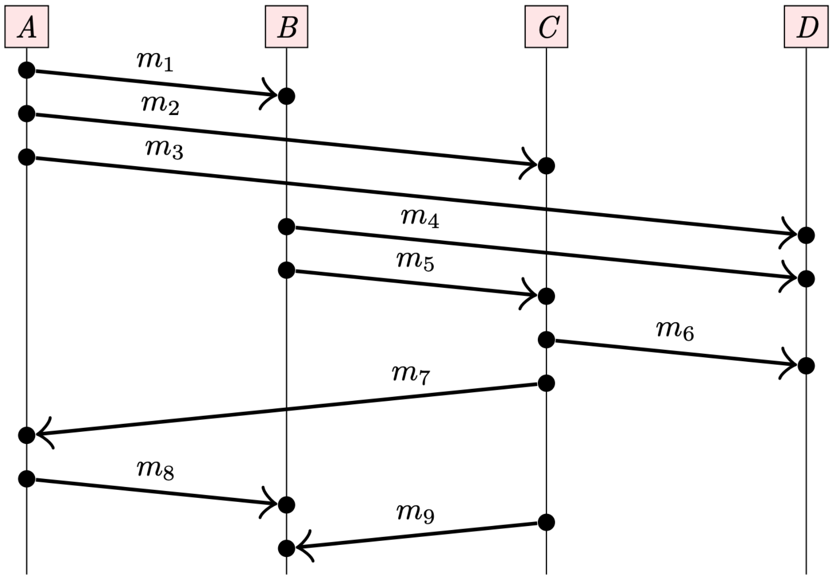
yz709-CDS-sup4

Exercise 13(optional)

- (Exercise 10) Given the sequence of messages in the following execution, show the Lamport timestamps at each send or receive an event.
- (Exercise 12) Given the same sequence of messages as in Exercise 10, show the vector clocks at each send or receive an event.

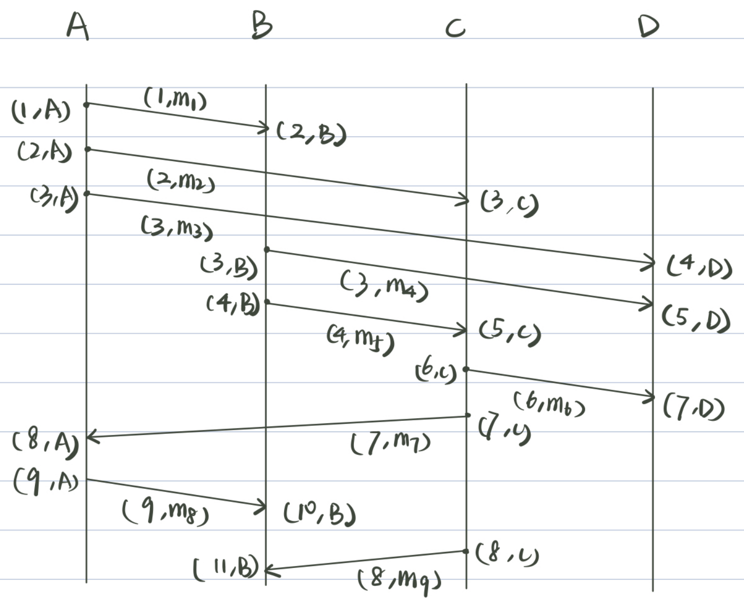


Using the Lamport and vector timestamps calculated in Exercise 10 and 12, state whether or not the following events can be determined to have a happens-before relationship.

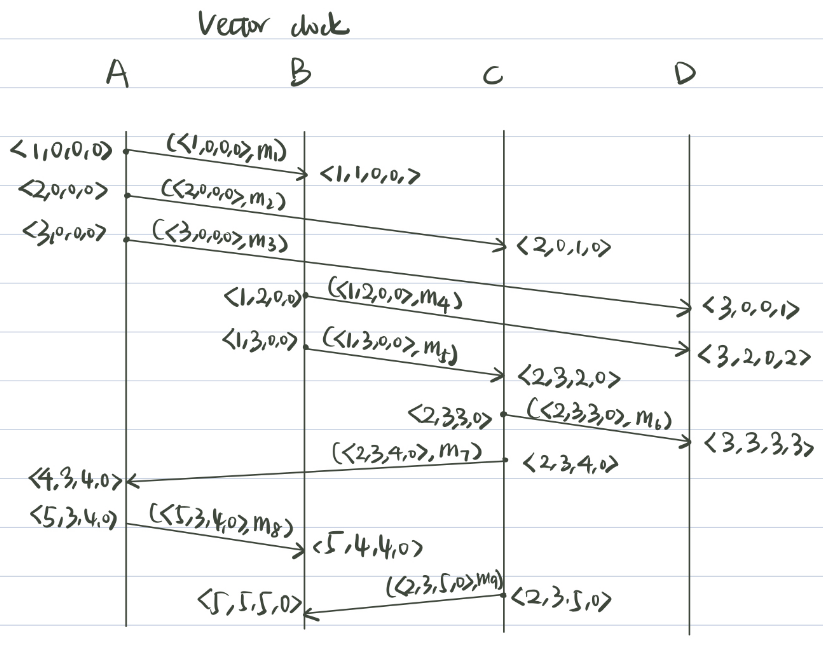
Events		*Lamport*	*Vector*
send(m_2)	send(m_3)		
send(m_3)	send(m_5)		
send(m_5)	send(m_9)		

Ex10

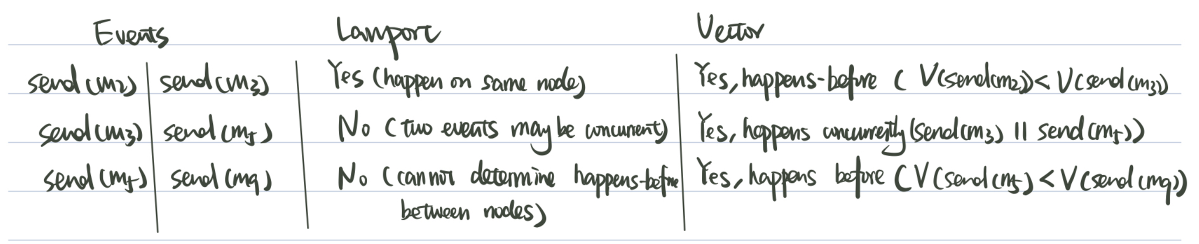
Lamport clock



EX12



EX13



Events		Lamport	Vector
send(m2)	send(m3)	Yes (happen on same node)	Yes, happens-before (V(send(m2))< V(send(m3))
send(m3)	send(m5)	No (two events may be concurrent)	Yes, happens concurrently (send(m3) ∥ send(m5))
send(m5)	send(m9)	No (cannot determine happens-before between nodes)	Yes, happens before (V(send(m5)) < V(send(m9))

💡 Comments:

Exercise 14(optional)

We have seen several types of physical clocks (time-of-day clocks with NTP, monotonic clocks) and logical clocks. For each of the following uses of time, explain which type of clock is the most appropriate: process scheduling; I/O; distributed filesystem consistency; cryptographic certificate validity; concurrent database updates.

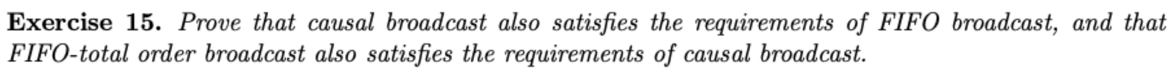
- Process scheduling:
  - Monotonic clocks are sufficient for local process scheduling.

    - Process scheduling happens in the operating system using the process scheduler. It has no interactions with the peripherals; we only need to use a system clock, which reports the time elapsed since the start of the system boot-up.
- I/O:
    - Monotonic clocks are sufficient for local I/O.
    - I/O devices are pieces of hardware used by a human or another system to communicate with a computer; they receive data packets or send data packets (i.e., keyboards, printers, headphones and touch screens)
    - We could use the monotonic clock to manage the data packets' order and use FIFO buffers to store data packets; there is no need to synchronise the time between computers and I/O devices because only the order of events and data packets matters.
- Distributed filesystem consistency:
    - A vector clock is sufficient for maintaining distributed filesystem consistency.
    - Event messages of updating, storing and deleting data on the filesystem are sent from distributed clients, so preserving the causality of events and the correct order is essential to ensure the consistency of the data stored. Only logical clocks are consistent with causal dependencies, and within them, vector clocks could distinguish between concurrent events and happens-before relationships.
- Cryptographic certificate validity:
    - A time-of-day clock with NTP is sufficient for cryptographic certificate validity.
    - Each ticket will have a valid period, and we usually have distributed servers that check the validity of the tickets; hence need to make sure the server clocks are synchronised. The order of the tickets doesn't matter because they are all independent of each other.
- Concurrent database updates:
    - A vector clock is sufficient for concurrent database updates.
    - Vector clock can distinguish between concurrent events from happens-before related events. In contrast, a Lamport clock would maintain a total ordering and cannot tell concurrent events from happens-before related events. So for concurrent database updates, we can push all conflict operations and retain

all possible values; the client needs to merge the conflicting operations and avoid data loss.

💡 Comments:

Exercise 15


Exercise 15. *Prove that causal broadcast also satisfies the requirements of FIFO broadcast, and that FIFO-total order broadcast also satisfies the requirements of causal broadcast.*

- Causal broadcast satisfies the requirements of FIFO broadcast:
    - Causal broadcast ensures a causal order; FIFO broadcast ensures messages broadcasted by the same node will be delivered in the same order.
    - If m_1 and m_2 are delivered from the same node, then they have a happens-before relation, hence a causality. Therefore, causal broadcast ensures messages broadcasted by the same node will be delivered in the same order, which is the requirement of FIFO broadcast.
- FIFO-total order broadcast satisfies the requirements of causal broadcast:
    - FIFO-total order broadcast ensures the delivery order on all nodes is the same. The order is a FIFO ordering such that messages broadcasted by the same node would be delivered in the same consistent order.
    - Since causal broadcast ensures causality:
        - If m_1 and m_2 are delivered from the same node, then since the causality is preserved on that node, on any other nodes, these two messages would be delivered in the same order.
        - If m_1 is delivered from node A, m_2 is delivered from node B, where we have $m_1 \rightarrow m_2$ due to transitivity. There must be an event where node B receives m_1 and then, later broadcasted m_2; if this is the case on node B, then it must be true for all nodes, thanks to total ordering.

💡 Comments:

Exercise 16

Give pseudocode for an algorithm that implements FIFO-total order broadcast using Lamport clocks. You may assume that each node has a unique ID and that the set of all node IDs is known. Further, assume that the underlying network provides reliable FIFO broadcast.

- Assume all nodes deliver at least one message; we start forming a total ordering of messages until all nodes deliver at least one message.

```
i:=0 # unique id generator

on initialisation do
  sendSeq:=0 # number of messages broadcasted by the node
  lampClock:=0 # local lamport clock

# one entry per node, counting the number of messages each sender has delivered
  delivered:=<0,0,...,0>
  id:=i # generate a unique id
  i:=i+1
  buffer:={} # holding back messages until they are ready to be delivered

# Map: record the latest Message this node has sent
  latestMessage[id]:=(lampClock,id)
end on

on request to broadcast m at node i do
  lampClock:=lampClock+1
  send(i, sendSeq, m, lampClock, i) via reliable broadcast
  sendSeq:=sendSeq+1
end on

on receiving (msg, msgLampClock, senderId) from reliable broadcast at node Ni do
  lampClock = max(lampClock, msgLamportClock)+1 # update local lamport clock
  buffer:=buffer.append((msgLampClock, senderId, msg)) # holding back message

  # get the latest message send across all nodes
  # only make progress when all nodes sent at least one message
  latestAllMessage:=(infinity,nodes[-1])
  for nid in nodes do
    latestAllMessage:=min(latestMessage[nid],latestAllMessage)
  end for

  # deliver all messages before latestAllMessage in total order
  while (!buffer.isempty()) do
    # get the minimum each time
    (msgLampClock, senderId, msg) = buffer.getMin()
    # if later than latestAllMessage, then break the loop
    if ((msgLamClock, senderId) > latestAllMessage) break
    # otherwise deliver the message to the application
    deliver msg to the application
    delivered[sender]:=delivered[sender] + 1
    buffer.remove((msgLampClock, senderId, msg))
  end while
end on
```

💡 Comments:

Exercise 17(optional)

Apache Cassandra, a widely-used distributed database, uses a replication approach similar to the one described here. However, it uses physical timestamps instead of logical timestamps, as discussed here. Write a critique of this blog post. What do you think of its arguments and why? What facts are missing from it? What recommendation would you make to someone considering using Cassandra?

- Arguments:
    - Cassandra breaks a row up into columns that can be updated independently, so we only need to deal with concurrent changes to a single field, not concurrent changes to a single record. This fine-grained approach helps reduce the number of conflicts because concurrent changes to different fields can be updated separately without data loss.
    - Clock synchronisation is nice to have in a Cassandra cluster but not critical because timestamps are only used to pick a winning update within a single field, not a record.
- Missing facts:
    - Cassandra still uses the "last write wins" strategy to resolve conflicts and determine which mutations represent the most up-to-date state of a single field. Hence we still need to figure out the order of update operations to a single field because when a read request occurs, Cassandra will pick the fields with the most recent timestamp if it sees multiple cells representing the same column.
- Recommendation:

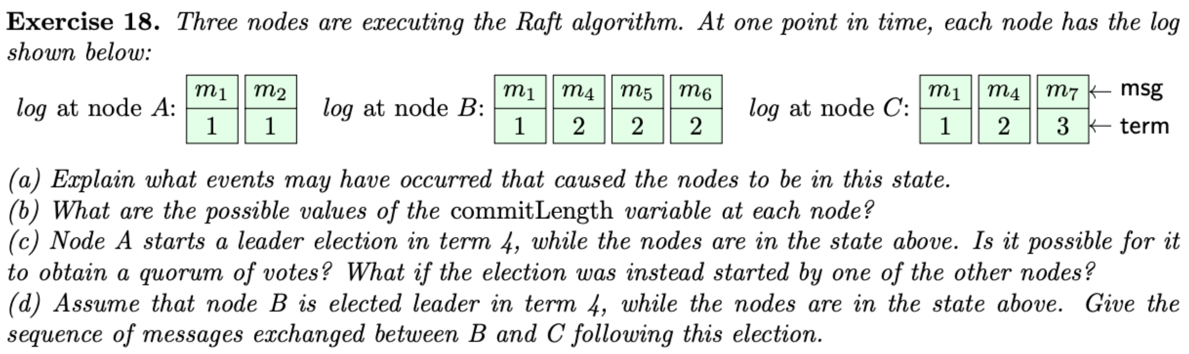
- All clocks of Cassandra cluster nodes have to be synchronised to prevent READ old data values. Because when nodes' clocks become out of sync, if you do multiple mutations to the same column and different coordinators are assigned, you can create some situations where writes that happened in the past are returned instead of the most recent one.

```
node A at time t0
node B at time t100

1. delete P from table where Q=11
# node B coordinates the request, and it is assigned timestamp t101
2. update table set P=20 where Q=11
# node A coordinates the request, and it is assigned timestamp t1
3. select P from table where Q=11
# since t101 > t1, no results is returned
```

Comments:

Exercise 18



Exercise 18. *Three nodes are executing the Raft algorithm. At one point in time, each node has the log shown below:*

log at node A:	m_1	m_2				
	1	1				
log at node B:	m_1	m_4	m_5	m_6		
	1	2	2	2		
log at node C:	m_1	m_4	m_7	← msg		
	1	2	3	← term		

(a) Explain what events may have occurred that caused the nodes to be in this state.
(b) What are the possible values of the commitLength *variable at each node?*
(c) Node A starts a leader election in term 4, while the nodes are in the state above. Is it possible for it to obtain a quorum of votes? What if the election was instead started by one of the other nodes?
(d) Assume that node B is elected leader in term 4, while the nodes are in the state above. Give the sequence of messages exchanged between B and C following this election.

- (a):
  - Node A is initially a leader, so it broadcasted m_1 and m_2 in term 1.
  - When broadcasting m_2, node A has no access to the other two nodes, node B suspects node A is not alive, so node B becomes a candidate and starts a new term 2. After getting the vote from node C, B receives a quorum of votes to become the leader in term 2.
  - When broadcasting m_5, node B has no access to node C, so after some heartbeat timeout, node C suspects node B is not alive, node C becomes

starts a new leader election in term 3. But at the same time, node B still think it is a leader and try to broadcast m_6 although no other nodes received it.

- (b):

```
commitLength[A] = 1 // m1
commitLength[B] = 2 // m1 & m4
commitLength[C] = 2 // m1 & m4
```

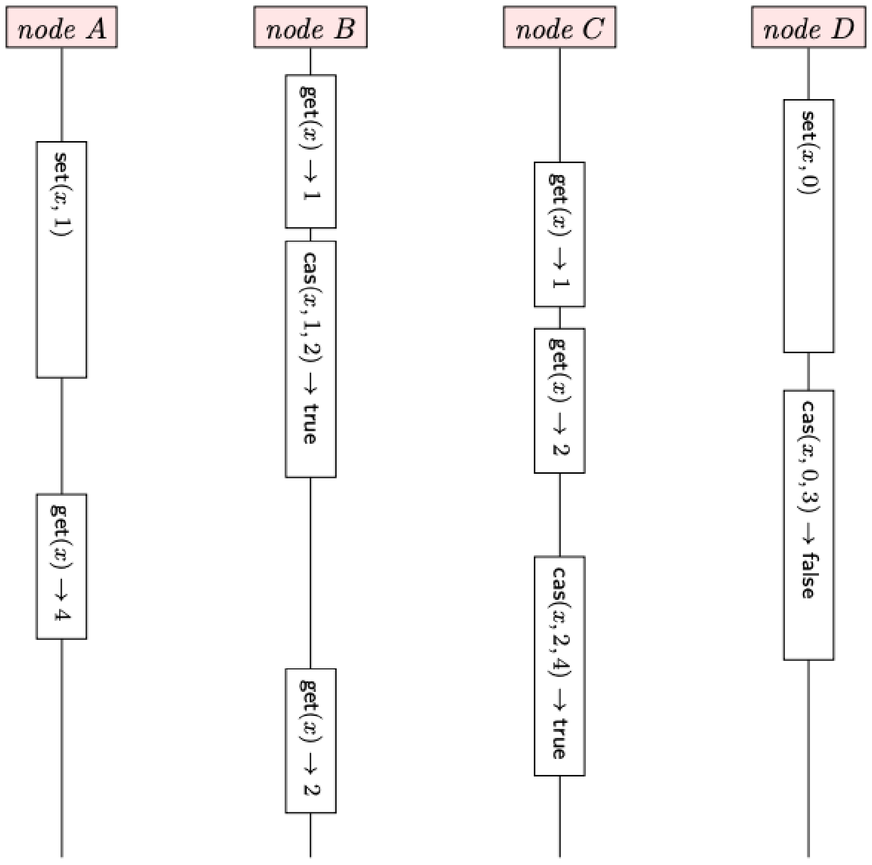
- (c):
  - From the perspective of node A, a quorum is two votes out of three.
  - If node A is isolated from node B and node C, it cannot get a quorum of votes because it cannot transmit messages to other nodes. However, if node A recovers and can send messages to node B and node C, then after seeing a higher term than themselves, node B turns from a leader into a follower, node C transfers from a candidate into a follower, hence it is possible to get ≥ 2 votes and form a quorum.
  - From the perspective of node B, node A is not alive, so forming a quorum requires a vote from node C, so it is possible to form a quorum.
  - From the perspective of node C, node A and node B are both not alive, so there is no way to form a quorum of 2 out of 3 votes.
- (d):
  - After electing to be a leader, node B still needs to acknowledge a quorum before delivering messages.
  - Node B needs to send acknowledge of proposing sending m, and if node C responds with granted, then node B can actually deliver m and broadcast this fact to the followers, in this case, node C , so that they can do the same. Otherwise, node C refused, which indicates that a later leader has been elected, node B needs to step back to a follower.

```
node B -> node C: can we deliver message m next in term 4
node C -> node B: okay
node B -> node C: deliver m to applications (commit)
```

💡 Comments:

Exercise 20

Exercise 20. *Is the following execution linearizable? If not, where does the violation occur?*



- No. The `get(x) -> 4` operation occurs earlier than `get(x) -> 2`, the atomic compare and swap `cas(x, 2, 4) -> true` occurs somewhere in between, and turn x from 2 into 4; hence there will be only three possible cases: both get 4, both get 2 or first get 2 and second get 4. Therefore, it is not possible that an earlier operation gets 4 and a later operation gets 2.

💡 Comments: