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:

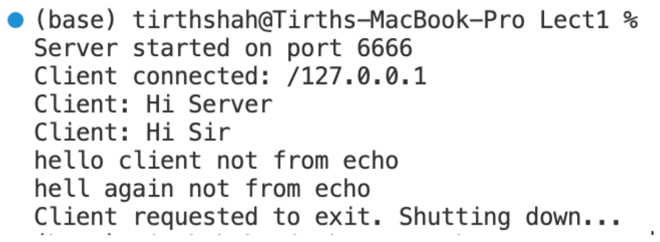
1. The client initiates a connection to the server and receives a confirmation message indicating successful connection.
2. The client can send messages to the server, which the server echoes back.
3. The server can also send messages to the client.
4. Upon entering exit(), the client sends this command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

Concurrency Considerations:

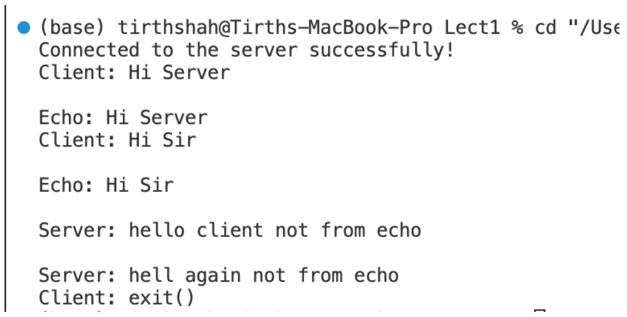
- Separate threads were used on both the client and the server to handle concurrent reading and writing of messages, ensuring that the server can send messages to the client even while the client is typing a message, and vice versa.

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:

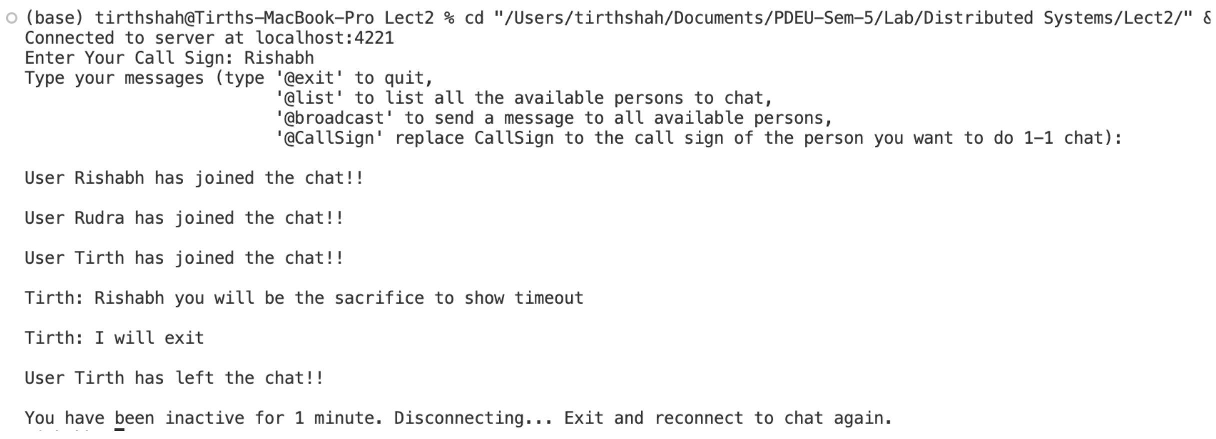
The chat server and client application were tested under various conditions to evaluate its performance and functionality. The following key aspects were analyzed:

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.

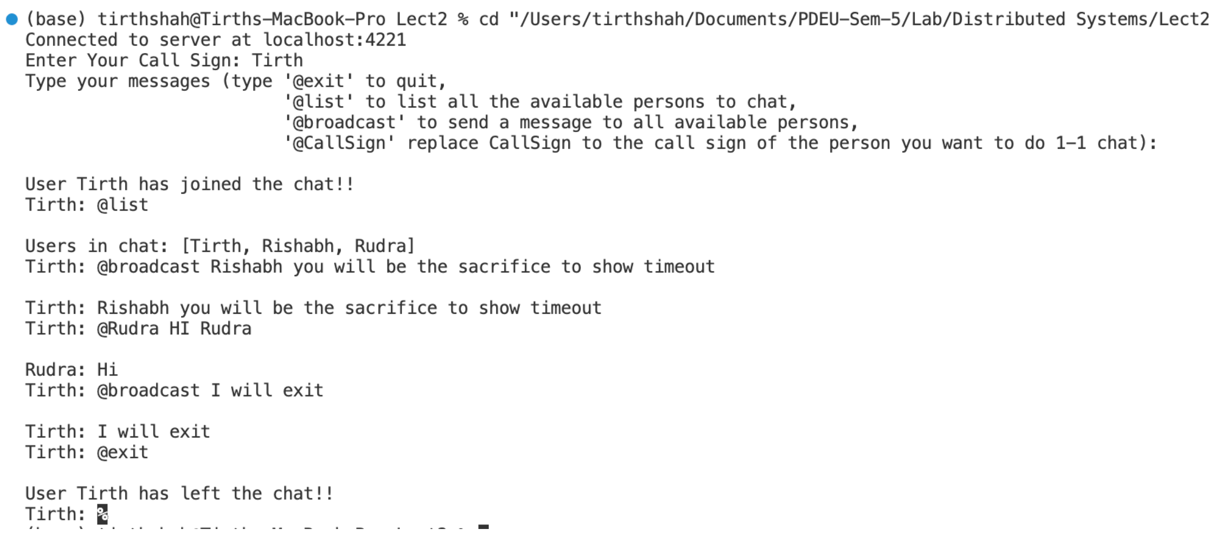
Test Case and Output:

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

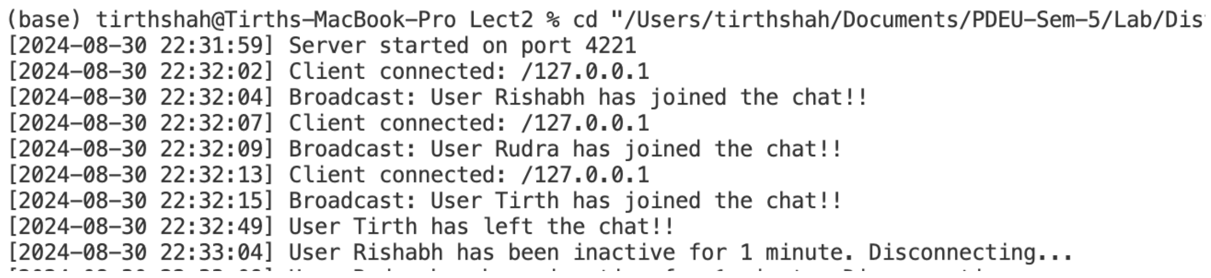


```
○ (base) tirthshah@Tirths-MacBook-Pro Lect2 % cd "/Users/tirthshah/Documents/PDEU-Sem-5/Lab/Distributed Systems/Lect2/" &
  Connected to server at localhost:4221
  Enter Your Call Sign: Rishabh
  Type your messages (type '@exit' to quit,
                          '@list' to list all the available 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.
```

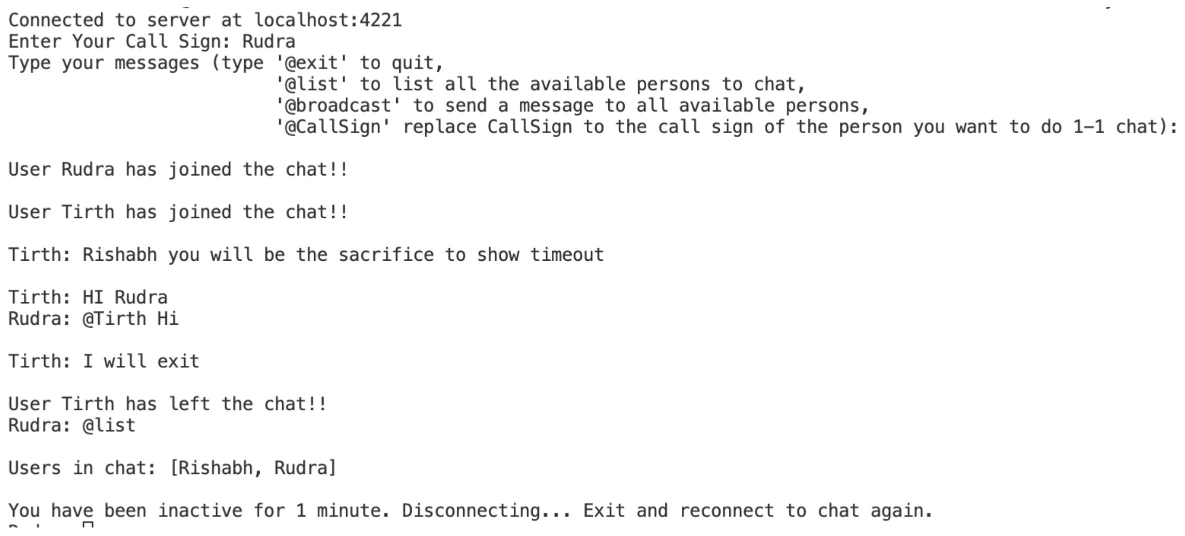
2. Tirth will show broadcast and list facilities and will show exit facility showing that it will not disturb server. It will also show how he chatted with Rudra



```
● (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:

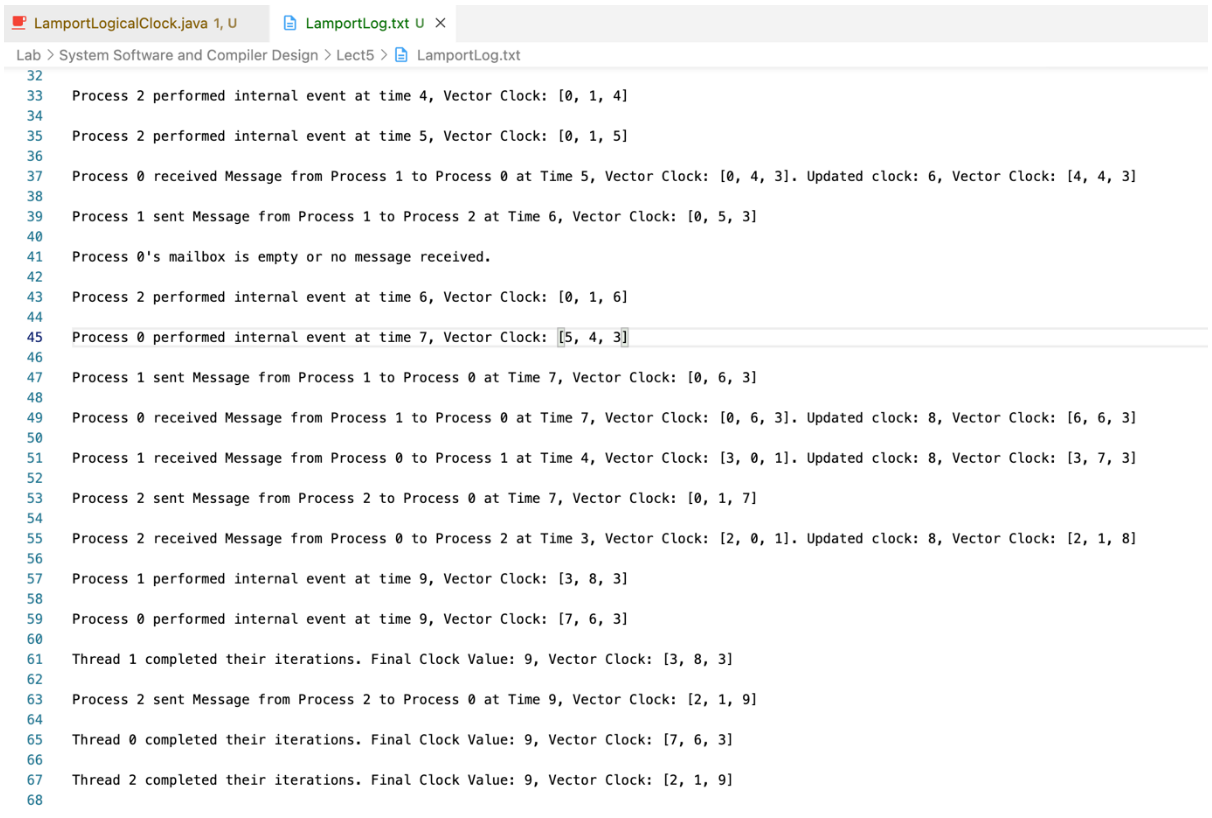
```
Simulation started at: 2024-08-31 23:07:48

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
Lab > System Software and Compiler Design > Lect5 > LamportLog.txt

```

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]

```

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.

Practical-3

Aim:

This assignment aims to familiarize students with the concepts of Remote Procedure Call (RPC) and Remote Method Invocation (RMI) in distributed systems. By completing this assignment, students will learn how to implement a basic distributed application where a client makes remote calls to a server using both RPC and RMI.

Theory:

RPC

Theory

Remote Procedure Call (RPC) is a protocol that allows a program to invoke procedures (functions) on a remote server as if they were local calls. This abstraction allows developers to build distributed applications where the client and server communicate over a network. In this project, we implement a simple RPC application using the ZeroRPC framework in Python, which provides a straightforward way to create a client-server architecture. The server exposes basic arithmetic operations, and the client can invoke these operations remotely.

Functions and Modules Used

- **ZeroRPC**: A simple RPC framework that allows for remote method invocation.
- **Python Built-in Functions**:
  - input(): To capture user input.
  - float(): To convert string input to float for arithmetic operations.
  - Exception handling with try and except to manage errors such as division by zero.

Source Code and Output:

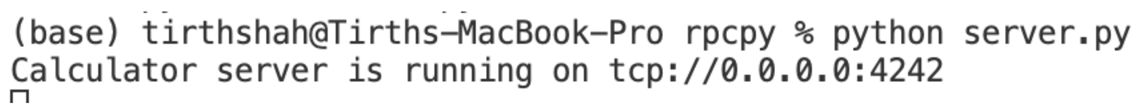
server.py

```
import zerorpc

class Calculator:
    def add(self, a, b):
        return a + b
    def subtract(self, a, b):
        return a - b

    def multiply(self, a, b):
        return a * b

    def divide(self, a, b):
        if b == 0:
            raise ValueError("Division by zero is not allowed.")
        return a / b

def main():
    server = zerorpc.Server(Calculator())
    server.bind("tcp://0.0.0.0:4242")  # Bind to a port
    print("Calculator server is running on tcp://0.0.0.0:4242")
    server.run()  # Start the server

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

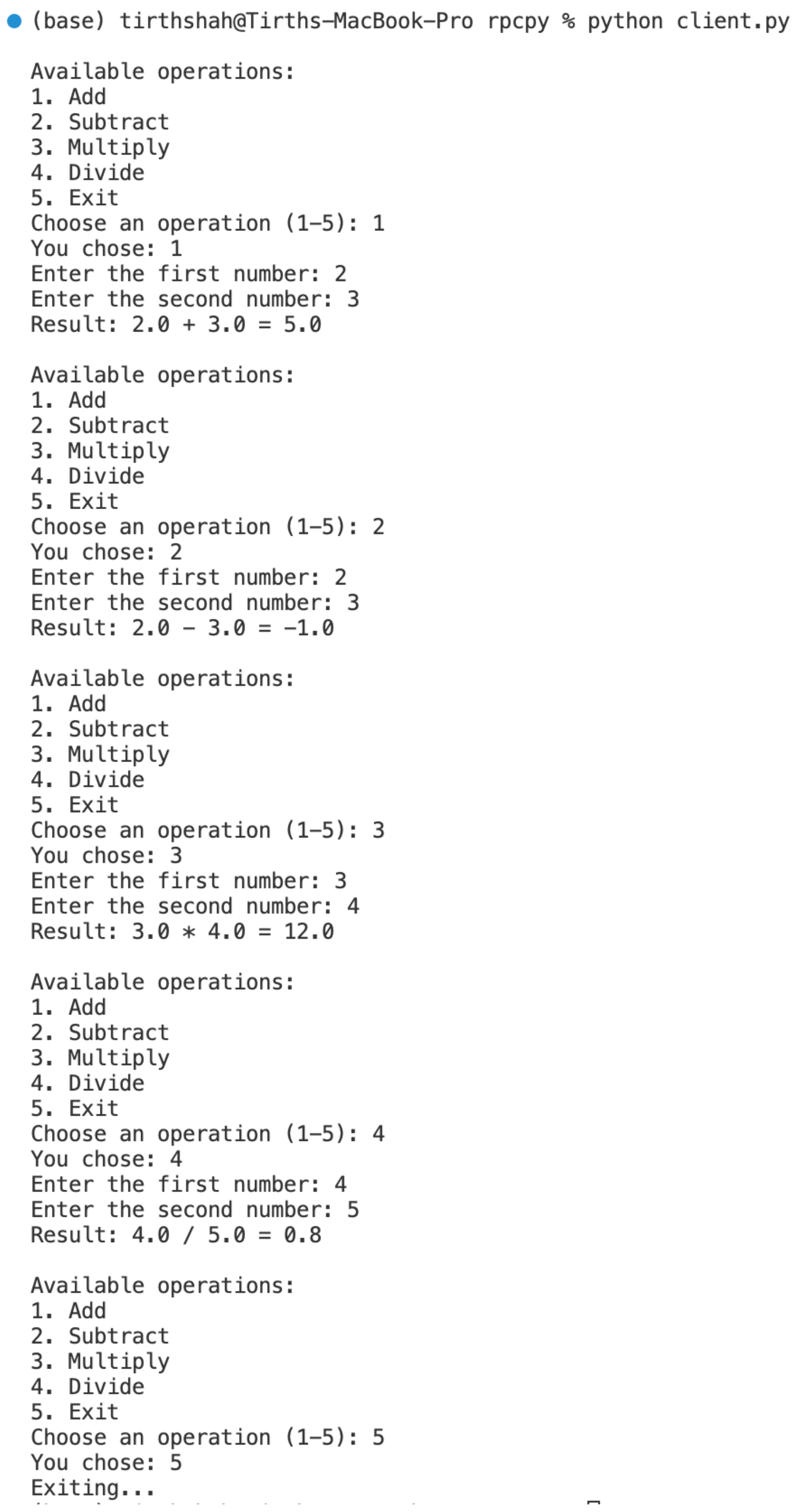
client.py

```
import zerorpc

def main():
    client = zerorpc.Client()
    client.connect("tcp://127.0.0.1:4242")  # Connect to the server

    while True:
        print("\nAvailable operations:")
        print("1. Add")
        print("2. Subtract")
        print("3. Multiply")
        print("4. Divide")
        print("5. Exit")

        choice = int(input("Choose an operation (1-5): "))
        print("You chose: {}".format(choice))  # Debug print

        if choice == 5:
            print("Exiting...")
            break

        a = float(input("Enter the first number: "))
        b = float(input("Enter the second number: "))

        try:
            if choice == 1:
                result = client.add(a, b)
                print("Result: {} + {} = {}".format(a, b, result))
            elif choice == 2:
                result = client.subtract(a, b)
                print("Result: {} - {} = {}".format(a, b, result))
            elif choice == 3:
                result = client.multiply(a, b)
                print("Result: {} * {} = {}".format(a, b, result))
            elif choice == 4:
                result = client.divide(a, b)
                print("Result: {} / {} = {}".format(a, b, result))
            else:
                print("Invalid choice. Please choose a valid operation.")

        except zerorpc.exceptions.RemoteError as e:
            print("Error: {}".format(e))

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

Output:



```
(base) tirthshah@Tirths-MacBook-Pro rpcpy % python server.py
Calculator server is running on tcp://0.0.0.0:4242
```



```
● (base) tirthshah@Tirths-MacBook-Pro rpcpy % python client.py

  Available operations:
  1. Add
  2. Subtract
  3. Multiply
  4. Divide
  5. Exit
  Choose an operation (1-5): 1
  You chose: 1
  Enter the first number: 2
  Enter the second number: 3
  Result: 2.0 + 3.0 = 5.0

  Available operations:
  1. Add
  2. Subtract
  3. Multiply
  4. Divide
  5. Exit
  Choose an operation (1-5): 2
  You chose: 2
  Enter the first number: 2
  Enter the second number: 3
  Result: 2.0 - 3.0 = -1.0

  Available operations:
  1. Add
  2. Subtract
  3. Multiply
  4. Divide
  5. Exit
  Choose an operation (1-5): 3
  You chose: 3
  Enter the first number: 3
  Enter the second number: 4
  Result: 3.0 * 4.0 = 12.0

  Available operations:
  1. Add
  2. Subtract
  3. Multiply
  4. Divide
  5. Exit
  Choose an operation (1-5): 4
  You chose: 4
  Enter the first number: 4
  Enter the second number: 5
  Result: 4.0 / 5.0 = 0.8

  Available operations:
  1. Add
  2. Subtract
  3. Multiply
  4. Divide
  5. Exit
  Choose an operation (1-5): 5
  You chose: 5
  Exiting...
```

Analysis and Flow

1. **Server**:
   - The server is implemented using ZeroRPC. It defines a Calculator class that includes methods for addition, subtraction, multiplication, and division.
   - The server listens for client requests on a specified port (4242).
   - When a client invokes an operation, the corresponding method is executed, and the result is sent back.
2. **Client**:
   - The client connects to the server and presents a menu of operations to the user.
   - The user selects an operation and enters two numbers.
   - The client calls the appropriate method on the server and displays the result.

Conclusion

This project demonstrates a simple RPC implementation using ZeroRPC in Python, providing a clear example of how to set up client-server communication for basic arithmetic operations. The structure allows for easy scalability and extension of functionality, showcasing the power of remote procedure calls in distributed systems.

RMI Banking System

1. Source Code

```java
// Bank.java
package rmiexample;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Bank extends Remote {
    int createAccount(String accountHolderName) throws RemoteException;
    double getBalance(int accountId) throws RemoteException;
    void deposit(int accountId, double amount) throws RemoteException;
    boolean withdraw(int accountId, double amount) throws RemoteException;
}



// BankAccount.java
package rmiexample;

public class BankAccount {
    private String accountHolderName;
    private double balance;

    public BankAccount(String accountHolderName) {
        this.accountHolderName = accountHolderName;
        this.balance = 0.0;
    }

    public synchronized double getBalance() {
        return balance;
    }

    public synchronized void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    public synchronized boolean withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            return true;
        }
        return false;
    }
}
```

```java
// BankClient.java
package rmiexample;

import java.rmi.Naming;
import java.util.Scanner;

public class BankClient {
    public static void main(String[] args) {
        try {
            Bank bank = (Bank) Naming.lookup("rmi://localhost/BankServer");
            Scanner scanner = new Scanner(System.in);
            int accountId;
            double amount;

            while (true) {
                System.out.println("1. Create Account");
                System.out.println("2. Check Balance");
                System.out.println("3. Deposit");
                System.out.println("4. Withdraw");
                System.out.println("5. Exit");
                System.out.print("Choose an option: ");
                int option = scanner.nextInt();

                switch (option) {
                    case 1:
                        System.out.print("Enter account holder name: ");
                        String name = scanner.next();
                        accountId = bank.createAccount(name);
                        System.out.println("Account created with ID: " +
accountId);
                        break;
                    case 2:
                        System.out.print("Enter account ID: ");
                        accountId = scanner.nextInt();
                        double balance = bank.getBalance(accountId);
                        System.out.println("Current balance: " + balance);
                        break;
                    case 3:
                        System.out.print("Enter account ID: ");
                        accountId = scanner.nextInt();
                        System.out.print("Enter amount to deposit: ");
                        amount = scanner.nextDouble();
                        bank.deposit(accountId, amount);
                        System.out.println("Deposited: " + amount);
                        break;
                    case 4:
                        System.out.print("Enter account ID: ");
                        accountId = scanner.nextInt();
                        System.out.print("Enter amount to withdraw: ");
                        amount = scanner.nextDouble();
                        boolean success = bank.withdraw(accountId, amount);
                        if (success) {
                            System.out.println("Withdrawn: " + amount);
                        } else {
                            System.out.println("Insufficient funds.");
                        }
                        break;
                    case 5:
                        scanner.close();
                        System.exit(0);
                        break;
                    default:
                        System.out.println("Invalid option.");
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

// BankServer.java
package rmiexample;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.HashMap;
import java.util.Map;

public class BankServer extends UnicastRemoteObject implements Bank {
    private Map<Integer, BankAccount> accounts;
    private int nextAccountId;

    public BankServer() throws RemoteException {
        accounts = new HashMap<>();
        nextAccountId = 1; // Start account IDs from 1
    }

    @Override
    public synchronized int createAccount(String accountHolderName) throws
RemoteException {
        BankAccount newAccount = new BankAccount(accountHolderName);
        accounts.put(nextAccountId, newAccount);
        return nextAccountId++;
    }

    @Override
    public synchronized double getBalance(int accountId) throws RemoteException {
        BankAccount account = accounts.get(accountId);
        if (account != null) {
            return account.getBalance();
        }
        throw new RemoteException("Account not found");
    }

    @Override
    public synchronized void deposit(int accountId, double amount) throws
RemoteException {
        BankAccount account = accounts.get(accountId);
        if (account != null) {
            account.deposit(amount);
        } else {
            throw new RemoteException("Account not found");
        }
    }

    @Override
    public synchronized boolean withdraw(int accountId, double amount) throws
RemoteException {
        BankAccount account = accounts.get(accountId);
        if (account != null) {
            return account.withdraw(amount);
        }
        throw new RemoteException("Account not found");
    }

    public static void main(String[] args) {
        try {
            BankServer server = new BankServer();
            java.rmi.registry.LocateRegistry.createRegistry(1099);
            java.rmi.Naming.rebind("BankServer", server);
            System.out.println("Bank Server is running...");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

2. Instructions to Run

2.1 Setup

1. **Compile the Java Files**:

bash

Copy code

```
javac rmiexample/*.java
```

2. **Start the RMI Registry**: Open a terminal and run:

bash

Copy code

```
rmiregistry
```

3. **Start the Bank Server**: In another terminal, run:

bash

Copy code

```
java rmiexample.BankServer
```

4. **Run the Bank Client**: In a third terminal, run:

bash

Copy code

```
java rmiexample.BankClient
```

3. Documentation

3.1 System Architecture

The system consists of a client-server architecture using Java RMI (Remote Method Invocation). The server provides banking functionalities such as deposit, withdrawal, and balance inquiry through a remote interface (Bank). The client interacts with the server via the RMI framework, which handles communication and object serialization.

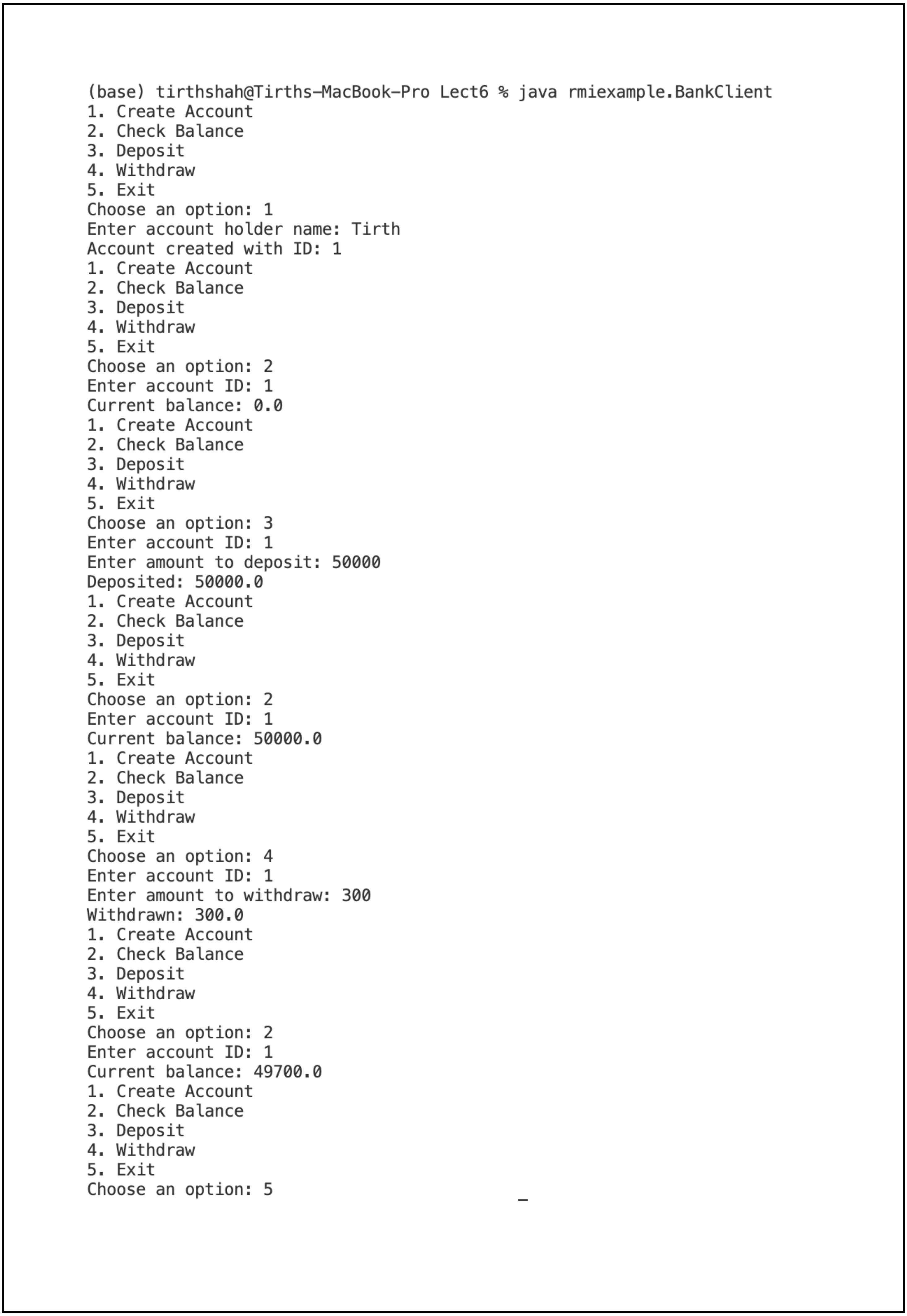
3.2 Design Decisions

- **Synchronized Methods**: The methods in the BankServer class are synchronized to ensure thread safety, preventing race conditions during concurrent access.
- **Remote Interface**: The Bank interface defines the methods that can be invoked remotely, ensuring a clear contract between the client and server.
- **Registry Port**: The default RMI registry port (1099) is used, which can be changed for different environments.

3.3 Flow of Operations

1. The client looks up the BankService in the RMI registry.
2. The client invokes methods like deposit and withdraw, which are handled by the BankServer.
3. The server updates the balance and returns the current balance to the client.

4. TestCase:



```
(base) tirthshah@Tirths-MacBook-Pro Lect6 % java rmiexample.BankClient
1. Create Account
2. Check Balance
3. Deposit
4. Withdraw
5. Exit
Choose an option: 1
Enter account holder name: Tirth
Account created with ID: 1
1. Create Account
2. Check Balance
3. Deposit
4. Withdraw
5. Exit
Choose an option: 2
Enter account ID: 1
Current balance: 0.0
1. Create Account
2. Check Balance
3. Deposit
4. Withdraw
5. Exit
Choose an option: 3
Enter account ID: 1
Enter amount to deposit: 50000
Deposited: 50000.0
1. Create Account
2. Check Balance
3. Deposit
4. Withdraw
5. Exit
Choose an option: 2
Enter account ID: 1
Current balance: 50000.0
1. Create Account
2. Check Balance
3. Deposit
4. Withdraw
5. Exit
Choose an option: 4
Enter account ID: 1
Enter amount to withdraw: 300
Withdrawn: 300.0
1. Create Account
2. Check Balance
3. Deposit
4. Withdraw
5. Exit
Choose an option: 2
Enter account ID: 1
Current balance: 49700.0
1. Create Account
2. Check Balance
3. Deposit
4. Withdraw
5. Exit
Choose an option: 5
```

```
(base) tirthshah@Tirths-MacBook-Pro Lect6 % java rmiexample.BankClient
1. Create Account
2. Check Balance
3. Deposit
4. Withdraw
5. Exit
Choose an option: 1
Enter account holder name: Rudra
Account created with ID: 2
1. Create Account
2. Check Balance
3. Deposit
4. Withdraw
5. Exit
Choose an option: 3
Enter account ID: 2
Enter amount to deposit: 500
Deposited: 500.0
1. Create Account
2. Check Balance
3. Deposit
4. Withdraw
5. Exit
Choose an option: 4
Enter account ID: 2
Enter amount to withdraw: 20.4
Withdrawn: 20.4
1. Create Account
2. Check Balance
3. Deposit
4. Withdraw
5. Exit
Choose an option: 2
Enter account ID: 2
Current balance: 479.6
1. Create Account
2. Check Balance
3. Deposit
4. Withdraw
5. Exit
Choose an option: 5
```

5. Conclusion

The RMI Banking System successfully demonstrates the use of Java RMI for remote method invocation, showcasing basic banking functionalities. The client-server architecture provides a clear separation of responsibilities, and the use of synchronized methods ensures thread safety. Future enhancements could include user authentication and transaction history management.

Comparison Report: RPC vs. RMI

Introduction

In the realm of distributed systems, Remote Procedure Call (RPC) and Remote Method Invocation (RMI) are two popular paradigms that facilitate communication between different processes, possibly running on different machines. While both technologies aim to enable remote communication, they have distinct characteristics, advantages, and limitations. This report provides a comparative analysis of RPC and RMI based on ease of implementation, language support, serialization, concurrency handling, and communication mechanisms.

---

1. Ease of Implementation

RPC

Setting up a distributed system using RPC is generally straightforward. RPC frameworks provide tools and libraries that abstract the complexities of network communication, allowing developers to define remote procedures with ease. The overall process usually involves defining the service interface in an IDL (Interface Definition Language), generating client and server stubs, and implementing the service logic. However, developers must handle error scenarios and network issues manually.

RMI

RMI simplifies the process of creating distributed applications in Java. The framework automatically handles much of the underlying communication, including object serialization and network management. The process involves defining remote interfaces, implementing server-side logic, and using RMI registry for binding remote objects. RMI is more intuitive for Java developers since it adheres to Java's object-oriented principles. Nonetheless, it requires a good understanding of Java's serialization mechanisms.

Strengths:

- RPC is language-agnostic, allowing for broader application across different programming languages.
- RMI integrates seamlessly with Java, making it easier for Java developers.

Weaknesses:

- RPC can be more cumbersome due to the need for manual stub generation and handling network issues.
- RMI's reliance on Java makes it less flexible in heterogeneous environments.

---

2. Language Support

RPC

RPC supports multiple programming languages, enabling developers to implement the server and client in different languages. Popular RPC implementations, like gRPC, facilitate cross-language communication by generating code stubs for various languages from a single service definition.

RMI

RMI is Java-specific, meaning that both the client and server must be implemented in Java. This limitation makes RMI unsuitable for environments where multiple programming languages are employed.

Strengths:

- RPC's multi-language support provides flexibility and broader compatibility in diverse environments.
- RMI's specialization allows for optimizations specific to Java's object model.

Weaknesses:

- RMI's Java constraint limits its applicability in multi-language projects.

---

3. Serialization

RPC

RPC typically relies on a binary or text-based serialization format to encode and decode data being transmitted between client and server. Developers can choose the serialization format that best suits their needs, such as JSON, XML, or Protocol Buffers. This flexibility allows for efficient data interchange but requires careful consideration of the chosen format's performance and compatibility.

RMI

RMI uses Java's built-in serialization mechanism to convert objects into byte streams for transmission. The process is automated; when an object is sent over the network, it is serialized and then deserialized on the receiving end. However, this can lead to performance overhead, especially when dealing with large object graphs, as every object must implement the Serializable interface.

Strengths:

- RPC allows for more control over serialization formats, which can optimize performance and compatibility.
- RMI simplifies the process with built-in Java serialization.

Weaknesses:

- RPC's flexibility can lead to inconsistencies if different serialization formats are used across components.
- RMI's reliance on Java serialization can introduce performance issues.

---

4. Concurrency

RPC

Concurrency in RPC is generally handled at the server level. The server can manage multiple client requests simultaneously using threading or asynchronous processing. However, developers must implement their concurrency control mechanisms to avoid race conditions and ensure data integrity.

RMI

RMI natively supports concurrency, allowing multiple clients to invoke methods on a remote object simultaneously. The RMI runtime handles concurrent requests, which simplifies development. However, developers need to manage shared resources explicitly to avoid conflicts.

Strengths:

- RMI provides built-in concurrency support, reducing the complexity of handling multiple client requests.
- RPC can be designed for high concurrency through custom implementations.

Weaknesses:

- RPC's concurrency handling can be cumbersome and error-prone without careful design.
- RMI's automatic concurrency handling may introduce contention issues if not managed properly.

---

5. Communication Mechanism

RPC

RPC typically utilizes lower-level communication protocols like TCP or UDP, making it suitable for both synchronous and asynchronous communication. The flexibility in choosing the transport layer allows RPC to be adapted for different network environments and performance needs.

RMI

RMI primarily uses TCP for communication, which ensures reliable message delivery. The RMI framework abstracts the communication details, allowing developers to focus on higherlevel logic. While this simplifies development, it also limits the flexibility of communication options.

Strengths:

- RPC offers flexibility in communication protocols, making it adaptable to various network conditions.
- RMI simplifies the communication model by relying on a well-defined protocol.

Weaknesses:

- RPC's flexibility can lead to complexity in implementation.
- RMI's reliance on TCP may limit performance in certain scenarios.

Conclusion

Both RPC and RMI serve vital roles in the development of distributed systems, each with its strengths and weaknesses. RPC's language-agnostic nature and flexibility in serialization and communication protocols make it a robust choice for heterogeneous environments. In contrast, RMI's seamless integration with Java provides a simplified and efficient framework for Java developers, though it comes with language constraints.

The choice between RPC and RMI ultimately depends on the specific requirements of the distributed application, including language preferences, performance needs, and the complexity of implementation.
