Report for distributed systems project

Tanmay Sachan (2018111023)

Project Description

The aim of the project was to analyse various mutual exclusion algorithms in a distributed setting under message delivery guarantees such as FIFO and non-FIFO.

Video for the project explanation - https://youtu.be/o_CfStus0J8

Framework and Languages used

The algorithms were written in Python using the OpenMPI framework for message passing between processes.

Algorithms Implemented

1. Lamport's Algorithm - Uses Timestamped messages and queues within each process to order requests. If a process is at the top of its queue, and it receives a message from all other nodes later than its request, then it gets to execute the critical section.
2. Ricart Agarwala - Uses a defer queue to defer reply to requests of critical sections. When a process is executing the CS, or has a higher priority for CS over other requests, it simply does not reply and stores the deferred reply in a queue, to be sent when it is done executing CS itself.

3. Roucairol Carvalho - An optimization over Ricart Agarwala that allows a process to re-enter CS when it has just finished executing CS and no other process is requesting for it.
4. Suzuki Kasami - It is a token based algorithm. Uses sequence numbers on every process to know which CS they will be executing next. Uses a queue on the token to redirect it to the next site. The sequence numbers help separate old requests from new requests.
5. Raymond’s (The nodes are arranged in a binary tree) - It is a token based algorithm. Uses a spanning tree of processes to redirect the token to each site that requests it. Efficient travel of token when the load on the system is high.

Of the algorithms covered in the course, only Ricart and Roucairol support NON-FIFO.

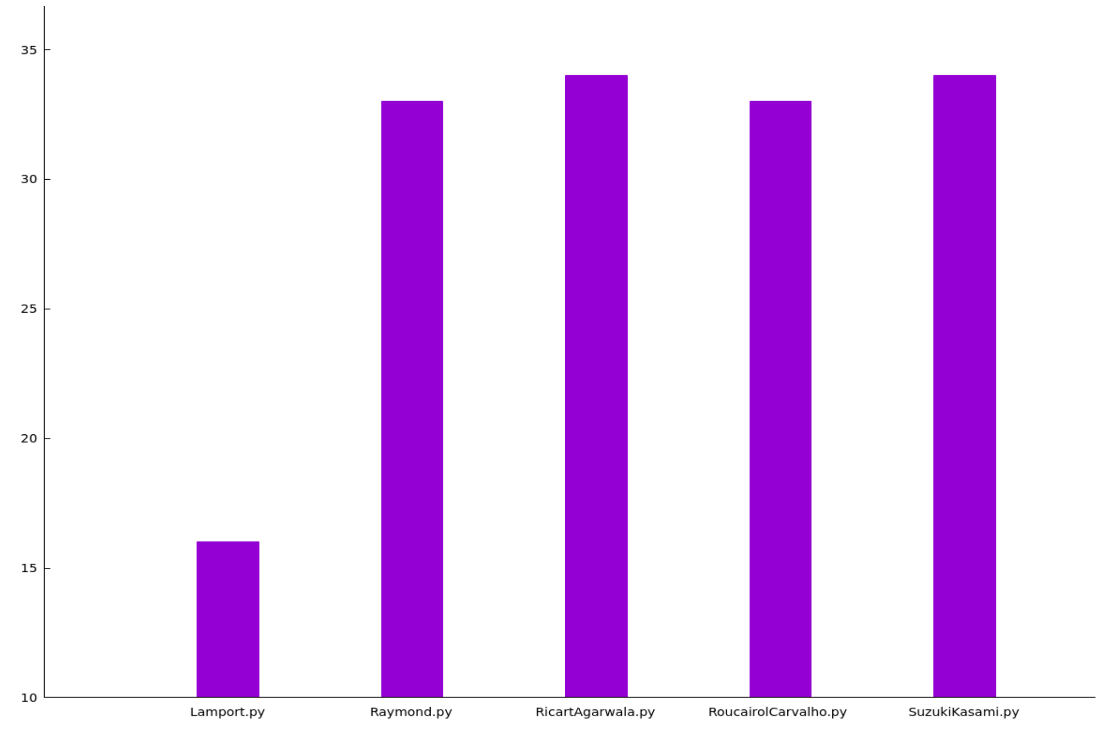
Benchmarking

The algorithms were benchmarked based on the number of critical section executions within a fixed time interval (100 seconds).
To simulate time delays, a normal random time interval between 0 and 1 second was used each time a message was sent.

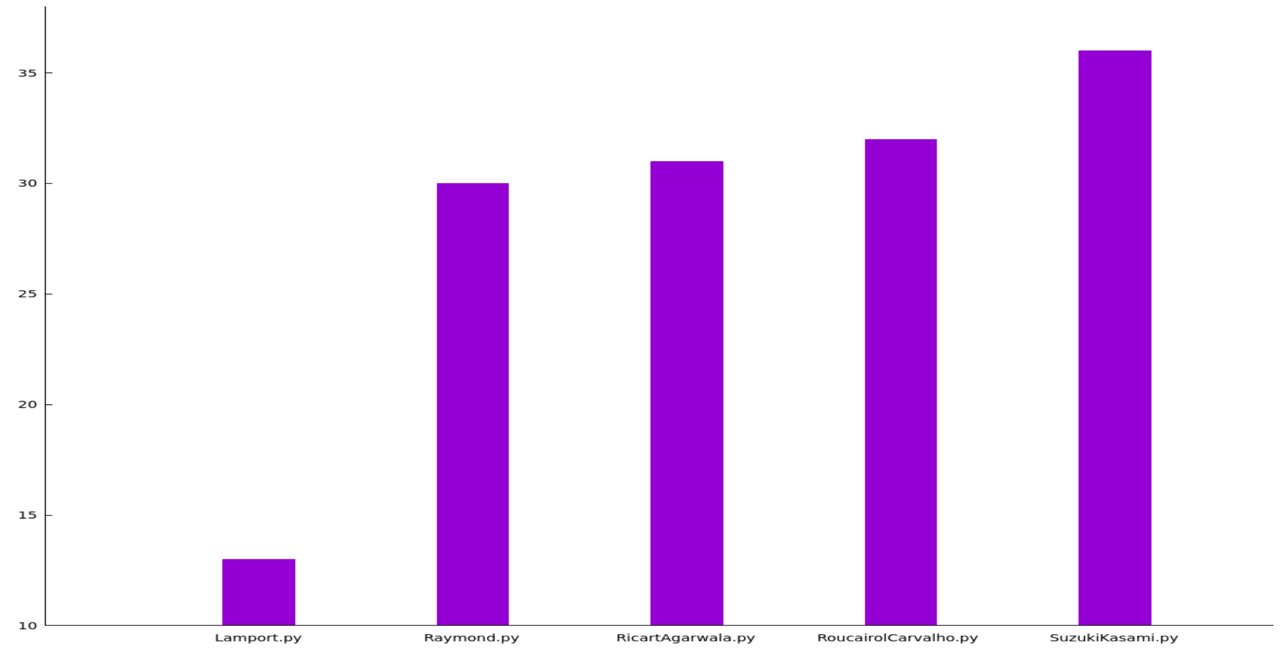
The performance of the algorithms were measured under different conditions, which are graphed below.
(High load refers to a high CS request count, there almost never exists a time where the system has no pending request for CS)

The Y-axis on the graphs represents the number of critical sections executed in 100 seconds of run time.

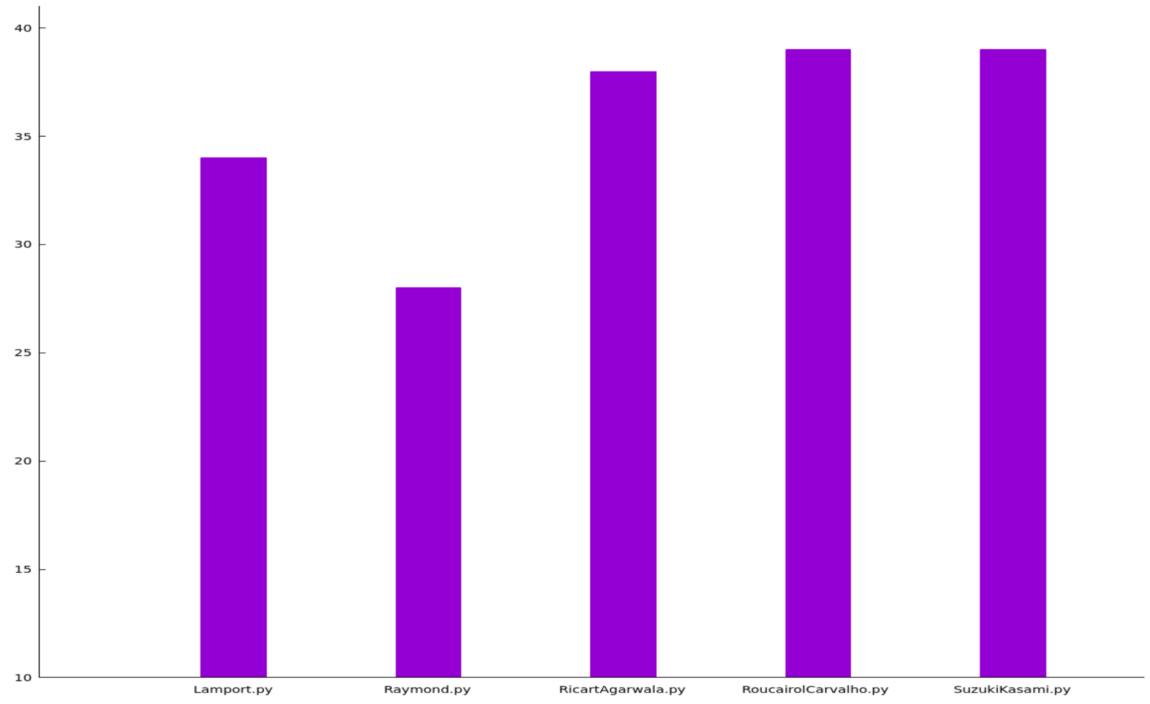
FIFO, High load, High number of processes



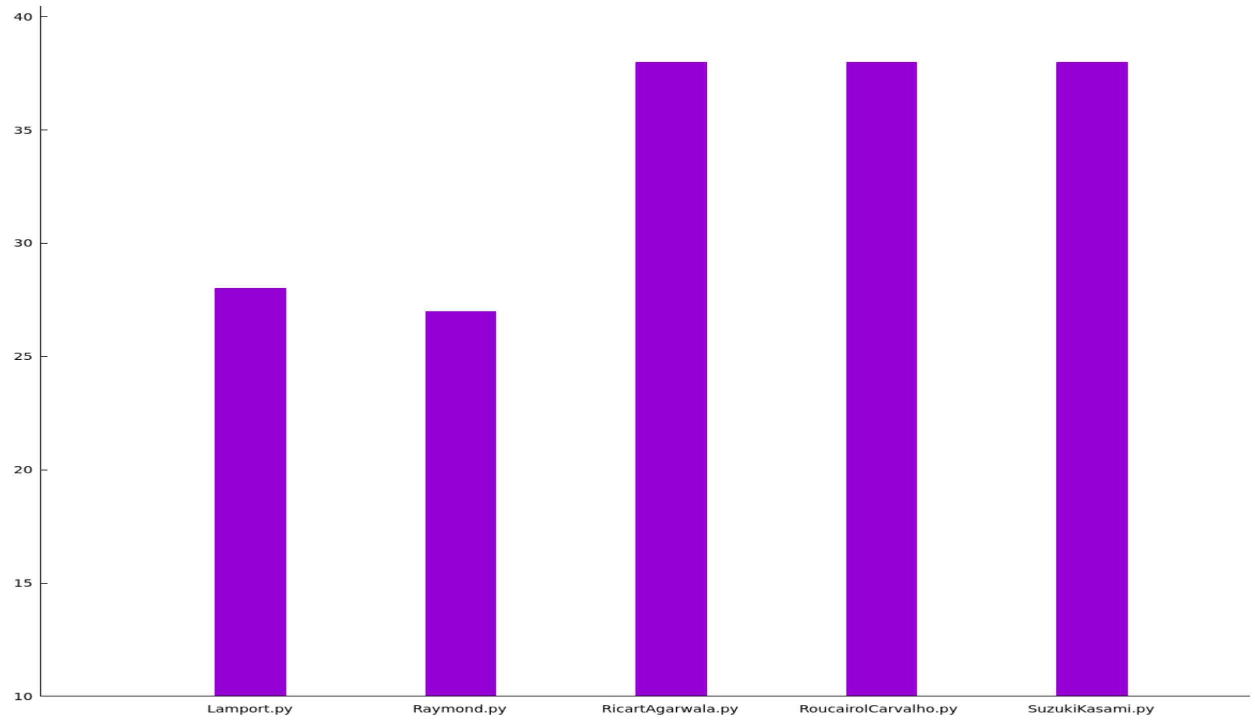
FIFO, Low load, High number of processes



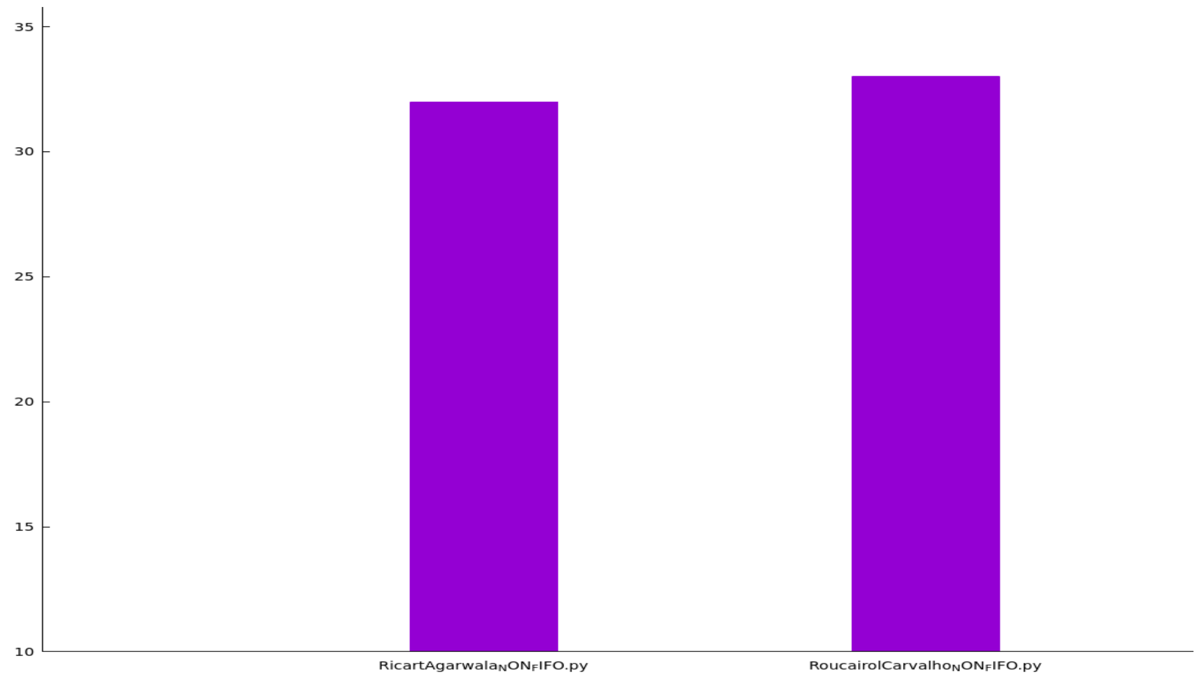
FIFO, High load, low number of processes



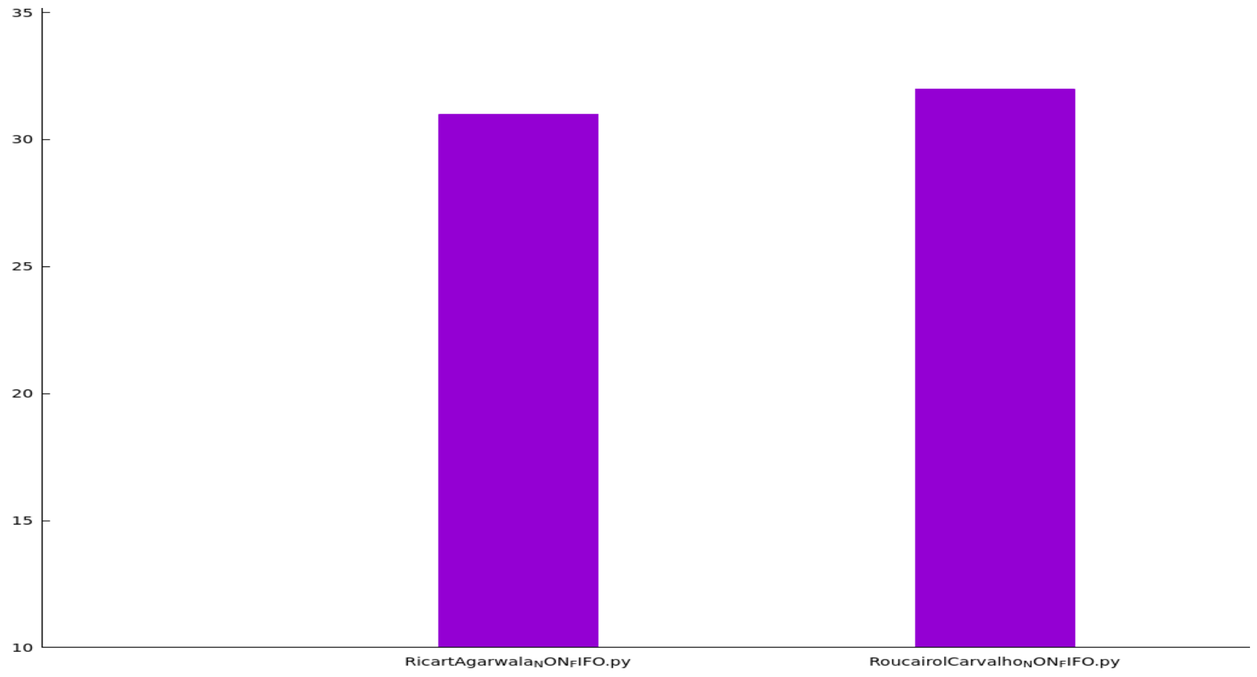
FIFO, Low load, low number of processes



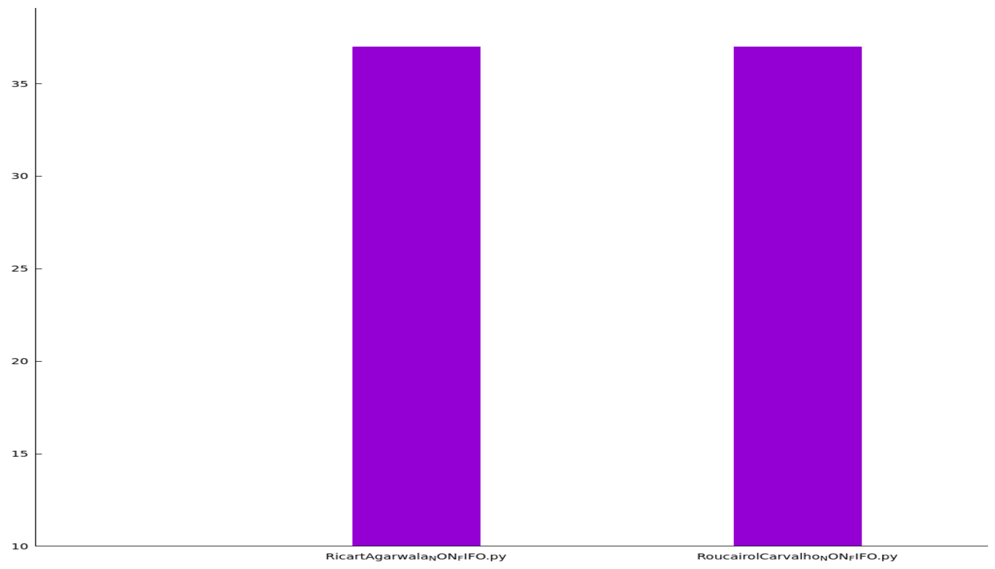
NON-FIFO, High load, High number of processes



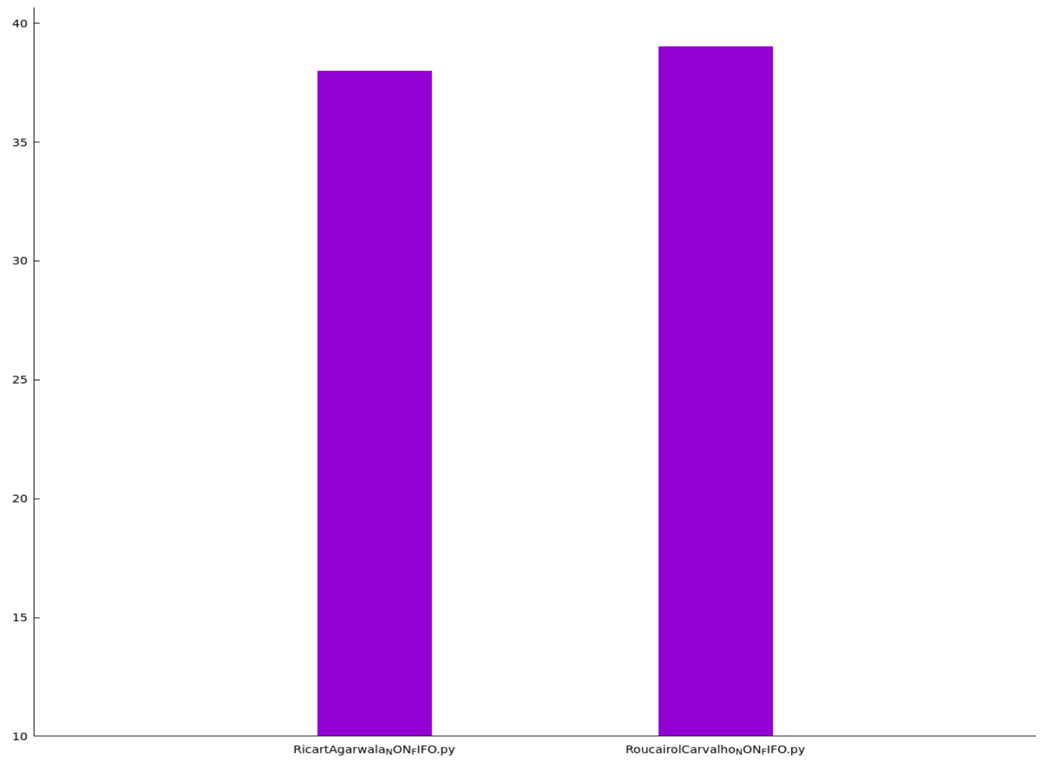
NON-FIFO, Low load, High number of processes



NON-FIFO, High load, Low number of processes



NON-FIFO, Low load, Low number of processes



Results and Conclusion

From the various graphs we see the behaviour of the different algorithms with respect to each other.

We can see Lamports underperforms with a high number of processes, since it has the most number of message transfers before allowing critical section entry.

When we have a low number of processes, we see the performance of all the algorithms goes up.

Suzuki kasami throughout stays as the most performant algorithm.

We also see the performance of Raymond’s drop considerably when we switch from high load to low load. This can be attributed to the fact that messages have to travel much less in a high load system in Raymonds.

Since we have only 2 algorithms here that can perform in a non-fifo setting, we benchmark them here. However the graphs are very close and there’s not much conclusion that can be drawn here.