Homework 3 Solutions: CS425 FA25

Note to Students: Recommended solutions for HW3 (FA25) are below. For many questions, alternate solutions are possible and reasonable correct solutions will be accepted during grading. So please refrain from asking questions about solutions until you receive your HW grades back.

1. (Solution and Grading by: Ashish Kashinath.)

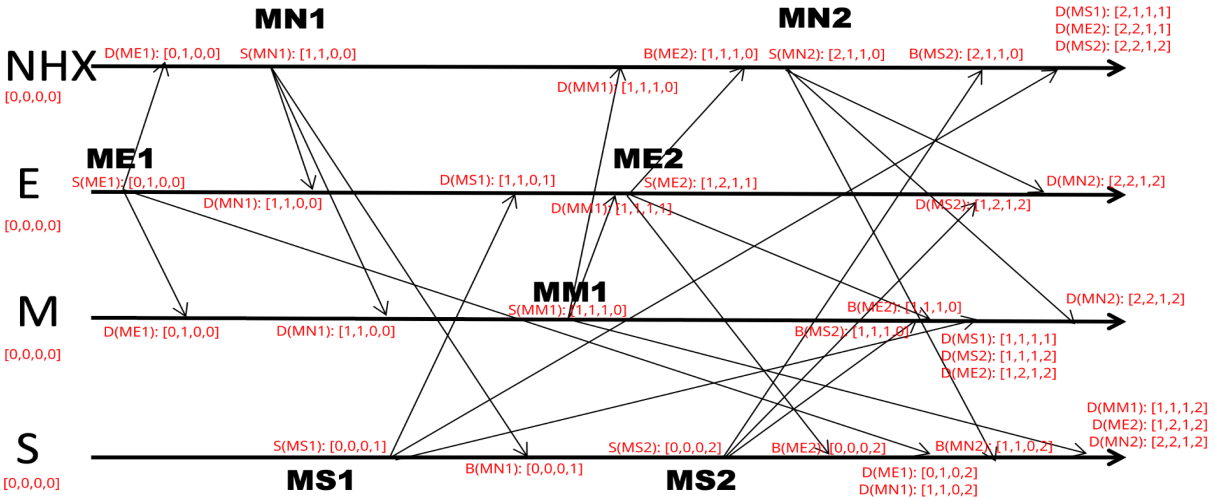
No, the synchronous consensus algorithm would not be correct in this case. We show this using a counterexample with N=6 processes and f=4 failures. The counterexample, which shows the synchronous consensus algorithm fails,follows ideas from the contradiction-style proof of correctness of synchronous consensus from Coursera week 7 Lecture 3.2/Lecture 15.

Counter-example: Suppose the processes are numbered P1 through P6. Since f=4, the synchronous consensus algorithm would terminate in (f+1)=5 rounds wherein in both rounds 3 and 4, one multicast message is dropped. We show failure of the algorithm by showing that at the end of the algorithm, two non-faulty processes P3 and P6 can have different values.

Round	P1={1}	P2={2}	P3={3}	P4={4}	P5={5}	P6={6}
1	Send to P2 and crashes ❌	{1,2,3,4,5,6}	all except P1 → {2,3,4,5,6}	all except P1 → {2,3,4,5,6}	all except P1 → {2,3,4,5,6}	all except P1 → {2,3,4,5,6}
2		Sends to P3 and crashes ❌	{1,2,3,4,5,6}	all except P2 → {2,3,4,5,6}	all except P2 → {2,3,4,5,6}	all except P2 → {2,3,4,5,6}
3			A Multicast is dropped {1,2,3,4,5,6} 🔇	All except P3 → {2,3,4,5,6}	All except P3 → {2,3,4,5,6}	All except P3 → {2,3,4,5,6}
4			An Empty Multicast is dropped/No Change {1,2,3,4,5,6} 🔇	All except P3/No Change → {2,3,4,5,6}	All except P3/No Change → {2,3,4,5,6}	All except P3/No Change → {2,3,4,5,6}
5			No Change → {1,2,3,4,5,6}	Crashes at the beginning ❌	Crashes at the beginning❌	No Change → {2,3,4,5,6}

Note that P3 does not send any multicast in rounds 4 and 5, as its values did not change since round 3. We have also accepted solutions that have considered multicasts from P3 in round 4. At the end of the 5 rounds, P3 and P6 are non-faulty nodes,one of which has the value 1 and one of them does not have the value 1, violating the agreement requirement of synchronous consensus correctness, when computed, say via min(.) function.

2. (Solution and Grading by: Nishant Sheikh)



At NHX:

1. R(ME1): [0,1,0,0]
    a. D(ME1): [0,1,0,0]
2. S(MN1): [1,1,0,0]
    a. D(MN1): [1,1,0,0]
3. R(MM1): [1,1,1,0]
    a. D(MM1): [1,1,1,0]
4. R(ME2): [1,1,1,0]
    a. B(ME2): [1,1,1,0]; buffer = [ME2]
5. S(MN2): [2,1,1,0]
    a. D(MN2): [2,1,1,0]; buffer = [ME2]
6. R(MS2): [2,1,1,0]
    a. B(MS2): [2,1,1,0]; buffer = [ME2, MS2]
7. R(MS1): [2,2,1,2]
    a. D(MS1): [2,1,1,1]; buffer = [ME2, MS2]
    b. D(ME2): [2,2,1,1]; buffer = [MS2]
    c. D(MS2): [2,2,1,2]; buffer = []

(Note: after D(MS1), we can deliver ME2 and MS2 in either order, since causal ordering conditions are satisfied for both. See L16 slide 35)

At E:

1. S(ME1): [0,1,0,0]
    a. D(ME1): [0,1,0,0]
2. R(MN1): [1,1,0,0]
    a. D(MN1): [1,1,0,0]
3. R(MS1): [1,1,0,1]

    a. D(MS1): [1,1,0,1]
4. R(MM1): [1,1,1,1]
    a. D(MM1): [1,1,1,1]
5. S(ME2): [1,2,1,1]
    a. D(ME2): [1,2,1,1]
6. R(MS2): [1,2,1,2]
    a. D(MS2): [1,2,1,2]
7. R(MN2): [2,2,1,2]
    a. D(MN2): [2,2,1,2]

At M:

1. R(ME1): [0,1,0,0]
    a. D(ME1): [0,1,0,0]
2. R(MN1): [1,1,0,0]
    a. D(MN1): [1,1,0,0]
3. S(MM1): [1,1,1,0]
    a. D(MM1): [1,1,1,0]
4. R(MS2): [1,1,1,0]
    a. B(MS2): [1,1,1,0]; buffer = [MS2]
5. R(ME2): [1,1,1,0]
    a. B(ME2): [1,1,1,0]; buffer = [MS2, ME2]
6. R(MS1): [1,2,1,2]
    a. D(MS1): [1,1,1,1]; buffer = [MS2, ME2]
    b. D(MS2): [1,1,1,2]; buffer = [ME2]
    c. D(ME2): [1,2,1,2]; buffer = []
7. R(MN2): [2,2,1,2]
    a. D(MN2): [2,2,1,2]

(Note: after D(MS1), we can deliver ME2 and MS2 in either order, since causal ordering conditions are satisfied for both. See L16 slide 35)

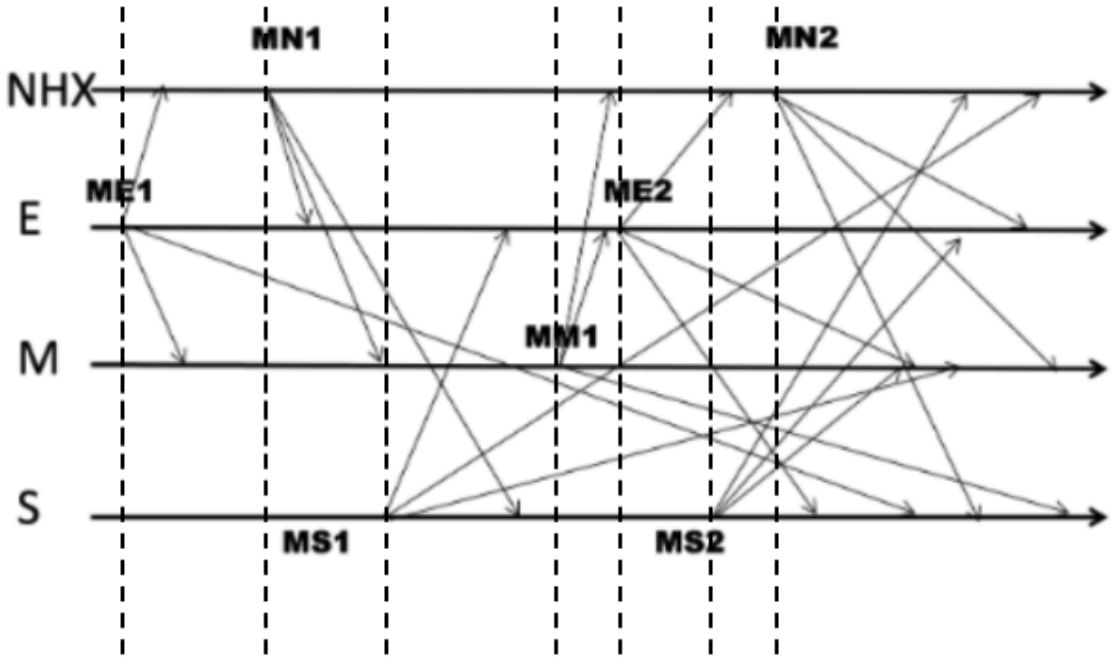
At S:

1. S(MS1): [0,0,0,1]
    a. D(MS1): [0,0,0,1]
2. R(MN1): [0,0,0,1]
    a. B(MN1): [0,0,0,1]; buffer = [MN1]
3. S(MS2): [0,0,0,2]
    a. D(MS2): [0,0,0,2]; buffer = [MN1]
4. R(ME2): [0,0,0,2]
    a. B(ME2): [0,0,0,2]; buffer = [MN1, ME2]
5. R(ME1): [1,1,0,2]
    a. D(ME1): [0,1,0,2]; buffer = [MN1, ME2]
    b. D(MN1): [1,1,0,2]; buffer = [ME2]
6. R(MN2): [1,1,0,2]

    a. B(MN2): [1,1,0,2]; buffer = [ME2, MN2]
7. R(MM1): [2,2,1,2]
    a. D(MM1): [1,1,1,2]; buffer = [ME2, MN2]
    b. D(ME2): [1,2,1,2]; buffer = [MN2]
    c. D(MN2): [2,2,1,2]; buffer = []

(Note: after D(MM1), we can deliver ME2 and MN2 in either order, since causal ordering conditions are satisfied for both. See L16 slide 35)

3. (Solution and Grading by: Madhav.)



The assumption (as per the question) that we are working with is that the latency between the processes and the sequencer is 0. What this inherently implies is that the order of delivery at each process is the order at which messages were received at the sequencer (since the sequencer will stamp increasing sequence numbers in the order in which messages are received). And given these assumptions, the order of arrival at the sequencer is determined by the relative physical times of the multicast sends.

Order of receipt at the sequencer (with respective sequence numbers) is as follows:
ME1 (0) -> MN1 (1) -> MS1 (2) -> MM1 (3) -> ME2 (4) -> MS2 (5) -> MN2 (6)
This is also the total order of delivery of messages at each process.

Also note that since the latency between the sequencer and any process is 0, the instant at which the multicast is issued is also the instant at which *each* process receives the message from the sequencer <S(M), M> (indicated by the vertical dotted lines), which can then potentially be buffered at each process as well.

Notation: R(M) = Receive M, S(M) = Send M, D(M) = Deliver M
It helps to think of the buffer as a min-heap sorted on the sequence number.
At NHX:

a. On R(S<ME1, 0>) -> buffer.Add(S<ME1, 0>), buffer = <(S(ME1), 0)>
b. On R(ME1) -> D(ME1) (deliver immediately and discard corresponding sequencer message).

c. On R(S<MN1, 1>) -> buffer.Add(S<MN1, 1>), buffer = <(S(MN1), 1)>
d. On R(MN1) -> D(MN1) (while discarding the corresponding sequencer message)
    i. Note that the order in which c and d happen can be interchanged as well - this is okay.
e. On R(S<MS1, 2>) -> buffer.Add(S<MS1, 2>), buffer = <(S(MS1), 2)>
f. On R(S<MM1, 3>) -> buffer.Add(S<MM1, 3>), buffer = <(S(MS1), 2), (S(MM1), 3)>
g. On R(MM1) -> buffer.Add(MM1), buffer = <(S(MS1), 2), (MM1, S(MM1), 3)> // combine MM1 along with its sequence number message.
h. On R(S<ME2, 4>) -> buffer.Add(S<ME2, 4>), buffer = <(S(MS1), 2), (MM1, S(MM1), 3), (S(ME2), 4)>
i. On R(S<MS2, 5>) -> buffer.Add(S<MS2, 5>), buffer = <(S(MS1), 2), (MM1, S(MM1), 3), (S(ME2), 4), (S(MS2), 5)>
j. On R(ME2) -> buffer.Add(ME2), buffer = <(S(MS1), 2), (MM1, S(MM1), 3), (ME2, S(ME2), 4), (S(MS2), 5)>
k. ON R(S<MN2, 6>) -> buffer.Add(S<MN2, 6>), buffer = <(S(MS1), 2), (MM1, S(MM1), 3), (ME2, S(ME2), 4), (S(MS2), 5), (S(MN2), 6)>
l. On R(MN2) -> buffer.Add(MN2), buffer = <(S(MS1), 2), (MM1, S(MM1), 3), (ME2, S(ME2), 4), (S(MS2), 5), (MN2, S(MN2), 6)>
m. On R(MS2) -> buffer.Add(MS2), buffer = <(S(MS1), 2), (MM1, S(MM1), 3), (ME2, S(ME2), 4), (MS2, S(MS2), 5), (MN2, S(MN2), 6)>
n. On R(MS1) -> D(MS1), D(MM1), D(ME2), D(MS2), D(MN2), buffer = <>

At E:

a. ME1 sequencer message and self delivery can happen immediately.
b. On R(S<MN1, 1>) -> buffer.Add(S<MN1, 1>), buffer = <(S(MN1), 1)>
c. On R(MN1), D(MN1), buffer = <>
d. On R(S<MS1, 2>) -> buffer.Add(S<MS1, 2>), buffer = <(S(MS1), 2)>
e. On R(MS1), D(MS1), buffer = <>
f. On R(S<MM1, 3>) -> buffer.Add(S<MM1, 3>), buffer = <(S(MM1), 3)>
g. On R(MM1), D(MM1), buffer = <>
h. On R(S<ME2, 4>) -> buffer.Add(S<ME2, 4>), buffer = <(S(ME2), 4)>
i. On R(ME2), D(ME2), buffer = <>
j. On R(S<MS2, 5>) -> buffer.Add(S<MS2, 5>), buffer = <(S(MS2), 5)>
k. On R(MS2), D(MS2), buffer = <>
l. On R(S<MN2, 6>) -> buffer.Add(S<MN2, 6>), buffer = <(S(MN2), 6)>

m. On R(MN2), D(MN2), buffer = <>

At M:

a. On R(S<ME1, 0>) -> buffer.Add(S<ME1, 0>), buffer = <(S(ME1), 0)>
b. On R(ME1) -> D(ME1), buffer = <>
c. On R(S<MN1, 1>) -> buffer.Add(S<MN1, 1>), buffer = <(S(MN1), 1)>
d. On R(MN1) -> D(MN1), buffer = <>
e. On R(S<MS1, 2>) -> buffer.Add(S<MS1, 2>), buffer = <(S(MS1), 2)>
f. On R(S<MM1, 3>) -> buffer.Add(S<MM1, 3>), buffer = <(S(MS1), 2), (S(MM1), 3)>
g. On R(MM1) -> buffer.Add(MM1), buffer = <(S(MS1), 2), (MM1, S(MM1), 3)>
h. On R(S<ME2, 4>) -> buffer.Add(S<ME2, 4>), buffer = <(S(MS1), 2), (MM1, S(MM1), 3), (S(ME2), 4)>
i. On R(S<MS2, 5>) -> buffer.Add(S<MS2, 5>), buffer = <(S(MS1), 2), (MM1, S(MM1), 3), (S(ME2), 4), (S(MS2), 5)>
j. On R(S<MN2, 6>) -> buffer.Add(S<MN2, 6>), buffer = <(S(MS1), 2), (MM1, S(MM1), 3), (S(ME2), 4), (S(MS2), 5), (S(MN2), 6)>
k. On R(MS2) -> buffer.Add(MS2), buffer = <(S(MS1), 2), (MM1, S(MM1), 3), (S(ME2), 4), (MS2, S(MS2), 5), (S(MN2), 6)>
l. On R(ME2) -> buffer.Add(ME2), buffer = <(S(MS1), 2), (MM1, S(MM1), 3), (ME2, S(ME2), 4), (MS2, S(MS2), 5), (S(MN2), 6)>
m. On R(MS1) -> D(MS1), D(MM1), D(ME2), D(MS2), buffer = <(S(MN2), 6)>
n. On R(MN2) -> D(MN2), buffer = <>

At S:

a. On R(S<ME1, 0>) -> buffer.Add(S<ME1, 0>), buffer = <(S(ME1), 0)>
b. On R(S<MN1, 1>) -> buffer.Add(S<MN1, 1>), buffer = <(S(ME1), 0), (S(MN1), 1)>
c. On R(S<MS1, 2>) -> buffer.Add(S<MS1, 2>), buffer = <(S(ME1), 0), (S(MN1), 1), (S(MS1), 2)>
d. On R(MS1) -> buffer.Add(MS1), buffer = <(S(ME1), 0), (S(MN1), 1), (MS1, S(MS1), 2)>
e. On R(MN1) -> buffer.Add(MN1), buffer = <(S(ME1), 0), (MN1, S(MN1), 1), (MS1, S(MS1), 2)>
f. On R(S<MM1, 3>) -> buffer.Add(S<MM1, 3>), buffer = <(S(ME1), 0), (MN1, S(MN1), 1), (MS1, S(MS1), 2), (S(MM1), 3)>
g. On R(S<ME2, 4>) -> buffer.Add(S<ME2, 4>), buffer = <(S(ME1), 0), (MN1, S(MN1), 1), (MS1, S(MS1), 2), (S(MM1), 3), (S(ME2), 4)>

h. On R(S<MS2, 5>) -> buffer.Add(S<MS2, 5>), buffer = <(S(ME1), 0), (MN1, S(MN1), 1), (MS1, S(MS1), 2), (S(MM1), 3), (S(ME2), 4), (S(MS2), 5)>
i. On R(MS2) -> buffer.Add(MS2), buffer = <(S(ME1), 0), (MN1, S(MN1), 1), (MS1, S(MS1), 2), (S(MM1), 3), (S(ME2), 4), (MS2, S(MS2), 5)>
j. On R(S<MN2, 6>) -> buffer.Add(MN6), buffer = <(S(ME1), 0), (MN1, S(MN1), 1), (MS1, S(MS1), 2), (S(MM1), 3), (S(ME2), 4), (MS2, S(MS2), 5), (S(MN2), 6)>
k. On R(ME2) -> buffer.Add(ME2), buffer = <(S(ME1), 0), (MN1, S(MN1), 1), (MS1, S(MS1), 2), (S(MM1), 3), (ME2, S(ME2), 4), (MS2, S(MS2), 5), (S(MN2), 6)>
l. On R(ME1) -> D(ME1), D(MN1), D(MS1), buffer = <(S(MM1), 3), (ME2, S(ME2), 4), (MS2, S(MS2), 5), (S(MN2), 6)>
m. On R(MN2) -> buffer.Add(MN2), buffer = <(S(MM1), 3), (ME2, S(ME2), 4), (MS2, S(MS2), 5), (MN2, S(MN2), 6)>
n. On R(MM1) -> D(MM1), D(ME2), D(MS2), D(MN2), buffer = <>

4. (Solution and Grading by: Talha.)

   [In this solution, M means majority, i.e. N/2 + 1]

   i. Election = L, Bill = L
      a. Liveness: Since the algorithm is not safe (as shown below in b), FLP's impossibility theorem is irrelevant here. Paxos is not live in general because election preemption can go on indefinitely: if each proposer's messages get delayed long enough for another proposer to start a new election, no proposal ever completes. That reasoning still applies here: the system can stall forever due to continuous elections. So liveness ***cannot*** be guaranteed.
      b. Safety: Unsafe for large N. Safety in Paxos depends on quorum intersection:
         - bill–bill intersection: Any two bill quorums must overlap so that two conflicting values cannot both be chosen. This requires $2L > N$. But with $L = 4N/9 + 1$, we get $2L \approx 8N/9 + 2 \leq N$ for $N \geq 18$. Hence, two disjoint bill quorums can form, leading to conflicting accepted values.
         - election–bill intersection: Every election quorum must overlap with any previous bill quorum to ensure a new leader learns the already accepted value. This requires $L + L > N$ which also fails for large N.
      c. Performance: Faster than standard Paxos, since both bill and election phases use a smaller quorum. This reduces the number of acks that you have to wait for in each phase.

   ii. Election = L, Bill = M
      a. Liveness: Liveness cannot be guaranteed. The reason is the same as in case i.
      b. Safety: Unsafe for large N. Safety requires that election and bill quorums intersect, i.e. $L + M > N$. For large N, $L + M \approx 17N/18 + 2$, which is $\leq N$ for $N \geq 36$. Thus, a leader could gather an election quorum disjoint from an earlier bill quorum, allowing it to propose a conflicting value.
      c. Performance: Faster than the standard Paxos, because in the election phase it has to wait for fewer acks as $L < M$.

   iii. Election = M, Bill = L
      a. Liveness: Liveness cannot be guaranteed. The reason is the same as in case i.
      b. Safety: Unsafe for large N. Safety requires that the following quorums should intersect. However,
         - bill–bill: requires $2L > N$, but fails when $N \geq 18$.
         - election–bill: requires $M + L > N$. For large N, this fails when $N \geq 36$.
      c. Performance: Faster than the standard Paxos, because in the bill phase it has to wait for fewer acks as $L < M$.

   iv. Election = L, Bill = L, Law = L

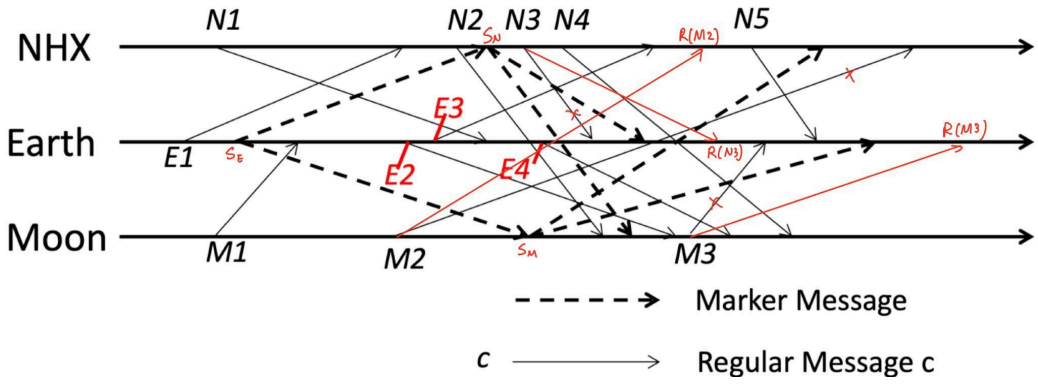
      There is no ack in the Law phase; there is no quorum size. Therefore, this case is the same as case i.

5. (Solution and Grading by: Hanbo Guo.)
    a. Causality violations: R(N3), R(M3); FIFO violations: Send(M2).
    b. Why R(N3), R(M3), M2?
    Causality violation in this question happens when the Send(X) (send event of X) is not recorded in the snapshot, but the R(X) (receive event of X) is captured in the channel state.
        - Channel NHX->Earth on Earth: captures $R(N3)$, but $Send(N3)$ is not captured in NHX's local state;
        - Channel Moon->Earth on Earth: captures $R(M3)$, but $Send(M3)$ is not captured in Moon's local state;

    FIFO violation, which means messages in a channel should be sent and received in the same order.
        - Channel Moon->NHX: M2 is sent before Moon->NHX marker $S_{M->N}$, but received after $S_{M->N}$ at NHX ($Send(M2) -> Send(S_{M->N})$, but $R(S_{M->N}) -> R(M2)$);

    What is the root cause?
    Send and receive events are not FIFO ordered. The system model assumes one channel from one process to another, and channels are FIFO-ordered.
    (To receive full marks, students need to identify FIFO being violated and show understanding of what is causality violation.)

    c. One example fix is as below:



6. (Solution and Grading by: Gabriella Xue.)

   Assumptions:
   - Nodes have unique IDs.
   - Each node knows its successor.
   - Assume messages are eventually delivered.
   - The algorithm fails if the total number of nodes < k

   Protocol:

1. Election initiation
   When a process detects a failure among any current k leaders, it:
   - Picks a new epoch e = (p.id, localCounter++).
   - Sets maxEpochSeen = e.
   - Sends ELECTION[e, IDs = {p.id}] to its successor.
2. On receiving ELECTION[e, IDs]
   At node q:
   - If e < maxEpochSeen: discard the message (it's from an older election).
   - If e > maxEpochSeen:
     - Set maxEpochSeen = e (adopt the newer election)
     - Continue with the new election.
   - Insert q's ID into IDs, then keep only the lowest k IDs.
   - If q is the initiator of epoch e and the message has completed a full circle, go to step 3.
   - Otherwise, forward ELECTION[e, IDs] to q's successor.
3. Election completion at initiator
   When the initiator of epoch e receives back ELECTION[e, IDs]:
   - It knows the token has traversed all participating non-faulty nodes.
   - IDs now holds the k elected nodes (lowest k IDs among non-faulty nodes).
   - It sends COORDINATOR[e, IDs] once around the ring.
4. On receiving COORDINATOR[e, IDs]
   At node q:
   - If e < maxEpochSeen: ignore (old election).
   - If e >= maxEpochSeen:
     - Set maxEpochSeen = e.
     - Set elected = IDs.
     - Forward COORDINATOR[e, IDs] to the successor, stopping when it returns to the initiator.
5. Handling failures / timeouts
   If a process does not receive a COORDINATOR for the current maxEpochSeen within its timeout, it suspects some node on the ring has failed (possibly the token holder) and:
   - Initiates a new election with a strictly larger epoch e' > maxEpochSeen.
   - Because everyone always adopts the highest epoch seen, older elections are eventually ignored.

Safety:
Let us assume messages are eventually delivered. We also assume that after some point in time, there are no new failures during an ongoing election, so that the ring of non-faulty nodes remains connected long enough for the token to circulate once. Safety requires that all non-faulty processes decide on the same set of k leaders and that these leaders are the k lowest ID non-faulty nodes. Safety is guaranteed because the election token circulates around the ring and carries the current top-k candidate list. The protocol eventually terminates and selects k leaders. Safety holds as long as there are at least k non-faulty nodes, as failures during the protocol re-initiate the election. Even if the token holder or the initiator fails, some other node (typically the next live node that detects the stall) will re-initiate a new election.

Liveness:
We use the same assumptions for Liveness as we do in Safety. Liveness requires eventual leader election. In our protocol, if a node fails during and the election message is blocked, other nodes detect the failure through timeouts when they do not receive the COORDINATOR message. This suspicion triggers the re-initiation of the election, eventually excluding the failed node from participation. Therefore, as long as at least k non-faulty nodes remain and the ring remains connected, the leader can be selected.

Fault tolerance: Safety and Liveness hold as long as k non-faulty nodes remain; therefore, it tolerates up to f = N-k failures.

7. (Solution and Grading by: Chirag.)

   Recall: In Mutual exclusion (Ricart-Agarwala - RA):

   1. Safety means that NO two processes can simultaneously get access to the exclusive part.
   2. Liveness: Liveness means a process that requests access will eventually get the access.
   3. Causal ordering: If two request R1, R2 are such that R1 -> R2, then R1 should be granted before R2

   Now let us look at the Modified RA algorithm (call it MRA) that uses (Pi, Ti) instead of (Ti, Pi). Lexicographical order on (Pi, Ti) will always prioritize any request from a process with lower Pi.

   Intuitively, it feels like the MRA should satisfy safety because each request (Pi, Ti)is still unique and will violate liveness because higher pid processes may never get access if lower pid processes keep requesting access. But the way the RA algorithm works, the answer is exactly the opposite! That's why intuition can be a guide but can't replace carefully analyzing/formalizing the protocols.

   Safety: *MRA violates safety (4 points).*
   Key point to note: In Original RA, safety is preserved by a process with lower timestamp acting as a gatekeeper and disallowing any request with higher timestamp from going before it. When we reverse the order of Pi and Ti, that ability is lost. (Safety in RA does NOT depend on the fact that Requests have unique id alone)
   Consider this example:
   1. 3 processes P1, P2, P3. P1 is currently holding the critical section (CS). P3 sends "*Request*" messages to all. P2 will "*Reply*". P1 will not *Reply* yet; it queues P3's *Request*.
   2. After sometime, P2 sends *Request* to all. P3 will *Reply* since process id of P2 is less than P3. P1 will not *Reply* yet; ; it queues P2's *Request*.
   3. Now both P2 and P3 are waiting on just P1's *Reply* to enter the CS.
   4. After sometime, P1 releases the CS and sends *Reply* to all queued requests, i.e P2 and P3
   5. P2 and P3 both have all 2 *Replie*s, and get access to CS => **Safety violation**

   Liveness: MRA guarantees liveness*(4 points)*.
   Key point to note: What ensures liveness is the "Reply to ALL" step while exiting the CS
   Proof: We will show a process's *Request* will get a *Reply* from ALL other processes eventually - which means it gets to enter the CS
   1. There are only two reasons that a process Q's *Request* will not get a *Reply* from process P:
      a. P is in *Held* state OR

    b. P is in *Wanted* state and Q's id > P's id
2. In case a, P will eventually release the CS and when it does, according to RA, it sends *Reply* to ALL *Requests* it has received so far. So Q will get the *Reply* eventually
3. In case b, either:
    a. Process P eventually gets to enter the CS (going from *Wanted* to *Held*), then releases CS after sometime and sends *Reply* to all *Requests* as above OR
    b. Process P itself never gets to enter CS. But this can never happen because:
        i. If it does, we can repeat the argument at (1) with P instead of Q
        ii. P can only be blocked by a process with id less than P.
        iii. We can keep repeating this until we reach P1 (lowest id)
        iv. Nothing can block P1's *Request* - all process will send *Reply* to P1's *Requests* eventually
4. Thus eventually, any process Q will get *Replies* from all other processes and can enter CS (Note however that this access can violate safety and multiple processes maybe inside the CS. One can wonder, of what value is such a liveness?)

Causal ordering: NOT guaranteed. Counterexample *(2 points)*:
Say we have (P4, 34) -> (P0, 35) i.e happen before relationship between the two. If the processes send *Request*s to access CS at these timestamps, it is possible (P0, 35) gets to enter CS before (P4, 34) since pid's 0 < 1 (In original RA, P4 would have been the gatekeeper preventing P0 from entering since P4 has lower timestamp. But in modified RA, we remove that ability of P4 and it has to send Reply to P0.)

8. (Solution and Grading by: Kartik.)
   1. This is correct. Since all processes fail without delivering any messages, none of the correct processes violated any of the Virtual Synchrony rules.
   2. This is correct. This is an example of partitioning in virtual synchrony.
   3. This is incorrect. Views must be consistent in Virtual Synchrony. P3 can not consider P1 and P2 to be in its view, while P1 and P2 don't consider P3 to be in its view. The next view change at P3 should have been {P3}.
   4. This is correct. This is an example of partitioning in virtual synchrony.
   5. This situation is correct. When Pi joined the system, all processes received the same view change. When Pi left the system, all other processes agreed on the view change which removed Pi, and Pi received a view with itself. This is fine, and an example of view partitioning. In between the view changes, no Virtual Synchrony rules were violated since no multicasts were sent.
   6. This is incorrect. Ni's multicast should have been delivered by all correct processes in the same view as the multicast send event. Alternatively, If Ni is no longer in the view, then the multicast should not be delivered.
   7.This is incorrect. All correct processes in a view must deliver the same set of multicasts. Ni should have delivered the multicasts after it was included in the view.

9. (Solution and Grading by: Lilia.)
   I. Yes, the algorithm is safe. For any two processes Pi and Pj, their voting sets must intersect, and to enter both their critical sections, they still must receive approval from all members of their voting sets.
   II. Yes, it is deadlock-free.
   Among all processes in the deadlock cycle, let Pi be the process waiting for the vote from the process Pmin with the smallest ID in the cycle.

   If Pi is waiting on Pj, Pj has voted for a different process Pk and is waiting on it. Since Pj has the vote of Pi, Pj must be waiting on a process Pk with a higher ID than Pi. If the cycle is Pi -> Pj -> Pk -> … -> Pi, then i < j < k < i which is not possible.

   Therefore we cannot have a deadlock.

   III. It is starvation-free with FIFO/fair queuing, not if no fair queuing.

   Proof assuming FIFO:

   Suppose P calls enter(). P can progress to the Hold state once P receives votes sequentially from V = {P1, …, Pk}.

   Assuming FIFO ordering, we want to show that P's requests for each process of V will eventually reach the front of the queue because the number of requests preceding each is bounded, and that each process will be dequeued.

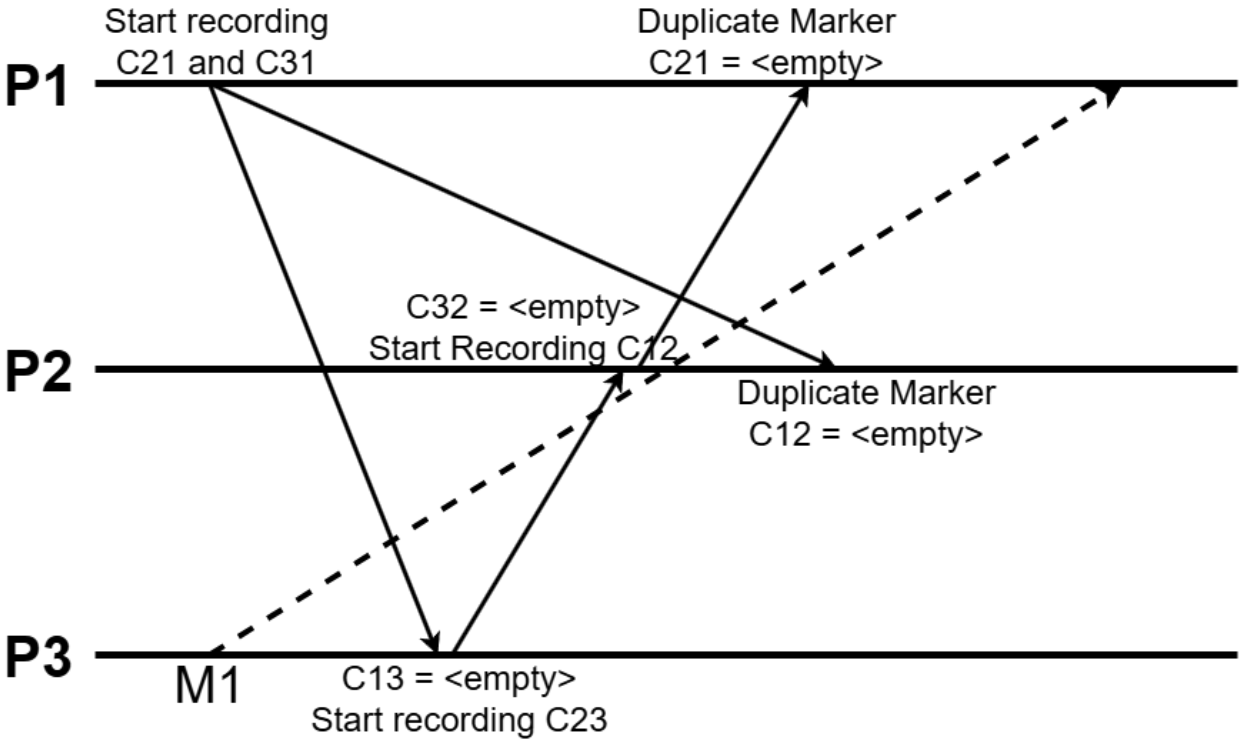
   For each Pi in V, if P does not receive an immediate reply from Pi, then P queues Pi. Note that P is in a finite number of voting groups. If a process Pj appears in the queue before Pi, it must requeue itself in a second enter call again before sending the request again to P. By FIFO queuing, a subsequent request must be queued after Pi. This means that the number of requests occurring before it in the queue is bounded.

   By assumption of no deadlocks, processes that are sent a response will reply, so a process at the front of the queue will be dequeued.

   Therefore, it is starvation-free assuming FIFO ordering.

10. (Solution and Grading by: Tianchen Wang.)

i. The algorithm is wrong. (No doubt the spacecraft doors won't open.)
Below is a counterexample:



Two main issues are:

1. Message M1, which should be recorded in the state of channel C31, will be dropped because P3 never sends a marker to P1.
2. Some processes may never receive (N-1) markers, in this case, P3. The algorithm will not terminate.

(Any other counterexample that shows the incorrectness is accepted.)

ii. Two fixes:

1. If Pi is seeing the Marker Pi for the first time:
   for j=1 to N except i: // Removed k
   Pi sends out a Marker message on outgoing channel Cij
2. In else:
   Stop recording Cki and mark that channel's state as *all the messages that have arrived on it since recording was turned on for Cki.*

In this way, the algorithm becomes the Chandy-Lamport Global Snapshot Algorithm

iii. Open-ended question. Yeah!