Shri Keshavinee R **205001101**

UCS1701 - Distributed Systems

ASSIGNMENT - 1

1) Highlight the abstract difference between a conventional distributed system and an ad-hoc distributed system.

The abstract difference between a conventional distributed system and an ad-hoc distributed system lies in their primary design and operational characteristics:

Conventional Distributed System:

- Purpose: Conventional distributed systems are designed and structured for specific, predefined tasks or applications. They are created with a particular use case or set of use cases in mind.
- Centralization: They often exhibit a more centralized and structured architecture where nodes or components have predefined roles and responsibilities.
- Consistency: Conventional distributed systems prioritize maintaining consistency and reliability of data and services across the network.
- Scalability: They are typically built with scalability in mind but usually require careful planning and configuration to accommodate growth.

Ad-Hoc Distributed System:

- Purpose: Ad-hoc distributed systems are more flexible and adaptable. They are created on-the-fly to solve a specific problem or address an immediate need. They lack predefined, fixed purposes.
- Decentralization: These systems are inherently decentralized and often have a more fluid and dynamic structure. Nodes can join or leave the network as needed, without strict predefined roles.
- Consistency: Ad-hoc distributed systems may prioritize speed and responsiveness over strict consistency, making them more suitable for certain dynamic or real-time scenarios.
- Scalability: Ad-hoc systems are typically more flexible and can adapt to changing requirements or a fluctuating number of participants without extensive prior planning.

In summary, the key distinction is that conventional distributed systems are purpose-built, structured, and prioritize consistency, while ad-hoc distributed systems are more fluid, decentralized, and prioritize adaptability and responsiveness to changing requirements. Ad-hoc systems are often employed for tasks that require quick deployment and are less concerned with strict data consistency or long-term planning.

2) Explain the reason that the vector clocks for conventional distributed systems are not suitable for ad-hoc distributed systems

Vector clocks, a concept often used in distributed systems, are a mechanism to track and order events in a distributed environment. While they are suitable for many conventional distributed systems, they may not be ideal for ad-hoc distributed systems due to the following reasons:

Predefined Structure vs. Dynamic Nature:
Conventional Distributed Systems: In conventional distributed systems, the structure is often predefined, and nodes have fixed roles and responsibilities. Vector clocks can be set up to accommodate these specific roles, ensuring that events are correctly ordered.

Ad-Hoc Distributed Systems:
Ad-hoc systems are highly dynamic, and participants can join and leave the network rapidly. Vector clocks rely on a known structure, which can be challenging to maintain in these fluid environments.

Complexity and Overhead:
Vector clocks can become complex and introduce significant overhead when dealing with a large number of nodes or events. In an ad-hoc system with a constantly changing set of participants and frequent events, managing vector clocks can become cumbersome and resource-intensive.

Synchronization Challenges:
Ad-hoc systems often have varying degrees of connectivity and synchronization among nodes. Vector clocks rely on accurate synchronization among nodes to maintain consistency. In an ad-hoc environment, achieving and maintaining this level of synchronization can be difficult.

Trade-offs between Consistency and Performance:
Vector clocks prioritize maintaining strict event ordering and consistency. In some ad-hoc scenarios, maintaining perfect consistency might not be as critical as achieving high performance or responsiveness. Ad-hoc systems often need to make trade-offs that vector clocks may not readily support.

Flexibility and Adaptability:

- Ad-hoc distributed systems are typically designed to be flexible and adaptable, which means they may need to adjust their event tracking mechanisms on-the-fly. Vector clocks, once established, may not easily adapt to these changing requirements.
- In ad-hoc distributed systems, other event tracking mechanisms, such as Lamport clocks or causal ordering, may be more suitable due to their ability to handle the dynamic and less structured nature of these systems. These mechanisms are often more lightweight and can offer a good balance between event ordering and system performance, making them a better fit for ad-hoc scenarios where adaptability and responsiveness are essential.

3) Discuss the new design idea for vector clocks concerning the reasons mentioned for ii

To address the challenges associated with using traditional vector clocks in ad-hoc distributed systems, a new design idea can be considered. This design should aim to provide a more flexible and adaptable approach to event tracking and ordering in dynamic, decentralized environments. Here are some concepts that could be part of this new design:

Dynamic Vector Clocks:
Instead of a fixed, predefined structure, the vector clock design for ad-hoc systems could incorporate a dynamic element. Nodes or participants can join or leave the network, and vector clocks should adapt accordingly, dynamically adding or removing components as needed.

Scalable Data Structures:

Utilize scalable data structures that can handle a varying number of participants efficiently. This ensures that the overhead associated with maintaining vector clocks doesn't become a limiting factor in the system's performance.

Decentralized Event Tracking:
Reduce the reliance on centralized or structured event tracking. Allow each node to manage its vector clock locally, with mechanisms in place to exchange information with other nodes as necessary.

Asynchronous Event Ordering:
Consider asynchronous event ordering mechanisms that allow for a more relaxed form of consistency. This can help improve system performance while still providing a reasonable level of order among events.

Adaptive Synchronization:
Incorporate adaptive synchronization techniques that can dynamically adjust the level of synchronization among nodes based on the current network conditions. This can help avoid the rigidity of traditional vector clocks.

Event Causality:
Focus on capturing event causality, which is often more critical in ad-hoc systems than strict global event ordering. Causality-based approaches, such as Lamport clocks, may be more suitable for this purpose.

Practical Trade-offs:
Provide mechanisms for making practical trade-offs between consistency and performance. Ad-hoc systems may need to relax consistency requirements under certain conditions, and the vector clock design should accommodate this flexibility.

Efficient Metadata Management:
Develop efficient ways to manage metadata related to events and vector clocks, ensuring that it doesn't overwhelm the system with unnecessary data.

Machine Learning or Heuristics:

Implement machine learning or heuristic-based approaches to predict event ordering in situations where strict ordering is challenging to achieve due to the dynamic nature of ad-hoc systems.

Self-Configuration:
Enable self-configuration of event tracking mechanisms, allowing the system to adjust automatically to different usage patterns and requirements.

In summary, the new design idea for vector clocks in ad-hoc distributed systems should prioritize adaptability, scalability, and asynchronous event ordering while still maintaining an acceptable level of event causality. This design would be better suited to the dynamic and unpredictable nature of ad-hoc environments, where traditional vector clocks may not be the most practical solution.

4)Adapt the implementation rules for the new design discussed for iii
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--.

Initialization:

Each process maintains a queue with its own identifier, clock value (initialized to 0), time-to-live (ttl), and a counter (initialized to 1) for tracking events.

Sending Event/All Events:

- When a process (e.g., 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 (which limits how long it's in the queue).
- If sending an event to all processes (broadcast), similar adjustments are made for each process (e.g., Ck.ttl-- for process k).

Receiving Event:

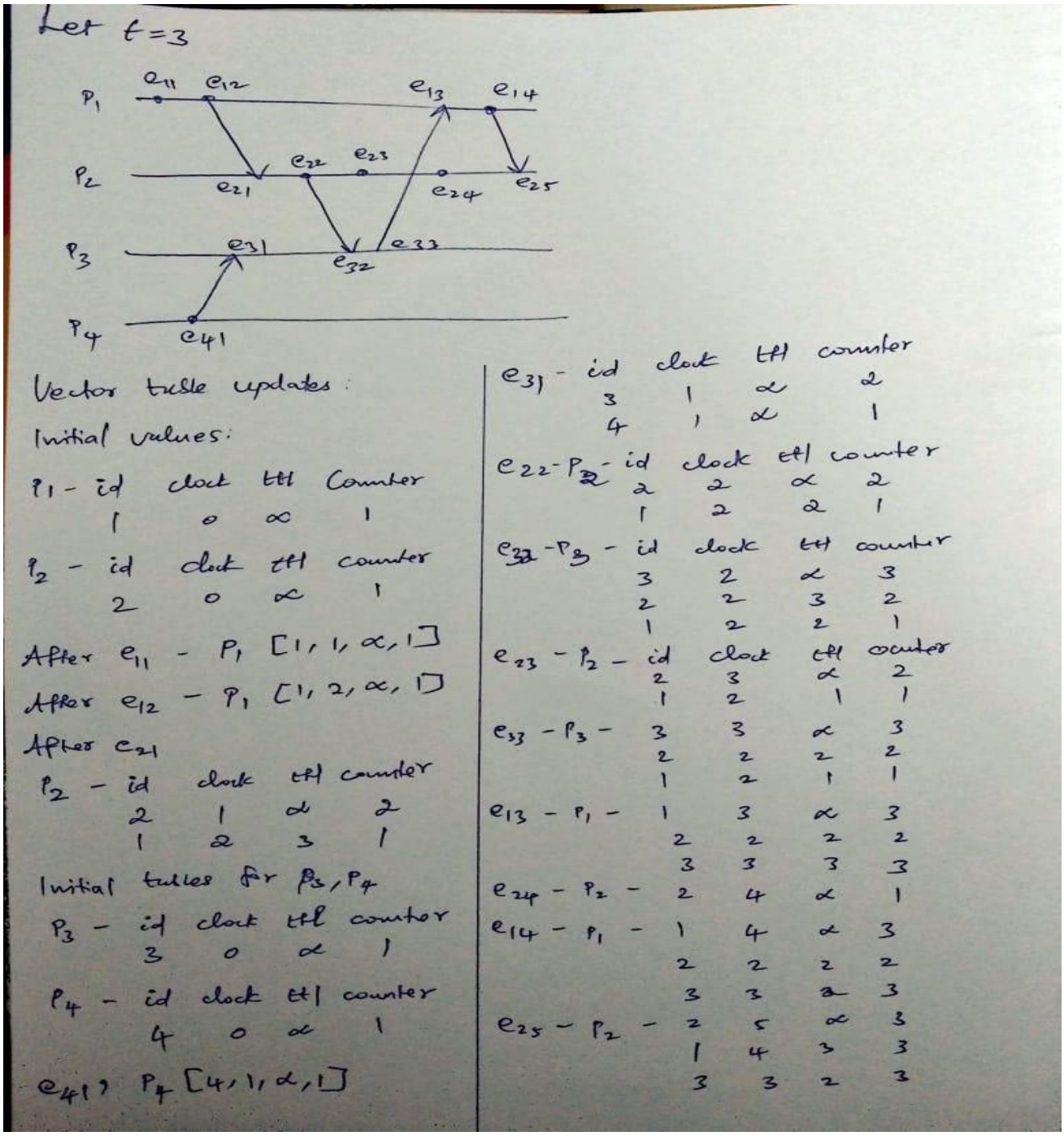
- When process i receives an event from process k:
- Check if the sender's (k) identifier is not in the vector clock of the receiver (j).
- If not, increment the counter and add an entry to the vector clock of process j, noting the identifier and the clock value from the received timestamp.
- If the identifier is already in the vector clock of j, update the clock value to the maximum of the current clock value and the one from the received timestamp (ensuring it represents the latest event).

Expired Process:

- Periodically, check the time-to-live (ttl) of each process in the queue.
- If a process (e.g., Ci) has a ttl of 0, it's removed from the queue, and the counter is decremented (Ci.counter--) as it's no longer participating in the system.
- This process helps manage the evolution of vector clocks for various processes in a distributed system, ensuring that events are correctly

ordered and tracked as they occur. It provides a mechanism for keeping vector clocks updated and removing processes that are no longer active.

5) Simulate the working of the newly designed vector clock through relevant and detailed examples for various cases depicting different degrees of ad-hocness



The working of the newly designed vector clock through relevant examples for various cases depicting different degrees of ad-hocness, ranging from less ad-hoc to highly ad-hoc scenarios.

Example 1: Less Ad-Hoc Scenario

In this scenario, we have a distributed system with a somewhat stable structure. 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]

In this less ad-hoc scenario, vector clocks are relatively straightforward because the structure is somewhat stable, and node interactions are predictable.

Example 2: 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.

In this moderately ad-hoc scenario, vector clocks still work relatively well, but the occasional arrivals and departures of Node C introduce some dynamic elements.

Example 3: Highly Ad-Hoc Scenario

In this scenario, the environment is highly dynamic, with frequent node departures, arrivals, and rapidly changing network conditions.

- 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.

The use of causality-based vector clocks and adaptive synchronization mechanisms is critical to maintain event ordering and consistency while accommodating the ad-hoc nature of the system.