SSN COLLEGE OF ENGINEERING, KALAVAKKAM

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

UCS1701 – Distributed Systems

Assignment 1

Name: Jayannthan P T *Dept: CSE 'A'* *Roll No.: 205001049*

DESIGN OF VECTOR CLOCKS FOR AD-HOC DISTRIBUTED SYSTEMS

Consider a modern ad-hoc distributed system like vehicular or mobile networks. As the characteristics of ad-hoc systems varies from the conventional distributed systems, the design challenges of ad-hoc systems should be studied for renovating the insight towards key technological shift.

a. **Adapt the conventional representation of vector clocks to suit the ad-hoc distributed system. (10 Marks)**
b. **Formulate the implementation rules for the adapted design of vector clock synchronization concerning the design challenges of the ad-hoc distributed system. (20 Marks)**
c. **Elaborate the working of the adapted design through a relevant example. (30 Marks)**

Discuss the answers in detail.

i. **Highlight the abstract difference between a conventional distributed system and an ad- hoc distributed system. [5]**

Conventional Distributed Systems	Ad Hoc Distributed Systems
Static with permanent link	Dynamic with frequent node joins/leaves
Reliable	Unreliable, nodes may leave before message delivery
Nodes know participating processes	Nodes lack knowledge of participants
Easier with snapshots	Difficult due to dynamic nature
Relatively easy	Identifying and mitigating faults can be cumbersome

ii. **Explain the reason that the vector clocks for conventional distributed systems are not suitable for ad-hoc distributed systems [5]**

Vector clocks are not suitable for ad-hoc distributed systems due to:

- Vector clocks rely on a known structure, which is challenging to maintain in dynamic ad-hoc systems.
- Managing vector clocks can become cumbersome and resource-intensive in rapidly changing ad-hoc environments.

- Ad-hoc systems may struggle to achieve and maintain the necessary synchronization levels for vector clocks.
- Ad-hoc systems may prioritize performance over strict consistency, which vector clocks emphasize.
- Ad-hoc systems require event tracking mechanisms that can easily adapt to changing requirements, while vector clocks are less flexible in this regard.
- Lamport clocks and causal ordering may better suit the dynamic and less structured nature of ad-hoc systems, offering a good balance between event ordering and performance.

iii. **Discuss the new design idea for vector clocks concerning the reasons mentioned for *ii*[10]**

A new design idea for vector clocks in ad-hoc distributed systems should focus on:

- Allow dynamic adjustments to vector clocks as nodes join or leave the network.
- Use efficient data structures to handle varying numbers of participants without performance degradation.
- Enable local management of vector clocks by each node and establish mechanisms for inter-node information exchange.
- Implement asynchronous event ordering for improved system performance while maintaining reasonable event order.
- Dynamically adjust synchronization levels based on network conditions to avoid rigidity.
- Emphasize event causality over strict global event ordering, considering causality-based approaches like Lamport clocks.
- Offer mechanisms for making trade-offs between consistency and performance to accommodate the dynamic nature of ad-hoc systems.
- Efficiently manage event and vector clock metadata to prevent unnecessary data overload
- Implement machine learning or heuristic-based approaches to predict event ordering in dynamic ad-hoc environments.
- Enable automatic adjustment of event tracking mechanisms to varying usage patterns and requirements in ad-hoc systems.

iv. **Adapt the implementation rules for the new design discussed for *iii* [10]**

The implementation rules for the new design idea for vector clocks in ad-hoc distributed systems can be summarized as follows:

- Each process maintains a queue with its identifier (Ci.id), clock value (Ci.clock, initialized to 0), time-to-live (Ci.ttl), and a counter (Ci.counter, initialized to 1) for tracking events.
- Sending Event/All Events:
    - ➢ When a process (i) sends an event:
        - ▪ Increment its own clock value (Ci.clock++) to represent the occurrence of the event.
        - ▪ Decrease the time-to-live (Ci.ttl) of the process.

        - If sending an event to all processes (broadcast), adjust time-to-live (Ck.ttl--) for each process (k).
- Receiving Event:
    - When process i receives an event from process k:
        - Check if the sender's (k) identifier is not in the vector clock of the receiver (j).
        - If not, increment the counter (Ci.counter++) and add an entry to the vector clock of process j, noting the identifier (k) and the clock value from the received timestamp (tm).
- If the identifier is already in the vector clock of j, update the clock value (Cj.clock) to the maximum of the current clock value and the one from the received timestamp (tm.clock).
- Expired Process:
    - Periodically, check the time-to-live (Ci.ttl) of each process in the queue.
    - If a process (Ci) has a time-to-live of 0, remove it from the queue and decrement the counter (Ci.counter--) as it's no longer participating in the system.

Algorithm:

All the processes should maintain a queue initially with its own identifier in the queue and clock value. Vector is an array of structures and for process i will look like [{i, 0, t, 1}]
Ci.id = i, Ci.clock = 0 and Ci.ttl = t, Ci.counter = 1 For process i, Ci.ttl = ∞

Sending Event/All events:

```
    Ci.id == i -> Ci.clock++
    Ck.ttl–
```

Receiving Event:

```
    Sender -> i
    Receiver -> j
    Other Processes -> k
    Time stamp sent -> tm
```

If process id is not in the vector clock of j, then

```
    ++counter
    Cj[counter].id = k
    Cj[counter].clock = Clock of value of that process in the timestamp
```

Else

```
    Cj.id == k
    Cj.clock = max(Cj.clock, tm.clock)
```

Expired Process:

If Ci.id == k and Ci.ttl == 0, remove it from the queue, Ci.counter--.

v. **Simulate the working of the newly designed vector clock through relevant and detailed examples cases depicting different degrees of ad-hocness [30]**

Less Ad-Hoc Scenario
Nodes have predefined roles, and communication is relatively reliable.

- Node A: Sends a message to Node B.
- Node B: Processes the message and sends a response.
- Node C: Occasionally joins the network. Vector Clocks in this scenario:
- A: [1, 0, 0]
- B: [0, 1, 0]

Node A sends a message to Node B:

- A increments its own vector clock: A: [2, 0, 0]
- Sends the message with its updated vector clock: [2, 0, 0]

Node B processes the message:

- B increments its own vector clock: B: [0, 2, 0]
- Sends a response with its updated vector clock: [0, 2, 0] Node C joins the network:
- C initializes its vector clock: C: [0, 0, 1]

Structure is somewhat stable

Moderately Ad-Hoc Scenario
In this scenario, we have a more dynamic environment with occasional node departures and arrivals.

- Node A: Sends a message to Node B.
- Node B: Processes the message and sends a response.
- Node C: Occasionally joins and leaves the network.

Vector Clocks in this scenario:

- A: [1, 0, 0]
- B: [0, 1, 0]

Node A sends a message to Node B:

- A increments its own vector clock: A: [2, 0, 0]
- Sends the message with its updated vector clock: [2, 0, 0]

Node B processes the message:

- B increments its own vector clock: B: [0, 2, 0]
- Sends a response with its updated vector clock: [0, 2, 0]

Node C joins and leaves the network:

- C initializes its vector clock: C: [0, 0, 1]
- Later, Node C leaves the network.

Vector clocks still work relatively well, but the occasional arrivals and departures of Node C introduce some dynamic elements.

Highly Ad-Hoc Scenario

- Node A: Sends a message to Node B.
- Node B: Processes the message and sends a response.
- Node C, D, and E: Join and leave the network frequently.

Vector Clocks in this scenario:

- A: [1, 0, 0]
- B: [0, 1, 0]

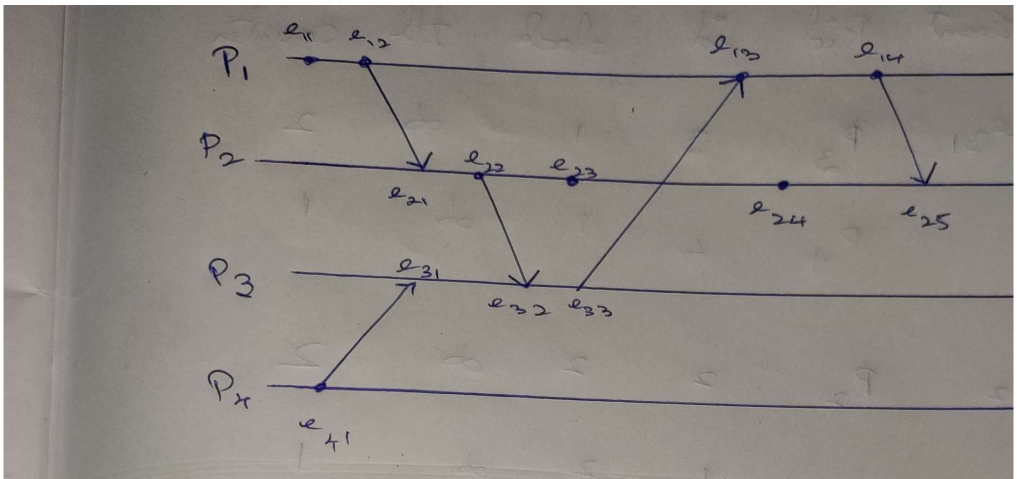
Node A sends a message to Node B:

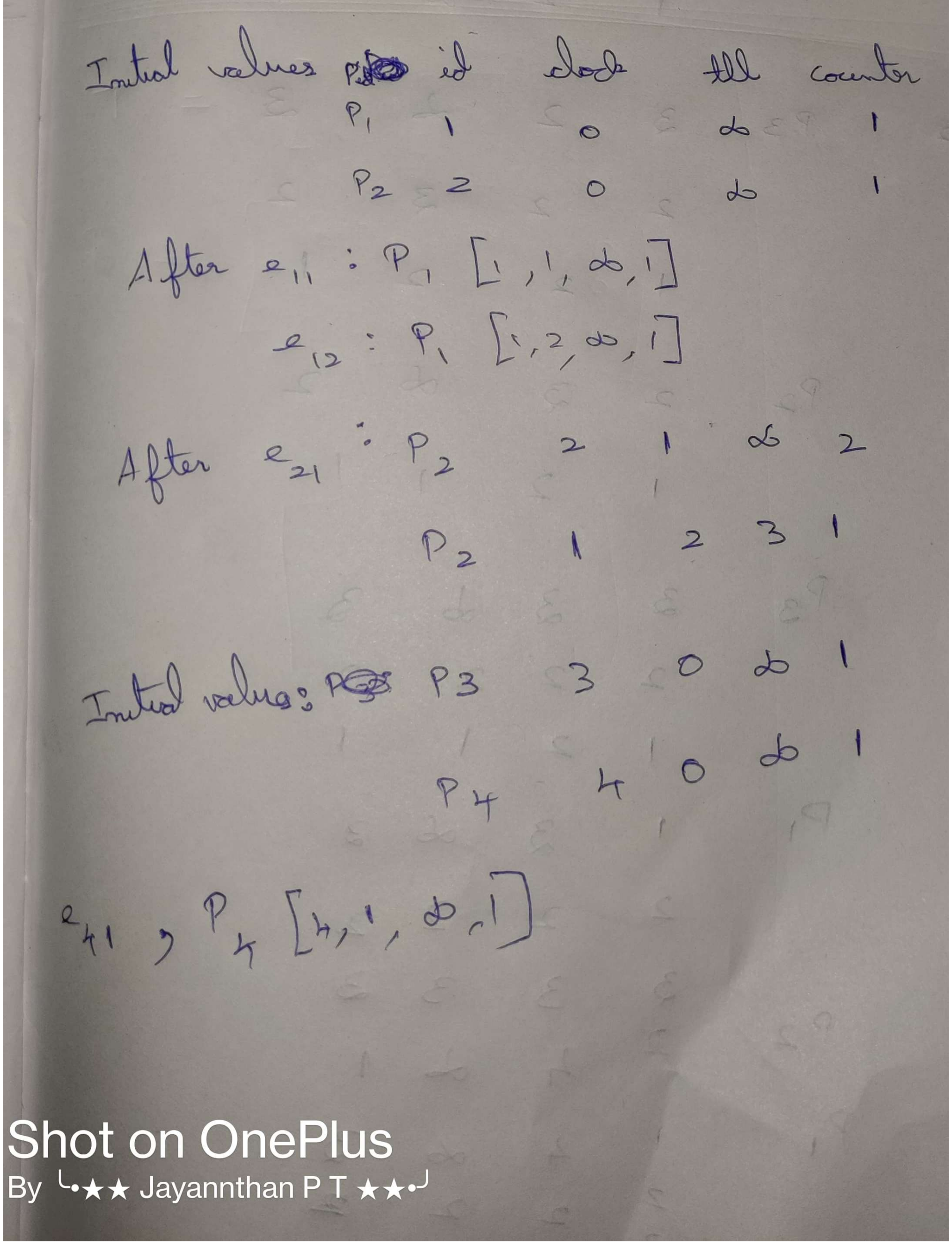
- A increments its own vector clock: A: [2, 0, 0]
- Sends the message with its updated vector clock: [2, 0, 0]

Node B processes the message:

- B increments its own vector clock: B: [0, 2, 0]
- Sends a response with its updated vector clock: [0, 2, 0]

Node C, D, and E join and leave the network at various times, causing frequent updates to their vector clocks.





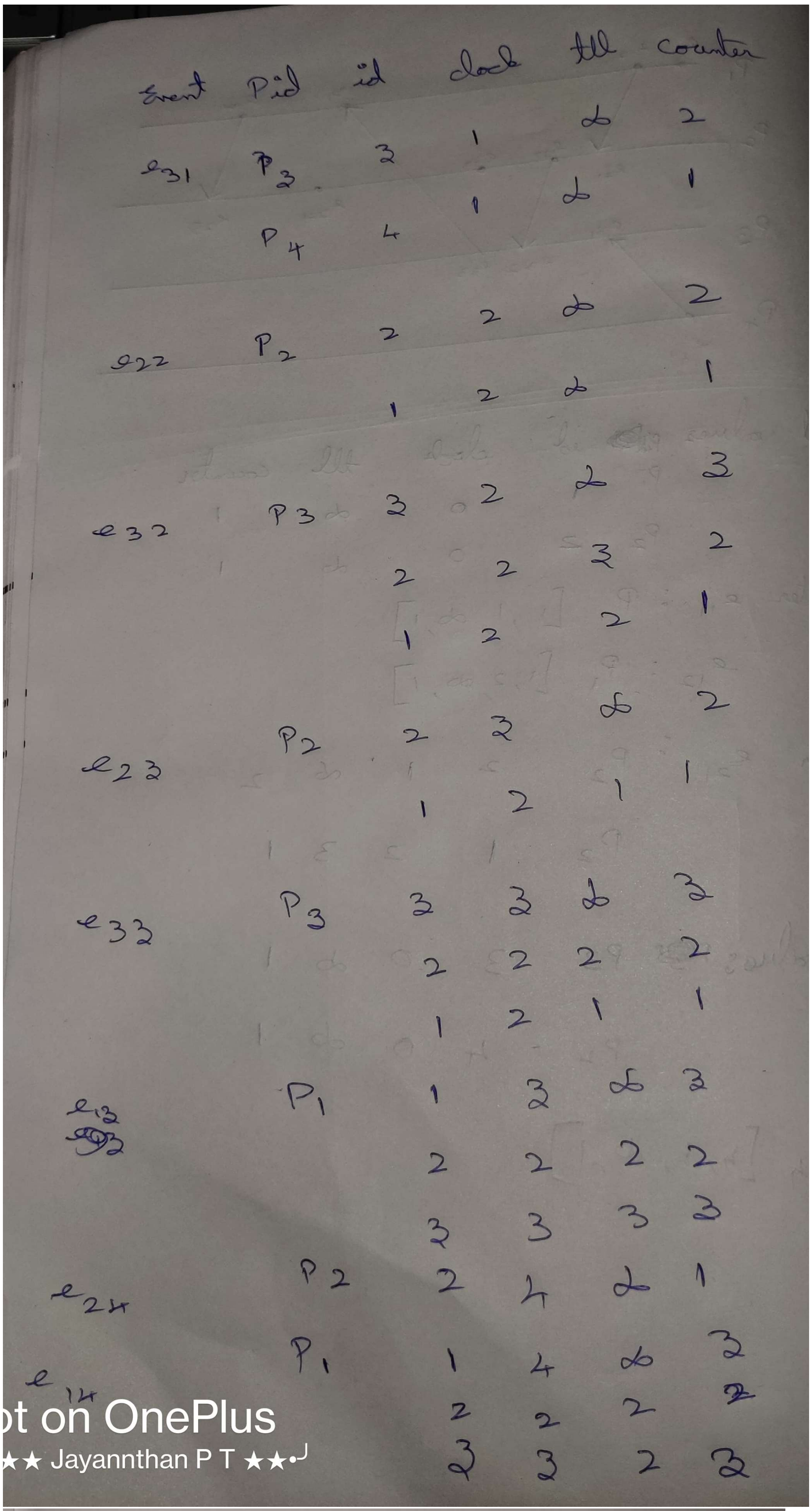
Initial values	Pid	id	clock	ttl	counter
	P1	1	0	∞	1
	P2	2	0	∞	1

After e_{11} : P_1 [1, 1, ∞, 1]

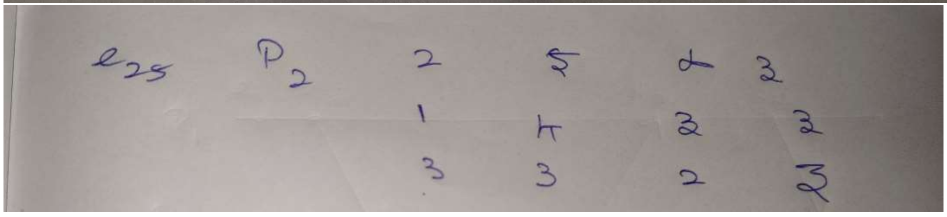
e_{12} : P_1 [1, 2, ∞, 1]

After e_{21} :	P_2	2	1	∞	2
	P_2	1	2	3	1
Initial values:	P3	3	0	∞	1
	P4	4	0	∞	1

e_{41} , P_4 [4, 1, ∞, 1]



Event	Pid	id	clock	ttl	counter
e_{31}	P_3	3	1	∞	2
	P_4	4	1	∞	1
e_{22}	P_2	2	2	∞	2
		1	2	∞	1
e_{32}	P_3	3	2	∞	3
		2	2	3	2
		1	2	2	1
e_{23}	P_2	2	3	∞	2
		1	2	1	1
e_{33}	P_3	3	3	∞	3
		2	2	2	2
		1	2	1	1
e_{13}	P_1	1	3	∞	3
		2	2	2	2
		3	3	3	3
e_{24}	P_2	2	4	∞	1
e_{14}	P_1	1	4	∞	3
		2	2	2	2
		3	3	2	3



e_{25}	P_2	2	5	∞	3
		1	4	3	3
		3	3	2	3