CS 425: Distributed Systems
Homework 3

Net ID: ps71

9/8/2024

Question 1: 3…2…1… Liftoff! You're off to Saturn. During liftoff you're browsing code (what else?). Within the first minute after launch, you realize that one of the runs for the synchronous consensus (the same as that discussed in class) may have a bug. Concretely, see the synchronous consensus algorithm for a system of N > 5 machines, and imagine it being run in an asynchronous system. If both the 3rd and 4th rounds deliver NO messages at all (messages are sent, just dropped and never delivered, and the algorithm moves on to the next round), will all such runs of the algorithm still be correct? If yes, prove it. If no, show a counter- example. Quick, you're about to exit the Earth's atmosphere!

To understand whether the synchronous consensus algorithm will still work correctly in an asynchronous system when the 3rd and 4th rounds deliver no messages, let's break it down:

Synchronous Consensus Algorithm

The synchronous consensus algorithm relies on a series of rounds in which processes (or machines) exchange messages. In each round, machines send messages to each other, and based on the messages they receive, they update their states. The algorithm generally guarantees consensus as long as no more than a certain number of failures occur (for example, Byzantine or crash failures).
In a **synchronous** system, the algorithm assumes that:

- There is a known bound on message delivery time, and
- Every message sent within a round is delivered by the end of that round.

In an **asynchronous** system, there are no guarantees about message delivery time, and messages can be delayed indefinitely, making it difficult to ensure that all nodes will agree on a common value within a specific number of rounds.

The Problem

In this case, you have a system of $N > 5$ machines, and messages are dropped entirely in both the **3rd** and **4th rounds** (messages are sent, but never delivered). You need to determine if the algorithm still guarantees correctness in this asynchronous scenario.

Key Considerations

1. **Message Dropping in Asynchronous Systems:**
    - If messages are dropped entirely in certain rounds, the nodes do not receive any information for those rounds. This can cause the nodes to miss critical information that is needed to update their states and converge on a consensus value.
2. **Rounds and State Updates:**

- In synchronous consensus, each round typically involves broadcasting the current state or value of a process and receiving messages from other processes. If no messages are delivered in rounds 3 and 4, the nodes will be "stuck" with the values they had at the end of round 2.

3. **Correctness of Consensus:**
   - For consensus to be correct, all nodes must eventually agree on the same value. However, if multiple nodes have different values by the end of round 2, and they do not exchange messages in rounds 3 and 4, they will not be able to converge to the same value.
   - In particular, if the nodes do not receive any new information in rounds 3 and 4, they will continue to hold onto potentially different values, violating the agreement condition of consensus.

Counterexample

Consider the following counterexample to show that the algorithm will **not** always be correct:

- Suppose the initial values of the processes are split: some processes have value 0, and others have value 1.
- By the end of round 2, some processes might still hold value 0, while others hold value 1.
- In rounds 3 and 4, no messages are delivered, so the processes cannot exchange any new information. Therefore, the processes holding value 0 will continue to hold value 0, and the processes holding value 1 will continue to hold value 1.
- This results in a scenario where the processes do not agree on the same value, violating the correctness property of consensus.

Conclusion

The algorithm will **not** be correct if rounds 3 and 4 deliver no messages in an asynchronous system. The lack of message delivery in these rounds can prevent the processes from reaching a common value, resulting in a failure to achieve consensus.

Question 2: You have detachment from the rocket! Suddenly you see your pet cat (who you had named "Doraemon" when you adopted her) in the corridor of the spaceship—you try to follow her but she disappears. You wonder if it's your imagination. Anyway, you figure you have bigger cats to catch… Along with Pilot Rheya Cooper, you switch the communications on. You see a chart of the multicast communications between your spacecraft New Horizons X (NHX), Earth station (E), Moon station (M), and unmanned Saturn (S). If these stations use the FIFO Ordering algorithm, mark the timestamps at the point of each multicast send and each multicast receipt. Also mark multicast receipts that are buffered, along with the points at which they are delivered to the application.

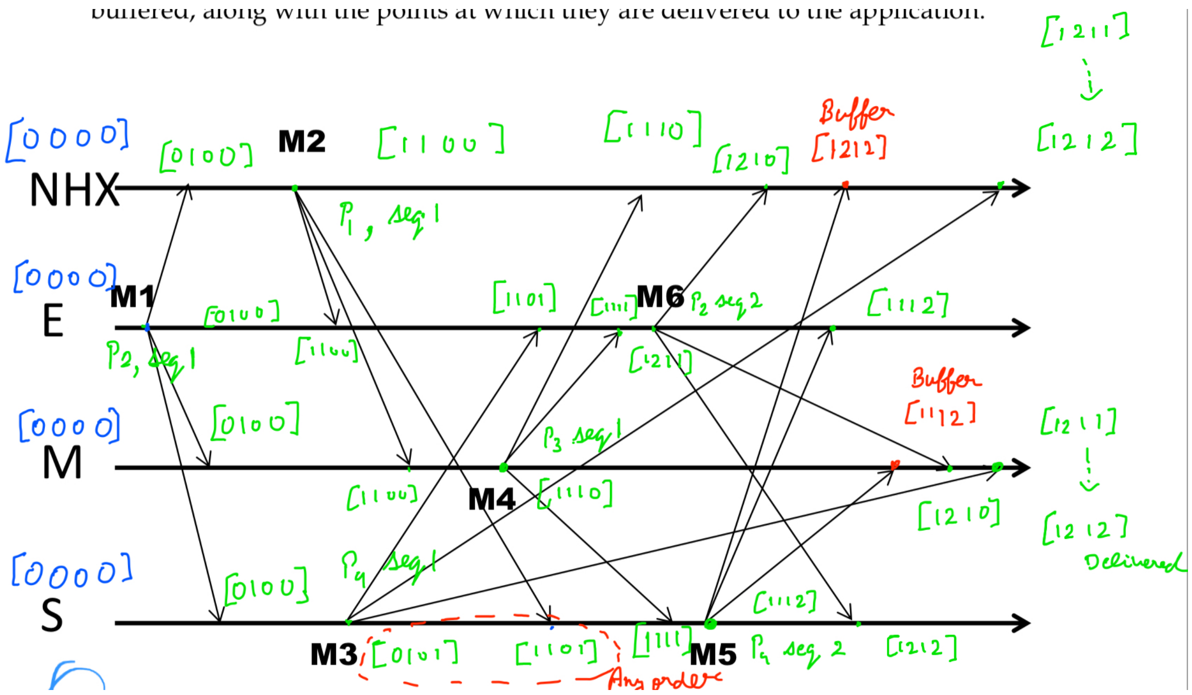


Figure 1: FIFO ordering of multicast communication

Question 3: As New Horizons X is passing through the Van Allen belts, the spacecraft's reactor and engines suddenly shut down. Oops, you realize that you should have used causal ordering in the previous timeline (Question #3). Commander Amelia Brand asks you to redo it quickly before your spacecraft crashes! Again, mention clearly all timestamps and all buffered messages.

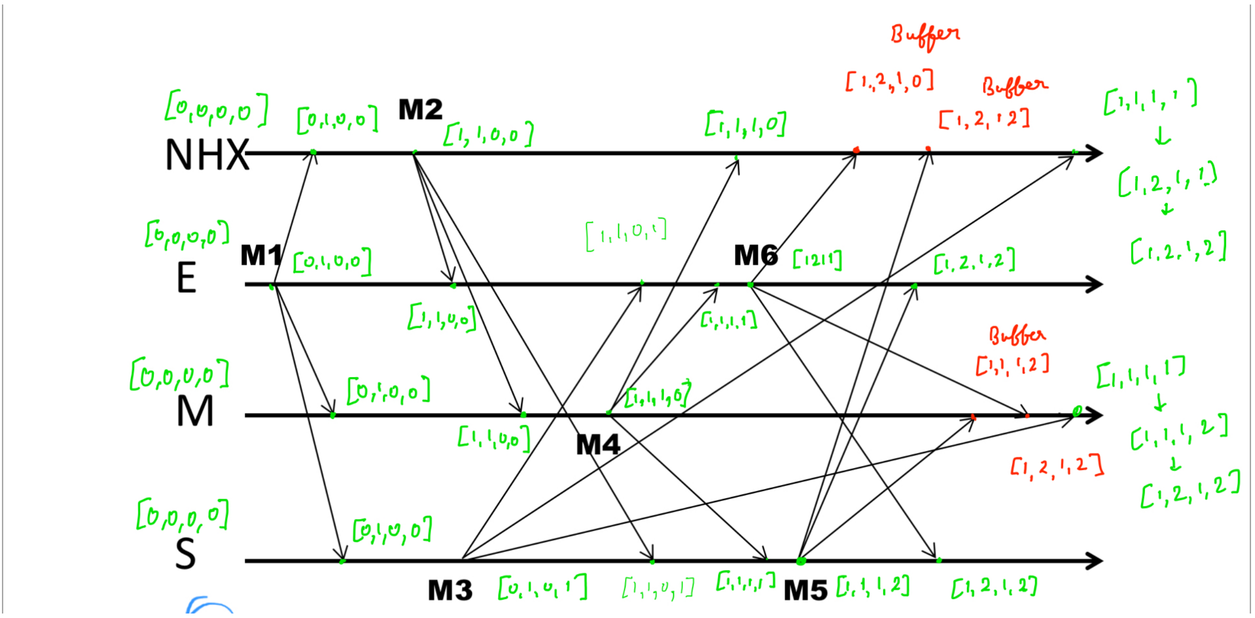


Figure 2: Casual ordering of multicast communication

Question 4: Just this morning you also saw your pet dog (whom you had named "Einstein" when you adopted him on Earth) roaming inside the spaceship's corridor. You called out to him by name and he stared at you, but then he ran away. You are perplexed, and you talk to the captain of your spaceship, Commander Amelia Brand. She says you are tired and asks you to take some rest. But there are miles to go before you sleep... To fix the consensus algorithm, Commander Amelia Brand and another of your 10 fellow astronauts, have together written a variant of the stock implementation of Paxos. You realize there is a bug. In a datacenter with N processes (N large enough), you realize that in some places in the implemented algorithm, instead of majority (for a quorum), it uses L=((5N/12)+1) processes. There are 3 variant algorithms:

Part (a): Election phase uses L instead of N/2+1, but Bill phase uses N/2+1.

Part (b): Election phase uses N/2+1, but Bill phase uses L.

Part (c): All phases use L instead of N/2+1.

For each of the above 3 cases and fail-stop model, answer the following 3 questions:

Part (d): Is this new version live? Justify.

Part (e): Is this new version safe? Justify.

Part (f): Is this new version faster or slower than using the majority? Why?

For each of the variants of the consensus algorithm you described, let's analyze their liveness, safety, and performance relative to using a majority consensus ($N/2 + 1$):

Definitions

- **Liveness**: The system continues to operate and make progress despite some failures.
- **Safety**: The system guarantees that nothing bad happens, such as contradictory decisions being made.
- **Majority (Quorum)**: More than half, $N/2 + 1$, ensures that there is always overlap between any two quorums, which is crucial for achieving consensus in distributed systems.

Common Calculation

L is defined as $\left(\frac{5N}{12}\right) + 1$. This is slightly less than half of N since $5/12 \approx 0.4167$, which is less than 0.5. Hence, $L < N/2 + 1$ when N is large.

Variant Analysis

1. Election phase uses L, Bill phase uses $N/2 + 1$

a. Liveness:

- *Possibly not live*: If more than $N - L$ processes fail (which is less than half of the total processes), the Election phase may fail to reach a quorum, halting progress even though the Bill phase could potentially still operate. Thus, the system's ability to progress is compromised under certain failure conditions.

b. Safety:

- *Potentially safe*: Safety in the Bill phase is maintained as it still requires a majority. However, decisions made during the Election phase with fewer than a majority could lead to situations where not enough correct or consistent processes are available to validate or execute decisions properly.

c. Speed:

- *Potentially faster during the Election phase*: Since L is less than a majority, fewer processes are needed to make progress, potentially speeding up this phase under low failure conditions. However, this can compromise liveness and safety as noted.

2. Election phase uses $N/2 + 1$, Bill phase uses L

a. Liveness:

- *Likely live*: Both phases can continue operation under fail-stop failures up to $N - L$, but the Bill phase might be a bit more susceptible to failures because it requires fewer processes. However, since the Election phase requires a majority, it secures the necessary condition for the Bill phase to initiate.

b. Safety:

- *Safety risk*: By reducing the requirement in the Bill phase to less than a majority, it's possible for two disjoint sets of processes (each forming an L-sized quorum) to make different decisions, leading to inconsistency.

c. Speed:

- *Mixed*: While the Bill phase might be faster due to a lower threshold, the Election phase operates at standard speed (requiring a majority). This mixed approach trades off some safety for potential gains in speed in one phase.

3. All phases use L

a. Liveness:

- *Potentially not live*: Since L is less than a majority, the algorithm might not progress if just above $N - L$ processes fail, which is a smaller fraction of failures compared to needing a majority.

b. Safety:

- *Likely unsafe*: Since both phases operate with less than a majority, there's a higher risk of inconsistent decisions due to potential non-overlapping L-sized groups.

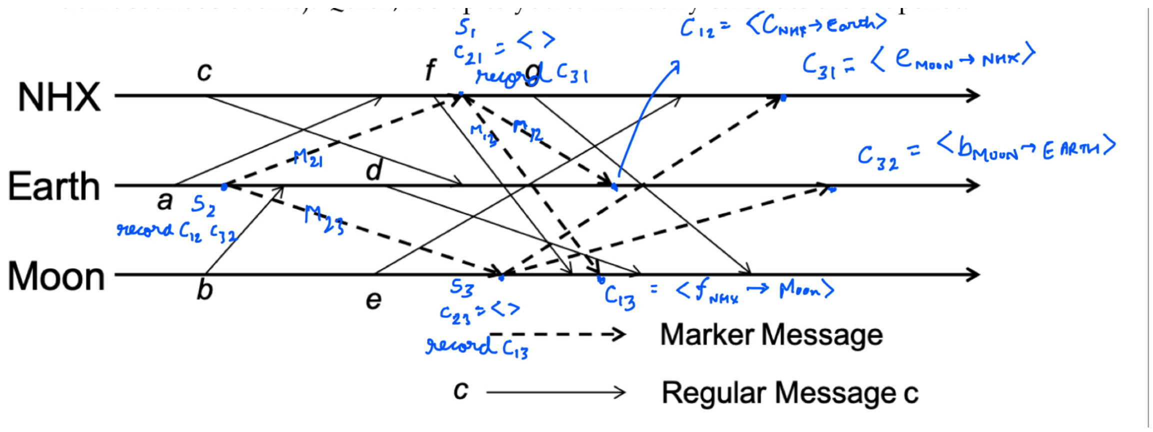
c. Speed:

- *Faster under low failure conditions*: All phases requiring fewer processes can make faster decisions under scenarios with few or no failures, but at a significant trade-off in safety and potential liveness issues under higher failure conditions.

Summary

Using L instead of a majority introduces risks in terms of safety and liveness, particularly as the number of required processes L does not guarantee overlapping quorums necessary for consistent consensus in distributed systems. The system might operate faster in low failure scenarios but at considerable risks.

Question 5: Now your spaceship is passing by the Dark Side of the Moon. It's a glorious view! To ensure things are working properly Commander Amelia Brand and Pilot Rheya Cooper ask you to run the Chandy-Lamport snapshot algorithm on the ongoing communications between your spacecraft, and the manned Earth station, and manned Moon station. But due to a crash at the different stations, the algorithm only outputs the following timeline. In the figure, a, b, c, … are regular application messages. You can use S(a) to denote the send event of a and R(a) to denote its receipt event. Markers shown as dotted lines. Can you help the intern find the snapshot recorded by this run? Don't forget to include both process states and channel states. For process states, you can use the name of the latest event at that process (For initial state, just say "Initial state". Note that Markers don't count as events). Quick, it's up to you to manually calculate the snapshot!



NHX
Earth
Moon
c
f
g
a
b
d
e
S_1
$C_{21} = \langle\rangle$
record C_{31}
$C_{12} = \langle C_{NHX \to Earth}\rangle$
$C_{31} = \langle e_{MOON \to NHX}\rangle$
M_{13}
M_{12}
M_{21}
$C_{32} = \langle b_{MOON \to EARTH}\rangle$
S_2
record C_{12} C_{32}
M_{23}
S_3
$C_{23} = \langle\rangle$
record C_{13}
$C_{13} = \langle f_{NHX} \to Moon\rangle$
Marker Message
Regular Message c

Figure 3: Chandy-Lamport Snapshot

Question 6: You're still doing well physically and emotionally in this long trip, mostly because you were trained well at Illinois. You're about halfway through the trip to Saturn. As you're retiring to your room to sleep, you see both your cat Doraemon and dog Einstein walking together in the spaceship corridor. You call out to them, but they run away again. Trippy, Dude! Before you can chase them, you notice the spacecraft wobbling quite a bit, and you need to fix this. You trace the wobbling problem to the on-board storage system, and the fact that there is no leader election algorithm in there! Quick, you need to design one! The datacenter onboard (with hundreds of machines) uses a ring-based DHT (among the machines) with a Chord-like routing algorithm with each peer maintaining 3 ring successors and 3 ring predecessors, and Chord finger tables. This system needs to elect a leader that has the second-highest DHT Id in the system.

Part (a): Design a leader election protocol that is efficient in that it uses very few messages (O(1) Chord DHT routing messages per participant). The only messages you can use are the Chord DHT routing messages (e.g., "route to ID X").

The leader election protocol should efficiently use Chord DHT routing messages. Here's how the election can be structured:

1. Initialization Phase:
   - Each node knows its own DHT ID and the IDs of its 3 successors and 3 predecessors, as well as having access to the finger table, as per the Chord algorithm.
   - The goal is to elect the node with the second-highest DHT ID.
2. Election Trigger:
   - One of the nodes initiates the election process by sending an election message. This can be any node in the system.
   - The message is routed to the node with the highest known ID, utilizing the finger table and Chord routing (O(1) routing messages to reach the next node).
3. Passing Messages:
   - Each node that receives the message forwards it to its highest successor based on the finger table, effectively routing the message through the Chord network.
   - Once the node with the highest DHT ID is reached, it records its own ID and forwards the message to its immediate predecessor, routing the message using the finger table.
4. Finding Second-Highest:
   - The node with the highest ID forwards the message to the node with the next-highest ID (its immediate predecessor in the ring).
   - This node (with the second-highest ID) records itself as the leader and sends a notification to all participants that the leader has been elected.
   - The notification propagates through the system, routed via Chord routing using O(1) messages per node.

5. Summary of the Protocol:

    - The protocol leverages the Chord ring structure and finger tables to efficiently route the election message to the node with the highest DHT ID, which in turn identifies its predecessor as the leader with the second-highest DHT ID.

Part (b): Argue briefly why your algorithm satisfies safety and liveness when finger tables are all correct and there are no failures during execution (formal proof not needed).

- **Safety:** The algorithm ensures that exactly one leader is elected because the message is routed to the node with the second-highest DHT ID, and the election terminates once this node is identified. Since each node forwards the election message based on its finger table and direct neighbors, no two nodes can mistakenly assume leadership.
- **Liveness:** As long as finger tables are correct and no failures occur, the message will eventually reach the node with the second-highest ID. This ensures that the election process completes, guaranteeing liveness. Chord's routing mechanism provides efficient message passing, and there are no loops, meaning the algorithm terminates.

Part (c): What is the completion time and number of messages in your leader election protocol (both asymptotic)?

- **Completion Time:** The completion time is dominated by the time it takes for a message to route through the Chord ring from one node to another. Since Chord routing takes logarithmic time (O(log N)), the total time to reach the node with the highest ID and then its immediate predecessor is O(log N).
- **Number of Messages**: Each node forwards the election message via Chord routing, which takes O(1) messages per hop. Since there are O(log N) hops in total (to reach the highest ID and then the second-highest), the total number of messages is also O(log N).

Part (d): Discuss briefly what might happen if failures occur during the election run, while finger tables stay inconsistent.

If failures occur during the election process, the system's finger tables may become inconsistent. This can cause several issues:

- **Message Loss**: If a node fails, the election message may be lost, and the election may not complete. The system will need a mechanism to detect failures and restart the election.
- **Routing Errors**: Inconsistent finger tables may cause the message to be routed incorrectly, potentially leading to incorrect election results (e.g., electing the wrong leader).
- **Partitioning**: In the worst case, failures could partition the network, preventing some nodes from participating in the election. This can lead to multiple leaders being elected in different partitions, violating the safety property.

Question 7: You are now going through the dreaded Asteroid belt between Saturn and Jupiter! To make things worse, the leader node elected has named itself "Hal", and this single leader seems to be acting up! Hal is threatening to take over the entire space ship and kill everyone on board! Jeez Louise, you feel like you've seen this movie somewhere! Anyway, to ensure that a single leader node cannot make such decisions, you need multiple leaders in the on-board cluster. Solve the k-leader election problem (for a given value of k). It has to satisfy the following two conditions: 1. Safety- For each non-faulty process p, p's elected = of a set of k processes with the lowest ids, OR = NULL. 2. Liveness- For all runs of election, the run terminates AND for each non-faulty process p, p's elected is not NULL. Modify the Bully Algorithm described in lecture to create a solution to the k- Leader Election problem. You may make the same assumptions as the Bully Algorithm, e.g., synchronous network. Briefly discuss why your algorithm satisfies the above Safety and Liveness, even when there are failures during the algorithm's execution.

To modify the Bully Algorithm for the k-leader election problem, the goal is to elect the k processes with the lowest IDs as leaders, while ensuring Safety and Liveness. Let's first review the Bully Algorithm and then modify it to handle the k-leader election problem.

Original Bully Algorithm (Overview):

- Every process has a unique ID.
- When a process detects that the current leader has failed, it initiates an election by sending an election message to all processes with higher IDs.
- The process with the highest ID takes the role of the leader and notifies others.

Modifications for k-Leader Election: Assumptions:

- The system is synchronous (like in the Bully Algorithm).
- The communication happens via reliable channels.
- Processes know the total number of processes and their unique IDs.

Modified k-Leader Bully Algorithm:

1. Election Trigger:
   - Any process can initiate an election when it detects a failure of one of the leaders or if no leader exists.
   - The process p that detects a failure starts the election by sending an election message to all processes with higher IDs.
2. Response Handling:
   - Upon receiving an election message, processes with higher IDs respond, indicating that they are available for election.
   - These processes then propagate the election message to all processes with even higher IDs than themselves.
3. Collecting Responses:

- Once a process initiates an election, it collects responses from processes with higher IDs.
- Each process that receives responses keeps track of the k processes with the lowest IDs among the responders.

4. Selecting k Leaders:
   - After gathering responses, the initiator process sorts the responding IDs and picks the k smallest IDs as the leaders.
   - The initiator sends a "leader notification" to all processes in the system, informing them about the elected k leaders.
5. Leader Confirmation:
   - When a process receives a "leader notification" message, it updates its local view of the leaders to the list of k processes with the lowest IDs.
   - Each process stores this list and acknowledges the new leaders.
6. Fault Tolerance:
   - If some processes fail during the election (including leaders), the election restarts, and new processes with higher IDs can initiate the election.
   - As long as at least one non-faulty process is alive, the election will terminate with k leaders being selected.

Why the Algorithm Satisfies Safety and Liveness:

1. **Safety:**
   - The algorithm guarantees that each non-faulty process eventually agrees on a set of k leaders with the smallest IDs.
   - Processes keep track of responses and only accept the k lowest IDs, ensuring that there is no disagreement on who the leaders are.
   - If a process doesn't receive enough responses (due to failures), it initiates a new election or returns NULL until k processes are selected.
2. **Liveness:**
   - The election process is guaranteed to terminate because the system is synchronous, and messages will eventually be delivered within a known time.
   - For every non-faulty process p, the elected leaders will not be NULL because the process that initiates an election ensures that k processes with the lowest IDs are selected as leaders.
   - Even if failures occur, the election is retried until successful.

Handling Failures:

1. If one or more leaders fail, other non-faulty processes will detect the failure and initiate a new election. This ensures that there is always a set of k leaders.
2. Since the system is synchronous, processes will either receive a response or assume a failure if no response is received within a timeout. This prevents the algorithm from getting stuck.

Algorithm Example:

1. Suppose we have 5 processes with IDs: 1, 2, 3, 4, 5.
2. Let's assume p1 and p2 fail, and p3 detects this and initiates an election.
3. p3 sends an election message to p4 and p5.
4. p4 and p5 respond, and p3 gathers responses.

5. Since p3 started the election and is a candidate itself, it selects the k smallest IDs from the responders (including itself). Let's say k=3.
6. The leaders are then p3, p4, and p5. This is communicated to all processes, and all non-faulty processes agree on this leader set.

Thus, the modified Bully Algorithm ensures the election of multiple leaders while maintaining both Safety and Liveness under failure conditions.

Question 8: Bam! Your New Horizons X spacecraft has just suffered a massive strike from an asteroid! Alarms are going off all around you. And a dog can be heard yelping and a cat can be heard welping in agony, somewhere inside the spacecraft. You talk to Commander Brand and your colleague on board Pilot Rheya Cooper about this, and they counsel you that the animals are your imagination. Anyway, back to work… Commander Amelia Brand, Pilot Rheya Cooper, and you quickly figure out that the alarms are because of the mutual exclusion algorithm implemented in the system – if you can fix it, the spacecraft will return to normal operations. You see that the datacenter uses the Ricart-Agrawala algorithm for mutual exclusion but instead of using the usual and boring (Lamport timestamp, process id) pair, the algorithm instead uses (Lamport timestamp, FIFO local sequence number) pair, where FIFO local sequence number is the local sequence number of that event at that process. The rest of the Ricart-Agrawala algorithm remains unchanged. Your fellow astronaut says this algorithm, even without failures: a) violates safety, b) violates liveness, and c) does not satisfy causal ordering. Is he right on any of these counts (which ones)? Give a proof or counter-example.

1. **Safety (Mutual Exclusion)** The Ricart-Agrawala algorithm ensures mutual exclusion by using timestamps and process IDs to guarantee that no two processes can be in their critical section simultaneously.

   In this modified version, the algorithm replaces the process ID with the FIFO local sequence number, but the key here is the Lamport timestamp. The Lamport timestamp ensures a consistent ordering of events across all processes. If two requests have the same timestamp, the FIFO local sequence number will break the tie within the same process.

   Since the FIFO local sequence number is unique within each process, it doesn't interfere with the global ordering provided by the Lamport timestamp. Therefore, the modified algorithm still ensures that requests are granted in a globally consistent order based on the Lamport timestamp, preventing more than one process from entering the critical section at the same time.

   Conclusion: The algorithm does not violate safety. Mutual exclusion is preserved.

2. **Liveness** Liveness in the Ricart-Agrawala algorithm ensures that every process that requests access to the critical section will eventually get it, assuming there are no failures or indefinite delays in communication.

   The change to use the FIFO local sequence number instead of the process ID does not introduce any new waiting or deadlock conditions. The Lamport timestamp continues to ensure that requests are processed in a consistent, global order. Each process still sends and receives requests and replies in the same way as in the original algorithm.

   Thus, as long as the original Ricart-Agrawala algorithm guarantees liveness, this modified version also does.

   Conclusion: The algorithm does not violate liveness.

3. **Causal Ordering** Causal ordering refers to ensuring that events that are causally related (i.e., one event must happen before another) are processed in that order. In the Ricart-Agrawala algorithm, causal ordering isn't strictly enforced—it is designed to ensure mutual exclusion, not causality. However, Lamport timestamps generally respect causality, since events with lower timestamps are considered to have occurred earlier.

   The modified algorithm still uses Lamport timestamps, so causal ordering is preserved based on timestamps. The FIFO local sequence number is a purely local mechanism to order events within a process, so it doesn't interfere with the global ordering established by the Lamport timestamps.

Conclusion: The algorithm does not violate causal ordering.

Your fellow astronaut is incorrect on all counts:

- Safety is preserved (mutual exclusion holds).
- Liveness is preserved (no process is indefinitely delayed).
- Causal ordering is respected (Lamport timestamps ensure this).

The modifications do not introduce any violations of these properties.

References