CS425 Fall 2022 – Homework 3

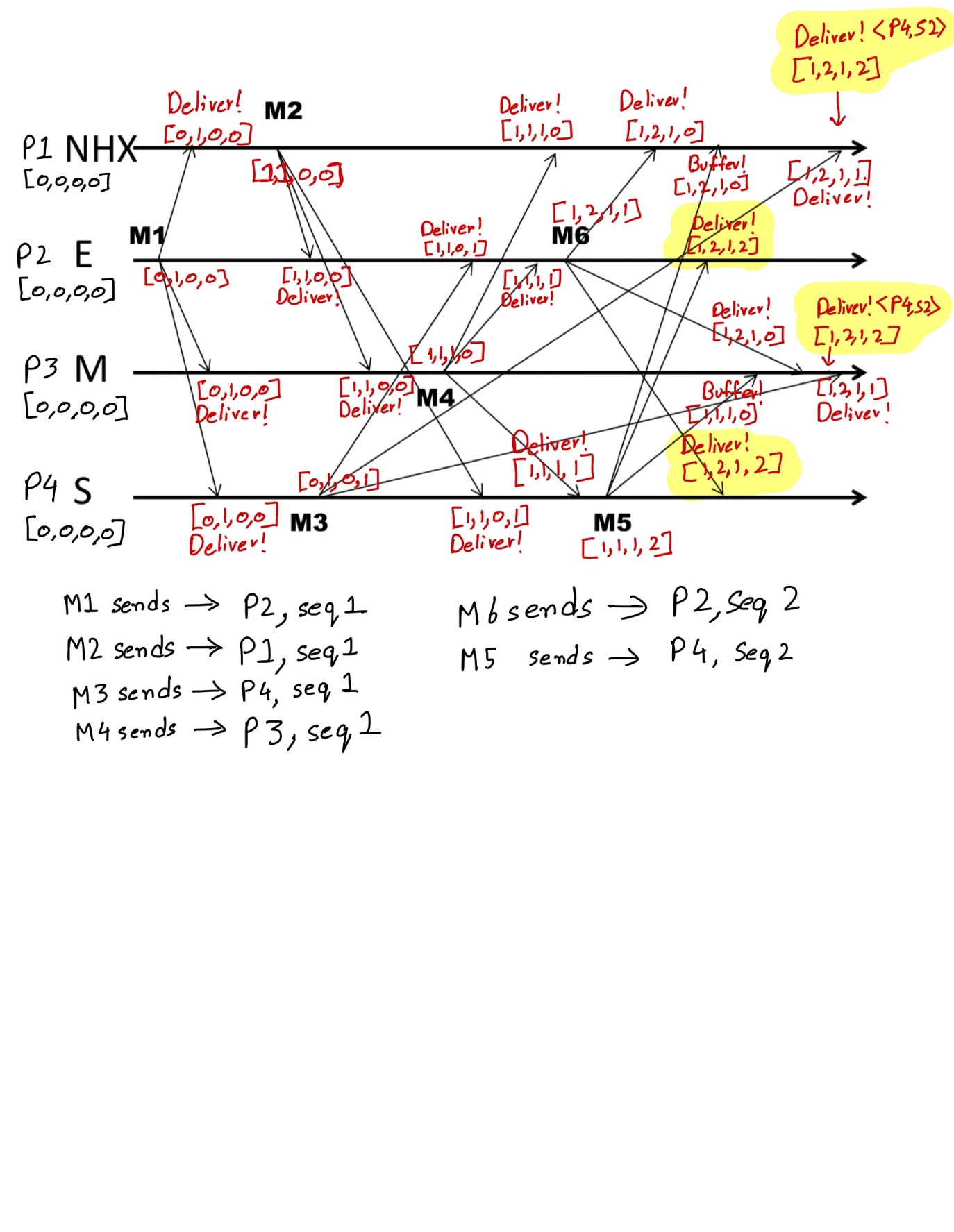
(a.k.a “2022: A Space Odyssey”)

Due: October 28, 2022 2:00PM

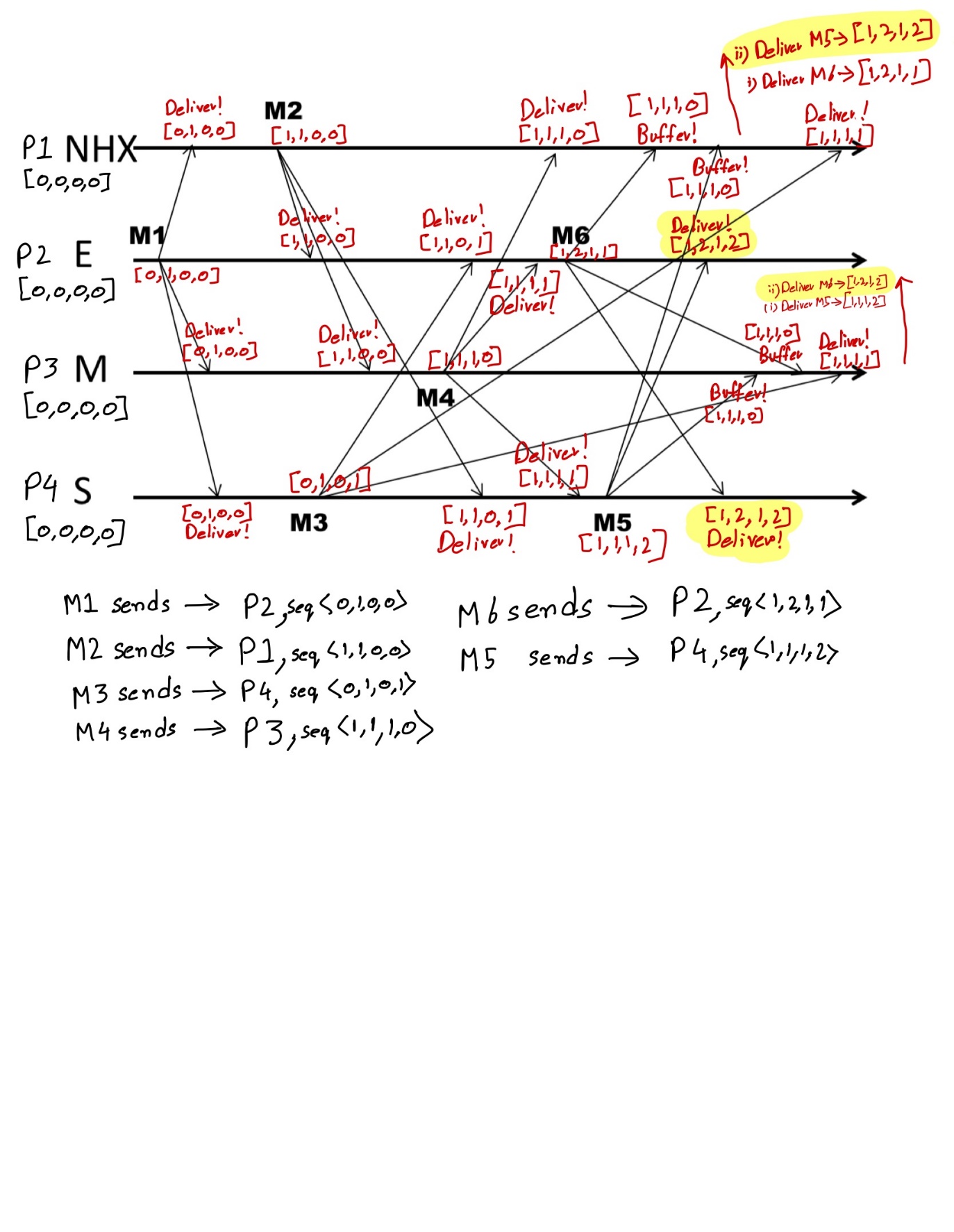
Mohan Amritesh Dasari

mdasari2

***Solution to Question 2:***



***Solution to Question 3:***



***Solution to Question 5: Diagram

Description automatically generated***

The snapshot calculation begins on Earth.

**Earth:**

The calculation begins at M(e), at this point, before the marker messages are sent, Earth’s process state is saved as S(a) (sending of a is the most recent state of the process). M(e) is sent to both ‘NHX’ and ‘Moon’ Processes. Earth then waits for both ‘NHX’ and ‘Moon’ to send it back a marker message and starts recording their respective channels.

Earth receiver the marker message M(N) from ‘NHX’ to stop recording of the channel. In this recording Earth receives R(c) from ‘NHX’ and stores/saves it as the state of channel ‘NHX-to-Earth’.

Next Earth receives the marker message M(M) from ‘Moon’ to stop recording of the channel. In this recording of the channel Earth receives R(b) and that is stored/saved as the state of the channel ‘Moon-to-Earth’.

Earth combines this Process State, and 2 Channel States makes a Snapshot.

**Moon:**

The calculation begins when Moon receives it’s first marker message M(e) from Earth. Once it receives this marker message, Moon’s process state is saved as S(e) (sending of e is the most recent state of the process). M(M) is sent to both ‘NHX’ and ‘Earth’ Processes. Moon then waits for ‘NHX’ to send a marker message back and starts recording this channel.

Moon receives a marker message M(N) from ‘NHX’ to stop recording of the channel. In this recording Moon receives R(f) from ‘NHX’ and stores/saves it as the state of channel ‘NHX-to-Moon’.

Since Moon starts the snapshot process after it receives a marker from Earth, it sets that channel to empty and saves/stores this state in as the Channel State for ‘Earth-to-Moon’.

Moon combines this Process State, and 2 Channel States makes a Snapshot.

**NHX:**

The calculation begins when NHX receives it’s first marker message M(e) from Earth. Once it receives this marker message, NHX’s process state is saved as S(f) (sending of f is the most recent state of the process). M(N) is sent to both ‘Moon’ and ‘Earth’ Processes. NHX then waits for ‘Moon’ to send a marker message back and starts recording this channel.

NHX receives a marker message M(M) from ‘Moon’ to stop recording of the channel. In this recording NHX receives R(e) from ‘Moon’ and stores/saves it as the state of channel ‘Moon-to-NHX’.

Since NHX starts the snapshot process after it receives a marker from Earth, it sets that channel to empty and saves/stores this state in as the Channel State for ‘Earth-to-NHX’.

NHX combines this Process State, and 2 Channel States makes a Snapshot.

***Solution to Question 6:***

*Part(i):*

One of the processes (say Pi) starts the leader election process and finds the highest DHT ID in its neighbours as well as compares it to its DHT ID. If it finds the highest DHT ID is one of the neighbours, it sends the Election Message to the neighbour with the highest DHT ID. When that process receives the election message, it does the same above-mentioned process again. This process checks for the highest DHT ID amongst its neighbours and itself and sends the election message to the highest DHT ID neighbour. This process continues until we find the node with the highest DHT ID neighbour. Once that happens, the node with the highest DHT ID will have all its neighbours DHT ID lower than its ID. Now once the highest node gets the election message, it finds the highest DHT ID neighbour that it has that is lower than its DHT ID value. This basically helps us find the second highest DHT ID in the system. Once that node is found, this highest DHT ID node sends an elected message to this node, which sets itself as the leader. Once the leader has been set it sends out a new leader message to its neighbours. The neighbours on receiving set this node as their leader node and send the same message to their neighbours. (Gossip-Style transmission).

*Part(ii):*

This algorithm satisfies Safety because once we have elected the node with the second highest DHT ID as the leader, this message is forwarded to all non-faulty nodes to set this node as their leader node. Once the elected leader is found to be errored or has left the network, the node will set its elected leader value as null and as the election message is going through the network, every time a node receives this message, it asks its neighbours to set their elected leader value as null as well, until a suitable leader is found. And once we find the node to be elected, the gossip style transmission basically sets the elected leader value to the proper leader, hence safety is not violated.

This algorithm satisfies Liveness because, once the node with the second highest DHT ID value receives the elected message, it sets itself as the leader and sends out a multicast message to all of its neighbours with this change, and those neighbours in turn keep multicasting the same message to all of their neighbours and in this way the message reaches all nodes in the network. Once this message is received by a node, that node sets the elected leader value to the proper value which is not null, hence Liveness is not violated.

*Part(iii):*

Number of messages in the leader election protocol:

Worst Case:

Finding Highest DHT ID:

(N/3) – 1 Messages

Every time a hop is made, it will skip 2 nodes as we have 3 successors and 3 predecessors. Removing one hop to ensure that the highest one doesn’t come under the neighbours of the farthest node from the node which starts the election process.

Electing Second Highest DHT ID:

1 Message

Once the Highest DHT Node receives an election message and determines that it has the highest DHT ID, it just looks at its Finger-Table entries and find the highest DHT ID amongst its neighbours and just sends that node an elected message.

New Leader Elected Messages:

6\*N Messages

Once the new leader is elected, it sends 6 messages to its neighbours to set the elected leader value and since it is gossip style, each node will send 6 messages to ask their neighbours to set the value. So Max number of messages sent in this stage will be 6\*N

Total Number of Messages sent:

6\*N + 1 + (N/3) – 1

=> 19N/3

Completion Time will be O(N)

*Note*: In retrospection, we did not necessarily need to employ gossip style message transmission. Instead, just finding the lowest DHT ID node amongst the neighbouring nodes and just sending the same message to the neighbours with DHT ID greater than the current node would have greatly reduced the number of messages sent to elect a new leader.

*Part(iv)*:

Whenever a message is broadcast, we wait for a reply from the receiving process. In case the process crashes or disconnects, the reply message timeouts and we start a new election process once the finger table entries are fixed. For example if the process with highest DHT ID doesn’t respond back in time, we reset and fix the current process’s finger table entries and restart the election process.

***Solution to Question 7:***

One major difference in this case will be that instead of searching for the highest ID for the coordinator, we’ll be searching for smallest IDs for electing coordinators.

When a process finds the coordinator has failed:

* If it knows its id is the smallest
  + It elects itself as one of the coordinators and sends out a coordinator message to all the processes with higher IDs. This message will contain a list of processes elected and the value of k i.e., the number of processes to be elected as coordinators.
* Else it initiates an election by sending an election message
  + Sends it to only processes that have a lower id than itself.
  + If received no answer within timeout, then it sets itself as one of the coordinators and sends out the above Coordinator message to all higher id processes.
  + If receives ‘OK’ then there is some non-faulty lower process <= so, wait for coordinator message. This process expects a Coordinator message from all the processes that reply with ‘OK’. If non received after another timeout, start a new election run.
* A Process that receives an Election message replies with ‘OK’ message and starts its own leader election protocol (unless it has already done so).
* On receiving a ‘Coordinator Message’, the process checks if it has received Coordinator messages from all process ID’s lower than itself
  + If yes, get the coordinator message with the largest length of the list of processes.
    - If len(list) < k, then add current node’s id to the list and send the modifies ‘Coordinator Message’ to all higher id processes.
    - If len(list) == k, then do not modify the list and send the same ‘Coordinator Message’ to all processes.
    - Lower Id processes on receiving this, just set the coordinator list to the received list.
  + If no, wait till you receive all coordinator messages from all lower id processes.
  + If timeout, start a new election process.

Current modifications to the Bully Algorithm, satisfies Safety as if we do not have k processes in the list of coordinators, we will end up restarting the election and when that happens, we will set the list of coordinators to null. Hence at any point in time, we will either have k processes as coordinators or NULL as the value of coordinators.

This modified algorithm also satisfies Liveness, as if any process timeouts at any given step, we restart the election process and once the election process finishes without any timeouts, we have all processes with the list of elected coordinators which will be not null.

***Solution to Question 8:***

*Safety*: At most one process executes in Critical Section at any time

*Liveness*: Every request for a Critical Section is granted eventually

*Causal Ordering*: requests are granted in the order they were made

**Safety**: The change that instead of using the original (Lamport Timestamp, process id), we are using (Lamport Timestamp, FIFO local Sequence). By using this, instead of having one process id per message, we’ll be sending a list of processes to get individual access to the said resource. This change in no way violates safety as the head of the local FIFO sequence will still be the original process ID. Once the original process has completed its access of the said resource, it gives it to the next head of the FIFO queue. The only issue could be that there could be multiple processes with the same process requesting access, leading to duplication of same process requests.

**Liveness**: Since we are sending the FIFO sequence, eventually all the processes in the list will get mutual exclusion access. Hence liveness is also not violated. Need to add here that if one of the processes goes into a deadlock with the access, a timer terminates its access and gives it to the next process in the queue.

**Causal Ordering**: Causal Ordering will be violated, as the FIFO sequence is the local FIFO Sequence of the process requesting access, which may not always be in the causal ordering. From the lectures, we know that Causal Ordering ensures that FIFO is maintained but FIFO does not necessarily mean that Causal Ordering is maintained. Hence this algorithm violates causal ordering.

One thing to note in this algorithm is that since multiple processes receive request from every process (say pi) requesting access which in turn leads to multiple processes having the same request in their FIFO queues. Even if one of the processes decides to give access to process pi to the resource, the other processes with the same request from pi are not updated regarding this and thus end up giving duplicate or redundant access to process pi.

***Solution to Question 9:***

*Part(a)*: **Incorrect**

Does not satisfy virtual synchrony as Virtual Synchrony states that a multicast message sent in a particular view should be delivered to all processes in the same view, which does not happen in this case. P1 multicast message in view V11 but it is being delivered in V12, which is incorrect.

*Part(b)*: **Correct**

This satisfies virtual synchrony as Virtual Synchrony does not track undelivered multicasts. In other word if a process sends a multicast message and it is not delivered to any process then it is simply ignored, hence the next view does not break virtual synchrony.

*Part(c)*: **Incorrect**

Does not satisfy virtual synchrony for the same reason as part(a). p2 violates this by delivering M32 in view V12. Virtual Synchrony states that either deliver that message in the same view or do not deliver the multicast message at all.

*Part(d)*: **Incorrect**

Does not satisfy virtual synchrony as M32 should have been delivered at p1 in the same view V11 as the broadcast in in. Same goes for M45 for p2, this multicast should have been delivered at p2 in the same view V11. This would help in maintaining virtual synchrony.

*Part(e)*: **Correct**

Provided that there is no multicast message between p3 delivering the view V12 and p1 and p2 delivering the same view. Since both Multicast messages M32 and M45 are delivered in the same view as they were broadcast in. This does not violate Virtual Synchrony.

*Part(f)*: **Incorrect**

Since the process from which the multicast M32 is sent crashes, it doesn’t matter if this multicast is delivered or not at p2 and p3. Since process p1 crashed, the new view delivered by p2 and p3 should have been {p2,p3}, but instead it is {p2} and {p3} at p2 and p3 respectively. Hence this violates virtual synchrony.

*Part(g)*: **Incorrect**

Since all processes changed view before receiving the multicast message M32, this violates virtual synchrony. Ideally, M32 should have been delivered to p1, p2 and p3. Then when p4 joined update views of all processes to {p1,p2,p3,p4} and continue with new multicasts.

***Solution to Question 10:***

*Part(i)*:

Diagram

Description automatically generated

Earth is the initiator of the snapshot. Imagine the blue lines as Marker Messages.

Earth to NHX -> M4

Earth to Moon -> M5

*Earth*:

* At M1:
  + Records its own state
  + Starts Recording Channel Moon-to-Earth
  + Starts Recording Channel NHX-to-Earth
* On Receiving M2 from NHX:
  + Do nothing as this is not the last marker
* On Receiving M3 from Moon:
  + Since this is the (N-1)th Marker, we record the final states of all channels
  + Update Channel State of NHX-to-Earth as <R(c), R(h)>
  + Update Channel State of Moon-to-Earth as <>

*NHX*:

* At M2:
  + Records its own state
  + Starts Recording Channel Moon-to-NHX
  + Starts Recording Channel Earth-to-NHX after setting it as empty
* On Receiving M3 from Moon:
  + Do nothing as this is not the (N-1)th Marker
* On Receiving M4 from Earth:
  + Since this is the (N-1)th Marker, we record the final states of all channels
  + Update Channel State of Earth-to-NHX as <>
  + Update Channel State of Moon-to-NHX as <>

*Moon*:

* At M3:
  + Records its own state
  + Starts Recording Channel NHX-to-Moon
  + Starts Recording Channel Earth-to-Moon after setting it as empty
* On Receiving M2 from Moon:
  + Do nothing as this is not the (N-1)th Marker
* On Receiving M5 from Earth:
  + Since this is the (N-1)th Marker, we record the final states of all channels
  + Update Channel State of Earth-to-Moon as <R(d)>
  + Update Channel State of NHX-to-Moon as <R(f), R(g)>

I do not believe that this algorithm is correct. Instead of recording the state of the communication channel when it receives the marker back, it waits for all the markers to return and then it saves that state, which is not how it should be. This makes the return marker on most channels not a useful message as it is just used to maintain counter. This is shown in the above example, At Earth after receiving the (N-1)th Marker, we can notice how we have also recorded receiving h as a state in the Channel NHX-to-Earth. This method also adds additional delay to the snapshot process, during which the chances of the process crashing increases.

*Part(ii)*:

To fix this algorithm, instead of waiting for (N-1) markers to reach the initial process, we should make intermediate snapshots every time we receive a marker. Intermediate snapshots as in just record the state of the communication channels as soon as we receive the return marker through that channel. Now even in case the initial process crashes in the middle of snapshot creation, we still have a fallback backup of some states and communication channel states to understand where it crashed rather than losing the entire thing.