Introduction of Process Synchronization

Process Synchronization is the coordination of execution of multiple processes in a multi-process system to ensure that they access shared resources in a controlled and predictable manner. It aims to resolve the problem of race conditions and other synchronization issues in a concurrent system.
The main objective of process synchronization is to ensure that multiple processes access shared resources without interfering with each other and to prevent the possibility of inconsistent data due to concurrent access. To achieve this, various synchronization techniques such as semaphores, monitors, and critical sections are used.

What is Process?

A process is a program that is currently running or a program under execution is called a process. It includes the program's code and all the activity it needs to perform its tasks, such as using the CPU, memory, and other resources. Think of a process as a task that the computer is working on, like opening a web browser or playing a video.

Types of Process

On the basis of synchronization, processes are categorized as one of the following two types:

- **Independent Process**: The execution of one process does not affect the execution of other processes.
- **Cooperative Process**: A process that can affect or be affected by other processes executing in the system.

What is Race Condition?

When more than one process is executing the same code or accessing the same memory or any shared variable in that condition there is a possibility that the output or the value of the shared variable is wrong so for that all the processes doing the race to say that my output is correct this condition known as a race condition. Several processes access and process the manipulations over the same data concurrently, and then the outcome depends on the particular order in which the access takes place. A race condition is a situation that may occur inside a critical section. This happens when the result of multiple thread execution in the critical section differs according to the order in which the threads execute. Race conditions in critical sections can be avoided if the critical section is treated as an atomic instruction. Also, proper thread synchronization using locks or atomic variables can prevent race conditions.

Example

Let's say there are two processes P1 and P2 which share a common variable (shared=10), both processes are present in – queue and waiting for their turn to be executed. Suppose, Process P1 first come under execution, and the CPU store a common variable between them (shared=10) in the local variable (X=10) and increment it by 1(X=11), after then when the CPU read line sleep(1),it switches from current process P1 to process P2 present in ready-queue. The process P1 goes in a waiting state for 1 second.

Now CPU execute the Process P2 line by line and store common variable (Shared=10) in its local variable (Y=10) and decrement Y by 1(Y=9), after then when CPU read sleep(1), the current process P2 goes in waiting for state and CPU remains idle for some time as there is no process in ready-queue, after completion of 1 second of process P1 when it comes in ready-queue, CPU takes the process P1 under execution and execute the remaining line of code (store the local variable (X=11) in common variable (shared=11)), CPU remain idle for sometime waiting for any process in ready-queue,after completion of 1 second of Process P2, when process P2 comes in ready-queue, CPU start executing the further remaining line of Process P2(store the local variable (Y=9) in common variable (shared=9)).

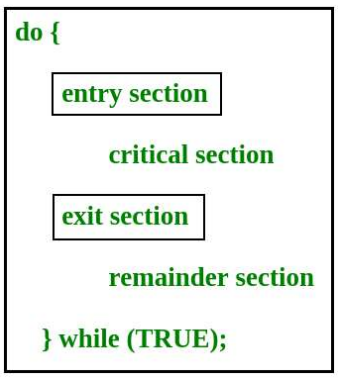
Initially Shared = 10

Process 1	**Process 2**
int X = shared	int Y = shared
X++	Y–
sleep(1)	sleep(1)
shared = X	shared = Y

Note: We are assuming the final value of a common variable(shared) after execution of Process P1 and Process P2 is 10 (as Process P1 increment variable (shared=10) by 1 and Process P2 decrement variable (shared=11) by 1 and finally it becomes shared=10). But we are getting undesired value due to a lack of proper synchronization.

Critical Section Problem

A critical section is a code segment that can be accessed by only one process at a time. The critical section contains shared variables that need to be synchronized to maintain the consistency of data variables. So the critical section problem means designing a way for cooperative processes to access shared resources without creating data inconsistencies.



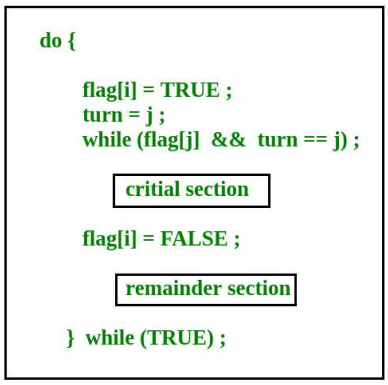
In the entry section, the process requests for entry in the **Critical Section.**
Any solution to the critical section problem must satisfy three requirements:

- **Mutual Exclusion**: If a process is executing in its critical section, then no other process is allowed to execute in the critical section.
- **Progress**: If no process is executing in the critical section and other processes are waiting outside the critical section, then only those processes that are not executing in their remainder section can participate in deciding which will enter the critical section next, and the selection can not be postponed indefinitely.
- **Bounded Waiting**: A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Peterson's Solution

Peterson's Solution is a classical software-based solution to the critical section problem. In Peterson's solution, we have two shared variables:

- boolean flag[i]: Initialized to FALSE, initially no one is interested in entering the critical section
- int turn: The process whose turn is to enter the critical section.



Peterson's Solution preserves all three conditions

- **Mutual Exclusion** is assured as only one process can access the critical section at any time.
- **Progress** is also assured, as a process outside the critical section does not block other processes from entering the critical section.
- **Bounded Waiting** is preserved as every process gets a fair chance.

Disadvantages of Peterson's Solution

- It involves busy waiting. (In the Peterson's solution, the code statement- "while(flag[j] && turn == j);" is responsible for this. Busy waiting is not favored because it wastes CPU cycles that could be used to perform other tasks.)
- It is limited to 2 processes.
- Peterson's solution cannot be used in modern CPU architectures.

Advantages of Process Synchronization

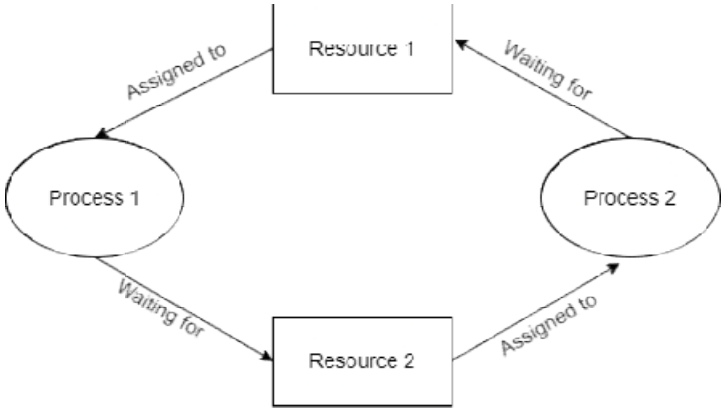
- Ensures data consistency and integrity
- Avoids race conditions
- Prevents inconsistent data due to concurrent access
- Supports efficient and effective use of shared resources

Disadvantages of Process Synchronization

- Adds overhead to the system
- This can lead to performance degradation
- Increases the complexity of the system
- Can cause deadlock if not implemented properly.

Distributed System – Types of Distributed Deadlock

A Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource occupied by some other process. When this situation arises, it is known as Deadlock.



Deadlock

A Distributed System is a Network of Machines that can exchange information with each other through Message-passing. It can be very useful as it helps in resource sharing. In such an environment, if the sequence of resource allocation to processes is not controlled, a deadlock may occur. In principle, deadlocks in distributed systems are similar to deadlocks in centralized systems.

Three commonly used strategies to handle deadlocks are as follows:

- **Avoidance:** Resources are carefully allocated to avoid deadlocks.
- **Prevention:** Constraints are imposed on the ways in which processes request resources in order to prevent deadlocks.
- **Detection and recovery:** Deadlocks are allowed to occur and a detection algorithm is used to detect them. After a deadlock is detected, it is resolved by certain means.

Types of Distributed Deadlock:
There are two types of Deadlocks in Distributed System:

Resource Deadlock: A resource deadlock occurs when two or more processes wait permanently for resources held by each other.

- A process that requires certain resources for its execution, and cannot proceed until it has acquired **all** those resources.
- It will only proceed to its execution when it has acquired all required resources.
- It can also be represented using **AND** condition as the process will execute only if it has all the required resources.
- Example: Process 1 has R1, R2, and requests resources R3. It will not execute if any one of them is missing. It will proceed only when it acquires all requested resources i.e. R1, R2, and R3.

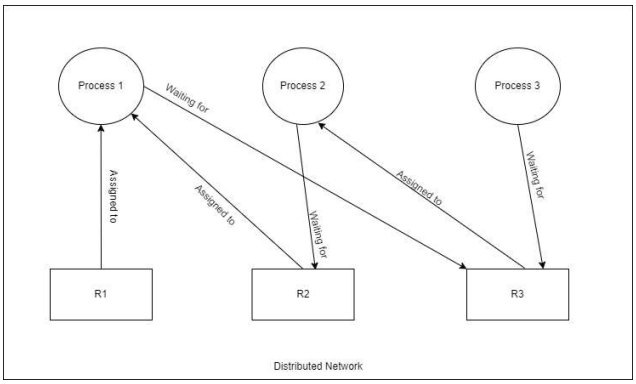


figure 1: Resource Deadlock

Communication Deadlock: On the other hand, a communication deadlock occurs among a set of processes when they are blocked waiting for messages from other processes in the set in order to start execution but there are no messages in transit between them. When there are no messages in transit between any pair of processes in the set, none of the processes will ever receive a message. This implies that all processes in the set are deadlocked. Communication deadlocks can be easily modeled by using WFGs to indicate which processes are waiting to receive messages from which other processes. Hence, the detection of communication deadlocks can be done in the same manner as that for systems having only one unit of each resource type.

- In Communication Model, a Process requires resources for its execution and proceeds when it has acquired **at least one** of the resources it has requested for.
- Here resource stands for a process to communicate with.
- Here, a Process waits for communicating with another process in a set of processes. In a situation where each process in a set, is waiting to communicate with another process which itself is waiting to communicate with some other process, this situation is called communication deadlock.
- For 2 processes to communicate, each one should be in the unblocked state.
- It can be represented using **OR** conditions as it requires at least one of the resources to continue its Process.

- Example: In a Distributed System network, Process 1 is trying to communicate with Process 2, Process 2 is trying to communicate with Process 3 and Process 3 is trying to communicate with Process 1. In this situation, none of the processes will get unblocked and a communication deadlock occurs.

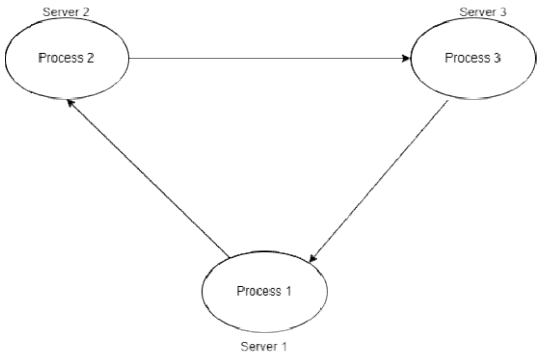


figure 2: Communication Deadlock

Architecture Styles in Distributed Systems

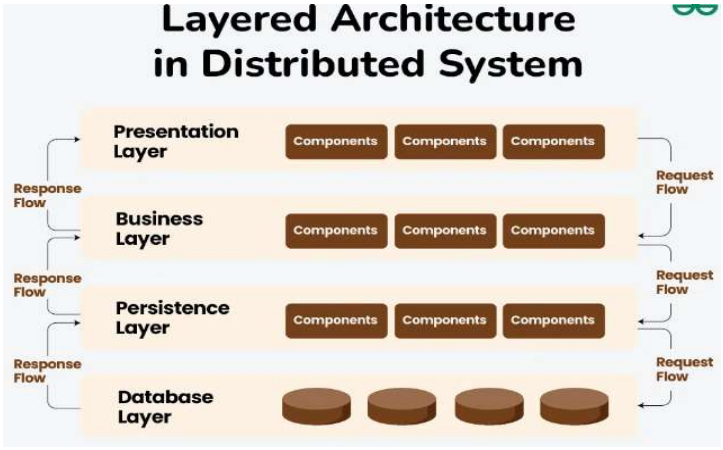
Architecture styles in distributed systems define how components interact and are structured to achieve scalability, reliability, and efficiency. This article explores key architecture styles—including Peer-to-Peer, SOA, and others—highlighting their concepts, advantages, and applications in building robust distributed systems.

Architecture Styles in Distributed Systems

1. Layered Architecture in Distributed Systems

Layered Architecture in distributed systems organizes the system into hierarchical layers, each with specific functions and responsibilities. This design pattern helps manage complexity and promotes separation of concerns. Here's a detailed explanation:

- In a layered architecture, the system is divided into distinct layers, where each layer provides specific services and interacts only with adjacent layers.
- This separation helps in managing and scaling the system more effectively.
-



Layers and Their Functions

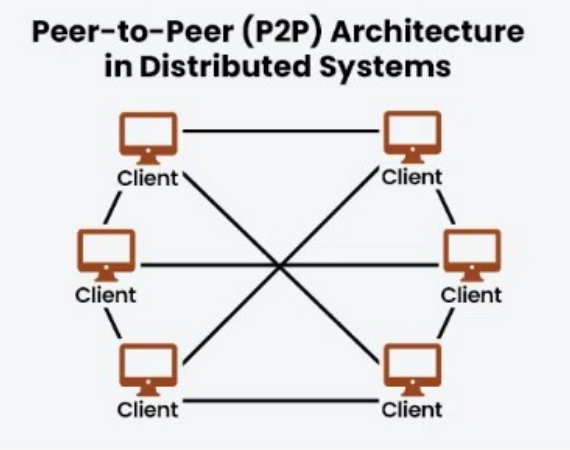
- **Presentation Layer**
    - **Function**: Handles user interaction and presentation of data. It is responsible for user interfaces and client-side interactions.
    - **Responsibilities**: Rendering data, accepting user inputs, and sending requests to the underlying layers.
- **Application Layer**
    - **Function**: Contains the business logic and application-specific functionalities.

2. Peer-to-Peer (P2P) Architecture in Distributed Systems

Peer-to-Peer (P2P) Architecture is a decentralized network design where each node, or "peer," acts as both a client and a server, contributing resources and services to the network. This architecture contrasts with traditional client-server models, where nodes have distinct roles as clients or servers.

- In a P2P architecture, all nodes (peers) are equal participants in the network, each capable of initiating and receiving requests.
- Peers collaborate to share resources, such as files or computational power, without relying on a central server.

    - **Responsibilities**: Processes requests from the presentation layer, executes business rules, and provides responses back to the presentation layer.
- **Middleware Layer**
    - **Function**: Facilitates communication and data exchange between different components or services.
    - **Responsibilities**: Manages message passing, coordination, and integration of various distributed components.
- **Data Access Layer**
    - **Function**: Manages data storage and retrieval from databases or other data sources.
    - **Responsibilities**: Interacts with databases or file systems, performs data queries, and ensures data integrity and consistency.



Peer-to-Peer (P2P) Architecture

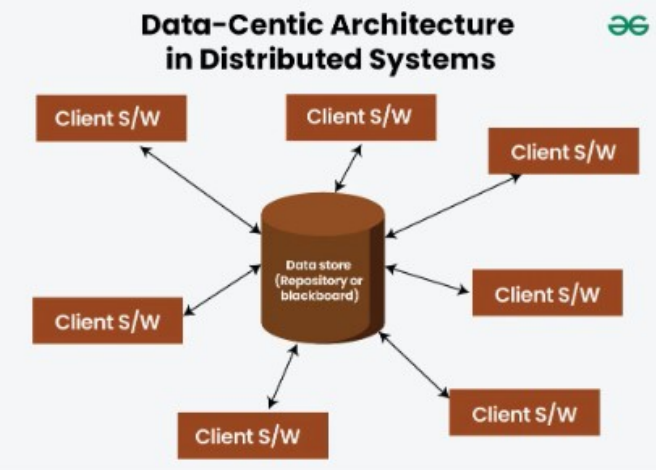
Key Features of Peer-to-Peer (P2P) Architecture in Distributed Systems

- **Decentralization**
    - **Function**: There is no central server or authority. Each peer operates independently and communicates directly with other peers.
    - **Advantages**: Reduces single points of failure and avoids central bottlenecks, enhancing robustness and fault tolerance.
- **Resource Sharing**
    - **Function**: Peers share resources such as processing power, storage space, or data with other peers.
    - **Advantages**: Increases resource availability and utilization across the network.
- **Scalability**
    - **Function**: The network can scale easily by adding more peers. Each new peer contributes additional resources and capacity.
    - **Advantages**: The system can handle growth in demand without requiring significant changes to the underlying infrastructure.
- **Self-Organization**
    - **Function**: Peers organize themselves and manage network connections dynamically, adapting to changes such as peer arrivals and departures.
    - **Advantages**: Facilitates network management and resilience without central coordination.

3. Data-Centic Architecture in Distributed Systems

Data-Centric Architecture is an architectural style that focuses on the central management and utilization of data. In this approach, data is treated as a critical asset, and the system is designed around data management, storage, and retrieval processes rather than just the application logic or user interfaces.

- The core idea of Data-Centric Architecture is to design systems where data is the primary concern, and various components or services are organized to support efficient data management and manipulation.
- Data is centrally managed and accessed by multiple applications or services, ensuring consistency and coherence across the system.



Data-Centic Architecture

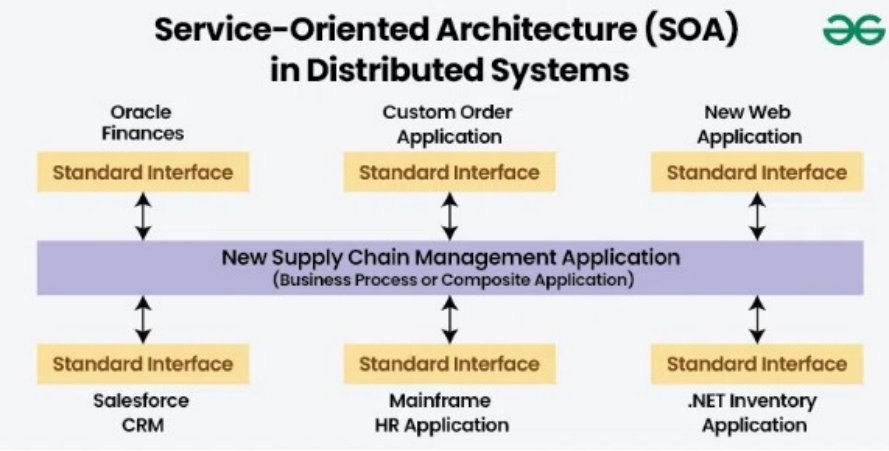
Key Principles of Data-Centic Architecture in Distributed Systems

- **Centralized Data Management**:
    - **Function**: Data is managed and stored in a central repository or database, making it accessible to various applications and services.
    - **Principle**: Ensures data consistency and integrity by maintaining a single source of truth.
- **Data Abstraction**:
    - **Function**: Abstracts the data from the application logic, allowing different services or applications to interact with data through well-defined interfaces.
    - **Principle**: Simplifies data access and manipulation while hiding the underlying complexity.
- **Data Normalization**:
    - **Function**: Organizes data in a structured manner, often using normalization techniques to reduce redundancy and improve data integrity.
    - **Principle**: Enhances data quality and reduces data anomalies by ensuring consistent data storage.

4. Service-Oriented Architecture (SOA) in Distributed Systems

Service-Oriented Architecture (SOA) is a design paradigm in distributed systems where software components, known as "services," are provided and consumed across a network. Each service is a discrete unit that performs a specific business function and communicates with other services through standardized protocols.

- In SOA, the system is structured as a collection of services that are loosely coupled and interact through well-defined interfaces. These services are independent and can be developed, deployed, and managed separately.
- They communicate over a network using standard protocols such as HTTP, SOAP, or REST, allowing for interoperability between different systems and technologies.



Service-Oriented Architecture (SOA)

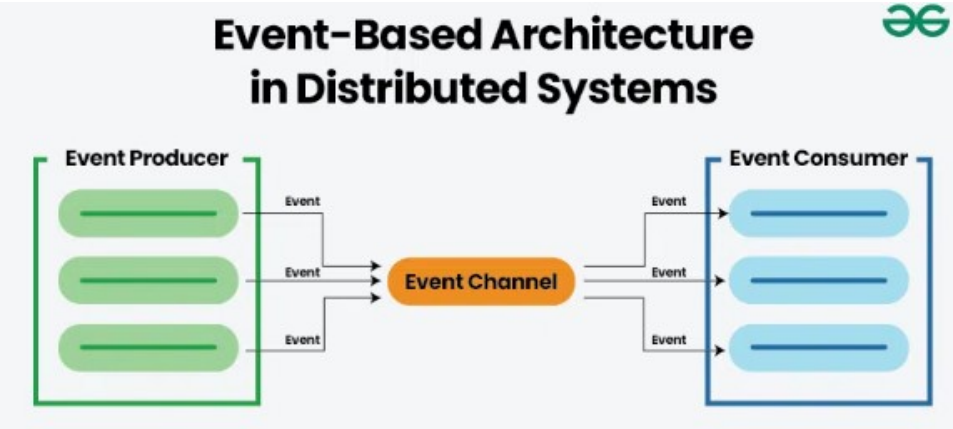
Key Principles of Service-Oriented Architecture (SOA) in Distributed Systems

- **Loose Coupling**:
    - **Function**: Services are designed to be independent, minimizing dependencies on one another.

    - **Principle**: Changes to one service do not affect others, enhancing system flexibility and maintainability.
- **Service Reusability**:
    - **Function**: Services are created to be reused across different applications and contexts.
    - **Principle**: Reduces duplication of functionality and effort, improving efficiency and consistency.
- **Interoperability**:
    - **Function**: Services interact using standardized communication protocols and data formats, such as XML or JSON.
    - **Principle**: Facilitates communication between diverse systems and platforms, enabling integration across heterogeneous environments.
- **Discoverability**:
    - **Function**: Services are registered in a service directory or registry where they can be discovered and invoked by other services or applications.
    - **Principle**: Enhances system flexibility by allowing dynamic service discovery and integration.

5. Event-Based Architecture in Distributed Systems

Event-Driven Architecture (EDA) is an architectural pattern where the flow of data and control in a system is driven by events. Components in an EDA system communicate by producing and consuming events, which represent state changes or actions within the system.

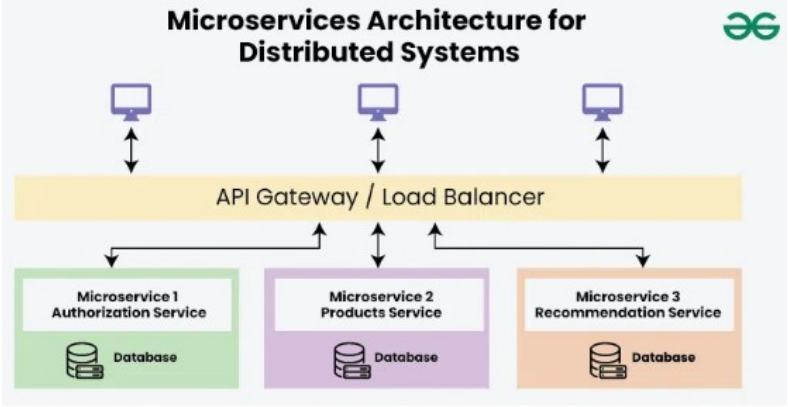


Key Principles of Event-Based Architecture in Distributed Systems

- **Event Producers**: Components or services that generate events to signal state changes or actions.
- **Event Consumers**: Components or services that listen for and react to events, processing them as needed.
- **Event Channels**: Mechanisms for transmitting events between producers and consumers, such as message queues or event streams.
- **Loose Coupling**: Producers and consumers are decoupled, interacting through events rather than direct calls, allowing for more flexible system interactions.

6. Microservices Architecture for Distributed Systems

Microservices Architecture is a design pattern where an application is composed of small, independent services that each perform a specific function. These services are loosely coupled and interact with each other through lightweight communication protocols, often over HTTP or messaging queues.



Microservices Architecture

Key Principles of Microservices Architecture for Distributed Systems

- **Single Responsibility**: Each microservice focuses on a single business capability or function, enhancing modularity.
- **Autonomy**: Microservices are independently deployable and scalable, allowing for changes and updates without affecting other services.
- **Decentralized Data Management**: Each microservice manages its own data, reducing dependencies and promoting scalability.
- **Inter-service Communication**: Services communicate through well-defined APIs or messaging protocols.

Logical Clock in Distributed System

Logical clocks are a concept used in distributed systems to order events without relying on physical time synchronization. They provide a way to establish a partial ordering of events based on causality rather than real-time clock values.

- By assigning logical timestamps to events, logical clocks allow distributed systems to maintain consistency and coherence across different nodes, despite varying clock speeds and network delays.
- This ensures that events can be correctly ordered and coordinated, facilitating fault tolerance and reliable operation in distributed computing environments.

Types of Logical Clocks in Distributed System

1. Lamport Clocks

Lamport clocks provide a simple way to order events in a distributed system. Each node maintains a counter that increments with each event. When nodes communicate, they update their counters based on the maximum value seen, ensuring a consistent order of events.

Characteristics of Lamport Clocks:

- **Simple to implement**.
- **Provides a total order of events** but doesn't capture concurrency.

- **Not suitable for detecting causal relationships** between events.

Algorithm of Lamport Clocks:

1. **Initialization**: Each node initializes its clock LLL to 0.
2. **Internal Event**: When a node performs an internal event, it increments its clock LLL.
3. **Send Message**: When a node sends a message, it increments its clock LLL and includes this value in the message.
4. **Receive Message**: When a node receives a message with timestamp T: **It sets L=max (L,T)+1**

2. Vector Clocks

Vector clocks use an array of integers, where each element corresponds to a node in the system. Each node maintains its own vector clock and updates it by incrementing its own entry and incorporating values from other nodes during communication.

Characteristics of Vector Clocks:

- **Captures causality and concurrency** between events.
- Requires **more storage and communication overhead** compared to Lamport clocks.

Algorithm of Vector Clocks:

1. **Initialization**: Each node PiP_iPi initializes its vector clock ViV_iVi to a vector of zeros.
2. **Internal Event**: When a node performs an internal event, it increments its own entry in the vector clock Vi[i]V_i[i]Vi[i].
3. **Send Message**: When a node PiP_iPi sends a message, it includes its vector clock ViV_iVi in the message.
4. **Receive Message**: When a node PiP_iPi receives a message with vector clock Vj:
   - It updates each entry: **Vi[k]=max (Vi[k],Vj[k])**
   - It increments its own entry: **Vi[i]=Vi[i]+1**

3. Matrix Clocks

Matrix clocks extend vector clocks by maintaining a matrix where each entry captures the history of vector clocks. This allows for more detailed tracking of causality relationships.

Characteristics of Matrix Clocks:

- **More detailed tracking of event dependencies**.
- **Higher storage and communication overhead** compared to vector clocks.

Algorithm of Matrix Clocks:

1. **Initialization**: Each node PiP_iPi initializes its matrix clock MiM_iMi to a matrix of zeros.
2. **Internal Event**: When a node performs an internal event, it increments its own entry in the matrix clock Mi[i][i]M_i[i][i]Mi[i][i].
3. **Send Message**: When a node PiP_iPi sends a message, it includes its matrix clock MiM_iMi in the message.
4. **Receive Message**: When a node PiP_iPi receives a message with matrix clock Mj:
   - It updates each entry: **Mi[k][l]=max (Mi[k][l],Mj[k][l])**
   - It increments its own entry: **Mi[i][i]=Mi[i][i]+1**

4. Hybrid Logical Clocks (HLCs)

Hybrid logical clocks combine physical and logical clocks to provide both causality and real-time properties. They use physical time as a base and incorporate logical increments to maintain event ordering.

Characteristics of Hybrid Logical Clocks:

- **Combines real-time accuracy with causality**.
- **More complex to implement** compared to pure logical clocks.

Algorithm of Hybrid Logical Clocks:

1. **Initialization**: Each node initializes its clock HHH with the current physical time.
2. **Internal Event**: When a node performs an internal event, it increments its logical part of the HLC.
3. **Send Message**: When a node sends a message, it includes its HLC in the message.
4. **Receive Message**: When a node receives a message with HLC T:
   - It updates its **H = max (H,T)+1**

5. Version Vectors

Version vectors track versions of objects across nodes. Each node maintains a vector of version numbers for objects it has seen.

Characteristics of Version Vectors:

- **Tracks versions of objects**.
- **Similar to vector clocks**, but specifically for versioning.

Algorithm of Version Vectors:

1. **Initialization**: Each node initializes its version vector to zeros.
2. **Update Version**: When a node updates an object, it increments the corresponding entry in the version vector.
3. **Send Version**: When a node sends an updated object, it includes its version vector in the message.
4. **Receive Version**: When a node receives an object with a version vector:
   - It updates its version vector to the maximum values seen for each entry.

Vector Clocks

Vector clocks are a mechanism used in distributed systems to track the causality and ordering of events across multiple nodes or processes. Each process in the system maintains a vector of logical clocks, with each element in the vector representing the state of that process's clock. When events occur, these clocks are incremented, and the vectors are exchanged and updated during communication between processes.

- The key idea behind vector clocks is that they allow a system to determine whether one event happened before another, whether two events are concurrent, or whether they are causally related.
- This is particularly useful in distributed systems where a global clock is not available, and processes need to coordinate actions without central control.

By comparing vector clocks, the system can identify if an event on one node causally happened before, after, or concurrently with an event on another node, enabling effective conflict resolution and ensuring consistency.

Use Cases of Vector Clocks in Distributed Systems

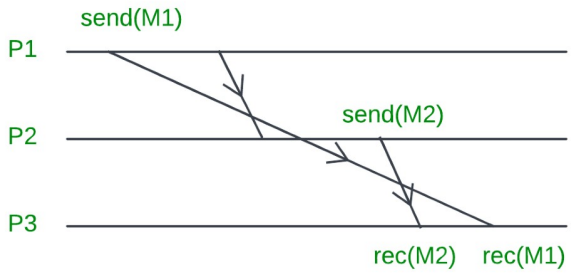
Vector clocks have several important use cases in distributed systems, particularly in scenarios where tracking the order of events and understanding causality is critical. Here are some key use cases:

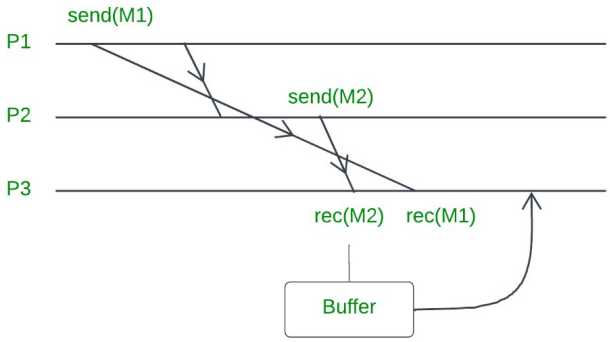
- **Conflict Resolution in Distributed Databases**: In distributed databases like Amazon DynamoDB or Cassandra, vector clocks are used to resolve conflicts when different replicas of data are updated independently.
- **Version Control in Collaborative Editing**: In collaborative editing tools (e.g., Google Docs), multiple users can edit the same document simultaneously.
- **Detecting Causality in Event-Driven Systems**: In event-driven systems where the order of events is crucial, such as in distributed logging or monitoring systems.
- **Distributed Debugging and Monitoring**: When debugging or monitoring distributed systems, understanding the order of operations across different nodes is essential.
- **Ensuring Consistency in Distributed File Systems**: In distributed file systems like Google File System (GFS) or Hadoop Distributed File System (HDFS), multiple clients may access and modify files concurrently.
- **Concurrency Control in Distributed Transactions**: In distributed transaction processing, ensuring that transactions are processed in the correct order across different nodes.
- **Coordination of Distributed Systems:** In systems that require coordination across distributed components, such as microservices architectures.

Causal Ordering of Messages

Causal ordering of messages is one of the four semantics of multicast communication namely unordered, totally ordered, causal, and sync-ordered communication. Multicast communication methods vary according to the message's reliability guarantee and ordering guarantee. The causal ordering of messages describes the causal relationship between a message send event and a message receive event.

For example, if send(M1) -> send(M2) then every recipient of both the messages M1 and M2 must receive the message M1 before receiving the message M2. In Distributed Systems the causal ordering of messages is not automatically guaranteed.





Reasons that may lead to violation of causal ordering of messages

1. It may happen due to a transmission delay.
2. Congestion in the network.
3. Failure of a system.

Protocols that are used to provide causal ordering of messages

1. Birman Schipher Stephenson Protocol
2. Schipher Eggli Sandoz Protocol

Both protocol's algorithm requires that the messages be delivered reliably and both prefer that there is no network partitioning between the systems. The general idea of both protocols is to deliver a message to a process only if the message immediately preceding it has been delivered to the process. Otherwise, the message is not delivered immediately instead it is stored in a buffer memory until the message preceding it has been delivered.

The ISIS System

The ISIS system was developed by Ken Birman and Joseph in 1987 and 1993. It is a framework for reliable distributed communication which is achieved through the help of process groups. It is a programming toolkit whose basic features consist of process group management calls and ordered multicast primitives for communicating with the process group members. ISIS provides multicast facilities such as unordered multicast (FBCAST), casually ordered multicast (CBCAST), totally ordered multicast (ABCAST), and sync-ordered multicast (GBCAST).

The ISIS CBCAST Protocol

ISIS uses vector timestamps to implement causally ordered multicast between the members of a process group. It is assumed that all the messages are multicast to all the members of the group including the sender. ISIS uses UDP/IP protocol as its basic transport facility and sends acknowledgments and retransmits packets as necessary to achieve reliability. Messages from a given member are sequenced and delivered in order. There is no assumption that hardware support for broadcast or multicast exists. If IP multicast is implemented, then ISIS can exploit it to send a single UDP packet to the appropriate multicast address. IP multicast takes

advantage of hardware like ethernet, for multicast facilities. Otherwise, packets are sent point-to-point to the individual group members.

Let the members of the group be p*1*,p*2*,p*3*,…p*n*. Once more we shall define, for each p*i,* a vector timestamp denoted by VT*i* which is used to order multicast delivery. It will turn out that VT[*i*] is the count of multicast messages sent by p*i* that causally lead up to the latest message delivered to p*j*. The following is the vector timestamp update algorithm:

1. All the processes p*i initialize* VT*i* to zeroes.
2. When p*i* multicasts a new message, it first increments VT[*i*] by 1. Then it piggybacks the value vt = VT*i* on the message.
3. When a message bearing a timestamp vt is delivered to p*j,* p*j*'s timestamp is updated as VT*j* = merge(VT*j*, vt).

Chandy–Lamport's global state recording algorithm

Chandy and **Lamport** were the first to propose a algorithm to capture consistent global state of a distributed system. The main idea behind proposed algorithm is that if we know that all message that have been sent by one process have been received by another then we can record the global state of the system.

Any process in the distributed system can initiate this global state recording algorithm using a special message called **MARKER**. This marker traverse the distributed system across all communication channel and cause each process to record its own state. In the end, the state of entire system (Global state) is recorded. This algorithm does not interfere with normal execution of processes.

Assumptions of the algorithm:

- There are finite number of processes in the distributed system and they do not share memory and clocks.
- There are finite number of communication channels and they are unidirectional and FIFO ordered.
- There exists a communication path between any two processes in the system
- On a channel, messages are received in the same order as they are sent.

Algorithm:

- **Marker sending rule for a process *P* :**
  - Process **p** records its own local state
  - For each outgoing channel **C** from process **P**, **P** sends marker along **C** before sending any other messages along **C**.
    (**Note:** Process Q will receive this marker on his incoming channel C1.)
- **Marker receiving rule for a process *Q* :**
  - If process **Q** has not yet recorded its own local state then

Record the state of incoming channel C1 as an empty sequence or null.

After recording the state of incoming channel C1, process **Q** Follows the marker sending rule

  - If process **Q** has already recorded its state

Record the state of incoming channel **C1** as the sequence of messages received along channel C1 after the state of **Q** was recorded and before **Q** received the marker along C1 from process P.

Need of taking snapshot or recording global state of the system:

- **Checkpointing:** It helps in creating checkpoint. If somehow application fails, this checkpoint can be reused
- **Garbage collection:** It can be used to remove objects that do not have any references.
- It can be used in deadlock and termination detection.
- It is also helpful in other debugging.

Huang's Termination detection algorithm

Huang's algorithm is an algorithm for detecting termination in a distributed system. The algorithm was proposed by **Shing-Tsaan Huang** in 1989 in the Journal of Computers. In a distributed system, a process is either in an active state or in an idle state at any given point of time. Termination occurs when all of the processes becomes idle and there are no any in transit(on its way to be delivered) computational message. **Assumptions of the algorithm:**

- One of the co-operating processes which monitors the computation is called the **controlling agent**.
- The initial weight of **controlling agent** is 1
- All other processes are initially idle and have weight 0.
- The computation starts when the controlling agent send a computation message to one of the processes.
- The process become active on receiving a computation message.
- Computation message can be sent only by controlling agent or an active process.
- Control message is sent to controlling agent by an active process when they are becoming idle.
- The algorithm assigns a weight **W** (such that $0 < \mathbf{W} < 1$) to every active process and every in transit message.

Notations used in the algorithm:

- **B(DW):** Computation message with weight **DW**
- **C(DW)**: Control message with weight **DW**

Algorithm:

- **Rule to send B(DW) –**
    - Suppose Process P with weight **W** is sending **B(DW)** to process Q
    - Split the weight of the process P into **W1** and **W2**. Such that

$W = W1 + W2$ and $W1 > 0, W2 > 0$

- Set weight of the process P as **W1** (i.e **W = W1**)
- Send **B(W2)** to *process Q*, here **DW = W2**.
- **Note:** Only the Controlling agent or any active process can send Computation message.
- **On receiving B(DW) by process Q –**
    - Add the weight **DW** to the weight of process Q i.e for process Q, **W = W + DW**
    - If process Q was idle, it will become active on receiving **B(DW)**.
- **Rule to send C(DW) –**

    - Any active process having weight **W** can become idle by sending **C(W)** to controlling agent
    - Send a control message **C(W)** to the controlling agent. Here **DW** = **W**.
    - Set weight of the process as 0 i.e **W = 0**. (After this process will become idle.)
- **On receiving C(DW) by controlling agent –**
    - Add the weight received through control message to the weight of controlling agent i.e **W = W + DW**
    - After adding, if the weight of controlling agent becomes 1 then it can be conclude that the computation has terminated.

Mutual exclusion in distributed system

Mutual exclusion is a concurrency control property which is introduced to prevent race conditions. It is the requirement that a process can not enter its critical section while another concurrent process is currently present or executing in its critical section i.e only one process is allowed to execute the critical section at any given instance of time.
Mutual exclusion in single computer system Vs. distributed system: In single computer system, memory and other resources are shared between different processes. The status of shared resources and the status of users is easily available in the shared memory so with the help of shared variable (For example: Semaphores) mutual exclusion problem can be easily solved. In Distributed systems, we neither have shared memory nor a common physical clock and therefore we can not solve mutual exclusion problem using shared variables. To eliminate the mutual exclusion problem in distributed system approach based on message passing is used. A site in distributed system do not have complete information of state of the system due to lack of shared memory and a common physical clock.

Requirements of Mutual exclusion Algorithm:
- **No Deadlock:** Two or more site should not endlessly wait for any message that will never arrive.
- **No Starvation:** Every site who wants to execute critical section should get an opportunity to execute it in finite time. Any site should not wait indefinitely to execute critical section while other site are repeatedly executing critical section
- **Fairness:** Each site should get a fair chance to execute critical section. Any request to execute critical section must be executed in the order they are made i.e Critical section execution requests should be executed in the order of their arrival in the system.
- **Fault Tolerance:** In case of failure, it should be able to recognize it by itself in order to continue functioning without any disruption.

Solution to distributed mutual exclusion: As we know shared variables or a local kernel can not be used to implement mutual exclusion in distributed systems. Message passing is a way to implement mutual exclusion. Below are the three approaches based on message passing to implement mutual exclusion in distributed systems:
1. Token Based Algorithm:
- A unique **token** is shared among all the sites.
- If a site possesses the unique token, it is allowed to enter its critical section
- This approach uses sequence number to order requests for the critical section.

- Each requests for critical section contains a sequence number. This sequence number is used to distinguish old and current requests.
- This approach insures Mutual exclusion as the token is unique

Example : Suzuki–Kasami Algorithm

2. Non-token based approach:

- A site communicates with other sites in order to determine which sites should execute critical section next. This requires exchange of two or more successive round of messages among sites.
- This approach use timestamps instead of sequence number to order requests for the critical section.
- When ever a site make request for critical section, it gets a timestamp. Timestamp is also used to resolve any conflict between critical section requests.
- All algorithm which follows non-token based approach maintains a logical clock. Logical clocks get updated according to Lamport's scheme

Example : Ricart–Agrawala Algorithm

3. Quorum based approach:

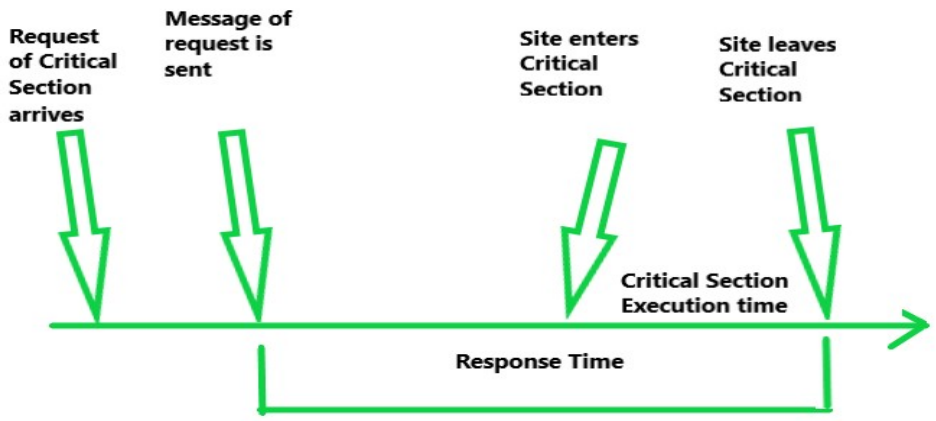
- Instead of requesting permission to execute the critical section from all other sites, Each site requests only a subset of sites which is called a **quorum**.
- Any two subsets of sites or Quorum contains a common site.
- This common site is responsible to ensure mutual exclusion

Performance Metrics for Mutual Exclusion

The criteria of performance of mutual exclusion in a distributed system are measured in the following methods:

1. Response time

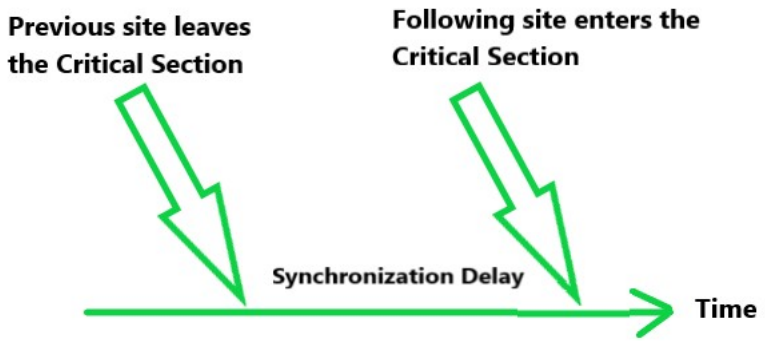
The interval of time when a request waits for the end of its critical section execution after its solicitation messages have been conveyed.



Response Time

2. Synchronization Delay

The time required for the next process to enter the critical section after a process leaves the critical section is known as Synchronization delay.



Synchronization Delay

3. Message complexity

The number of messages needed to execute each critical section by the process.

4. Throughput

Throughput is the amount at which the system executes requests for the critical section.

Throughput = 1/(Synchronization delay + Avg Critical Section execution time)

5. Low and High Load Performance

The amount of request that arrives for critical section execution denotes the load. If more than 1 request is present for the critical section then it is known as Low Load. If there is always a pending request then it is known as High Load. In heavy load conditions, after a request is executed, a site promptly starts activities to execute its next demand of Critical Section. A site is only occasionally in the inactive state in heavy load conditions. For some mutual exclusion algorithms, the performance metrics can be registered effectively under low and heavy loads through simple mathematical reasoning.