**Indiana University Purdue University Indianapolis**

**Department of Computer and Information Science**

**CSCI 53700**

**Fall 2018**

Assignment – 1

Devanapally Sai Krishna

2000276942

Contents

[1.Introduction: 3](#_Toc526809452)

[2.Design – Choices: 3](#_Toc526809453)

[2.1Primary choice: 3](#_Toc526809454)

[2.2 Secondary choices: 4](#_Toc526809455)

[3.Implementation details: 4](#_Toc526809456)

[3.1 Main Class: 4](#_Toc526809457)

[3.2 PO Class: 4](#_Toc526809458)

[3.3 MO Class: 5](#_Toc526809459)

[4. Interaction Model: 5](#_Toc526809460)

[5. Failure Model: 5](#_Toc526809461)

[6. ANALYSIS and DESIGN: 6](#_Toc526809462)

[6.1 System without failures: 6](#_Toc526809463)

[6.2 System with 33.33% arbitrary failure in PO2: 7](#_Toc526809464)

[7. Screen Shots: 8](#_Toc526809465)

# Introduction:

The aim for the assignment is to create and simulate clock drifts between 4 process objects and 1 master object on a single processor using Java Threads. Every process object also referred as PO and the Master object referred as MO has its own Logical clock proposed by Lamport, which will be incremented individually for every PO and MO events which occur within them or between them. And to solve the consistency of clock drifts we will be using a modified Berkley Algorithm; where we will be taking average of all the logical clocks in the system and keep updating all the clocks by sending the offset values to its respective Process object to adjust the clock according. By doing this we will be able to identify which event happened before/after another event. The system also exhibits Byzantine or Arbitrary failures, where at a certain point the system one of the logical clocks in PO will suddenly increase the value it sends.

# Design – Choices:

## Primary choice:

* The implementation of the entire system is to be done on a single processor, for this to work the solution is using threads
* In case of threads on a single process, concurrency can be simulated by serialization.
* In our case concurrency must be achieved between PO’s and MO
* For data communication between threads there must be a common data structure, which will be accessed by passing it as a reference different thread.
* In our case we have used Array Blocking queue which is sub class of Blocking queue data structure which was introduced in Java 7. Blocking queue Interface is of java concurrency package which can be accessed by using java.util.concurrent.BlockingQueue. And Array Blocking queue is one of its implementing classes.
* The advantage of using a Blocking queue as the data structure is, it was designed for concurrency aspect of java which make it efficient and easy to work with java threads. The use of Array Blocking queue gives us a blocking queue backed by an array which makes it easy for easy access and retrieval of the elements and has all the properties of a queue in our case FIFO and specifying the capacity is what we want.
* We start by creating two Array Blocking queues which will be passed as arguments for the thread's MO and PO’s. One of them is used to send logical clock values of PO’s to MO and the other to send the offset values from MO to respective PO’s.
* All the PO’s are started with queue references along with a process id as a reference when sending back the offset values.
* In PO thread there are 3 events, send, receive, internal communication which are done at a random

1 -> Internal Communication

2 -> Send Event

3 -> Receive Event

Which gives us the probability of each event to 33.33%.

* For each of these event the logical clock value increases by one.
* In MO thread there are 2 events internal communication and send event which are done at random.

1 -> Internal communication

2 -> Send Event

* In both the cases logical clock of MO is increased my one.

## 2.2 Secondary choices:

* The other possible choices which were tried were by using a Hash-map as a data structure to send and receive values but there was an issue of data overwriting.
* The next attempt was done by using linked blocking queue but the problem with that data structure was when once a value is pushed out and due to the size restriction, the reusability of the queue was not possible.

# 3.Implementation details:

## 3.1 Main Class:

* Here we are creating 2 Array Blocking queues as mentions earlier for sending and receiving data, offset values respectively.
* Next step is creating and starting 4 PO threads and a MO thread.
* When creating new objects, an extra argument is passed to check a failure condition is also passed.

## 3.2 PO Class:

* Since run method starts and runs the thread it must be overridden with required communication part.
* In our case we have 3 conditions which are called by random.
  1. Here the process checks the failure condition if it is an “YES” then the clock increases by 20 or in other case the clock will increase by one and there is no communication. The thread goes to sleep for context switching; it creates time for PO’s to start their threads.
  2. It is a send event. Here the process checks the send queue if there is already any sent message from that PO, if there is one it overwrites that data or else if the queue with respect to that particular PO is empty it pushes the data on to the queue. And the thread goes to sleep for context switching.
  3. It is a receive event. Here the process checks the other queue for using the ID parameter to find the correct offset in the second queue and updated the logical clock of PO by adding 1 to the send value. To satisfy the condition mentioned in the question.

## 3.3 MO Class:

* Here also we overwrite run method with required operations; to calculate the offset part and push it onto the second queue.
* Here we have 2 conditions which are called by random.
  1. It simulates internal communication and increments internal clock value by 1 and goes to sleep for context switching.
  2. Here the average of all the PO’s is calculated in the first if condition and the values of the PO’s are pushed from the first queue to the respective PO variable in the MO class by checking the ID. In the else condition the offset values are calculated and pushed onto the second queue based on the ID of the PO. By doing so it possible to pull the value from the queue in the MO receive condition.

# 4. Interaction Model:

* Interaction model gives us the distributed computing entities communication patterns.
* In our case it is difficult to say which PO or MO communicates first or the sequence of communications between the threads.
* Thus, we will be getting an asynchronous interaction model where we cannot determine which communication happened before what.
* The logical clocks behave as internal clocks also which satisfy another condition of interaction model.

# 5. Failure Model:

* Omission Failures may happen in this system. In that case, when a process fails completely it will not send any value to the MO to computer the average logical clock. If this is the case, system will keep running without the value of the failed processes omitting the failed process since every PO has its unique id and for every iteration the queue is drained to make room for next logical clock value.
* Arbitrary failure: In our case we are simulating arbitrary failures or byzantine failures whenever the failure parameter in the main class is set to YES and PO calls the internal communication part it increments the clock value by 20.

# 

# 6. ANALYSIS and DESIGN:

## 6.1 System without failures:

On the X – Axis is the time and on Y – Axis is the logical clock. As we can see from the graph if one of the clocks drifts it shows an impact on all the other clocks and MO tries to adjust it so it causes a ripple in the all the clocks.

## 6.2 System with 33.33% arbitrary failure in PO2:

On the X – Axis is the time and Y – Axis is logical clock. Here PO2 exhibits arbitrary failure and chance for this is 33.33% in PO2 thread. As we can see if there is an failure in the system that is, if one PO2 increases its value arbitrarily it cause all the other clocks including MO to drop its logical clock values.

# 

# 7. Screen Shots:

Sample screen shots of the run results.

  