



Jihočeská univerzita
v Českých Budějovicích
University of South Bohemia
in České Budějovice

LAMPORT'S LOGICAL CLOCK

Distributed Algorithm (Semester Project)

(2022 – 23)

Raviteja Anumula Naga

(anumur00@jcu.cz)

Siddhi Nitin Gunaji

(gunajs00@jcu.cz)

Table of Contents

Introduction.....	2
Algorithm.....	2
Implementation	2
LamportClock.java:	2
LamportClockClient.java:.....	3
LamportClockImplementation.java:	4
LamportClockServer.java:	5
UML Class Diagram	6
Observations	7
Summary:.....	7
Output:	8
Conclusion:	9
References.....	9

Introduction

Lamport's Logical Clock was created by Leslie Lamport. It is a procedure to determine the order of events occurring. It provides a basis for the more advanced Vector Clock Algorithm. Due to the absence of a Global Clock in a Distributed Operating System Lamport Logical Clock is needed. [1]

Lamport's logical clock is a mechanism for assigning timestamps to events in a distributed system in a way that partially orders the events. The idea behind the logical clock is to assign a timestamp to each event in the system, such that the timestamp reflects the causal relationship between events.

The basic idea is that each process in the system maintains its own logical clock, which it increments each time it performs an event. When a process sends a message to another process, it includes its current logical clock value in the message. The receiving process then updates its own logical clock to be greater than the maximum of its current value and the value received in the message before it performs the event associated with the message.

This way, the logical clock values assigned to events reflect the causal order of the events. If event A happens before event B, then the logical clock value of event A will be less than the logical clock value of event B.

Algorithm

- **Happened before relation(->):** $a \rightarrow b$, means 'a' happened before 'b'.
- **Logical Clock:** The criteria for the logical clocks are:
 - [C1]: $C_i(a) < C_i(b)$, [$C_i \rightarrow$ Logical Clock, If 'a' happened before 'b', then time of 'a' will be less than 'b' in a particular process.]
 - [C2]: $C_i(a) < C_j(b)$, [Clock value of $C_i(a)$ is less than $C_j(b)$] [1]

Implementation

The following code implements a distributed system using the Lamport Clock algorithm, which is used to maintain a logical clock value that reflects the relative order of events in the system. The system consists of a server and a client, which communicate with each other using Remote Method Invocation (RMI).

LamportClock.java:

```
import java.rmi.Remote;  
import java.rmi.RemoteException;  
  
// Remote interface for Lamport Clock algorithm  
public interface LamportClock extends Remote {
```

```

// Send a message with the given timestamp
void sendMessage(int timestamp) throws RemoteException;

// Get the current value of the logical clock
int getLogicalClock() throws RemoteException;
}

```

The **LamportClock** interface defines a remote interface for the Lamport Clock algorithm in a distributed system. It extends the **Remote** interface, which is required for any remote object in RMI.

The interface defines two methods:

- **void sendMessage(int timestamp):** This method is used by the client to send a message to the server, along with a timestamp that reflects the logical clock value at the time the message was sent.
- **int getLogicalClock():** This method is used by the client to request the current value of the server's logical clock.

Both methods declare that they may throw a **RemoteException**, which is a checked exception that indicates a communication-related error occurred during the execution of a remote method call. This exception must be declared in the throws clause of any method that uses RMI, since it may be thrown during the execution of the method.

The **LamportClock** interface is implemented by the **LamportClockImplementation** class, which provides an implementation of the two methods and maintains the current value of the logical clock. The interface is also used by the **LamportClockClient** class, which invokes the methods on a **LamportClock** remote object to send messages to the server and request the current value of the logical clock.

LamportClockClient.java:

```

import java.rmi.registry.*;

// Client program for Lamport Clock algorithm
public class LamportClockClient {
    public static void main(String args[]) {
        try {
            // Specify the IP address and port number of the RMI registry
            String ip = "127.0.0.1";
            int port = 8081;

            // Look up the LamportClock remote object in the RMI registry
            Registry reg = LocateRegistry.getRegistry(ip, port);
            LamportClock stub = (LamportClock) reg.lookup("LamportClock");

            // Send three messages to the server
            stub.sendMessage(0);
            stub.sendMessage(0);

```

```

        stub.sendMessage(0);

        // Get the current value of the logical clock
        int logicalClock = stub.getLogicalClock();
        System.out.println("Logical call value: " + logicalClock);
    } catch (Exception e) {
        System.out.println("Error: " + e.toString());
    }
}
}

```

The **LamportClockClient** class represents the client in a distributed system that uses the Lamport Clock algorithm to maintain a logical clock value that reflects the relative order of events in the system. It communicates with a server using Remote Method Invocation (RMI).

The **main** method of the **LamportClockClient** class does the following:

1. It specifies the IP address and port number of the RMI registry, which is used to look up remote objects.
2. It looks up the **LamportClock** remote object in the RMI registry using the **registry.lookup** method. This returns a stub (a local representative) of the remote object that can be used to invoke its methods remotely.
3. It invokes the **sendMessage** method on the **LamportClock** remote object three times, passing in the timestamps 0, 0, and 0 as arguments. This sends three messages to the server, each with a timestamp of 0.
4. It requests the current value of the logical clock using the **getLogicalClock** method and stores the result in a local variable **logicalClock**.
5. It prints the value of **logicalClock**.

The **main** method is surrounded by a try-catch block to handle any exceptions that may be thrown during the execution of the code.

LamportClockImplementation.java:

```

import java.rmi.RemoteException;

// Implementation of Lamport Clock algorithm
public class LamportClockImplementation implements LamportClock {
    // Local logical clock value
    private int logicalClock;

    public LamportClockImplementation() throws RemoteException {
        super();
        this.logicalClock = 0;
    }

    // Send a message with the given timestamp
    @Override
    public void sendMessage(int timestamp) throws RemoteException {

```

```

        // Update the logical clock value to the maximum of the current value and the received
        timestamp + 1
        this.logicalClock = Math.max(this.logicalClock, timestamp + 1);
    }

    // Get the current value of the logical clock
    @Override
    public int getLogicalClock() throws RemoteException {
        return this.logicalClock;
    }
}

```

It is a Java class that implements the **LamportClock** interface, which defines two methods: **sendMessage** and **getLogicalClock**. The **LamportClockImplementation** class provides an implementation for these methods and is used as the server in a distributed system that uses the Lamport Clock algorithm to maintain a logical clock value. The class has a private field **logicalClock** that stores the current value of the logical clock. It is initialized to 0 in the constructor of the class. The **sendMessage** method is used to receive a message from a client, along with a timestamp that reflects the logical clock value at the time the message was sent. The method updates the local **logicalClock** value using the received timestamp and the Lamport Clock algorithm, which states that the logical clock value should be set to the maximum of the current value and the received timestamp + 1. The **getLogicalClock** method is used to request the current value of the logical clock. It returns the value of the **logicalClock** field.

LamportClockServer.java:

```

import java.rmi.RemoteException;
import java.rmi.registry.*;
import java.rmi.server.*;

public class LamportClockServer{

    public static void main(String args[]) throws RemoteException {
        // Set up the RMI registry
        String ip = "127.0.0.1";
        System.setProperty("java.rmi.server.hostname", ip);
        Registry reg = null;
        int port = 8081;

        try {
            reg = LocateRegistry.createRegistry(port);

            catch (Exception ce) {
                System.out.println();
                System.out.println("Error: "+ce.toString());
            }

            LamportClockImplementation hi = new LamportClockImplementation();
            try {
                LamportClock stub = (LamportClock) UnicastRemoteObject.exportObject(hi, 0);
                reg.rebind("LamportClock", stub);
            }
        }
    }
}

```

```

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

```

This file contains the code for the RMI server program. It sets up an RMI registry on the specified IP address and port number, creates an instance of the `LamportClockImplementation` class, and exports it as a remote object. It then binds the remote object to the RMI registry using the binding name "LamportClock".

To run the RMI system, you will need to start the server program first, and then run the client program. The client will send three messages to the server using the `sendMessage` method, and the server's logical clock value will be updated each time a message is received.

The **LamportClockServer** class has a **main** method, which is the entry point of the program. It starts by setting up the RMI registry, which is a remote service that allows remote objects to be bound to a unique name and made available for remote method invocation. The IP address and port number of the RMI registry are specified in the code. The **LocateRegistry.createRegistry** method is used to create a new registry on the specified port.

Next, an instance of the **LamportClockImplementation** class is created. This class represents the implementation of the Lamport Clock algorithm, and it provides an implementation of the **LamportClock** interface, which defines the methods that can be remotely invoked by the client.

The **LamportClockImplementation** object is then exported as a remote object using the **UnicastRemoteObject.exportObject** method. This allows the object to receive remote method invocations from clients.

Finally, the remote object's stub (a local representation of the remote object) is bound to a unique name in the registry using the **rebind** method. This allows clients to look up the remote object in the registry using its name and invoke its methods remotely. The server prints a message to indicate that it has been started.

UML Class Diagram

In the UML Class diagram, the **LamportClock** interface defines the methods that can be called by a Client on a Server. The **LamportClock** implementation class contains the actual code for these methods and is used by the Server to execute them. The Server exports a remote object that implements the **LamportClock** interface, and the Client looks up this object in the RMI registry and invokes its methods using RMI. The Client and Server both have local instances of the **LamportClock** implementation class, which they use to keep track of their logical clock values and to send and receive messages with timestamps.

Here is a UML class diagram that represents the Lamport Clock algorithm:

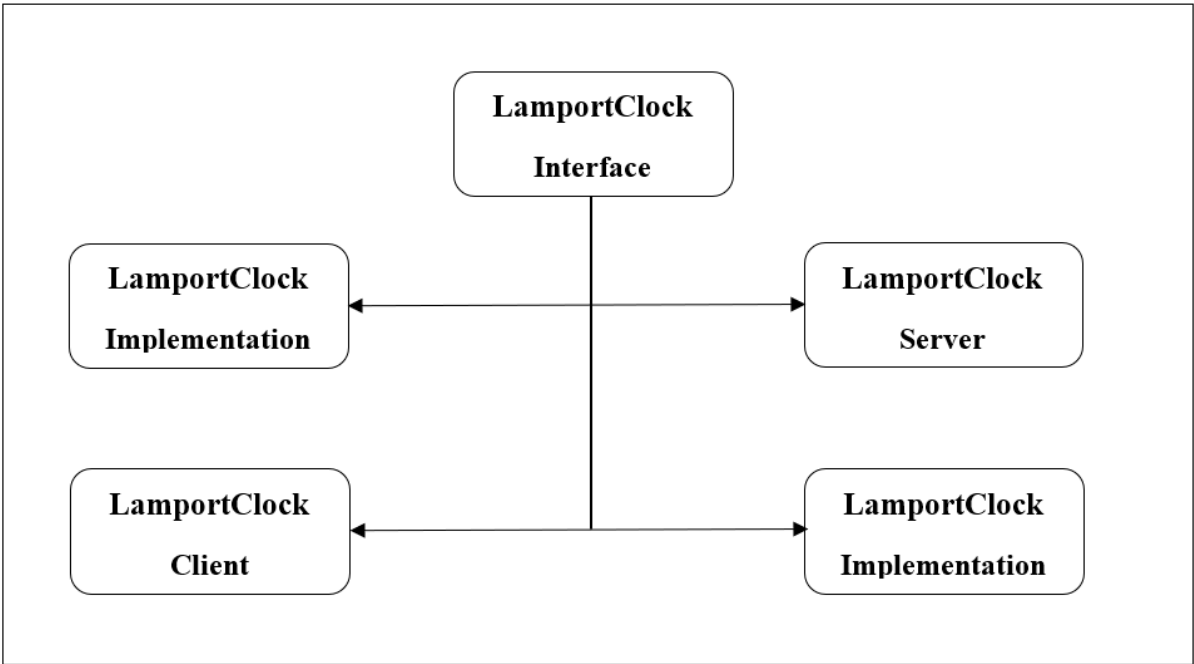


Table 1: UML Class Diagram

Observations

Summary:

The provided code implementation consists of three Java classes: LamportClock, LamportClockImplementation, and LamportClockServer. The LamportClock interface defines two methods: sendMessage and getLogicalClock. The LamportClockImplementation class is an implementation of the LamportClock interface and contains the logic for updating the logical clock value in response to messages received from the client. The LamportClockServer class sets up the RMI registry and binds the LamportClockImplementation object as a remote object.

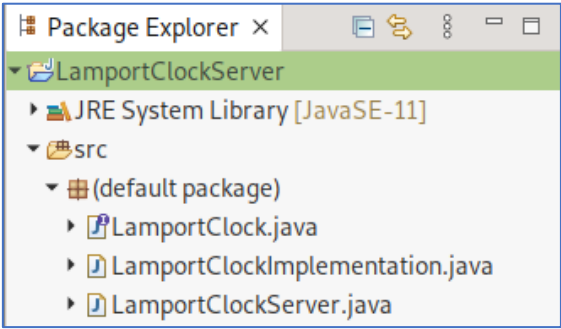


Table 2: Server Package

To use the Lamport Clock algorithm, the client program (LamportClockClient) connects to the RMI registry and looks up the LamportClock remote object. It then sends messages to the server using the sendMessage method and requests the current value of the logical clock using the getLogicalClock method.



Table 3: Client Package

Output:

- a) The output of the LamportClockServer program will be a message indicating that the server has been started.

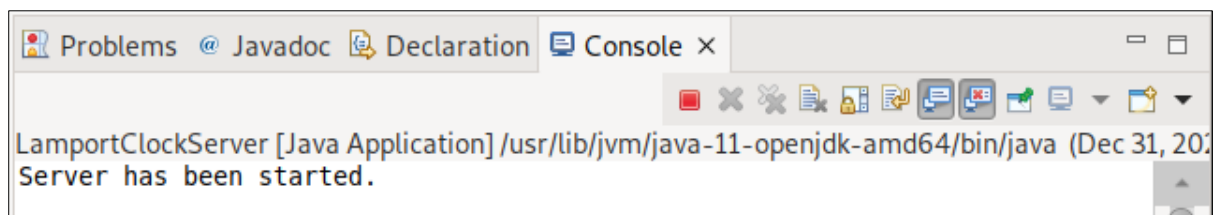


Table 4: LamportClockServer program output

- b) If the client program sends the following messages:

```
stub.sendMessage(0);
stub.sendMessage(0);
stub.sendMessage(0);
```

The output of the getLogicalClock method will be 1, since this is the maximum of the three timestamps + 1.

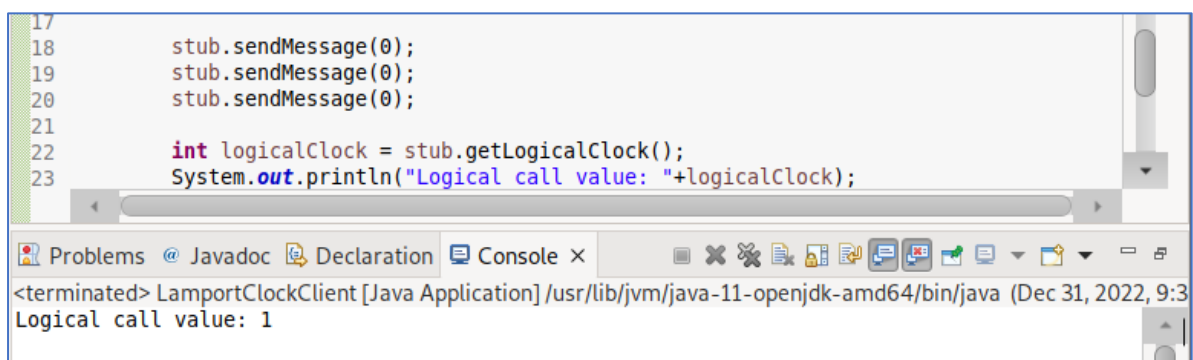
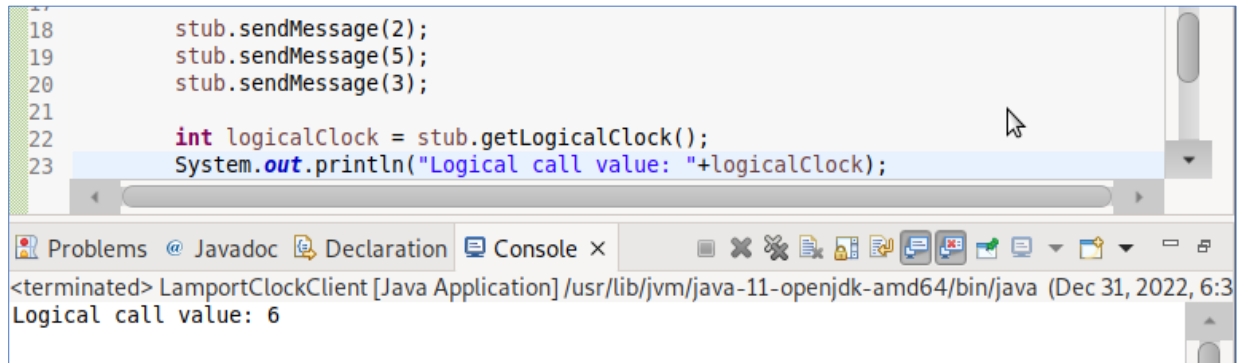


Table 5: Client program output

c) Similarly, If the client program sends the following messages:

```
stub.sendMessage(2);  
stub.sendMessage(5);  
stub.sendMessage(3);
```

Then, the output of the getLogicalClock method will be 6 since this is the maximum of the three timestamps + 1.

A screenshot of a Java IDE. The editor window shows a Java program with the following code:

```
18     stub.sendMessage(2);  
19     stub.sendMessage(5);  
20     stub.sendMessage(3);  
21  
22     int logicalClock = stub.getLogicalClock();  
23     System.out.println("Logical call value: "+logicalClock);
```

The console window at the bottom shows the output:

```
<terminated> LamportClockClient [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Dec 31, 2022, 6:3  
Logical call value: 6
```

Table 6: Client program output

Conclusion:

In conclusion, the Lamport's Clock algorithm is a useful technique for maintaining a logical clock value in a distributed system. It allows processes in the system to order events and maintain causality, which is important for maintaining consistency and correctness in the system.

References

[1] <https://www.geeksforgeeks.org/lamports-logical-clock/>

[2] "The Part-Time Parliament" by Leslie Lamport: This is the original paper that introduced the Lamport Clock algorithm and describes its use in distributed systems. It can be found here: <https://lamport.azurewebsites.net/pubs/lamport-paxos.pdf>

[3] "Lamport Timestamps" on Wikipedia: This Wikipedia article provides a summary of the Lamport Clock algorithm and its use in distributed systems. It can be found here: https://en.wikipedia.org/wiki/Lamport_timestamps