HW3 Solutions: CS425 FA23

1. (Solution and Grading by: Chaitanya.)

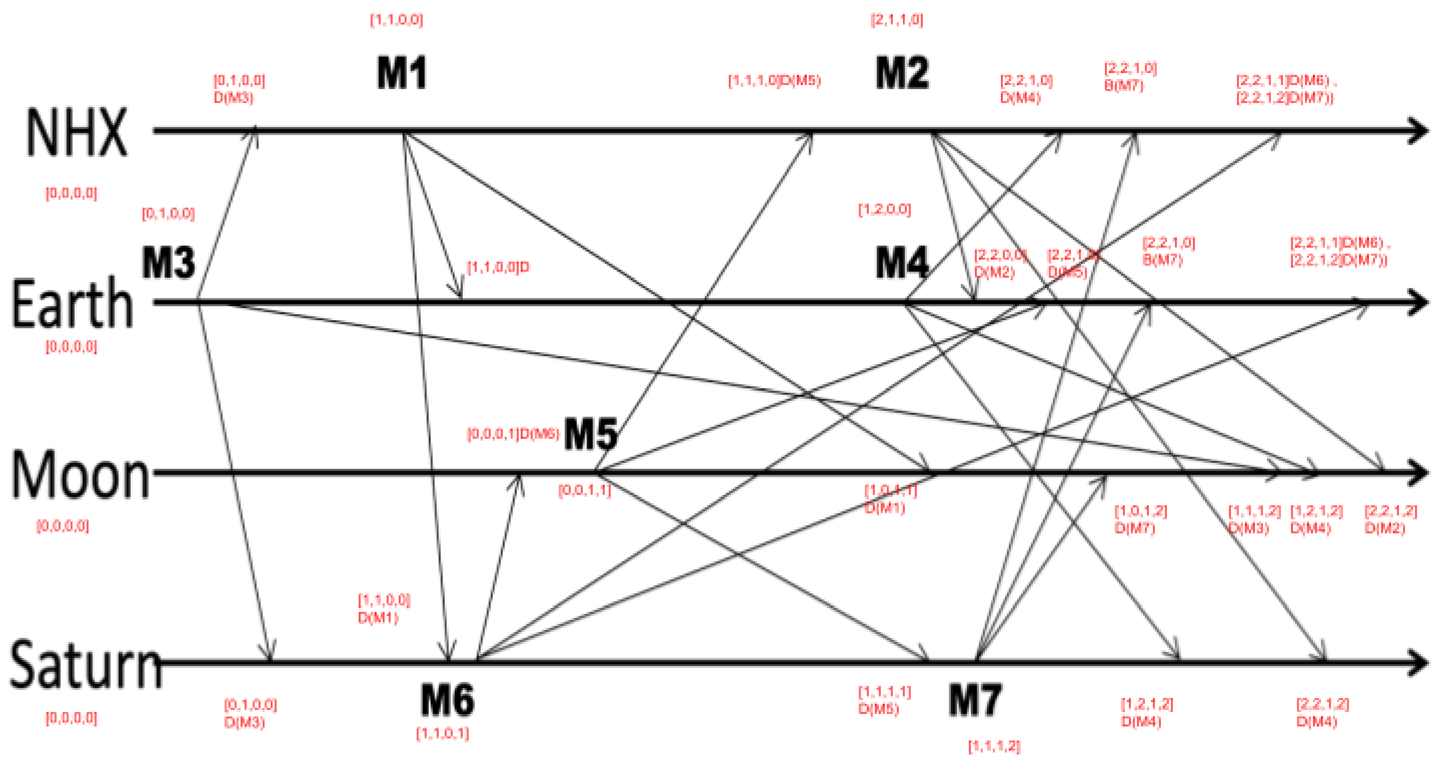
For an asynchronous system, there are no bounds on message transmission delays. Therefore, the max message transmission delay measured when the system was synchronous would not hold valid anymore. Consequently, there would be no way to enforce the following inequality:

Round length >> max transmission delay

If the aforementioned inequality does not hold, certain messages may not reach all the participating nodes even by the end of f+1 rounds (this is just one of the consequences that the system may end up with)! This would violate the safety property of consensus.

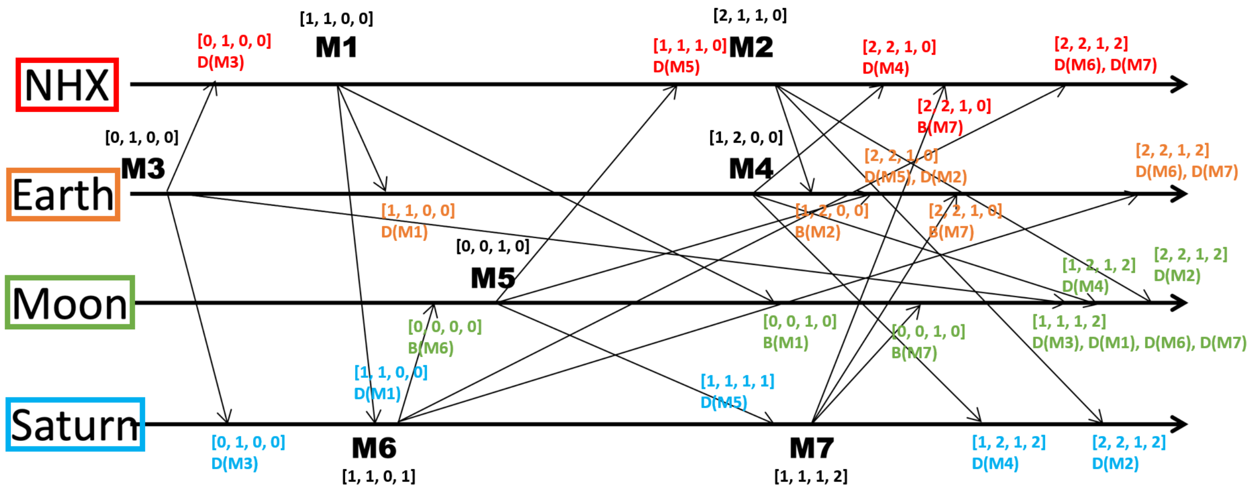
2. (Solution and Grading by: Kshitij and Shuyang.)

D(Mk) - Message Mk delivered
B(Mk) - Message Bk delivered



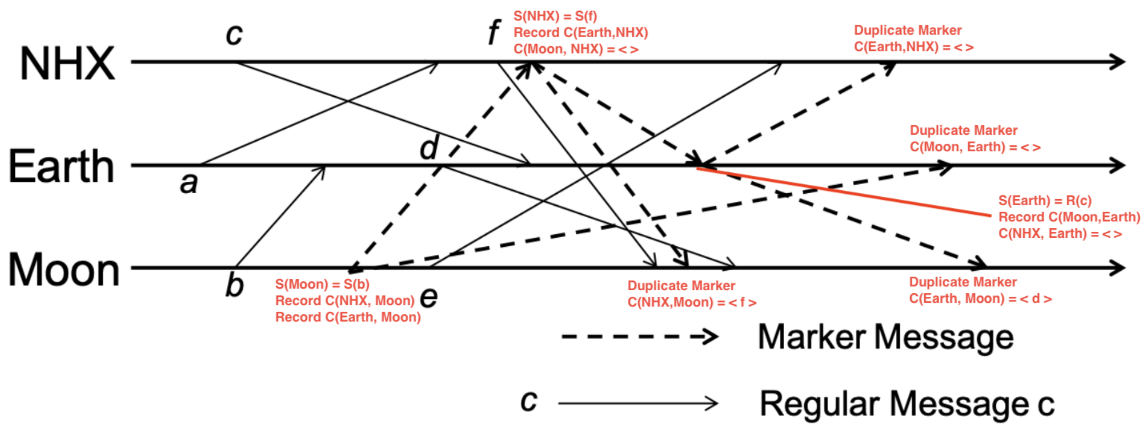
3. (Solution and Grading by: Christina and Neil.)

D(Mk) : Message Mk delivered
B(Mk) : Message Mk buffered



4. (Solution and Grading by: Aman and Shuyang.)
    a. Events are defined by both process and message. Suppose e=(p,m) and e' = (p,m'). We still have e!=e' as long as m!=m'. In other words, the event e can be prevented from being applied (to derive C) but this doesn't mean p(=p') cannot process other events!
    b. A protocol claiming to solve consensus must be able to do so in a finite number of steps (even if we don't know the value of that finite!). Hence C0 (and really any state!) must have a deciding run from it - if not, it immediately follows that no protocol can solve consensus in a finite number of steps.
    c. e must be applicable to the original (bivalent) configuration, but it can be any event applicable to the original bivalent configuration.

5. (Solution and Grading by: Lavanya and Colin.)



Process States:
NHX = S(f)
Earth = R(c)
Moon = S(b)

Channel States:
C(Earth, NHX) = Null
C(Moon, NHX) = Null
C(NHX, Earth) = Null
C(Moon, Earth) = Null
C(NHX, Moon) = f
C(Earth, Moon) = d

6. (Solution and Grading by: Shulin.)
   Modified Quorum-based election Algorithm:
   1. When a process detects a failure among its k leaders, it initiates a new round of leader election. It broadcasts an PROPOSAL message along with its id to all other processes. Each process maintains a list of received PROPOSAL messages. Once the length of the list reaches k, it moves on to the next voting phase.
   2. Voting phase: every process picks the top k processes with the lowest ID. It sends a VOTE message to those k processes while waiting for any VOTE message from other nodes.
   3. Election phase: When a process receives at least majority VOTE messages (N/2 + 1 votes where N is the total number of processes), it elects itself as a leader and broadcasts an ELECTION message to all other processes. A list of elected processes is maintained on every process. Once a process receives k ELECTION messages, we complete this round of leader election and inform every node about its list of elected processes.
   Safety is ensured since every process will only vote for up to k leaders with lowest IDs, and a process will only become a leader if it has the majority of votes.
   Liveliness is ensured because each process will eventually receive k acknowledgments from the elected leaders. Since the election of each leader requires a majority of votes, there will be an overlap in the majorities, ensuring that all non-faulty servers agree on the set of k leaders.

7. (Solution and Grading by: Maleeha.)
   Bully algorithm relies on a timeout to detect failures. Let’s say the current leader is very slow and we work with synchrony assumptions, a process x may incorrectly detect failure and initiate elections. Assuming the leader is very slow, let’s say the election completes. We violate the safety of the system.
   Note that in comparison, in the synchronous system, a process that does not respond within the upper bound of message RTT is definitely failed.

8. (Solution and Grading by: Shubhi.)
   Consider a system of N processes P1 through Pn (N=16).

   Assumptions -
   N = 16 and we use the matrix below to decide the voting set of each process, i.e., each process Pi's voting set is Vi = {Pi's row} union {Pi's column}.

P1	P2	P3	P4
P5	P6	P7	P8
P9	P10	P11	P12
P13	P14	P15	P16

   Let P1 currently be in the CS. Then P2 and P4 request to enter the CS. From above we can see that {P1, P2, P3, P4} are the only common processes in voting sets V1, V2, V3 and V4.

   The requests of P2 and P4 to enter the CS get queued up at the 4 intersecting processes P1, P2, P3 and P4. When P1 sends out the release message to V1 all the 4 intersecting processes in V1, that is, P1, P2, P3 and P4 will send a reply message to all the processes (P2 and P4) present in the queue. Thus, now both processes P2 and P4 have the permission to enter the CS. This violates safety.

9. (Solution and Grading by: Lilia.)
    a. True. Partitioning is possible in vsync.
    b. False. If P1 contains {P1, P2} then P2 should also deliver the same view
    c. True. A node can be removed for any reason (including false detections), regardless of whether it sent multicasts. (See L16 pg 45)
    d. False. All multicasts sent in a view (regardless of the failed state of sender afterwards) must be delivered atomically and reliably at all processes that survive into the next view (and must be delivered in the old view, i.e., before the next view).
    e. False. A node cannot join in the "middle of the view". A new view change has to be delivered, and the earlier multicast must be delivered in the previous view.

10. (Solution and Grading by: Taksh.)

a.

```
while True:
    if not is_holding_token:
        # Pass the token to the next process in the ring
        send_token()
        continue

    if requesting_access:
        if access_type == "read":
            # Check if it's safe to grant read access
            if no_write_access_requests() and (readers_count < k):
                grant_read_access()
                readers_count += 1
        elif access_type == "write":
            # Check if it's safe to grant write access
            if no_read_access_requests() and (writers_count < k/2):
                grant_write_access()
                writers_count += 1

    # Release access when done

        release_access()
        if access_type == "read":
            readers_count -= 1
        elif access_type == "write":
            writers_count -= 1
```

b. The algorithm guarantees safety because it only grants access when it's safe to do so. Read and write access are only given after checking the number of processing with the same access type. Access is granted only when the process holds the token, ensuring mutual exclusion.
   i. writers_count variable keeps track of the number of processes currently holding write access. When a process requests write access, it can only be granted if writers_count is 0 (no other process is writing), and readers_count is less than or equal to k/2 (ensuring that at most k/2 processes may read simultaneously).
   ii. At most k processes may obtain read access to the file simultaneously. This property is satisfied by using the readers_count variable to keep track of the

number of processes currently holding read access. When a process requests read access, it can only be granted if readers_count is less than or equal to k.

iii. If a process holds write access (access_type == "write"), it won't grant read access to other processes (no_write_access_requests() is ensured). Similarly, if a process holds read access (access_type == "read"), it won't grant write access to other processes (since it checks that writers_count is 0).

c. This algorithm may livelock if a process repeatedly requests access but never gets the token. To address this issue and reduce the frequency of livelocks, you can implement a timeout mechanism. If a process doesn't receive the token within a certain time frame, it can periodically re-request access. This way, processes won't get stuck indefinitely waiting for the token.

d. Best case:
(k) * T
The token starts with the process that is writing. For the best case, the first reader could be the next process after the writer. Therefore, it takes (k) time units for the token to circulate among the k processes waiting to read.

Worst case:
(N-1) * T
The token starts with the process that is writing. The token must circulate around the entire ring (N -1 processes) before it reaches all the processes waiting to read. This takes N -1 time units (T).

============== END of HOMEWORK 3 SOLUTION ==============