**Introduction:**

This assignment required the implementation of threads. I have implemented threads using Java in this assignment. All the threads are interacting with one another as mentioned in the assignment. The threads interacting are the threads of the Master Object(MO) and four Process Objects(PO1, PO2, PO3, PO4). The main thread that is of the program does not interact. So, five threads are running concurrently which simulates a distributed environment as if running on five different servers or computers. They have logical clocks which are controlled by Lamport's and Berkeley's Algorithms. The logical clocks are simple counters implemented inside functions which are incremented at send-receive events which happen during the thread execution lifecycle which are mentioned in the assignment.

Lamport's Happened Before Relationship: It states that when a send and receive event happen then it must be that the timing of the sent event is before the timing of the receiving event i.e., as mentioned in the assignment if PO1 sends a message to PO2 then (the clock value of PO1)<(the clock value of PO2). It should not be that (the clock value of PO1)>(the clock value of PO2) which is impossible.

Berkeley's Algorithm: The MO takes an average of all the logical clock values including its own and then sends an offset relating to each PO. The each PO then accordingly updates its logical clock.

In the thread simulation environment, the programs run as per their own execution but decided by the Operating system and the Java Runtime Environment(JRE).

**Logics of the Program:**

I have created four process object classes and also a master object class and then initialized them in a main class. The main class will just pass the thread name to the MO and the 4 POs. The threads are also started in the main class. The MO and 4 PO classes extend the Thread class in Java.

Class names:

Class containing main method: Mainclass

Class of Master Object Thread: Masterobj

Class of Process Object 1 Thread: Processobj1

Class of Process Object 2 Thread: Processobj2

Class of Process Object 3 Thread: Processobj3

Class of Process Object 4 Thread: Processobj4

The PO classes are similar with respect to the fact that they have similar functions and variables.

Variables and their descriptions are given below:(i=1 to 4)

1.probpoi: It is the probability for deciding if the process object will send a message to another process object.

2.probpoit:It is the probability for deciding that after t units of time the process object will send its timestamp to master object.

3.counterpoi:It is the counter implemented i.e., the clock of process object which will get incremented after every event in the particular process object.

4.offsetpoi:It is the offset value of a particular process object which is calculated by the master object.

5.flagpoi:It is the flag value which is a boolean value which will tell as to when the process object must go to adjust its clock value as per the Berkeley Algorithm.

Constructors:

Each Process object has only one constructor of its own. The constructor simply takes in the unique name of the Process Object from the main class. Its purpose is just to give a name to the Process Object. It has a threadname variable which can be a string variable. Any type of values can be passed to name the process object.

Functions inside the Process Objects:(i=1 to 4)

1.sendpoi: It sends the other PO randomly decided the timestamp and to the master object its clock value and process id.

2.receivepoi: It performs the function of receiving the timestamp/clock value from other POs and then adjusts the clock value as per Lamport's happened before relationship.

3.receiveoffsetpoi: It performs the function of receiving the offset value from the MO and then notifying the PO.

4.sendcorrectpoi:It is used to send the MO the clock value after t units of time.

5.correctpoi:It is used to correct the clock value as per the Berkeley algorithm by the PO itself.

6.outputpoi:It is used to output the current value of the PO clock.

7.run():It indicates the start of the PO thread instance.

The Master Object also has variables as follows:

1.countermo:It is the counter for MO which stores the clock value.

2.flagmo: It is the boolean flag indicator for notifying MO.

3.a:It is the array used to store the values of each clock of the PO as well as the MO.

4.b:It is the array used to temporarily store at t units of time the clock values of all POs and the MO.

Constructor:

The MO has its own constructor which just takes the value of string name as parameter and is used to give the thread a name.

Functions used inside the MO:

1.receivemo:It is used to receive the clock values of POs and then store them in an array.

2.correctorpo:It is the function called by the POs after t units of time in order to start the Berkeley Algorithm calculations in the MO.

3.correctmo1:It is the function used to calculate the offset for each and every PO and send it to them respectively.It performs the Berkeley Algorithm.

4.outputmo:It performs the function of performing the output of the clock value of the MO.

5.run():It indicates the starting instance of the MO thread.

**Interaction Model:**

The interaction model defines how the interprocess communication takes place in a distributed system which in this case is a simulation of threads. It also may show the delays and other details pertaining to the inter thread communication.

The following explains how each PO and MO interacts in the program as per the assignment in my design:

1.The Pos and the MO first get their thread names passed and are started in the main class.

2.After each thread starts it will start executing in parallel with the other threads.

3.Inside the run method in each of the POs it will randomly decide by use of probability whether it wants to send a message or not.

4.If it decides to send the message then it will go further and then decide which PO to send the message to using Random in Java.

5.Then it enters the sendpoi function and then sends the message to the PO decided randomly and also to the MO.

6.It sends the message using the receivepoi and receivemo functions.As those functions are all static they can be accessed by specifying the classname and the function.

7.In the receivepoi, the counter of the receiving PO is incremented or adjusted as per Lamport's happened before relationship. The timestamp of the incoming message should be less than its own timestamp. Otherwise the PO will take the value of the incoming clock value and then add 1 to it and adjust its clock value.

8.In the receivemo, the MO counter is updated and the clock value of the PO and MO are stored in the a array.

9.The PO returns and then increases its own counter in sendpoi.

10. After t units of logical time varied using probpoit, the PO will send a message to the MO containing its PO id and its clock value to correctorpo function in MO.

11.In correctorpo, the clock value of the MO is incremented and the clock value of the PO id is also put in the array and the contents of a array are put inside b array to prevent global access by other POs. Flagmo is set as true and in the for loop it will indicate in the next iteration to MO to call correctmo1 function.

12.The PO returns and the correctmo1 function calculates the average of all the POs without waiting for all the POs and taking their current timestamp/clock values and then calculates an offset.

13.The individual offset is then sent to each of the POs in receiveoffsetpoi and the flag is set and the MO returns.

14.Once the flag is set for each PO then they will enter the correctpoi function and then calculate the offset as (PO clock value+individual offset+1).

15.The MO after each offset sent will update its own counter as it takes 1 unit of logical time to send an offset and it is an event and even the POs add one to the offset.

16.The offset is negative if the average is lesser than the PO clock value and will be positive if the average is greater than the PO clock value.

17.The POs output their clock values in regular time intervals in the for loop making it easy to plot them out.

Special Points:

1.The offset being sent by the MO is sequential in nature to each PO hence MO counter has been incremented by one for each iteration.

2.The PO when it initiates a correct function after t units of time the MO takes the current value of the other POs stored in the array. So it does not have an updated copy as all other POs continue to interact with one another and they adjust back their clock.

3.The POs after receiving offset from the MO which has the previous time offset as array b values are considered and POs may have interacted ahead it will have some small factor of error.

4.The thread execution depends on the operating system and thus some threads may be given priority over others. It is the decision of the JVM or the operating system.

5.The clocks may drift over a period of time and thus are correct by both algorithms.

Problem I faced in doing the Interaction Model:

1.The clock value of MO is updated by the incoming PO and the clock value of the individual PO is updated by the incoming PO.I could not implement the PO or MO to increment its own counter as it would not print the results properly due to the fact that as I have used global(static) variables in my design. I tried using another flag variable in order to indicate the incoming PO/MO but the moment the

PO breaks out of the FOR loop the value of the clock value received may have been changed by another incoming PO and it would give inaccurate results.

2.The MO when I made it to update its own value it was printing the MO counter after a long interval of some 20 values. But in my design the values are printed out properly.

3.However, I was able to make the PO adjust its own clock value and the MO also to adjust its own clock value as per the Berkeley Algorithm. The Lamport part gets done by the incoming PO due to the global access error and inconsistency I have mentioned before.

4.Rest everything is correct as per the assignment given which has been implemented.

5.I have used static functions and static variables and thereby accessing the class method directly. I tried the another method in which the POs are initialized in one class and the MO class has special functions and the main method runs the program and the threads are run simultaneously and are having private variables and methods but I could not take out the object of the thread to make it access the other PO class and it was getting too complicated to trace and identify the mistakes. Hence, I adopted this approach in order to complete the given assignment.

**Failure Model:**

Failure Models are when a fault occurs in the software or in the networks of distributed systems. However, in the thread simulation it will not have any network problems but will have thread asynchronous activity.

The failure models in my design are as follows:

1. As it is being executed in a multithreaded fashion, the POs may interrupt another PO while accessing a resource.

2.The POs may be delayed for execution by the JVM or the operating system due to hardware or software constraints.

3.If there are hardware constraints then some threads may be executed in a sequential fashion and the purpose of parallelism will be defeated.

4.The system on which the program is running may crash due to some software faults which can be unknown or unrelated to the program in execution.

5.Due to insufficient memory the system may crash while trying to run the threads.

6.If one Process Object stops execution due to some unknown reason(Byzantine failure) then the clocks will not be synchronized as desired.

7.If a Process Thread suddenly sends a very high value then all the other process objects will have a jump in their clock values as there is no checking mechanism implemented.

**Analysis of Cases:**

**Case 1:**

In this case, only PO1 sends a message to MO after t interval.

probpo1: 99

probpo1t: 16

probpo2: 50

probpo2t: not considered

probpo3: 99

probpo3t: not considered

probpo4: 25

probpo4t: not considered

Here, the MO increases a little above and then syncs with the POs. The POs do not drift much.

Reason: Maybe due to the fact that the MO is sending offsets to all and its events are more.

**Case 2:**

In this case, all POs have different t intervals and also different probabilities.

probpo1: 99

probpo1t:16

probpo2:50

probpo2t:15

probpo3:99

probpo3t:19

probpo4:25

probpo4t:17

Here the POs are starting with sync but after midway they drift and sync again.

Reason: Maybe due to the fact that all PO2 may be sending/receiving repeatedly but others not so much. MO is also slightly higher there.

**Case 3:**

Here the POs have different probabilities but their t interval is the same.

probpo1:75

probpo1t:15

probpo2:50

probpo2t:15

probpo3:99

probpo3t:19

probpo4:25

probpo4t: 17

Here the POs start to sync drift, sync drift.

Reason: Maybe because the t interval is the same so at that time everyone syncs.

**Case 4:**

Here the POs have same probabilities and also their t interval is the same.

probpo1:99

probpo1t:15

probpo2:99

probpo2t:15

probpo3:99

probpo3t:15

probpo4:99

probpo4t:15

Here the POs are controlled and are always in sync but MO increases and then syncs.

Reason: It may be due to the fact that the probabilities are same and the t interval keeps them in sync. In case 3 due to different probabilities it was not so uniform. But MO will show spike as it has many events during that interval of t to send messages to all POs.

**Case 5:**

Here the probabilities are the same but the t interval is different.

probpo1:99

probpo1t:19

probpo2:99

probpo2t:12

probpo3:99

probpo3t:26

probpo4:99

probpo4t: 55

Here the probabilities are same and the clocks are somewhat in sync.

Reason: Maybe because at different t intervals the Berkeley Algorithm works and the clocks are not much away from each other.

**Case 6:** Byzantine Behavior

Here the PO4 does not start.

probpo1:99

probpo1t:19

probpo2:99

probpo2t:12

probpo3:99

probpo3t:26

probpo4:Not start

probpo4t:Not start

Here, due to some unknown failure the PO4 does not start. So, the clocks of other POs and MOs reduce.

Reason: Maybe due to the fact that the MO and POs consider PO4 clock value to be 0 and in averaging the offset is negative.

**Case 7:** (Byzantine Failure)

At iteration i=75, PO3 sets clock value as 10000 due to some internal fault.

probpo1:99

probpo1t:19

probpo2:99

probpo2t:12

probpo3:99

probpo3t:26

probpo4:99

probpo4t:35

Here the clocks until 75 sync and then they experience a jump.

Reason: As the Lamport correction adds all (the incoming values which are greater+1) and the MO also experiences a jump in clock value after implementing Berkeley Algorithm.

For each runtime we get a different output in each case but the pattern remains almost same.