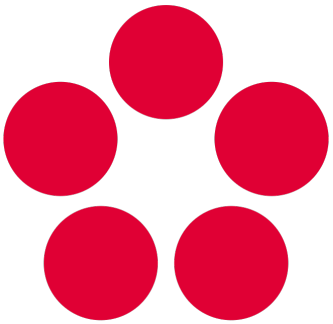


Jihočeská univerzita
v Českých Budějovicích
University of South Bohemia
in České Budějovice

Implementation of

LAMPORT'S ALGORITHM

Semester Project

DISTRIBUTED ALGORITHMS (UAI/503)

Submission by

GOKUL CHANDRABHANU NAMBIAR (B21445)
JOHN SARUN VARGHESE (B21463)

Table of Contents

Introduction:

In the process of concurrent execution of processes, each process needs to enter the critical section (or the section of the program which is shared across processes) at times for execution. It might so happen that because of the execution of multiple processes at once, the values stored in the critical section become inconsistent. In other words, the values depend on the sequence of execution of instructions – also known as a **race condition**.

The primary task of process synchronization is to get rid of race conditions while executing the critical section. This is primarily achieved through mutual exclusion. **Mutual exclusion** is a property of process synchronization which states that “no two processes can exist in the critical section at any given point of time”. The term was first coined by Dijkstra. Any process synchronization technique being used must satisfy the property of mutual exclusion, without which it would not be possible to get rid of a race condition. One such instance of a Mutual Exclusion solution is the algorithm we discuss here - **Lamport’s Algorithm**. Lamport's bakery algorithm is a computer algorithm devised by computer scientist Leslie Lamport.

Lamport envisioned a bakery with a numbering machine at its entrance so each customer is given a unique number. Numbers increase by one as customers enter the store. A global counter displays the number of the customer that is currently being served. All other customers must wait in a queue until the baker finishes serving the current customer and the next number is displayed. When the customer is done shopping and has disposed of his or her number, the clerk increments the number, allowing the next customer to be served. That customer must draw another number from the numbering machine in order to shop again.

Algorithm:

Lamport’s Algorithm:

- To enter Critical section:
  - When a process wants to enter the critical section, it sends a request message to all other processes and places the request on a queue.
  - When a process receives the request message from another process, it returns a timestamped reply message to the requesting process and places this request on the queue

- To execute the critical section:
  - A process can enter the critical section if it has received the message with timestamp larger than the timestamp from all other sites and also if its own request is at the top of the request queue

- To release the critical section:
  - When a process exits the critical section, it removes its own request from the top of its request queue and sends a timestamped release message to all other sites
  - When a process receives the timestamped release message from a process, it removes the request of that process from its request queue

Implementation of Lamport's Algorithm:

LamportServer.java

- For the Server setup to run our Lamport algorithm implementation, we set up the server hostname with our local machine IP Address.
- We assign a Port Number for using our Server through which the Client processes will be connecting for making requests and communicating with the Server.
- An instance of the ***LamportServerImpl*** class is used in the Server code for the purpose of binding with a registry
- A Java RMI Registry is created in our server, which is bound to the instance of the implementation class using the Port Number initiated above.
- The Server then sends a status to flag it is ready for operation.

LamportServerImpl.java

- The Server Implementation class serves as the method definition for all the relevant actions we have declared in the abstract interface ***Lamports***
- We have key functions that enable implementing Lamport's Algorithm here - lock(), unlock()
- Apart from these, we have our critical section, which for the sake of simplicity we have kept to simple baking operations that are detailed with three methods.
- The three critical section method for our code are - deposit() and withdraw().
- We initiate variables to hold the status of each incoming processes.
- The ***entering[]*** array holds the index of each process which requests to enter the critical section of the code.
- We also have the ***tickets*** list which holds the set of processes which are given a ticket based of our bakery algorithm, while waiting for its turn to be granted permission to execute.
- The list and array keep track of the process waiting and requesting critical section access using the respective ***processId*** of each thread we have initiated in the Client program, which serves as a unique identifier for each Client process.
- Method ***lock()*** used in the implementation further is detailed as:

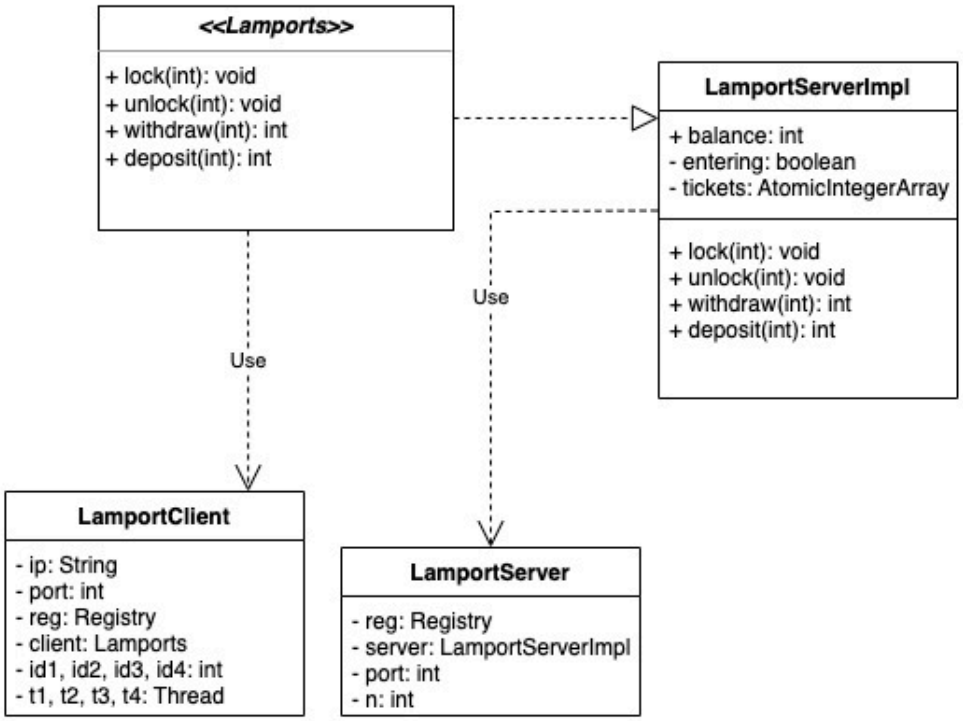
    - Finds the maximum ticket number of all existing processes.
    - Assigns next available ticket number to the current requesting process.
    - Holds execution of the process until it is the requesting processes' turn - both waiting for the process with ticket number in front to finish execution or until the process in front has a lower ticket number.
- Method ***unlock()*** executes once the thread finishes its execution of the required critical section code.
- It will release the hold on the critical section by resetting the tickets list which records the process holding the request ticket for execution.
- The critical section functions perform quick simple operations:
    - ***deposit()*** - adds to the existing balance variable a deposit amount being passed as parameter to the function.
    - ***withdraw()*** - deducts from the existing balance variable a withdrawal amount being passed as parameter to the function.

LamportClient.java

- Client request process here is simulated using multiple threads from the Client program.
- Each thread process is initiated with the IP Address and Port number for the Server we have initiated at our end for handling client requests.
- Using the IP Address and Port Number, each thread is able to locate the RMI Registry we have declared along with the Server Implementation.
- Having connected to the Registry, the thread is then able to invoke methods declared under the Server Implementation class ***LamportServerImpl***
- Each thread executes its respective 'run' block.
- Before and after each thread start invocation, we have methods that guide the mutual exclusion scenario for the critical section of our code, using the ***lock()*** and ***unlock()*** methods.
- The Lamport algorithm based methods ensure prevention of race conditions of the different thread processes during the runtime.

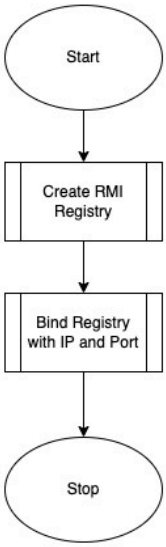
UML Diagram:

UML Class Diagram

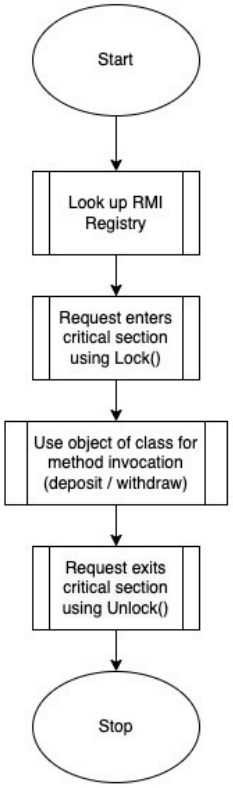


Flow Diagram:

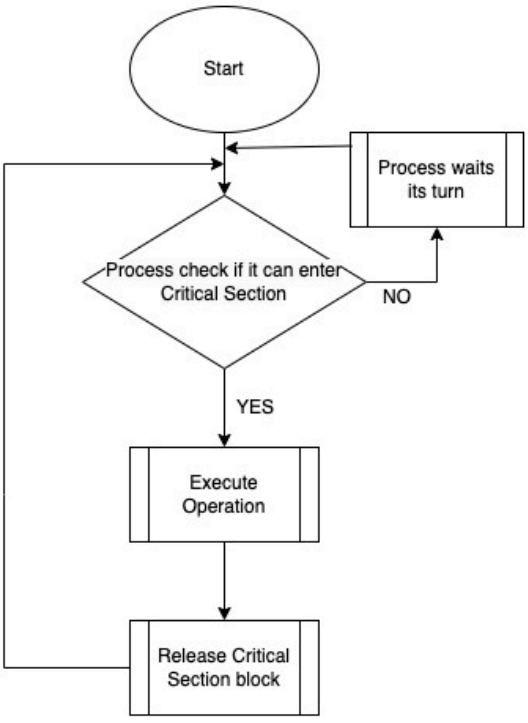
LamportServer.java



LamportClient.java



LamportServerImpl.java



Critical Section Demo:

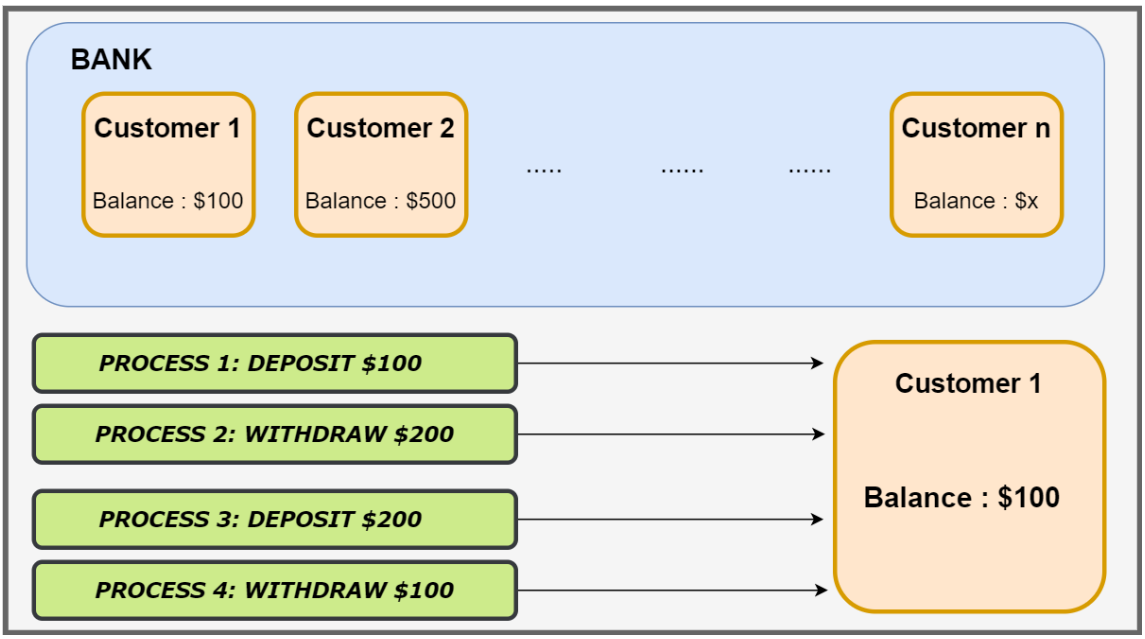


Fig 1. Banking Application Setup

We have an implementation for our Critical Section with two functions defined - withdraw() and deposit(). In this example, let us assume a bank allowing these operations, with 'n' customers, each having their own account balance as described in Fig 1 above.

With regards to the race condition in this example, we see that 'Customer 1' wants to perform four processes as shown in the Fig 1, simultaneously. In typical parallel processing instance, we can see that the processes being executed at random during runtime with no particular order to it, as shown in Fig 2. In this case, we can see that the account balance for the customer goes to a negative amount, which in a real world example would not be ideal for such a banking model.

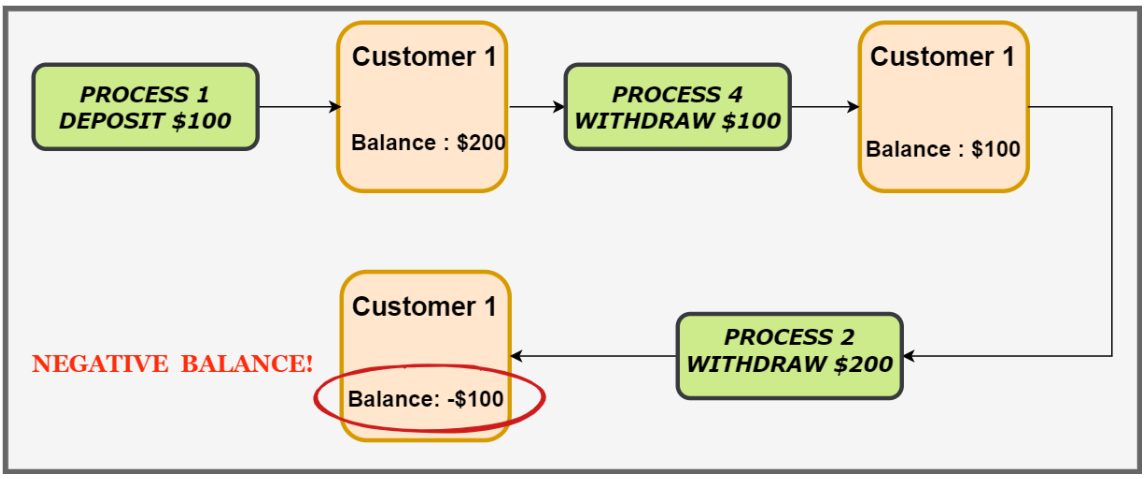


Fig 2. Simultaneous Critical Section execution without Lamport's Algorithm Implementation

In this scenario, we then apply Lamport's Algorithm for the process execution. The result would be as shown in Fig 3, where the processes are locked in an orderly fashion, which ensures that the logical integrity of the application is maintained. Such applications of the algorithm in real world applications help improve the stability and functionality of applications.

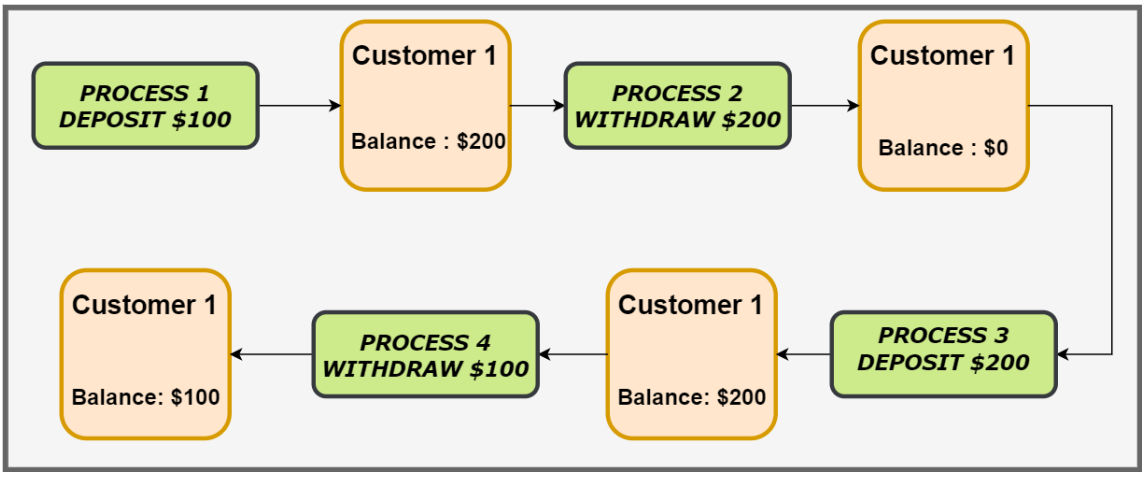


Fig 3. Simultaneous Critical Section execution with Lamport's Algorithm Implementation

Observations:

During the execution of our program, we have used threading for simulating multiple client requests. When we execute the threads in the order Thread 1 -> Thread 2 -> Thread 3 -> Thread 4, we see that the actual execution happens in a random order with each execution of the Client code. This pattern simulates the race condition we can see in a real-world scenario, where multiple clients attempt to access the same critical section block at the same time. We run the code both with and without the Lamport's Algorithm implementation. In our example, the shared resource that each thread attempts to access is the variable 'balance' in our bank.

```
// client.lock(id1); //LOCK PROCESS 1
t1.start(); //RUN THREAD 1
// client.unlock(id1); //UNLOCK PROCESS 1
// client.lock(id2); //LOCK PROCESS 2
t2.start(); //RUN THREAD 2
// client.unlock(id2); //UNLOCK PROCESS 2
// client.lock(id3); //LOCK PROCESS 3
t3.start(); //RUN THREAD 3
// client.unlock(id3); //UNLOCK PROCESS 3
// client.lock(id4); //LOCK PROCESS 4
t4.start(); //RUN THREAD 4
// client.unlock(id4); //UNLOCK PROCESS 4
```

Fig 4. Code Snippet (without Lamport's Algorithm Implementation)

```
z:\University of South Bohemia\Winter Semester\Distribut
\Lamport Final" && cmd /C ""C:\Program Files\Java\jre1.8
a\jdt_ws\Lamport Final_fe6761ca\bin" LamportClient "
Thread 1: : New balance: 200
Thread 4: : New balance: 100
Thread 2: : New balance: -100
Thread 3: : New balance: 100
```

Fig 5. Client Output (without Lamport's Algorithm Implementation)

Fig 4 and 5 describe the code snippet and the respective output for out Client side code, where we have not implemented Lamport's Algorithm.

```
client.lock(id1); //LOCK PROCESS 1
t1.start(); //RUN THREAD 1
client.unlock(id1); //UNLOCK PROCESS 1
client.lock(id2); //LOCK PROCESS 2
t2.start(); //RUN THREAD 2
client.unlock(id2); //UNLOCK PROCESS 2
client.lock(id3); //LOCK PROCESS 3
t3.start(); //RUN THREAD 3
client.unlock(id3); //UNLOCK PROCESS 3
client.lock(id4); //LOCK PROCESS 4
t4.start(); //RUN THREAD 4
client.unlock(id4); //UNLOCK PROCESS 4
```

Fig 6. Code Snippet (with Lamport's Algorithm Implementation)

Fig 6,7 and 8 describe the code snippet and the respective output from Client side and Server side code, where we have implemented Lamport's Algorithm.

```
Z:\University of South Bohemia\Winter Semester\Distrib
\Lamport Final" && cmd /C ""C:\Program Files\Java\jre1
a\jdt_ws\Lamport Final_fe6761ca\bin" LamportClient "
Thread 1: : New balance: 200
Thread 2: : New balance: 0
Thread 3: : New balance: 200
Thread 4: : New balance: 100
```

Fig 7. Client Output (with Lamport's Algorithm Implementation)

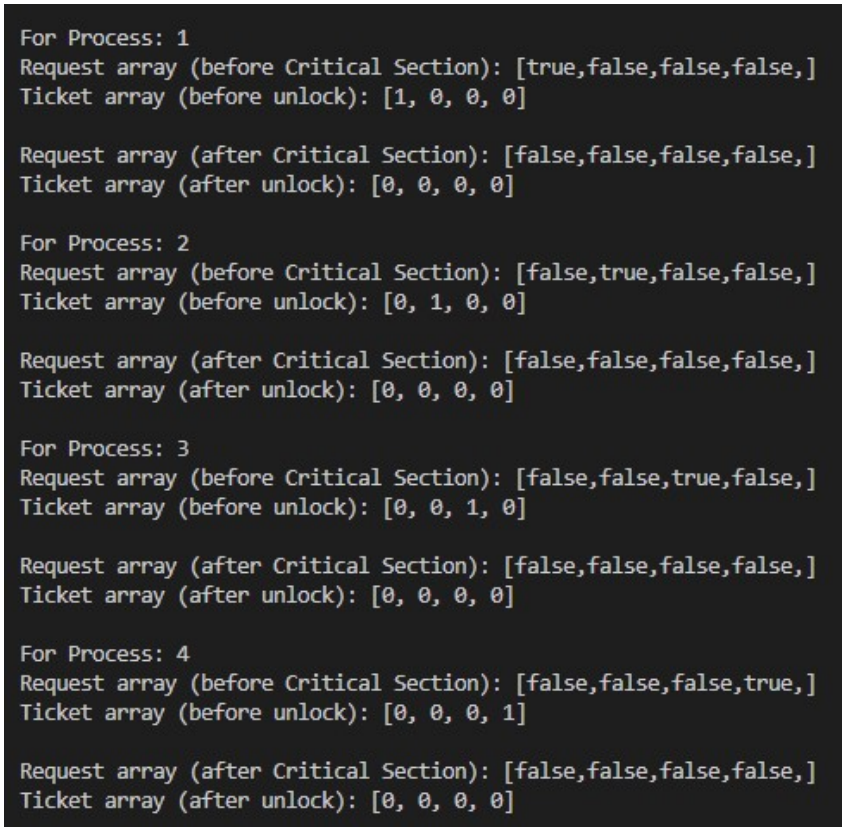


Fig 8. Server Output (with Lamport's Algorithm Implementation)

Conclusion:

We see that when we execute with the Lamport's Algorithm implemented, the lock() and unlock() method holds and releases each process requesting access to the critical section, in a way that ensures the process executes in a FIFO manner with the next process executing only once the process before it finishes its code block execution. This in contrast to the execution without the algorithm implemented, where we see that the critical section methods are invoked in a random order which may sometimes depending on the program logic, be risky in a real-world scenario. WIth the code output seen above, we see that the with the algorithm implemented, the process execution happens in an orderly fashion, and the run without the algorithm implementation, the balance variable even goes to a negative value, as the process invocation is in random state.

References:

- Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7), 558-565.
- Attiya, H., & Welch, J. (2004). Distributed computing: fundamentals, simulations, and advanced topics (2nd ed.). John Wiley & Sons.
- Lynch, N. A. (1996). Distributed algorithms (Vol. 6). San Francisco, CA: Morgan Kaufmann Publishers Inc.