NUST
NATIONAL UNIVERSITY
OF SCIENCES & TECHNOLOGY

Assignment-02

Name:	Muhammad Ali Tahir
CMS:	411327
Section:	BSCS-12C

Parallel and Distributed Computing

Synchronization in Distributed Systems - Report

Glossary:

1. Introduction: The Challenge of Time in Distributed Systems

Distributed systems, by their very nature, consist of multiple independent computing devices (nodes) connected via a network. These nodes collaborate to achieve a common goal. However, each node possesses its own local clock, which, due to various factors, may deviate from real-world time and from the clocks of other nodes. This discrepancy, known as clock skew, poses significant challenges for maintaining consistency and order within the system.

2. The Need for Synchronization in Distributed Systems

Synchronization of physical clocks is crucial for a variety of reasons, including:

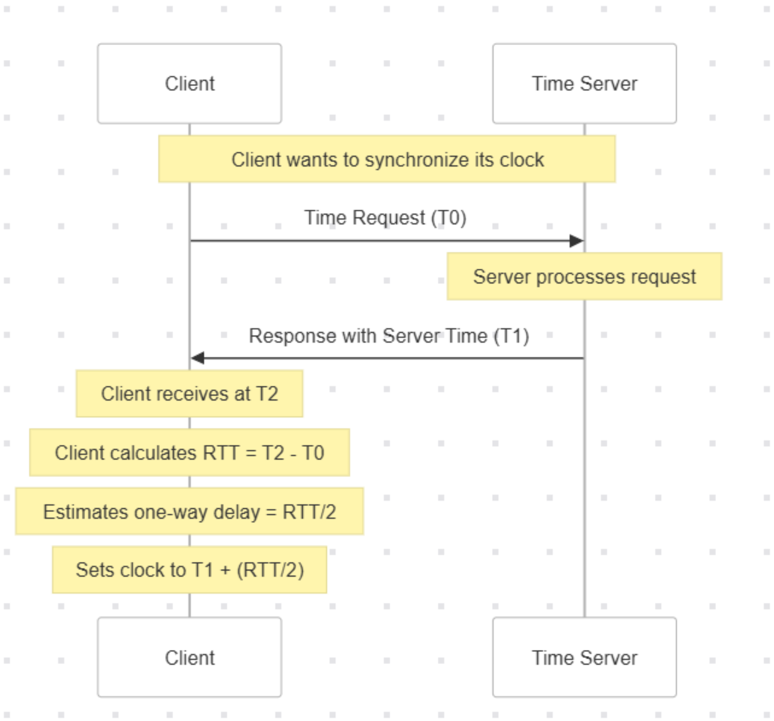
- **Ordering of Events:**
    - Many distributed applications rely on the correct temporal ordering of events. For instance, in a distributed database, transaction logs must be applied in the order they occurred. Without synchronized clocks, determining the true order of events becomes impossible, leading to inconsistencies.
    - Consider a scenario with two nodes, A and B. Node A sends a message to node B at its local time tA, and node B receives the message at its local time tB. If tA>tB due to clock skew, the system may incorrectly infer that the message was received before it was sent.
- **Consistency and Agreement:**
    - Distributed consensus algorithms, such as Paxos or Raft, depend on accurate timestamps to ensure that all nodes agree on a single, consistent state.
    - Distributed locking mechanisms also rely on synchronized clocks to prevent race conditions and data corruption.
- **Real-time Applications:**
    - Real-time systems, such as those used in industrial control or multimedia streaming, require precise timing to function correctly. Clock synchronization is essential for meeting strict deadlines and maintaining quality of service.
- **Debugging and Auditing:**
    - Accurate timestamps are vital for debugging distributed applications and auditing system behavior. Without synchronized clocks, it becomes difficult to trace the sequence of events and identify the root cause of errors.

- **Security:**
    - Time stamps are used in many security protocols, such as Kerberos, to prevent replay attacks. If clocks are not synchronized, these protocols may fail.
- **Causality:**
    - The "happened-before" relation, central to understanding causality in distributed systems, depends on the accurate ordering of events. If clocks are skewed, the perception of causality can be distorted.

3. Physical Clock Synchronization Algorithms

1. Cristian's Algorithm:

- **Working Diagram**



Client
Time Server
Client wants to synchronize its clock
Time Request (T0)
Server processes request
Response with Server Time (T1)
Client receives at T2
Client calculates RTT = T2 - T0
Estimates one-way delay = RTT/2
Sets clock to T1 + (RTT/2)
Client
Time Server

Tool Used: Mermaid.io

- **Concept:**
  - Cristian's algorithm is a simple client-server protocol for synchronizing a client's clock with a time server.
  - It relies on a single round-trip message exchange.
- **Working:**

1. **Time Request:**
   - The client sends a request message to the time server asking for the current time.
2. **Time Response:**
   - The time server receives the request and immediately sends back a response message containing its current timestamp.
3. **Round-Trip Time (RTT) Calculation:**
   - The client measures the time elapsed between sending the request and receiving the response. This is the RTT.
4. **Clock Adjustment:**
   - The client estimates the one-way network delay as half of the RTT.
   - The client adds this estimated delay to the timestamp received from the server to obtain an adjusted time.
   - The client then sets its local clock to this adjusted time.

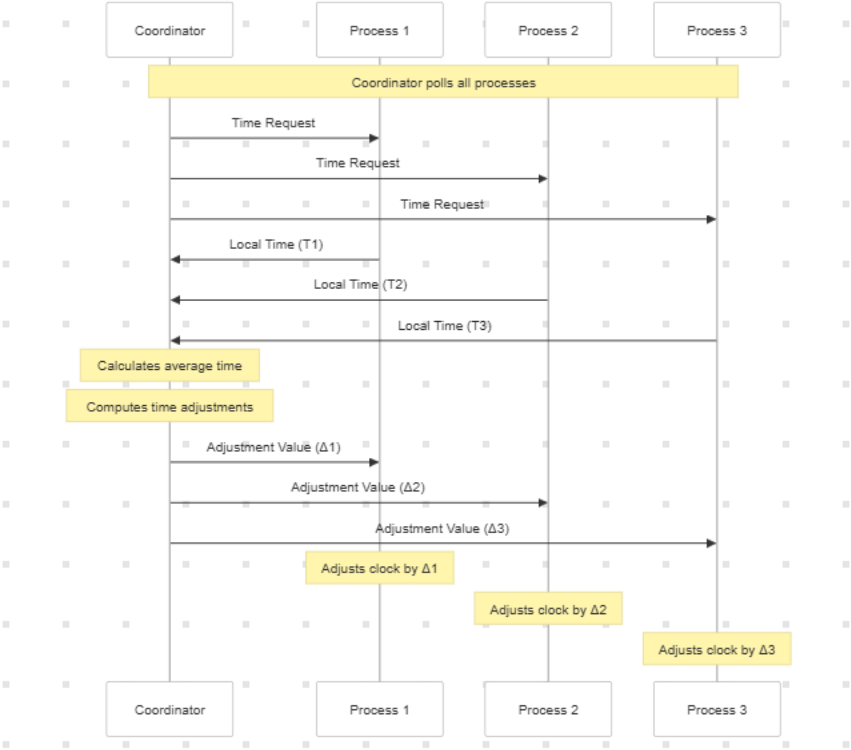
- **Limitations:**
  - Accuracy is directly affected by network latency.
  - Assumes symmetrical network delays (equal uplink and downlink delays), which is often not true.
  - Susceptible to server load and delays.
  - Single server failure creates a single point of failure.

Cristian Algorithm Example

- Alice's computer (client) needs to synchronize with Bob's time server.
- Alice sends a time request to Bob.
- Bob responds with a timestamp of 12:00:00.
- Alice measures the RTT as 20 milliseconds.
- Alice calculates the estimated one-way delay as 10 milliseconds.
- Alice adjusts her clock to 12:00:10.

2. Berkeley Algorithm:

- **Working Diagram**



Tool Used: Mermaid.io

- **Concept:**
    - The Berkeley algorithm is designed for synchronizing clocks within a local area network (LAN).
    - It uses a master-slave approach, where a master node coordinates the synchronization.
- **Working:**

1. **Master Polling:**
    - The master node periodically sends requests to all slave nodes in the network, asking for their current clock values.
2. **Slave Responses:**
    - Each slave node responds with its current clock value.
3. **Average Calculation:**
    - The master node collects the clock values from all slaves.
    - It calculates the average of these clock values, potentially excluding outliers.
4. **Adjustment Messages:**
    - The master node calculates the difference between the average time and each slave's time.
    - It sends adjustment messages to each slave, instructing them to advance or retard their clocks accordingly.
5. **Clock Adjustment:**
    - Each slave node adjusts its clock based on the received adjustment message.

- **Limitations:**
    - Single point of failure (master node).
    - Relies on reliable communication between the master and slaves.

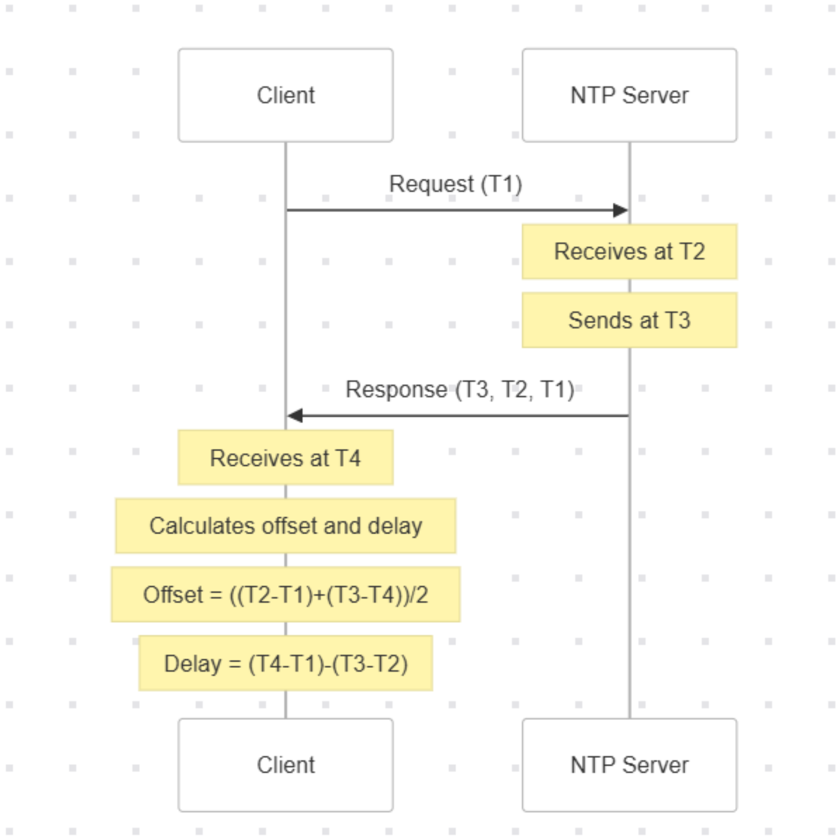
  - Accuracy of the average is dependent on the quality of the slave clock values.

Berkeley Algorithm Example

- A master node (M) synchronizes three slave nodes (S1, S2, S3).
- M polls the slaves:
  - S1: 10:00:01
  - S2: 09:59:58
  - S3: 10:00:04
- M calculates the average: 10:00:01.
- M sends adjustments:
  - S1: -1 second
  - S2: +3 seconds
  - S3: -3 seconds.
- The slaves adjust their clocks accordingly.

3. Network Time Protocol (NTP): Detailed Description

- **Working Diagram**



Tool Used: Mermaid.io

- **Concept:**
    - NTP is a protocol for synchronizing computer clocks over packet-switched networks.
    - It uses a hierarchical system of time servers, with stratum levels indicating the distance from a reference clock.
- **Working:**

1. **Stratum Hierarchy:**
    - Stratum 0: Reference clocks (e.g., atomic clocks, GPS).
    - Stratum 1: Servers directly connected to stratum 0 clocks.

        - Stratum 2: Servers connected to stratum 1 servers, and so on.

2. **Message Exchange:**
    - Clients exchange NTP packets with servers.
    - These packets contain timestamps of when the request was sent, when it was received, and when the response was sent.
3. **Offset and Delay Calculation:**
    - NTP uses the timestamps to calculate the offset (difference between client and server clocks) and the delay (network round-trip time).
4. **Filtering and Selection:**
    - NTP clients typically connect to multiple servers.
    - NTP uses algorithms to filter out inaccurate time samples and select the most reliable time sources.
5. **Clock Adjustment:**
    - NTP gradually adjusts the client's clock to the time provided by the server, taking into account the calculated offset and delay.
6. **Continuous Synchronization:**
    - NTP clients periodically exchange packets to maintain accurate time.

- **Key Features:**
    - High accuracy (sub-millisecond).
    - Resilience and scalability.
    - Compensation for network delays and clock drift.
    - Secure NTP (NTS) provides authentication and encryption.

Network Time Protocol (NTP) Example

- A laptop (client) synchronizes with an NTP server.
- The laptop sends an NTP request.
- The server responds with timestamps.

- The laptop calculates the offset and delay.
- The laptop filters and selects the best time source from multiple servers.
- The laptop gradually adjusts its clock.
- The laptop continues to send NTP requests to maintain time

Advantages and limitations of Cristian's, Berkeley and NTP:

Cristian's Algorithm:

- **Advantages:**
    - Simplicity: Easy to implement and understand.
    - Low overhead: Minimal communication overhead for individual clients.
- **Limitations:**
    - Accuracy: Highly dependent on network latency, making it less accurate in variable network conditions.
    - Symmetrical delay assumption: Assumes equal uplink and downlink delays, which is often not accurate.
    - Single point of failure: Relies on a single time server, making it vulnerable to server failures.
    - Server load: Server load can drastically increase the time it takes to respond, and thus reduce accuracy.

Berkeley Algorithm:

- **Advantages:**
    - Internal synchronization: Effective for synchronizing clocks within a local network.
    - Averaging: Averages clock values, reducing the impact of individual clock drifts.
- **Limitations:**

- Single point of failure: Relies on a master node, which can be a single point of failure.
- LAN dependency: Best suited for LANs, less effective in wide-area networks.
- Reliability: Relies on all slave nodes responding to the master.
- Accuracy: The accuracy is limited to the accuracy of the clocks within the LAN.

Network Time Protocol (NTP):

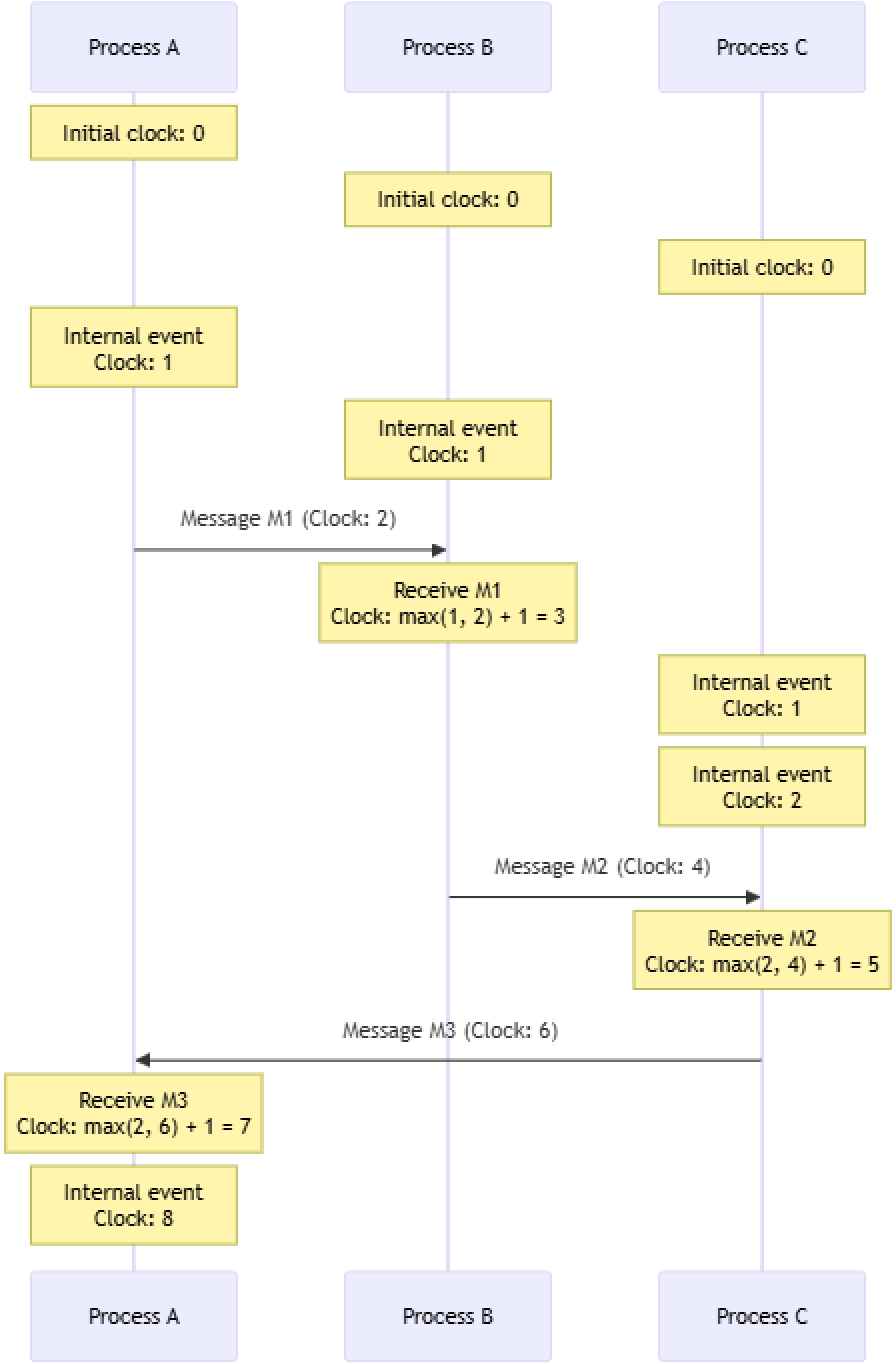
- **Advantages:**
    - High accuracy: Achieves high accuracy (sub-millisecond) through sophisticated algorithms and hierarchical time servers.
    - Scalability: Designed for large-scale networks, including the internet.
    - Resilience: Uses redundant time servers and stratum levels for fault tolerance.
    - Compensates for network delays: Uses algorithms to minimize the impact of network delays.
    - Wide spread adoption: Is a very common and well tested protocol.
- **Limitations:**
    - Complexity: Complex implementation, requiring careful configuration and maintenance.
    - Network congestion: Accuracy can be affected by network congestion.
    - Security vulnerabilities: Vulnerable to certain security attacks, although Secure NTP (NTS) addresses these.
    - Overhead: Has a higher overhead than the other two algorithms.

4. Logical Clock Synchronization

Logical clocks are used to capture the causal relationships between events in a distributed system without relying on physical time. They provide a way to order events based on their occurrence within the system's logical timeline.

1. Scalar Clocks (Lamport's Clocks)

- **Working Diagram**



Tool Used: Mermaid.io

- **Concept:**
  - Lamport's clocks assign a numerical timestamp to each event, ensuring that if event A "happened before" event B, then the timestamp of A is less than the timestamp of B.
  - They capture the causal ordering of events but do not necessarily reflect the actual time of occurrence.
- **Rules:**

Local Increment:

Before each event at a process, increment the clock by 1.

Message Passing:

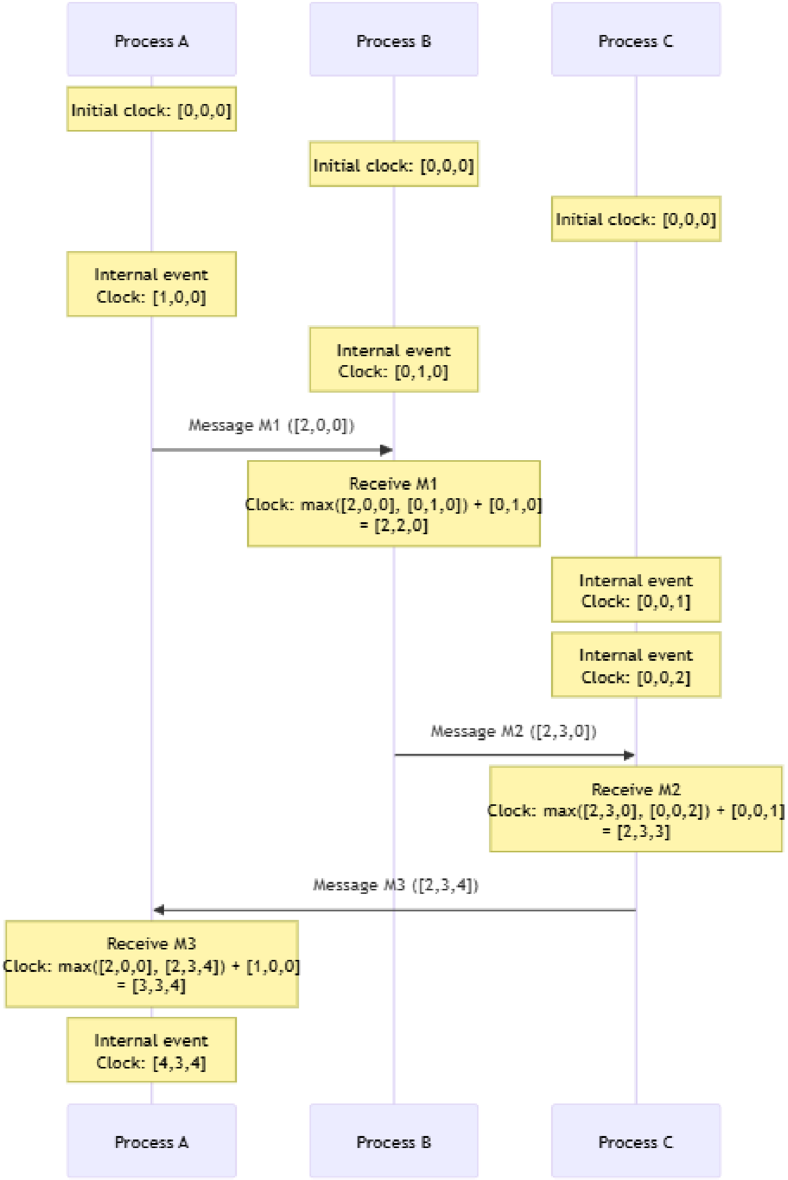
1. When a process sends a message, it includes its current clock value.
2. When a process receives a message, it sets its clock to the maximum of its current clock and the [1] received clock, and then increments it by 1.

- **Example:**
  - Process P1: Events A, B, C
  - Process P2: Events D, E, F
  - Initial clock values: P1 = 0, P2 = 0
  - A (P1): Clock = 1
  - P1 sends message to P2 with clock = 1
  - D (P2): Clock = max(0, 1) + 1 = 2
  - E (P2): Clock = 3
  - P2 sends message to P1 with clock = 3
  - B (P1): Clock = max(1, 3) + 1 = 4
  - C (P1): Clock = 5
  - F (P2): Clock = 4
- **Limitations:**

- Lamport's clocks can only capture a partial ordering of events. If two events have the same timestamp, their causal relationship is unknown.
- The reverse is not always true, if A's time stamp is less than B's time stamp, A did not necessarily happen before B.

2. Vector Clocks

- **Working Diagram**



Tool Used: Mermaid.io

- **Concept:**
    - Vector clocks provide a more precise way to capture causal relationships.
    - Each process maintains a vector of timestamps, where the i-th element of the vector represents the logical time of the i-th process.
- **Rules:**

Local Increment:

Before each event at process i, increment the i-th element of its vector by 1.

Message Passing:

1. When a process sends a message, it includes its vector clock.
2. When a process receives a message, it updates each element of its vector to the maximum of its current value and the received value, and then increments its own element by 1.

- **Example:**
    - Processes P1, P2, P3
    - Initial vector clocks: P1 = [0, 0, 0], P2 = [0, 0, 0], P3 = [0, 0, 0]
    - A (P1): P1 = [1, 0, 0]
    - P1 sends message to P2: P1 = [1, 0, 0]
    - D (P2): P2 = [1, 1, 0]
    - E (P2): P2 = [1, 2, 0]
    - P2 sends message to P3: P2 = [1, 2, 0]
    - G (P3): P3 = [1, 2, 1]
    - P3 sends message to P1: P3 = [1, 2, 1]
    - B (P1): P1 = [2, 2, 1]
- **Causality:**

    - Vector clock A "happened-before" vector clock B if and only if each element of A is less than or equal to the corresponding element of B, and at least one element is strictly less.

- **Advantages:**
    - Vector clocks provide a total ordering of causally related events.
    - They can detect concurrent events.
- **Disadvantages:**
    - Vector clocks increase in size with the amount of processes.
    -

3. Real-World Scenario: Distributed Version Control System (DVCS)

- **Scenario:**
    - A DVCS like Git involves multiple developers working on the same codebase, committing changes from different locations.
    - We need to track the order of commits and resolve conflicts.
- **Why Logical Clocks?**
    - Physical clocks are unreliable across different machines and time zones.
    - We care about the causal order of commits, not their precise physical timestamps.
    - Vector clocks can be used to:
        - Track the dependencies between commits.
        - Detect concurrent commits (branches).
        - Determine which commits need to be merged to resolve conflicts.
- **How it Works:**
    - Each developer's client maintains a vector clock.
    - When a developer commits, their vector clock is incremented.

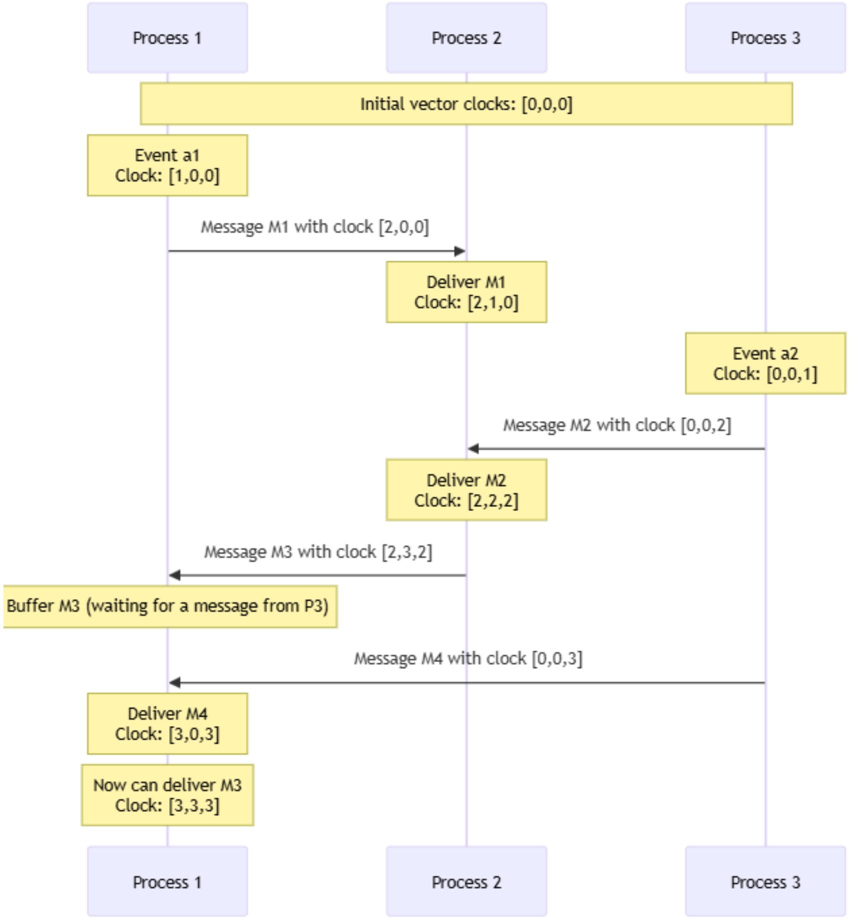
- When developers pull changes from the remote repository, their vector clocks are updated by merging the remote vector clocks.
- When resolving conflicts, the DVCS uses the vector clocks to determine the order of changes and identify conflicting modifications.

5. Causal Ordering Algorithms

Causal ordering ensures that if a message A "happened before" message B, then A is delivered before B. This is crucial for maintaining consistency in distributed systems.

1. Birman-Schipper-Stephenson (BSS) Algorithm: Detailed Review

- **Working Diagram**



Tool Used: Mermaid.io

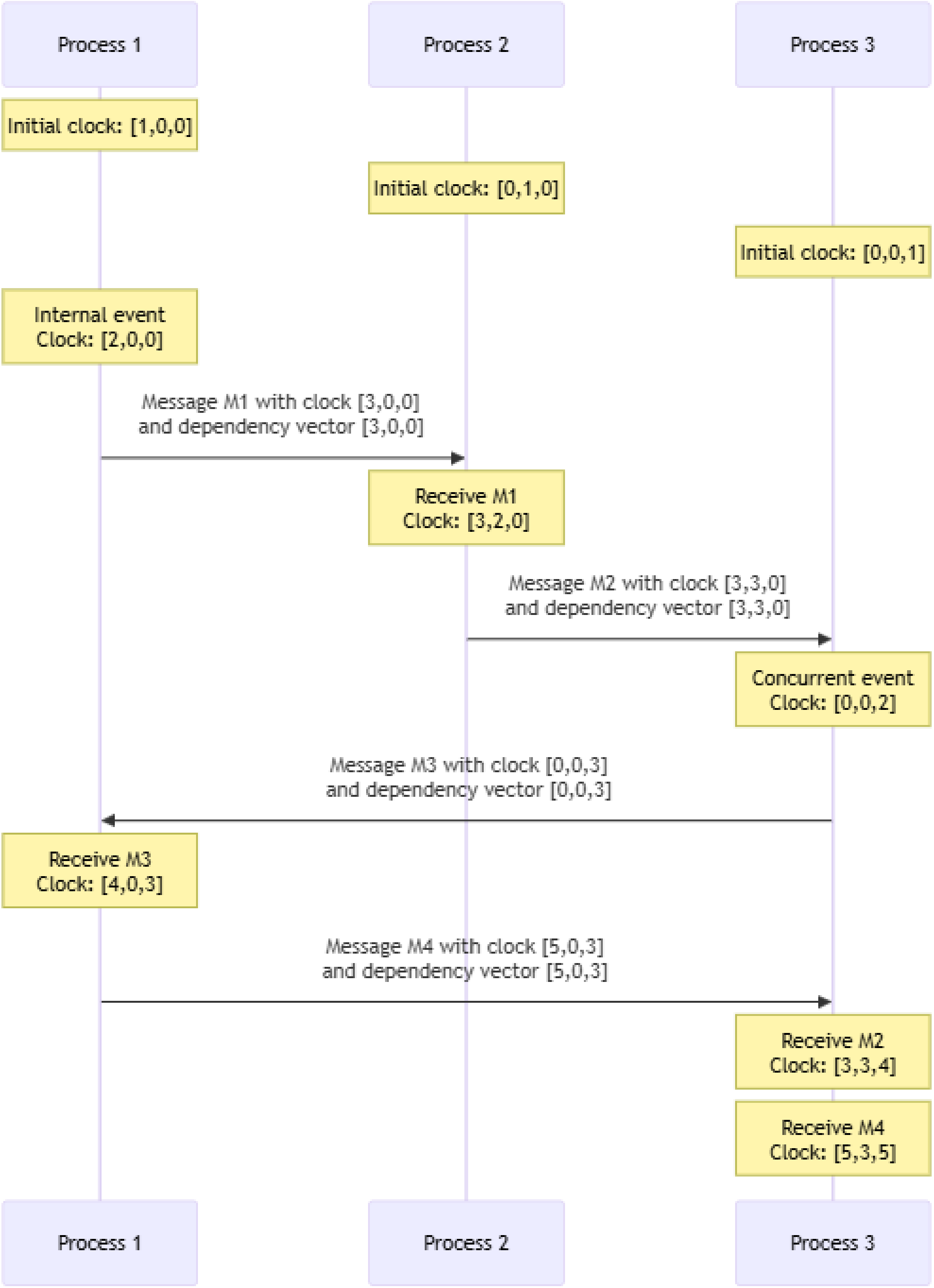
- **Core Idea:**

BSS ensures causal message delivery using vector clocks. Each process maintains a vector clock, and messages are delivered only when all causally preceding messages are known to have been received.

- **Mechanism:**
    - Each process Pi maintains a vector clock VCi.
    - When Pi sends a message m, it attaches VCi to m.
    - When Pj receives m with vector clock VCm, it compares VCm with its own VCj.
    - Pj delivers m only if:
        - VCm[i]=VCj[i]+1 (meaning m is the next event from Pi).
        - VCm[k]≤VCj[k] for all k⍰=i (meaning Pj has seen all events that Pi has seen).
    - If the conditions are not met, m is buffered.
- **Example:**
    - Processes: P1, P2, P3.
    - Initial vector clocks: VC1=[0,0,0], VC2=[0,0,0], VC3=[0,0,0].
    - P1 sends message m1 to P2: VCm1=[1,0,0].
    - P2 receives m1: VC2 becomes [1,0,0].
    - P2 sends message m2 to P3: VCm2=[1,1,0].
    - P1 sends message m3 to P3: VCm3=[2,0,0].
    - P3 receives m2: VC3 becomes [1,1,0]. m2 is delivered.
    - P3 receives m3: VC3 becomes [2,1,0]. m3 is buffered, because VCm3[2]>VC3[2].
    - If P2 then sent a message to P3, VC3 would be updated, and then m3 would be delivered.
- **Pros:** Ensures strict causal ordering.
- **Cons:** High overhead, potential for message delays.

3. ***Schwarz & Mattern (SES) Algorithm: Detailed Review***

- **Working Diagram**



Tool Used: Mermaid.io

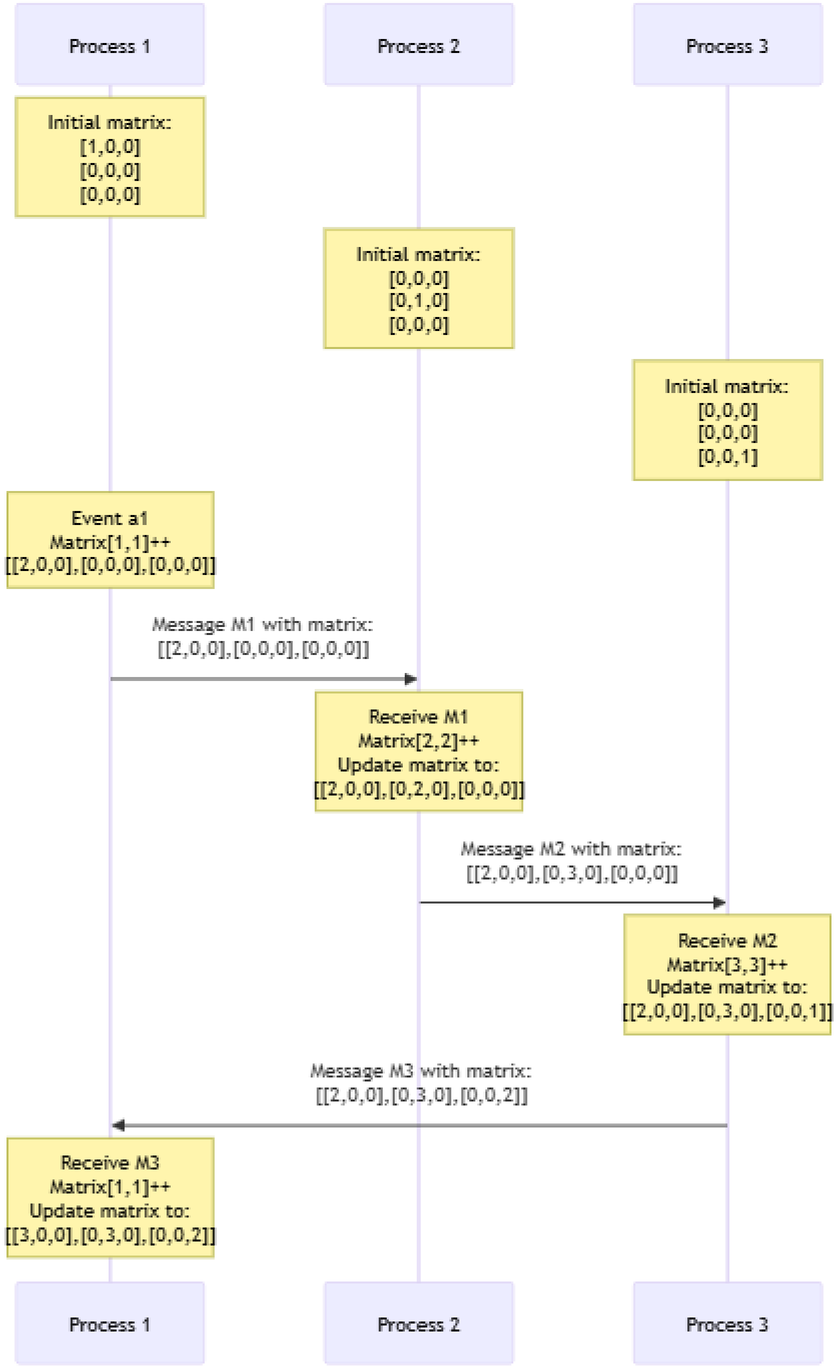
- **Core Idea:**

SES uses sequence numbers to approximate causal ordering. It's less strict than BSS but more efficient.

- **Mechanism:**
    - Each process Pi maintains a sequence number SNi.
    - When Pi sends a message m, it attaches SNi to m and increments SNi.
    - Each process Pj maintains an array Rj, where Rj[i] stores the last sequence number received from Pi.
    - When Pj receives m from Pi with sequence number SNm, it compares SNm with Rj[i]+1.
    - If SNm=Rj[i]+1, m is delivered, and Rj[i] is updated. Otherwise, m is buffered.
- **Example:**
    - Processes: P1, P2, P3.
    - Initial sequence numbers: SN1=0, SN2=0, SN3=0.
    - Initial receive arrays: R1=[0,0,0], R2=[0,0,0], R3=[0,0,0].
    - P1 sends m1 to P2: SNm1=0, SN1 becomes 1.
    - P2 receives m1: SNm1=R2[1]+1, m1 is delivered, R2[1] becomes 0.
    - P2 sends m2 to P3: SNm2=0, SN2 becomes 1.
    - P1 sends m3 to P3: SNm3=1, SN1 becomes 2.
    - P3 receives m2: SNm2=R3[2]+1, m2 is delivered, R3[2] becomes 0.
    - P3 receives m3: SNm3=R3[1]+1, m3 is delivered, R3[1] becomes 1.
- **Pros:** Lower overhead, more efficient than BSS.
- **Cons:** Approximates causality, can introduce unnecessary delays.

4. ***Matrix Clocks: Detailed Review***

- **Working Diagram**



Tool Used: Mermaid.iod

-
- **Core Idea:** Matrix clocks provide a complete representation of causal relationships. Each process maintains a matrix that tracks the knowledge of events at all other processes.
- **Mechanism:**

  - Each process Pi maintains a matrix Mi.
  - Mi[j,k] represents Pi's knowledge of the latest event at Pk that was causally preceded by an event at Pj.
  - When Pi sends a message m, it attaches Mi to m.
  - When Pj receives m with matrix clock Mm, it updates Mj as follows:
    - Mj[k,l]=max(Mj[k,l],Mm[k,l]) for all k,l.
    - Mj[j,k]=Mj[k,k] for all k.
  - A message is deliverable when Mj[sender,sender]=Mj[sender,reciever]+1
- **Example:**
  - Processes: P1, P2.
  - Initial Matrixes: M1=[[0,0],[0,0]], M2=[[0,0],[0,0]].
  - P1 sends m1 to P2: Mm1=[[1,0],[0,0]], M1 becomes [[1,0],[0,0]].
  - P2 receives m1: M2 becomes [[1,0],[0,0]], M2 becomes [[1,0],[1,0]]. m1 is delivered.
  - P2 sends m2 to P1: Mm2=[[1,0],[1,1]], M2 becomes [[1,0],[1,1]].
  - P1 receives m2: M1 becomes [[1,0],[1,1]], M1 becomes [[1,1],[1,1]]. m2 is delivered.
- **Pros:** Provides complete causal ordering information.
- **Cons:** High storage and computational overhead

<u>How Causal Message Ordering is Ensured:</u>

1. ***<u>Birman-Schiper-Stephenson (BSS):</u>***

- Vector Clocks: BSS uses vector clocks to explicitly track the causal history of each process.

Each element in the vector represents a process, and the value indicates the number of events that process has seen.

- Delivery Condition: A message is delivered only when the receiving process's vector clock shows that it has seen all the events that the sending process has seen *and* that the message is the next expected event from the sending process.

- Buffering: If the delivery condition is not met, the message is buffered until the missing causal dependencies are satisfied. This ensures that messages are delivered in the order of their causal dependencies**.**

2. ***<u>Schwarz & Mattern (SES):</u>***

- Sequence Numbers: SES uses sequence numbers to approximate causal ordering. Each process assigns a sequence number to its messages.
- Expected Sequence: A receiving process keeps track of the last sequence number received from each sending process.
- Delivery Condition: A message is delivered only if its sequence number is the next expected sequence number from the sending process.
- Potential Causality: SES ensures "potential causality," meaning that if a message is delivered, it is likely that all causally preceding messages have been delivered. However, it can sometimes introduce unnecessary delays.

3. Matrix Clocks:

- Matrix Representation: Matrix clocks provide a complete representation of causal relationships using a matrix. Each element M[i, j] represents process i's knowledge of the latest event at process j.
- Matrix Updates: When a message is received, the receiving process updates its matrix by taking the element-wise maximum of its current matrix and the received matrix, and then updating its own row to reflect it's knowledge.
- Delivery Condition: A message is deliverable when the recieving processes row in the matrix clock shows that all preceding messages from the sending process have been recived. This ensures that the message is delivered only after all causally preceding messages have been received.

Comparison of Efficiency and Complexity:

Feature	**BSS**	**SES**	**Matrix Clocks**
Causal Ordering Accuracy	**Strict**	**Potential**	**Complete**
Efficiency	**Lower**	**Higher**	**Lowest**
Time Complexity (Message Delivery)	**O(n)**	**O(1)**	**O(n^2)**
Space Complexity (Process State)	**O(n)**	**O(n)**	**O(n^2)**
Message Overhead	**O(n)**	**O(1)**	**O(n^2)**

Buffering		**Frequent**	**Less frequent**	**Less frequent**
Network Overhead		**Moderate**	**Low**	**High**

Explanation:

- ***BSS:***
    - Provides strict causal ordering but incurs high overhead due to vector clock comparisons and buffering.
    - Its time and space complexity scales linearly with the number of processes.
- ***SES:***
    - Offers a more efficient approach using sequence numbers but only ensures potential causality.
    - It has constant time complexity for message delivery and linear space complexity.
- ***Matrix Clocks:***
    - Provides complete causal ordering information but has the highest overhead due to matrix operations.
    - Its time and space complexity scales quadratically with the number of processes.