Homework 3 Solutions: CS425 FA26

Question 1

a. The step that fails is the one that turns a missing message into a crash: “a third process pk must have sent v to pi, but then crashed before sending v to pj.” That inference is the engine of the whole contradiction. Under asynchrony, pk need not have crashed at all: its message to pj may simply still be in flight. So basically the main assumption that is leading to the contradiction that completes the proof is not true anymore, so the proof can’t be used.
b. Three processes with inputs 0, 1, 1 and f = 1, so two rounds, deciding min. Nobody crashes. Delay P1’s round-1 message to P3, and P2’s round-2 relay to P3, past the decision point. P1 and P2 finish holding {0,1} and decide 0; P3 finishes holding {1} and decides 1. Agreement is broken by delay alone.

Question 2

State vector layout: [NHX, E, S]

At NHX:
===
R(E1): E1 = [0,1,0]
- D(E1): [0,1,0]

S(N1) = [1,1,0]
- D(N1): [1,1,0]

R(E2): E2 = [1,2,1]
- B(E2): [1,1,0], buffer = {E2}

S(N2) = [2,1,0]
- D(N2): [2,1,0], buffer = {E2}

R(S2): S2 = [1,1,2]
- B(S2): [2,1,0], buffer = {E2, S2}

R(S1): S1 = [1,1,1]
- D(S1): [2,1,1]
- D(S2): [2,1,2]
- D(E2): [2,2,2]

(note: S2 and E2 could be delivered in either order here)

At E:
===
S(E1) = [0,1,0]
- D(E1): [0,1,0]

R(N1): N1 = [1,1,0]
- D(N1): [1,1,0]

R(S1): S1 = [1,1,1]
- D(S1): [1,1,1]

S(E2) = [1,2,1]
- D(E2): [1,2,1]

R(S2) = [1,1,2]
- D(S2): [1,2,2]

R(N2) = [2,1,0]
- D(N2): [2,2,2]

At S:
===
R(N1): N1 = [1,1,0]
- B(N1): [0,0,0], buffer = {N1}

R(E1): E1 = [0,1,0]
- D(E1): [0,1,0], buffer = {N1}
- D(N1): [1,1,0], buffer = {}

S(S1) = [1,1,1]

- D(S1): [1,1,1]

S(S2) = [1,1,2]

- D(S2): [1,1,2]

R(N2): N2 = [2,1,0]

- D(N2): [2,1,2]

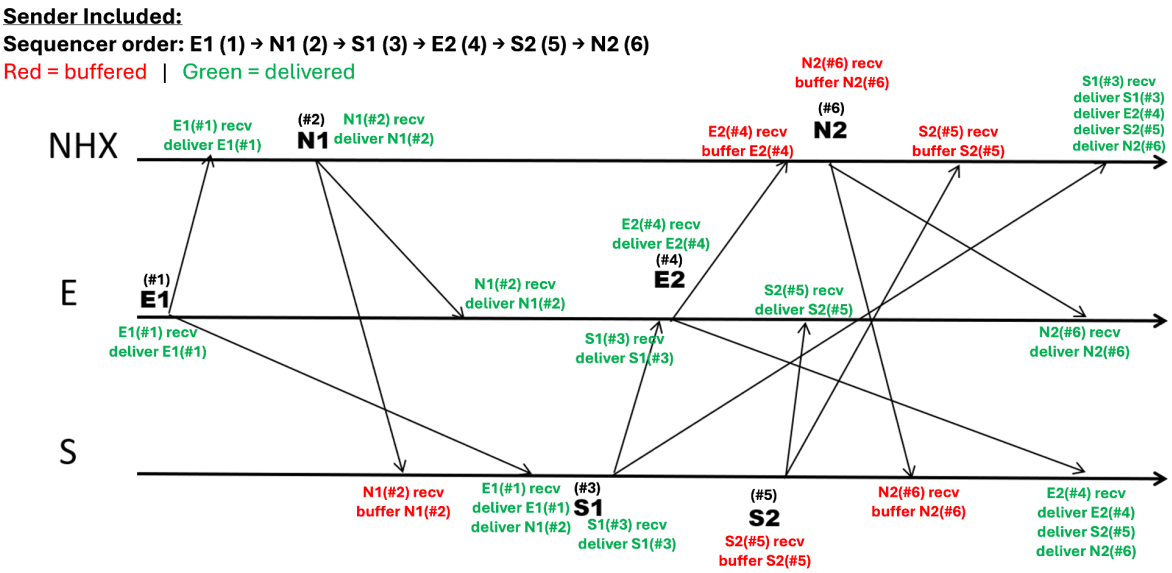
R(E2): E2 = [1,2,1]

- D(E2): [2,2,2]

Question 3

In sequencer-based total ordering, each multicast message receives a global sequence number from the sequencer. Since the latency between each process and the sequencer is zero, messages are assigned sequence numbers according to their physical send times. From the diagram, the global ordering is: **E1 (1) → N1 (2) → S1 (3) → E2 (4) → S2 (5) → N2 (6).**

Sender Included



- **NHX: E2 (#4), N2 (#6), S2 (#5)**
- **E: None**
- **S: N1 (#2), S2 (#5), N2 (#6)**

Question 4

(i) (p'=p) does not imply (e'=e). The set C excludes only the specific event (e=(p,m)), not all events at process (p). Thus, (e'=(p,m')), where (m' != m), can still occur.
Since both events involve the same process, they cannot necessarily be reordered.
Case II instead considers a finite deciding run σ in which (p) takes no steps. Applying σ to (C_0) gives a decided configuration (A). Since σ does not involve p, it commutes with e, and with the sequence: e' followed by e. Therefore e(A) = σ(e(C0)) = σ(D0), which is 0-valent, and e(e'(A)) = σ(e(e'(C0))) = σ(D1), which is 1-valent.

(ii) Yes. Lemma 3 holds for any event e applicable to the bivalent configuration C. However, fair event selection is required in the full FLP proof to construct an admissible infinite run, ensuring that every non-faulty process continues executing and its messages are eventually delivered.

(iii) Yes, it is safe. Paxos safety depends on one condition: any Election quorum Q1 must share at least one node with any Bill quorum Q2, i.e., |Q1| + |Q2| > N. Here (N/3 + 1) + (2N/3 + 1) = N + 2 > N. Even with floors, $\lfloor N/3 \rfloor + \lfloor 2N/3 \rfloor \geq N - 1$, so the sum is still ≥ N + 1 > N. Any Q1 and Q2 therefore intersect.
However, liveness and fault tolerance are reduced, because the larger Bill quorum requires responses from more acceptors. In conclusion, the modified Paxos is safe, but less available under failures.

Question 5

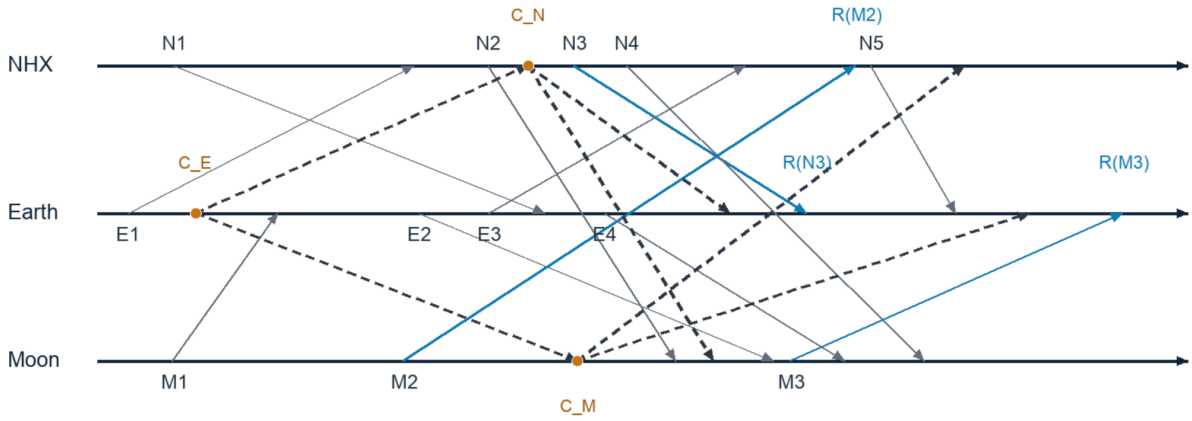
a. Causality violations: $R(N3)$ and $R(M3)$ are captured in the channel states, but their send events are absent from the senders' local snapshots. FIFO violation: M2 is sent before the Moon→NHX marker but received after it. (answer any two of those is fine)

b. Root cause: The channels violate FIFO ordering, which Chandy–Lamport assumes. N3 and M3 overtake their markers, while the Moon→NHX marker overtakes M2.

c. Keep all markers and send events unchanged. Adjust only these receive events:

- Move $R(N3)$ after the NHX→Earth marker, but before $R(N5)$.
- Move $R(M3)$ after the Moon→Earth marker.
- Move $R(M2)$ before the Moon→NHX marker, but after NHX's snapshot.

The corrected timeline is shown below. Blue arrows indicate the modified deliveries; dashed arrows are markers. (correct any two of those is fine)



Question 6

Assume N processes with unique IDs, with k between 1 and N, and eventual message delivery, as in the lecture. The processes form a logical ring, and messages travel clockwise. The desired output is the set of the k smallest IDs among non-faulty processes.

The lecture's asynchronous model does not permit an algorithm that guarantees both Safety and Liveness in every execution containing failures. Therefore, I first present a modification that works without failures, then explain its limitations and guaranteed fault tolerance.

Modified Ring Election Algorithm

Initially, every process sets `elected = NULL`.

1. Any process r may initiate an election by sending `ELECTION(initiator = r.id, S = {r.id})` to its successor. The initiator's ID is stored separately from the candidate set S.
2. When another process q receives this message, it adds its ID to S and retains only the k smallest IDs. If fewer than k IDs have been collected, it retains all of them. It then forwards the message.
3. When the message returns to its initiator, it has completed a full traversal. The initiator sets `elected = S` and sends `ELECTED(initiator = r.id, S)` around the ring.
4. Every process receiving this announcement sets `elected = S` and forwards it. The initiator removes the announcement when it returns, completing the election.

For simultaneous initiations, use the mechanism from slide 27: every message includes its initiator's ID, and each process remembers the greatest initiator ID it has seen. Messages from smaller-ID initiators are suppressed. Each process initiates at most once within this election run, so there are finitely many competing initiations.

The priority of an initiator is independent of the selection criterion. A higher-ID initiator's election takes priority, but the elected set still contains the smallest k process IDs.

Safety Without Failures

The election message maintains the following invariant:

Its candidate set contains the smallest k IDs among all processes visited so far, or all visited IDs if fewer than k processes have been visited.

The invariant holds initially. At each hop, adding the receiving process's ID and retaining the smallest k preserves it. A discarded ID cannot become necessary later because at least k smaller IDs have already been encountered.

When an election message returns to its initiator, it has visited every process. Its candidate set therefore contains exactly the global smallest k IDs. Every announcement carries this same correct set, regardless of which process initiated it.

Thus, every non-NULL output is correct, satisfying Safety.

Liveness Without Failures

Among the finitely many initiators, one has the greatest ID. Its election message is never suppressed by a competing initiator.

Because no process fails, every recipient forwards the message. Eventual delivery ensures that the message returns to its initiator. The resulting announcement also completes a full traversal.

Consequently, every process obtains a non-NULL result and the election terminates.

Complexity Without Failures

With one initiator, the election message traverses N links, and the announcement traverses another N links.

- Message complexity: 2N transmissions, or O(N).
- Completion time: 2N sequential message-transmission steps.
- Candidate-set size per message: O(k).
- Total candidate-ID transmission cost: O(Nk).

If each hop takes at most a fixed amount of time, completion time is O(N). Eventual delivery alone provides no fixed real-time bound.

With m simultaneous initiators, the message complexity is O(mN). This becomes $O(N^2)$ when all N processes initiate.

Guaranteed Fault Tolerance

Suppose $k < N$ and consider $N = k + 1$ processes. If the smallest-ID process crashes before the election, Liveness requires the remaining k processes to decide on their own IDs. But if that process is merely delayed, the others cannot distinguish this execution from the crash execution and may decide before receiving its messages. Their decision is then incorrect because it excludes a live process, violating Safety. Waiting indefinitely would violate Liveness in the crash execution.

If $k = N$, one crash already leaves too few non-faulty processes to produce the required output. Therefore, under the lecture's assumptions, the guaranteed tolerance of arbitrary crashes during election is zero.

Question 7

Yes, it is a valid Raft state. One of the sequences that could have produced it is given below:

	Raft log				
Event	**S1**	**S2**	**S3**	**S4**	**S5**
All logs start out empty	**-**	**-**	**-**	**-**	**-**
Term 1: S1 wins the election. It appends 3 entries and fails before *AppendEntries* are sent out.	1,1,1 (leader)	-	-	-	-
Term 2: S4 times out first and wins the election as all logs of S2,S3, and S5 are still empty (since S1 has failed)	1,1,1	-	-	- (leader)	-
S4 appends 2 entries and fails immediately after *AppendEntries* on S5 succeeds.	1,1,1	-	-	2,2 (leader)	2,2
Immediately after S4 fails, S1 comes back up. It discovers that they are currently in term 2, updates its term and becomes a Follower.	1,1,1	-	-	2,2	2,2
Term 3: Since the leader S4 has failed, nodes receive no keepalive *AppendEntries*. S1 times out first and requests votes. S2, S3 have an empty log and they grant their vote. S5 refuses to grant its vote since the term of its last log entry (2) is greater than that of S1 (1). However, S1 only needs 3 votes to become the leader (itself, S2 S3).	1,1,1 (leader)	-	-	2,2	2,2
S1 replicates its log to both S2 and S3.	1,1,1 (leader)	1,1,1	1,1,1	2,2	2,2
Before S1 can replicate any further, S4 comes back up, discovers a new term and becomes a follower. This snapshot is the desired state.	**1,1,1 (leader → term 3)**	**1,1,1**	**1,1,1**	**2,2**	**2,2**

The Raft log entries above are mentioned in a <term>, <term> fashion. So a log of 1,1,1,2,2 means three entries of term 1 and two entries of term 2.

Question 8

(i) Election protocol

All communication, including notifications, uses Chord end-to-end routing messages. Only the elected leader is exempt from the per-node O(1) initiation requirement.

Each participant P sends route(0, ELECTION(e)). This message reaches Z = successor(0), the peer with the lowest ID. Let L = successor(Z). Since Z has the lowest ID, L has the second-lowest ID.

Upon the first ELECTION request for epoch e, Z sends route(L.id, BECOME_LEADER(e)) and ignores duplicate requests for that epoch. L then sends route(L.successor.id, ELECTED(e, L)). Each peer that receives it records L as the leader and forwards it once to its own successor, until the notification returns to L.

(ii) Safety and liveness

Safety: All requests to ring ID 0 reach the same unique peer Z. Its successor is the same unique peer L, and the notification carries this same leader identity to every peer.

Liveness: With correct routing and no failures, the election requests reach Z, Z's message reaches L, and the ELECTED notification traverses the entire successor ring. Therefore, every peer learns the leader.

(iii) Completion time and message complexity

The protocol uses N ELECTION requests, one BECOME_LEADER message, and N ring notifications, so Θ(N) end-to-end routing messages in total. Every peer, including Z and L, initiates O(1) routing requests. The requests take O(log N) hops each and the notifications one hop each, so the total number of hop transmissions is O(N log N).

Assuming concurrent requests and negligible processing delays, identifying the leader takes O(log N) time. The notification travels around the ring sequentially, taking O(N) time, so total completion time is O(N).

(iv) Failures and inconsistent routing information

Incorrect finger-table entries alone do not necessarily cause a lookup to fail. If successor pointers remain correct, the finger entries point to live peers, and routing can fall back to successor pointers, lookups can still reach the correct destination, though they may be slower.

However, if a failure leaves successor pointers inconsistent, or a route depends on a stale pointer to a failed peer without a fallback, messages may fail to reach the correct

node. If Z fails during the election, later route(0) requests may reach a different node, which may notify its own successor, so two peers may each believe they are the leader. If L fails, the elected leader is unavailable, and a failed peer on the ring can stop the ELECTED notification from reaching the remaining peers. Thus safety and liveness are not guaranteed during the failure.

After the routing state stabilizes, the participants can retry using a new election epoch.

Note: Under the clarification posted on Piazza, the coordinator Z is also exempt. In that case, each request can carry P.id, and Z can reply to each participant with route(P.id, ELECTED(e, L)). This uses Θ(N) messages, and assuming concurrent replies, bounded per-hop delay, and negligible processing delays, completion time is O(log N). This approach is also accepted.

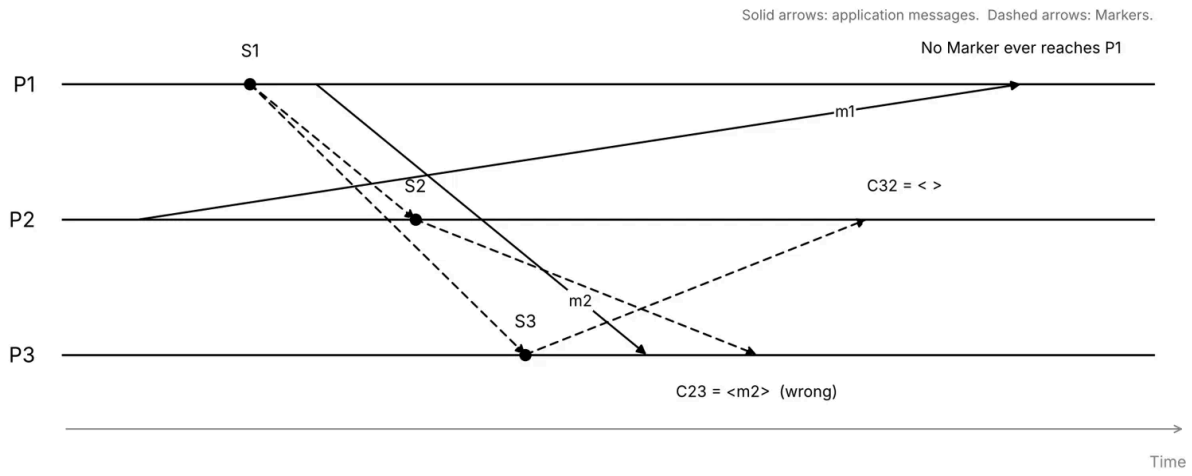
Note: Another accepted approach uses only the leader exemption. Each participant P first performs lookup(0), which returns Z's ID through the standard lookup reply. P then sends route(Z.id + 1, REGISTER(e, P.id)), which reaches L. L replies to each participant with route(P.id, ELECTED(e, L)). Each non-leader initiates O(1) requests, L initiates Θ(N), and the protocol uses Θ(N) messages with O(log N) completion time.

Question 9

- Yes, a leader without X can be elected because the commit quorum (3) and election quorum (4) does not have to overlap as 3 + 4 = 7 is not greater than 7. So a candidate can win with votes from servers that never received X.
- Example sequence:
    1. Term 1: A replicates X to B and C and marks X as committed. D, E, F,G have empty logs.
    2. A crashes
    3. D times out, moves to term 2, votes for itself and requests votes.
    4. E, F, G vote for D (since their logs are empty). B and C refuses (since their logs aren't empty)
    5. wins with 4 votes (D, E, F, G) and becomes leader without X in logs.

Question 10

i. No, the algorithm is not correct. Counterexample: three processes P1, P2, P3, where Cij is the channel from Pi to Pj, P1 is the initiator, and all channels are FIFO. Solid arrows are application messages and dashed arrows are Markers.



1. P2 sends application message m1 to P1. It is slow and stays in transit for a while.
2. P1 initiates. It records its state S1, sends a Marker to P2 and to P3, and starts recording C21 and C31.
3. P1 sends application message m2 to P3. It is sent after the Marker, so it is behind the Marker on C13.
4. P2 receives the Marker on C12 (its first Marker). It records S2, which includes the send of m1. It sets C12 to empty and sends a Marker only to P3 (it skips P1, since k = 1). It starts recording C32.
5. P3 receives the Marker on C13 (its first Marker). It records S3, sets C13 to empty and sends a Marker only to P2 (it skips P1). It starts recording C23.
6. P3 receives m2 on C13, after the Marker, as FIFO requires. m2 is not in S3, and P3 is not recording C13.
7. P3 receives P2's Marker on C23 (its second Marker). By the algorithm, C23 becomes all messages received on all incoming channels since P3's first Marker, which is <m2>.
8. P2 receives P3's Marker on C32. C32 becomes all messages P2 received since its first Marker, which is empty.
9. m1 arrives at P1 on C21 while P1 is still recording.

Two things go wrong.

(a) The algorithm never terminates. A process never sends a Marker back to the process its first Marker came from, so that process can get at most N-2 Markers. Here, P2 and P3 both got their

first Marker from P1, so P1 receives 0 of the N-1 = 2 Markers it needs. Recording on C21 and C31 never stops, and m1 is never reported. m1 was sent before S2 and arrives after S1, so it should be in C21.

(b) The snapshot violates causality. It records C23 = <m2>, which means m2 is in transit in the snapshot. But m2 was sent after P1 recorded S1, so its send event is not in P1's recorded state. A message whose send is not in the snapshot cannot be in transit in it. Also, m2 did not even travel on C23.

ii. Two fixes:

1. In the first-Marker branch, send Markers for j = 1 to N except i. That is, remove k from the exception.
2. In the else branch, stop recording C_ki and set its state to the messages that arrived on C_ki since recording was turned on for C_ki. Do not use the messages on the other channels.

With these two changes, the algorithm is the Chandy-Lamport algorithm from lecture. On the timeline above, P2 and P3 now also send Markers to P1. By FIFO, m1 reaches P1 before P2's Marker. The algorithm terminates with C21 = <m1> and every other channel state empty, which is a correct snapshot.

iii. "One small step for a human, one giant leap for Doraemon and Einstein, who are waving at me from the window."