**Advanced Operating Systems**

**Project 02 Report**

**Aim**

The objective of the project is to compare the performances of Lamport’s mutual exclusion protocol and Roucairol & Carvalho’s mutual exclusion protocol with respect to Message Complexity, Response Time and System Throughput.

**Methodology**

Both Lamport’s mutual exclusion protocol and Roucairol & Carvalho’s protocol are implemented for a sample distributed application. Then the two protocols are run separately for different configurations of the system parameters namely, number of processes (n), the mean delay between two subsequent requests by a process (d), the mean execution time of a critical section (c) and the max number of critical section requests made by a single process (r).

**Implementation**

Both Lamport’s and Roucairol & Carvalho’s mutual exclusion protocol are implemented in Java using TCP socket programming in a single project. The Lamport’s module and Roucairol & Carvalho’s module are implemented as separate service providers extending a common interface. The application module decides to call either Lamport’s service provider or Roucairol & Carvalho’s service provider in each experimental during runtime depending upon the argument value passed. If the argument value assigned is 0, then Lamport’s service provider is triggered else Roucairol & Carvalho’s service provider is triggered. All mutual exclusion related communication is handled by the service module. Every process has (n+1) threads; 1 for the Application module and n Receiver threads (1 for each node). The application module interacts with the service module only through csEnter() and csLeave() function calls, and also using processMsg() to process the control messages received by the process. After all critical sections have been executed the processes broadcast Termination messages and shutdown after receiving all termination messages.

**Configuration File**

The configuration file is the input to the program and is a plain-text formatted file no more than 100KB in size. Only lines which begin with an unsigned integer are considered to be valid. Lines which are not valid are ignored. The first valid line of the configuration file contains four tokens. The first token is the number of nodes in the system. The second token is the mean value for d (in milliseconds). The third token is the mean value for c (in milliseconds). The fourth token is the number of requests each node should generate. After the first valid line, the next n lines consist of three tokens. The first token is the node ID. The second token is the host-name of the machine on which the node runs. The third token is the port on which the node listens for incoming connections.

**Testing**

Two different mechanisms were used to test the correctness of the protocols. One was using a central shared file where the processes write logs when they enter and exit a critical section. This way no two enters or no two exits should be adjacent. Another testing mechanism used was a vector clock where the clock values are increased during enter and exit. This way when a process is executing the critical section only the its own value can be odd and the values of all others processes should be even.

**Execution Setup**

The application can be run in a process by passing three argument values.

First Argument - Node ID of the process starting from 0.

Second Argument - Path of the configuration file

Third Argument - Service Provider code (0 - Lamport’s code and 1 - Roucairol & Carvalho’s Code)

However, a script (launcher.sh) is created to run the application instance in the machines as mentioned in configuration file. The script passes the necessary arguments for each instance to run successfully in respective machines. Then the cleanup.sh script is run to gather all output data for this run.

**Experimental Setup**

Following are the set up information used in our experimentation

Operating System: Linux

Processor: 64-bit

RAM: 8 GB

Hard Disk Space: 500 GB

Available Hard Disk Space: 310 GB

Programming Language: Java

Socket Protocol: TCP

The experiments were run multiple times for various values of n, d and c using a compare.sh script which creates new configuration files for different values of n, d & c and then gathers all the data. All the experiments were ran for both the protocols and averaged over multiple trials. The results of the experiments are noted and represented in graph in the below section.

n = {5, 10, 15}

d = {100, 200, 500, 1000, 2000}

c = {100, 200, 500, 1000, 2000}

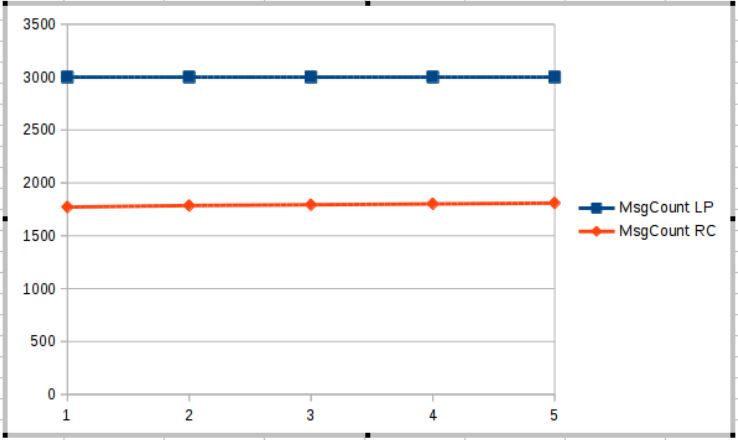
trails = 3

**Observations**

n=10, d=1000, c={100, 200, 500, 1000, 2000}, r=10

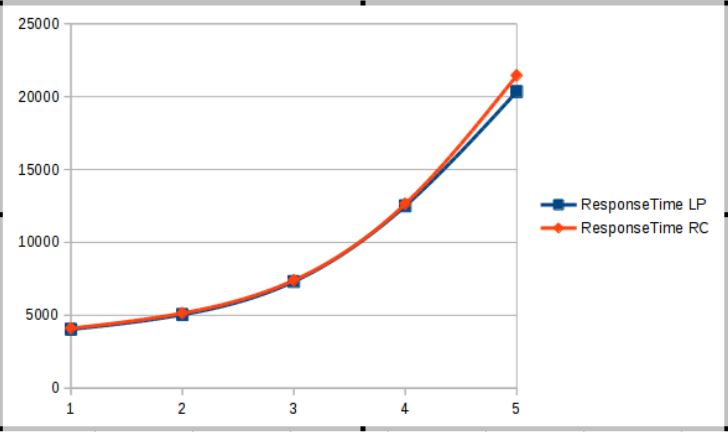
*X-axis : mean value of c {100, 200, 500, 1000, 2000}*

*Y-axis : Message Count*



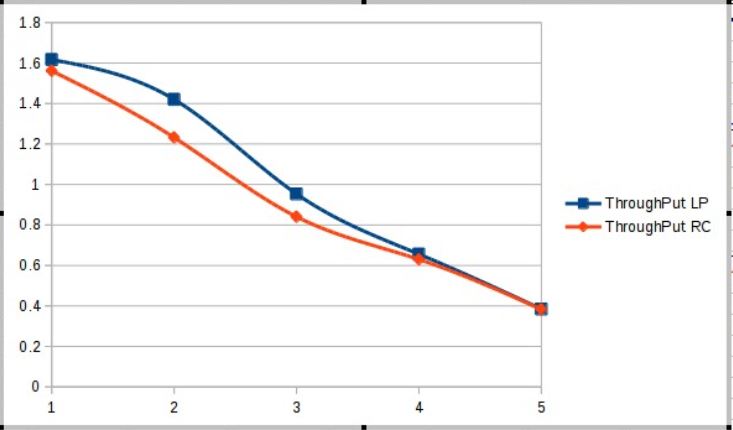
*X-axis : mean value of c {100, 200, 500, 1000, 2000}*

*Y-axis : Response Time*



*X-axis : mean value of c {100, 200, 500, 1000, 2000}*

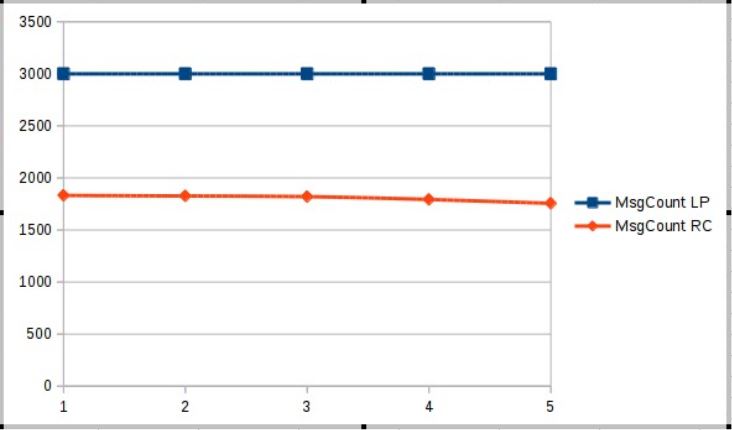
*Y-axis : System Throughput*



n=10, d={100, 200, 500, 1000, 2000}, c=500, r=10

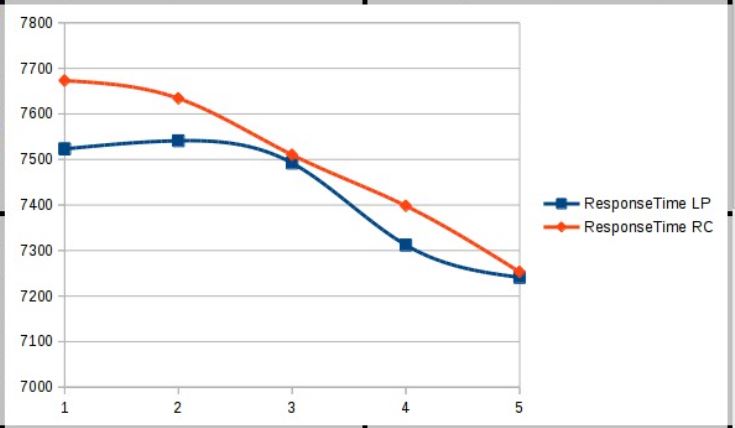
*X-axis : mean value of d {100, 200, 500, 1000, 2000}*

*Y-axis : Message Count*



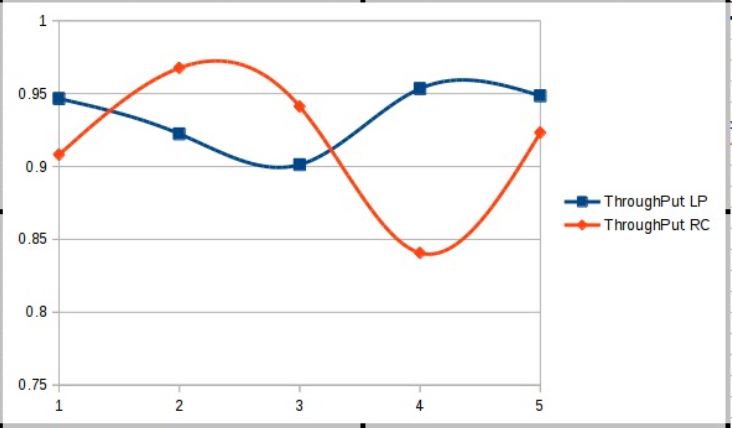
*X-axis : mean value of d {100, 200, 500, 1000, 2000}*

*Y-axis : Response Time*



*X-axis : mean value of d {100, 200, 500, 1000, 2000}*

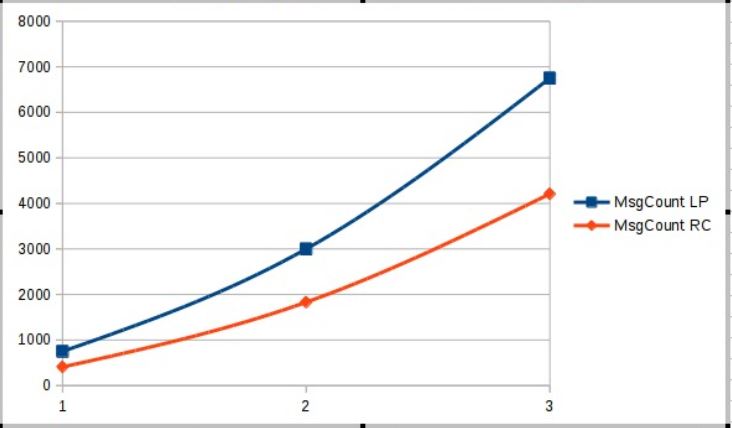
*Y-axis : System Throughput*



n={5, 10, 15}, d=100, c=100, r=10

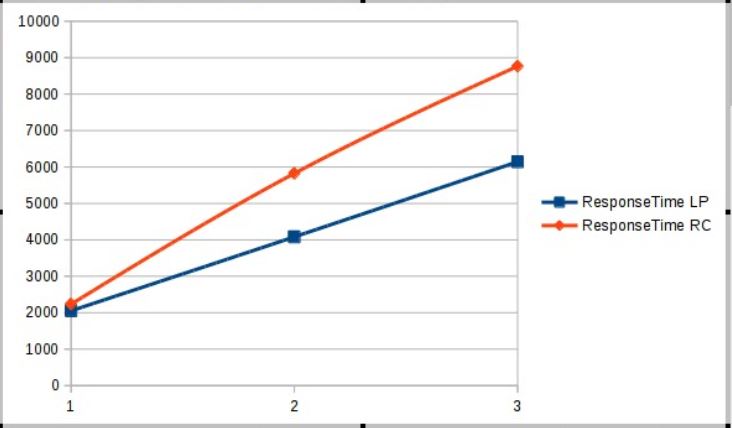
*X-axis : n {5, 10, 15}*

*Y-axis : Message Count*



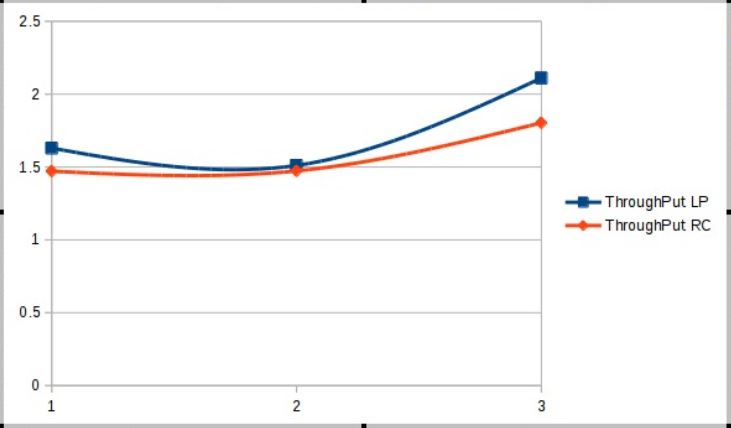
*X-axis : n {5, 10, 15}*

*Y-axis : Response Time*



*X-axis : n {5, 10, 15}*

*Y-axis : System Throughput*



**Conclusion**

Thus the message complexity of Roucairol & Carvalho’s protocol is much lesser (almost two-thirds) than that of Lamport’s protocol. At the same time, Lamport’s protocol has slightly lesser response time and slightly higher throughput than that of Roucairol & Carvalho’s protocol.