

# **CSCI 53700: Distributing Computing**

## **Assignment Number 4**

**Date: November 29, 2018.**

**Submitted by: Anushka H Patil**

**Aim:** This assignment is intended to re-enforce the principles of remote method invocations using the Java-RMI model of distributed-object computing. To re-implement the first assignment (Simulation of Berkley's Algorithm and Lamport Clocks) using Java-RMI. Run the distributed system for a substantial period. Periodically (and at the end of the execution) compare the values of the logical clocks of POs. Print out the clock drifts for all POs. Vary the values of probabilities (used for inter-PO communication) and re-execute the system. Analyze the results (as reflected by the values of logical clocks and clock drifts) as a function of these probabilities.

## RMI, Remote Method Invocation:

It is a mechanism that allows an object residing in one machine to access/invoke an object running on another machine. It provides remote communication between Java programs.

Package: java.rmi.

RMI Implementation:

- 1) Client Program: It requests the remote objects on the server and tries to invoke its methods.
- 2) Server Program: It creates a remote object and reference of that object is made available for the client
- 3) Stubs: A stub is a representation (proxy) of the remote object at client. It resides in the client system; it acts as a gateway for the client program.
- 4) RMI Registry: RMI registry is a namespace on which all server objects are placed. Each time the server creates an object, it registers this object with the RMI registry using `bind()`. To invoke a remote object, the client needs a reference of that object. At that time, the client fetches the object from the registry using its bind name (using `lookup()` method).

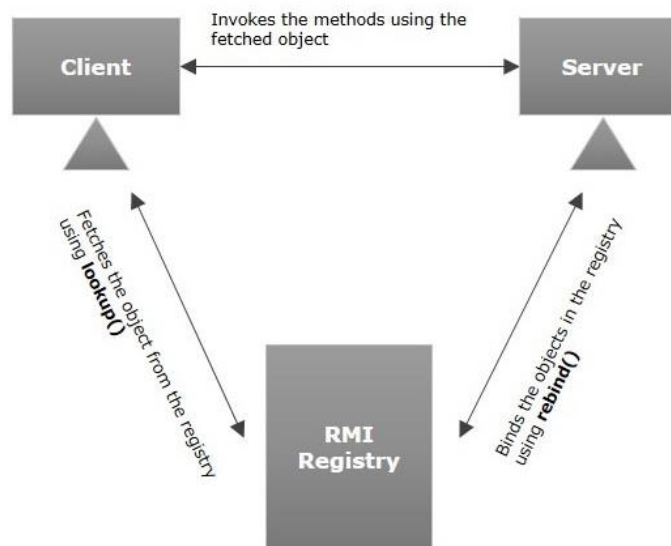


Figure 1. RMI Registry

source: [https://www.tutorialspoint.com/java\\_rmi/java\\_rmi\\_introduction.html](https://www.tutorialspoint.com/java_rmi/java_rmi_introduction.html)

In our case, it is more of like peer-peer communication wherein processes communicate with each other along with the master. Thus, in our case every process object binds itself in the registry using the `Naming.bind()` function. They fetch the other process objects and master object from the registry by performing a `lookup()` action. In this manner, I have implemented the RMI for inter-process communication.

## Interaction Model:

- 1) Synchronous Model: It based on assumptions and certain constraints about time, order of events and nature of the system.
- 2) Asynchronous Model: It is more suitable for the real world scenarios as it doesn't rely on any assumptions about the time and order of events in a distributed system.

For my design, I have made use of asynchronous model, which isn't based on any assumptions associated with time or event of order. My model consists of 5 parts:

- 1) Simulation
- 2) Process
- 3) Master
- 4) Message
- 5) RMIIInterface

### Simulation.java:

This file is responsible for creating a distributed simulation. This file receives two extra arguments from the user: *ObjectType* and *ObjectId*

- *ObjectType*: 0 represents Process Object and 1 represents Master Object
- *ObjectId*: It is a unique id for each object

### Process.java:

**RMI Part:** Based on the *objectId* it creates stub for each process. This stub is then bind along with an interface using the *Naming.bind()* function. Thus, now we have a remote process object associated with a stub name using the Naming class's bind. In this manner, every process has its remote object registered with the RMI registry.

The processes then can call other process by performing a lookup and obtain its reference to invoke the remote method *putMessage()*. To maintain the references of the processes I have used made use of a *processList*.

- Process 1 -- machine 1 (in-csci-rrpc01.cs.iupui.edu - 10.234.136.55)
- Process 2 -- machine 2 (in-csci-rrpc02.cs.iupui.edu - 10.234.136.56)
- Process 3 -- machine 3 (in-csci-rrpc03.cs.iupui.edu - 10.234.136.57)
- Process 4 -- machine 4 (in-csci-rrpc04.cs.iupui.edu - 10.234.136.58)

**Remote method:** *putMessage()* it has two parameters *time* and *sender*. *time* is the logical time of the sender, *sender* is the sender's id. The *putMessage()* creates a message object using these two parameters and adds it to the respective queue of the processes and master.

**Clock Functionality:** The *runProcess()* function calls 3 other functions: *sendMessage()*, *receiveMessage()* and *internalEvent()*. I have used switch case to decide the function for each iteration, a random number is created at run time and it makes this decision.

- 1) *sendMessage()*: This function is responsible for sending messages. It sends an offset to Master Object when 't' counter time is passed. It uses this function to send messages to other process. A remote method *putMessage()* is invoked to *sendMessage*. Every time it sends a message it increments its logical clock.
- 2) *receiveMessage()*: This function is responsible for receiving messages. If the sender is a master object, it adjusts its clock using the offset. If it receives message from other process, it uses the concept of lamport to adjust its clock. "Happened Before Relation"
- 3) *internalEvent()*: This function is independent and doesn't affect the working of any other processes

### **Master.java:**

**RMI Part:** Like the process, master also creates a stub using the *objectId*, in case of master the *objectId* is 5. It also maintains the reference of the process using *processList*.

- Master -- machine 5 (in-csci-rrpc05.cs.iupui.edu - 10.234.136.59)

**Remote method:** *putMessage()* it has two parameters *time* and *sender*. *time* is the logical time of the sender, *sender* is the sender's id. The *putMessage()* creates a message object using these two parameters and adds it to the respective queue of the processes and master.

**Clock Functionality:** The *runMaster()* function calls 3 other functions: *sendMessage()*, *receiveMessage()* and *internalEvent()*. For every iteration, if conditions fulfilled all the functions run.

- 1) *receiveMessage()*: This function is responsible for receiving messages. Master object receives messages only from the process. Once, it receives the logical clock, it uses an arraylist *mastertimeclock* to store the logical clocks. Based on the concept of Lamport it then adjusts its logical clock.
- 2) *sendMessage()*: This function is responsible for executing the Berkley's Algorithm. It uses the *mastertimeclock* arraylist to maintain the logical clocks for each process. Once, all the processes send their logical time, it computes the average and later the offset. It sends offset to each process to update their logical clock by invoking the *putMessage()*. It then updates, its own logical clock. If the condition i.e. all logical clocks are obtained isn't satisfied it doesn't compute the offset. I have made use of a Boolean function *check()* to check whether all process have send their logical clock.
- 3) *internalEvent()*: This function is independent and doesn't affect the working of any other processes

### **Message.java:**

Message has two parameters: *time*, *sender*. This is used to maintain the logical clock and sender information for a message.

### **RMIInterface.java:**

It is an interface that declares the *putMessage()* methods that may be invoked from a remote machine.

### **Failure Model:**

Byzantine or arbitrary failures: To exhibit this, I have made use of an if statement which checks whether the counter is divisible by a random number that is generated at run time. If this condition is satisfied it reduces the value of logical clock to half. This is how byzantine failure occurs in this model.

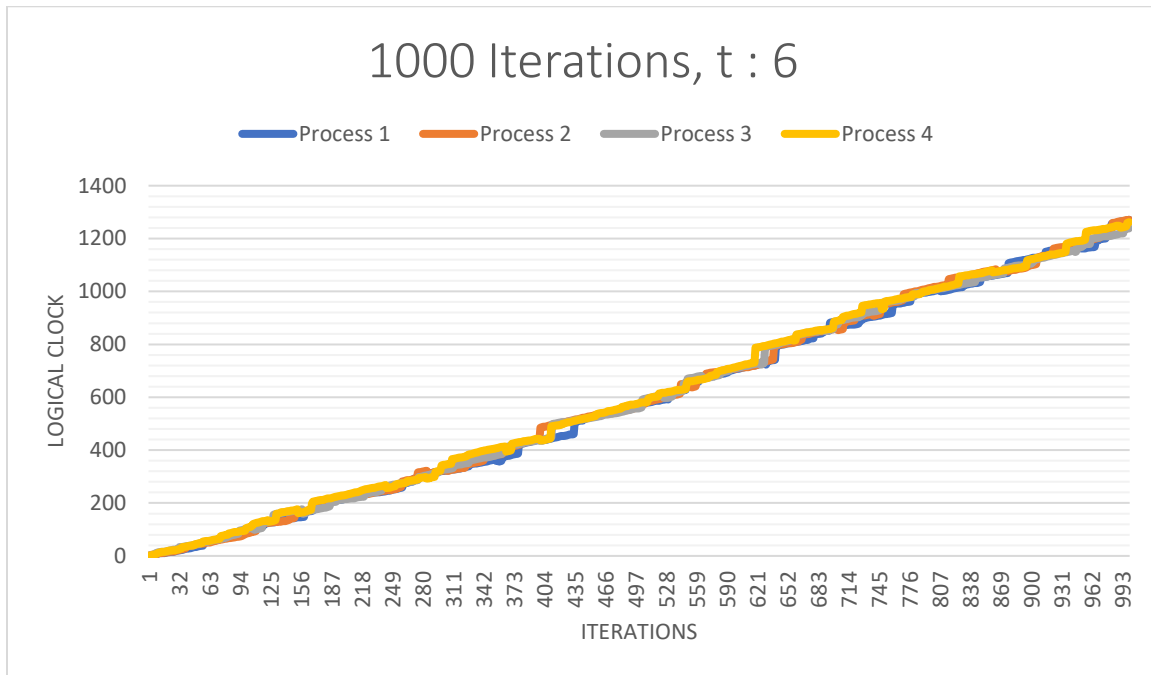
### **Security:**

Encryption and Decryption using same key. When the Message is send the time parameter is encrypted using the key and when the message is received it is decrypted using the same key.

- $\text{time} + \text{key} \rightarrow \text{Encryption}$
- $\text{time} - \text{key} \rightarrow \text{Decryption}$

## Graphs

I ran the processes for 1000 iteration with time set to 6. After every 6 events the process sends its time to the master. The master on reception of times from all the process computes an offset for each process. On reception of this offset the process adjust their clocks.

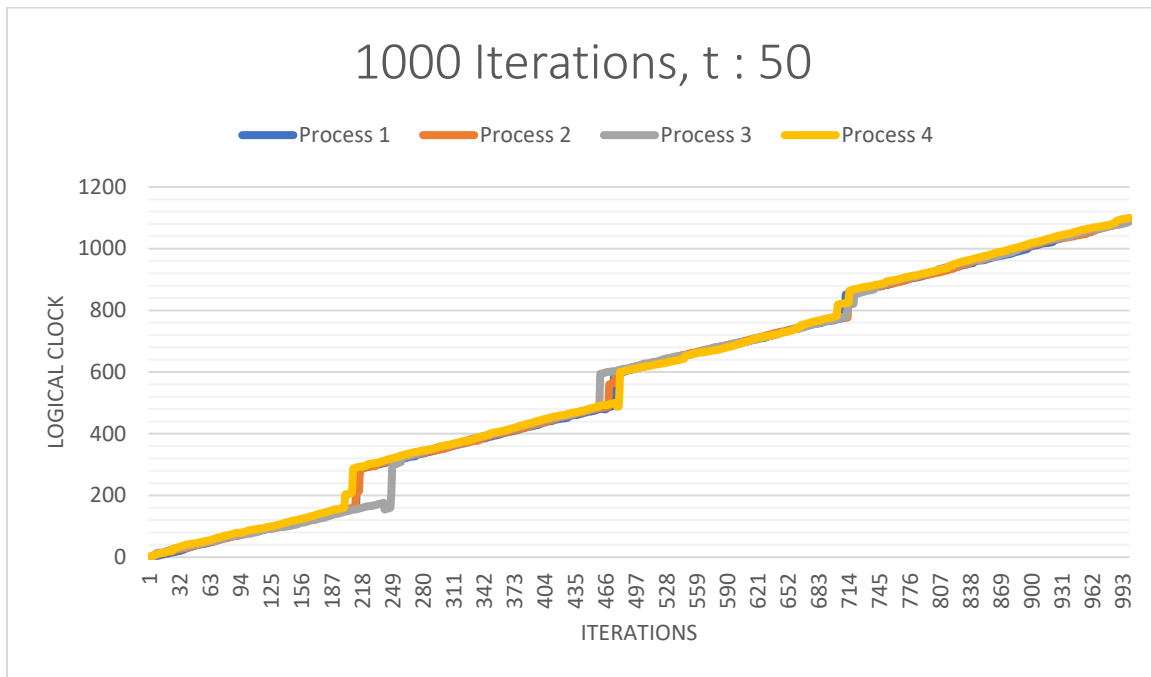


Graph 1: 1000 Iterations, time: 6

In this case, I had commented the Byzantine or arbitrary failures section to show that no failure occurred while this was in process. Here, we can see how the clock consistency and drift takes place.

The  $t$  value was set to 6, but you might notice that the master object adjusting the clocks varies. This is because for every process we randomly select the event. The probability for selecting a receive event varies for each of the 4 process. Thus, every process receives the offset at different iteration and adjust it likewise.

I ran the processes for 1000 iteration with time set to 50. After every 50 events the process sends its time to the master. The master on reception of times from all the process computes an offset for each process. On reception of this offset the process adjust their clocks.

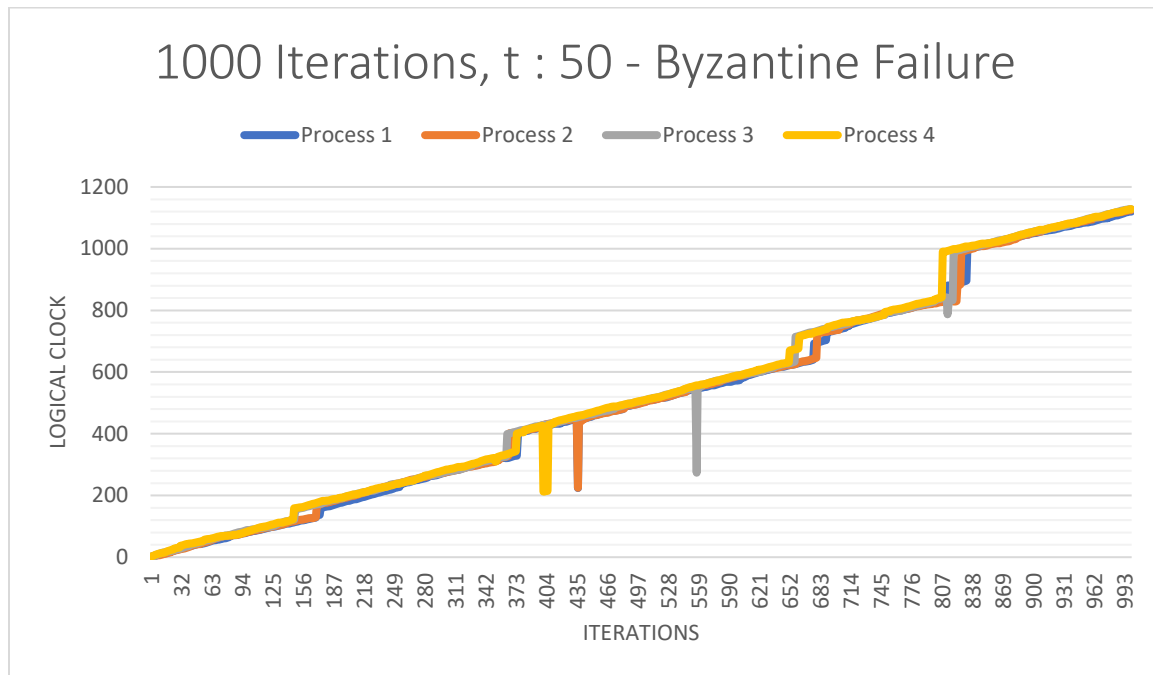


Graph 2: 1000 Iterations, time: 50

In this case, I had commented the Byzantine or arbitrary failures section to show that no failure occurred while this was in process. Here, we can see how the clock consistency and drift takes place.

The  $t$  value was set to 50, but you might notice that the master object adjusting the clocks varies. This is because for every process we randomly select the event. The probability for selecting a receive event varies for each of the 4 process. Thus, every process receives the offset at different iteration and adjust it likewise. Here, we can notice that when we ran it for  $t$  value set to 6, it converged more often than  $t$  value at 50.

I ran the processes for 1000 iteration with time set to 50. After every 50 events the process sends its time to the master. The master on reception of times from all the process computes an offset for each process. On reception of this offset the process adjust their clocks.



Graph 3: 1000 Iterations, time: 50 with Byzantine Failure

In this case, Byzantine or arbitrary failures occurred. Here, we can see how the clock consistency is affected and drift takes place when byzantine failure occurs in the system. In, this case the byzantine failure occurs 4 times:

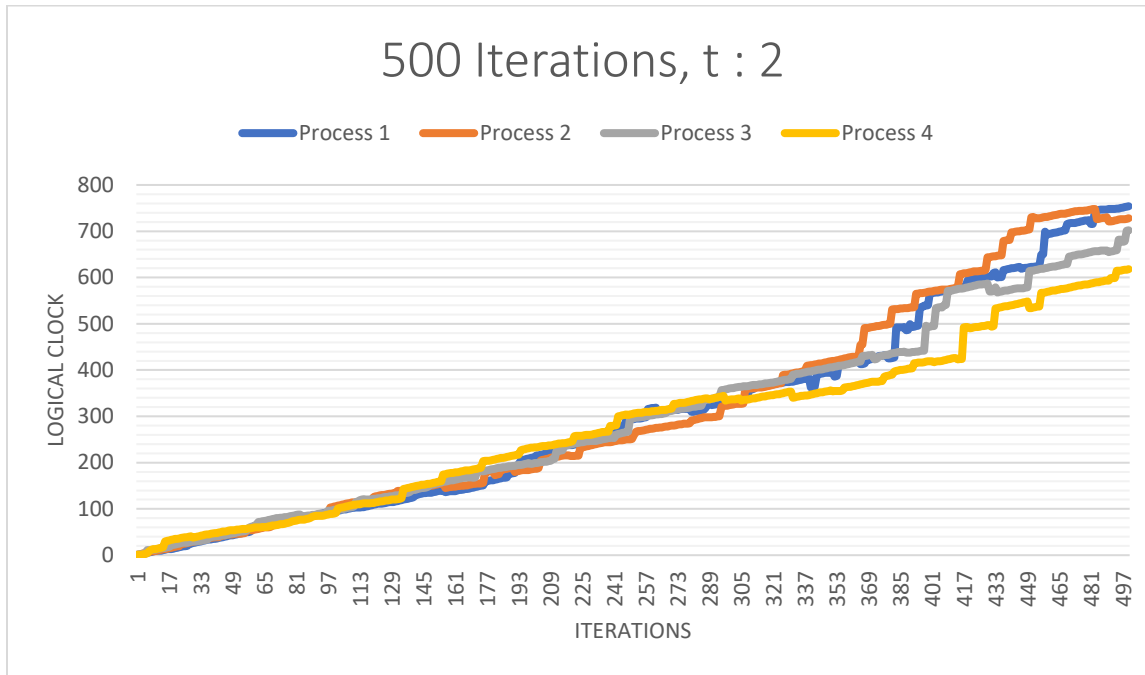
- Process 1: Iteration 435
- Process 2: Iteration 435
- Process 3: Iteration 559
- Process 4: Iteration 404

In my case, byzantine failure occurs at receiving end and it reduces the logical value by half, every time the failure is caused. I have kept it half to properly show the failure in the system. In normal cases byzantine failure doesn't reduce half it could change the logical clock to anything. Thus, these failures are displayed clearly in the graph and show how they change the clock consistency.

We can notice that after every interval of time 50, the processes send their time and on reception of time from all the processes the master computes and average and an offset with respect to the average and logical times of each process. The processes then adjust its clock. Thus, we can notice that even after byzantine failure the process are synced after the adjustment.



I ran the processes for 500 iteration with time set to 2. After every 2 events the process sends its time to the master. The master on reception of times from all the process computes an offset for each process. On reception of this offset the process adjust their clocks.



In this case, I had commented the Byzantine or arbitrary failures section to show that no failure occurred while this was in process. Here, we can see how the clock consistency and drift takes place.

The  $t$  value was set to 2, but you might notice that the master object adjusting the clocks varies. This is because for every process we randomly select the event. The probability for selecting a receive event varies for each of the 4 process. Thus, every process receives the offset at different iteration and adjust it likewise. Since I have ran it for 500 iterations, it gives a better vision of master adjustments.

The Probability for selecting the events for 1000 iteration is as follows

Process	Send Event	Receive Event	Internal Event	Byzantine Failure
1	0.332	0.328	0.343	0.040
2	0.306	0.323	0.341	0.030
3	0.325	0.319	0.346	0.010
4	0.342	0.314	0.304	0.040

I ran the processes for 1000 iterations to obtain the probabilities

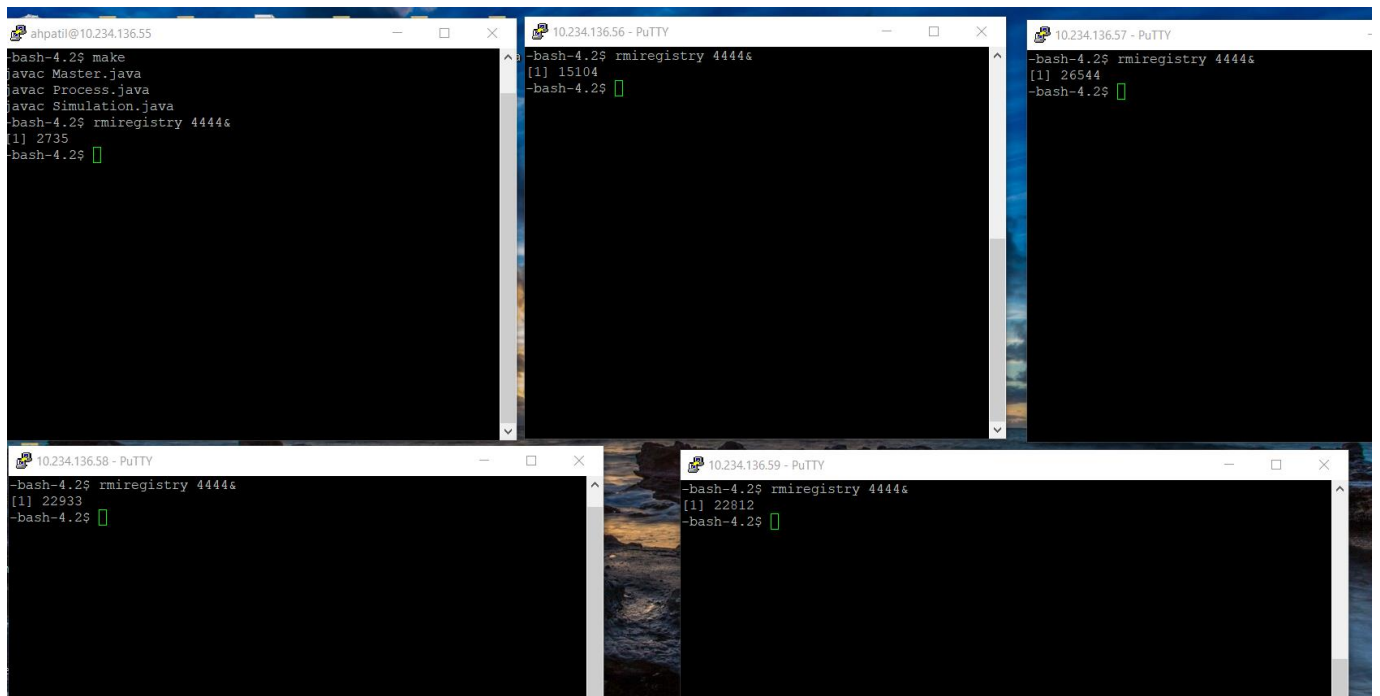
## Run Samples

- 1) Run the Make file using command make



```
ahpatil@10.234.136.55
-bash-4.2$ make
javac Master.java
javac Process.java
javac Simulation.java
-bash-4.2$
```

- 2) Open an instance of putty on all the machines listed below and start the rmiregistry on each machine
  - a. in-csci-rrpc01.cs.iupui.edu - 10.234.136.55
  - b. in-csci-rrpc02.cs.iupui.edu - 10.234.136.56
  - c. in-csci-rrpc03.cs.iupui.edu - 10.234.136.57
  - d. in-csci-rrpc04.cs.iupui.edu - 10.234.136.58
  - e. in-csci-rrpc05.cs.iupui.edu - 10.234.136.59



```
ahpatil@10.234.136.55
-bash-4.2$ make
javac Master.java
javac Process.java
javac Simulation.java
-bash-4.2$ rmiregistry 4444&
[1] 2735
-bash-4.2$

10.234.136.56 - PuTTY
-bash-4.2$ rmiregistry 4444&
[1] 15104
-bash-4.2$

10.234.136.57 - PuTTY
-bash-4.2$ rmiregistry 4444&
[1] 26544
-bash-4.2$

10.234.136.58 - PuTTY
-bash-4.2$ rmiregistry 4444&
[1] 22933
-bash-4.2$

10.234.136.59 - PuTTY
-bash-4.2$ rmiregistry 4444&
[1] 22812
-bash-4.2$
```

### 3) Running the processes on their respective machines

```
ahpatti@10.234.136.55:~$ -bash-4.2$ rmiregistry 4444&
[1] 2904
-bash-4.2$ java -Djava.security.policy=policy Simulation 1 1
Process 1 Started!
Process 1 Stub name: //in-csci-rrpc01.cs.iupui.edu:4444/Process1

Do you wish to connect to all the processes and master?
1. yes
2. no
1

10.234.136.56 - PuTTY
-bash-4.2$ rmiregistry 4444&
[1] 15104
-bash-4.2$ java -Djava.security.policy=policy Simulation 1 2
Process 2 Started!
Process 2 Stub name: //in-csci-rrpc02.cs.iupui.edu:4444/Process2

Do you wish to connect to all the processes and master?
1. yes
2. no
1

10.234.136.57 - PuTTY
-bash-4.2$ rmiregistry 4444&
[1] 26544
-bash-4.2$ java -Djava.security.policy=policy Simulation 1 3
Process 3 Started!
Process 3 Stub name: //in-csci-rrpc03.cs.iupui.edu:4444/Process3

Do you wish to connect to all the processes and master?
1. yes
2. no
1

10.234.136.58 - PuTTY
-bash-4.2$ rmiregistry 4444&
[1] 22933
-bash-4.2$ java -Djava.security.policy=policy Simulation 1 4
Process 4 Started!
Process 4 Stub name: //in-csci-rrpc04.cs.iupui.edu:4444/Process4

Do you wish to connect to all the processes and master?
1. yes
2. no
1

10.234.136.59 - PuTTY
-bash-4.2$ rmiregistry 4444&
[1] 22570
-bash-4.2$ java -Djava.security.policy=policy Simulation 0 5
Master Started!
Stub name: //in-csci-rrpc05.cs.iupui.edu:4444/Master

Do you wish to connect to processes?
1. yes
2. no
1
```

### 4) Once all the processes have been started select the yes option to connect to all processes and master

```
ahpatti@10.234.136.55:~$ -bash-4.2$ rmiregistry 4444&
[1] 2904
-bash-4.2$ java -Djava.security.policy=policy Simulation 1 1
Process 1 Started!
Process 1 Stub name: //in-csci-rrpc01.cs.iupui.edu:4444/Process1

Do you wish to connect to all the processes and master?
1. yes
2. no
1

10.234.136.56 - PuTTY
-bash-4.2$ rmiregistry 4444&
[1] 15104
-bash-4.2$ java -Djava.security.policy=policy Simulation 1 2
Process 2 Started!
Process 2 Stub name: //in-csci-rrpc02.cs.iupui.edu:4444/Process2

Do you wish to connect to all the processes and master?
1. yes
2. no
1

10.234.136.57 - PuTTY
-bash-4.2$ rmiregistry 4444&
[1] 26544
-bash-4.2$ java -Djava.security.policy=policy Simulation 1 3
Process 3 Started!
Process 3 Stub name: //in-csci-rrpc03.cs.iupui.edu:4444/Process3

Do you wish to connect to all the processes and master?
1. yes
2. no
1

10.234.136.58 - PuTTY
-bash-4.2$ rmiregistry 4444&
[1] 22933
-bash-4.2$ java -Djava.security.policy=policy Simulation 1 4
Process 4 Started!
Process 4 Stub name: //in-csci-rrpc04.cs.iupui.edu:4444/Process4

Do you wish to connect to all the processes and master?
1. yes
2. no
1

10.234.136.59 - PuTTY
-bash-4.2$ rmiregistry 4444&
[1] 22570
-bash-4.2$ java -Djava.security.policy=policy Simulation 0 5
Master Started!
Stub name: //in-csci-rrpc05.cs.iupui.edu:4444/Master

Do you wish to connect to processes?
1. yes
2. no
1
```

## **Comparisons between the trends of results with those obtained in the first assignment**

- 1) In assignment 01, I created a pseudo distributed simulation. Using RMI, now I'm able to create a real distributed simulation.
- 2) In assignment 01, everything was local. The process could easily make a local call and put their messages in the master's queue. Now, there is separation of concerns. Wherein a remote invocation is made over the remote method.
- 3) My graphs of assignment 01 and assignment 04 show variations, most of which is because of the probabilities of event selection, which is done at random.
- 4) In RMI, I didn't have much of work in case of clock consistency. I made use of the code and logic from assignment 01. Just changed the local calls to remote calls by creating a remote interface which has a remote method.
- 5) In assignment 01, there was no security in terms of logical clock values i.e. no encrypted message. In this assignment, I have provided security to the time using public key.

### **Pros:**

- 1) The master and process successfully communicate with each.
- 2) The "Happens before relationship" is acquired by using the concept by Lamport.
- 3) The failure doesn't affect the working of other processes in the system.
- 4) All the processes run properly and effectively execute their functionalities.
- 5) Provides a mechanism for clock consistency
- 6) Provides security by using the encryption mechanism

### **Cons:**

- 1) Since the clock adjustment is based on the Lamport's logical clock, a drift in one process's logical clock can cause drift in other processes.

### **Conclusion:**

Thus, using this model I was able to effectively implement a distributed simulation using Java RMI. The logical clocks worked successfully based on the concept by Lamport and the techniques based on Berkley's algorithm. This model efficiently used the principles of clock consistency, drifts and inter-process communication.