Lamport Clock Algorithm Implementation using Java Remote Method Invocation (Java RMI)

Rawat, Ankit Singh
Sucuzhanay Mora, Darwin Rolando

Introduction

The project is on the implementation of Lamport clock algorithm (developed by Leslie Lamport) using Java RMI aims to determine the order of events occurring in a distributed system.

Distributed systems are systems that consist of multiple independent computers that communicate and work together to perform a task on a network. One of the challenges in distributed systems is maintaining the synchronization of the clocks of the different computers, as the clocks may drift over time due to differences in hardware and software. This problem is often referred as problem of clock synchronization.

One solution to this challenge is the concept of logical clocks. The mechanism is used to capture chronological order and casual relationship between systems in a distributed environment. Using the concept of logical clocks, Leslie Lamport in 1978 introduced Lamport timestamp algorithm.

In this project, the Lamport clock algorithm was implemented using Java RMI (Remote Method Invocation), a Java API that allows programs to invoke methods on remote objects. Java RMI was chosen as the implementation language due to its ability to facilitate communication between distributed systems and its support for object-oriented programming.

The implemented system was tested to ensure that it accurately preserve the order of actions of multiple systems, and the performance of the system was evaluated. The results of the project demonstrate the effectiveness of Lamport clock algorithm for preserving the order of events.

Algorithm Description

Lamport clock algorithm is a method to maintain the order of occurring events using their chronological order and casual relationship. The algorithm uses messages sent between the computers to ensure that the logical clock values remain synchronized and does partial ordering on events (Not every pair of events needed to be compared). It is named after its creator, Leslie Lamport.

- Happened-before relation: a $\longrightarrow$ b

  This relation is the central concept with logical clocks which means event ‘a’ happened before event ‘b’.

- Logical Clock: (Ci/Cj is logical clock time at any timestamp i/j)
  - [C1]: Ci(a) < Ci(b) - If ‘a’ happened before ‘b’, then time of ‘a’ will be less than time of ’b’.
  - [C2]: Ci(a) < Cj(b) – Clock value of Ci(a) is less than Cj(b).

Algorithm:

- Each computer in the distributed system maintains a logical clock value, which is initially set to zero.
- Whenever an event occurs on a computer, the logical clock value is incremented by one.
- When a message is sent from one computer to another, the sender includes its current logical clock value in the message.
- When a computer receives a message, it compares the logical clock value in the message to its own logical clock value, takes the maximum from local and received logical clock values and increment it by 1.
- This process is repeated for each event and message in the system, ensuring that the events maintain the order of occurrence.

Pseudocode:

- *Logical_clock_Sen = 0 // Initialize local clock value to 0 (Sender).*
- *Logical_clock_Rec = 0 // Initialize local clock value to 0 (Receiver).*

- Sender side:

    - When any event occurred:
      Logical_clock_Sen = Logical_clock_Sen + 1; // Increment local logical clock value by 1

    - When a message is sent from one computer to another:
      send(message, Logical_clock_Sen); // Include local clock value in the message

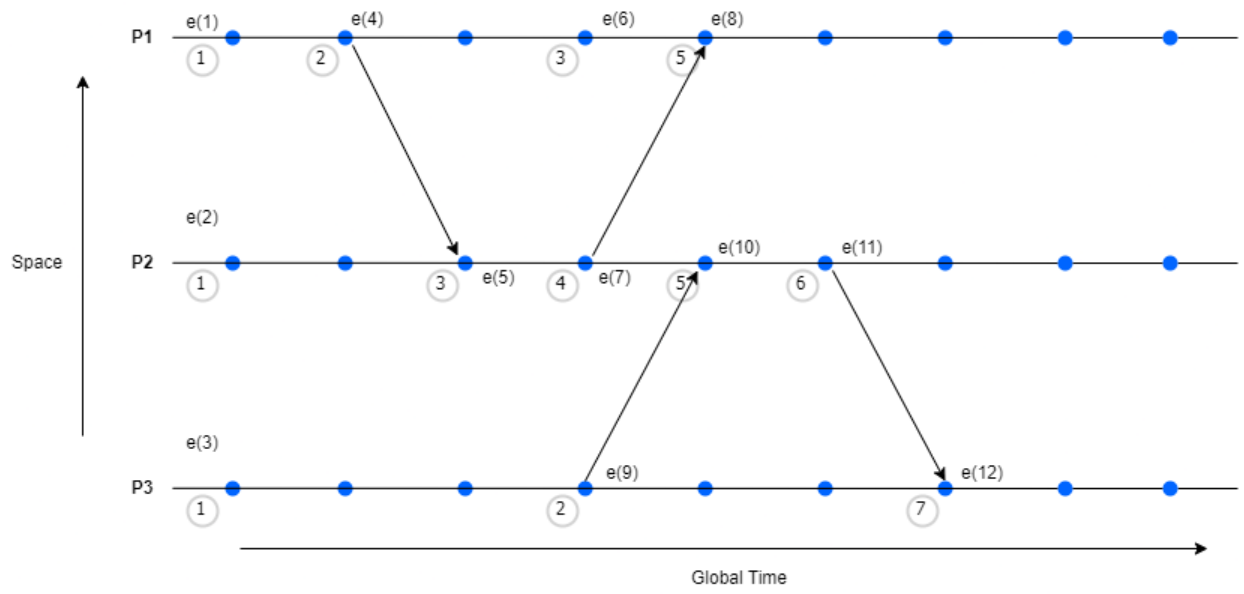
- Receiver Side:

    - When any event occurred:
      Logical_clock_Rec = Logical_clock_Rec + 1; // Increment local clock value by 1
    - When a computer receives a message:
      (message, Logical_clock_Sen) = receive()
      Logical_clock_Rec = max(Logical_clock_Rec, Logical_clock_Sen) + 1
      // Update local logical clock value to the maximum of the local and received logical clock values and increment local clock value by 1

Concurrent Events:

- The algorithm does a partial ordering, means it is not necessary to compare every event, so if two events can't be compared or do not have casual path, they are called concurrent.
- According to happened before:
  If event E1 $\longrightarrow$ E2, then logical timestamp of E1 < E2, however we cannot exactly tell that E1 happened before E2 or E1 and E2 are concurrent.
- To detect the concurrent events, concept of Vector clocks was introduced later.

Example

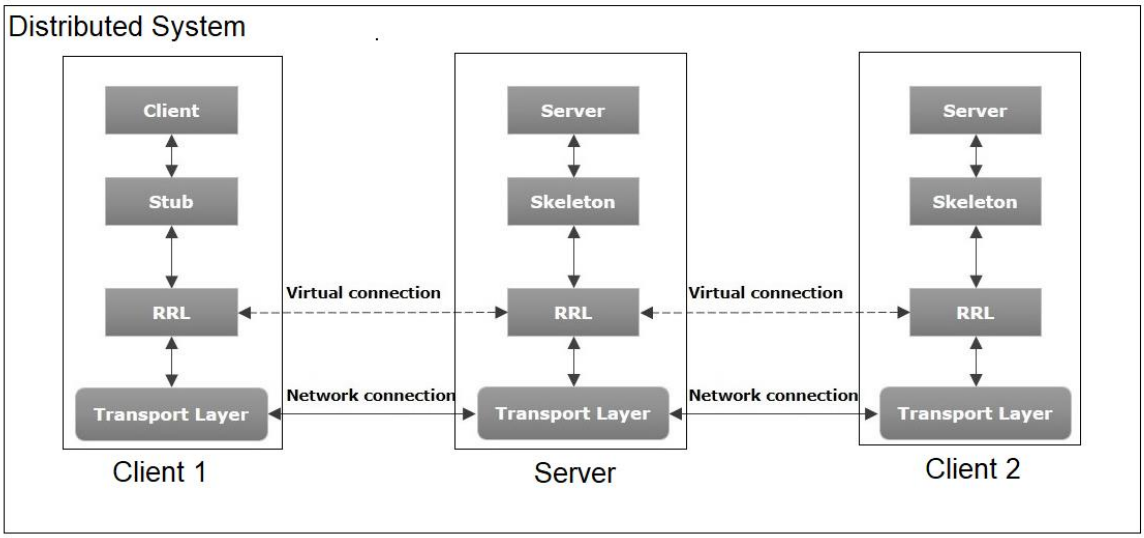


In the above diagram, there are 3 processes (P1, P2 and P3) having 12 events in total:

- In events e(1), e(2) and e(3) processes P1, P2 and P3 are initializing their logical clock time respectively to 0 and then 1.
- At e(4), P1 sent a message to P2 with its logical clock value and P2 received, compared for maximum value and incremented by 1 i.e.
  P1 – Send(message, 2)
  P2 – Receive(message, 2) -> max(2,2)+1 = 3
- After processing the request, P2 sent the response back including its logical clock value to P1. At P1 side:
  P2 – Send(message, 4)
  P1 – Receive(message, 4) -> max(3,4)+1 = 5
- Similar steps were followed by communication between process P3 and P2 from event e(9) to e(12).

Note: In P1, e(6) is not interacting with any other process and we cannot compare it with any other event as it will be concurrent. Like, by just checking the timestamps, we could conclude that e(6) in P1 has happened before e(10) in P2, but this isn't necessarily true.

System Architecture

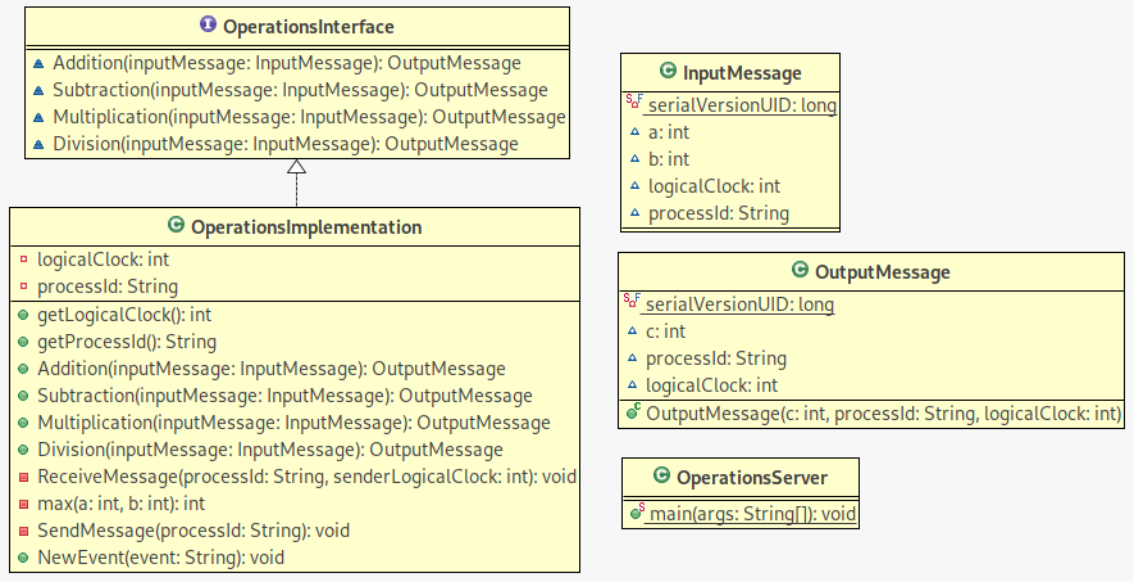


The system developed is based on a simple client server architecture where both clients 1 and 2 communicate with the server for any of the four operations from addition, subtraction, multiplication and division in a distributed system.

Implementation

The implementation was developed using Java RMI. On the server side, mathematical operations were created in order to be consumed by the clients. The server and clients have their own logical clock, internal events, and will send and receive messages. At all these events, the logical clock of each process (server or clients) will be calculated using the Lamport`s clock algorithm.

- ***Server Diagram Class***



In the server, the interface *OperationsInterface* defines the mathematical operations to be implemented.

```
public interface OperationsInterface extends Remote
{
    OutputMessage Addition(InputMessage inputMessage) throws RemoteException;
    OutputMessage Subtraction(InputMessage inputMessage) throws RemoteException;
    OutputMessage Multiplication(InputMessage inputMessage) throws RemoteException;
    OutputMessage Division(InputMessage inputMessage) throws RemoteException;
}
```

The class *OperationsImplementations*, in addition to implement the mathematical operations declared in the interface, we can find private methods and attributes that serve us to implement the algorithm, i.e., *ReceiveMessage, SendMessage, NewEvent, max, logicalClock, processId*.

```java
public OutputMessage Addition(InputMessage inputMessage)
{
    ReceiveMessage(inputMessage.processId, inputMessage.logicalClock);
    int c = inputMessage.a + inputMessage.b;
    NewEvent("Addition");
    SendMessage(inputMessage.processId);
    OutputMessage outputMessage = new OutputMessage(c, this.processId, this.logicalClock);
    return outputMessage;
}
```

```java
private void ReceiveMessage(String processId, int senderLogicalClock)
{
    NewEvent("Receive message");    // receive a message is an event
    int maxLogicalClock = max(this.logicalClock, senderLogicalClock);
    this.logicalClock = maxLogicalClock + 1;
    System.out.println("Message received from process: " + processId + " (logical clock:" + senderLogicalClock
            + ")\tLocal logical clock: " + logicalClock);
}
```

```java
private void SendMessage(String processId)
{
    NewEvent("Send message");   // send a message is an event
    System.out.println("Message send to process: " + processId + "\tLocal logical clock: " + this.logicalClock);
}
```

```java
public void NewEvent(String event)
{
    this.logicalClock++;
    System.out.println("Internal Event: " + event + " \tLocal logical clock: " + logicalClock);
}
```

The objects of the classes *InputMessage* and *OutputMessage* are the messages send and received between server and clients. In addition to the attributes needed for the mathematical operations, they have the attributes *logicalClock* and *processId* need for the algorithm implementation.

The main method of the class *OperationsServer* creates the registry, rebind the object with the operations implemented and initialize the logical clock of the server to 0.

```java
public static void main(String args[])
{
    // creating instance of implemented class
    OperationsImplementation operationsImplementation = new OperationsImplementation();
    System.out.println("Process: " + operationsImplementation.getProcessId() + "\tLocal logical clock: " +
            operationsImplementation.getLogicalClock());

    // important for the rmi registry location
    String ip = "127.0.0.1";
    System.setProperty("java.rmi.server.hostname", ip);

    Registry registry = null;

    try
    {
        int port = 10908;
        registry = LocateRegistry.createRegistry(port);
        operationsImplementation.NewEvent("Create registry");   // event create registry
    }
    catch (Exception ex)
    {
        System.out.println();
        System.out.println("Error: " + ex.toString());
    }
```

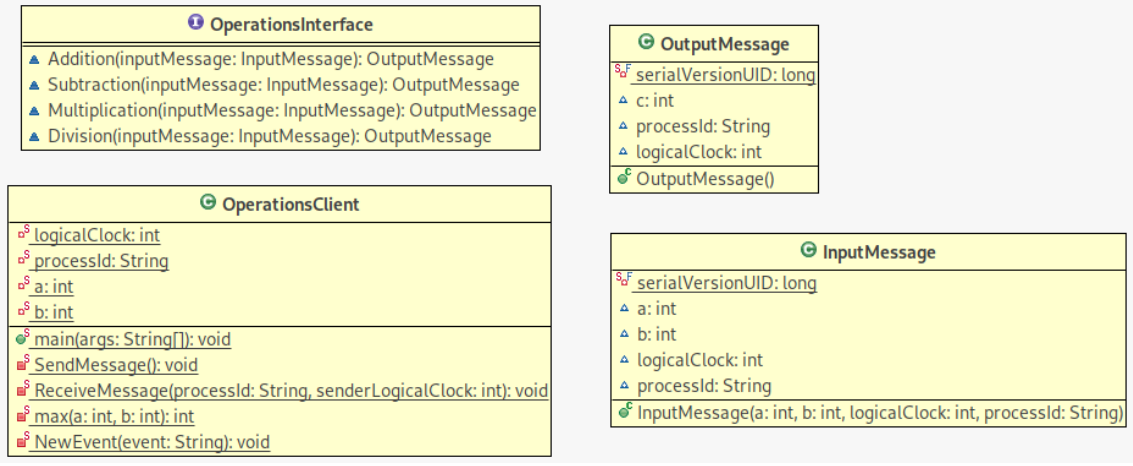
```java
    try
    {
        OperationsInterface skeleton = (OperationsInterface)UnicastRemoteObject.exportObject(operationsImplementation, 0);

        registry.rebind("Operations", skeleton);
        operationsImplementation.NewEvent("Rebind object"); // event rebind

        System.out.println("Server has been started.");
    }
    catch (Exception ex)
    {
        System.out.println("Error: " + ex.toString());
    }

    return;
}
```

- ***Client Diagram Class***



In the client, the interface *OperationsInterface* contains the definition of the methods to be consumed from the server.

```
public interface OperationsInterface extends Remote
{
    OutputMessage Addition(InputMessage inputMessage) throws RemoteException;
    OutputMessage Subtraction(InputMessage inputMessage) throws RemoteException;
    OutputMessage Multiplication(InputMessage inputMessage) throws RemoteException;
    OutputMessage Division(InputMessage inputMessage) throws RemoteException;
}
```

The classes *InputMessage* and *OutputMessage* have the same function as described in the server side.

The class *OperationsClient* has methods and attributes needed for the algorithm implementation, i.e., *ReceiveMessage, SendMessage, NewEvent, max, logicalClock, processId* (similar to the server). The main method of this class gets the registry, creates the stub, initializes the logical clock of the client to 0 and ask to the user for a mathematical operation.

```
private static int logicalClock = 0;
private static String processId = "P0";
private static int a = 4;
private static int b = 2;

public static void main(String args[])
{
    try
    {
        System.out.println("Process: " + processId + "\tLocal logical clock: " + logicalClock);

        String ip = "127.0.0.1";
        int port = 10908;
        Registry registry = LocateRegistry.getRegistry(ip, port);
        NewEvent("Get registry");       // event get registry

        OperationsInterface stub = (OperationsInterface)registry.lookup("Operations");
        NewEvent("Stub creation");      // event stub creation

        OutputMessage outputMessage = new OutputMessage();
        InputMessage inputMessage = new InputMessage(a, b, logicalClock, processId);

        System.out.println("Number 1: " + a + "\tNumber 2: " + b);
```

```
        while(true)
        {
            System.out.println("Choose an operation: 1 Addition | 2 Subtraction | 3 Multiplication | 4 Division");
            Scanner input = new Scanner(System.in);
            String msg = input.nextLine();
            //input.close();
            NewEvent("Choose operation");   // event ask for operation

            int operation = Integer.parseInt(msg);

            SendMessage();
            inputMessage.logicalClock = logicalClock;

            switch (operation)
            {
                case 1:
                    outputMessage = stub.Addition(inputMessage);
                    break;
                case 2:
                    outputMessage = stub.Addition(inputMessage);
                    break;
                case 3:
                    outputMessage = stub.Addition(inputMessage);
                    break;
                case 4:
                    outputMessage = stub.Addition(inputMessage);
                    break;
            }
```

```
            ReceiveMessage(outputMessage.processId, outputMessage.logicalClock);
            System.out.println("Operation answer: " + outputMessage.c);
        }
    }
    catch (Exception ex)
    {
        System.out.println("Err: " + ex.toString());
    }
}
```

Results

Process P1 Server

```
Process: P1     Local logical clock: 0
Internal Event: Create registry         Local logical clock: 1
Internal Event: Rebind object   Local logical clock: 2
Server has been started.
```

Process P0 Client 1

```
Process: P0     Local logical clock: 0
Internal Event: Get registry    Local logical clock: 1
Internal Event: Stub creation   Local logical clock: 2
Number 1: 4     Number 2: 2
Choose an operation: 1 Addition | 2 Subtraction | 3 Multiplication | 4 Division
1
Internal Event: Choose operation        Local logical clock: 3
Internal Event: Send message    Local logical clock: 4
Message send to process: P0     Local logical clock: 4
Internal Event: Receive message Local logical clock: 5
Message received from process: P1 (logical clock:7)     Local logical clock: 8
Operation answer: 6
Choose an operation: 1 Addition | 2 Subtraction | 3 Multiplication | 4 Division
```

Process P1 Server

```
Internal Event: Receive message         Local logical clock: 3
Message received from process: P0 (logical clock:4)     Local logical clock: 5
Internal Event: Addition        Local logical clock: 6
Internal Event: Send message    Local logical clock: 7
Message send to process: P0     Local logical clock: 7
```

Process P2 Client 2

```
Process: P2     Local logical clock: 0
Internal Event: Get registry    Local logical clock: 1
Internal Event: Stub creation   Local logical clock: 2
Number 1: 6     Number 2: 3
Choose an operation: 1 Addition | 2 Subtraction | 3 Multiplication | 4 Division
2
Internal Event: Choose operation        Local logical clock: 3
Internal Event: Send message    Local logical clock: 4
Message send to process: P0             Local logical clock: 4
Internal Event: Receive message Local logical clock: 5
Message received from process: P1 (logical clock:11)    Local logical clock: 12
Operation answer: 9
Choose an operation: 1 Addition | 2 Subtraction | 3 Multiplication | 4 Division
```

Process P1 Server

```
Internal Event: Receive message         Local logical clock: 8
Message received from process: P2 (logical clock:4)     Local logical clock: 9
Internal Event: Addition        Local logical clock: 10
Internal Event: Send message    Local logical clock: 11
Message send to process: P2     Local logical clock: 11
```

Table with the outputs from 3 processes P0, P1 and P2.

Event	Process 0 Logical Clock (Client 1)	Process 1 Logical Clock (Server)	Process 2 Logical Clock (Client 2)
P0, P1, P2: Initialization	0	0	0
P1: Create registry	0	1	0
P1: Rebind object	0	2	0
P0, P2: Get registry	1	2	1
P0, P2: Stub creation	2	2	2
P0: Choose operation	3	2	2
P0: Send message to P1	4	2	2
P1: Receive message from P0	4	3	2
P1: Lamport Algorithm	4	5	2
P1: Process operation	4	6	2
P1: Send message to P0	4	7	2
P0: Receive message from P1	5	7	2
P0: Lamport Algorithm	8	7	2
P2: Choose operation	8	7	3
P2: Send mesagge to P1	8	7	4
P1: Receive message from P2	8	8	4
P1: Lamport algorithm	8	9	4
P1: Process operation	8	10	4
P1: Send message to P2	8	11	4
P2: Receive message from P1	8	11	5
P2: Lamport Algorithm	8	11	12