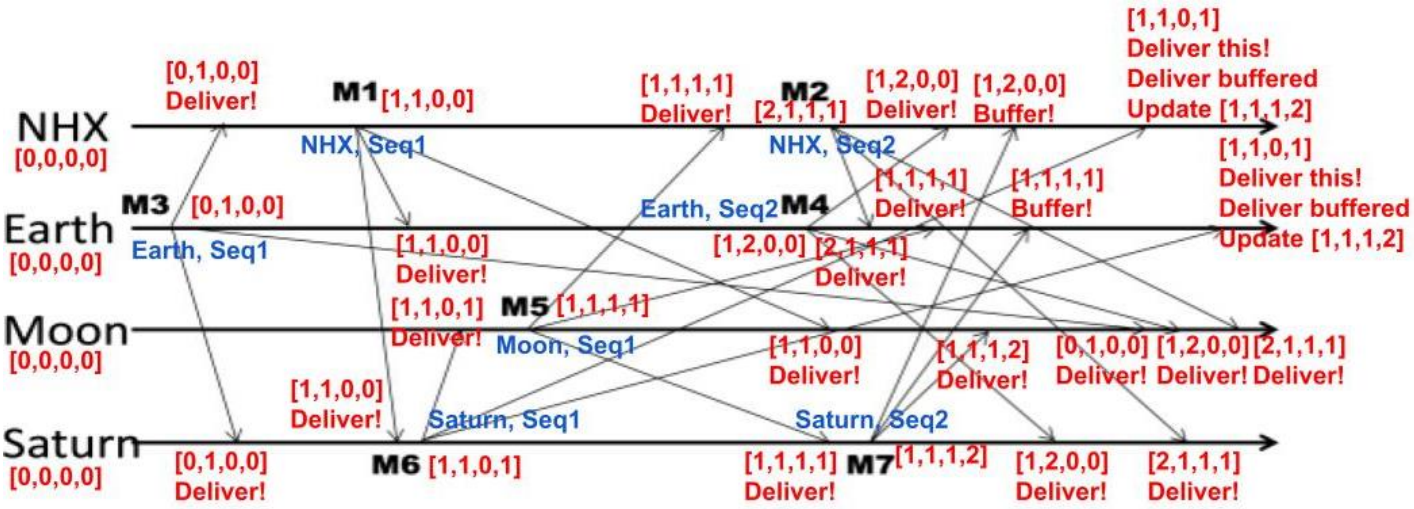
1.

Synchronous distributed system is known to follow the model where each message is received within bounded time, drift of each process' local clock has a known bound, and each step in a process is restricted to a bound amount of time. Although in Asynchronous distributed system it follows a model where there in no bound set on the process execution, drift rate of the clock is arbitrary, and there are no bounds on message transmission delays. The difference between these systems makes consensus solvable in synchronous systems and impossible to solve in asynchronous systems.

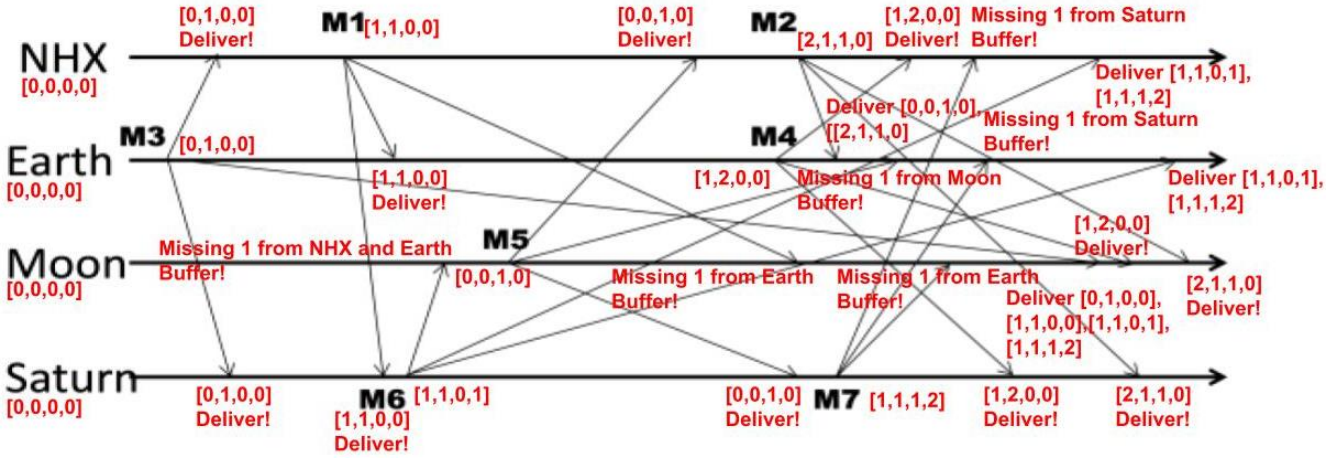
The point where synchronous consensus proof breaks down when the system is asynchronous:

- It is known that synchronous systems have set bounds on message delivery and processing timings while asynchronous systems can have message delays and processing delays either arbitrarily long or short. Due to this bound set for synchronous system, all processes operate in rounds to achieve consensus, where round length is the maximum transmission delay. In the case of asynchronous systems, the transmission delay is not bounded with a maximum transmission delay, which breaks down the system when it is asynchronous.
- In synchronous consensus proof there is a set limit of processes that are expected to fail, and it makes sure that an extra round is initiated so that all rounds receive the same value. In the asynchronous system, processes can fail at any time and could take arbitrary amount of time to rejoin, which makes the system initiated by synchronous systems useless in asynchronous systems.

2.


NHX
[0,0,0,0]
Earth
[0,0,0,0]
Moon
[0,0,0,0]
Saturn
[0,0,0,0]
[0,1,0,0]
Deliver!
M1 [1,1,0,0]
NHX, Seq1
[1,1,1,1]
Deliver!
M2 [2,1,1,1]
NHX, Seq2
[1,2,0,0]
Deliver!
[1,2,0,0]
Buffer!
[1,1,0,1]
Deliver this!
Deliver buffered
Update [1,1,1,2]
M3 [0,1,0,0]
Earth, Seq1
Earth, Seq2
M4
[1,1,1,1]
Deliver!
[1,1,1,1]
Buffer!
[1,1,0,1]
Deliver this!
Deliver buffered
Update [1,1,1,2]
[1,1,0,0]
Deliver!
[1,2,0,0]
[2,1,1,1]
Deliver!
[1,1,0,1]
Deliver!
M5 [1,1,1,1]
Moon, Seq1
[1,1,0,0]
Deliver!
[1,1,1,2]
Deliver!
[0,1,0,0]
Deliver!
[1,2,0,0]
Deliver!
[2,1,1,1]
Deliver!
[1,1,0,0]
Deliver!
Saturn, Seq1
Saturn, Seq2
[0,1,0,0]
Deliver!
M6 [1,1,0,1]
[1,1,1,1]
Deliver!
M7
[1,1,1,2]
[1,2,0,0]
Deliver!
[2,1,1,1]
Deliver!

3.


NHX
[0,0,0,0]
Earth
[0,0,0,0]
Moon
[0,0,0,0]
Saturn
[0,0,0,0]
M1
M2
M3
M4
M5
M6
M7
[0,1,0,0] Deliver!
[1,1,0,0]
[0,0,1,0] Deliver!
[2,1,1,0]
[1,2,0,0] Deliver!
Missing 1 from Saturn Buffer!
Deliver [1,1,0,1], [1,1,1,2]
Deliver [0,0,1,0], [[2,1,1,0]
Missing 1 from Saturn Buffer!
[0,1,0,0]
[1,1,0,0] Deliver!
[1,2,0,0]
Missing 1 from Moon Buffer!
Deliver [1,1,0,1], [1,1,1,2]
[1,2,0,0] Deliver!
Missing 1 from NHX and Earth Buffer!
[0,0,1,0]
Missing 1 from Earth Buffer!
Missing 1 from Earth Buffer!
Deliver [0,1,0,0], [1,1,0,0],[1,1,0,1], [1,1,1,2]
[2,1,1,0] Deliver!
[0,1,0,0] Deliver!
[1,1,0,1]
[1,1,0,0] Deliver!
[0,0,1,0] Deliver!
[1,1,1,2]
[1,2,0,0] Deliver!
[2,1,1,0] Deliver!

5.

Initial State:

- Record state of initial state
- Send out markers to $C_{Moon \rightarrow Earth}$ and $C_{Moon \rightarrow NHX}$
- Turn on recording on channels $C_{Earth \rightarrow Moon}$ and $C_{NHX \rightarrow Moon}$
- **Mark $C_{NHX \rightarrow Moon}$ = <S(f) -> R(f)>**
- **Mark $C_{Earth \rightarrow Moon}$ = <S(d) -> R(d)>**

F:

- Record own state as F
- **Mark $C_{Moon \rightarrow NHX}$ = <>**
- Turn on recording on other incoming $C_{Earth \rightarrow NHX}$
- Send out markers to $C_{NHX \rightarrow Earth}$ and $C_{NHX \rightarrow Moon}$
- **Mark $C_{Earth \rightarrow NHX}$ = <>**

D:

- Record own state as D
- **Mark $C_{NHX \rightarrow Earth}$ = <>**
- Turn on recording on other incoming $C_{Moon \rightarrow Earth}$
- Send out markers to $C_{earth \rightarrow NHX}$ and $C_{Earth \rightarrow Moon}$
- **Mark $C_{Moon \rightarrow Earth}$ = <>**

Initial State : Mark $C_{NHX \rightarrow Moon}$ = <S(f) -> R(f)> ; Mark $C_{Earth \rightarrow Moon}$ = <S(d) -> R(d)>

F: Mark $C_{Moon \rightarrow NHX}$ = <> ; Mark $C_{Earth \rightarrow NHX}$ = <>

D: Mark $C_{NHX \rightarrow Earth}$ = <> ; Mark $C_{Moon \rightarrow Earth}$ = <>

6.

Solution for the k-Leader Election problem:

1. Each process will be separated into a group, where each process declares itself as a candidate.
2. Process sends its candidate information to other processes in its group.
3. Processes in each group wait and respond once to lowest id or is null in their group.
4. Process with highest majority (quorum) is selected as leader in each group.
5. In case where no processes receive majority in a group, a new round of election is conducted in the group until a process is declared as a leader.

The algorithm above maintains safety and liveness by dividing processes in groups allowing failure of a node in one group not to effect picking leaders in other groups. It is made sure in the algorithm that even when there is failure, a new round of election will be conducted until a leader is selected in a group, which makes sure Liveness property of the algorithm is maintained. The algorithm also makes sures that all processes in each group wait to hear from other processes in the group before picking their choice of leader, which makes sure Safety property of the algorithm is maintained.

7. Bully algorithms particularly can be challenging in the case of asynchronous systems due to the nature of arbitrary time spent in message delivery and unexpected process failures.
   For example:
   - There 5 processes P0, P1, P2, P3, and P4
   - P4 is considered as the current coordinator.
   - For reasons of network delay and arbitrary time spent in message delivery process P2 is currently experiencing a delay in sending and receiving messages to P4.
   - P2 by not receiving responses from the coordinator P4, consider coordinator P4 as failed and sends election request to process with higher id than itself.
   - P2, which is currently experiencing network delay from other nodes, receives no answer within timeout and calls itself leader while sending Coordinator message to all lower id processes.
   - P2's network becomes stable and processes that have lower id than P2, consider P2 as its leader while processes that have higher id than P2 consider P4 as its leader.
   - This can continue where there could be multiple leaders in an asynchronous system, leading to inconsistencies and problems in decision-making and coordination.

8. In the case of Maekawa algorithm, it maintains mutual exclusion by ensuring that only one process can access the critical section at a time and to break the mutual exclusion property of the algorithm multiple processes should access the critical section at the same time.

   A concrete scenario where this algorithm violates mutual exclusion:

   - There are 5 processes P0, P1, P2, P3, P4
   - Process group P0 = {P2, P3}; P1= {P2, P4}; P2= {P1, P4}; P3= {P1, P2, P4}; P4= {P1, P2, P3}
   - In this case P0 sends request messages to all processes in its group.
   - P2 and P3 sends their approval reply message back to P0.
   - Upon receiving their approval reply message, P0 access the critical section.
   - During this period, P1 and P4 send request messages to all processes in its group.
   - P0 sends release messages to all processes in its group.
   - Upon receiving the release message from P0, P2 and P3 immediately multicasts a Reply message to all waiting processes.
   - In this scenario processes P1 and P4 receives reply message from all its group process at the same time.
   - Causing P1 and P4 to access critical section at the same time, which in conclusion violates mutual exclusion.

9.

i. Incorrect. For a new view to be established, there must be an update to the original membership list, which can result from processes like joining, leaving, or failing. Therefore, when a new view is delivered to nodes present in the view, it should include all the nodes within that view, as the view effectively represents the membership list.

ii. Incorrect. This statement violates two rules. First, the creation of a new view should be triggered by a view change. In the case of V1 and V2 having the same view {V1, V2}, there is no reason to set V2 as a new view, as no view change has occurred. Additionally, the statement delivers two separate views for P1 and P2 in V2, which contradicts the requirement for a new view to have the same membership for all nodes within it, contingent on the nodes present in the new view.

iii. True. A node can join a view through a view change without the necessity to send any multicast messages while in that view. It can also leave the view through a view change, resulting in the creation of a new view.

iv. Incorrect. All multicasts, whether sent by a failed or non-failed node, should be received within the view where the multicast was initially sent. The statement is incorrect because it suggests that the multicast of a failed node can be received by a node that survived into the next view, which is not consistent with the rules.

v. Incorrect. If a node joins, leaves, or fails, it will indeed necessitate the creation of a new view. The statement's claim that a node can join in the middle of a view and become included in the current view is false, as nodes can only join through a view change, which creates a new view. Consequently, the remainder of the statement is also incorrect.