Distributed Computing

1 -- a) Describe a scenario where a distributed computing system could significantly improve the processing time of data compared to a single processor system.

Distributed Computing for Faster Data Processing: A Genomics Example
Imagine you are a researcher working on a project to analyze the human genome for potential genetic markers of disease. This involves processing vast amounts of data from DNA sequencing, a task that could take an enormous amount of time on a single processor system. Here is where a distributed computing system shines:

Scenario:

- **Data Distribution:** The massive genomic dataset is divided into smaller chunks and distributed across multiple computers (nodes) within the network. Each node might hold information from different individuals or different segments of the genome.
- **Parallel Processing:** Each node works on its assigned chunk of data simultaneously, performing tasks such as mapping reads to a reference genome, identifying variations, and calculating statistical correlations. This parallel processing significantly reduces the overall time required compared to a single processor handling everything sequentially.
- **Communication and Aggregation:** As each node finishes its task, it communicates its results back to a central server. The server then aggregates the results from all the nodes, creating a complete picture of the analysis.

Benefits of Distributed Computing:

- **Speed:** The parallel processing capability drastically reduces the time needed to analyze the data compared to a single processor system. This allows researchers to get results faster and make quicker progress in their research.
- **Scalability**: If the dataset size increases or the analysis becomes more complex, additional nodes can be easily added to the distributed system to handle the extra workload.
- **Cost-Effectiveness**: Utilizing a network of readily available computers can be more cost-effective than investing in a single, high-performance machine.
- **Fault Tolerance**: If one node fails, the others can continue processing, preventing a single point of failure and ensuring the project's progress.

Real-world Examples:

- **Folding at home:** This distributed computing project utilizes the idle processing power of personal computers around the world to simulate protein folding, aiding in research for diseases like Alzheimer's and cancer.
- **The Human Genome Project**: This massive international research effort relied heavily on distributed computing to process and analyze the vast amount of data involved in mapping the human genome.

In conclusion, distributed computing offers a powerful solution for scenarios involving large-scale data processing, like genomics research, by significantly reducing processing time and offering scalability, cost-effectiveness, and fault tolerance.

1 -- b) Discuss the challenges and design considerations that must be addressed when migrating from a centralized computing architecture to a distributed system architecture in a healthcare data analysis company.

Moving from a centralized to a distributed architecture in a healthcare data analysis company offers significant advantages, but also presents unique challenges. Here are some key points to consider:

Challenges:

- **Data Security and Privacy**: Healthcare data is highly sensitive and subject to strict regulations like HIPAA. Ensuring data security and patient privacy becomes more complex in a distributed environment.
- **Data Consistency and Integrity**: Maintaining data consistency across multiple nodes is crucial to avoid errors and ensure reliable analysis.
- **Network Infrastructure and Performance**: Distributed systems depend on a reliable and high-performance network infrastructure to handle communication and data transfer between nodes.
- **Complexity of System Management**: Managing and monitoring a distributed system is inherently more complex than a centralized system.
- **Fault Tolerance and Redundancy**: Failure of individual nodes can disrupt the system.

Design Considerations:

- **Choosing the Right Distributed Architecture:** Different architectures (e.g., client-server, peer-to-peer) offer different trade-offs in terms of complexity, scalability, and fault tolerance. The choice depends on the specific needs of the healthcare data analysis company.
- **Data Partitioning Strategy**: Determine how to divide the data across different nodes. Options include horizontal partitioning (by patient, data type) or vertical partitioning (by data features).
- **Data Storage and Management**: Choose appropriate distributed database technologies or data lakes that can efficiently handle large-scale healthcare data while ensuring security and consistency.
- **Processing Framework**: Select a suitable framework (e.g., Apache Spark, Hadoop) for distributed data processing that aligns with the company's analytical needs and programming expertise.
- **Scalability**: Design the system with future growth in mind, ensuring it can easily accommodate increasing data volumes and user demands.

By carefully addressing these challenges and design considerations, healthcare data analysis companies can successfully migrate to a distributed system architecture, unlocking its benefits of scalability, performance, and fault tolerance while maintaining data security and integrity.

2 -- a) Explain how logical clocks could be implemented in a distributed system used for managing a multi-city bus reservation system.

In a multicity bus reservation system, ensuring the correct order of events (e.g., booking a se at) is crucial. However, achieving perfect clock synchronization across a distributed system is impractical. Logical clocks offer a solution by providing a way to track the relative ordering of events without relying on physical time. Here is how they could be implemented:

Lamport Timestamps:

1. **Each node (city server) maintains its own logical clock:** This clock is simply a counter tha t gets incremented with each event that occurs locally on that node.
2. **Attaching Timestamps to Events:** Each time an event happens (e.g., a user requests a sea t reservation), the node includes its current logical clock value as a timestamp with the e vent message.
3. **Updating Clocks on Message Exchange:** When a node receives a message, it compares t he timestamp of the message with its own logical clock.
   - If the received timestamp is greater, the node updates its own clock to be one greate r than the received timestamp.
   - If the received timestamp is lower or equal, the node simply increments its own clock by one.
4. **Total Ordering of Events:** By following these rules, Lamport timestamps establish a "happens-before" relationship between events. If event A has a lower timestamp than event B, we can infer that A happened before B, or at least they are concurrent (happened at the same time).

Implementation Example:

- **Scenario:** User A in City 1 and User B in City 2 both attempts to book the last seat on a bus traveling between the two cities.
- **Process:**
  - City 1 server receives User A's request and assigns it timestamp 1.
  - City 2 server receives User B's request slightly later and assigns it timestamp 1
  - City 1 server sends the reservation message to City 2 with timestamp 1.
  - City 2 server receives the message, updates its clock to 2 (1 + 1), and then processes User B's request, assigning it timestamp 3 (2 + 1).

Result:

Even though both requests initially had the same timestamp (1), the Lamport timestamp rul es ensure that User A's reservation (timestamp 1) is considered to have happened before Us er B's reservation (timestamp 3). Thus, User A gets the last seat and User B receives a notific ation that the bus is full.

Benefits:

- **No need for perfect clock synchronization:** Simplifies system design and reduces reli ance on external time sources.
- **Ensures causal ordering:** Events that have a causal relationship (one influences the ot her) are guaranteed to be ordered correctly.
- **Helps in conflict resolution:** Allows identification of conflicting events and implement ation of appropriate resolution strategies.

Limitations:

- **Does not capture true concurrency:** Events with the same timestamp may have happ ened concurrently, but Lamport timestamps cannot distinguish this.
- **Additional mechanisms may be needed:** For scenarios requiring total ordering of all events, additional mechanisms like a central coordinator or vector clocks might be ne cessary.

Logical clocks provide a valuable tool for maintaining order in distributed systems like a multi -city bus reservation system, ensuring data consistency and reliable operation.

2 -- b) Propose an efficient implementation of vector clocks in a distributed social media platform to ensure that posts and comments are synchronized in the correct causal order.

Vector clocks offer a more sophisticated approach than Lamport timestamps to track the cau sal relationships and ordering of events in a distributed social media platform. Here's an effic ient implementation to ensure posts and comments are synchronized correctly:

Data Structures:

- **Each node (user's device or server) maintains a vector clock:** The vector clock has an ent ry for each node in the system, initialized to zero.
- **Each post and comment have its own vector clock:** This clock captures the state of the n ode's vector clock at the time the post/comment was created.

Event Handling:

1. **Local Event:** When a user creates a post or comment on their device:
   - Increment the local node's entry in the vector clock by 1.
   - Attach the current vector clock to the post/comment as metadata.
2. **Message Broadcast:** The device broadcasts the new post/comment with its attached vec tor clock to other nodes (e.g., followers or friends) in the network.
3. **Receiving a Message:** When a node receives a post/comment:

- **Compare Vector Clocks:**
  - For each entry in both vector clocks (received and local), update the local entry with the maximum value between the two.
  - This ensures that the local node's clock reflects the knowledge of all events that hav e happened across the system.
- **Causal Ordering:**
  - Compare the updated local vector clock with the received vector clock.
  - If the local clock is greater than or equal to the received clock in all entries, the recei ved event can be displayed immediately as it is causally after or concurrent with all k nown events.
  - If the received clock is greater in any entry, the event is causally "in the future" and i s buffered until the local clock catches up (i.e., receives necessary updates from othe r nodes).

Example:

- **User A (clock [1,0,0]) makes a post.**
- **User B (clock [0,1,0]) sees the post and comments on it, resulting in a clock of [1,2,0].**
- **User C (clock [0,0,1]) is a follower of both A and B. When they receive both updates:**
  - User A's post updates C's clock to [1,0,1] (max of [1,0,0] and [0,0,1]).
  - User B's comment updates C's clock to [1,2,1] (max of [1,0,1] and [1,2,0]).
  - User C can see both the post and comment immediately as their clock reflects knowl edge of both events.

Benefits:

- **Accurate Causal Ordering:** Ensures posts and comments are displayed in the correct ord er based on their causal relationships, even with concurrent events.
- **Efficient Synchronization:** Nodes only need to exchange vector clocks, which are small da ta structures, making synchronization efficient.
- **Offline Tolerance:** Users can create posts and comments even when offline, and the syst em will synchronize them correctly when they come back online.

Challenges:

- **Vector Clock Size:** The size of the vector clock grows with the number of nodes in the sys tem. Techniques like pruning or using hierarchical structures can mitigate this issue.
- **Garbage Collection:** Mechanisms are needed to remove outdated information from vect or clocks to avoid unbounded growth.

3 -- a) Describe how snapshot recording algorithms for FIFO channels can be utilized in a distributed banking transaction system.

Snapshot recording in FIFO (First In, First Out) channel-based distributed systems is a method to capture a consistent global state of the system. It's crucial for analysing, debugging, and maintaining high availability in distributed systems.

Technical Process

1. **Initiation:**
   - Triggered by an event or condition, a designated process (initiator) starts the snapshot.
   - The initiator records its current state (data, variables, counters) and sends a "marker" message to other processes.
2. **Algorithm Execution:**
   - Upon receiving the marker, each process records its state and forwards the marker along its outgoing channels.
   - Regular operations continue during this process, leveraging the FIFO property to **maintain consistency.**
3. **State Collection:**
   - Recorded states from all processes are aggregated to form a complete snapshot.
   - This can be centralized or decentralized, depending on the system's architecture.
4. **Snapshot Recording Program:**
   - Can be an integrated part of the application or an external module.
   - Responsible for initiating, coordinating, and collecting snapshot data.

Why FIFO Matters

- **Guaranteed Message Ordering:** The FIFO property ensures that messages sent over a channel are received in the same order they were sent. This is essential for building a consistent snapshot because it prevents messages sent before the marker from being mistaken as happening after the snapshot was initiated.
- **Causal Relationships:** In a distributed system, events across different processes can influence each other (e.g., process A sending a message to process B, which triggers another event). FIFO channels help preserve these causal relationships. When building the snapshot, if a message 'm' is in the channel history, it guarantees that the event that caused 'm' to be sent is already included in the snapshot of the sender process.
- **Consistency**: Without FIFO, a process might receive a marker, take its local snapshot, and then receive a message that was sent earlier but got delayed in the network. This would create an inconsistent snapshot where the effects of that delayed message are missing. FIFO channels ensure that no "out-of-order" messages corrupt the snapshot.

Let's illustrate this with an example:

Suppose you have a distributed banking system. Here's how FIFO ensures a correct snapshot:

1. **Transaction in Progress:** Customer A transfers $100 from Account 1 to Account 2. Messages are sent between the processes managing these accounts.
2. **Snapshot Initiated:** A snapshot is triggered to capture the system's state for auditing.
3. **FIFO in Action:** Due to FIFO, if the message about the $100 transfer is in the channel history when the receiver process takes its snapshot, it's *guaranteed* that the sender process has already accounted for the transfer in its own snapshot. This ensures consistency: either the $100 has been deducted and is "in transit", or it hasn't left Account 1 yet.

Without FIFO: If the message about the transfer could be received *after* the snapshot, you could have a situation where the $100 seems to have vanished from the system, violating the integrity of the recorded state.

3 -- b) Analyze the necessary and sufficient conditions for consistent global snapshots in a distributed stock trading system, and how these conditions affect system performance during high volatility.

In a distributed stock trading system, achieving a consistent global snapshot is crucial for maintaining data integrity and fairness. A consistent snapshot represents a state of the system where all stock prices and account balances are accurate and reflect a single point in time, even with concurrent trades and updates happening across different nodes.

Necessary Conditions:

- **Atomicity of Transactions:** Each trade or update must be treated as an atomic unit, meaning it either completes fully or fails entirely, leaving the system state unchanged. This prevents partial updates that could lead to inconsistencies.
- **Causal Ordering of Messages:** Messages related to trades or updates must be delivered in the order they were sent. This ensures that the effects of one action are visible before subsequent actions are processed.
- **No Global Clock:** Due to the distributed nature of the system, relying on a single global clock is impractical. Instead, mechanisms like timestamps or logical clocks are used to establish a partial order of events.

Sufficient Conditions:

- **Two-phase Commit Protocol:** This protocol ensures atomicity of transactions across multiple nodes. It involves a prepare phase where all nodes agree to commit, followed by a commit phase where the changes are made permanent.
- **Lamport Timestamps or Vector Clocks:** These mechanisms help establish a partial ordering of events, ensuring causal consistency. They assign unique identifiers to events based on the sending node and the local time or a vector of logical clocks.
- **Global Snapshot Algorithms:** Specific algorithms like Chandy-Lamport or Lai-Yang can be employed to capture a consistent global snapshot of the system state. These algorithms rely on the exchange of markers and local state information to ensure a consistent view.

Impact of High Volatility on Performance:

During periods of high volatility, the frequency of trades and updates increases significantly. This puts additional strain on the system, potentially impacting performance in the following ways:

- **Increased Communication Overhead:** Frequent updates lead to more messages being exchanged between nodes, increasing network traffic and processing overhead.
- **Contention and Locking:** As more transactions compete for resources, contention and locking can occur, leading to delays and performance bottlenecks.
- **Rollback and Recovery:** In case of failures or inconsistencies, rollback and recovery mechanisms need to be activated, which can be time-consuming and further impact performance.

Mitigation Strategies:

- **Scalability:** Employing a scalable architecture with distributed processing and data storage can help distribute the load and improve performance.
- **Caching and Replication:** Caching frequently accessed data and replicating critical components can reduce latency and improve availability.
- **Optimizations:** Optimizing communication protocols, reducing message sizes, and employing efficient locking mechanisms can help minimize overhead.
- **Fault Tolerance:** Implementing robust fault tolerance mechanisms like redundancy and failover ensures system availability during failures.

By carefully considering these conditions and their implications, it is possible to design a distributed stock trading system that maintains consistent global snapshots and performs well even under high volatility.

4 -- a) Illustrate the use of group communication in a distributed gaming environment where players interact in real-time.

Group communication plays a vital role in creating seamless and engaging real-time interactions within distributed gaming environments. Here's how it works:

Scenarios and Communication Patterns:

- **Multiplayer Lobbies and Matchmaking:**
  - **Publish-Subscribe:** Servers publish available game sessions or lobbies, and players sub scribe to be notified of matching options based on chosen criteria. This facilit ates efficient matchmaking without requiring players to continuously poll the server.
- **In-Game Actions and State Synchronization:**
  - **Multicast:** When a player performs an action (e.g., movement, attack), the information is multicast to all other players in the game session. This ensures everyone has a consistent view of the game state, enabling real-time interaction and responsiveness.
- **Team Coordination and Strategy:**
  - **Group Messaging:** Team members can communicate through dedicated chat channels, sharing strategies, coordinating actions, and fostering collaboration.
  - **Voice Chat:** Real-time voice communication adds another layer of interaction, allowing for quick reactions, callouts, and a more immersive experience.

Protocols and Technologies:

- **User Datagram Protocol (UDP):** UDP is often preferred for real-time gaming due to its low latency and efficiency. It prioritizes speed over reliability, which is acceptable for most in-game actions as minor losses can be interpolated or corrected.
- **Remote Procedure Calls (RPCs):** RPCs allow for direct communication between client s and the server, facilitating actions like joining/leaving games, updating player stats, and handling game logic.
- **IP Multicast:** Efficiently distributes information to multiple clients simultaneously, red ucing network traffic and server load.
- **Middleware Solutions:** Frameworks like Photon or RakNet provide abstractions and functionalities for group management, message routing, and reliable delivery, simplifying development.

Benefits of Group Communication:

- **Real-time Interaction:** Enables players to react to each other's actions instantly, creating a dynamic and engaging experience.
- **Scalability:** Efficiently handles many concurrent players without compromising performa nce.
- **Flexibility:** Supports various interaction models, from large-scale battles to small team-based competitions.
- **Reduced Server Load:** Multicast and efficient protocols minimize server processing and b andwidth requirements.

Challenges and Considerations:

- **Latency and Jitter:** Network delays and fluctuations can impact responsiveness and lead to inconsistencies in the game state. Techniques like lag compensation and prediction hel p mitigate these issues.
- **Reliability and Ordering:** While UDP provides speed, ensuring reliable delivery and mess age ordering is crucial for critical game events. Techniques like sequence numbers and ac knowledgments address these concerns.

- **Security:** Protecting against cheating, hacking, and malicious behaviour is vital. Encryption, authentication, and server-side validation mechanisms are essential.

Examples:

- **Massively Multiplayer Online Role-Playing Games (MMORPGs):** Players interact in a persistent world, forming groups, battling monsters, and engaging in social activities.
- **First-Person Shooter (FPS) Games:** Fast-paced action requires constant communication and coordination between teammates.
- **Real-time Strategy (RTS) Games:** Players issue commands to units, manage resources, and engage in strategic manoeuvres, relying on efficient group communication.

By effectively utilizing group communication, distributed gaming environments facilitate real-time interaction, fostering collaboration, competition, and a sense of community among players.

****

4 -- b) Design a termination detection protocol using distributed snapshots for a complex multi-agent simulation system where agents dynamically enter and exit the simulation.

Termination Detection Protocol with Dynamic Agents using Distributed Snapshots
This protocol uses the Chandy-
Lamport algorithm as a foundation, adapting it to handle dynamic agent entry and exit withi
n a multi-agent simulation.

Assumptions:

- Agents communicate asynchronously through message passing.
- Each agent maintains a local state and a list of its outgoing channels (connections to othe
r agents).
- A unique identifier is assigned to each agent upon entry.

Protocol:

1. **Snapshot Initiation:**
    - Any agent suspecting termination (e.g., completing its tasks or reaching a specific state) initiates a snapshot by recording its local state and sending "marker" messages along all its outgoing channels.
2. **Marker Propagation:**
    - Upon receiving a marker for the first time on a channel, an agent:
        - Records its local state.
        - Sends marker messages along all its outgoing channels except the one where it re
ceived the marker.
    - If an agent receives a marker on a channel where it has already recorded its state, it i
gnores the marker.
3. **Agent Entry:**
    - When a new agent enters the simulation, it establishes connections with other agent
s as required.
    - Upon receiving its first message (not a marker), the new agent records its initial state and starts participating in the protocol as usual.
4. **Agent Exit:**
    - When an agent exits, it sends a special "exit" message to all its connected neighbors.
    - Upon receiving an "exit" message, the neighbor removes the corresponding channel from its list and notes the exit in its local state.

5. **Termination Detection:**

- An agent concludes termination when:
    - It has recorded its state.
    - It has received markers on all its incoming channels.
    - It has received "exit" messages from all agents it previously sent a marker to (ensuring no in-transit messages).
- The first agent to detect termination initiates a "termination" message that propagates through the system like a marker, informing all agents to stop.

Data Structures:

- **Local State:** Agent-specific data relevant to the simulation.
- **Channel List:** Tracks outgoing connections to other agents.
- **Recorded State:** A snapshot of the local state at the time of receiving the first marker.
- **Received Markers:** A set to track channels on which markers have been received.
- **Exit List:** Keeps track of agents that have exited.

****

5 -- a) Discuss how Lamport's algorithm can be applied to manage access to a distributed database system among multiple users.

Lamport's logical clock algorithm, with its ability to establish a partial ordering of events, can be applied to manage concurrent access to a distributed database system and ensure data consistency. Here's how it works:

Implementation:

1. **Timestamping Transactions:**

- Each transaction is assigned a timestamp using the local logical clock of the node where it originates. This timestamp reflects the "happened-before" relationship between transactions.

2. **Clock Synchronization:**

- Nodes exchange their clock values along with transactions.
- Upon receiving a message, a node updates its local clock to be the maximum of its current clock and the received clock value + 1. This ensures that the causal order of events is preserved.

3. **Concurrency Control:**

- A concurrency control mechanism, like locking or optimistic concurrency control, is used to prevent conflicts between concurrent transactions.
- Lamport timestamps are used to determine the order in which conflicting transactions should be executed. For instance, the transaction with the lower timestamp is processed first, ensuring consistency.

4. **Distributed Commit Protocol:**

- A distributed commit protocol, such as two-phase commit, is used to ensure that transactions are either committed on all nodes or aborted on all nodes. This prevents inconsistencies and data loss.

Example:

1. Node A initiate's transaction T1 with timestamp 10.
2. Node B initiates transaction T2 with timestamp 12.
3. T1 is sent to Node B. B updates its clock to 13 (max (12, 10+1)).
4. T2 is sent to Node A. An update its clock to 14 (max (13, 12+1)).
5. Both transactions attempt to access the same data item.
6. Concurrency control mechanism determines that T1 has the lower timestamp and should be executed first.
7. T1 commits successfully.
8. T2 is processed, but it may need to be aborted or rolled back if it conflicts with the changes made by T1.

5 -- b) Compare and contrast the efficiency of Raymond’s tree-based algorithm versus Suzuki-Kasami’s broadcast-based algorithm in a large-scale distributed file system.

Raymond's tree-based algorithm and Suzuki-Kasami's broadcast-based algorithm are both designed to achieve total ordering of messages in distributed systems, but they employ different approaches to achieve this goal. Let's compare and contrast their efficiency in the context of a large-scale distributed file system:

1. **Raymond's Tree-Based Algorithm**:

- **Approach**: Raymond's algorithm constructs a logical tree structure among processes, where each process has a parent and possibly multiple children. Messages are forwarded up and down the tree until they reach all processes, ensuring total ordering.
- **Efficiency**:
  - **Scalability**: Raymond's algorithm may suffer from scalability issues in large-scale distributed systems. As the number of processes increases, the depth of the logical tree may also increase, leading to longer message propagation delays.
  - **Overhead**: The maintenance of the logical tree structure incurs overhead in terms of message routing and tree management. This overhead may become significant in large-scale systems with a high degree of dynamism or churn.
  - **Reliability**: Raymond's algorithm provides reliable total ordering of messages, ensuring that all processes receive messages in the same order. However, achieving this reliability may come at the cost of increased latency and resource consumption.

2. **Suzuki-Kasami's Broadcast-Based Algorithm**:

- **Approach**: Suzuki-Kasami's algorithm relies on a token-based approach, where a token is passed among processes to control message broadcasting. Only the process holding the token is allowed to broadcast messages, ensuring that messages are delivered in a total order.
- **Efficiency**:
  - **Scalability**: Suzuki-Kasami's algorithm exhibits better scalability compared to Raymond's algorithm, particularly in large-scale distributed systems. The token-based approach allows for efficient message delivery without the need for maintaining complex tree structures.
  - **Overhead**: The overhead of Suzuki-Kasami's algorithm is relatively low compared to Raymond's algorithm. Since the token is passed sequentially among processes, there is minimal overhead associated with message routing or tree maintenance.
  - **Reliability**: Suzuki-Kasami's algorithm provides reliable total ordering of messages like Raymond's algorithm. However, it may achieve this with lower latency and resource consumption, particularly in large-scale systems.

Conclusion:

For large-scale distributed file systems with potentially high contention, **Suzuki-Kasami's broadcast-based algorithm offers superior performance, scalability, and fault tolerance compared to Raymond's tree-based approach**. However, in smaller networks or low-contention environments, Raymond's algorithm might be a simpler and more efficient choice. The optimal choice depends on the specific characteristics and requirements of the distributed file system.

6 -- a) Explain how the Chandy-Misra-Haas deadlock detection algorithm for the AND model can be applied in a distributed manufacturing control system.

Imagine a distributed manufacturing system where multiple controllers manage different parts of the production process. Each controller acts as a process in the AND model, holding resources (e.g., machine access, raw materials) and requesting resources from other controllers to complete tasks. Deadlocks can arise when controllers are stuck waiting for resources held by others, leading to production stalls.

CMH Algorithm Overview:

- **Resource Model:** Assumes the AND model, where a process needs to acquire all request ed resources simultaneously to proceed.
- **Detection Mechanism:**
  - **Probes:** Controllers periodically initiate probes containing their current resource hold ings and requests.
  - **Probe Propagation:** Probes are forwarded through the system, following the "waits-for" graph (i.e., process A waits for a resource held by process B).
  - **Deadlock Detection:** A controller detects a deadlock if it receives a probe that
    - Originated from itself.
    - Has visited all processes involved in the cycle of waiting resources.

Application in Manufacturing Control:

1. **Resource Representation:** Resources in the manufacturing system (machines, materials) are modeled as resources in the AND model. Each controller holds a set of resources and can request resources from other controllers.
2. **Probe Initiation:** Controllers periodically initiate probes containing:
   - **Held resources:** List of resources currently held by the controller.
   - **Requested resources:** List of resources the controller is waiting to acquire.
   - **Probe initiator:** Identifier of the controller that started the probe.
   - **Visited controllers:** List of controllers the probe has already visited.
3. **Probe Propagation:**
   - When a controller receives a probe, it adds its identifier to the "visited controllers" list.
   - If the controller is not waiting for any resources or is waiting for resources from controllers already visited by the probe, it discards the probe.
   - Otherwise, it forwards the probe to the controllers it is waiting for resources from.
4. **Deadlock Detection:**
   - If a controller receives a probe that originated from itself and has visited all controllers involved in the resource wait cycle, it detects a deadlock.
5. **Deadlock Resolution:**
   - Once a deadlock is detected, the system can implement various resolution strategies, such as:
   - **Resource Preemption:** Forcibly taking resources from one or more controllers to break the cycle.
   - **Rollback:** Reverting one or more controllers to a previous state to break the cycle.
   - **Process Termination:** Terminating one or more controllers to release resources.

Overall, the CMH algorithm offers a valuable tool for maintaining efficiency and preventing costly downtime in distributed manufacturing control systems by enabling early detection and resolution of deadlocks.

6 -- b) Propose a strategy for resolving deadlocks in a distributed system that controls access to multiple satellite communication channels.

A distributed system manages access to multiple satellite communication channels fo r various ground stations. Deadlocks can occur when multiple stations request and hold chan nels in a way that creates a circular dependency, preventing progress.

Proposed Strategy:

This strategy combines elements of resource preemption and rollback with prioritizat ion to effectively resolve deadlocks while minimizing disruption.

1. Deadlock Detection:

- Implement a deadlock detection algorithm such as Chandy-Misra-Haas (CMH) or edge chasing. These algorithms identify cycles in the resource depend ency graph, indicating a deadlock situation.

2. Deadlock Resolution:

- **Prioritization:** Assign priorities to ground stations based on their mission criticality, data urgency, or other relevant factors.
- **Victim Selection:** When a deadlock is detected, identify the "victim" station, which will be pre-empted or rolled back to break the cycle. Selection criteria could include:
  - **Lowest Priority:** Choose the station with the lowest priority to minimize disruption to critical operations.
  - **Least Progress:** Select the station that has made the least progress towards its goal, minimizing wasted effort.
  - **Rollback Cost:** Consider the cost of rollback for each station, aiming to minimize data loss or retransmission overhead.
- **Resource Pre-emption:**
  - If the victim has acquired channels that are essential for other stations involved in the deadlock, pre-empt those channels and assign them to the waiting stations. This allows blocked stations to proceed.
  - The victim station will need to re-request the pre-empted channels later.
- **Rollback and Retry:**
  - If pre-emption is not feasible or desirable, roll back the victim station's state to a point before it acquired the conflicting channels. This may involve discarding partially transmitted data or resetting the communication session.
  - The victim station can then retry its request for channels, hopefully avoiding the deadlock condition due to the changed state of the system.

3. Communication and Coordination:

- Establish a centralized coordinator node or a distributed consensus mechanism to ma nage deadlock resolution. This ensures consistent decision-making and avoids conflicts during resource preemption or rollback.
- Implement clear communication protocols for notifying stations about preemption or rollback decisions, allowing them to adjust their operations accordingly.

Additional Considerations:

- **Deadlock Prevention:** Explore techniques like resource ordering or timeouts to minimize the occurrence of deadlocks.
- **Fairness:** Implement mechanisms to prevent starvation of low-priority stations. This could involve periodically increasing their priority or using a round-robin approach during victim selection.
- **Optimization:** Continuously evaluate and refine the priority system and victim selection criteria based on historical data and system performance.

Overall, this strategy provides a balanced approach to deadlock resolution in distributed satellite communication systems, ensuring efficient resource utilization while maintaining fairness and minimizing disruptions to critical operations.

7 -- a) Describe a scenario in which Byzantine agreement is necessary in a distributed system used for inter-bank transactions.

Imagine a distributed system handling inter-bank transactions, where multiple banks act as nodes in the network. These nodes need to reach consensus on the validity and order of transactions despite potential failures or malicious actors within the system. This is where Byzantine agreement becomes crucial.

Scenario:

1. **Transaction Initiation:** Bank A initiates a transfer of $1 million to Bank B for a client. This transaction is broadcasted to the network of participating banks (nodes).
2. **Node Failure:** One of the nodes, Bank C, experiences a system failure. It may become unresponsive or start sending conflicting information about the transaction to different nodes.
3. **Byzantine Behavior:** Another node, Bank D, might be compromised or act maliciously. It could attempt to double-spend the funds by sending a conflicting transaction claiming the same $1 million is being transferred to Bank E.
4. **Reaching Consensus:** Despite the failure of Bank C and the malicious behaviour of Bank D, the remaining honest nodes need to reach an agreement. They need to determine if the transaction from Bank A to Bank B is valid and should be processed or if it should be rejected due to the conflicting information.

Byzantine Agreement in Action:

This is where a Byzantine Fault Tolerant (BFT) algorithm, like Practical Byzantine Fault Tolerance (PBFT), comes into play. The BFT algorithm ensures that:

- **Agreement:** All honest nodes reach the same conclusion about the validity and order of the transaction.
- **Validity:** If the initiating node (Bank A) is honest, then all honest nodes agree on the transaction's validity.
- **Termination:** The consensus process eventually finishes, and a decision is reached within a bounded time.

Benefits of Byzantine Agreement:

- **Resilience:** The system can withstand failures and malicious attacks from a limited nu mber of nodes without compromising the integrity of the transactions.
- **Consistency:** All honest nodes maintain a consistent view of the transaction history, ensuring trust and reliability in the inter-bank system.
- **Availability:** The system can continue to operate and process transactions even when facing failures or attacks.

Conclusion:

In a distributed inter-bank transaction system, Byzantine agreement plays a critical role in ensuring the security and reliability of financial operations. By enabling nodes to reach consensus despite failures and malicious behaviour, BFT algorithms maintain the integrity of the system and protect the financial interests of all participants.

7 -- b) Analyze the implications of achieving consensus in a distributed cryptocurrency exchange system in the presence of faulty or malicious nodes.

In a distributed cryptocurrency exchange system, achieving consensus is crucial for maintaining a consistent, reliable, and secure trading environment. However, the presence of faulty or malicious nodes introduces complexities and challenges that need careful consideration. Let us analyze the implications:

Challenges:

- **Double-spending:** Malicious nodes may attempt to spend the same cryptocurrency units twice, defrauding other users or the exchange itself.
- **Transaction Manipulation:** Faulty or malicious nodes could attempt to alter transaction details, such as the sender, recipient, or amount, leading to financial losses or disputes.
- **Denial-of-Service Attacks:** Malicious actors may try to disrupt the system's operation by flooding it with spurious transactions or messages, impacting legitimate users and hindering trading activities.
- **51% Attacks:** In Proof-of-Work based systems, if a single entity gains control of the majority of the network's hashing power, they could potentially manipulate the blockchain, reverse transactions, or double-spend.

Implications of achieving consensus:

- **Security:** Consensus mechanisms, like Proof-of-Stake (PoS) or Byzantine Fault Tolerant (BFT) algorithms, ensure that the majority of honest nodes agree on the validity and order of transactions, making it difficult for malicious actors to manipulate the system. This enhances the overall security and trustworthiness of the exchange.
- **Consistency:** Achieving consensus guarantees that all nodes maintain a consistent view of the ledger, preventing discrepancies and ensuring the integrity of account balances and transaction history. This consistency is crucial for smooth operation and user confidence.
- **Fault Tolerance:** Consensus mechanisms allow the system to continue functioning even if some nodes fail or behave maliciously. This fault tolerance ensures the availability and resilience of the exchange, minimizing downtime and potential financial losses.
- **Decentralization:** Distributed consensus protocols promote decentralization by removing the need for a central authority to validate transactions. This reduces the risk of single points of failure and censorship, fostering a more open and transparent trading environment.

Trade-offs:

- **Scalability:** Achieving consensus can be computationally expensive and time-consuming, potentially impacting the scalability of the exchange, especially with a large number of nodes and high transaction volume.
- **Performance:** The time it takes to reach consensus can introduce latency in transaction processing, affecting the user experience, particularly in high-frequency trading scenarios.
- **Complexity:** Implementing and maintaining robust consensus mechanisms requires significant technical expertise and resources, adding complexity to the development and operation of the exchange.

Overall:

Achieving consensus in a distributed cryptocurrency exchange system with faulty or malicious nodes is vital for maintaining security, consistency, and fault tolerance. While challenges and trade-offs exist, the benefits of a decentralized, secure, and reliable trading environment outweigh the costs, making consensus mechanisms essential for the long-term success and sustainability of cryptocurrency exchanges.

8 -- a) Explain how data indexing in P2P networks can improve search efficiency in a distributed file-sharing application.

In a distributed P2P file-sharing application, where files are scattered across numerous nodes, efficiently locating desired content can be a significant challenge. This is where data indexing plays a crucial role in improving search efficiency.

Challenges of Search in P2P Networks:

- **Decentralization:** Without a central server, searching for specific files requires querying multiple nodes, which can be time-consuming and resource-intensive.
- **Dynamic Nature:** Nodes constantly join, leave, and update their file lists, making it difficult to maintain an accurate and up-to-date picture of available content.
- **Heterogeneity:** Files may exist in various formats and with different metadata, making it challenging to perform accurate and relevant searches.

How Data Indexing Improves Search Efficiency:

1. **Distributed Indexing:** Instead of relying on a central index, each node maintains an index of the files it stores. This distributes the indexing workload and avoids a single point of failure.
2. **Metadata Management:** Indexing involves extracting and storing relevant metadata, such as file names, sizes, types, and keywords. This metadata helps in performing more precise and efficient searches based on specific criteria.
3. **Search Algorithms:** Specialized search algorithms, like Distributed Hash Tables (DHTs), can be used to efficiently route search queries to the nodes most likely to have the desired content. DHTs map keywords or file hashes to specific nodes, enabling direct retrieval without querying the entire network.
4. **Content-Based Search:** Indexing can enable content-based search, where files are indexed based on their actual content rather than just metadata. This allows users to find files even if they don't know the exact file name or keywords.
5. **Dynamic Updates:** Indexing mechanisms can be designed to automatically update the index as nodes join, leave, or modify their file lists. This ensures the index remains accurate and reflects the current state of the network.

Benefits of Data Indexing:

- **Faster Search:** By efficiently directing search queries to relevant nodes, indexing significantly reduces the time required to locate desired files.
- **Reduced Network Traffic:** Targeted searches minimize unnecessary network traffic, improving overall system performance and scalability.
- **Enhanced User Experience:** Efficient search capabilities provide a more user-friendly experience, allowing users to find the files quickly and easily they need.
- **Scalability:** Distributed indexing allows the system to scale effectively as the number of nodes and files grows without compromising search performance.

Conclusion:

Data indexing is crucial for enhancing search efficiency in P2P file-sharing applications. By enabling efficient location and retrieval of files across a distributed network, indexing improves user experience, scalability, and overall system performance.

8 -- b) Discuss the design challenges and security concerns in a P2P network used for a decentralized video streaming service and propose methods to mitigate these risks.

P2P video streaming offers advantages like reduced server load and potentially better-quality streams. However, it also comes with design challenges and security concerns:

Challenges:

- **Scalability and Efficiency:** Finding optimal paths for video chunks and balancing upload/download among peers with varying capacities can be complex.
- **Content Discovery:** Efficiently locating the desired video content across a dynamic P2P network requires robust search mechanisms.
- **Free Riding:** Some users might consume content without contributing upload bandwidth, affecting overall performance.

Security Concerns:

- **Copyright Infringement:** P2P networks can be susceptible to unauthorized sharing of copyrighted content.
- **Malicious Content:** There is a risk of malware or malicious streams being injected into the network.
- **Privacy Concerns:** User data like IP addresses and viewing habits might be exposed if not anonymized.

Mitigation Strategies:

- **Scalability and Efficiency:**
    - Implement chunk replication strategies to ensure availability.
    - Utilize intelligent algorithms to select optimal paths for video chunks based on peer capabilities and network conditions.
    - Encourage collaboration among peers for efficient content delivery.
- **Content Discovery:**
    - Utilize distributed hash tables (DHTs) for efficient content indexing and search.
    - Implement keyword-based search mechanisms with secure hashing to protect content privacy.
- **Free Riding:**
    - Implement reputation systems to reward contributing users and discourage free riding.
    - Throttle bandwidth allocation for non-contributing peers.
- **Copyright Infringement:**
    - Integrate digital watermarking techniques to identify unauthorized copies.
    - Implement content encryption and access control mechanisms.
- **Malicious Content:**
    - Utilize digital signatures to verify the authenticity of content source.
    - Implement peer reputation systems to identify and isolate malicious peers.
- **Privacy Concerns:**
    - Employ encryption techniques to anonymize user data during P2P communication.
    - Utilize privacy-preserving search mechanisms that don't reveal user search queries.

9 -- a) Explain how causal order of messages can be ensured in a distributed system used for online classroom sessions where messages need to reflect the sequence of lecture content delivery.

Ensuring causal order of messages in a distributed system used for online classroom sessions is essential to maintain the coherence and integrity of the lecture content delivery. Causal ordering ensures that messages are delivered in an order that reflects the causality of events, meaning that if event A causes event B, then event A must be delivered before event B.

Techniques for Causal Ordering:

1. **Vector Clocks:**
   - Each message carries a vector timestamp containing logical clocks for all processes (professors, students).
   - The clock value for the sending process is incremented for each message sent.
   - Receivers compare vector timestamps. A message can be delivered only if its timestamp is strictly greater than all previously delivered messages (considering each process's clock).
2. **Lamport timestamps:**
   - Each message carries a timestamp generated by the sender (a monotonically increasing counter).
   - Receivers compare timestamps. A message can be delivered only if its timestamp is greater than any previously delivered message's timestamp.
   - This method is simpler but might not capture all causal relationships (e.g., concurrent messages from different senders).
3. **Application-Level Ordering:**
   - The application itself (the online classroom platform) tracks message dependencies and enforces order based on content.
   - Messages indicating transitions between topics or steps in a lecture can be used as markers for causal ordering of subsequent messages related to that topic or step.

Choosing the Right Method:

- Vector clocks offer stronger guarantees but require more complex implementation.
- Lamport timestamps are simpler but might miss some causal dependencies.
- Application-level ordering can be efficient but relies on the application's logic to capture causal relationships accurately.

Additional Considerations:

- **Partial Ordering:** Not all messages might have a causal relationship. Efficient delivery of messages with no causal dependency can improve performance.
- **Scalability:** Techniques need to be scalable to handle a large number of participants in the online classroom session.

By using one or a combination of these techniques, online classroom systems can ensure messages are delivered in a way that reflects the natural flow of the lecture, enhancing the learning experience for students.

9 -- b) Design an application-level multicast protocol for a distributed event management system that handles thousands of events, ensuring total order of event-related messages with optimal efficiency.

This protocol ensures total order delivery of event messages for thousands of events within a distributed system, focusing on efficiency.

Components:

- **Event Broker:** Acts as a central coordinator for event subscriptions and message delivery.
- **Event Clients:** Applications or services interested in subscribing to and receiving messages related to specific events.

Protocol Flow:

1. **Event Registration:**
    - Event organizers register events with the Event Broker, providing details like event ID, name, and access control rules.
2. **Client Subscription:**
    - Clients subscribe to events by sending a "Subscribe" message to the Event Broker, specifying the desired event IDs.
    - The Event Broker maintains a subscription list for each event, containing subscribed clients.
3. **Event Message Delivery (Total Ordering):**
    - When an event organizer publishes a message related to an event (e.g., update, reminder), it sends the message (containing event ID, timestamp, and payload) directly to all subscribed clients for that event.
    - **Total Ordering:** To achieve total order delivery:
        - Each client maintains a sequence number, initially 0.
        - The message includes the event ID, a unique message ID, and the client's current sequence number.
        - Upon receiving a message, a client:
            - ➢ Increments its sequence number.
            - ➢ Buffers the message if it has received messages for the same event with a higher sequence number.
            - ➢ Delivers messages in order based on their sequence numbers within the event.
4. **Failure Handling:**
    - Clients periodically send heartbeat messages to the Event Broker.
    - If a client misses heartbeats, the Event Broker removes it from subscription lists.
    - Clients can re-subscribe after recovery.

Optimizations for Efficiency:

- **Batching:** Clients can batch acknowledgements for multiple received messages, reducing network traffic.
- **Filtering:** Clients can filter messages at their end based on additional criteria (e.g., importance) to avoid processing unnecessary data.
- **Content Delivery Networks (CDNs):** For geographically distributed clients, the Event Broker can utilize CDNs to cache and deliver event messages efficiently.

Considerations:

- **Security:** Implement authentication and authorization mechanisms to control access to events and prevent unauthorized message publishing.
- **Network Congestion:** Consider congestion control mechanisms if message volume for an event becomes overwhelming.

10 -- a) Illustrate the use of Ricart-Agrawala's algorithm in a distributed system managing access to a limited number of electric vehicle charging stations.

Ricart-Agrawala's algorithm is a distributed mutual exclusion algorithm used to coordinate access to shared resources among processes in a distributed system. In the context of managing access to electric vehicle (EV) charging stations, this algorithm can be used to ensure that only one EV is allowed to charge at a station at any given time, preventing conflicts and resource contention. Here is how the algorithm can be illustrated in this scenario:

1. **Initialization**:
   - Each EV charging station is considered a critical section or a shared resource that requires mutual exclusion.
   - Each EV charging station maintains a queue of requests from EVs wanting to charge.

2. **Requesting Access**:
   - When an EV arrives at a charging station and wants to charge, it sends a request message to all other charging stations in the distributed system, indicating its intention to access the station.
   - Upon receiving a request message, each charging station applies the Ricart-Agrawala algorithm to determine whether to grant access to the requesting EV.

3. **Responding to Requests**:
   - If a charging station receives a request message while it doesn't have any pending requests or is currently serving another EV, it immediately grants access to the requesting EV by sending a reply message.
   - If a charging station receives a request message while it has pending requests or is currently serving another EV, it compares the timestamp of the incoming request with its own timestamp. If the incoming request has a lower timestamp, indicating higher priority, the charging station defers its own request and sends a reply message to the requesting EV. Otherwise, it denies the request and adds it to its queue.

4. **Receiving Replies**:
   - Upon receiving a reply message from a charging station, the requesting EV checks whether it has received replies from all other charging stations. If it has received replies from all stations or has higher priority requests, it proceeds to access the charging station. Otherwise, it waits until all replies are received or until its priority increases.

5. **Releasing Access**:
   - Once an EV completes charging, it releases the charging station, allowing other EVs in the queue to access it based on their priority and timestamp.

By employing Ricart-Agrawala's algorithm in this manner, the distributed system managing access to electric vehicle charging stations can ensure that only one EV is allowed to charge at a station at any given time, preventing conflicts and ensuring fair access to the limited resources. Additionally, the algorithm provides a mechanism for handling concurrency and contention in a distributed environment, improving system efficiency and reliability.

10 -- b) Evaluate the performance of Maekawa’s algorithm in a distributed cloud environment where instances dynamically request and release multiple resources, discussing potential bottlenecks and improvements.

Maekawa's algorithm offers advantages like scalability and fairness in distributed mutual exclusion, making it a potential candidate for managing resource access in a dynamic cloud environment. However, its performance needs careful evaluation considering the dynamic nature of cloud deployments.

Strengths:

- **Scalability:** Maekawa's reliance on quorums, which are subsets of all processes, makes it suitable for large-scale cloud deployments compared to permission-based algorithms requiring messages from all nodes.
- **Fairness:** The algorithm ensures all processes have an opportunity to access resources, preventing starvation.

Weaknesses:

- **Message Overhead:** Maekawa's voting process can generate a significant number of messages for requesting and releasing resources, especially when dealing with multiple resources and frequent requests. This can impact network performance.
- **Quorum Selection:** Choosing optimal quorums is crucial. If quorums are too large, it increases message overhead. If too small, it might lead to deadlocks.
- **Dynamic Resource Requests:** Frequent creation and deletion of cloud instances can lead to frequent changes in quorum membership, requiring updates and reconfiguration.

Performance Bottlenecks:

- **Network Latency:** High network latency can significantly increase the time it takes to acquire and release resources, hindering application performance.
- **Deadlock Potential:** With complex resource dependencies and frequent changes, the chances of deadlocks where multiple processes wait for each other's resources can increase.

Potential Improvements:

- **Optimistic Locking:** Techniques like optimistic locking can be used to reduce message overhead. A process attempts to acquire a resource and validates ownership later. This can be efficient for scenarios with low contention.
- **Leases and Timeouts:** Leases can be assigned to resources, allowing temporary ownership and reducing deadlocks. Timeouts can be used to detect and recover from unresponsive processes.
- **Hierarchical Quorums:** Implementing hierarchical quorums can improve scalability. For frequently accessed resources, smaller quorums can be used, while larger quorums can be reserved for less frequently accessed resources or conflict resolution.
- **Caching:** Caching quorum membership information can reduce message overhead when requesting access to frequently used resources.

Alternative Approaches:

- **Centralized Resource Manager:** For specific use cases with predictable resource access patterns, a centralized resource manager might be more efficient than a fully distributed approach.
- **Hybrid Approach:** A hybrid approach combining elements of Maekawa's algorithm with optimistic locking or other techniques can offer a balance between scalability and performance.

Evaluation Considerations:

- **Workload Characteristics:** The performance of Maekawa's algorithm will depend on the specific workload. Frequent resource requests and complex dependencies will impact performance more significantly.
- **Network Conditions:** Network latency and bandwidth limitations can significantly affect message overhead and overall performance.

11 -- a) Describe a scenario where Chandy-Misra-Haas deadlock detection for the OR model could be used in a distributed system coordinating multiple autonomous vehicles at an intersection.

Imagine a busy four-way intersection managed by a distributed system coordinating multiple autonomous vehicles (AVs). Here's how CMH deadlock detection can be applied:

1. **OR Model and Resource Requests:** Each lane approaching the intersection can be modelled as a resource in the OR model (only one vehicle can occupy a lane at a time). Vehicles waiting to enter the intersection send out "request to enter" messages to a central coordinator or a designated "lead vehicle" acting as a local coordinator for that lane.
2. **Deadlock Potential:** Deadlock can occur if multiple AVs are waiting to enter the intersection and their desired entry paths create a circular dependency. For example:
   - Car A wants to enter lane L1 but needs car B to move from lane L2 (blocking L1).
   - Car B wants to enter lane L2 but needs car C to move from lane L3 (blocking L2).
   - Car C wants to enter lane L3 but needs car A to move from lane L1 (blocking L3).
3. **CMH Implementation:**
   - Each AV maintains a local state indicating its position and desired lane to enter.
   - When an AV encounters a situation where it cannot proceed (blocked by another vehicle), it initiates a CMH probe message.
   - The probe message contains the ID of the blocked AV and the lane it's waiting to enter.
   - The probe message is forwarded to the coordinators (central or designated lead vehicle) of the lanes requested by the blocked AV and subsequent vehicles in the potential deadlock chain.
4. **Deadlock Detection:**
   - If a probe message circles back to the originating AV, it signifies a deadlock has been detected. This cycle represents the circular dependency between waiting vehicles.
5. **Deadlock Resolution:**
   - Upon deadlock detection, the coordinator or designated lead vehicle can initiate a deadlock resolution protocol. This might involve:
     - Re-routing some vehicles to alternative lanes.
     - Implementing a priority scheme to allow specific vehicles to proceed first.
     - Utilizing negotiation strategies between the involved AVs.

Benefits:

- **Distributed Detection:** CMH allows for deadlock detection without a central authority, which is suitable for a distributed system coordinating autonomous vehicles.
- **Scalability:** The algorithm scales well with the number of vehicles approaching the intersection.

Limitations:

- **Message Overhead:** Probe messages can create additional network traffic, especially with frequent deadlocks or a high number of vehicles.
- **Delayed Detection:** Deadlock detection might not be instantaneous, depending on message propagation time.

The CMH algorithm offers a viable approach to detecting and resolving deadlocks in a distributed system of autonomous vehicles at an intersection. By efficiently coordinating resource allocation and resolving conflicts, this method contributes to ensuring smooth and safe traffic flow in a complex environment.

11 -- b) Develop a comprehensive deadlock resolution strategy for a distributed airline booking system where multiple agents may hold and wait for seats on various flights simultaneously.

A comprehensive deadlock resolution strategy for a distributed airline booking system where multiple agents may hold and wait for seats on various flights simultaneously should focus on preventing, detecting, and resolving deadlocks efficiently while minimizing disruptions to the booking process. Here's a step-by-step approach:

Prevention Techniques:

1. **Timeouts:** Implement timeouts for held reservations. If a confirmation for one flight isn't received within a specified timeframe, the system automatically releases the held seats on that flight, allowing other agents to book them. Timeouts should be long enough to allow for reasonable user interaction but short enough to prevent indefinite blocking.
2. **Lock Escalation:** Utilize a hierarchical locking mechanism. Initially, agents hold advisory locks (non-blocking) on seats. If multiple agents hold advisory locks for seats needed to complete a booking, the system can escalate the lock on one or more flights to a mandatory lock (blocking) for a limited time. This allows one agent to proceed while others wait briefly, reducing the likelihood of deadlocks.
3. **Deadlock Detection:** Implement a deadlock detection algorithm like Chandy-Misra-Haas (CMH) as a fallback. CMH allows distributed detection of circular dependencies between held reservations.

Resolution Techniques:

1. **Preemption:** If deadlock detection occurs, the system can preempt some held reservations. This involves selecting a reservation to cancel and releasing the associated seats. Preemption strategies can be based on factors like:
    - **Cost:** Cancel reservations with the lowest revenue impact.
    - **Waiting Time:** Cancel reservations that have been held for the longest time.
    - **Number of Held Seats:** Cancel reservations holding the fewest seats to impact fewer passengers.

2. **Rollback:** The system can rollback a portion of the booking process. This involves releasing held reservations for all flights associated with a deadlocked booking attempt. Agents would have to restart the booking process, potentially with updated availability information.
3. **Negotiation:** In some cases, the system can attempt to facilitate negotiation between agents involved in the deadlock. This could involve offering alternative flights or suggesting adjustments to the booking itinerary to break the circular dependency.

Additional Considerations:

- **Notification:** Inform agents about failed bookings due to deadlocks and suggest alternative options.
- **Logging and Monitoring:** Track deadlock occurrences and analyze the root cause to identify potential improvements to timeouts, lock escalation strategies, or overall system design.
- **Performance Optimization:** Regularly evaluate the impact of deadlock prevention and resolution techniques on system performance and user experience.

12 -- a) Explain how an agreement in a failure-free system can be quickly achieved in a distributed sensor network monitoring environmental conditions across various geographic locations.

In a failure-free distributed sensor network for environmental monitoring, achieving agreement on a specific value (e.g., average temperature across all sensors) can be optimized for speed using techniques that exploit the lack of failures:

1. Leader Election:

- Elect a single sensor node as the leader through a distributed election algorithm. This avoids the overhead of messages being sent to all nodes for agreement.
- Election algorithms like Bully Algorithm or Ring Election can be used.

2. Data Collection and Aggregation:

- All sensor nodes periodically measure the environmental condition (e.g., temperature).
- Each node sends its measurement directly to the leader node.

3. Agreement at the Leader:

- The leader node receives data from all operational sensor nodes.
- It calculates the agreed-upon value (e.g., average temperature) based on the received data.

4. Dissemination of Agreed Value:

- The leader broadcasts the agreed-upon value to all sensor nodes in the network.

Benefits:

- **Speed:** This approach minimizes message overhead compared to protocols where all nodes communicate with each other.
- **Scalability:** The leader can handle data aggregation even as the network grows (assuming the leader has sufficient resources).

Limitations:

- **Single Point of Failure:** If the leader fails, the system needs to re-elect a new leader, causing a temporary disruption in agreement.
- **Overhead of Leader Election:** The initial leader election process adds some overhead, but for frequently collected data, this becomes less significant.

Alternative for Very Large Networks:

- **Hierarchical Aggregation:** Divide the network into clusters with a leader for each cluster. Cluster leaders send aggregated data to a higher-level leader responsible for calculating the final agreed-upon value. This reduces load on a single leader in very large networks.

Assumptions:

This approach assumes a failure-free system. In a real-world scenario, incorporating failure detection and recovery mechanisms would be essential for long-term reliability.

12 -- b) Design a robust consensus protocol for a distributed blockchain system that operates in a message-passing synchronous system with possible node failures, ensuring that the system maintains integrity and prevents double-spending.

This protocol is designed for a synchronous system where messages are delivered reliably but with a fixed delay. It tolerates Byzantine failures, meaning nodes can crash, exhibit arbitrary behaviour, or even collude maliciously. This ensures integrity and prevents double-spending in the blockchain.

Components:

- **Nodes:** Each node maintains a copy of the blockchain and participates in the consensus protocol.
- **Leader:** In each round, a leader node is chosen (e.g., through rotating schedule or Byzantine fault tolerant leader election).
- **Rounds:** The protocol operates in rounds. Each round focuses on agreeing on a single block.

Protocol:

1. **Transaction Pool:** Nodes propose transactions to be included in the next block.
2. **Block Proposal:** The leader collects transactions and creates a new block proposal.
3. **Pre-vote Phase:** The leader broadcasts the block proposal to all nodes. Each node verifies the transactions and broadcasts a pre-vote message if they accept the block.
4. **Commit Phase:** If a node receives pre-vote messages from a majority (⅔ or more) of honest nodes, it broadcasts a commit message.
5. **Decision:** The leader waits for commit messages from a majority. If received, the block is finalized and appended to the blockchain. All honest nodes update their local copy.
6. **Leader Change:** If the leader doesn't receive enough pre-votes or commit messages within a timeout period, the round is considered failed. A new leader is chosen, and the process restarts from step 2.

Security Properties:

- **Agreement:** All honest nodes agree on the same final block in each round.
- **Liveness:** Even with failures, honest nodes eventually agree on a new block.
- **Integrity:** Only valid transactions are included in the blockchain.
- **Double-Spending Prevention:** Once a transaction is included in a finalized block, it cannot be reversed or spent again.

Byzantine Fault Tolerance:

- Malicious nodes can propose invalid blocks or send inconsistent messages.
- Pre-vote and commit phases ensure a majority of honest nodes agree before finalizing a block.
- Even if the leader is malicious, honest nodes will only pre-vote and commit on valid blocks.

Scalability:

- Leader election and message broadcasts can become bottlenecks in large networks.
- Practical Byzantine Fault Tolerance (PBFT) variants optimize leader change and message exchange for better scalability.

Additional Considerations:

- This is a simplified overview; real-world implementations have additional features like checkpointing and block finality guarantees.
- The choice of leader selection algorithm and timeout periods affect performance and security trade-offs.

This protocol offers a robust and secure foundation for distributed blockchain systems in synchronous environments.

13 -- a) Discuss the role of structured overlays, specifically CHORD DHT, in improving data retrieval in a peer-to-peer file-sharing application.

In peer-to-peer (P2P) file-sharing applications, traditional methods like flooding can be inefficient for data retrieval. Structured overlays, particularly Content-Addressable Hash Tables (CHORD DHT), offer significant improvements in this area.

Challenges of Unstructured Overlays:

- **Flooding:** Early P2P networks used flooding, where a search query is broadcast to all peers. This can lead to high network traffic and slow response times, especially in large networks.

- **Scalability:** As the network grows, flooding becomes less effective and finding data becomes increasingly difficult.

Benefits of CHORD DHT:

- **Structured Network:** CHORD creates a well-defined virtual ring where each node has a unique identifier. Files are mapped to specific nodes based on their hash value, ensuring efficient routing of search queries.

- **Lookup Efficiency:** When a user searches for a file, the query is hashed and directed towards the node responsible for that specific hash range. This significantly reduces the number of hops needed to find the desired data.

- **Scalability:** CHORD scales well as the network grows. New nodes can be easily integrated into the ring, maintaining efficient routing capabilities.

- **Fault Tolerance:** CHORD can handle node failures. If a responsible node is unavailable, the search query can be directed to neighbouring nodes, ensuring data retrieval even with churn (nodes joining and leaving frequently).

How CHORD Improves Data Retrieval:

1. **File Hashing:** Each file is assigned a unique hash value using a cryptographic hash function.

2. **Node Responsibility:** Nodes in the CHORD ring is responsible for storing files whose hash values fall within their assigned range.

3. **Search Routing:** When a user searches for a file, the search term is hashed. The query is then routed towards the node responsible for that specific hash range.

4. **Data Retrieval:** The responsible node checks if it has the requested file. If found, the data is sent directly to the user. If not, the query might be forwarded to neighbouring nodes based on the CHORD routing protocol.

13 -- b) Address the security concerns in a peer-to-peer network used for a decentralized identity verification system, proposing techniques to enhance security against sybil attacks and data tampering.

In a peer-to-peer network used for decentralized identity verification, security concerns such as sybil attacks and data tampering are significant challenges that need to be addressed to ensure the integrity and reliability of the system. Here are some techniques to enhance security against these threats:

1. **Proof of Work (PoW) or Proof of Stake (PoS)**:
   - Implement a consensus mechanism like PoW or PoS to prevent sybil attacks. These mechanisms require nodes to prove their computational or stake-based investment in the network before participating in identity verification processes.
   - PoW requires nodes to perform computationally intensive tasks, making it costly for attackers to create multiple identities.
   - PoS requires nodes to stake a certain amount of cryptocurrency, which is at risk of forfeiture in case of malicious behaviour, discouraging sybil attacks.
2. **Identity Reputation Systems**:
   - Develop reputation systems that assign trust scores to nodes based on their behaviour and history within the network.
   - Nodes with higher reputation scores can be given more authority in the identity verification process, while those with lower scores may face stricter scrutiny or restrictions.
   - Reputation systems incentivize honest behaviour and discourage sybil attacks by making it more difficult for malicious nodes to gain trust.
3. **Identity Verification Challenges**:
   - Introduce challenges or puzzles during the identity verification process to ensure that participating nodes are genuine and not sybil identities.
   - These challenges can be designed to require real-world resources or time, making it impractical for attackers to create multiple identities to manipulate the system.
4. **Decentralized Ledger with Immutable Records**:
   - Utilize a blockchain or similar decentralized ledger technology to store identity verification records in a tamper-evident and immutable manner.
   - Each identity verification transaction should be cryptographically signed and recorded on the ledger, ensuring that it cannot be altered or tampered with retroactively.
   - Immutable records enhance security against data tampering and provide a transparent audit trail for identity verification activities.
5. **Peer Reputation and Consensus**:
   - Implement a peer-to-peer reputation system where nodes collectively vote on the legitimacy of identity verification requests and outcomes.
   - Consensus among multiple nodes can be required to finalize identity verification decisions, reducing the influence of potentially malicious or sybil nodes.
   - Reputation-based voting ensures that only trusted nodes are involved in the identity verification process, enhancing security and reliability.
6. **Randomized Peer Selection**:
   - Randomly select peers for participating in identity verification processes to reduce the likelihood of collusion among malicious nodes.
   - Randomized selection makes it difficult for attackers to predict which nodes will be involved in verification activities, thereby mitigating the impact of sybil attacks and collusion attempts.

14 -- a) Demonstrate how snapshot recording algorithms for non-FIFO channels can be implemented in a distributed multi-player gaming system where players interact and make moves in real-time.

Snapshot recording for non-FIFO (First In, First Out) channels in distributed computing systems presents unique challenges compared to FIFO systems. In non-FIFO channels, messages can arrive out of the order they were sent, which complicates the process of obtaining a consistent global snapshot.

Relevance in Non-FIFO Channels:

1. **Handling Out-of-Order Messages**: The primary challenge in non-FIFO systems is dealing with the possibility that messages may arrive at a destination in a different order than they were sent. This can lead to inconsistencies in the global state if not properly managed during the snapshot process.
2. **Complex Snapshot Algorithms:** Snapshot algorithms for non-FIFO systems must account for the possibility of out-of-order message delivery. They often need additional mechanisms to track the state of the messages in transit and determine whether they were sent before or after the snapshot initiation.
3. **Increased Overhead:** Due to the additional complexity in tracking and managing messages, snapshot algorithms for non-FIFO systems tend to have higher computational and communication overhead compared to FIFO systems.

To handle the inconsistencies, the snapshot algorithm might include additional steps, such as:

- **Recording the State of Messages in Transit**: The system might tag messages with timestamps or sequence numbers to identify their position relative to the snapshot.
- **Reconciliation Process**: After the initial snapshot, there might be a reconciliation phase where the processes exchange information to determine the status of in-transit messages at the time of the snapshot.

The Challenge with Non-FIFO

In non-FIFO channels, messages may be delivered out of the order they were sent. Why does this matter for snapshots?

- **Lost Causality:** A process might take a snapshot, then receive a message that was actually sent before the snapshot initiation. This creates a false image of the system's state at a given time, violating consistency.
- **"Impossible" States:** Consider a chat system where messages between users might arrive out of order due to non-FIFO channels. A snapshot might capture a state where a user seems to have replied to a message that, according to the snapshot, hasn't been received yet!

Practical Example: Distributed Game Lobby

1. **The State:** Imagine a multiplayer game where players can join a lobby. The system must track players, their readiness status, game preferences, etc. This information is constantly updated through messages across processes.
2. **Non-FIFO Issue:** Due to network conditions, messages might arrive out of order (e.g., a player's "ready" message might be received before the initial "player joined" message).
3. **Snapshot Needed:** You want a snapshot to analyse the state of the game lobby for matchmaking purposes.
4. **The Problem:** A naive snapshot could show players in the lobby that haven't yet signalled their readiness, or players marked as ready who haven't officially "joined" in the system's view. This inconsistent snapshot jeopardizes matchmaking.
5. **Non-FIFO Solution:** A snapshot algorithm suitable for non-FIFO channels will likely use piggybacking. Markers would include a sequence number or timestamp, allowing the reconstruction of a consistent state even if messages arrived out of their intended order.

How it Works (Simplified)
A non-FIFO snapshot algorithm in this system would likely:

1. **Attach Metadata:** Markers and messages would carry additional information (timestamps, sequence numbers, or dependency vectors) to help track causality.
2. **Reconstruction:** When the snapshot is collected, the algorithm would use this metadata to reorder events and messages logically, creating a consistent global view, even if the physical arrival order didn't reflect the true execution order.

14 -- b) Design a system model and define an algorithm for recording consistent global snapshots in a distributed cryptocurrency mining operation that must handle asynchronous and out-of-order transactions.

Designing a system for recording consistent global snapshots in a distributed cryptocurrency mining operation involves ensuring that the recorded snapshots accurately reflect the state of the blockchain at a given point in time, despite the asynchronous and out-of-order nature of transactions. Here's a system model and an algorithm for achieving this:

1. **System Model**:

- The system consists of a network of mining nodes responsible for validating transactions and adding blocks to the blockchain.
- Transactions are propagated through the network asynchronously and may arrive at nodes out of order.
- Each node maintains a local copy of the blockchain and participates in the mining process to extend the blockchain with new blocks.

2. **Algorithm for Recording Consistent Global Snapshots**:

Initialization:

- Each mining node maintains a vector clock to track the logical ordering of events and transactions.
- Initialize a snapshot marker variable to mark the beginning and end of the snapshot recording process.

Snapshot Initiation:

- At regular intervals or upon receiving a trigger signal, a node initiates a snapshot recording process.
- The initiating node broadcasts a snapshot marker message to all other nodes in the network.

Local State Recording:

- Upon receiving the snapshot marker message, each node records its local state:
  - Record the current state of the blockchain, including all validated transactions and the latest block.
  - Increment its own vector clock to mark the snapshot initiation event.

Transaction Recording:

- Asynchronously process incoming transactions during the snapshot recording process.
- Each node records incoming transactions along with their associated vector clock timestamps.
- Ensure that recorded transactions are tagged with vector clock timestamps to maintain causal ordering.

Snapshot Termination:

- After all nodes have recorded their local states and transactions, terminate the snapshot recording process.
- Broadcast an end marker message to signal the completion of snapshot recording.

Global Snapshot Reconstruction:

- Once all nodes have completed recording their states and transactions, reconstruct the global snapshot:
    - Collect recorded local states and transactions from all nodes.
    - Merge the recorded states and transactions while respecting the causal ordering defined by vector clocks.
    - Ensure that the reconstructed snapshot reflects a consistent global state of the blockchain at the time of snapshot initiation.

Consistency Checks:

- Perform consistency checks on the reconstructed global snapshot:
    - Verify that the recorded transactions adhere to the consensus rules of the cryptocurrency.
    - Validate the integrity and consistency of the blockchain structure.
    - Ensure that the reconstructed state is consistent across all participating nodes.

3. **Fault Tolerance**:
    - Implement fault tolerance mechanisms to handle node failures and network partitions during the snapshot recording process.
    - Use redundancy, replication, and consensus protocols to ensure that the snapshot recording process can proceed even in the presence of failures.

15 -- a) Describe how Suzuki-Kasami's broadcast-based algorithm could be used to synchronize access to a distributed ledger in a financial blockchain network.

Suzuki-Kasami's Broadcast (SKB) algorithm can be a foundation for synchronizing access to a distributed ledger in a financial blockchain network. Here's how it can be applied:

Challenges:

- **Consensus:** Multiple nodes (miners or validators) need to agree on the order of transactions included in the ledger.
- **Byzantine Failures:** Nodes might crash, exhibit arbitrary behavior, or collude maliciously.

SKB Algorithm in Blockchain Context:

1. **Transaction Proposal:** A node proposes a new transaction to be added to the ledger.
2. **Atomic Broadcast:** The node broadcasts the transaction using a variant of SKB adapted for blockchain:
   - The message includes the transaction data and an identifier originating from the proposing node.
   - The broadcast uses a reliable channel to ensure all honest nodes receive the message exactly once and in the same order.
3. **Echo Phase:**
   - Upon receiving the broadcast, each node timestamps the message with its local clock and re-broadcasts it.
   - This echo phase ensures all honest nodes have a copy of the transaction and the timestamps from other nodes.
4. **Agreement Phase:**
   - Nodes collect all the echoed messages, including their own.
   - They compare the timestamps associated with the transaction.
   - If a node receives echoes from a majority of honest nodes with consistent timestamps, it considers the transaction accepted and adds it to its local copy of the ledger.

Benefits for Blockchain:

- **Partial Ordering:** SKB enforces a partial order on transactions based on the message timestamps. This helps prevent conflicts and double-spending.
- **Byzantine Fault Tolerance:** Even with malicious nodes, as long as a majority are honest, the agreement phase ensures consistent transaction ordering across the network.

Limitations and Considerations:

- **Scalability:** SKB can become inefficient in large blockchain networks due to the message overhead in the echo phase. Alternative algorithms like PBFT (Practical Byzantine Fault Tolerance) might be more suitable for scalability.
- **Leader Election:** A leader node selection mechanism might be needed for the initial broadcast to improve efficiency.
- **Finality Guarantees:** SKB alone might not provide strong finality guarantees (ensuring a transaction cannot be reversed after a certain point). Additional mechanisms like block finality rules might be required.

Overall, Suzuki-Kasami Broadcast offers a foundation for Byzantine Fault Tolerance in blockchain networks. However, for real-world deployments, it's often combined with other protocols or optimizations to address scalability and finality concerns.

15 -- b) Propose modifications to Raymond's tree-based algorithm to enhance its performance and fault tolerance in a distributed content delivery network that dynamically adjusts to changes in traffic and node failures.

Here are some modifications to Raymond's tree-based algorithm to enhance its performance and fault tolerance in a dynamic Content Delivery Network (CDN):

1. Dynamic Tree Rebalancing:

- **Traffic-aware Reconfiguration:** The static tree structure is a major limitation of Raymond's algorithm. In a CDN with fluctuating traffic patterns, a balanced tree is crucial for efficient content delivery. Implement periodic or event-driven tree rebalancing based on real-time traffic data.
  - **Metrics:** Monitor metrics like content request frequency, bandwidth utilization, and node response times.
  - **Rebalancing Strategies:** Algorithms like neighbour-aware balancing or graph partitioning can be used to dynamically adjust the tree structure, ensuring nodes with high traffic are closer to the root (token holder) for faster access.

2. Fault Tolerance Improvements:

- **Replica Tokens:** A single point of failure exists if the token holder node fails. Introduce a mechanism with multiple replica tokens distributed across designated nodes. This ensures the system remains functional even if the current token holder fails.
- **Leases and Heartbeats:** Assign leases to token holders with a defined expiry time. Nodes can periodically exchange heartbeat messages. If a node does not receive a heartbeat within the lease period from the token holder, it suspects a failure and initiates recovery procedures:
  - **Backup Paths:** Pre-establish backup paths between nodes. When a primary path fails, nodes can access the privilege (token) through a pre-defined backup path leading to another replica token holder.
  - **Leader Election:** Alternatively, the node can trigger a leader election process to choose a new replica token holder from among the remaining healthy nodes.

3. Leader Election and Scalability:

- **Leader Rotation:** A static leader can become a bottleneck in large CDNs. Implement a leader election mechanism. This can be:
  - **Periodic Election:** Elect a new leader at regular intervals to distribute the load.
  - **Work-based Election:** Choose the leader based on factors like workload or processing power, ensuring the most capable node takes on the responsibility.
- **Hierarchical Trees:** For very large CDNs with a vast number of nodes, consider a hierarchical tree structure. Sub-trees can have their own leaders, reducing the load on a single root node and improving overall scalability.

4. Performance Optimizations:

- **Batching Requests:** Sending individual requests for the token can lead to message overhead. Allow nodes to batch requests and send them together, reducing network traffic. Define a threshold for batch size based on network conditions.
- **Caching Mechanisms:** Frequently accessed content can be cached at each node, reducing the need to access the root for every request. Implement a cache invalidation strategy to ensure cached content remains up-to-date.

By incorporating these modifications, Raymond's tree-based algorithm can be transformed into a more robust and adaptable solution for distributed content delivery, capable of handling dynamic traffic patterns and potential node failures in a scalable and efficient manner.