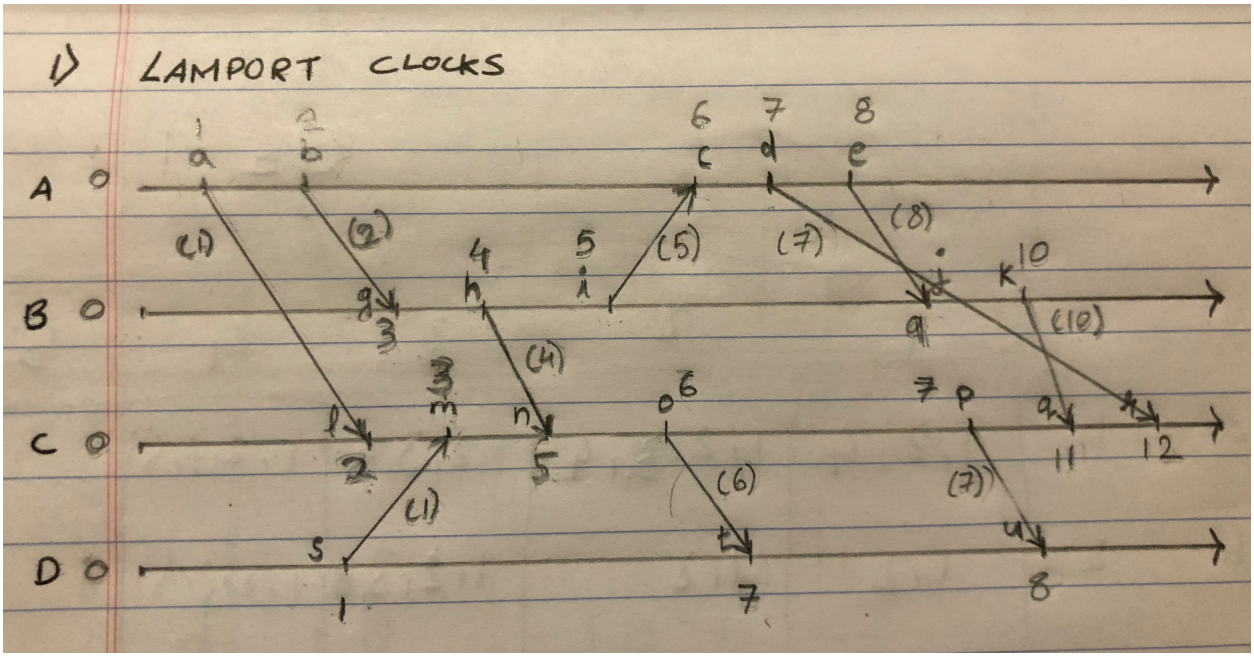
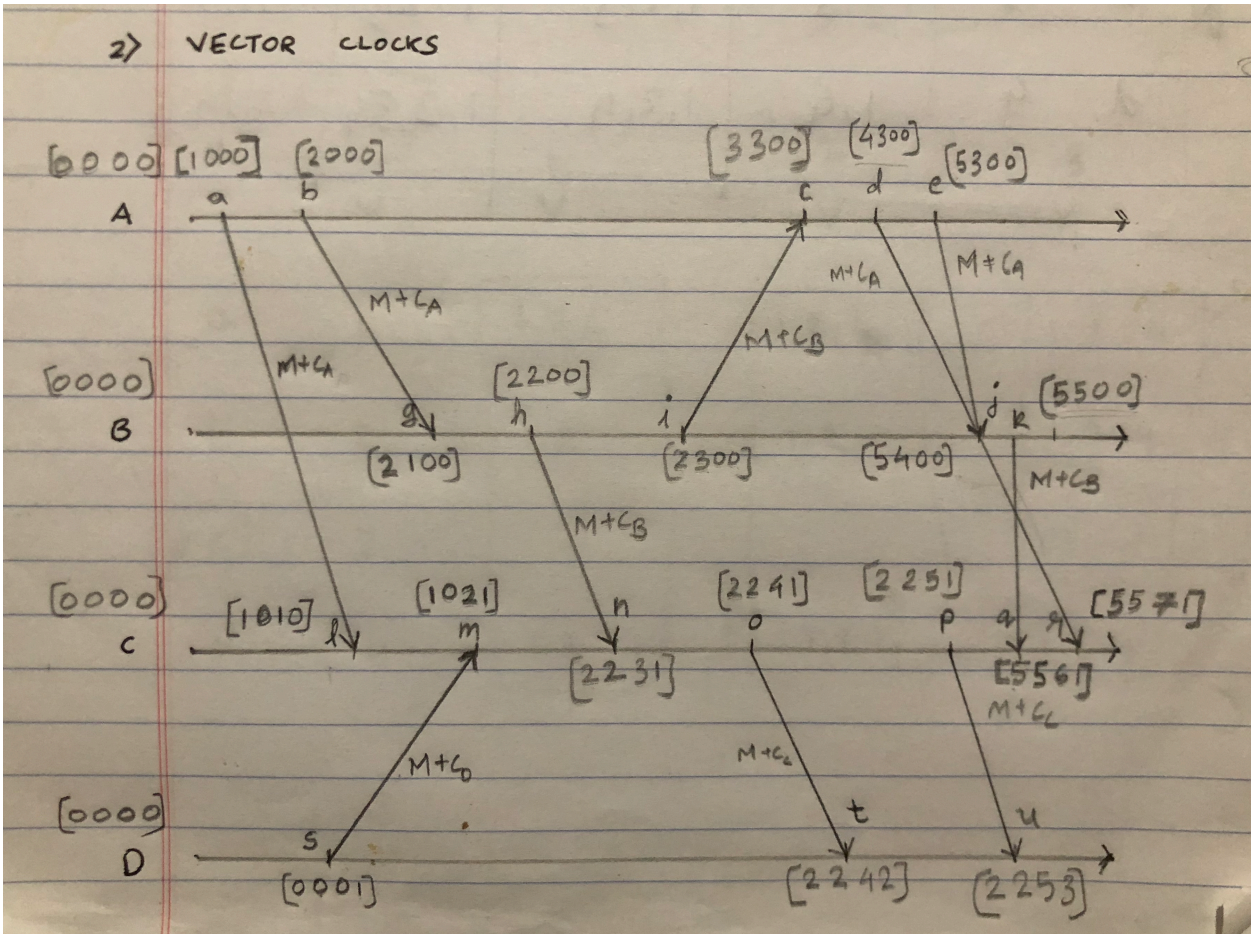
Problem 1:

a. **Lamport** Clocks



Event	Clock value	Event	Clock value	Event	Clock value	Event	Clock value
A	1	G	3	L	2	Q	11
B	2	H	4	M	3	R	12
C	6	I	5	N	5	S	1
D	7	J	9	O	6	T	7
E	8	K	10	P	7	U	8

b. **Vector** Clocks



Event	Clock value	Event	Clock value	Event	Clock value	Event	Clock value
A	1000	G	2100	L	1010	Q	5561
B	2000	H	2200	M	1021	R	5571
C	3300	I	2300	N	2231	S	0001
D	4300	J	5400	O	2241	T	2242
E	5300	K	5500	P	2251	U	2253

Problem 2: Prove that the Lamport clocks algorithm presented in class provides a total order.
Solution:

To prove that Lamport clocks algorithm provides total order, the relation R should follow the properties-

- *a* R *b* AND *b* R *a* => *a* = *b* (anti-symmetry)
- *a* R *b* AND *b* R *c* => *a* R *c* (transitivity)
- *a* R *b* OR *b* R *a* (connex property)

The Lamport clocks algorithm uses "happened before" (->) relation to provide ordering of events.
Let $C_i(x)$ be the function that provides the clock value of process p_i at event *x*.

For a, b $\in$ p_i anti-symmetry, transitivity and connex property (since p_i increments the clock after every event) are trivially true.

For a $\in$ p_i and b $\in$ p_j,

1. The anti-symmetry property is trivially true.
2. If a -> b, then $C_i(a) < C_j(b)$….by definition of "happened before" relation. Similarly, If b -> c, then $C_j(b) < C_k(c)$. Hence, we can say that $C_i(a) < C_k(c)$. Therefore, the transitivity property holds for Lamport clocks.
3. The only ambiguity left in Lamport clocks is for the events that belong to different processes have the same clock value. This ambiguity is resolved by comparing the processes' p_i and p_j ids and ordering them based on the id values. Hence, the relation always follows the connex property.

From 1, 2 and 3 we conclude that Lamport clocks follow the anti-symmetry, transitivity and connex property. Hence, **the Lamport clocks provide total ordering**.

Problem 3: Prove that the Vector clocks algorithm presented in class provides a partial order.
Solution:

To prove that Vector clocks algorithm provides strict partial order, the relation R should follow the properties-

- not *a* R *a* (irreflexivity)
- *a* R *b* AND *b* R *c* => *a* R *c* (transitivity)
- *a* R *b* AND not *b* R *a*(anti-symmetry)

The Vector clocks algorithm uses "happened before" (->) relation to provide ordering of events.
Let C(x) be the function that provides the clock value at event *x*.
Let $C_i(x)$ be the function that provides the clock value of process p_i at event *x*.

1. Irreflexivity property is trivially true for "happened before".

2. If a -> b, then $C_i(a) < C_j(b)$….by definition of "happened before" relation. Similarly, If b -> c, then $C_j(b) < C_k(c)$. Hence, we can say that $C_i(a) < C_k(c)$. Therefore, the transitivity property holds for Vector clocks.
3. If a -> b, then $C_i(a) < C_j(b)$….by definition of "happened before" relation. Therefore, $C_j(b) < C_i(a)$ cannot be true. Hence, not b -> a holds. Thus, anti-symmetry is satisfied.

From 1, 2 and 3 we conclude that Vector clocks follow the anti-symmetry, transitivity and irreflexivity property. Hence, **the Vector clocks provide partial ordering**.

Problem 4: Consider the Chandy-Lamport Snapshot algorithm presented in class, now also assume that communication is not FIFO anymore. Propose an algorithm to record a global state when communication is not FIFO. You can use vector clocks.

Solution:

In a non-FIFO system, a *marker* (i.e. used by Chandy-Lamport algorithm) cannot be used to separate messages into that go into the recorded state and those that do not. Mattern's algorithm solves the problem by assuming a single initiator process. We will use vector clocks for a consistent global snapshot of the system.

Algorithm:

1. The initiator selects a future vector time *t* at which a global snapshot of the system is to be recorded. This *t* is broadcasted, and initiator waits for an acknowledgment for this message. Since, we assumed no crashes in the system, all the processes receive the time *t.*
2. After receiving the broadcast every process stores *t* and sends an ACK to the initiator.
3. After receiving the ACK, the initiator increments its clock to *t* and broadcasts a dummy message.
4. After receiving the dummy message each process increases its clock value to >= *t.*
5. Each process *i* then takes a local snapshot LS_i and sends it to the initiator.
6. All messages sent along channel C_{ij} whose timestamp is smaller than *t* and are received by process p_j after recording LS_j

Correctness:

Terms:
LS_i – Local snapshot of process p_i
SC_{ij} – State of channel C_{ij} between process p_i and p_j
M_{ij} – message passing from p_i to p_j
$send(m_{ij})$ and $rec(m_{ij})$ are the send and receive events.
A global state is consistent if it follows conditions of the consistent cuts:
C1: $send(m_{ij}) \in LS_i$ then, $m_{ij} \in SC_{ij}$ XOR $rec(m_{ij}) \in LS_j$
C2: $send(m_{ij}) \notin LS_i$, then $m_{ij} \notin SC_{ij}$ AND $rec(m_{ij}) \notin LS_j$

….(ref. Distributed Computing: Principles, Algorithms and Systems: K. Ajay and S. Mukesh)

1. After recording a local snapshot LSi, any message mij sent by process pi has a timestamp > t. Assume that process pj receives this message before recording LSj. Hence, pj's clock will read a value > t. Thus, this t could not have been chosen by the initiator to record the snapshot.
2. This implies pj must have already recorded snapshot which contradicts the assumption. Therefore, mij cannot be received by pj before recording LSj. Thus, C2 is satisfied.
3. By rule 6, Each message mij with a timestamp < t is included in LSj if pj receives mij before taking snapshot. Else mij is included in SCij. Thus, C1 is also satisfied.

Hence the algorithm is correct.

Problem 5: Consider the consensus algorithm in synchronous systems with crash failures presented in class and assume that a message can sometimes be delayed and arrive in the next round rather than only in the round it was sent. Show how can you modify the algorithm presented in class and prove that it is correct.

Solution:

A distributed system with *n* processes and tolerates at most *f* failures such that $f < n$. Since the messages can be sometimes delayed and arrive in next round, we have to consider the worst-case scenario. That is, assume there are *f* failures and each message is delayed and arrives in the next round. Therefore, $f + 2$ rounds are required to achieve at consensus.

Algorithm:

```
V := set of values initially {vi};
for k := 1 to f + 2 do
    receive Sj from all processes Pj, j != i       // for delayed responses
    send {v ∈ V | Pi has not already sent v} to all;
    receive Sj from all processes Pj, j != i
    V : V U Sj
endfor

y := min(V)
```

Description:

Consider 5 processes {p1, p2, p3, p4, p5} with 2 failures.

a. After round 1 – p1 knows about p2, p3 and let the response from p5 arrive in next round and suppose p4 crashes. Only p1 has value from p4.
b. After round 2 – p1 gets the value from p5(round 1) and p2(round 2) and suppose p3 crashes. Still it is not 100% sure that all processes have value from p3.
c. After round 3 – p1 gets the values from p5(round 2) and p2(round 3). Still the value from p5(round 3) is yet to be received.
d. After round 4 – p1 gets the values from p5(round 3) and all the processes have same value.

Correctness:

a. Assume by contradiction that after f + 2 rounds 2 processes arrive at different values. This means that they do not have identical final sets after f + 2 rounds. This cannot be because of crashes as crash failures are handled in f + 1 rounds. So, we have to only prove for delay.
b. This can only be the case when a new value v was sent in round f + 2. This implies that a process failed in round f + 1 and hence after round f + 2, the sets are not identical with the processes.
c. That means there are f + 1 failures. But this contradicts our assumption that there are f failures.

Hence, f + 2 rounds are required at minimum to arrive at a consensus.