clock is incremented according to specific rules when certain events occur:

1. **Internal Event:** When a process performs an internal event, it increments its logical clock by 1.
2. **Send Event:** When a process sends a message, it increments its logical clock by 1 and attaches this updated clock value to the message.

3. **Receive Event:** Upon receiving a message, a process sets its logical clock to the maximum of its own clock and the received clock, then increments the result by 1.

### **Vector Clocks:**

While Lamport's Logical Clocks can order events, they may not always capture causal relationships between events. Vector Clocks, an extension of Lamport's clocks, are used to address this limitation. In a Vector Clock system:

- Each process maintains an array (vector) where each element represents the clock value of a particular process.
- During communication, the vector clocks are exchanged, and the receiving process updates its vector by taking the element-wise maximum of its own vector and the received vector.

This method ensures that all causal relationships between events are captured, making Vector Clocks more accurate than Lamport's Logical Clocks in complex scenarios.

### **Functions & Modules Used:**

#### **1. Message Class:**

- **Attributes:** senderId, receiverId, timeStamp, vectorClockOfSender.
- **Purpose:** Represents a message sent between processes. Includes the logical clock value and vector clock of the sender.
- **Methods:**
  - `getSenderId()`: Returns the sender's ID.
  - `getReceiverId()`: Returns the receiver's ID.
  - `getTimeStamp()`: Returns the timestamp of the message.
  - `getVectorClockOfSender()`: Returns the vector clock of the sender.
  - `toString()`: Returns a string representation of the message.

## 2. SharedBuffer Class:

- **Attributes:** ownerId, messageQueue.
- **Purpose:** Acts as a mailbox for a process, storing incoming messages in a queue.
- **Methods:**
  - `addMessage(Message message)`: Adds a message to the queue.
  - `retrieveMessage()`: Retrieves and removes a message from the queue or returns null if the queue is empty.
  - `logEvent(String event)`: Logs events to a file named LamportLog.txt.

## 3. ProcessThread Class:

- **Attributes:** ownerId, sharedBuffer, logicalClock, vectorClock, rand, iterations.
- **Purpose:** Represents a process in the distributed system. It performs internal, send, and receive events, updating its logical and vector clocks accordingly.
- **Methods:**
  - `run()`: Executes the process's operations for a specified number of iterations.
  - `performInternalEvent()`: Handles internal events by incrementing the logical and vector clocks.
  - `performSendEvent()`: Handles send events by sending messages to other processes and updating the clocks.
  - `performReceiveEvent()`: Handles receive events by updating the clocks based on the received message.
  - `logEvent(String event)`: Logs the process's events to the LamportLog.txt file.
  - `getClockValue()`: Returns the current logical clock value.
  - `getVectorClock()`: Returns the current vector clock.

#### 4. LamportLogicalClock Class:

- **Attributes:** buffers.
- **Purpose:** Serves as the main class, responsible for initializing and managing processes in the simulation.
- **Methods:**
  - `main(String[] args)`: Initializes processes, assigns them to threads, and starts the simulation.
  - `getBuffer(int processId)`: Retrieves the buffer (mailbox) of a specified process.

#### Analysis:

#### Simulation Setup:

- **Number of Processes:** Three processes are used in the simulation, each running in its own thread.
- **Iterations:** Each process performs a fixed number of iterations (10 in this case).
- **Events:** The simulation randomly selects one of three possible events (internal, send, or receive) during each iteration.

#### Execution Flow:

1. **Initialization:** Each process starts with a logical clock value of 0 and an initialized vector clock (array) where all elements are set to 0.
2. **Event Execution:**
  - **Internal Event:** The process increments its logical clock and updates its vector clock for its own ID.
  - **Send Event:** The process increments its logical and vector clocks, creates a message containing these clocks, and sends it to another process's buffer.
  - **Receive Event:** The process retrieves a message from its buffer, updates its clocks by taking the maximum values, and logs the event.
3. **Logging:** Each event is logged to a `LamportLog.txt` file, capturing the event type, current logical clock, and vector clock values.



### **Output Stored In Log File:**

Simulation started at: 2024-08-31 23:07:48

Process 2 sent Message from Process 2 to Process 0 at Time 1, Vector Clock: [0, 0, 1]

Process 0 received Message from Process 2 to Process 0 at Time 1, Vector Clock: [0, 0, 1].

Updated clock: 2, Vector Clock: [1, 0, 1]

Process 1's mailbox is empty or no message received.

Process 1 sent Message from Process 1 to Process 2 at Time 1, Vector Clock: [0, 1, 0]

Process 1's mailbox is empty or no message received.

Process 0's mailbox is empty or no message received.

Process 2 received Message from Process 1 to Process 2 at Time 1, Vector Clock: [0, 1, 0].

Updated clock: 2, Vector Clock: [0, 1, 2]

Process 1 performed internal event at time 2, Vector Clock: [0, 2, 0]

Process 0's mailbox is empty or no message received.

Process 2's mailbox is empty or no message received.

Process 0 sent Message from Process 0 to Process 2 at Time 3, Vector Clock: [2, 0, 1]

Process 2 sent Message from Process 2 to Process 1 at Time 3, Vector Clock: [0, 1, 3]

Process 1 received Message from Process 2 to Process 1 at Time 3, Vector Clock: [0, 1, 3].

Updated clock: 4, Vector Clock: [0, 3, 3]

Process 0 sent Message from Process 0 to Process 1 at Time 4, Vector Clock: [3, 0, 1]

Process 1 sent Message from Process 1 to Process 0 at Time 5, Vector Clock: [0, 4, 3]

Process 2 performed internal event at time 4, Vector Clock: [0, 1, 4]

Process 2 performed internal event at time 5, Vector Clock: [0, 1, 5]

Process 0 received Message from Process 1 to Process 0 at Time 5, Vector Clock: [0, 4, 3].

Updated clock: 6, Vector Clock: [4, 4, 3]

Process 1 sent Message from Process 1 to Process 2 at Time 6, Vector Clock: [0, 5, 3]

Process 0's mailbox is empty or no message received.

Process 2 performed internal event at time 6, Vector Clock: [0, 1, 6]

Process 0 performed internal event at time 7, Vector Clock: [5, 4, 3]

Process 1 sent Message from Process 1 to Process 0 at Time 7, Vector Clock: [0, 6, 3]

Process 0 received Message from Process 1 to Process 0 at Time 7, Vector Clock: [0, 6, 3].

Updated clock: 8, Vector Clock: [6, 6, 3]

Process 1 received Message from Process 0 to Process 1 at Time 4, Vector Clock: [3, 0, 1].

Updated clock: 8, Vector Clock: [3, 7, 3]

Process 2 sent Message from Process 2 to Process 0 at Time 7, Vector Clock: [0, 1, 7]

Process 2 received Message from Process 0 to Process 2 at Time 3, Vector Clock: [2, 0, 1].

Updated clock: 8, Vector Clock: [2, 1, 8]

Process 1 performed internal event at time 9, Vector Clock: [3, 8, 3]

Process 0 performed internal event at time 9, Vector Clock: [7, 6, 3]

Thread 1 completed their iterations. Final Clock Value: 9, Vector Clock: [3, 8, 3]

Process 2 sent Message from Process 2 to Process 0 at Time 9, Vector Clock: [2, 1, 9]

Thread 0 completed their iterations. Final Clock Value: 9, Vector Clock: [7, 6, 3]

Thread 2 completed their iterations. Final Clock Value: 9, Vector Clock: [2, 1, 9]

```
LamportLogicalClock.java 1, U  LamportLog.txt U x
Lab > System Software and Compiler Design > Lect5 > LamportLog.txt
32
33 Process 2 performed internal event at time 4, Vector Clock: [0, 1, 4]
34
35 Process 2 performed internal event at time 5, Vector Clock: [0, 1, 5]
36
37 Process 0 received Message from Process 1 to Process 0 at Time 5, Vector Clock: [0, 4, 3]. Updated clock: 6, Vector Clock: [4, 4, 3]
38
39 Process 1 sent Message from Process 1 to Process 2 at Time 6, Vector Clock: [0, 5, 3]
40
41 Process 0's mailbox is empty or no message received.
42
43 Process 2 performed internal event at time 6, Vector Clock: [0, 1, 6]
44
45 Process 0 performed internal event at time 7, Vector Clock: [5, 4, 3]
46
47 Process 1 sent Message from Process 1 to Process 0 at Time 7, Vector Clock: [0, 6, 3]
48
49 Process 0 received Message from Process 1 to Process 0 at Time 7, Vector Clock: [0, 6, 3]. Updated clock: 8, Vector Clock: [6, 6, 3]
50
51 Process 1 received Message from Process 0 to Process 1 at Time 4, Vector Clock: [3, 0, 1]. Updated clock: 8, Vector Clock: [3, 7, 3]
52
53 Process 2 sent Message from Process 2 to Process 0 at Time 7, Vector Clock: [0, 1, 7]
54
55 Process 2 received Message from Process 0 to Process 2 at Time 3, Vector Clock: [2, 0, 1]. Updated clock: 8, Vector Clock: [2, 1, 8]
56
57 Process 1 performed internal event at time 9, Vector Clock: [3, 8, 3]
58
59 Process 0 performed internal event at time 9, Vector Clock: [7, 6, 3]
60
61 Thread 1 completed their iterations. Final Clock Value: 9, Vector Clock: [3, 8, 3]
62
63 Process 2 sent Message from Process 2 to Process 0 at Time 9, Vector Clock: [2, 1, 9]
64
65 Thread 0 completed their iterations. Final Clock Value: 9, Vector Clock: [7, 6, 3]
66
67 Thread 2 completed their iterations. Final Clock Value: 9, Vector Clock: [2, 1, 9]
68
```

## Calculations:

Initial state:

VC: P0 [0,0,0], P1 [0,0,0], P2 [0,0,0]

LC: P0 0, P1 0, P2 0

1. Process 2 sends message to Process 0 Calculation:  $\max(0, 0) + 1 = 1$

VC: P0 [0,0,0], P1 [0,0,0], P2 [0,0,1]

LC: P0 0, P1 0, P2 1

2. Process 0 receives message from Process 2 Calculation:  $\max(0, 1) + 1 = 2$   
VC: P0 [1,0,1], P1 [0,0,0], P2 [0,0,1]  
LC: P0 2, P1 0, P2 1
3. Process 1 sends message to Process 2 Calculation:  $\max(0, 0) + 1 = 1$   
VC: P0 [1,0,1], P1 [0,1,0], P2 [0,0,1]  
LC: P0 2, P1 1, P2 1
4. Process 2 receives message from Process 1 Calculation:  $\max(1, 1) + 1 = 2$   
VC: P0 [1,0,1], P1 [0,1,0], P2 [0,1,2]  
LC: P0 2, P1 1, P2 2
5. Process 1 internal event Calculation:  $\max(1, 1) + 1 = 2$   
VC: P0 [1,0,1], P1 [0,2,0], P2 [0,1,2]  
LC: P0 2, P1 2, P2 2
6. Process 0 sends message to Process 2 Calculation:  $\max(2, 2) + 1 = 3$   
VC: P0 [2,0,1], P1 [0,2,0], P2 [0,1,2]  
LC: P0 3, P1 2, P2 2
7. Process 2 sends message to Process 1 Calculation:  $\max(2, 2) + 1 = 3$   
VC: P0 [2,0,1], P1 [0,2,0], P2 [0,1,3]  
LC: P0 3, P1 2, P2 3
8. Process 1 receives message from Process 2 Calculation:  $\max(2, 3) + 1 = 4$   
VC: P0 [2,0,1], P1 [0,3,3], P2 [0,1,3]  
LC: P0 3, P1 4, P2 3
9. Process 0 sends message to Process 1 Calculation:  $\max(3, 3) + 1 = 4$   
VC: P0 [3,0,1], P1 [0,3,3], P2 [0,1,3]  
LC: P0 4, P1 4, P2 3
10. Process 1 sends message to Process 0 Calculation:  $\max(4, 4) + 1 = 5$   
VC: P0 [3,0,1], P1 [0,4,3], P2 [0,1,3]  
LC: P0 4, P1 5, P2 3
11. Process 2 internal event Calculation:  $\max(3, 3) + 1 = 4$   
VC: P0 [3,0,1], P1 [0,4,3], P2 [0,1,4]  
LC: P0 4, P1 5, P2 4
12. Process 2 internal event Calculation:  $\max(4, 4) + 1 = 5$   
VC: P0 [3,0,1], P1 [0,4,3], P2 [0,1,5]  
LC: P0 4, P1 5, P2 5

13. Process 0 receives message from Process 1 Calculation:  $\max(4, 5) + 1 = 6$

VC: P0 [4,4,3], P1 [0,4,3], P2 [0,1,5]

LC: P0 6, P1 5, P2 5

14. Process 1 sends message to Process 2 Calculation:  $\max(5, 5) + 1 = 6$

VC: P0 [4,4,3], P1 [0,5,3], P2 [0,1,5]

LC: P0 6, P1 6, P2 5

15. Process 2 internal event Calculation:  $\max(5, 5) + 1 = 6$

VC: P0 [4,4,3], P1 [0,5,3], P2 [0,1,6]

LC: P0 6, P1 6, P2 6

16. Process 0 internal event Calculation:  $\max(6, 6) + 1 = 7$

VC: P0 [5,4,3], P1 [0,5,3], P2 [0,1,6]

LC: P0 7, P1 6, P2 6

17. Process 1 sends message to Process 0 Calculation:  $\max(6, 6) + 1 = 7$

VC: P0 [5,4,3], P1 [0,6,3], P2 [0,1,6]

LC: P0 7, P1 7, P2 6

18. Process 0 receives message from Process 1 Calculation:  $\max(7, 7) + 1 = 8$

VC: P0 [6,6,3], P1 [0,6,3], P2 [0,1,6]

LC: P0 8, P1 7, P2 6

19. Process 1 receives message from Process 0 Calculation:  $\max(7, 4) + 1 = 8$

VC: P0 [6,6,3], P1 [3,7,3], P2 [0,1,6]

LC: P0 8, P1 8, P2 6

20. Process 2 sends message to Process 0 Calculation:  $\max(6, 6) + 1 = 7$

VC: P0 [6,6,3], P1 [3,7,3], P2 [0,1,7]

LC: P0 8, P1 8, P2 7

21. Process 2 receives message from Process 0 Calculation:  $\max(7, 3) + 1 = 8$

VC: P0 [6,6,3], P1 [3,7,3], P2 [2,1,8]

LC: P0 8, P1 8, P2 8

22. Process 1 internal event Calculation:  $\max(8, 8) + 1 = 9$

VC: P0 [6,6,3], P1 [3,8,3], P2 [2,1,8]

LC: P0 8, P1 9, P2 8

23. Process 0 internal event Calculation:  $\max(8, 8) + 1 = 9$

VC: P0 [7,6,3], P1 [3,8,3], P2 [2,1,8]

LC: P0 9, P1 9, P2 8

24. Process 2 sends message to Process 0 Calculation:  $\max(8, 8) + 1 = 9$

VC: P0 [7,6,3], P1 [3,8,3], P2 [2,1,9]

LC: P0 9, P1 9, P2 9

Final state:

VC: P0 [7,6,3], P1 [3,8,3], P2 [2,1,9]

LC: P0 9, P1 9, P2 9

### **Analysis of Results:**

- The simulation ensures that all events are correctly ordered according to the logical clocks.
- When processes communicate, the receiving process updates its logical clock based on the message's timestamp, ensuring consistency across the system.
- Vector Clocks provide additional information, helping identify the causality of events, which is particularly useful in complex scenarios.

### **Conclusion:**

The implementation of Lamport's Logical Clock in a simulated distributed system successfully demonstrates how logical clocks can be used to order events in environments without a global clock.

The extension to Vector Clocks provides a more nuanced view of event causality, enabling better conflict resolution and event ordering.

Through this assignment, the foundational principles of distributed systems, message passing, and clock synchronization have been explored, highlighting the importance of logical clocks in achieving consistent event ordering.

The code's modular structure, including the Message, SharedBuffer, ProcessThread, and LamportLogicalClock classes, ensures clarity and reusability, adhering to best practices in software development.

The results, as logged in LamportLog.txt, confirm the correct operation of the logical and vector clocks, meeting the assignment's objectives and learning outcomes.

Further extensions, such as real-time communication or network-based implementations, could build on this foundation, adding real-world relevance and complexity.