Distributed Computing Course Project

N Venkatasubrahmanian, EE14BTECH11018,
Arjun D'Cunha, CS14BTECH11039

April 26, 2017

1 Problem

Compare the performance of Singhal's dynamic information structure algorithm and Lodha-Kshemkalyani's fair mutual exclusion algorithm on various metrics.

2 Filenames

Due to Java restrictions, the filename of a program must match the main class name used within it. Hence, the two submitted programs have the names "Singhal.java" and "Lodha.java".

3 Network Topology

Both the algorithms under consideration for this project require fully connected underlying networks to operate. As part of this project, we permit arbitrary(but connected) network topologies, and using a supplied spanning tree calculate a forwarding table for messages.

4 Description - Singhal's algorithm

Singhal's algorithm is an optimization of the Ricart-Agrawala algorithm where nodes try to dynamically track which other nodes they should obtain permission from to enter the critical section(referred to as the dynamic

information structure). Initially, nodes are assigned "priorities" based on their process IDs, such that the lowest priority node needs the permission of every other node, while the highest priority node would need no permission to enter the critical section. As time passes, the nodes track who they need to get permission from to enter the CS, and the algorithm tries to ensure that once the system has, in a sense, stabilized, nodes that frequently enter the CS would only request permission from other nodes that frequently enter the CS. The algorithm maintains 2 data structures - a request and an inform set. The request set tracks which nodes permission to enter the CS has to be obtained from, while the inform set tracks which nodes need to be notified when this node exists the CS. The algorithm uses a Lamport clock to act as a distinguishing priority. Nodes also use flags to represent their own status between requesting entry to the CS, executing the CS, and not being involved with the CS at all. Initially, each node initializes its request set to consist of all nodes whose IDs are less than it, and sets its requesting and executing flags to be false. The requirement for a node to enter the CS is for its request set to be empty - conceptually, this means that all nodes have implicitly granted it permission to proceed. Whenever it exists the CS, it sends a REPLY message to all nodes in its inform set, to let them know that it is done. If a node receives a request of lesser priority than its own when it is making a request, it simply adds that node to its inform set, otherwise, it grants permission to that node, and immediately requests permission from it if it was not already in its request set. If it was not in the CS, and was not requesting to enter it, it directly grants permission and adds that node to its request set. Any requests received while in the middle of executing the CS are simply added to the inform set. On receiving a REPLY message from another node, it removes it from its request set.

5 Description - Lodha-Kshemkalyani's algorithm

In Lodha-Kshemkalyani's algorithm, a priority queue of outstanding requests is maintained at each node. To enter the CS, nodes must obtain permission from all other nodes by sending a REQUEST message. However, the algorithm does have two optimizations over the basic Ricart - Agrawala algorithm :

- REPLIES are cumulative(which comes into play where network delays outweigh a short CS execution time). If a process receives a REPLY from a process, it is treated as equivalent to receiving a REPLY from every process that was ahead of the sender as well, and hence all of those requests get removed from the queue.

- If 2 nodes are making concurrent requests, the request with higher priority is implicitly considered to have obtained permission from the other node when it receives the lower-priority REQUEST message, saving on an extraneous REPLY/REQUEST message sequence.

Nodes which are not involved with the CS still send REPLY messages automatically to REQUEST messages, and add them to the queue. If a node is waiting for permission, it grants permission to requests with a higher priority. Once a node has permission from all processes, it can enter the CS. Requests received while it is in the CS are deferred. Once a node exits the CS, it may have up to 2 different types of requests enqueued:

- Requests concurrent with its own request - it sends a FLUSH message to these

- Requests causally after its own request - it sends a REPLY message to these

The requests made causally after its own request can themselves be of 2 varieties:

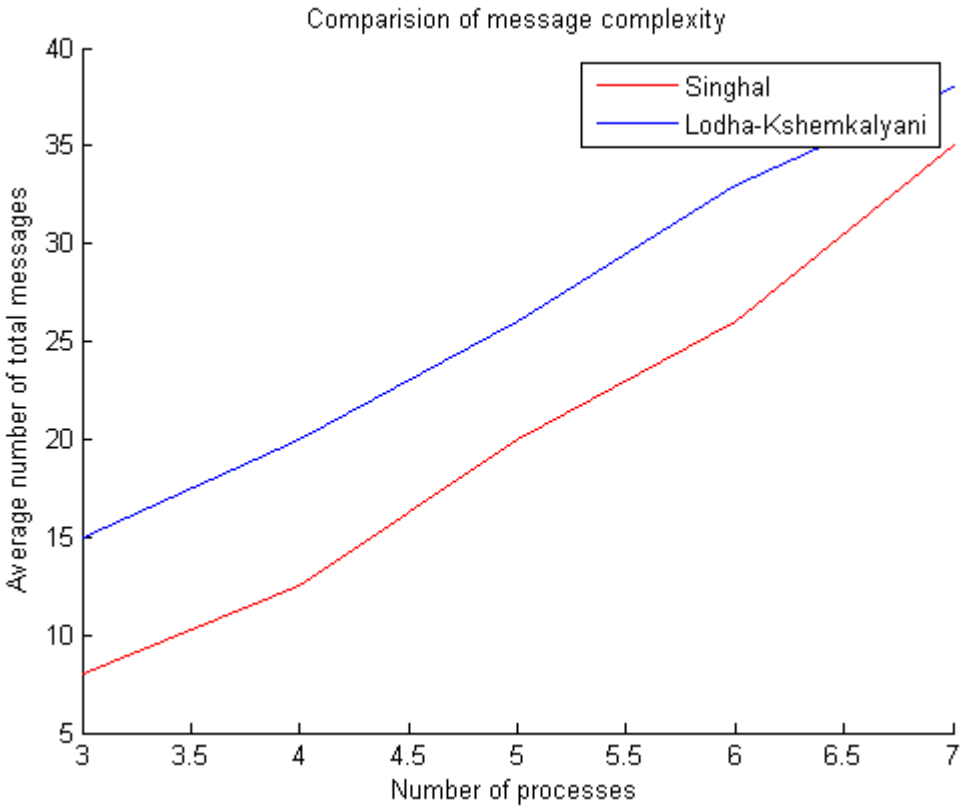
- Nodes that granted permission to this node directly, and then asked for its permission with a lower priority

- A node that granted permission to this node when it finished execution of its CS earlier, and is now asking for permission with a lower priority

6 Testing Conditions

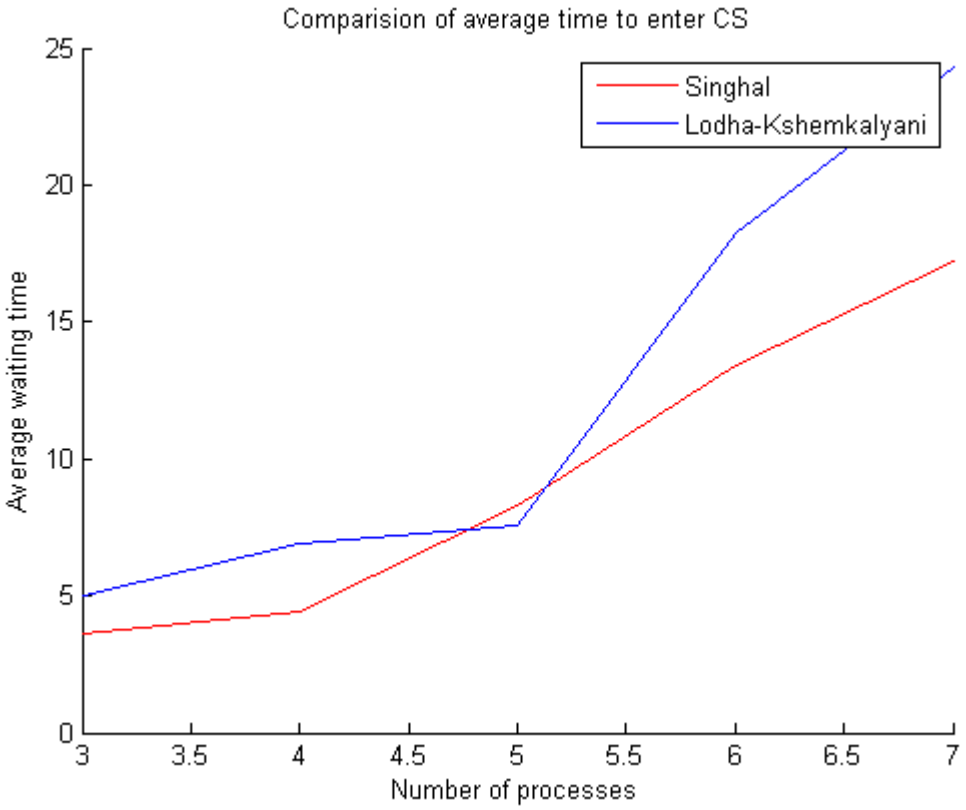
For the purpose of the project analysis, we used a star topology network. We varied the number of processes from 3 to 7, keeping the maximum number of requests per node at 5. We assumed that the work done within and outside the CS were exponentially distributed with means 1 and 3 seconds respectively.

7 Message Complexity Comparision



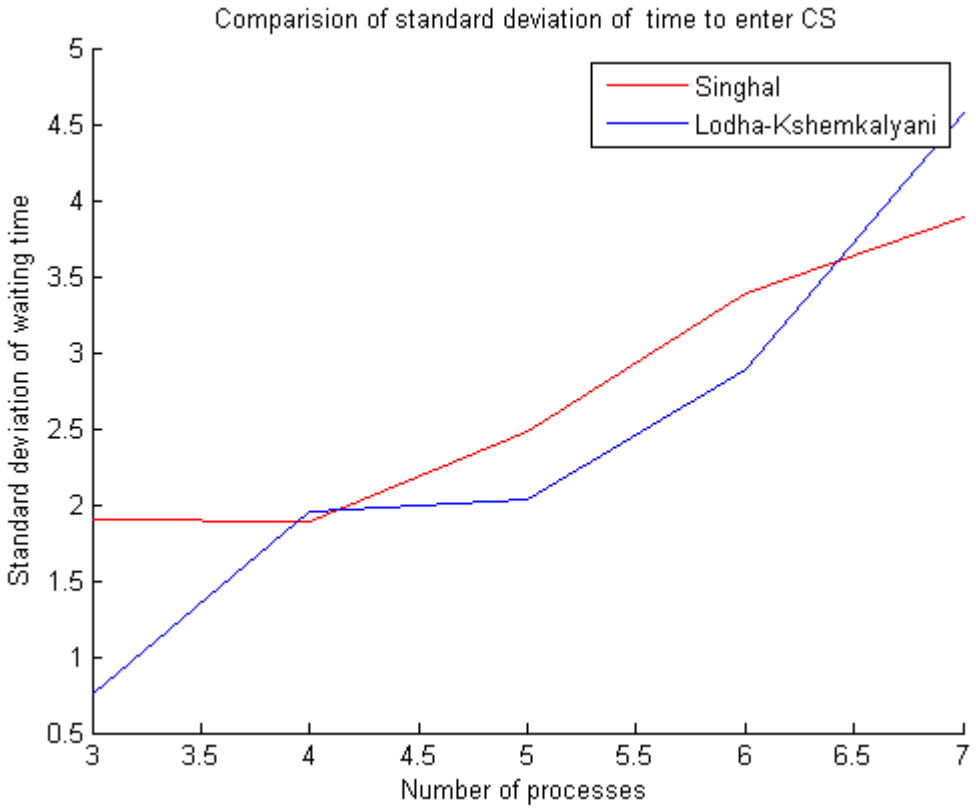
We can see that Singhal's algorithm significantly outperforms Lodha-Kshemkalyani's algorithm in terms of message complexity. This is to be expected because the explicit goal of Singhal's algorithm is to be adaptive to the CS request patterns to reduce the number of messages. On the other hand, the LK algorithm only removes one round trip from when it deals with concurrent requests, and uses cumulative replies, which are more of a time optimization than a message complexity optimization(as extraneous messages will be sent in such a case anyway)

8 Delay Comparision



We can see that Singhal's algorithm significantly outperforms Lodha-Kshemkalyani's algorithms in terms of average time to enter CS. This is related to the reduced message complexity, as processes have to, on average, wait for fewer responses to enter the CS. On the other hand, the LK algorithm still effectively has to wait to get permission from nodes completely uninvolved in the CS contention, which increases the running time more as the number of nodes increases.

9 Fairness Comparision



We can see that Lodha-Kshemkalyani's algorithm is more fair than Singhal's algorithm on average, as the standard deviation of waiting time to enter the CS tends to be lesser. This is to be expected, as the LK algorithm explicitly aims for fairness by maintaining requests in a priority queue to avoid starvation. On the other hand, if a node that normally does not contend for the CS suddenly needs to enter it, it will need to wait for quite some time to get permission from all of the active CS requesting nodes, which causes waiting times farther from the mean. However, for the test cases, we can see that the difference was still not too great, and in real distributed systems(which are often not meant to be real-time), infrequent delays are tolerable in exchange for significant increases in throughput.

10 Conclusions

We can see that Singhal's algorithm achieves higher throughput than Lodha-Kshemkalyani's algorithm, at the cost of being less fair. It would be more useful in cases where a few processes request the CS a lot, and the rest only infrequently do. Lodha-Kshemkalyani's algorithm would be a good choice when a decrease in throughput is acceptable, but latency is important.

11 Acknowledgements

We would like to thank Dr. Sathya Peri and IIT Hyderabad for giving us the opportunity to work on this interesting comparitive study project.