CS6378: Project 2

Lamport's M distributed Mutual Exclusion Algorithm

Team:
Rongala, Nikhil
Bandarupalli, Sri Harsha
Yeturu, Pavan Preetham

Initial setup

- Each process reads its port and index from the configuration file and then starts up a server on that port and waits for other processes to connect. Once n-1 connections are made, it broadcasts its node index.
- Once this is done, every non-zero node sends a START message to node zero, signaling that its ready to start. Node zero, after receiving all n-1 START messages and itself ready to start, sends a START message to all other nodes signaling to start the application.

Implementation

- The project was coded in Java using SCTP channels that ensure that FIFO property is followed as it's the assumption for the algorithm to work correctly.
- The clock used is scalar clock.
- Every process has an object of class Mutex which stores a priority queue of requests.
- The Mutex class has a method *cs_enter( )*, which the application calls before executing its critical section. The method ensures that
    - L1 (the last received message timestamps from all other processes is greater than the timestamp of the request) and
    - L2 (head of the priority queue is the application's request) are satisfied.
- The *cs_leave()* method pops the head of the priority queue and broadcasts release messages.
- *Termination*: After a non-zero process finishes all its requests, it sends a message an END message to node-zero. However a thread still keeps running to act appropriately on receiving any message. Once node zero is finished with all its requests and received n-1 END messages, it broadcasts another END message and all processes terminate on receiving that.

Verification:

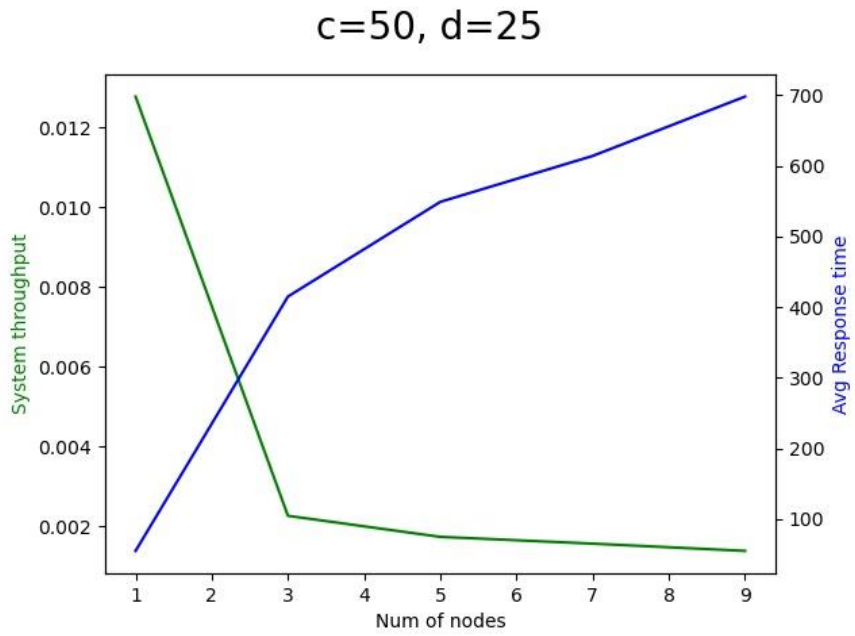
- To verify that no critical section request is executed at the same time as another, some information is gathered during the critical section execution and this information is sent to node zero after the execution of every request, which calls the verification routine before coming to a halt.
- The information sent by non-zero node or stored by node zero is:
    - A vector of last received message timestamps just before the critical section execution
    - The CS request
    - Clock before execution
    - Clock after execution
- The verification routine:

- Node zero maintains a priority queue of all the *completed* requests which are either sent by other nodes or stored by it.
- After all the requests have completed, node zero checks for each request in the priority queue:
  - If the last received message from the previous request's node has a timestamp greater than the clock after the previously executed request and
  - The clock before execution of the current request is greater than the timestamp of the last received message from the previous request's node.
- These 2 conditions verify that the clocks are incremented correctly, there is no intersection of requests and also that they were executed in order of their timestamp.

Experimental Evaluation

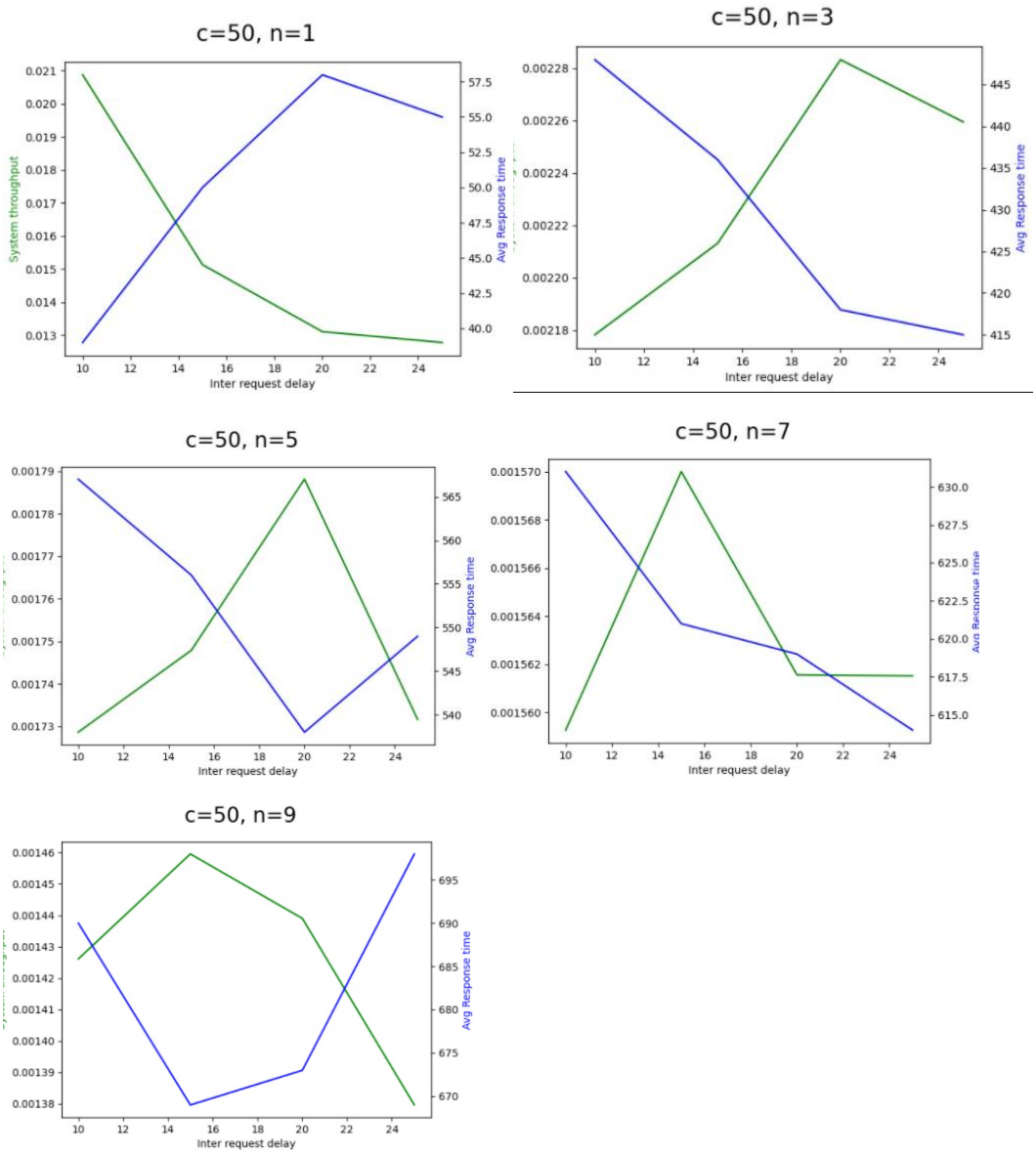
- The project is run for various values of n (number of nodes), d (inter request delay) and c (cs execution time).
- The range of values for
  n: [1,3,5,7,9]
  d: [10,15,20,25]
  c: [10,20,30,40,50]
- The message complexity is always 3*(n-1) as each request requires a REQUEST, REPLY AND RELEASE message. The average response time and throughput are plotted with varying n, d, c.

Varying n:



For all values of d and c, the graph follows the same pattern i.e. the avg response time increases and the system throughput plunges with increasing number of nodes. Apart from when there is only 1 process, the elevation or drop in the plots for n>1 doesn't have a steep/deep variation.

Varying d:

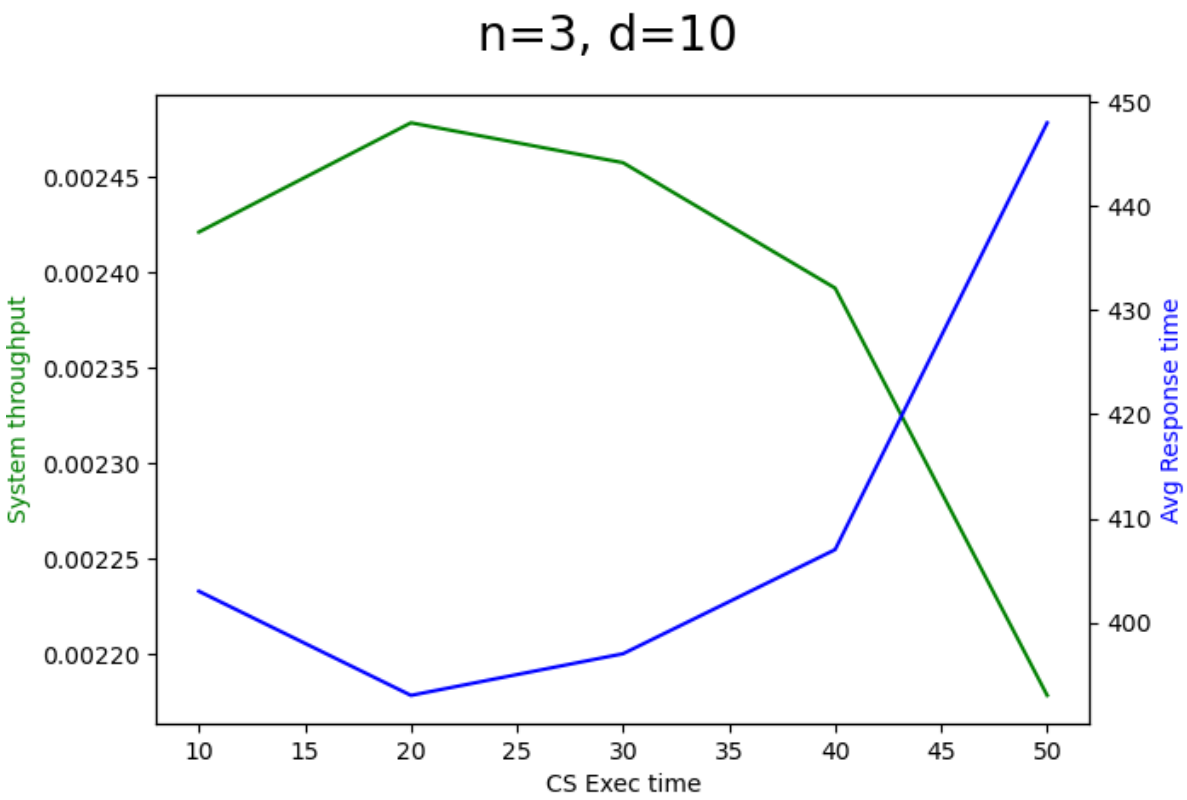


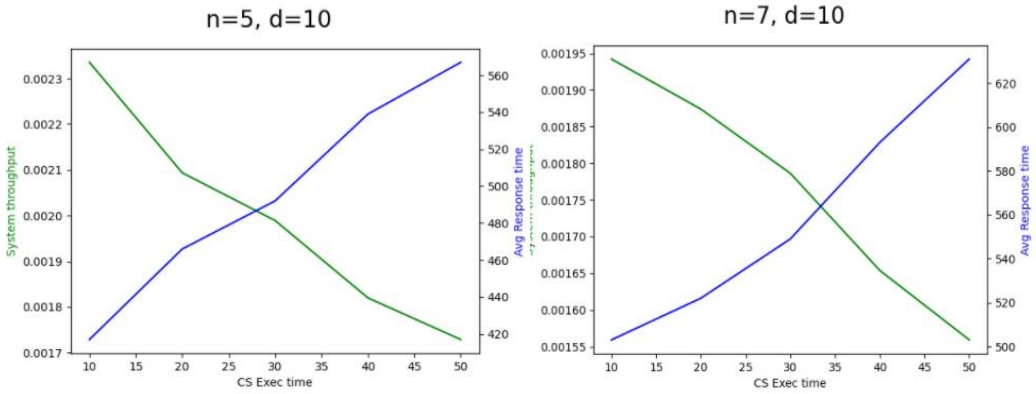
These plots suggest a very confusing pattern. When d is increased, clearly the system throughput takes a hit, but it seems to peak before that. These may be because of the gaps in time created by the inter-request delay being filled by the other processes.

If we assume that there exists a pattern and observe the average response time plots, each plot can be seen as showing us a part of the pattern, which is that it first drops and then keeps increasing with increasing d (n=1 is the exception). This is in line with the way system throughput changes as they are inversely related.

Varying c:

After observing the plots for various values of n and d, most of the plots suggest a directly proportional and exponential relationship between avg response time and cs execution time, and an inversely proportional exponential relationship between system throughput and cs execution time. This is most probably due to the exponential probability distribution of csExecution time.





END of REPORT