Introduction to distributed system

Chapter – 1

 # What is distributed system?

A Distributed System is a collection of autonomous computers interconnected through a communication network. The information (knowledge) between the computers is exchanged only through messages over a network.

 # why and how to build and use distributed systems (DS)

Why Build Distributed Systems:

1. **Low-Cost Connectivity:** Thanks to technological advancements, it's now cheaper and easier to connect computers together.
2. **Mobile Computing:** We can now have powerful computing on mobile devices at an affordable cost.
3. **Connected Environments:** More and more environments, like homes and offices, are becoming smart and connected.
4. **Business Needs:** Businesses have been using distributed systems for years to meet their requirements.
5. **Consumer Demand:** Recently, everyday people are using distributed systems more in their daily lives.

How to Build and Use Distributed Systems:

1. **Common Problems:** Many of the problems we face with single computers are still there with distributed systems.
2. **Infrastructure is Key:** Distributed systems depend on things like networks and services. Without them, remote access won't work.
3. **Connectivity is Vital:** To use distributed systems remotely, you need a good internet connection.
4. **Unique Challenges:** Distributed systems have their own set of issues because they involve multiple computers working together.
5. **Unsolvable Problems:** Some problems in distributed systems are so complex that we can't completely solve them.

We build distributed systems because it's now cheaper and easier to connect computers, and we use them because they're becoming a part of our daily lives. However, building and using them can be tricky because they come with their own set of challenges, especially when it comes to making sure everything works smoothly when computers are far apart. Some problems in distributed systems are so tough that we can't solve them completely.

 # Peer-2-Peer Networks for Transactions:

Peer-to-Peer (P2P) networks for transactions allow individuals to exchange money or data directly without a middleman.

Example: Bitcoin is a P2P network where users can send digital currency (Bitcoins) directly to each other's digital wallets without needing a bank or payment service. Transactions are recorded on a public ledger called the blockchain, ensuring transparency and security.

 # Distributed Systems:

- Definition: A network of interconnected computers working together with a coordinator.
- Example: Cloud computing services like AWS or Google Cloud, where multiple servers collaborate under central control.

 # Decentralized Systems:

- Definition: A network of autonomous computers with no central authority. In simpler terms, it's a system where everyone makes their own decisions based on what they see around them, and the overall outcome is the result of how all these individual decisions interact, rather than someone at the top giving orders.
- Example: Blockchain networks like Bitcoin, where nodes validate and record transactions independently without a central authority.

 # Difference:

- A distributed system is a network of interconnected nodes that work together as a unified system, sharing resources and collaborating to achieve a common goal.
- A decentralized system is one where decision-making and control are distributed among multiple autonomous entities or nodes, operating independently and often using consensus mechanisms.
- Distributed systems can have centralized control and decision-making, while decentralized systems distribute control and decision-making among nodes.
- Distributed systems focus on scalability by distributing tasks and data, while decentralized systems focus on fault tolerance and avoiding single points of failure.
- Distributed systems often follow client-server or service-oriented architectures, while decentralized systems can have various architectural patterns like peer-to-peer or blockchain networks.

It's worth noting that these terms can overlap, and some systems can exhibit characteristics of both distributed and decentralized architectures.

 # Ubiquitous Computing:

Ubiquitous computing is about making everyday objects and environments smarter and more responsive by embedding computing and communication capabilities into them, enhancing our daily experiences and efficiency.

Example: Imagine a "smart home" where various devices like thermostats, lights, and security cameras are interconnected and aware of your presence and preferences. They can adjust the temperature, lighting, and security settings automatically based on your habits and the time of day. This creates a more comfortable and efficient living environment without you having to manually control each device.

 # Edge vs Fog computing:

Edge computing and fog computing both aim to bring computation closer to where data is generated to reduce latency and improve efficiency. Edge computing focuses on ultra-low latency and immediate decision-making, while fog computing adds a layer of intermediate processing for more complex tasks and localized data analysis. Both paradigms are important for the growing Internet of Things (IoT) ecosystem and real-time applications.

Edge Computing:

- Processes data at the very edge (e.g., IoT devices).
- Minimizes latency for instant decision-making.
- Ideal for real-time applications like autonomous vehicles.

Fog Computing:

- Adds an intermediate layer between edge and cloud.
- Balances latency and complexity.
- Suited for applications needing localized processing and some centralization (e.g., smart cities).

In a nutshell, edge computing is ultra-fast for immediate decisions, while fog computing strikes a balance between local and centralized processing for more complex tasks.

 # Benefits and limitations of distributed system

Distributed systems offer numerous advantages like scalability, reliability, and performance, but they also come with complexities and challenges related to communication, data consistency, security, and maintenance. The decision to use a distributed system should consider both the benefits and limitations in the context of specific use cases and requirements.

Benefits of Distributed Systems:

1. **Scalability:** Easily handle more work by adding more machines.
2. **Reliability:** Reduce the impact of failures with redundancy.
3. **Performance:** Improve speed by sharing work.
4. **Geographical Reach:** Serve users globally with reduced delay.
5. **Resource Optimization:** Use computing power and storage efficiently.
6. **Data Availability:** Ensure data is always accessible.
7. **Fault Tolerance:** Keep working even if parts fail.
8. **Cost Efficiency:** Save money by using resources effectively.

Limitations of Distributed Systems:

1. **Complexity:** They're harder to design and manage.
2. **Communication Overhead:** Data exchange can slow things down.
3. **Data Consistency:** Ensuring data is consistent can be tricky.
4. **Security:** More vulnerable to attacks, requiring strong defenses.
5. **Maintenance Complexity:** Coordinating updates is complex.
6. **Software Compatibility:** Compatibility can be challenging.
7. **Network Reliability:** Dependence on networks makes them sensitive to failures.
8. **Scalability Challenges:** Rapid scaling can lead to problems.

 # Location-Independent Services:

- Definition: Digital services accessible from anywhere, not tied to a specific location.
- Examples: Cloud email, social media, online collaboration tools.
- Comparison: Work the same for users everywhere, not influenced by location.

 # Location-Based Services:

- Definition: Digital services that use a user's location to provide specific information or functionality.
- Examples: GPS navigation apps, local business finders, geofencing notifications.
- Comparison: Tailor content and services based on the user's current geographic location.

 # Dependencies of Distributed Systems:

1. **Network Infrastructure:** Distributed systems rely on a robust network to connect and communicate between nodes. Network reliability and bandwidth are crucial factors.
2. **Hardware Resources:** Physical servers, computers, and devices form the foundation of distributed systems. Proper hardware setup and maintenance are essential.
3. **Software Components:** These systems involve various software elements, including operating systems, middleware, and application software, all of which must work cohesively.
4. **Communication Protocols:** Distributed systems require well-defined communication protocols and standards to ensure seamless data exchange among distributed components.
5. **Security Measures:** Strong security mechanisms are necessary to protect data and communication within the distributed system from unauthorized access and threats.
6. **Synchronization Mechanisms:** To ensure consistency and coordination, distributed systems employ synchronization mechanisms to manage concurrent processes and data access.

Synchronous Systems:

- Events happen in a coordinated and time-dependent manner.

- Operations follow a predefined order and timing.
- Examples include real-time communication like phone calls.

Asynchronous Systems:

- Events occur independently and are not tightly synchronized.
- Operations can happen at different times.
- Examples include email and distributed systems with non-blocking operations.

In short, synchronous systems follow a strict timing and order, while asynchronous systems allow for flexibility in timing and independence of events.

→ A distributed system is called synchronous when: There exists a system-wide known upper limit for the divergence of local clocks between different nodes of the system.

→ A distributed system is called asynchronous when: One or more of the conditions for a synchronous system are not met.

Chapter – 2

1. Programming Models in Distributed Systems:

- Different programming models include remote procedure call (RPC), message-passing, shared memory, and distributed objects. These models help developers write code for distributed systems.

2. **Shared Memory Model:**

- **Explanation:** Shared memory allows multiple processes to access the same memory space, enabling them to communicate and share data directly.
- **Example:** In a multi-threaded application, threads can access shared data structures like a queue or a buffer, facilitating communication without explicitly passing messages.

3. **Difference Between Shared Memory and Distributed Memory Models:**

- Shared Memory: Processes access the same memory space, suitable for communication within a single machine.
- Distributed Memory: Processes have separate memory spaces, requiring explicit message-passing for communication in a distributed system.

4. **Process Switching:**

- **Definition:** Process switching is the transition of a CPU from one process to another. It involves saving the state of the current process and loading the state of the next.

- **Overhead:** Process switching incurs overhead in terms of time and resources due to the need to save and restore process states.

5. **Difference Between Heavy-Weighted Processes and Light-Weighted Threads:**

- Heavy-Weighted Processes: Independent units of execution with their own memory space. They are more resource-intensive.
- Light-Weighted Threads: Smaller units of execution within a process, sharing the same memory space. They are more resource-efficient.

6. **Example of Threads in Distributed Systems:**

- In a web server, each incoming client request can be processed by a separate thread. Threads can handle multiple client requests concurrently, improving the server's responsiveness.

7. **Architectural Pattern for Always-Available Server:**

- Use a load balancer in front of multiple server instances. If one server fails, the load balancer directs traffic to healthy servers, ensuring continuous availability.

8. **Interaction Paradigms and Hardware Architectures:**

- Different hardware architectures (e.g., shared memory vs. distributed memory) influence interaction paradigms. Shared memory systems may use direct function calls, while distributed systems often rely on message-passing due to separate memory spaces.

9. **Abstractions in Message-Passing (Distributed Memory Models):**

- Abstractions include message queues, remote procedure calls (RPC), and publish-subscribe mechanisms. These allow processes to exchange data and communicate in a distributed environment.

10. **Synchronization and Time Coupling in Message-Passing:**

- Synchronization ensures processes coordinate their actions correctly. Time coupling considerations involve handling differences in clock times between distributed nodes to maintain order and consistency in message delivery.

Chapter – 3

 # Process System Model:

A Process System Model is a framework used to represent how processes interact and execute tasks within a system, aiding in system design, concurrency, communication, and resource management. It helps analyze and design complex software systems.

The "Process System Model" provides a structured framework for understanding and analyzing systems. It comprises two essential aspects:

1. Static Structure:

- This aspect defines the system's static components, which remain fixed throughout its operation.
- It consists of:
    - A finite set of states (Q), including a specific initial state (q0) and a set of final states (QF).
    - A finite set of rules (R) that delineate permissible state transitions or changes (Q × Q).

2. Dynamic Behavior:

- This facet characterizes the system's behavior over time, emphasizing the sequences of steps or actions known as "processes" that the system undergoes.
- A process is a sequence of steps (e.g., q0 → a!1 q1 → a!2 q2 → a!3 q3 → ...), where:
    - Each step involves an action (ai+1) leading to a state change (qi → qi+1), with i in the range [0, 1].
    - All states (qi) within a process are permissible (i.e., they belong to the set Q).
    - Each action in a process adheres to the rules specified in R.
    - Importantly, each action is considered atomic, meaning it is indivisible and cannot be interrupted.

 # Processes as Execution Sequences

Sequential Execution:

- Tasks are done one after the other, in a predefined order.
- Like following a recipe step by step.
- **Example:** When following a recipe to bake a cake, you complete each step (mixing, baking, frosting) one at a time, in a specific order.

Parallel Execution:

- Multiple tasks run at the same time, often on different processors.
- Like different assembly line robots working simultaneously in a car factory to speed up production.
- **Example:** In a car manufacturing plant, different robots weld, paint, and assemble various parts of a car simultaneously, speeding up production.

 # Deterministic Systems:

- Behavior is entirely predictable.
- No randomness or uncertainty.
- Example: Simple mathematical calculations.

Determinate Systems:

- Outcome can be determined to some extent.
- May have some level of uncertainty due to external factors.
- Example: Weather forecasting with external variables like sudden temperature changes.

 # Blocking, Deadlocks, Starvation

Blocking:

- **Definition:** A process is temporarily halted while waiting for a resource or event.
- **Cause:** Can be caused by I/O operations, network communication, or synchronization.
- **Nature:** Temporary and often resolved when the required resource or event becomes available.
- **Example:** Waiting for a response from a remote server before continuing execution.

Deadlocks:

- **Definition:** Multiple processes are mutually blocked, each waiting for a resource held by another.
- **Cause:** Occurs when processes compete for resources and hold some while waiting for others.
- **Nature:** Permanent until resolved externally using techniques like deadlock detection and recovery.
- **Example:** Process A holds Resource X and waits for Resource Y, while Process B holds Resource Y and waits for Resource X.

Starvation:

- **Definition:** A process or resource is consistently denied access or service.
- **Cause:** Can result from resource allocation policies or unfair scheduling.
- **Nature:** Ongoing and may persist unless fairness mechanisms are introduced.
- **Example:** Lower-priority processes continuously delayed by higher-priority ones.

Relationship between These Issues:

- Deadlocks often lead to blocking because processes involved in a deadlock are blocked and cannot proceed.
- Starvation can occur alongside blocking when certain processes are consistently delayed or waiting, even if they eventually make progress.

 # How to detect deadlocks?

A deadlock in a distributed system is a state in which two or more processes or nodes are unable to proceed because each is waiting for the other(s) to release a resource, terminate, or perform some other action.

There are various techniques and algorithms can be used to detect deadlocks in distributed systems:

1. Use a **Resource Allocation Graph (RAG)** to represent resource requests and allocations and check for cycles.
2. Employ a **Wait-for Graph** to show processes waiting for resources; a cycle indicates a deadlock.
3. Implement **Timeouts and Deadlock Detection** to monitor resource requests and process states, triggering detection if requests go unfulfilled for too long.
4. Utilize **Distributed Deadlock Detection Algorithms** for systems with multiple nodes, involving communication to monitor resource states.
5. Consider **Banker's Algorithm** for deadlock detection, which periodically checks resource allocation and requests.
6. Explore **Dynamic Resource Allocation** to dynamically allocate and deallocate resources based on the system's current state.
7. Focus on **Message and Communication Deadlock Detection** in message-passing systems, monitoring message exchanges and infinite waiting.

 # Asynchronous vs. Synchronous Communication:

Asynchronous Communication:

- **Definition:** In asynchronous communication, the sender and receiver do not need to be synchronized in time. The sender sends a message without waiting for an immediate response from the receiver.
- **Nature:** Asynchronous communication is non-blocking. After sending a message, the sender can continue with its tasks without waiting for confirmation or a response.
- **Latency:** It may introduce variable latency, as the receiver can process the message at its own pace.
- **Example:** Email is an example of asynchronous communication. You send an email, and the recipient reads it at their convenience, not necessarily immediately.

Synchronous Communication:

- **Definition:** Synchronous communication requires sender and receiver synchronization. The sender sends a message and waits for an immediate response or acknowledgment from the receiver.
- **Nature:** Synchronous communication is blocking. The sender is paused until it receives a response or confirmation from the receiver.
- **Latency:** It typically has lower latency, as the sender knows when the receiver has received and processed the message.
- **Example:** A phone call is an example of synchronous communication. You speak and listen in real-time, requiring immediate interaction.

 # SMS vs. DMS Synchronization Specification:

→ Shared Memory Systems (SMS):

- Common memory space.
- Synchronization with locks and shared variables.
- Communication through shared memory.

Synchronization Specification in SMS:

- **Mutual Exclusion:** Ensure that only one process accesses a shared resource at a time, often using locks.
- **Memory Consistency:** Define rules for the order in which memory operations are visible to processes to maintain data consistency.
- **Concurrency Control:** Implement mechanisms to manage concurrent access to shared data structures without conflicts.
- **Deadlock Prevention:** Use techniques like deadlock detection and prevention to avoid processes getting stuck.

→ Distributed Memory Systems (DMS):

- Separate memory spaces.
- Synchronization via message-passing.
- Communication through messages.
- Focus on message ordering, distributed locking, and consensus.

Synchronization Specification in DMS:

- **Message Ordering:** Maintain the correct order of messages to ensure consistency and prevent race conditions.
- **Distributed Locking:** Coordinate access to shared resources across distributed nodes using distributed lock managers.
- **Consensus:** Achieve agreement among distributed nodes even in the presence of failures using consensus algorithms.
- **Clock Synchronization:** Ensure synchronized clocks across nodes for accurate timing and coordination.

 # Peterson's Algorithm allows two processes to safely share a critical section:

- Each process has a `flag` and `turn` variable.
- To enter, a process sets its `flag` to `true` and `turn` to the other process's ID.
- It checks if the other process is interested (`flag` is `true`) and it's their turn, then it waits.
- If the other process is not interested or it's their turn, it enters the critical section.
- After exiting, it sets its `flag` to `false`.

Case-Based Analysis:

- P0 wants to enter, sets `flag[0] = true` and `turn = 1`.
- P1 wants to enter, sets `flag[1] = true` and `turn = 0`.
- Both check each other's flags and turn to determine who enters or waits.
- Only one process enters at a time, ensuring mutual exclusion.

De-Coupling using Selective Receives:

In distributed systems (DS), decoupling refers to the ability of processes or components to interact independently without strong dependencies on each other. Selective receives is a communication mechanism that supports decoupling by allowing processes to choose which messages they want to receive, rather than being forced to accept all incoming messages. This selective reception helps in building loosely coupled and flexible distributed systems.

Decoupling using selective receives in distributed systems, there are several programming language solutions to consider:

1. **Thread-per-Port:** Each sender has its dedicated thread, which checks for messages in a round-robin fashion. This approach ensures non-blocking computation processes.
2. **Receive-from-Any:** Multiple sending processes share the same receiving port. Blocking occurs only if no message is found, but this isn't an issue if there's no other work to be done.
3. **Time-Outs:** Programming languages may offer constructs for setting timeouts on receive operations. However, the program's behavior can vary due to changing loads and response times.
4. **Probe Mechanism:** Use a non-blocking "probe" to check for messages without blocking the program. This is suitable for responsive systems.
5. **Guarded Commands:** Implement send and receive operations as guarded commands, allowing for explicit synchronization and control over message exchange, often seen in languages like CSP or Ada.

Ultimately, the choice of approach depends on the specific requirements and characteristics of the distributed system, including its communication patterns and desired level of decoupling and predictability.

Message Passing Synchronization:

1. One-way Synchronization: A process sends a signal and continues its work while expecting a response from another process. For example, triggering a task and waiting for acknowledgment.

2. Mutual Exclusion for Shared Data: Ensures that only one process can write to shared data at a time. This can be achieved through a central server, migrating data, or distributed agreement mechanisms. The goal is to prevent conflicts and maintain data integrity.

Explanation:

- In one-way synchronization, a process initiates an action without waiting for an immediate response. This is like sending a message and expecting a reply later.
- Mutual exclusion for shared data ensures that access to critical data is controlled to prevent concurrent writes, which could lead to data corruption or inconsistencies.
- The mechanisms described (centralized server, migrating server, distributed agreement) all aim to achieve exclusive access to shared data, maintaining data integrity.

These synchronization techniques are fundamental for coordinating processes and ensuring data consistency in distributed systems.

To Overcome Unreliable Transport in Distributed Systems:

1. **Use Acknowledgments and Retransmissions:** Senders wait for acknowledgments from receivers and retransmit if needed.
2. **Implement Sequence Numbers:** Assign unique sequence numbers to messages to maintain order.
3. **Utilize Checksums and Error Detection:** Include checksums to detect message corruption.
4. **Set Timeouts and Heartbeats:** Use timeouts to monitor remote processes or nodes.
5. **Detect and Deduplicate Duplicates:** Track received message IDs to avoid duplicates.
6. **Apply Flow Control:** Prevent sender overload to avoid congestion.

Chapter – 4

Message Queuing Systems:

- **Definition:** Message queuing systems are software solutions that facilitate the exchange of messages between different components or systems in a distributed environment.
- **Key Features:**
    - Decouple sender and receiver.
    - Ensure message durability.
    - Load leveling to prevent overload.

    - Reliable communication with acknowledgment and retry.
    - Scalability for concurrent message consumption.
- **Examples:** Apache Kafka, RabbitMQ, ActiveMQ, and Azure Service Bus.

Importance of Message Queuing:

1. **Integration of Heterogeneous Applications:** Enables seamless communication between diverse, existing applications and middleware.
2. **Server-Side Component Integration:** Facilitates the integration of server-side components like Enterprise JavaBeans (EJBs) in distributed systems.
3. **Combining Online and Batch Processing:** Supports a mix of real-time online processing and batch processing.
4. **Reliability and Load Balancing:** Ensures reliable message delivery and helps balance workloads.
5. **Loosely-Coupled Systems:** Ideal for systems where components operate asynchronously and independently.
6. **Inter-Enterprise Communication:** Used for secure inter-organizational communication, often in large corporations and between enterprises.
7. **Universal Applicability:** Serves as a fundamental communication paradigm in distributed systems, applicable across various scenarios.

Challenges:

- May not offer a sufficient high-level view for complex applications.
- Configuration and administration of messaging systems can be intricate.
- Less suitable for tightly-coupled systems that require real-time, synchronous communication.

---------- 0 ----------

RPC (Remote Procedure Call):

- **Definition:** RPC is a protocol for invoking procedures in remote programs as if they were local, abstracting the complexities of remote communication.
- **Procedure Invocation:** One program (client) invokes procedures in another program (server) on a remote system.

- **Use Cases:** Used in distributed computing, client-server applications, and remote service access.
- **Examples:** gRPC, Apache Thrift, Java RMI.

gRPC:

- **Definition:** gRPC is an open-source, high-performance framework developed by Google for building efficient and distributed systems. It uses the Protocol Buffers (ProtoBuf) serialization format to enable communication between applications and services across different languages and platforms. gRPC is often used for building APIs, microservices, and distributed systems.
- **How It Works:**
    1. Developers define service methods and message types using Protocol Buffers (`proto` files).
    2. gRPC generates client and server code based on the service contract.
    3. Clients and servers communicate using HTTP/2 for efficient data transfer.
    4. Language-agnostic support allows clients and servers to be written in different languages.
    5. Built-in security via TLS/SSL ensures data encryption and authentication.
- **Use Cases:** Ideal for microservices, cloud-native applications, and scenarios requiring efficient, language-agnostic communication in distributed systems.

Advantages of gRPC:

1. **Efficiency:** gRPC uses HTTP/2, which offers multiplexing, header compression, and other features, making it more efficient than its predecessor, HTTP/1.1.
2. **Language-Agnostic:** Developers can write client and server code in different programming languages, promoting interoperability in heterogeneous environments.
3. **Automatic Code Generation:** gRPC generates client and server code based on the service contract, reducing the potential for errors and saving development time.
4. **Protocol Buffers:** It uses Protocol Buffers for defining service methods and message types, which provides a concise, language-agnostic way to define data structures.
5. **Bidirectional Streaming:** gRPC supports bidirectional streaming, allowing both the client and server to send streams of messages in a single connection.

Disadvantages of gRPC:

1. **Complexity:** gRPC's Protocol Buffers and code generation might add complexity for developers who are not familiar with these technologies.
2. **Overhead:** While efficient, gRPC introduces some additional overhead compared to low-level protocols like raw TCP sockets.

3. **Limited Browser Support:** For browser-based applications, gRPC may not be the best choice, as browser support is not as widespread as for traditional REST APIs.
4. **Code Generation Dependencies:** Developers must rely on gRPC code generation tools, which might not align with some development workflows.

UDP (User Datagram Protocol):

- **Definition:** UDP is a connection-less and lightweight transport layer protocol used in computer networking. It provides a simple way to exchange data between devices on a network without establishing a dedicated connection.
- **Connectionless:** Unlike TCP, which is connection-oriented, UDP does not establish a connection before data transmission. It simply sends data packets to the destination without verifying if the recipient is available or acknowledging the receipt of data.
- **Features:**
    1. **No Connection Setup:** There is no three-way handshake as in TCP.
    2. **Minimal Overhead:** UDP has lower overhead because it lacks the reliability mechanisms of TCP, making it faster.
    3. **Unacknowledged:** UDP does not guarantee the delivery of data, and packets can be lost or arrive out of order.
    4. **Stateless:** Each UDP packet is independent of others, and the protocol does not maintain connection state information.
    5. **Used for Real-Time Applications:** UDP is often used in real-time applications like online gaming, streaming, VoIP, and DNS queries, where low latency is more critical than guaranteed delivery.
- **How It Works:**
    1. Data is encapsulated into UDP packets, which include source and destination port numbers, packet length, and a checksum for error detection.
    2. The UDP packet is sent to the recipient's IP address using IP routing.
    3. The recipient's UDP stack processes the packet based on the destination port number, delivering the data to the appropriate application.
- **Use Cases:** UDP is suitable for applications where speed and low overhead are more important than data reliability. Examples include online gaming, live video streaming, real-time communication, and IoT (Internet of Things) devices.
- **Advantages:** Low latency, simplicity, and speed, making it ideal for real-time applications.
- **Disadvantages:** Lack of reliability; data may be lost or arrive out of order without notification.

  Disadvantages of UDP:

8. **Lack of Reliability:** UDP does not guarantee that data packets will reach their destination. It does not provide acknowledgments, retransmissions, or error checking. As a result, some packets may be lost or delivered out of order.
9. **No Flow Control:** UDP does not have mechanisms for flow control, so a sender may overwhelm a receiver with data, potentially causing congestion or packet loss.
10. **No Congestion Control:** UDP does not adjust its data rate based on network congestion, which can lead to network congestion and performance issues.
11. **Ordering of Packets:** UDP does not ensure the order of packet delivery. Packets may arrive at their destination in a different order than they were sent.

In summary, UDP is a lightweight, connectionless protocol used for fast data transmission in scenarios where occasional data loss or disorder is acceptable, such as real-time applications and situations where low overhead is crucial.

→ **Handling Message Ordering in UDP:**

To handle message ordering in UDP, you can implement your own logic within the application layer. Here are some common approaches:

1. Assign sequence numbers to packets.
2. Reorder packets at the receiver based on sequence numbers.
3. Use timestamps for additional ordering information.
4. Implement buffering to store out-of-order packets temporarily.
5. Implement application-level logic for reordering.
6. Use checksums and error detection for packet validation.
7. Consider retransmission for lost packets.

These methods ensure packets are processed in the correct order at the application layer.

AMQP (Advanced Message Queuing Protocol):

- **Definition:** AMQP is an open-source application layer protocol for message-oriented middleware. It enables the efficient and reliable exchange of messages between various software components of a distributed system.

AMQP typically uses transport layer protocols like TCP as the underlying transport mechanism to establish connections and transmit messages reliably across networks. It leverages the features provided by the transport layer to ensure message integrity and delivery, but it does not define its own transport layer functionality.

 # Structure of AMQP:

1. Producers and Consumers: In AMQP, there are message producers (senders) and message consumers (receivers). Producers create and send messages, while consumers receive and process them.
2. Exchanges: An exchange is a message routing agent in AMQP. It receives messages from producers and determines how to route them to the appropriate queues based on routing rules. Exchanges use various routing algorithms to make routing decisions. The four main types of exchanges are:

- Direct Exchange: Routes messages to queues based on a matching routing key specified by the producer.
- Fanout Exchange: Routes messages to all queues bound to it. It broadcasts messages to all connected consumers.
- Topic Exchange: Routes messages to queues based on pattern matching of the routing key.
- Headers Exchange: Routes messages based on header attributes instead of routing keys.

3. Queues: Queues store messages until they are consumed by one or more consumers. Each queue can have multiple consumers, and messages are typically delivered in a round-robin fashion to consumers within the same queue.
4. Bindings: Bindings connect exchanges to queues. They specify which queues should receive messages from a particular exchange based on routing keys or patterns.
5. Message Broker: A message broker, such as RabbitMQ or Apache ActiveMQ, acts as the intermediary that manages the routing and delivery of messages between producers and consumers. It enforces AMQP's rules and provides message storage and delivery guarantees.
6. Acknowledge and Publish/Subscribe: AMQP supports various message patterns, including publish/subscribe (pub/sub) and request/response. Acknowledgments ensure that messages are delivered and processed reliably.

The working mechanism of AMQP involves the following steps:

1. The producer and consumer establish a connection with the AMQP broker.
2. Channels are created within the connection for message exchange.
3. The producer publishes a message to an exchange.
4. The exchange routes the message to the appropriate queue(s) based on routing rules.
5. The broker stores the message in the queue(s) until it is consumed.
6. The consumer subscribes to a queue and starts consuming messages.
7. After processing a message, the consumer sends an acknowledgment to the broker.
8. AMQP ensures message delivery assurance, error handling, and retry mechanisms.
9. AMQP enables interoperability between different messaging systems and languages.

AMQP provides a standardized way to exchange messages between applications, ensuring reliability and efficient communication.

 # Advantages of AMQP:

1. **Interoperability**: AMQP is an open and standardized protocol, which means that applications built with different technologies and languages can communicate seamlessly as long as they support the protocol.
2. **Reliability**: AMQP provides features like message acknowledgment, guaranteed message delivery, and persistence, making it suitable for critical and reliable messaging scenarios.
3. **Scalability**: AMQP's design allows for the horizontal scaling of message brokers and distribution of workloads across multiple nodes, enabling the handling of high message volumes.
4. **Flexibility**: It supports various messaging patterns, including publish/subscribe, point-to-point, and request/response, making it adaptable to different use cases.
5. **Message Durability**: AMQP supports the durability of messages, ensuring that they are not lost even if the message broker or consumer crashes.

Disadvantages of AMQP:
1. **Complexity**: Implementing AMQP can be more complex than using simpler messaging protocols due to its features and configurations. This complexity may increase development and maintenance efforts.
2. **Latency**: While AMQP offers reliability, features like message acknowledgment and persistence can introduce some latency, making it less suitable for ultra-low-latency applications.
3. **Resource Intensive**: Message brokers that support AMQP may consume significant system resources, which can be a concern in resource-constrained environments.
4. **Learning Curve**: Developers and administrators may need to learn the intricacies of AMQP and specific message broker configurations, which can be time-consuming.
5. **Compatibility**: Not all messaging systems or devices support AMQP, which can limit its usability in certain environments.

In summary, AMQP is a robust and widely adopted messaging protocol for building scalable and reliable distributed systems. Its flexibility and reliability make it suitable for a wide range of messaging scenarios, but it may introduce some complexity and latency compared to simpler messaging protocols. The choice of whether to use AMQP depends on the specific requirements and trade-offs of the application or system being developed.

REST (Representational State Transfer): REST is an architectural style or approach for designing networked applications. It provides a set of constraints and principles to guide the design of web services. It is not a protocol or a standard but a set of constraints and principles that guide the design of web services.

RESTful Principles:

1. Statelessness: Every request from a client contains all the information needed to process the request. The server should not store information about the client's session.
2. **Client-Server Architecture:** The client is responsible for the user interface and user experience, and the server is responsible for storing and retrieving data. They can evolve independently as long as the interface remains consistent.
3. **Cache-ability**: Responses from the server can be cached by the client. This can improve performance and reduce server load.

4. **Layered System:** Intermediary servers can be used to improve scalability by distributing the load or enforcingsecurity policies.
5. **Uniform Interface:** REST provides a uniform and consistent interface to the data, which simplifies the architecture and decouples the client from the server.
6. **Code on Demand:** Servers can extend the functionality of a client by transferring executable code (though this principle is less common).

 # Implementing REST in Java:
Java offers the JAX-RS (Java API for RESTful Web Services) specification to create REST web services. The two main implementations of JAX-RS are Jersey and RESTEasy. Here, we'll use Jersey as an example.

1. Setting Up:
Create a Maven project and add Jersey dependencies to the `pom.xml`:

2. Creating a REST Endpoint:

```
@Path("/hello")
public class HelloResource {
@GET
@Produces(MediaType.TEXT_PLAIN)
public String sayHello()
        {
        return "Hello, REST!";
        }
}` ``
```

3. Configuring the Application:
Create a class extending `javax.ws.rs.core.Application` to register our REST resources.

```
@ApplicationPath("/api")
public class RestApplication extends Application {
        // ... any additional configuration or setup
}
```

The `@ApplicationPath` annotation defines the base URI of the RESTful application.

4. Deploying:
Deploy your application to a servlet container like Tomcat. With the above configuration, you'd access the `sayHello` endpoint at:

http://localhost:8080/YourContextPath/api/hello

5. Enhancements:
You can further expand your RESTful service by using HTTP methods annotations like `@POST`, `@PUT`, `@DELETE` to handle different types of requests. You can also utilize `@PathParam`, `@QueryParam`, and other annotations to handle various input parameters.

For larger applications, you might want to look into integrating a JAX-RS implementation with Spring Boot, which provides a wide range of tools and facilities for building production-ready applications.

 # What is time in the context of Distribute System:

In the context of distributed systems, time is a critical and complex aspect that is fundamental to ensuring consistency, coordination, and reliable operation of the system. Managing time accurately and consistently across distributed components or nodes is challenging due to factors such as network latency, varying clock speeds, and potential failures within the system.

There are two primary types of time relevant to distributed systems:

1. **Physical Time**: Physical time refers to the actual time as perceived by a clock in a specific node or device. However, in a distributed system, relying solely on physical time is problematic because clocks across different machines may not be synchronized due to network latency and other factors. Clocks on different machines can drift or have varying levels of accuracy.
2. **Logical Time**: Logical time is a concept used in distributed systems to order events or actions that occur across different nodes. Logical time is typically implemented using algorithms like Lamport timestamps or Vector clocks. These algorithms allow the system to maintain a partial ordering of events based on causality, even if physical clocks are not perfectly synchronized.

    - **Lamport Timestamps**: Lamport timestamps assign a unique timestamp to each event, based on a counter that increments with each event. The timestamps are used to establish a partial order of events.
    - **Vector Clocks**: Vector clocks extend the concept of Lamport timestamps to capture causality by associating a vector of timestamps with each event. This vector reflects the knowledge of events observed by a particular node.

Synchronization of clocks in a distributed system is crucial for various purposes, such as ensuring consistent ordering of events, coordinating actions, and implementing distributed algorithms. Clock synchronization protocols like the Network Time Protocol (NTP) help in achieving a degree of synchronization among distributed system clocks by adjusting for network delays and clock drift.

In summary, managing time in a distributed system involves dealing with physical time (clocks) and logical time (ordering of events), with the aim of achieving coordination, consistency, and reliability across the distributed components despite the inherent challenges of network communication and varying clock behaviors.

 # Resolving time-related issues in distributed systems, such as clock synchronization and event ordering, can be challenging but is crucial for ensuring system reliability and consistency. Here are some common approaches to address time issues:

1. **Clock Synchronization**:

   - **Network Time Protocol (NTP):** NTP is a widely used protocol that allows machines to synchronize their clocks with a highly accurate time source, such as a time server or a GPS clock. It helps reduce clock drift and maintain time consistency across the network.
   - **Precision Time Protocol (PTP):** PTP is a more accurate protocol for clock synchronization, often used in applications that require microsecond-level precision. It is commonly employed in industrial automation and financial trading systems.

2. **Logical Clocks**:

   - **Lamport Timestamps:** Lamport timestamps provide a way to order events in a distributed system. They are based on logical time rather than physical time. Each event is assigned a Lamport timestamp, allowing nodes to establish a partial order of events based on causality.
   - **Vector Clocks:** Vector clocks extend the concept of Lamport timestamps to capture more detailed causality information. They associate a vector of timestamps with each event, reflecting the knowledge of events observed by a particular node.

3. **Event Ordering**:

- **Use Consistent Algorithms:** When designing distributed algorithms, it's essential to use consistent algorithms that consider the challenges of time synchronization. Ensure that your algorithms can work correctly even when events arrive out of order.

4. **Timeouts and Deadlines**:

- Implement timeouts and deadlines for operations. If an expected event or response does not occur within a specified time frame, take appropriate actions, such as retrying the operation or handling a timeout gracefully.

5. **Causal Ordering**:

- Use causal ordering when events in your system must respect causality. Ensure that events that are causally related are ordered accordingly, even if physical time stamps differ.

Why we use logical clocks instead of physical clocks?

Logical clocks are preferred in distributed systems because they offer flexibility, resilience to clock issues, and the ability to track event causality independently of physical clock synchronization. They provide a reliable way to order events and maintain consistency in complex distributed environments.

Distributed Algorithms

1. Time and Causality

 # Lamport’s logical clocks: Lamport's Logical Clocks Algorithm assigns timestamps to events in a distributed system. Each event is associated with an incrementing timestamp. Messages carry timestamps, and events are ordered based on these timestamps, allowing causal relationships to be determined. This helps maintain consistency and reliability in distributed systems, even when physical clocks are not synchronized.

--Working principle:

In Lamport's Logical Clocks Algorithm:

1. Each process starts with a timestamp of 0.
2. When an event happens, the process increases its timestamp by 1.
3. Messages include the sender's timestamp.
4. When a process receives a message, it updates its timestamp to be greater than or equal to the sender's timestamp.

5. Events are ordered based on their timestamps, helping track causality and ensuring consistent event order in distributed systems.

There are Four main stages:

1. Initialization: : Each process in the system initializes its logical clock (a simple integer counter) to 0.

2. Event at a Process: When a process experiences an internal event:

• It increments its logical clock by a fixed amount (usually 1).

• It assigns this new timestamp to the event.

3. Sending a Message: When a process sends a message:

• It increments its logical clock.

• It sends the message along with its current clock value.

4. Receiving a Message: When a process receives a message:

• It updates its clock to be the maximum of its own clock and the received clock from the message, then increments it.

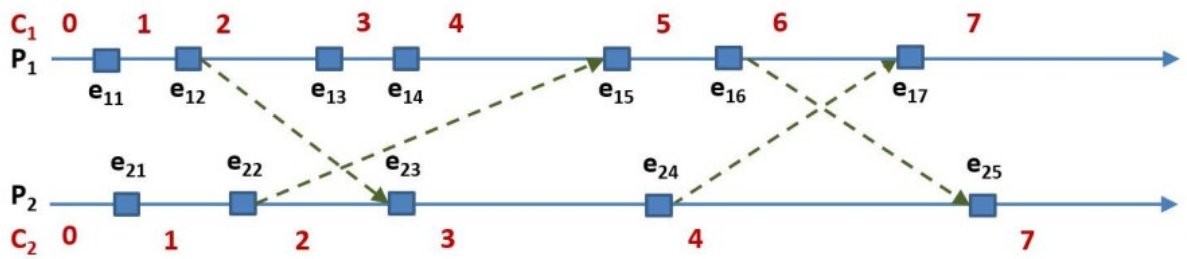
• The incremented value will be the timestamp for the receive event.

-- Key Point: The logical clocks ensure a partial ordering of events that respect the causal ordering. However, they don't ensure a total ordering of all events. That means if two events are concurrent (i.e., they don't have a causal relationship), their logical timestamps may not reflect a consistent order across all processes.

Example:

$$\triangleright\ e12 \xrightarrow{\sqsubset} e23\text{: } Max(\overbrace{2+1}^{rcv}, \overbrace{2+1}^{t_{msg}+d}) = 3$$

$$\triangleright\ e16 \xrightarrow{\sqsubset} e25\text{: } Max(\underbrace{4+1}_{rcv}, \underbrace{6+1}_{t_{msg}+d}) = 7$$

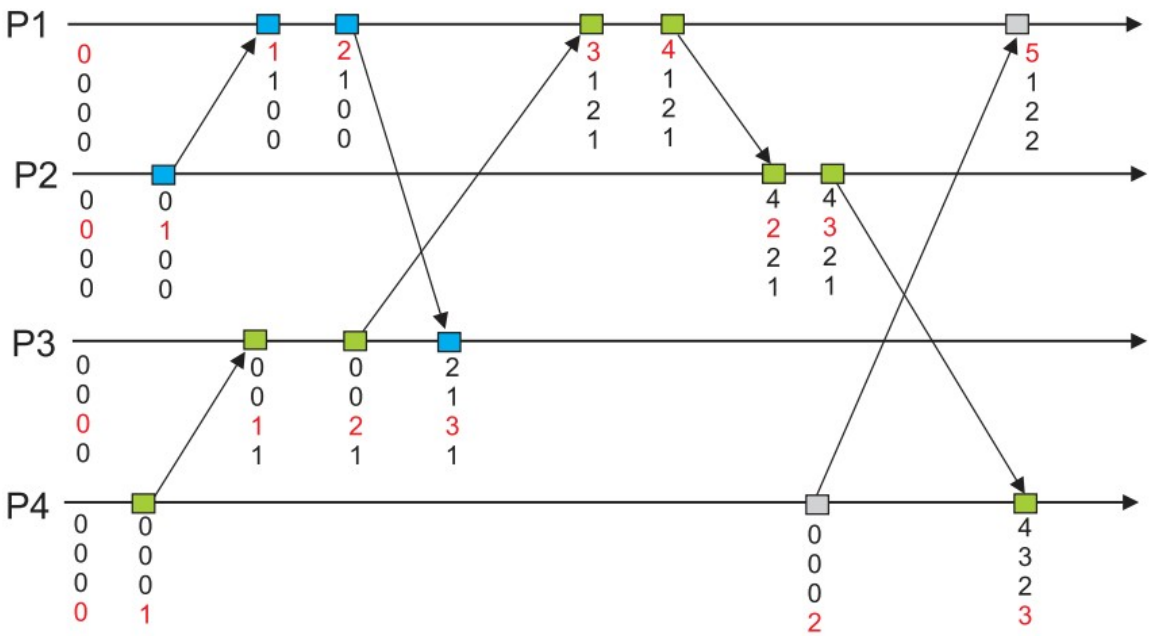


In summary, Lamport's Logical Clock offers a mechanism to order events in a distributed system based on causality. However, for a total order of all events, additional mechanisms or algorithms, such as Vector Clocks, are needed.

 # Vector Clock Algorithm: A vector clock is a data structure used for determining the partial ordering of events in distributed systems and detecting causality violations. It's essentially an array of logical clocks, one for each process in the system.

 # Working Principle:
1. Initialization: Every process in the system initializes its vector clock with all entries set to 0.
2. Event at a Process:
- The process increments its own entry in its vector clock.
3. Sending a Message:
- Before sending a message, the process increments its own entry in its vector clock.
- It then sends the message with its current vector clock.
4. Receiving a Message:
- Upon receiving a message, the process updates each entry in its vector clock to be the maximum of the value in its own vector clock and the value in the received vector clock.
- It then increments its own entry in its vector clock.

Example:



 # Comparison of Vector Clocks:
To determine the order of two events from their vector clocks:
- If all the entries in vector clock A are less than or equal to the entries in vector clock B, then A happened-before B.
- If all the entries in vector clock A are greater than or equal to the entries in vector clock B, then B happened-before A.
- If some entries in vector clock A are less than the corresponding entries in vector clock B and some entries are
greater, then A and B are concurrent events.

2. Message Ordering

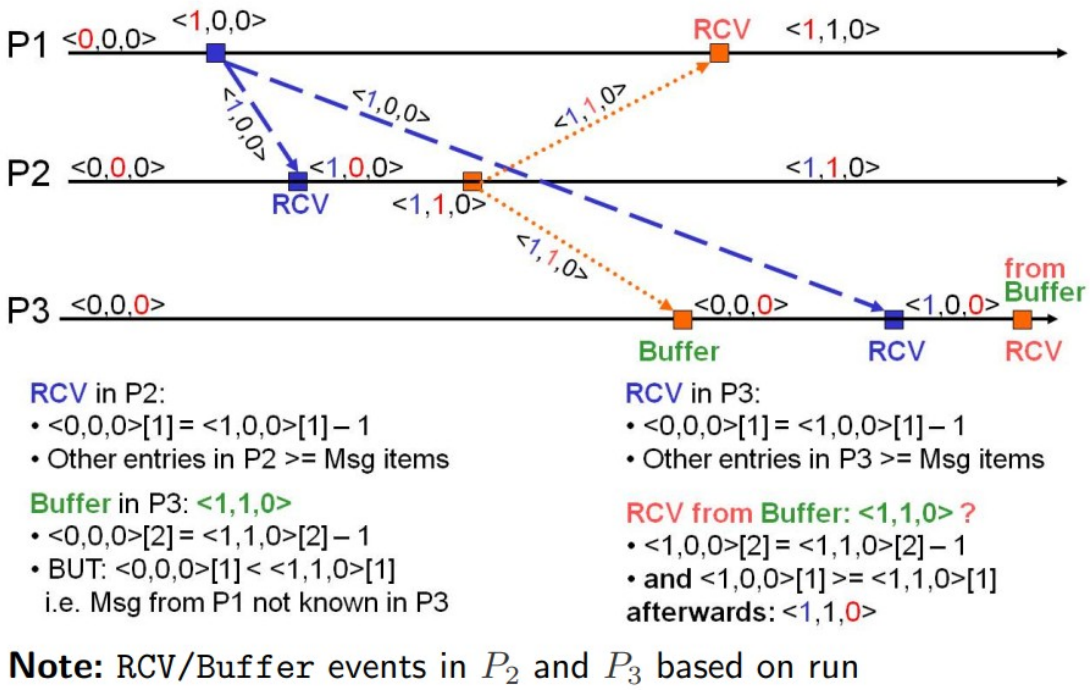
 # Birman-Schiper-Stephenson (BSS): The **Birman-Schiper-Stephenson (BSS)** protocol, also known as the Causal Broadcast algorithm, is designed to ensure that messages in a distributed system are delivered in causal order. Causal order ensures that if a message \(m1 \) causally precedes another message \(m2 \), then every process in the system that delivers both messages will deliver \(m1 \) before \(m2 \).

 # Working Principle:

1. Initialization: Each process maintains a vector clock, initialized to 0 for all entries.
2. Sending a Message:
- Before sending a message, a process increments its own entry in its vector clock.
- It sends the message along with its current vector clock.
3. Receiving a Message:
- When a process receives a message, it checks the vector timestamp of the incoming message against its own vector clock.
- The message is delivered immediately if the timestamp in the incoming message for its sender is greater than the local clock's value and the timestamps for all other processes are less than or equal to the local vector clock.
- Otherwise, the message is queued for later delivery.
- After delivering a message (either immediately or later when its delivery conditions are met), the process updates its vector clock by taking the component-wise maximum of its own vector clock and the timestamp in the delivered message, and then increments its own entry.

Example:

#



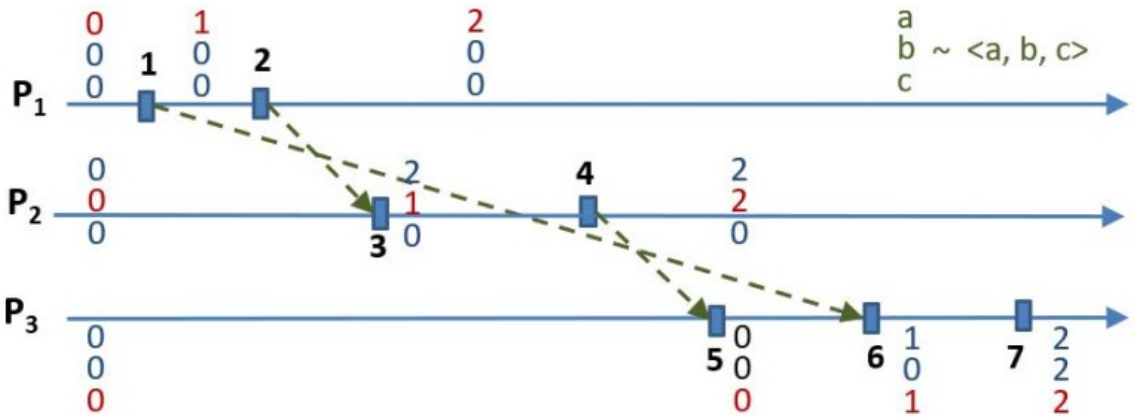
Schiper-Egli-Sandoz (SES):

The Schiper-Egli-Sandoz (SES) algorithm is an extension of the vector clocks mechanism to determine the causal order of events in distributed systems. While vector clocks can capture the causal order of events, they can be size-prohibitive in systems with a large number of processes.

The SES algorithm provides a more compact representation of the causality relation between events by using two mechanisms: direct dependency and potential dependency.

Working:
1. Initialization: Each process initializes two vector clocks: its direct clock (`D`) and its potential clock (`P`).
2. Event at a Process:
- The process increments its own entry in its direct vector clock.
3. Sending a Message:
- Before sending a message, the process increments its own entry in its direct vector clock.
- It sends the message along with its current direct and potential vector clocks.
4. Receiving a Message:
- The process updates its direct vector clock based on the received message.
- It updates its potential vector clock by merging its own potential vector clock and the received potential vector clock.
- Then, it updates its own entry in the direct vector clock

Example:



In Summary, the Schiper-Egli-Sandoz Algorithm uses vector clocks to establish an accurate event order in distributed systems by considering causality and event concurrency.

3. Distributed Mutual Exclusion

 # Suzuki-Kasami algorithm: The Suzuki-Kasami algorithm is a token-based distributed mutual exclusion algorithm. It's particularly effective in ensuring that only one process accesses a critical section in a distributed system at any given time. The core idea revolves around the use of a single token. Only the process that possesses the token can enter the critical section.

 # Working principle:

1. Initialization:
- One process starts with the token.
- Every process maintains a request queue of processes that have requested the token but haven't yet received it.
- Every process also maintains a number called its "request number", initialized to 0. It represents the number of times the process has requested access to the critical section.

2. Requesting the Critical Section:
- When a process wants to enter the critical section and doesn't have the token, it increases its request number by 1.
- It then broadcasts a request message to all other processes, containing its process ID and its updated request number.

3. Receiving a Request Message:
- Upon receiving a request message, a process updates its request queue and sends a timestamped acknowledgment back to the requester, letting the requester know it

received the request. This timestamped acknowledgment is essentially the receiver's current request number for the token.

4. Receiving the Token:
- When a process receives the token, it can enter the critical section.
- Once it exits the critical section, it checks its request queue to see which process should receive the token next. The token is then sent to the next process in the queue that has the highest request number.

 # Example:
Let's consider three processes: \(P1 \), \(P2 \), and \(P3 \). Assume \(P1 \) initially has the token.
```
P1: Token---->CS---->Release
||
Request to P2 & P3
v
P2: Request---->Receive Token---->CS---->Release
||
Request to P1 & P3
v
P3: Request---→...
```

1. \(P1 \) has the token and enters the critical section (CS).
2. While \(P1 \) is in the CS, \(P2 \) wants to enter the CS. \(P2 \) increases its request number and broadcasts a request to \(P1 \) and \(P3 \).
3. \(P1 \) and \(P3 \) send an acknowledgment to \(P2 \).
4. After \(P1 \) exits the CS, it checks its request queue and sees that \(P2 \) has the highest request number. \(P1 \) sends the token to \(P2 \).
5. \(P2 \) receives the token, enters the CS, and then later releases it. During its CS execution, \(P3 \) might also broadcast its request to enter the CS, and the algorithm proceeds similarly.

 # Lamport's algorithm is a famous approach for achieving mutual exclusion in a distributed system. It's based on logical clocks and timestamps. The algorithm ensures that only one process can enter a critical section at a time. Here's how it works:

Key Idea:

- Each process maintains a logical clock that keeps track of the order of events.
- When a process wants to enter a critical section, it sends a request to all other processes with its current timestamp.

- It waits until it has received replies from all other processes, and it can only enter the critical section when it has the highest timestamp among all processes.

Example: Let's say we have three processes: Process A, Process B, and Process C, and they want to access a shared resource (the critical section).

1. All processes start with their logical clocks at zero.
2. Process A wants to enter the critical section. It:
    - Increments its own logical clock to 1 (A's timestamp: 1).
    - Sends a request to Processes B and C with its timestamp (1).
3. Process B receives the request from A. It:
    - Increments its own logical clock to 1 (B's timestamp: 1).
    - Sends a reply to A with its own timestamp (1).
4. Process C receives the request from A. It:
    - Increments its own logical clock to 1 (C's timestamp: 1).
    - Sends a reply to A with its own timestamp (1).
5. Process A receives replies from both B and C. It compares the timestamps and finds that all timestamps are the same (1). Since A has the highest process ID (A < B < C), it can enter the critical section.
6. Process A enters the critical section and performs its task.
7. After exiting the critical section, Process A increments its logical clock to 2 (A's timestamp: 2) and sends release messages to B and C.
8. Processes B and C receive the release messages, update their timestamps to 2, and continue with their tasks.

This process ensures mutual exclusion. If another process simultaneously requests access to the critical section, the timestamps and logic will resolve who enters first.

Lamport's algorithm uses logical clocks to achieve mutual exclusion in a distributed system, ensuring that only one process accesses the critical section at a time while maintaining fairness among processes.

 # The Ricart-Agrawala algorithm is a distributed mutual exclusion algorithm that ensures that only one process accesses a critical section at a given time in a distributed system.

Working:
1. Initialization: Each process has three states:
- **RELEASED**: The process is not interested in entering the critical section.
- **WANTED**: The process wants to enter the critical section.
- **HELD**: The process is in the critical section.

2. Requesting the Critical Section:
- A process transitions from the RELEASED state to the WANTED state.
- It records its request time using its logical clock.
- It then sends a request message to all other processes and waits for their replies.

3. Deciding on Requests:
- Upon receiving a request message, a process behaves based on its state:
- If it's in the RELEASED state, it sends a reply immediately.
- If it's in the HELD state, it defers the reply.
- If it's in the WANTED state, it compares the timestamp of the incoming request with its own request timestamp. If its own timestamp is smaller (meaning its request came earlier), it defers the reply. Otherwise, it sends a reply.

4. Entering the Critical Section:
- A process enters the critical section once it has received a reply from all other processes. After this, it moves to the HELD state.
- After exiting the critical section, the process sends deferred replies to all the requests it has received. It thentransitions to the RELEASED state.

Example: Let's consider three processes: Process A, Process B, and Process C, along with a central server. They want to access a shared resource (the critical section).

1. All processes and the server start with their local clocks at zero.
2. Process A wants to enter the critical section:
    - It sends a request to the server.
    - It increments its own local clock to 1 and records this timestamp (A's timestamp: 1).
3. Process B also wants to enter the critical section:
    - It sends a request to the server.
    - It increments its local clock to 1 and records this timestamp (B's timestamp: 1).
4. Process C wants to enter the critical section as well:
    - It sends a request to the server.
    - It increments its local clock to 1 and records this timestamp (C's timestamp: 1).
5. The server receives the requests from A, B, and C. It compares the timestamps and observes that all timestamps are the same (1). However, it determines access based on the process IDs since A < B < C.
6. The server grants access to Process A to enter the critical section.
7. After exiting the critical section, Process A sends a reply to the server to indicate it has left.
8. The server now allows other processes to request access, and the process with the lowest timestamp (in this case, B) gains entry next.

This process continues, ensuring mutual exclusion. The central server manages access by comparing timestamps and giving access to processes in a fair manner.

Difference between Lamport’s and Ricart-Agrawala algorithm:

Lamport's Algorithm: It's a decentralized approach that uses logical clocks and timestamps to establish a partial order of events, allowing processes to decide when to enter a critical section. It lacks a central coordinator, which can lead to contention.

Ricart-Agrawala Algorithm: It's a centralized approach that relies on a central server to manage access to the critical section. The server compares timestamps to grant access, ensuring fairness and reducing contention.

4. Global Snapshots and Consistency

 # Global Snapshot Algorithm: A Global Snapshot algorithm is used in distributed systems to capture the state of the entire system at a particular point in time, including the states of all processes and the messages in transit between them. This algorithm is essential for debugging, monitoring, and ensuring the consistency of distributed systems.

Working Steps:

1. **Marker Introduction**: Initiate the global snapshot by introducing a marker into the system.
2. **Local State Recording**: When a process receives the marker, it records its local state (variables, data, etc.) and its empty message queue. Then, it forwards the marker to neighboring processes.
3. **Marker Collection**: The marker eventually reaches the collector process. Upon receiving the marker, the collector records its local state, which includes variables and the message queue.
4. **Global Snapshot**: The collector combines the recorded states and message queues from all processes to create a global snapshot, reflecting the entire system's state at that specific moment.

Example:

Consider a distributed system with three processes: A, B, and C. Process A initiates a global snapshot by sending a marker. The marker travels through the system, and each process records its

local state and message queue upon receiving the marker. The collector (Process C) collects the recorded states and message queues from all processes to form the global snapshot.

→ What is Marker?

In the context of the Global Snapshot algorithm, a "marker" is a special message or signal that is introduced into the distributed system to indicate the point in time at which the global snapshot is to be captured. The marker serves as a way to coordinate the processes and ensure that they record their local states and message queues consistently.

 # The Chandy-Lamport Snapshot Algorithm is used to capture a consistent global snapshot of a distributed system. It's similar to the Global Snapshot algorithm but doesn't rely on markers. Instead, it uses the "initiator" process to initiate the snapshot process. Here's how it works with an example:

Key Concept:

- An initiator process starts the snapshot and sends "snapshot" requests to its neighbors.
- Each process records its local state when it receives a snapshot request and sends markers to its neighbors.
- The process that initiates the snapshot collects the local states and markers to form a global snapshot.

Example:

Let's consider a distributed system with three processes: A, B, and C, and communication links between them. Process A will be the initiator of the snapshot:

1. **Initialization**: All processes start with their local states and message queues.
2. **Initiator's Request**: Process A, acting as the initiator, sends a "snapshot" request to its neighbors, B and C.
3. **Local State Recording**:
    - Process B, upon receiving the "snapshot" request, records its local state (e.g., variable values) and sends a marker to C.
    - Process C, upon receiving the "snapshot" request, records its local state and sends a marker to B.
4. **Marker Propagation**:
    - The markers sent by B and C propagate to their respective neighbors.
    - Process B receives C's marker and Process C receives B's marker.
5. **Local State Recording Continues**:
    - Process B, upon receiving C's marker, records its local state and sends a marker to A.

- Process C, upon receiving B's marker, records its local state and sends a marker to A.

6. **Snapshot Collection**:

- Process A collects the recorded local states and markers from B and C.
- The collected data represents the global snapshot of the distributed system at that specific point in time.

Now, Process A has captured a consistent view of the entire system's state, including the local states of B and C and the markers that traveled between processes.

The Chandy-Lamport Snapshot Algorithm is a useful tool for capturing a global snapshot without relying on markers, and it helps ensure the consistency of distributed systems for various purposes like debugging, monitoring, and maintaining data integrity.

→ **Global States**: A snapshot of the entire distributed system at a specific moment, capturing the combined states of all processes and messages in transit.

→ **Causality**: Establishing the order of events and understanding cause-and-effect relationships in a distributed system, ensuring correct event sequencing and coordination. Causality is often addressed using logical clocks and vector clocks.

5. Detecting Global System States

Termination detection is the process of determining when a distributed computation or a set of processes has completed its execution. It's essential for coordinating activities and ensuring that distributed systems do not run indefinitely. One of the popular algorithms for termination detection is the "Dijkstra-Scholten" algorithm.

Dijkstra-Scholten Algorithm:

The algorithm employs a tree structure where a node (process) becomes active due to the reception of a message. Once a process becomes active, it is considered a node in the tree with the sender as its parent. The process remains active until it has received acknowledgments for all messages it has sent.

→ Steps:

1. Activation: A process becomes active upon receiving a message from another process.
2. Message Sending:
- When an active process sends a message to another process, it increases its outstanding message count.
- The recipient process becomes a child of the sender in the logical tree.
3. Idle State:

- If a process has no further computation and has no outstanding messages (messages for which it hasn't received an acknowledgment), it sends an acknowledgment to its parent and becomes idle.

4. Propagation:

- When a process receives acknowledgments from all its children, and it has no outstanding messages, it becomes idle and sends an acknowledgment to its parent.

5. Termination:

- The algorithm terminates when the root (initiator) becomes idle.

Example:

Imagine we have 4 processes: \(P1 \), \(P2 \), \(P3 \), and \(P4 \).

1. \(P1 \) (the initiator) sends a message to \(P2 \). Now, \(P2 \) is a child of \(P1 \) in our logical tree.

2. \(P2 \) becomes active and sends messages to both \(P3 \) and \(P4 \). So, \(P3 \) and \(P4 \) are now children of \(P2 \).

3. Once \(P3 \) and \(P4 \) finish their computations and have no outstanding messages, they each send an acknowledgment to \(P2 \).

4. \(P2 \), after receiving acknowledgments from both children and having no other outstanding messages, becomes idle and sends an acknowledgment to \(P1 \).

5. \(P1 \) receives the acknowledgment from \(P2 \) and determines the system can safely terminate since no process is active and no messages are in transit.

Remember, this is a simple example. In larger systems, the tree can become more complex, but the basic principle remains the same.

 # Why is Termination always a problem in de-centralized algorithms?

Termination is a challenge in decentralized algorithms because there is no central control, processes operate independently, asynchrony complicates synchronization, nodes can join/leave dynamically, message delays vary, and achieving consensus is complex. Termination detection mechanisms are used to address these challenges.

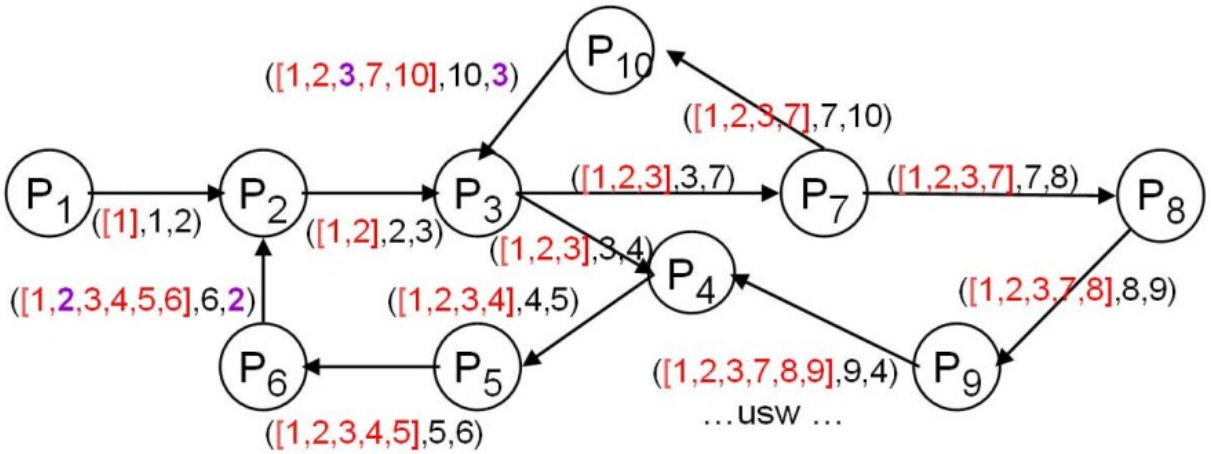
6. Distributed Deadlock Detection

Distributed deadlock occurs in distributed systems when multiple processes are waiting for resources held by each other, resulting in a cycle of dependencies and

causing all processes to be blocked. Detection and resolution mechanisms are needed to manage distributed deadlocks and allow processes to continue execution.

The Distributed Edge-Chasing Algorithm is a termination detection algorithm used in distributed systems. Specifically, it detects whether there exists any cycle in a distributed system, which might indicate a deadlock.

Basic Idea:
1. A process, suspecting a cycle, sends a "probe" message containing its own ID to one of its neighbors.
2. Each process that receives the probe checks if the ID in the probe matches its own. If it does, a cycle is detected.
3. If the ID doesn't match, the process forwards the probe to one of its neighbors.
4. If a process receives a probe that it has seen before but the ID doesn't match its own, it discards the probe. The algorithm continues until a cycle is detected or all processes have seen the probe.



Example:

Consider 5 processes, \(P1 \) through \(P5 \), connected in a topology where:
- \(P1 \) connects to \(P2 \) and \(P5 \)
- \(P2 \) connects to \(P3 \)
- \(P3 \) connects to \(P4 \)
- \(P4 \) connects back to \(P1 \)

Let's say \(P1 \) suspects a cycle:
1. \(P1 \) sends a probe with its ID to \(P2 \).
2. \(P2 \) forwards the probe to \(P3 \).
3. \(P3 \) forwards the probe to \(P4 \).
4. \(P4 \) receives the probe and forwards it to \(P1 \).
5. \(P1 \) recognizes its own ID in the probe and thus detects a cycle.

To visualize, you can draw arrows representing the path of the probe starting from \(P1 \) and ending back at \(P1 \). The formation of this loop visually represents the detection of the cycle.

 # Points to Note:
- The algorithm is simple but can be slow, especially if there's no cycle.
- The algorithm is also message-intensive; if there are many processes, it can generate a large number of probe messages.
- If the probe traverses all processes without any process identifying its own ID, then no cycle exists in the system.

6. Distributed Coordination

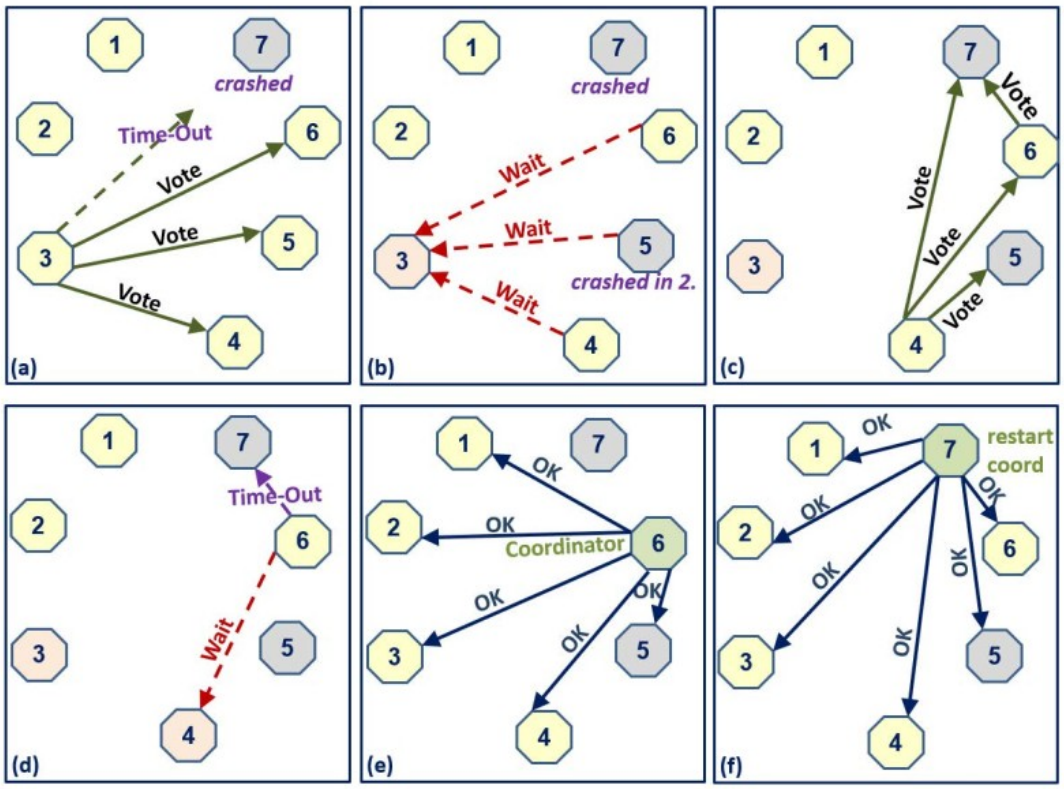
 # The Bully Algorithm is a leader election algorithm used in distributed systems to elect a coordinator or leader among a group of processes. The coordinator is typically the process with the highest priority or ID, and it takes charge of centralized tasks within the distributed system. The Bully Algorithm is designed to ensure that the most eligible process becomes the coordinator while also handling scenarios where the current coordinator fails.

→ Basic Idea:
1. If a process notices the coordinator is down, it starts an election.
2. To start an election, a process sends an "Election" message to all processes with higher IDs than itself.
3. If no one responds to the "Election" message after a certain timeout:
- The process assumes it's the coordinator and announces its leadership by sending a "Coordinator" message to all processes with lower IDs.
4. If a process with a higher ID receives the "Election" message:
- It sends a response to the initiator to stop its election process.
- The higher ID process then starts its own election process.
5. If a process that was down comes back up or a new process joins the system:
- It starts an election if it has a higher ID than the current coordinator.

→ Example:
Consider 5 processes, numbered 1 to 5. Let's assume 5 is the current coordinator.
Visualization:
```
1 -> 2 -> 3 -> 4 -> 5 (Coordinator)
```
Now, imagine process 3 detects that the coordinator (5) is down.
1. Process 3 sends "Election" messages to processes 4 and 5.
2. Process 4 responds to 3 and then sends "Election" messages to 5.

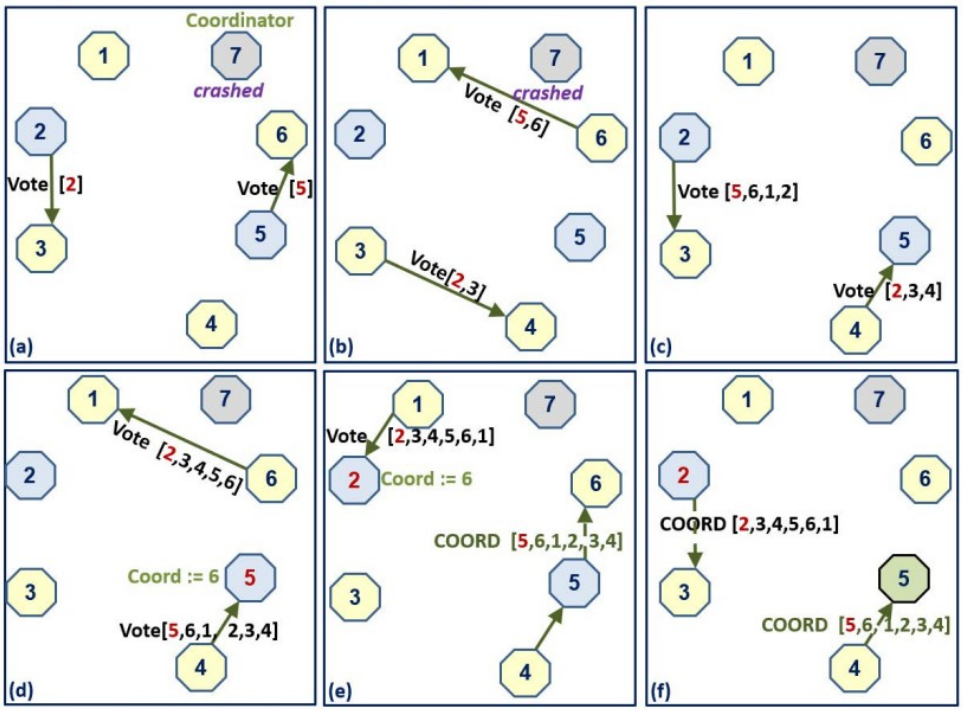
3. Process 5 does not respond since it's down.
4. Process 4, after a timeout, assumes it's the new coordinator.
5. Process 4 sends "Coordinator" messages to processes 1, 2, and 3 to announce its new role.

If process 5 comes back online:
1. It notices that the coordinator (process 4) has a lower ID.
2. Process 5 initiates an election and eventually takes back its role as the coordinator.

Visualization:```
1 -> 2 -> 3 -> 4 (Previous Coordinator) -> 5 (New Coordinator)
```

--
>




Points to Note:
- The algorithm guarantees that the process with the highest ID becomes the coordinator, provided it's alive.
- The Bully Algorithm can generate a lot of messages, especially if a lower-numbered process starts the election.
- Processes need to know about other processes in the system and their IDs.
- The algorithm is robust to failures, as any process noticing a coordinator failure can start a new election.

**# Lelann's Ring Algorithm** is a distributed algorithm used for leader election in a ring topology. The objective is to elect a leader (or coordinator) among all processes in a system. The assumption is that each process in the system knows only its immediate neighbor in a ring topology.
```

→ **Basic Idea:**
1. When a process detects the absence of a leader or wishes to initiate a leader election, it generates an "Election" message with its process ID and sends it to its neighbor.
2. Upon receiving an "Election" message, a process:
- Discards the message if it has already forwarded an "Election" message with the same or a higher ID.
- Replaces its ID in the message with its own if its ID is greater and forwards the message.
- Simply forwards the message if its ID is lesser.
3. When a process receives an "Election" message containing its own ID, it realizes it has the highest ID and announces itself as the leader by sending a "Coordinator" message around the ring.

→



Example:

Imagine we have 5 processes in a ring topology, numbered 1 to 5:

Visualization:
```
1
/ \
5 2
\ /
3
|4
```

```
Let's say process 3 wants to initiate a leader election:
1. Process 3 sends an "Election" message with ID 3 to process 2.
2. Process 2 compares its ID with the ID in the message, finds its ID (2) is lesser than 3, so it forwards the message to process 1.
3. Process 1 replaces the ID in the message with its own (since 1 > 3) and sends it to process 5.
4. Processes 5 and 4 forward the message as their IDs are less than the one in the message.
5. When the "Election" message with ID 1 comes back to process 1, it recognizes its own ID and understands that it has the highest ID. Process 1 then sends a "Coordinator" message to inform all processes that it's the leader.

→ Points to Note:
- The Lelann's Ring Algorithm can generate a lot of messages since each "Election" message has to travel around the ring.
- If two processes initiate the election roughly simultaneously, multiple election messages can circulate simultaneously.
- Like the Bully Algorithm, the process with the highest ID will always be elected leader, but the ring topology and process assumptions are different.

# Peterson's Ring Algorithm is a leader election algorithm used in distributed systems, particularly in a logical ring topology. It allows processes to elect a leader in a decentralized and coordinated manner. Unlike Lelann's Ring Algorithm, Peterson's approach is more structured and uses a phased method to reduce the number of circulating messages.

→ Basic Idea:
The algorithm works in phases, and in each phase, a process sends its ID only to its immediate neighbor, but with increasing distance in subsequent phases.
1. In the first phase, every process sends its ID to its immediate neighbor.
2. In the next phase, it sends to the neighbor 2 places away.
3. In the next, to the neighbor 4 places away, and so on, doubling the distance each time.
4. If a process receives an ID lesser than its own, it discards the message.
5. If it receives a greater ID, it stops participating in the election and forwards the higher ID.
6. When a process receives its own ID back, it knows it's the leader and broadcasts a "Coordinator" message.

### Example:

Imagine a ring of 8 processes, numbered 1 to 8:
Visualization:
```

```
1
/ \
8 2
\ /
3
/ \
7 4
\ /
5
/ \
6 6
```
Let's take process 3 as the initiator:
1. **Phase 1:** Process 3 sends its ID (3) to process 4 (1 step away). Meanwhile, every other process does the same, sending its ID to its immediate neighbor.
2. **Phase 2**: Process 3 sends its ID to process 5 (2 steps away). If process 5 has not received a higher ID by then, it'll forward the ID of 3 to process 7. Meanwhile, other processes are doing the same with a 2-step distance.
3. **Phase 3**: Process 3 sends its ID to process 7 (4 steps away). If process 7 hasn't seen a higher ID, it'll forward the ID of 3, which will eventually come back to process 3.
4. When process 3 receives its own ID, it understands it has the highest ID and announces itself as the leader by sending a "Coordinator" message.

However, if there was a process with an ID higher than 3, say process 8, by the time 8's ID reaches 3, process 3 will stop participating and only forward the ID.

→ **Points to Note:**
- Peterson's Ring Algorithm ensures that the process with the highest ID will be elected as the leader.
- Due to its phased approach, it reduces the number of circulating messages compared to Lelann's Ring Algorithm.
- The processes must know the total number of processes or have a mechanism to determine the end of phases.

 # Oral-message Algorithm: Lamport/Shostak/Pease:
Idea: An algorithm for achieving consensus in a distributed system even if some processes fail. It is designed to provide fault tolerance and ensure that processes in a distributed system can agree on a common value, even in the presence of unreliable communication.

Main Idea: The LSP Algorithm is based on the concept of "oral messages," which are messages that processes send to each other to propose values. The algorithm

proceeds in rounds, and processes take turns as the "proposer" and the "acceptor" in each round. The goal is to ensure that, by the end of the algorithm, all processes agree on a common value proposed by one of them.

Example: Think of a voting system where even if a few voting machines fail, the system can still reach a consensus on the election outcome using this algorithm.

-------- x ---------

 # Two-Phase Commit (2PC) and Three-Phase Commit (3PC) are distributed consensus algorithms used to achieve atomic commits across multiple nodes in distributed systems. They ensure that a transaction either fully completes (commits) on all nodes or fully aborts.

→ Two-Phase Commit (2PC):

Prepare Phase: The coordinator sends a "Prepare" message to all participants asking if they can commit or abort. Participants respond with a "Vote-Commit" if they can commit or a "Vote-Abort" if they cannot.

Commit Phase: If the coordinator receives a "Vote-Commit" from all participants, it sends a "Global-Commit" message to all participants. If the coordinator receives a "Vote-Abort" from any participant (or if a timeout occurs), it sends a "Global-Abort" message.

Example:
Suppose there are three banks, BankA, BankB, and BankC, that collaborate to transfer money across accounts using a distributed transaction.

BankA (Coordinator) wants to start a transaction.
BankA sends a "Prepare" message to BankB and BankC.
Both BankB and BankC reply with "Vote-Commit".
BankA then sends "Global-Commit" to both, and the transaction is committed in all three banks. If any bank had replied with "Vote-Abort", a "Global-Abort" would have been sent.

 # Three-Phase Commit (3PC): 3PC is an improvement over 2PC to handle situations where the coordinator might fail after sending the "Prepare" message. It introduces a new phase to avoid blocking.

Phases:

Can-Commit Phase:
The coordinator asks participants if they can proceed with the transaction.

Participants respond with a "Yes" if they can proceed or "No" if they cannot.

Pre-Commit Phase:
If all participants respond "Yes", the coordinator sends a "Pre-Commit" message. Participants acknowledge this by sending an "Acknowledgment" message back.

Commit Phase:
Upon receiving all acknowledgments, the coordinator sends a "Do-Commit" message. Participants then commit the transaction and acknowledge. If there's a failure or negative response at any point, the coordinator can decide to abort the transaction.

Example: Using the same banks as before:
BankA (Coordinator) wants to start a transaction.
BankA sends a "Can-Commit?" message to BankB and BankC.
Both reply with "Yes".
BankA sends "Pre-Commit" to both banks.
Both banks acknowledge.
BankA then sends "Do-Commit" to both, and the transaction is committed in all three banks.

If any bank had replied with "No", or if any failure occurred, the transaction would have been aborted

 # Differences:

Number of Phases: 2PC has two phases, while 3PC has three.
Blocking: 2PC can cause the system to block if the coordinator fails after sending "Prepare" and before receiving all votes. 3PC addresses this by introducing an additional phase.
Communication Rounds: 3PC requires more communication rounds than 2PC, which can increase latency. While 3PC mitigates some issues of 2PC, it is still not immune to all failure scenarios (like network partitions).

Real-world distributed systems often employ a variety of techniques or avoid block-oriented protocols altogether, opting for eventual consistency or other mechanisms.

 # How 3PC better than 2PC:

Three-Phase Commit (3PC) is better than Two-Phase Commit (2PC) in distributed systems because it addresses the limitations of 2PC. It offers improved handling of coordinator failures, reduces blocking scenarios, enhances robustness, provides a clearer commit decision process, and improves system availability. 3PC ensures that

transactions can proceed even if the coordinator fails, making it a more reliable choice for mission-critical systems.

 # Why do many DS use 2PC, when 3PC is better?

Two-Phase Commit (2PC) is still commonly used in distributed systems because of its simplicity and legacy compatibility, even though Three-Phase Commit (3PC) offers better reliability and reduced blocking scenarios. The choice depends on factors like system architecture, performance, and tolerance for coordination complexities.

 # The CAP theorem, also known as Brewer's theorem, is a fundamental principle in distributed computing that describes the trade-offs between three key properties of a distributed data store:

Consistency: All nodes in the distributed system see the same data at the same time. In other words, when a write operation is completed, all subsequent read operations from any node will return the most recent written value.

Availability: Every request (read or write) made to the distributed system receives a response, without guaranteeing that it contains the most up-to-date data. In this context, availability means that the system remains responsive even when some nodes or components are failing or unreachable.

Partition Tolerance: The system continues to operate, even when network partitions occur, isolating some nodes from others. Network partitions can result from network failures or delays.

The CAP theorem asserts that in a distributed system, you can achieve at most two out of the three properties simultaneously. In other words: If you prioritize Consistency and Availability, it may not be Partition Tolerant. In the event of a network partition, you might have to sacrifice either consistency or availability to keep the system running.

If you prioritize Availability and Partition Tolerance, it may not be Consistent. In this case, the system might return responses with outdated or conflicting data in situations where the network is partitioned.

If you prioritize Consistency and Partition Tolerance, you may experience Reduced Availability during network partitions because the system might refuse to respond to requests that could compromise data consistency.

Pros of CAP Theorem:

Understanding Trade-offs: The CAP theorem provides a clear framework for understanding the inherent trade-offs in distributed systems design. It helps developers and architects make informed decisions based on their specific application requirements.

Design Flexibility: By recognizing the CAP theorem, developers can design distributed systems that align with their priorities, whether that's strong consistency, high availability, or partition tolerance.

Cons of CAP Theorem:
Simplistic Model: Critics argue that the CAP theorem oversimplifies the complexity of real-world distributed systems. It doesn't account for nuances like eventual consistency or the specific implementations and optimizations used by various distributed databases.

Misleading Terminology: The terms "Consistency," "Availability," and "Partition Tolerance" can be misleading because they mean different things in the CAP context compared to how they are commonly used in database and system design.

Lack of Quantification: The CAP theorem doesn't provide a quantitative measure for balancing these trade-offs, making it challenging to determine the appropriate design decisions for a particular system.

In practice, distributed systems often aim for some degree of compromise between the CAP properties. For example, they may opt for eventual consistency (eventual convergence of data across nodes) instead of strong consistency to achieve better availability and partition tolerance. The choice depends on the specific application's requirements and the desired balance between the three properties.

 # What is the criticism being made on CAP theorem?

Critics argue that the CAP theorem oversimplifies the complexities of distributed systems, lacks practical implementation guidance, and doesn't consider nuances in emerging technologies and varying levels of consistency. Additionally, it assumes a binary view of network partitioning, which may not align with real-world network conditions. Architects should consider these criticisms when applying the CAP theorem.

Paper Analysis

1. The paper "The Byzantine Generals Problem" by Leslie Lamport, Robert Shostak, and Marshall Pease addresses the challenge of achieving consensus in distributed computing systems,

especially in the presence of faulty or malicious components. It uses a metaphor involving a group of generals who must coordinate their actions but cannot trust each other completely due to the possibility of traitors.

Key points from the paper:

- The generals must communicate via messages, and the authenticity of messages is not guaranteed.
- Consensus is achievable if and only if more than two-thirds of the generals are loyal when using only oral messages.
- Unforgeable written messages make the problem solvable regardless of the number of generals and potential traitors.

Various solutions to the Byzantine Generals Problem have been proposed over the years, including Byzantine Fault Tolerance (BFT), Practical Byzantine Fault Tolerance (PBFT), Tendermint Consensus Algorithm, Raft Consensus, and Paxos Algorithms. These solutions are crucial in ensuring the integrity and reliability of distributed systems, particularly in blockchain networks, where consensus among multiple parties is essential for system integrity.

2. The paper "Transactions in a Microservice World" by Frank Leymann discusses the impact of cloud and microservice technologies on transaction management in applications. It introduces the ACID paradigm and highlights the challenges of data replication. The paper emphasizes compensation-based transactions and introduces the concepts of SAGAs and compensation spheres as suitable approaches for managing transactions in microservice-based applications, particularly in the cloud.

- **SAGAs and Compensation Spheres**: The paper discusses the SAGAs concept, which is one of the first systematic uses of compensation pairs. In a SAGA, each step and its associated compensation function are implemented as a classical transaction. If an error occurs, the already successfully finished steps are undone by invoking their compensation functions in reverse order. The paper also introduces the concept of compensation spheres, which offer more flexibility than the strict sequence of executing steps in the SAGA model. Compensation spheres can include control flow structures like branches, loops, and parallel execution.

3. Consistency Tradeoffs in Modern Distributed Database System Design:

In his paper, Abadi delves into the latency-consistency tradeoff in distributed database systems (DDBS) and contrasts it with the CAP theorem. He discusses three scenarios for data replication, each with implications for latency and consistency. These scenarios involve sending updates to all replicas simultaneously, using a master replica for updates, and a mix of synchronous and asynchronous replication.

Abadi classifies various database systems based on the PACELC theorem, categorizing them as PA/EL (prioritizing availability during partition and latency during normal operations), PC/EC

(prioritizing consistency in both cases), PA/EC, or PC/EL. This classification includes systems like Dynamo, Cassandra, Riak, VoltDB, H-Store, Megastore, MongoDB, and PNUTS.

In summary, Abadi's paper provides a comprehensive understanding of the trade-offs in DDBS design, emphasizing that system designers must consider not only consistency and availability during network partitions but also the trade-off between latency and consistency during normal operations.

4. CAP Twelve Years Later: How the 'Rules' Have Changed:

Eric Brewer's paper revisits the CAP theorem, which deals with trade-offs in distributed systems—specifically, the balance between consistency (C), availability (A), and partition tolerance (P). Brewer suggests that, by explicitly managing partitions and employing recovery strategies, it's possible to optimize both consistency and availability while maintaining partition tolerance. A key concept is the "partition decision," involving trade-offs between latency and consistency in the face of timeouts. Strategies like Commutative Replicated Data Types (CRDTs) and compensating transactions are discussed for achieving consistency and availability during partition recovery. Real-world examples from Yahoo, banking, and design philosophies like ACID and BASE are provided. The paper highlights the flexibility of system designers in managing these trade-offs in practice.

5. Spanner, TrueTime & The CAP Theorem

Google's paper on "Spanner, TrueTime & The CAP Theorem" delves into Spanner, a distributed database, and TrueTime, a globally synchronized clock. It explores how Spanner, despite the CAP Theorem's constraints, manages to deliver strong consistency and high availability. TrueTime plays a pivotal role in achieving consistency for reads and ensuring repeatable analytics. Spanner provides external consistency, behaving as if transactions occur sequentially, even though they are distributed across servers. TrueTime allows Spanner to assign timestamps for transactions, enabling consistent reads without blocking writes. The multi-version concurrency control (MVCC) technique aids in reading without disrupting writes. Spanner offers various consistency guarantees, including external consistency, linearizability, serializability, and strong consistency. This paper showcases Spanner's ability to maintain consistency and availability on a global scale.