CS 647 HW1
Analysis

Q1) Consider some alternative notions of time, such as vector clocks. How easy would it be to switch your algorithm to use vector clocks instead of Lamport clocks?

Ans) For my implementation of the Mutual Exclusion using the Lamport clock, replacing it with the vector clock would not be difficult. The main reason for this is that we are already using a queue of some sort to store message timestamp in order to measure which node would acquire the lock. For the vector clocks, this queue would be replaced with the dictionary containing the node id as the key and timestamps as the value. In my implementation, I am already using the dictionary for performance enhancement. Instead of updating the clock of only the receiving node on every message received, in vector clocks, we would update the whole set of clock counters of nodes. To follow the flow of vector clocks, with each message received from a node x to node y, the timestamp in the dictionary of node y against the key id of node x would be updated to (max (node y current timestamp) (received timestamp value in the message)). Also, for every work done at node x, we increase its timestamp in the dictionary by some value (for simplicity, let's say by 1). Now for deciding the lock, in the dictionary, we would be finding which node id has the lowest timestamp value. If node id with the lowest timestamp matches with that of node it is running on, then it would get the lock. In my implementation, the periodic timer is used to check the dictionary to find the lock eligibility, the same could be done in the case of the vector clocks. In the case of Lamport clocks, after the release the node's requests get removed from the queue, however, in the case of vector clocks, we are just achieving the same by incrementing the counter in the dictionary for the released node id. Other than these changes, acks messages remained as is alongside the condition of node x would only acquire the lock once received acks from all other nodes. Also, the state machine for acquiring, owned, and releasing remains the same.

Q2) How many messages are sent for each acquisition, in terms of the number of processes, ignoring your shutdown extension? Is this efficient? If it's hard to tell, how might we evaluate whether it was acceptable?

Ans) Let the number of nodes(processes) is N. The number of messages sent by a node for the acquisition request is N-1 messages, the acknowledgments received at this node is N-1, after acquiring the lock, and to release the lock, N-1 messages are sent as a release. Adding these message count up, the total number of messages sent for each acquisition is 3(N-1) messages. Generally, with this count of messages, reliable message delivery plays a huge role in maintaining the performance of the system. In the case of dropped messages, the queue at some nodes would become inconsistent with others, making lock acquisition inconsistent overall. To evaluate the performance of this system, we can see this message count from two different perspectives, one system where work done at each node takes short time, other is a system where work done at each node takes a longer time. In the first system, the delay in messages would significantly bog down the whole system as the system would spend more time delivering messages rather than doing the actual work. In the second system, message

delay won't matter that much as work done at each node takes more time making the system spend more time doing work.

Q3) Lamport's algorithm in this case is (knowingly) not fault-tolerant. What would happen if one process simply stopped — whether because it crashed permanently or the network connection was severed — and never returned? You don't need to propose a fix, just predict what would happen with your implementation if one process just stopped running.

Ans) Here the results of a few scenarios in the case of failure of one node (process):

If one process simply stopped, the implementation would result in an error as the lock is gained once received acks from all the other nodes, in case if the lock has not been assigned, the system would be hung. However, if we set some limit of about receiving acks from 75% nodes, then the system would run as expected.

If the node has acquired the lock and failed before releasing the lock, other nodes would not able to acquire the lock as a release from the failed node has never been sent. With no release received, no other nodes would delete the requests from that node from their queue, resulting in no new nodes acquiring the lock.

The same situation would occur if the node expected to acquire the lock, failed before acquiring the lock, hence not sending the release.

If the node that failed request is long back in the queue, then the system would manage to do fine for a while, until reached that request. After that node is expected to acquire the lock and if in the failed state, then the overall situation would eventually be the same as mentioned above.

A similar situation would occur if the node failed just after releasing the lock and sending the new requests. For a while, the system would be fine until reached that request.

If a node failed just after releasing the lock, and before sending any new requests with the condition that the queue does not contain any more requests from this node, then the system would work as expected as failed node would never be expected to acquire the lock again.