CSCI 53700

DC ASSIGNMENT 4

REPORT

Arvind Nair

12/1/2014

Introduction:

The purpose of this assignment was to re-enforce the principles of remote method invocations using the Java-RMI model of distributed-object computing. It required reimplementation of the first assignment in which there were 4 Process Objects(POs) and one Master Object(MO) each having a counter as a logical clock which increments at every event and each PO would pass its timestamp to another PO and also MO depending on the probabilities given by a parameter which can be varied. The Receiving POs will adjust their clocks if their time is less than the received time. If it is more, then they will simply increment their clock value by one. Also after t units of time one of the POs will send its time stamp to the MO and the MO will calculate the average of all the time stamps of all the POs and send the offsets to all POs. Hence, each of the POs accordingly will correct their clocks as per the offset received. Each of the POs and the MO are running on different servers on the cluster allocated on pegasus on different machines and they will interact with each other using Java-RMI.

Java-RMI (Remote Method Invocation):

> "Java Remote Method Invocation (Java RMI) enables the programmer to create distributed Java technology-based to Java technology-based applications, in which the methods of remote Java objects can be invoked from other Java virtual machines*, possibly on different hosts. RMI uses object serialization to marshal and unmarshal parameters and does not truncate types, supporting true object-oriented polymorphism."[1]

Meaning:

In a distributed system, Java RMI enables the invocation of remote methods present in different locations but simplifies it and makes it look like a local method invocation. It makes use of the rmiregistry of java which runs in the background at server and the client(s) will invoke remote methods of the server by passing objects like a local method invocation after lookup and connection. These objects are serialized using object serialization in java in which the arguments are marshalled and unmarshalled (making data suitable for transmission over the network). These details are taken care of by the Java Virtual Machine(JVM) and the Java RMI and the programmer need not worry about the lower level details of their implementation. Java RMI provides a higher level of abstraction so that the programmer is not concerned with the lower level implementation specific details and should only be concerned with the functionalities offered by the software or programs that he/she has made.

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 by adding the offset and 1 to its clock value. The offset can be negative so the POs' or the MO's clock/counter can go back.

RMI Architecture:

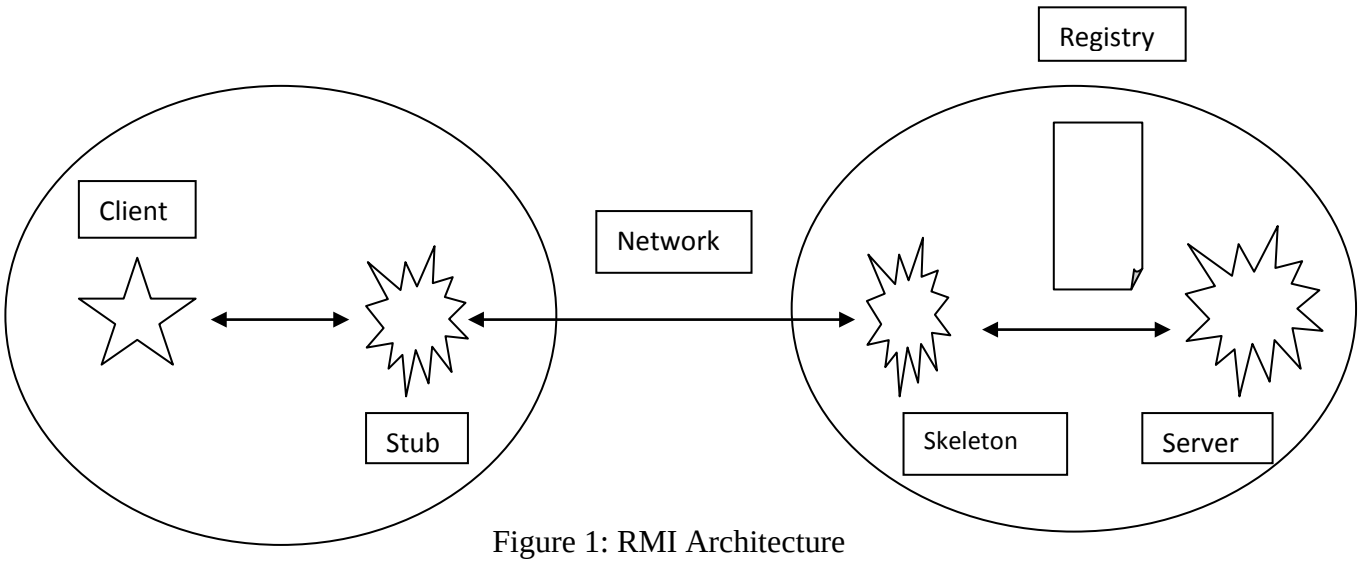

Figure 1: RMI Architecture

Working of Java-RMI:

1. RMI clients interact with the remote objects via their published interfaces which are defined in the interface file.

2.Remote objects are passed by reference.

3.The calling is similar to RPC and we have the notion of stub at client side and skeleton at server side.

4.The added part is the registry which is used to perform the registration and lookup of the remote objects to be referenced by the client.

Logics of the Program:

Use of RMI:

For implementing the assignment using Java-RMI we need to follow the protocols and the relevant procedure which is used. First we create an interface for the server which will have the methods which are to be remotely accessed by the client. The server then implements this

interface and has to have the functions specified in these interfaces. The server will then use Naming rebind to register in the rmiregistry. The client will then use Naming lookup to find out the server using the rmiregistry. It will then create an instance of the server class an use that object to pass in parameters to the server. The server then returns the result depending on the functionalities provided by the methods.

Application in my Program:

I have created five interfaces, five classes and five more classes which extends the thread class.

The MO and the 4 POs are all acting as servers in my design as they need to send and receive the time stamps.

Interfaces:

My program has 5 interfaces to fulfill the requirements of the assignment.

All of these interfaces extend the java.rmi.Remote interface which is used to identify the methods to be invoked from a remote machine. Only the functions/methods specified in the interface which extends java.rmi.Remote are available remotely.

The methods also are required to throw a java.rmi.RemoteException. It is used for error handling.

1.masterObjInterface: It contains the methods receivemo and correctorpo.

2.processObj1Interface: It contains the methods receivepo1 and receiveoffsetpo1.

The other 3 POs contain the same methods on similar lines of processObj1Interface.

In my design, I have kept only those methods as remote which have receive events from other POs or MO.

Classes:

My program has 5 classes each for the MO and 4 POs. The classes also present are the threads used to run the counter and the class which contains the main method extends the UnicastRemoteObject and implements the particular interface. This acts as the server which contains the remote methods to be invoked by the other threads as required in the assignment. The MOThread and POiThread classes(i=1,2,3,4) extend the Thread class as they run separately from the main class. The classes and their functions and variables are explained below in detail:

Class MasterObj:

It extends the UnicastRemoeObject and implements masterObjInterface for RMI.

Variables in MasterObj:

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.

Functions in MasterObj: (Can be Remotely Invoked)

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.

The main method creates the MO and binds it to a name in the registry so that its functions can be remotely called by creating an instance and calling its functions through that instance just like a simple local call. It also starts the MOThread which will run in the for loop.

Class MOThread:

It extends the Thread class and runs the for loop.

1. MOThread Constructor: It is used to assign a name to the thread.

2. run(): It is the point where the thread starts to run separately from the main method.

3.I have provided the functionality of asking the user if he is ready to connect to the 4 POs. When the 4 POs are registered only then should the user type y. This is done as a part of exception handling as when the MO does a lookup in the rmiregistry the POs must be up and running. So, I have incorporated this mechanism in my design using simple string check and if loop i.e., if the user enters y the reply is compared using string functionality and then the MO connects to the requested PO. The input mechanism makes the MO to wait so that the POs can be started before connection.

4. Inside the for loop are the functions to calculate the offset when asked by any PO and also to print out the MO counter value at each iteration.

Class ProcessObji: (i=1 to 4)

It extends the UnicastRemoeObject and implements processObj1Interface for RMI.

Variables in ProcessObji:

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.

Functions in ProcessObji: (Can be Remotely Invoked)

1.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.

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

The main method creates the PO and binds it to a name in the registry so that its functions can be remotely called by creating an instance and calling its functions through that instance just like a simple local call. It also starts the POThread which will run in the for loop.

Class POiThread: (i=1 to 4)

It extends the Thread class and runs the for loop.

1. POiThread Constructor: It is used to assign a name to the thread.

2. run(): It is the point where the thread starts to run separately from the main method.

3.I have provided the functionality of asking the user if he is ready to connect to the MO and 3 other POs. When the MO and 3 other POs are registered only then should the user type y. This is done as a part of exception handling as when the PO does a lookup in the rmiregistry the MO and other POs must be up and running. So, I have incorporated this mechanism in my design using simple string check and if loop i.e., if the user enters y the reply is compared using string functionality and then the PO connects to the requested MO or other POs. The input mechanism makes the PO to wait so that the MO and other POs can be started before connection.

4.Inside the for loop are the functions to notify MO to calculate the offsets, adjust the PO's clock value as per offset, probability calculation and sending messages to other POs and MO and also print out the PO counter.

Interaction Model:

The interaction model defines how the inter process communication takes place in a distributed system which in this case is run on different servers provided for this assignment. It also may show the delays and other details pertaining to the inter process communication. The only difference is that in the first assignment a simulation of distributed environment was done by using threads and the program was running on a single machine but in this assignment it is done on different servers.

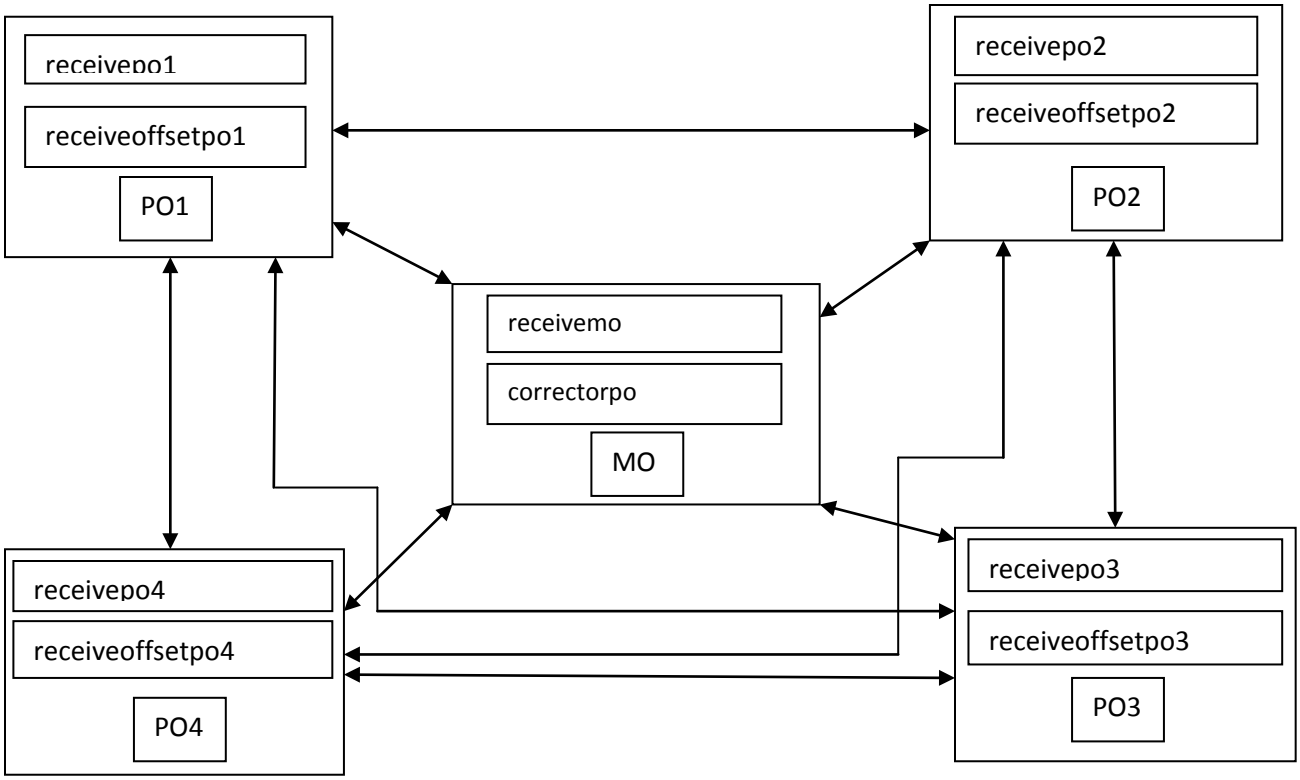

Figure 2: Interaction Between MO and POs

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

1. The MO and POs once started start their class containing the main method which runs continuously accepting remote requests/invocations and performing the required functions.

2. Then they start their respective threads which will run the for loops.

3. The user is prompted to confirm if the MO and the POs he wishes to connect are up and running. This is done as the MO and POs are located on different servers and it must be ensured that all are running before connecting.

4. After individual connections, the user is prompted to start each of the POs and the MO.

5. After starting each PO and MO starts the for loop and it will start executing in parallel on different machines.

6. Each of the POs it will randomly decide by use of probability whether it wants to send a message or not.

7. 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.

8.Then it sends the message to the PO decided randomly and also to the MO.

9.It sends the message using the receivepoi and receivemo functions and uses remote object reference of the POs and MO to be sent to access those functions.

10.In the receivepoi, the counter of the receiving PO is incremented or adjusted as per Lamport's happened before relationship by the PO server which is running in the main method. 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.

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

12.The PO which sent the message increases its own counter as the clock has to be incremented after an event occurrence.

13. 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 MO using correctorpo function which is accessed by the MO's remote reference.

14.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 calculate the offsets and send them to the POs.

15.The MO 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.

16.The individual offset is then sent to each of the POs in receiveoffsetpoi using the remote reference of the POs and the flag is set of each PO by the respective PO servers running the main method.

17.Once the flag is set for each PO then they will calculate the offset as (PO clock value+individual offset+1).

18.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.

19.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.

20.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 and as it is an event occurrence 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 class containing the main method handles the receive events and changes the variables in each of the POs and the MO. In my design, I have kept the variables as static so that any changes performed by the main method class will get reflected in the threads running the for loop. The variables are accessed in the for loop by using the name of the class containing the main method. (e.g: To access the MO counter inside the for loop in MOThread we use MasterObj.countermo).

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:

As the MO and the POs are located on different machines, it is not possible to start all of them at once like in the first assignment. Hence, I have used sleep function in the for loop which has enabled the operations to be delayed by 1.5 secs throughout the process so that only initially at startup the counter values may not be correct but after around 10 iterations it is correct. I have made it to iterate 120 times as in the end also some may finish early and so only the iterations from 10 to 110 will be accurate. The results in the graph as in analysis of cases shows a shift i.e. at time 0 only MO was running but PO4 may have started at time 4 but in graph it is at time 0 as in my design you have to manually start but the results are accurate.

Comparison to First Assignment:

1. In the first assignment, I used threads to simulate a distributed environment but in this assignment I have used actual distributed servers to implement the MO and POs.

2. In the first assignment for POs, I used functions for sending, correcting offset, output and also after t units of time sending a message to MO to calculate the offsets of all POs. However, in this assignment I have included all those functionalities inside the for loop in the POThreads.

3.In the first assignment for the MO, I used functions for correcting the offset and output but in this assignment, I have included all those functionalities inside the for loop in the MOThread.

4.Both the assignment variables make use of static variables.

5. In the first assignment, the POs and MO start together but in this assignment the POs and the MO start only after the user enters y in the console.

6. In the first assignment, the functions are accessed by using the class name and variables but in this assignment they are accessed by using remote object references using the rmiregistry. Thus Java-RMI has seen to it that the functionalities are implemented as calls on a local machine.

Failure Models:

Failure Models are when a fault occurs in the software or in the networks of distributed systems. The failure models in my design are as follows:

1. As it is being executed in a distributed 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. The rmiregistry running may stop and thus causing the environment to fail.

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 any of the servers may crash while trying to run the program.

6. If one Process Object or Master 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.

8. Network failure may cause errors to occur in the execution of the program.

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

Similar to assignment 1, 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.

But the MO does not spike as in Assignment 1(A1).

The iterations for Assignment 4(A4) are 120 hence counter values have gone to 250.

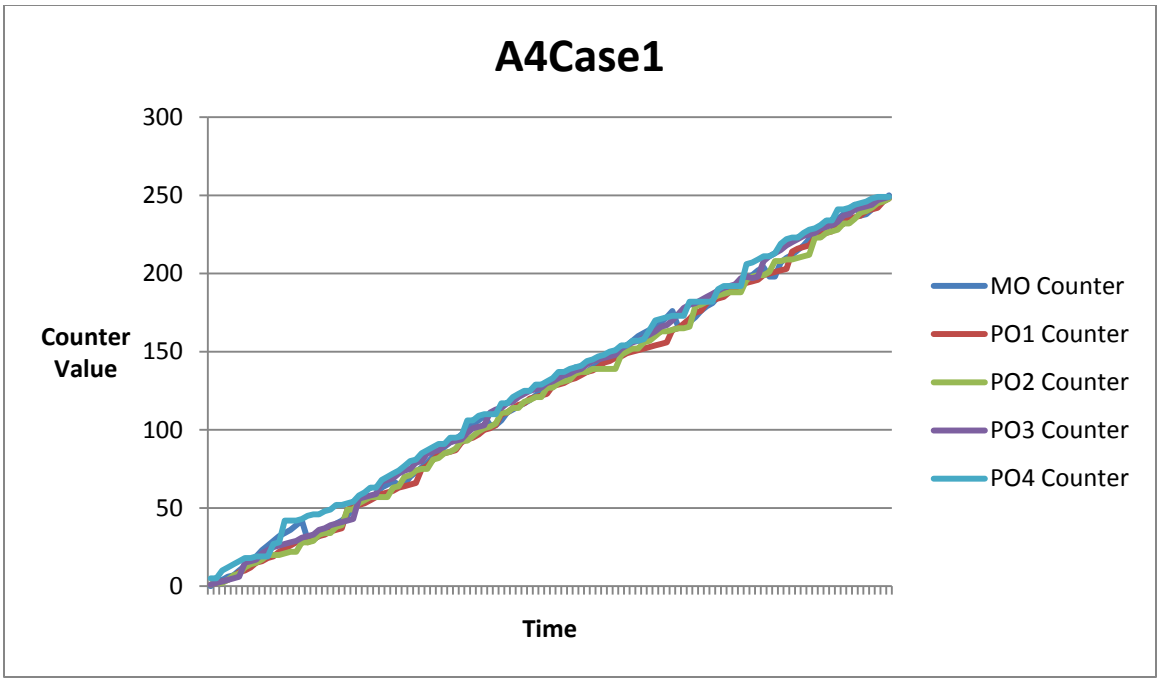

Figure 3:A4 Case 1 Graph

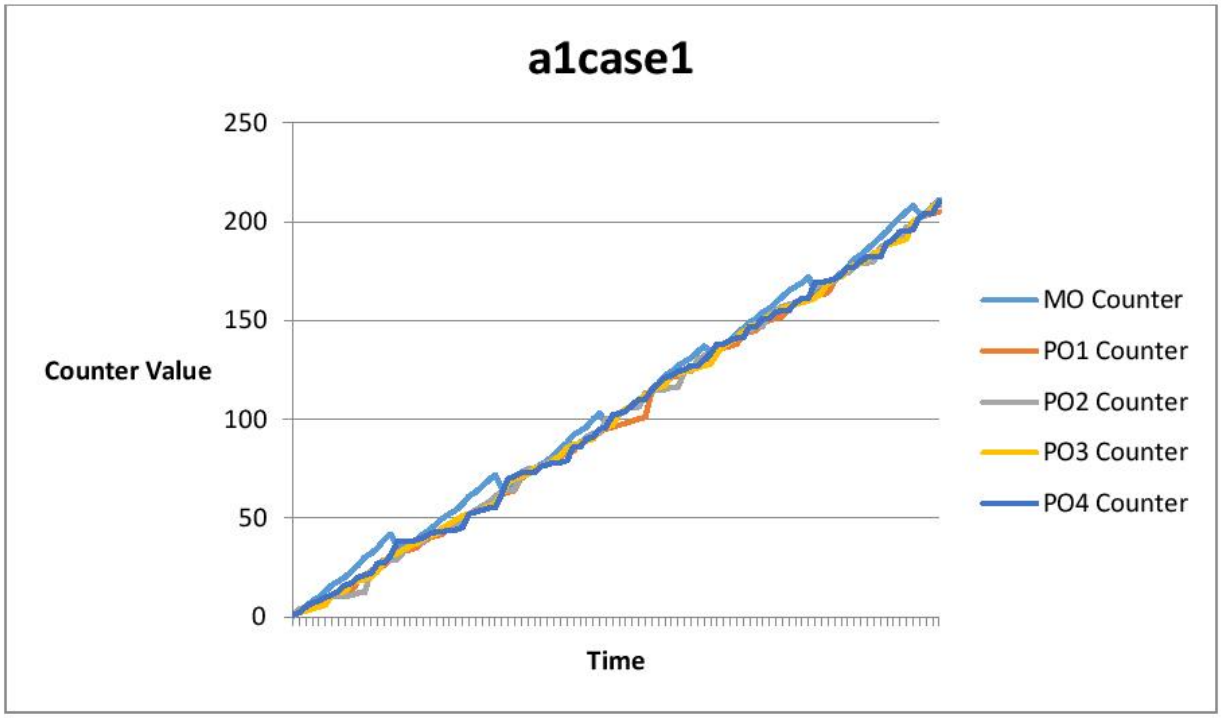

Figure 4: A1 Case 1 Graph

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

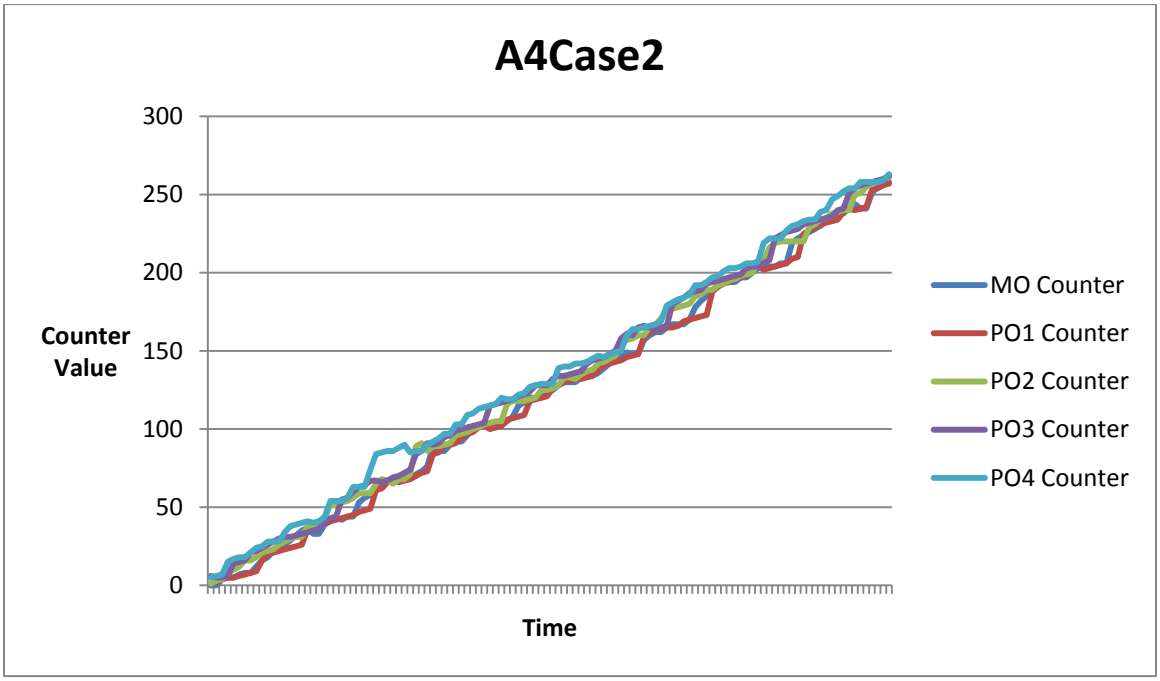

Figure 5: A4 Case 2 Graph

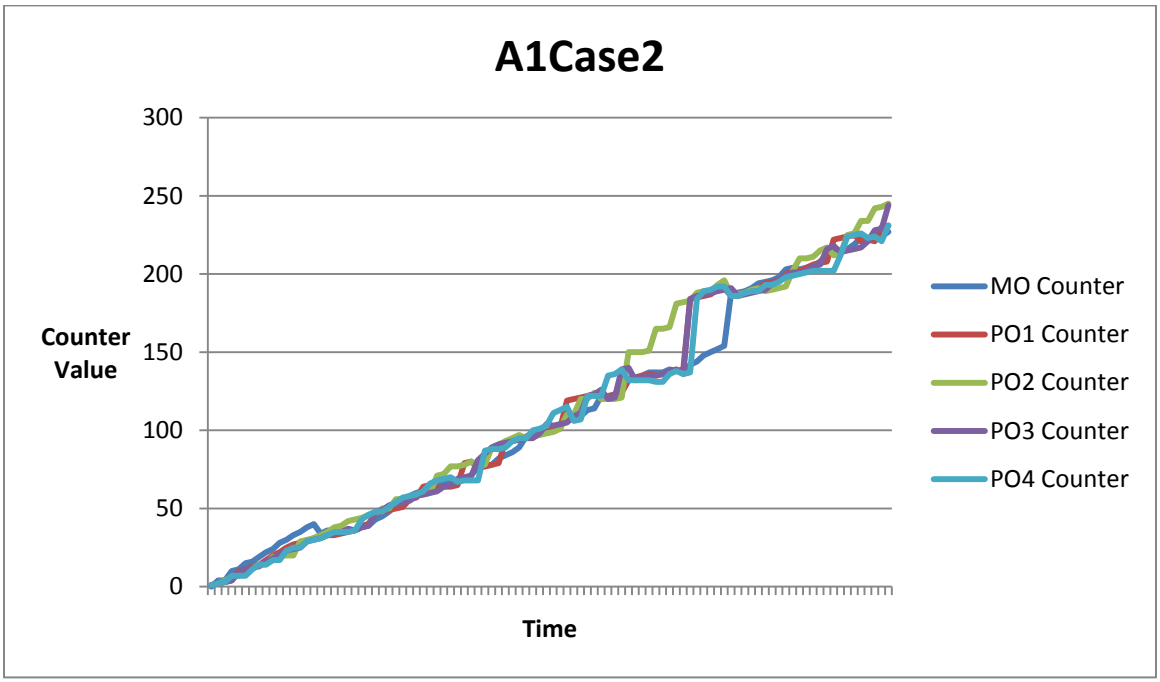

Figure 6: A1 Case 2 Graph

Here, similar to A1, the POs are starting with sync but after midway they drift and sync again.

There is a major out of sync after midway in A1 but in A4 there is a small one before midway.

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

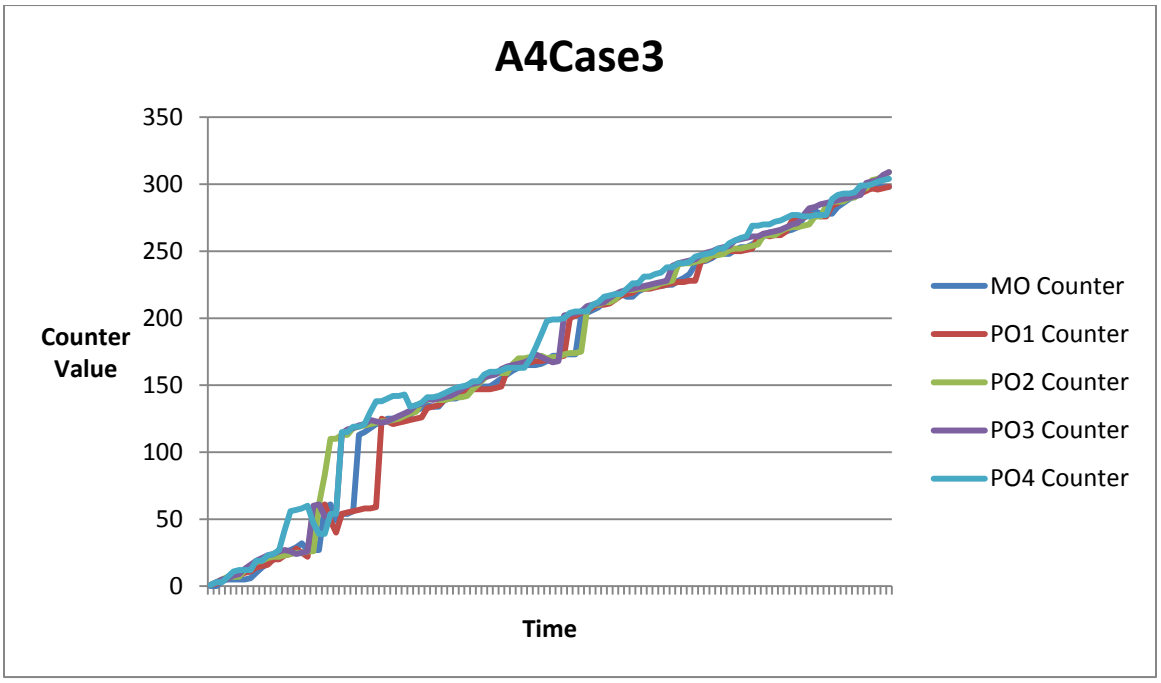

Figure 7: A4 Case 3 Graph

Here the POs start to drift initially and after a huge drift but then they sync.

But in A1 they used to sync drift, sync drift.

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

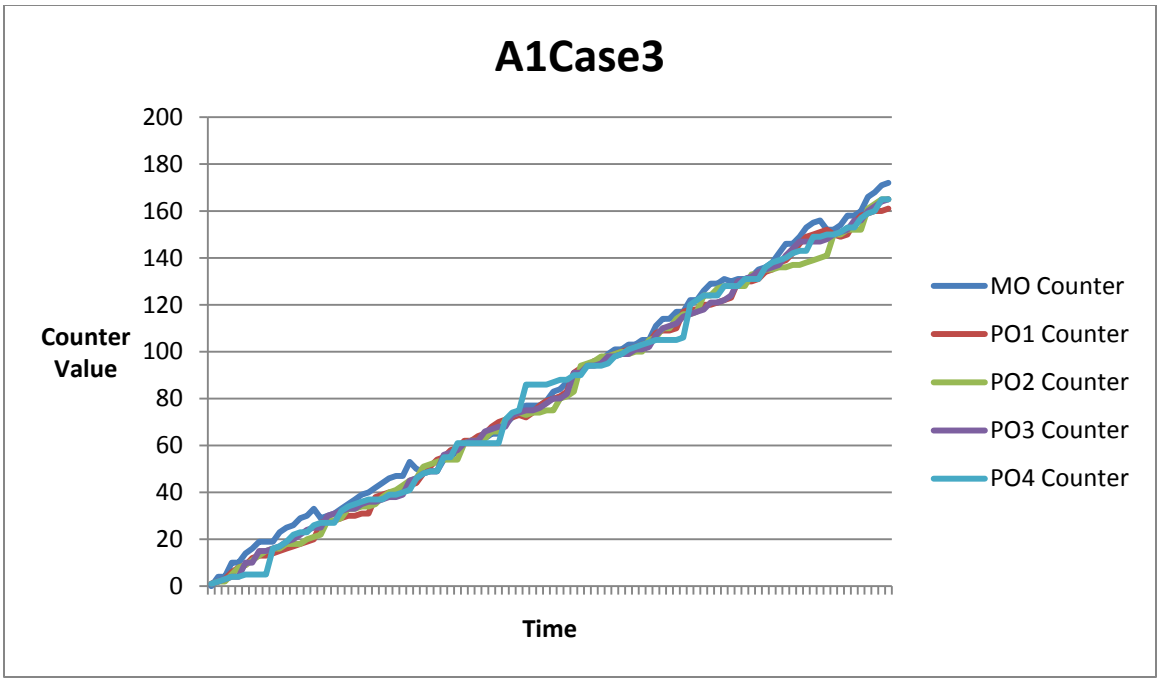

Figure 8: A1 Case 4 Graph

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

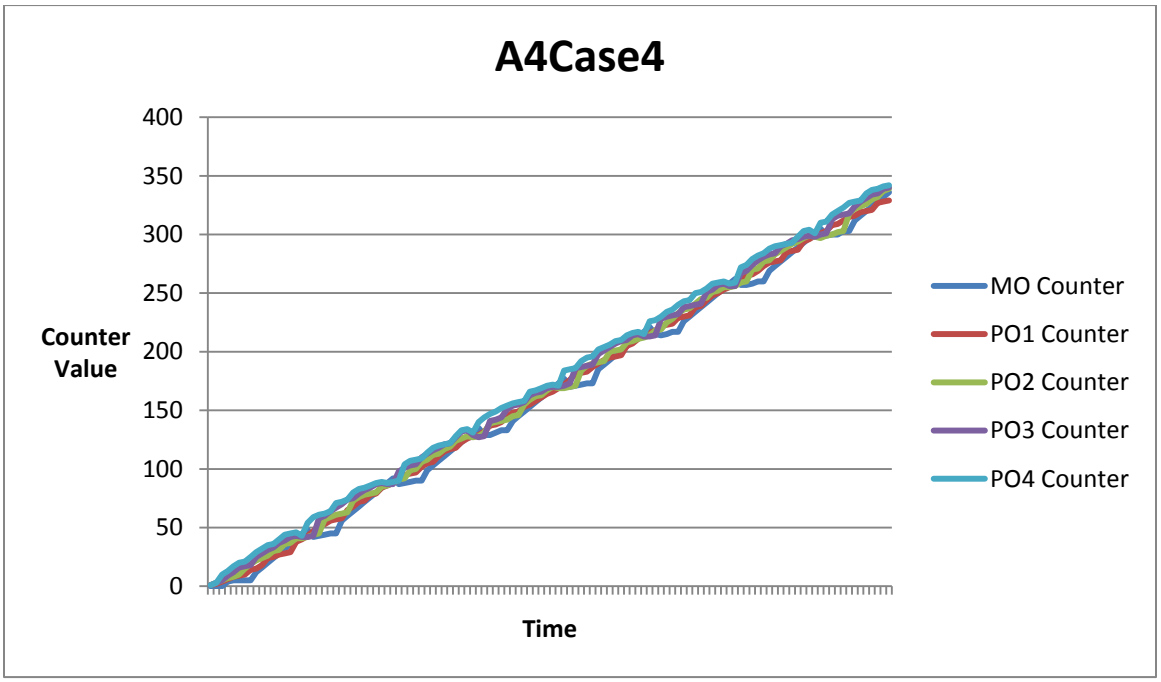

Figure 9: A4 Case 4 Graph

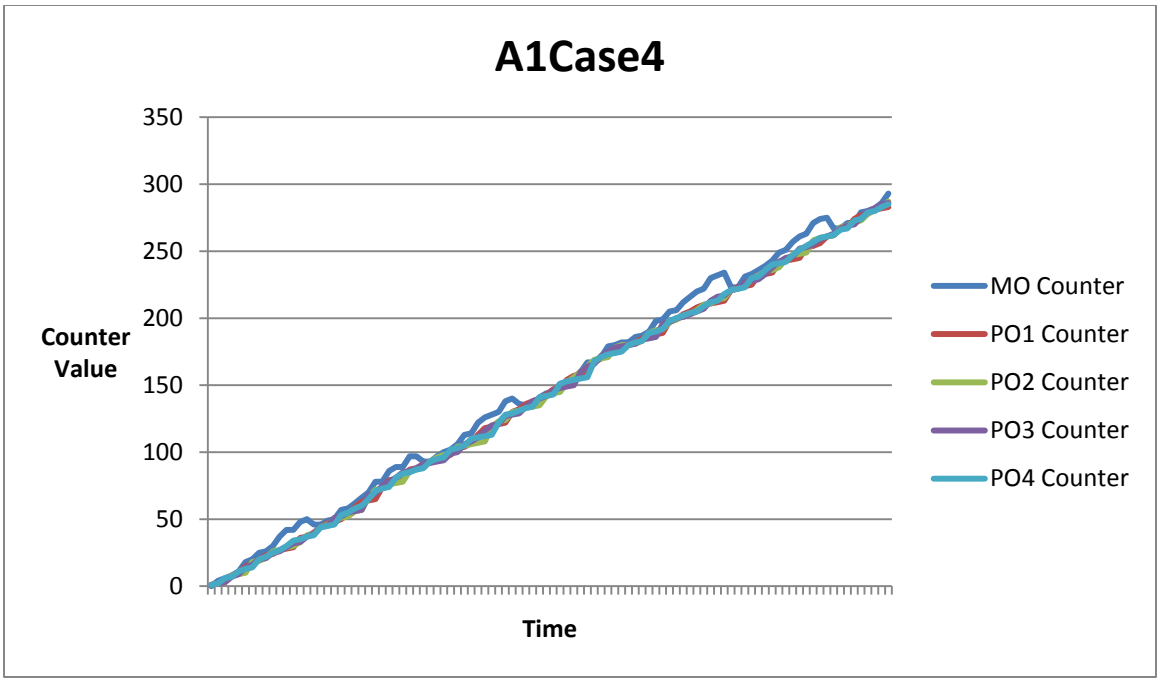

Figure 10: A1 Case 4 Graph

Here the POs are controlled and are always in sync. In A1, MO increases and then syncs but in A4 we do not observe any such change.

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.

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.

In A1 and A4 it appears similar.

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

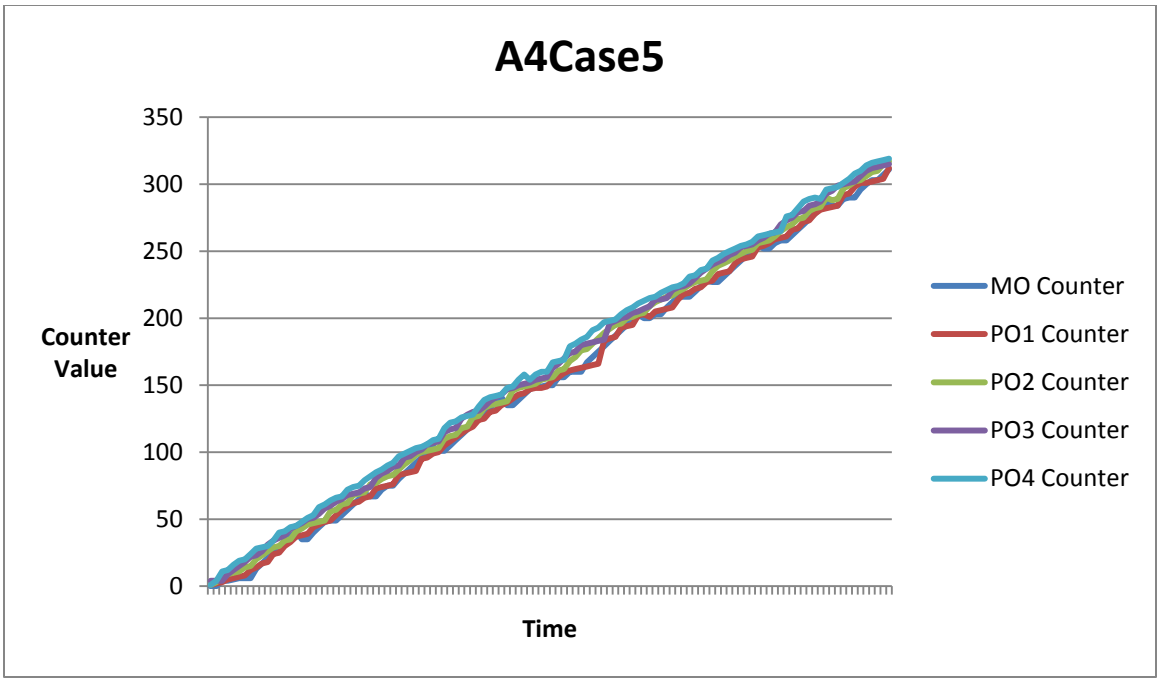

Figure 11: A4 Case 5 Graph

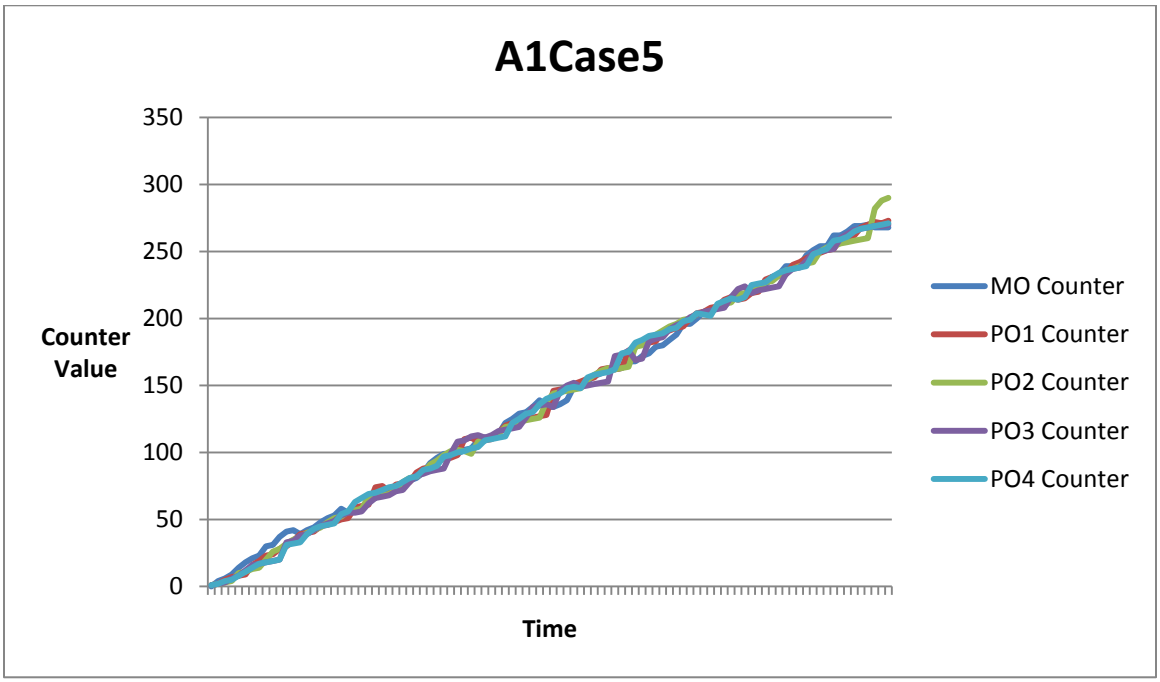

Figure 12: A1 Case 5

Case 6: Byzantine Behavior

Here the PO4 does not start.

probpo1:99; probpo1t:19; probpo2:99; probpo2t:12; probpo3:99; probpo3t:26; probpo4:Does Not start; probpo4t:Does Not start

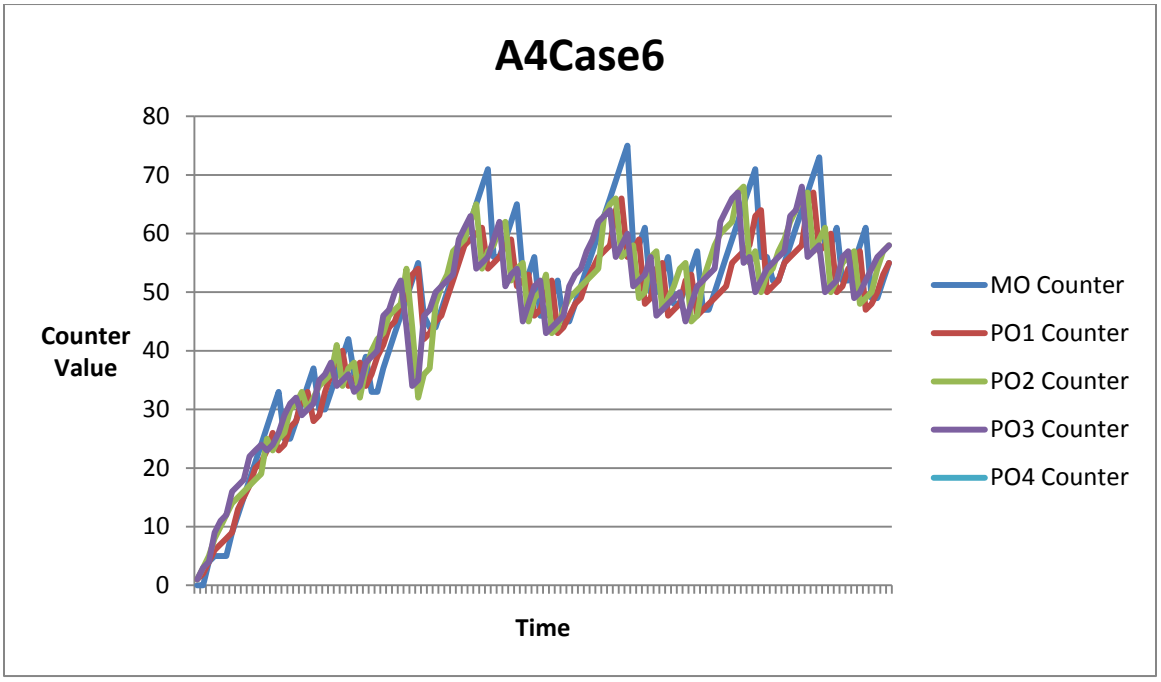

Figure 13: A4 Case 6 Graph

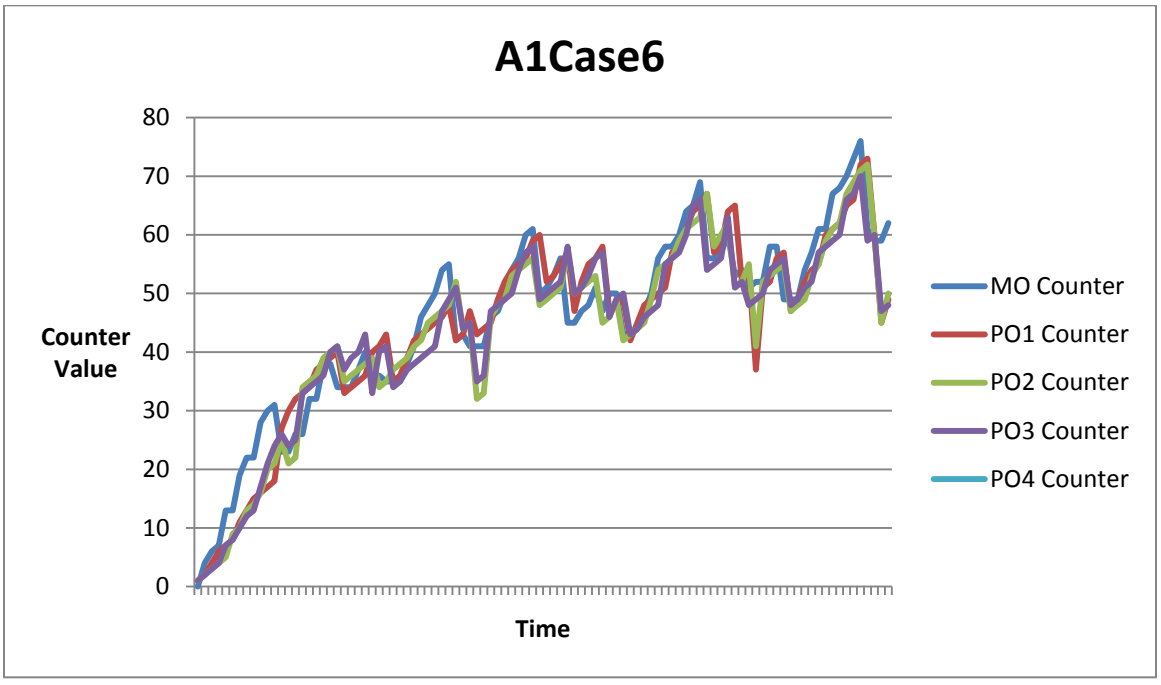

Figure 14: A1 Case 6 Graph

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

The A1 and A4 graphs are similar.

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.

In A4 only one jump is there but in A1 two jumps are there. In A1 it may have happened twice as the failure of PO3 may have sent to some PO and that PO may have forwarded to another PO due to probability variations at that instance and hence double jump as 10000 gets added twice.

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.

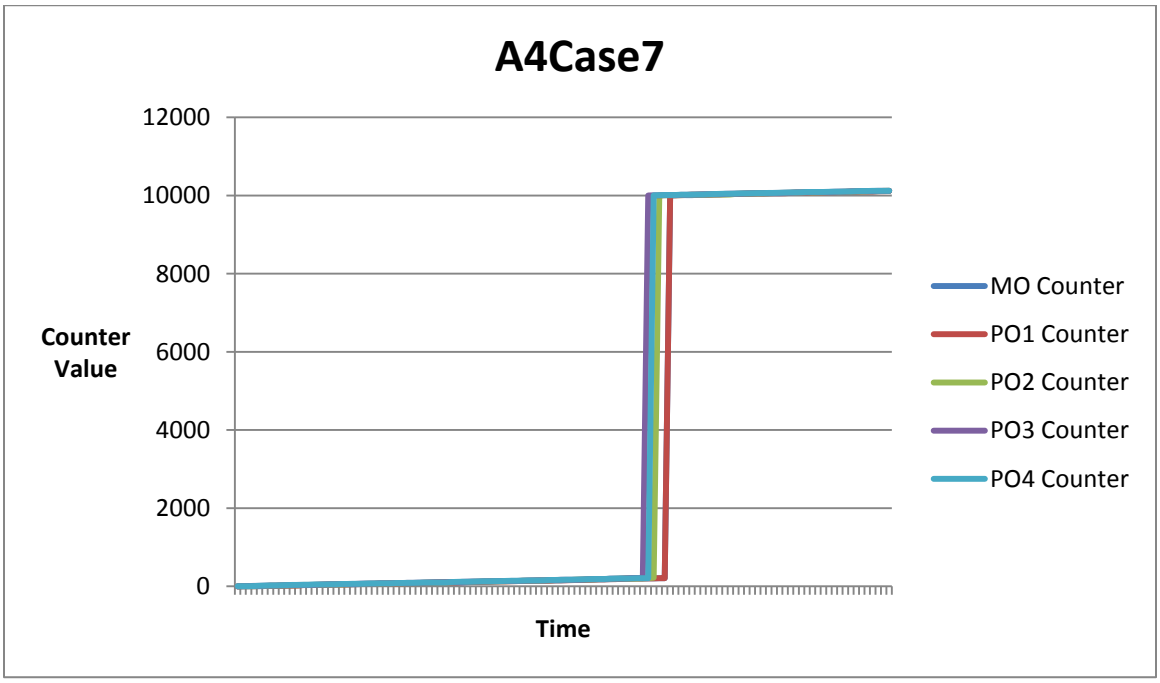

Figure 15: A4 Case 7 Graph

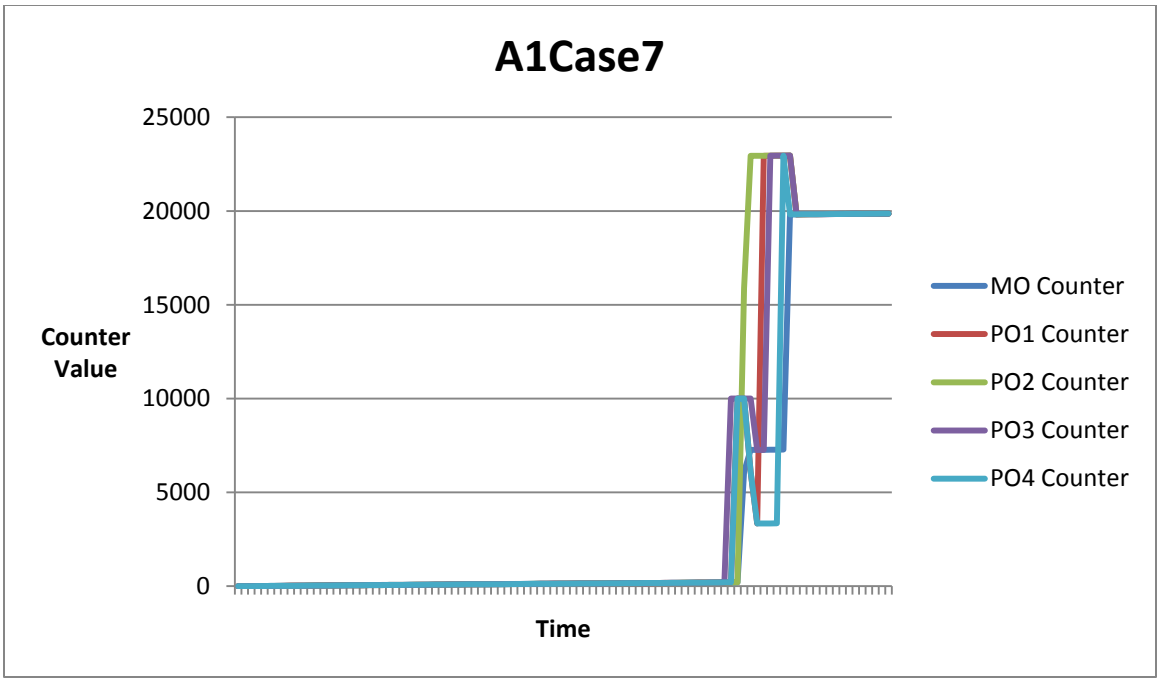

Figure 16: A1 Case 7 Graph

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

Outputs for Each Case:

Case 1:

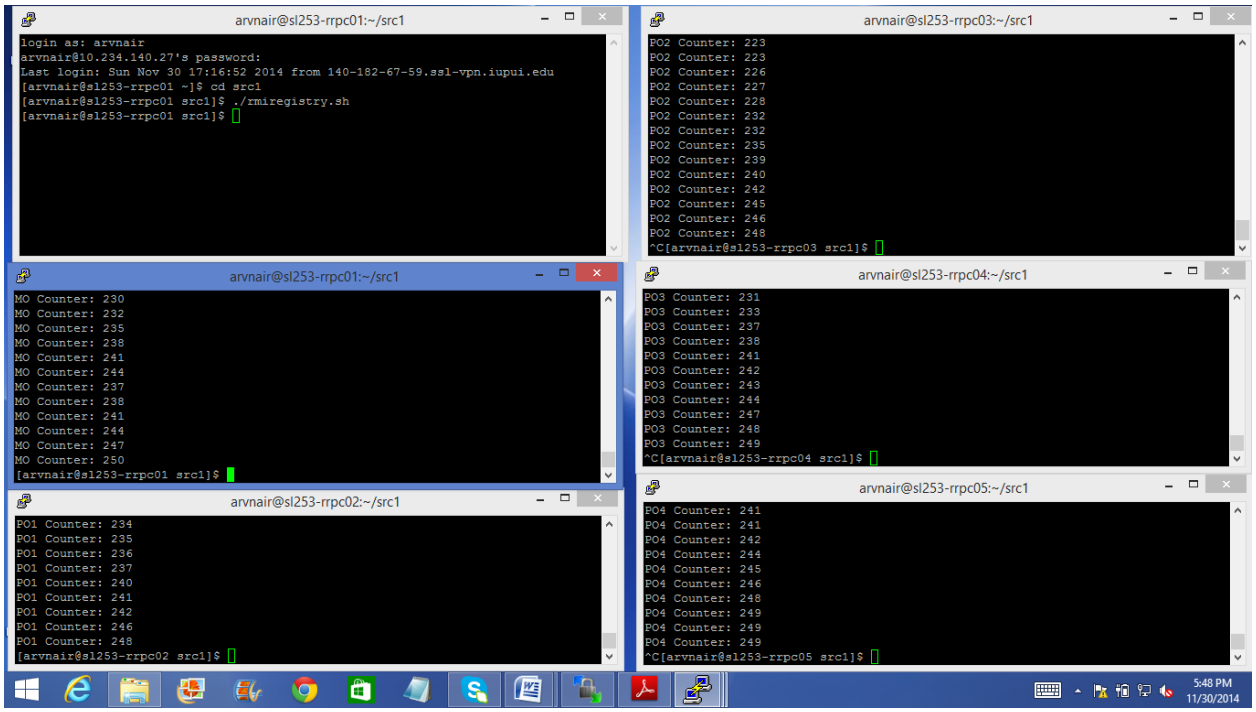

Figure 17: Case 1 Part 1 Screenshot

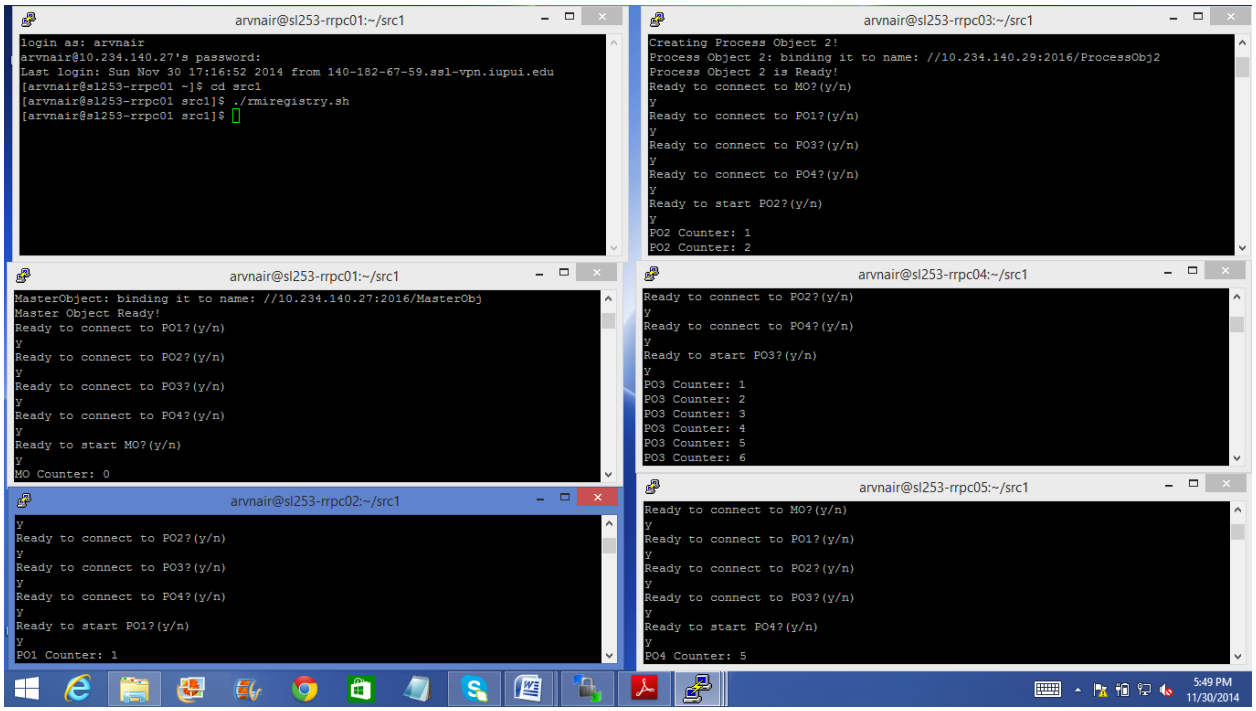

Figure 18: Case 1 Part 2 Screenshot

Case 2:

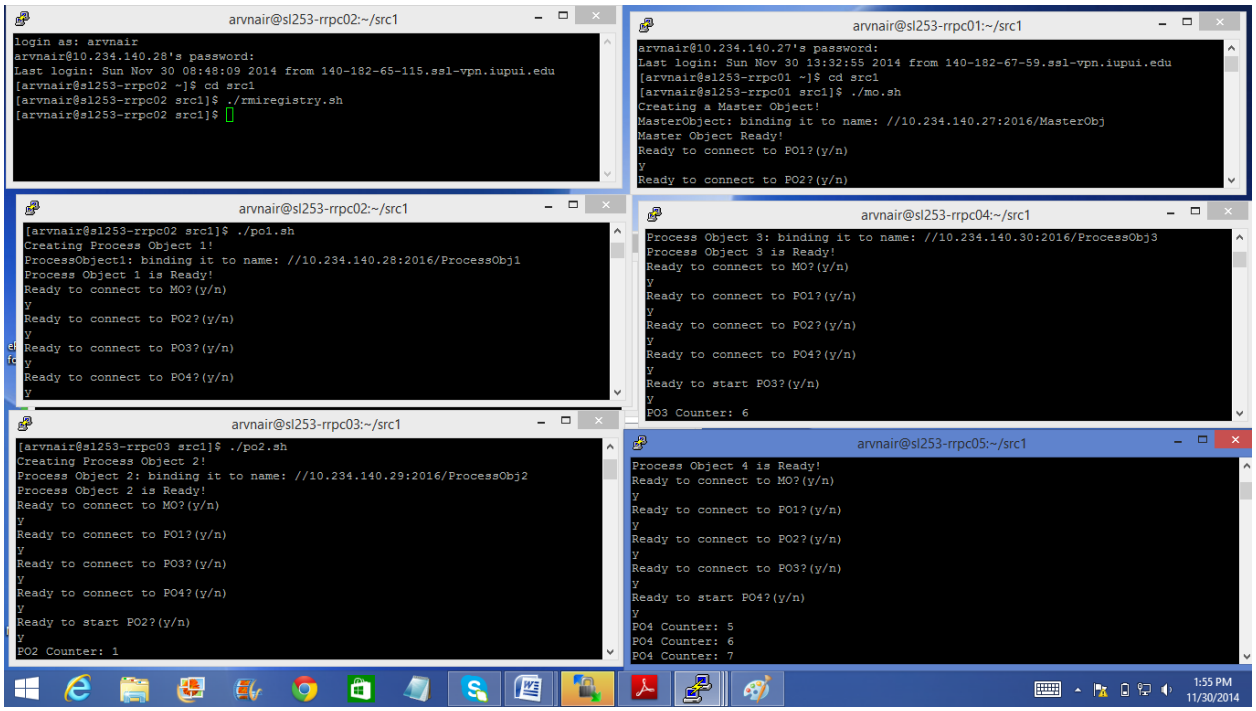

Figure 19: Case 2 Part 1 Screenshot

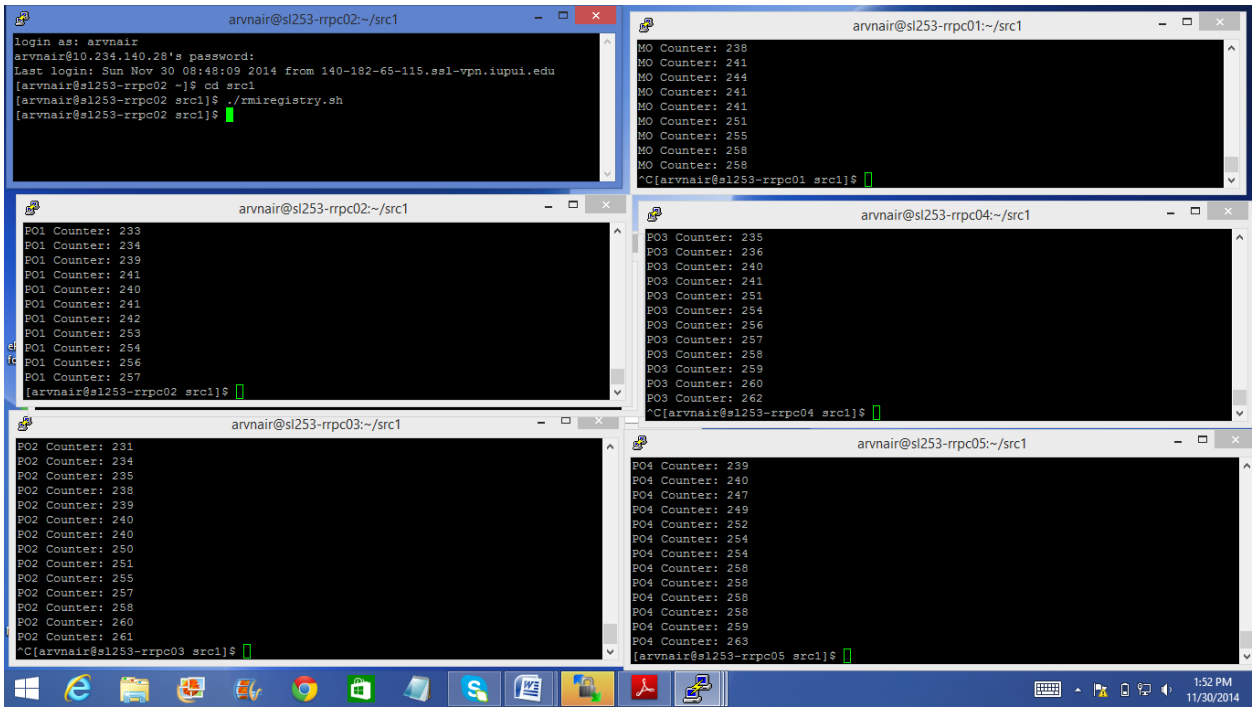

Figure 20: Case 2 Part 2 Screenshot

Case 3:

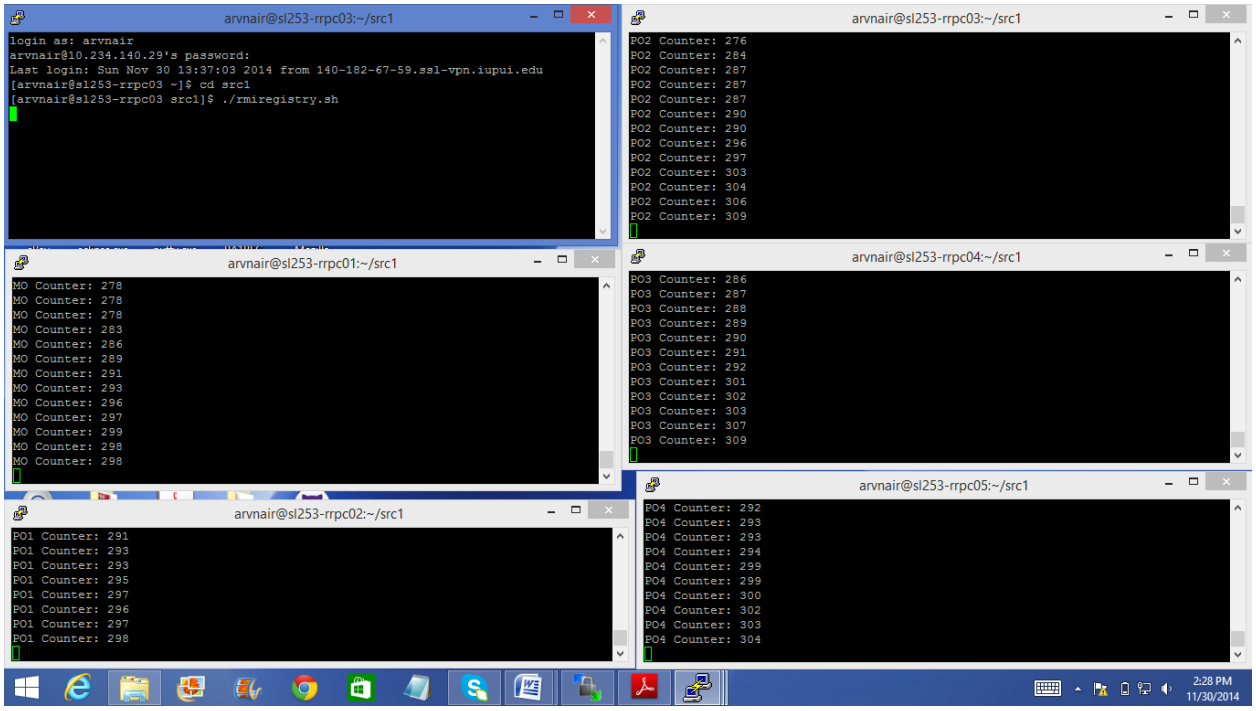

Figure 21: Case 3 Part 1 Screenshot

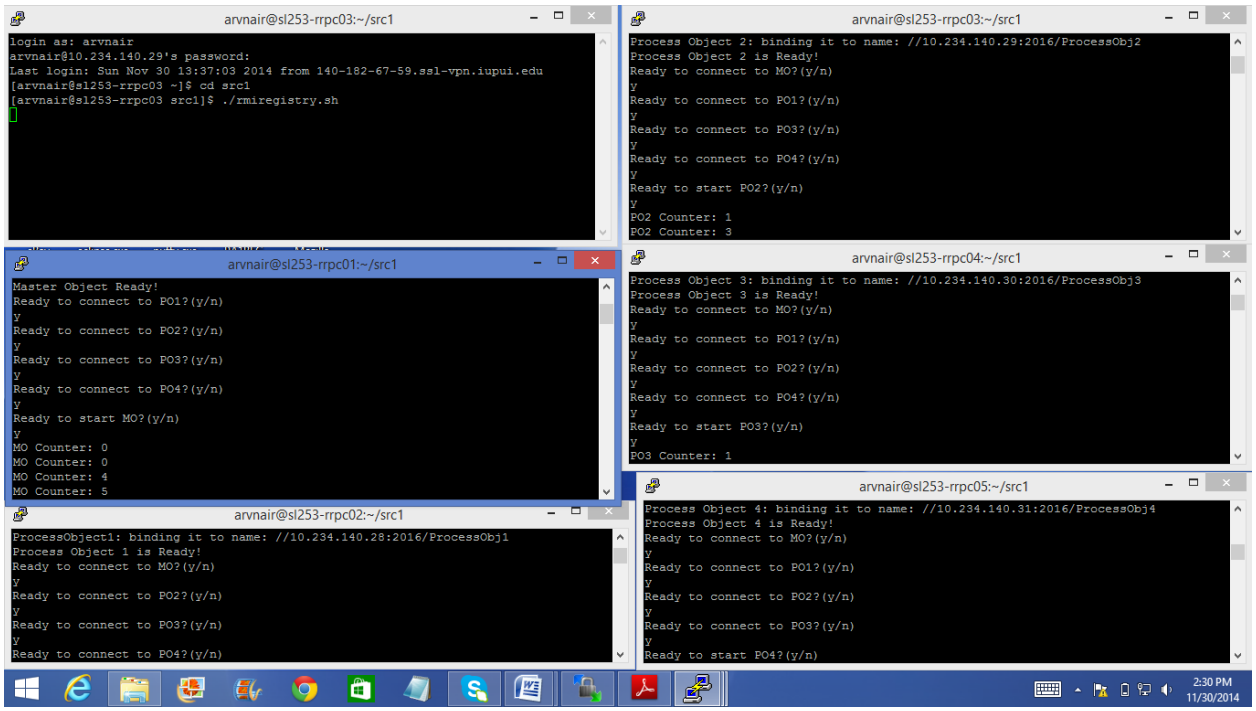

Figure 22: Case 3 Part 2 Screenshot

Case 4:

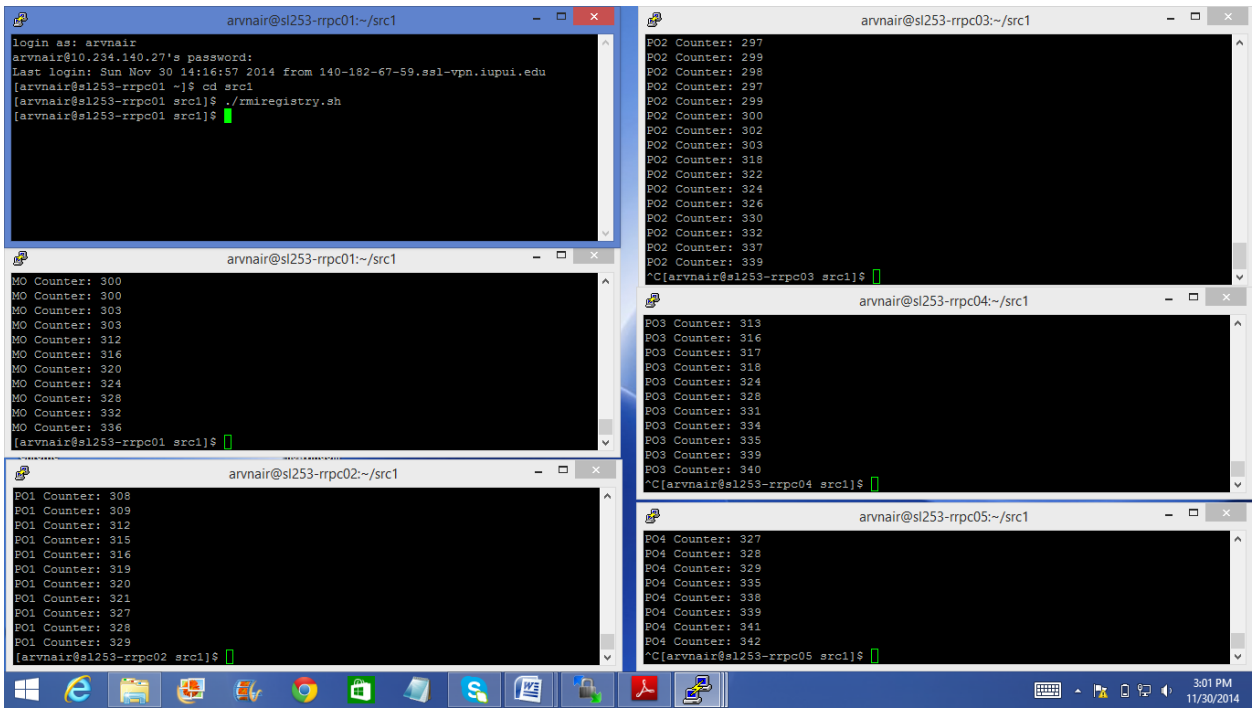

Figure 23: Case 4 Part 1 Screenshot

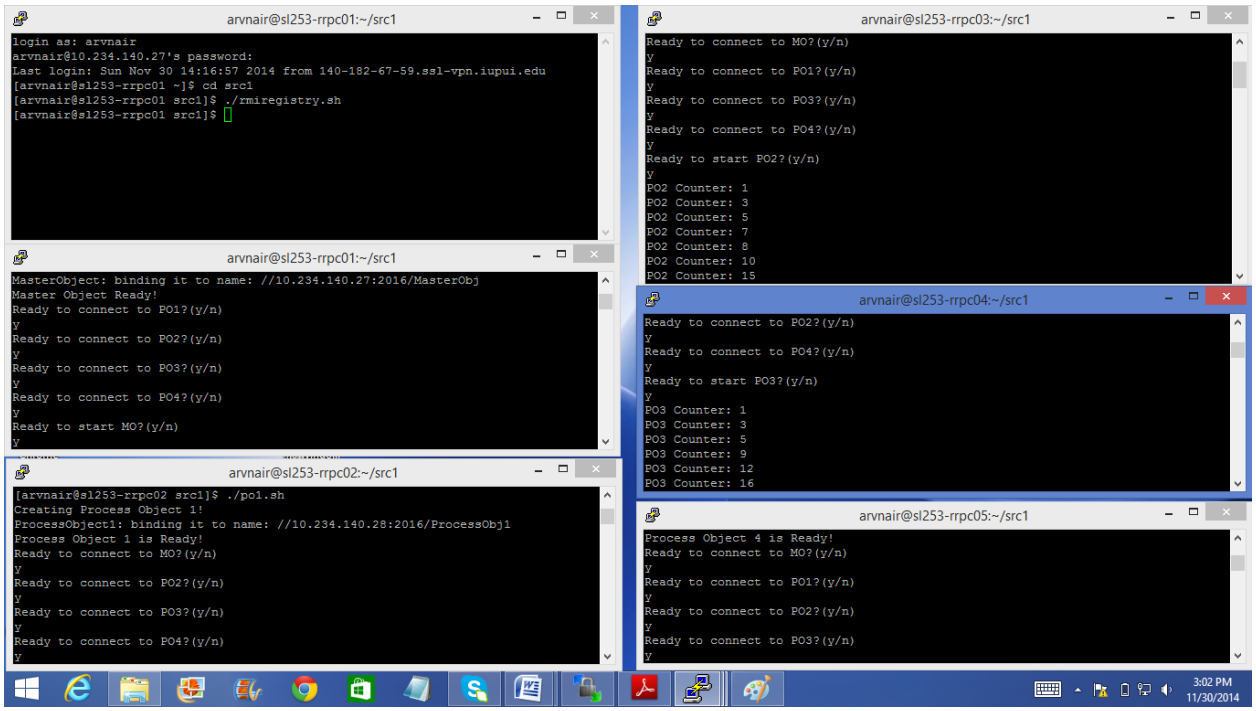

Figure 24: Case 4 Part 2 Screenshot

Case 5:

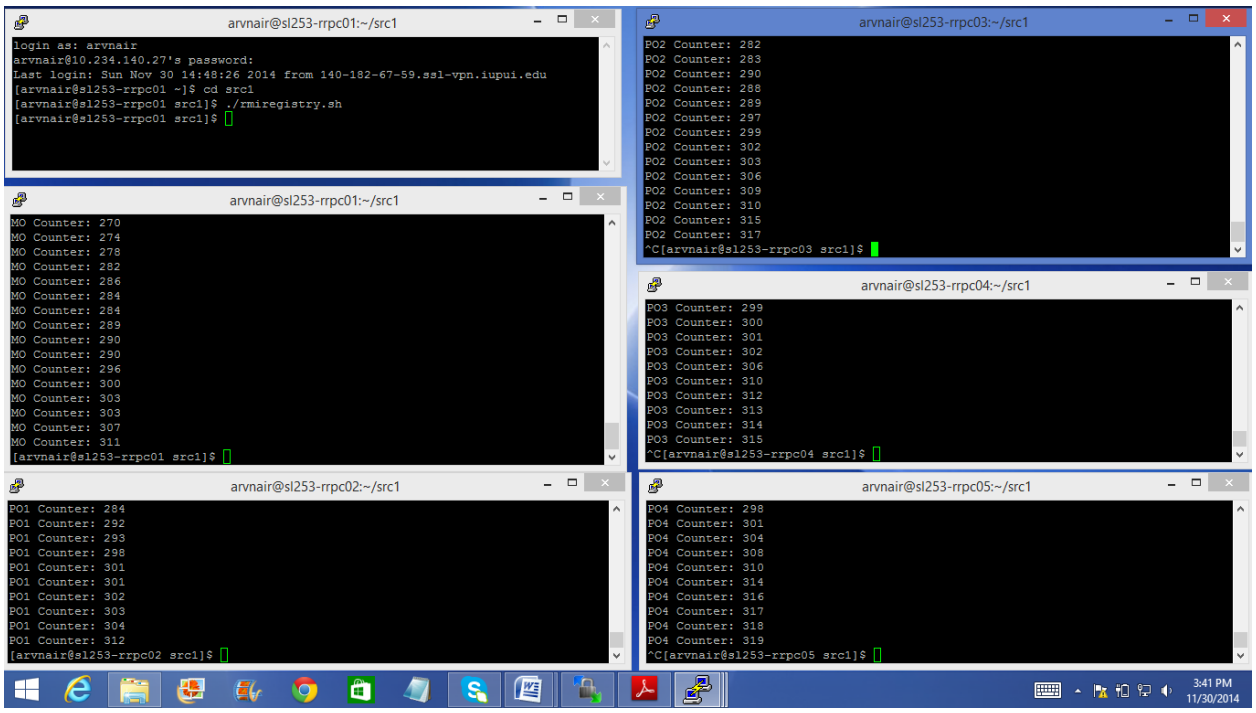

Figure 25: Case 5 Part 1 Screenshot

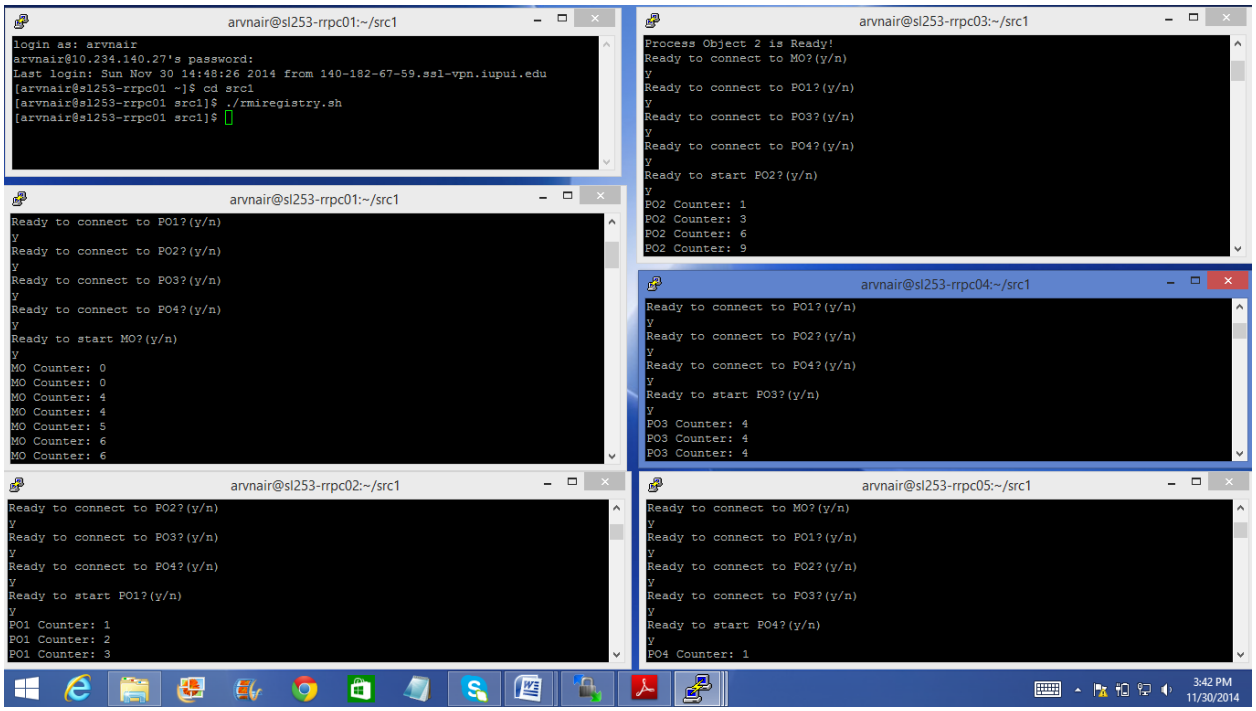

Figure 26: Case 5 Part 2 Screenshot

Case 6:

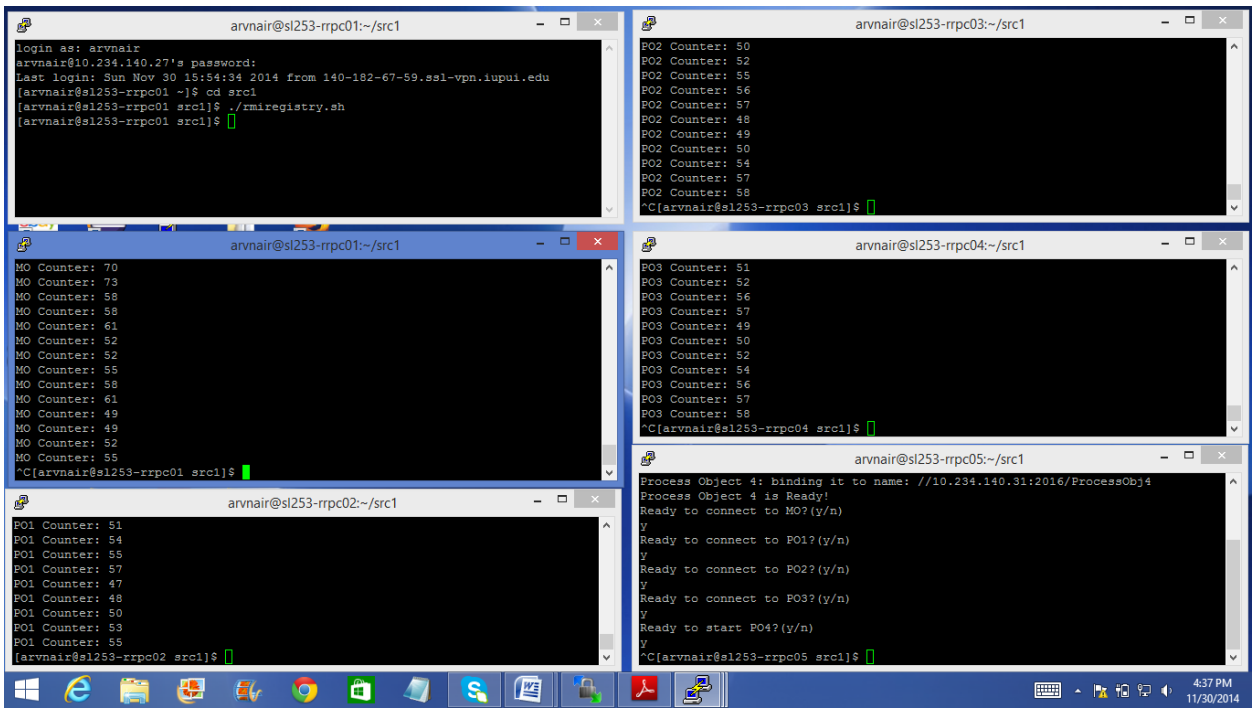

Figure 27: Case 6 Part 1 Screenshot

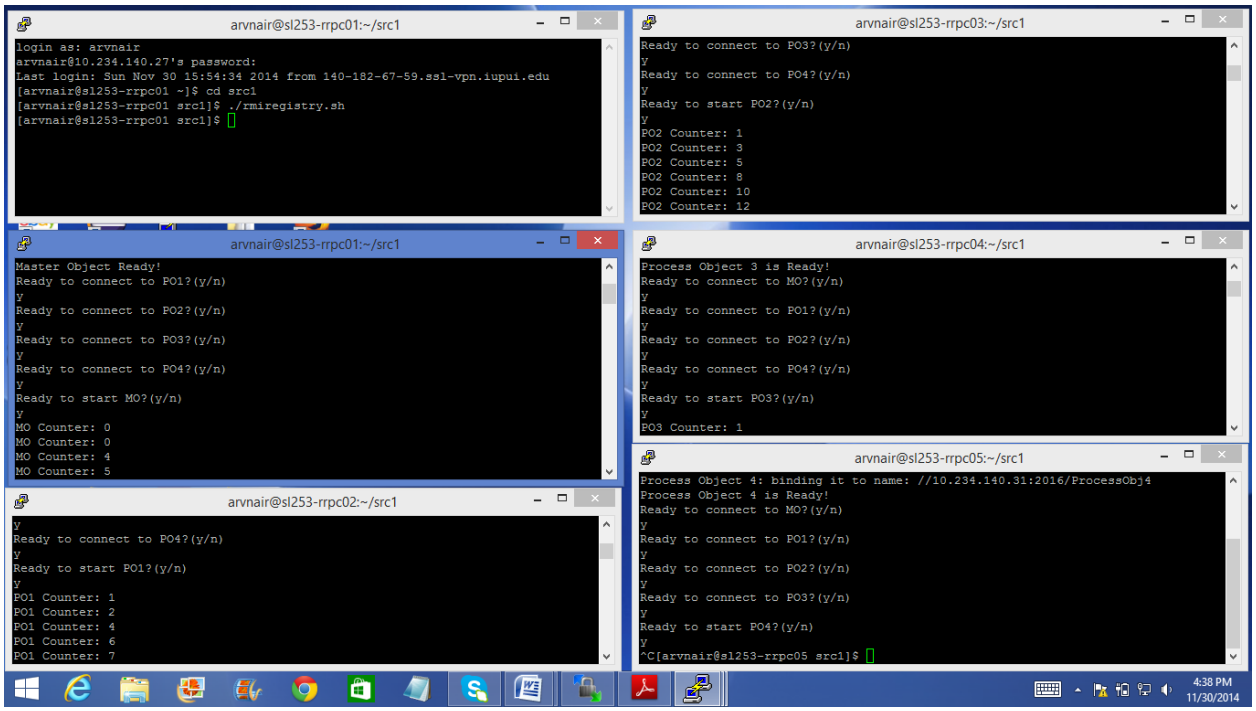

Figure 28: Case 6 Part 2 Screenshot

Case 7:

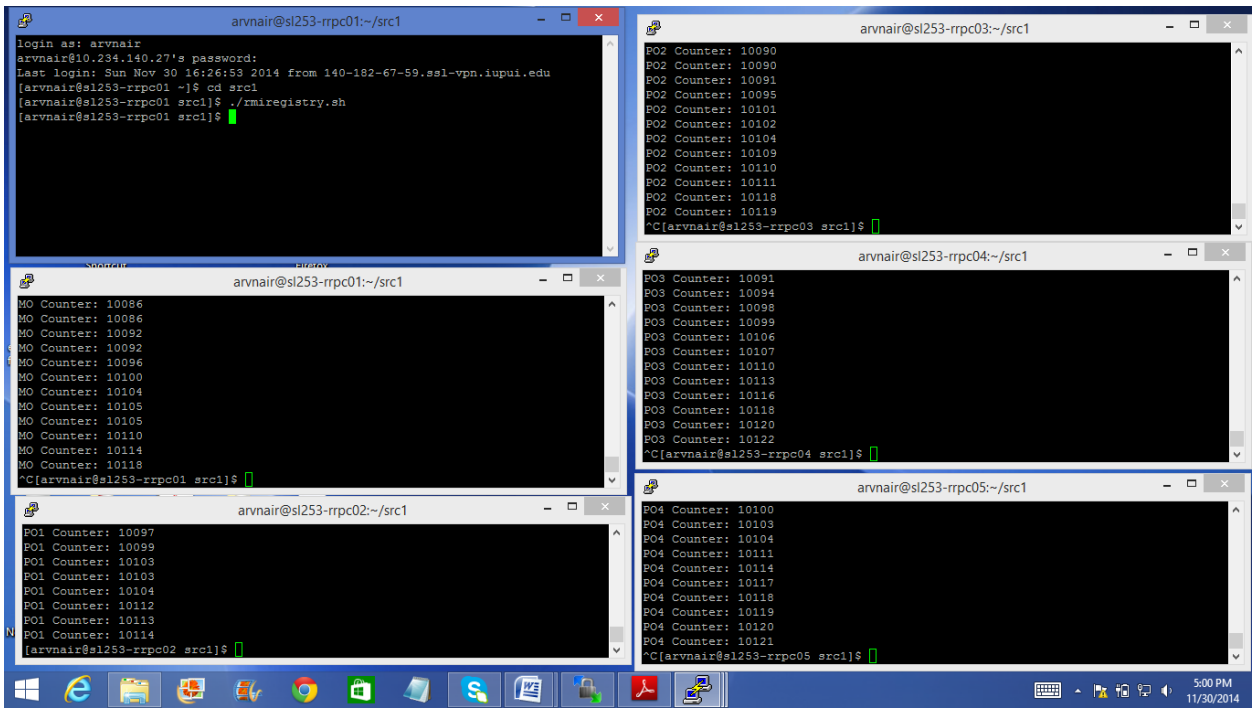

Figure 29: Case 7 Part 1 Screenshot

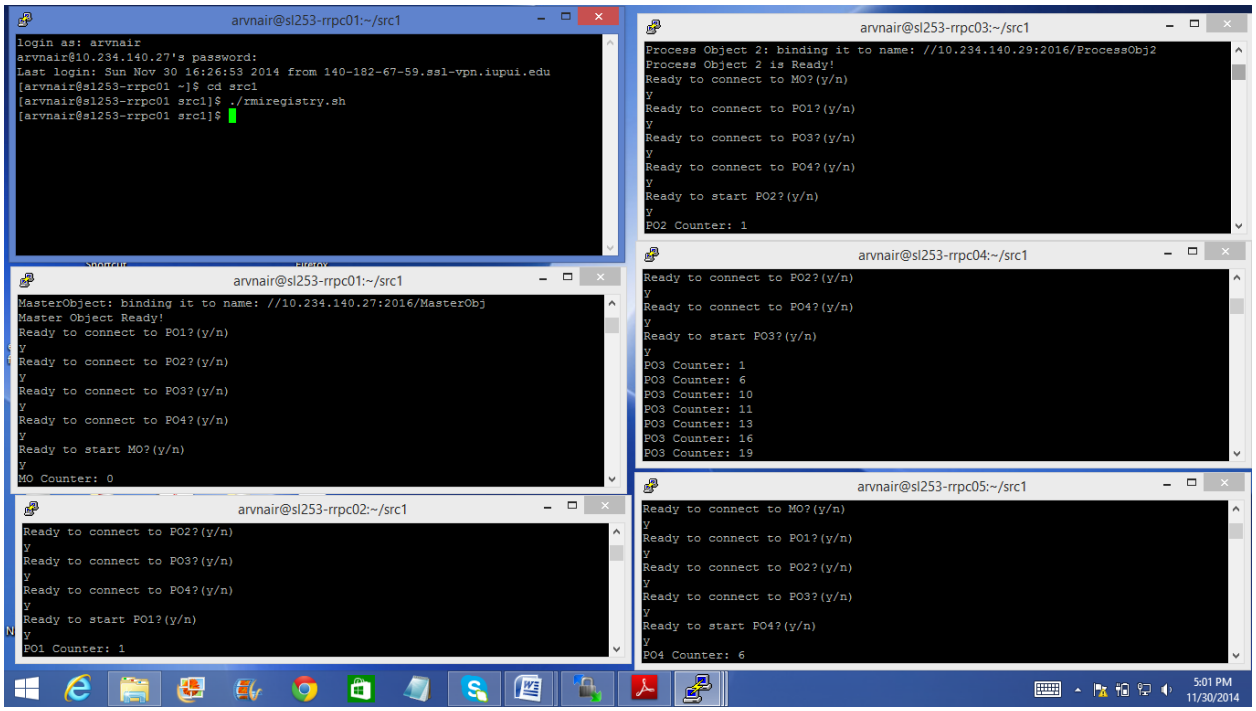

Figure 30: Case 7 Part 2 Screenshot

Conclusion:

To conclude, in comparison with Assignment 1, the Assignment 4 outputs almost remain the same, changing slightly due to manual startup, probabilities and network as it is the same implementation just in an actual distributed environment whereas in Assignment 1 it was a simulation of a distributed environment using threads. The clock drifts in Assignment 4 will be slightly more than the clock drifts in Assignment 1 as it takes additional amount of time to send and receive across the network the clock values and offsets as we are running on physically different machines compared to just a simulation using threads on a single machine.

References:

1. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html[1]

2. *DISTRIBUTED SYSTEMS Concepts and Design Fifth Edition*, George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair, Addison Wesley Publications.

3.Class Notes and Slides on RMI.