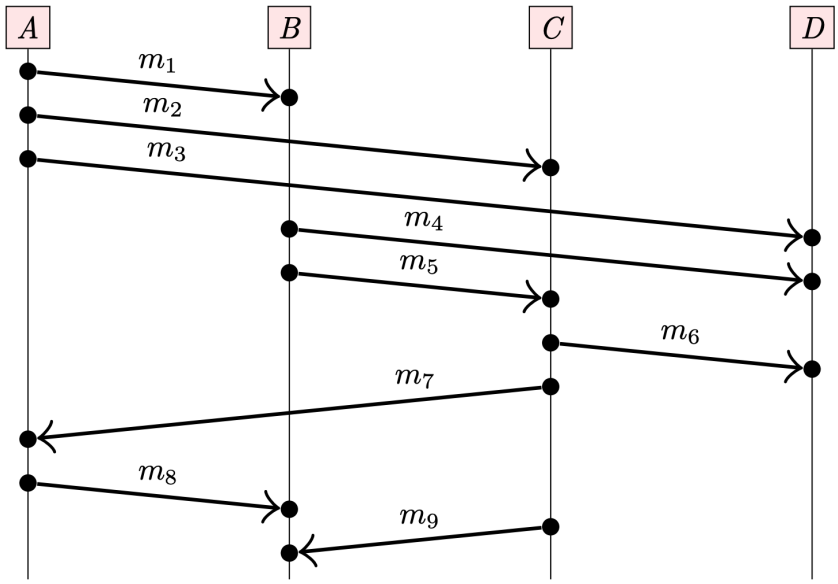
yz709-CDS-sup4

Exercise 13

- (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.

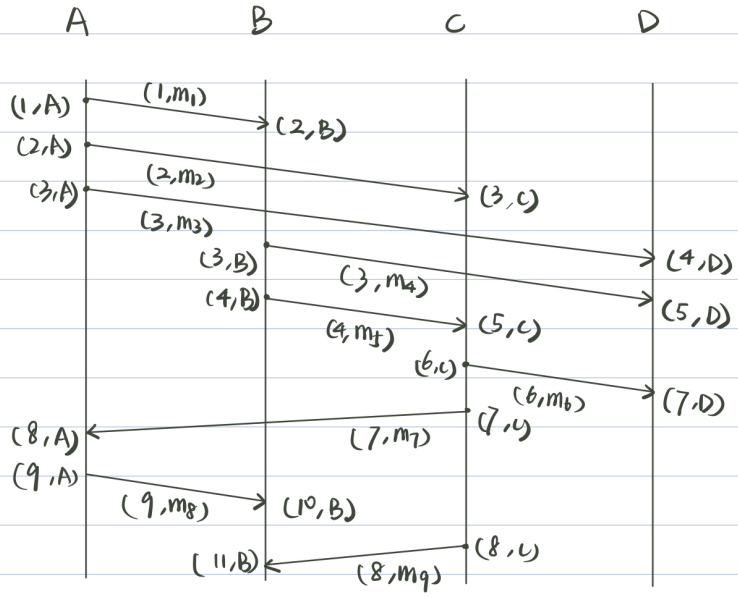


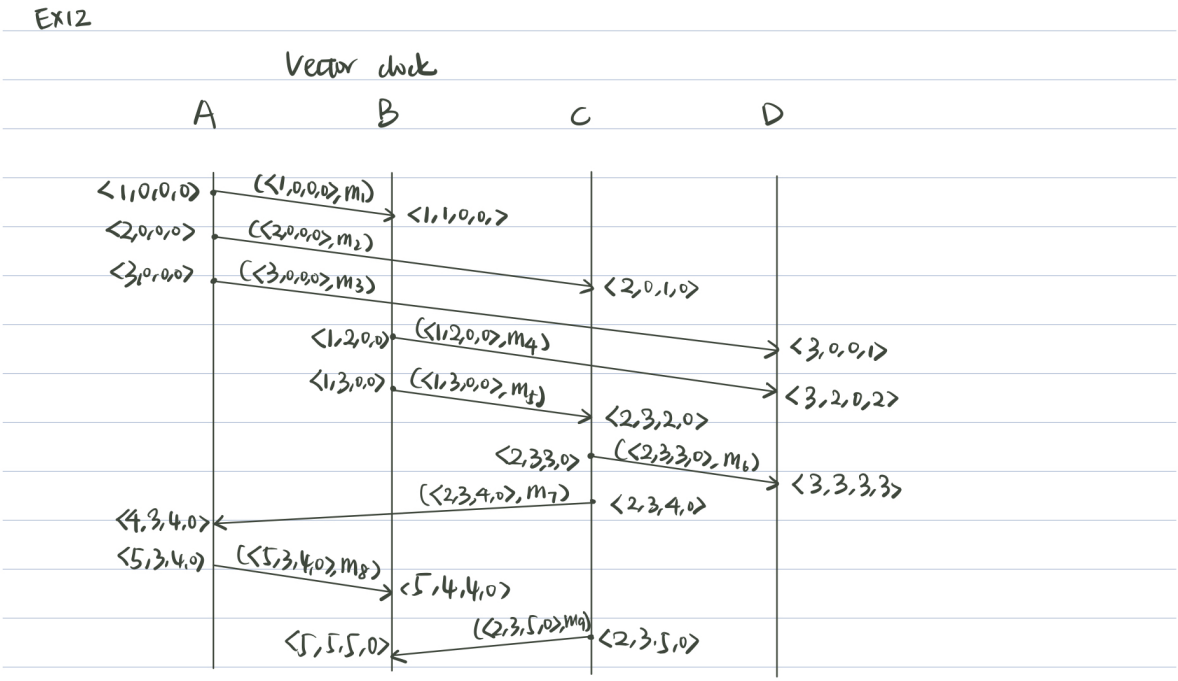
Using the Lamport and vector timestamps calculated in Exercise 10 and 12, state whether or not the following events can be determined to have a happens-before relationship.

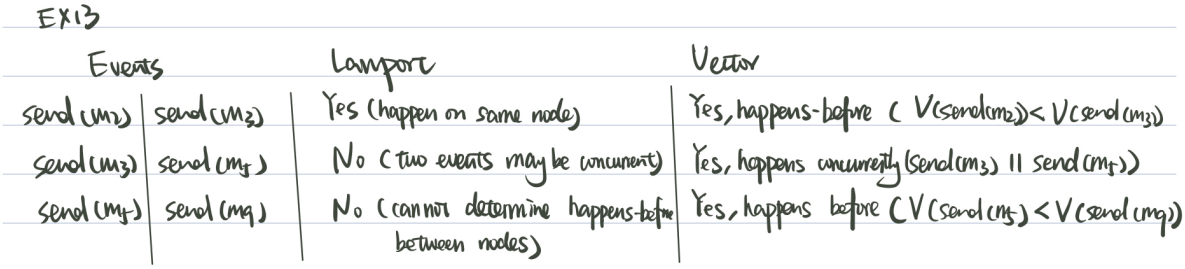
Events		*Lamport*	*Vector*
send(m_2)	send(m_3)		
send(m_3)	send(m_5)		
send(m_5)	send(m_9)		

Ex 10

Lamport clock







💡 Comments:

Exercise 14

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 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

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: