

UNIT III

Classical Distributed Algorithms:

What is Global Snapshot?

The term "Global Snapshot" in cloud computing can have two different meanings, depending on the context:

1. System state capture:

In the context of distributed systems and cloud infrastructure, a "Global Snapshot" can refer to a technique for capturing the complete state of a system at a specific point in time. This snapshot includes the state of all individual processes or nodes within the system, as well as any messages in transit between them.

There are different algorithms for taking global snapshots, such as the Chandy-Lamport algorithm. These algorithms ensure that the captured snapshot is consistent, meaning it accurately reflects the state of the system at the precise moment the snapshot was taken.

Global snapshots are useful for various purposes, such as:

- **Deadlock detection:** By analyzing the snapshot, it's possible to identify cycles in the system's dependency graph, which can indicate a deadlock situation.
- **System recovery:** If a system failure occurs, the snapshot can be used to restore the system to its previous state, minimizing downtime and data loss.
- **Performance analysis:** By taking snapshots over time, it's possible to understand the system's behavior and identify bottlenecks or areas for improvement.

2. Cloud storage resource:

In some cloud platforms, particularly Google Cloud, the term "Global Snapshot" is also used to refer to a specific type of storage resource for persistent disks. These snapshots are essentially point-in-time copies of a disk, allowing you to revert to previous states if needed.

Global Snapshots for storage disks offer features like:

- **Geo-redundancy:** Snapshots are automatically stored in multiple data centers, ensuring high availability and disaster recovery.

- **Sharing:** Snapshots can be easily shared across projects and accounts within the same cloud platform.
- **Versioning:** You can keep multiple snapshots of the same disk, allowing you to roll back to different versions if necessary.

Global Snapshot Algorithm

Global Snapshot Algorithms are a class of algorithms designed to capture a consistent, system-wide snapshot of a distributed system's state at a specific point in time. This includes the states of all processes within the system and any messages in transit between them. These algorithms are crucial for various tasks in distributed systems, including:

Key Purposes:

- **Deadlock Detection:** Identifying cycles in resource dependencies that can lead to deadlocks.
- **System Recovery:** Restoring a system to a previous consistent state after a failure.
- **Performance Analysis:** Understanding system behavior and identifying bottlenecks.
- **Debugging:** Tracking down errors or inconsistencies in a distributed system.
- **Checkpointing:** Creating a stable point for restarting long-running computations.
- **Distributed Database Consistency:** Ensuring data consistency across multiple nodes.

Common Algorithm:

- **Chandy-Lamport Algorithm (CLA):**
 - Initiated by any process.
 - Uses marker messages to create a consistent cut across processes.
 - Records local state and pending messages upon receiving a marker.
 - Propagates markers along outgoing channels.
 - Terminates when all processes have received markers on all incoming channels.

Other Notable Algorithms:

- **Lai-Yang Algorithm:** Improved efficiency for large-scale systems.

- Manivannan-Singhal Algorithm: Addresses message logging overhead.
- Kshemkalyani-Singhal Algorithm: Handles dynamic process groups.

Key Considerations:

- Message Ordering: Assumes FIFO (First-In, First-Out) message delivery.
- Channel Reliability: Relies on reliable communication channels.
- Process Failures: May require additional mechanisms for handling process crashes.
- Overhead: Snapshot algorithms can introduce overhead due to message logging and coordination.

Applications:

- Cloud Computing: System recovery, performance analysis, debugging.
- Distributed Databases: Consistency management, checkpointing.
- Fault-Tolerant Systems: State recovery, failure detection.
- Debugging and Monitoring: Tracking system behavior.

Conclusion:

Global snapshot algorithms play a vital role in managing and understanding the state of distributed systems. Choosing the appropriate algorithm depends on factors such as system size, message delivery guarantees, and fault tolerance requirements.

Consistent Cuts

Consistent cuts are a fundamental concept in distributed systems and cloud computing, particularly when dealing with global snapshots. Understanding them is crucial for ensuring data integrity and system stability. Here's a breakdown:

What are they?

- A consistent cut represents a specific point in time across a distributed system, where the state of all processes and messages in transit are captured simultaneously.
- It's like taking a snapshot of the entire system at a precise moment, ensuring all ongoing activities are reflected accurately.

Why are they important?

- Inconsistent cuts can lead to data inconsistencies and malfunctioning systems. Imagine capturing a snapshot when a file transfer is halfway through, you'd have an incomplete and unusable file.
- Consistent cuts guarantee integrity and correctness. They're vital for various tasks like:
 - Checkpointing: Saving a consistent state of the system for later recovery.
 - Performance Analysis: Analyzing system behavior at a specific point in time.
 - Deadlock Detection: Identifying resource dependencies that might lead to deadlocks.
 - Distributed Database Consistency: Maintaining consistent data across multiple nodes.

How are they achieved?

- Various algorithms implement consistent cuts, one of the most popular being the Chandy-Lamport Algorithm (CLA).
- CLA uses marker messages to delineate the cut point. All processes record their local states and pending messages upon receiving a marker.
- The algorithm ensures all messages sent before the cut have been recorded, while messages sent after the cut are ignored.
- This coordinated action guarantees a consistent snapshot across the entire system.

Key properties of consistent cuts:

- Causality: The cut respects causal relationships between events. Messages sent before the cut are guaranteed to be received before the cut on any other process.
- Atomicity: The cut captures the system state as a single, indivisible event. There's no gap or overlap between snapshots.
- Completeness: The cut includes the state of all processes and messages within the system at the chosen point in time.

Understanding consistent cuts is crucial for anyone working with distributed systems, cloud computing, or data consistency. They ensure accurate system states, facilitate recovery, and enable insightful analysis

Safety and Liveness

Safety and liveness are two fundamental properties that are often used to describe the behavior of distributed systems, including cloud computing systems. Let's break them down:

Safety:

- Intuitive meaning: "Nothing bad happens."
- Formal definition: A property that guarantees the system will never reach a bad state or violate a specific requirement.
- Examples:
 - Mutual exclusion: Guarantees that only one process can access a shared resource at a time, preventing data corruption.
 - Crash safety: Ensures the system remains consistent even if a process crashes.
 - Liveness properties can sometimes imply safety properties. For example, eventual consistency implies that data inconsistencies will eventually be resolved without user intervention.

Liveness:

- Intuitive meaning: "Something good eventually happens."
- Formal definition: A property that guarantees the system will eventually make progress and achieve a desired outcome.
- Examples:
 - Termination: Guarantees that the system will eventually finish a computation or reach a stable state.
 - Progress: Ensures that processes make progress and don't get stuck waiting for other processes.
 - Responsiveness: Guarantees that the system reacts to external requests within a reasonable amount of time.

Key differences:

- Focus: Safety is concerned with what the system does not do, while liveness is concerned with what the system does do.
- Guarantee: Safety guarantees prevent bad states, while liveness guarantees eventual progress.
- Complexity: Proving safety properties is often easier than proving liveness properties.

In cloud computing:

- Both safety and liveness are crucial for ensuring reliability and availability of services.
- Safety properties such as data consistency and fault tolerance are vital for protecting user data and preventing service disruptions.
- Liveness properties such as responsiveness and high availability ensure that services remain accessible and performant for users.

Conclusion:

Understanding safety and liveness is essential for anyone working with distributed systems or cloud computing. By ensuring both properties, you can build reliable and dependable systems that meet user expectations.

Multicast Ordering

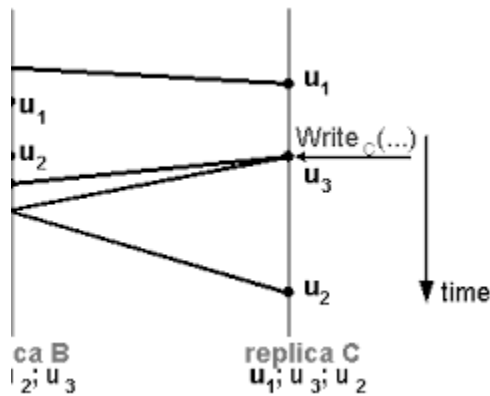
Multicast Ordering in Distributed Systems

Multicast is a communication paradigm where a single message is sent to a group of recipients simultaneously. However, in distributed systems, ensuring the order in which recipients receive these messages can be crucial for various applications. This is where multicast ordering comes in.

What is Multicast Ordering?

Multicast ordering refers to the different ways messages are delivered to recipients in a multicast communication, ensuring a specific order among them. This order can be:

- FIFO (First-in-First-out): Messages are delivered in the same order they were sent.



[Opens in a new window](#) 

www.researchgate.net

FIFO Ordering in cloud computing

- Causal: Messages are delivered respecting the causal dependencies between them. If message A causally precedes message B (e.g., A updates data used by B), then B is received only after A.

Message delivery rules; causal delivery

- The communication channel abstraction makes no assumptions about the order of messages; a real-life network might reorder messages.
- First-In-First-Out (FIFO) delivery → messages are delivered in the same order they are sent.
- Causal delivery → an extension of the FIFO delivery to the case where a process receives messages from different sources.
- Even if the communication channel does not guarantee FIFO delivery, FIFO delivery can be enforced by attaching a sequence number to each message sent. The sequence numbers are also used to reassemble messages out of individual packets.

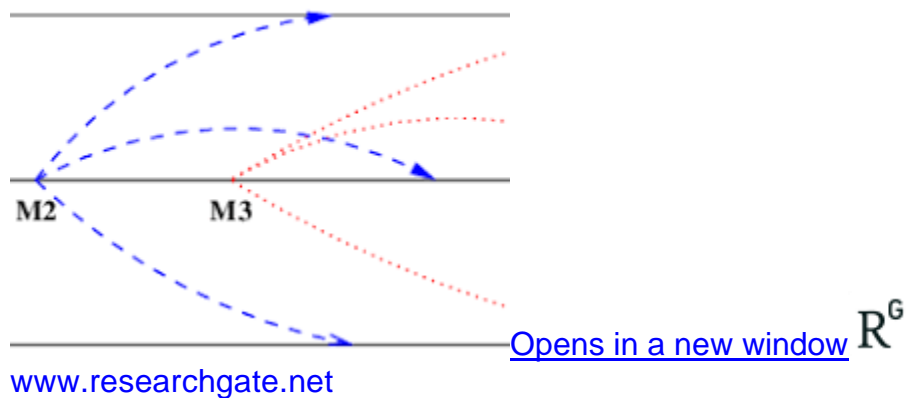
San C. Menezes

Cloud Computing: Theory and Practice
Chapter 7

[Opens in a new window](#)  slideplayer.com

Causal Ordering in cloud computing

- Total: Messages are delivered in the same order by all recipients, regardless of who sent them or the communication paths taken.



Total Ordering in cloud computing

Choosing the appropriate ordering depends on the specific application requirements. For example, FIFO ordering might be necessary for maintaining consistency in replicated databases, while causal ordering might be sufficient for distributed logging systems.

Why is Multicast Ordering Important?

Maintaining a specific order in message delivery can be crucial for various reasons:

- **Data consistency:** Ensures updates to shared data occur in the correct order across all recipients, preventing conflicting or corrupted data.
- **Causal relationships:** Preserves the relationships between events that happen across different processes, enabling accurate reasoning and analysis.
- **Synchronization:** Coordinates actions among multiple processes, preventing unexpected behavior or race conditions.
- **Deadlock prevention:** Helps avoid situations where processes wait for each other's messages indefinitely due to unpredictable delivery order.

Implementing Multicast Ordering

Achieving different types of multicast ordering can be challenging in distributed systems due to network delays and asynchronous communication. Different algorithms and protocols have been developed to provide varying levels of ordering guarantees.

Here are some examples:

- **FIFO Ordering:**

- Total Order Broadcast (TOB): Uses a central sequencer to assign unique sequence numbers to messages, ensuring delivery in the assigned order.
- Lamport timestamps: Each process maintains a logical clock and timestamps messages before sending. Recipients order messages based on timestamps and process IDs.
- Causal Ordering:
 - Vector clocks: Each process maintains a vector clock with entries for other processes, recording the number of messages it has received from each. Messages are delivered based on their causal relationships reflected in the vector clocks.
 - Happens-before relationship: Messages directly related by causal dependency are delivered accordingly.
- Total Ordering:
 - Byzantine Fault Tolerance (BFT): Replicates information and uses Byzantine agreement protocols to ensure consensus on message order even in the presence of faulty processes.

The choice of algorithm depends on factors like system size, performance requirements, and fault tolerance needs.

Implementing Multicast Ordering

Implementing Multicast Ordering in Distributed Systems

While multicast ordering is crucial for many distributed applications, achieving it poses challenges due to the complexities of asynchronous communication and network delays. Here's a breakdown of common implementation approaches:

1. FIFO (First-in-First-out) Ordering:

- Total Order Broadcast (TOB):
 - Employs a central sequencer responsible for assigning unique sequence numbers to messages.
 - Ensures all recipients deliver messages in the same, globally agreed-upon order.
 - Potential bottleneck and single point of failure at the sequencer.
- Lamport Timestamps:
 - Each process maintains a logical clock and timestamps messages before sending.

- Recipients order messages based on their timestamps and process IDs.
- Simpler than TOB, but doesn't guarantee total ordering in all cases.

2. Causal Ordering:

- Vector Clocks:
 - Each process maintains a vector clock, a data structure tracking the number of messages received from other processes.
 - Messages are delivered based on their causal relationships reflected in the vector clocks.
 - Overhead for maintaining vector clocks and potential complexity in understanding causal relationships.
- Happens-Before Relationship:
 - Direct causal dependencies between messages are explicitly tracked and enforced.
 - Simpler than vector clocks, but requires clear understanding of causal relationships in the application.

3. Total Ordering:

- Byzantine Fault Tolerance (BFT):
 - Replicates information and uses Byzantine agreement protocols to ensure consensus on message order.
 - Provides total ordering guarantees even in the presence of faulty processes.
 - High overhead due to replication and consensus mechanisms.

Key Considerations for Implementation:

- Application Requirements: Choose the ordering type that aligns with your application's specific needs (e.g., FIFO for consistency, causal for accuracy, total for global agreement).
- System Size: Consider the scalability of the algorithm as the number of processes increases.
- Performance: Evaluate the latency and throughput overhead introduced by the ordering algorithm.
- Fault Tolerance: Assess the algorithm's resilience to process failures and network partitions.
- Implementation Complexity: Consider the difficulty of implementing and maintaining the algorithm.

Additional Factors:

- **Network Infrastructure:** The underlying network characteristics can influence the effectiveness of ordering algorithms.
- **Security:** Protect against malicious actors attempting to disrupt message ordering.
- **Monitoring and Debugging:** Implement tools to track message delivery and identify potential ordering issues.

Reliable Multicast

Reliable multicast plays a crucial role in distributed systems and cloud computing, ensuring messages reach all intended recipients accurately and completely. It builds upon the base concept of multicast, where a single message is sent to multiple destinations simultaneously, but adds mechanisms for guaranteeing delivery despite network challenges like packet loss, delays, and reordering.

Why is reliable multicast important?

- **Data consistency:** Ensures replicated data remains consistent across multiple receiving nodes, crucial for databases, file systems, and collaborative applications.
- **Fault tolerance:** Guarantees message delivery even when some network components or processes fail, improving system reliability and availability.
- **Application efficiency:** Reduces the need for individual unicast transmissions to each recipient, saving bandwidth and processing resources.

How does it work?

Reliable multicast protocols implement various techniques to achieve accurate and complete delivery:

- **Negative acknowledgments (NACKs):** Receivers inform the sender of missing messages, triggering retransmission.
- **Sequence numbers:** Messages are tagged with unique identifiers for ordered delivery and duplicate detection.
- **Error correction:** Forward error correction (FEC) codes can reconstruct lost or corrupted data without retransmission.
- **Congestion control:** Mechanisms adjust transmission rates to avoid overwhelming receivers and network congestion.

Different types of reliable multicast:

- Tree-based protocols: Build a virtual tree structure for efficient message propagation and delivery guarantees.
- Mesh-based protocols: Allow flexible routing paths and redundancy in case of node failures.
- NACK-oriented protocols: Utilize NACKs as the primary feedback mechanism for retransmission.
- Time-based protocols: Employ timers and timeouts to manage retransmission and fault detection.

Challenges and considerations:

- Scalability: Maintaining efficiency and reliability as the number of receivers grows can be challenging.
- Latency: Overhead introduced by error correction and retransmission mechanisms can impact latency.
- Complexity: Choosing and implementing an appropriate protocol for specific application requirements demands careful consideration.

Examples of applications:

- Distributed file systems: Replicating files and updates across multiple servers for increased availability and access.
- Video conferencing: Delivering audio and video streams to multiple participants reliably and in real-time.
- Software updates: Distributing software updates to a large number of devices efficiently and consistently.

Reliable multicast continues to evolve with advancements in network technologies and distributed system architectures. Understanding its principles and potential benefits is crucial for developers and researchers working on robust and efficient communication patterns in today's complex computing landscapes.

Virtual Synchrony

Virtual Synchrony (VS) is a powerful concept in distributed systems that provides a framework for managing the state of distributed processes as if they were operating in a perfectly synchronized manner, even in the presence of network delays, failures, and unpredictability.

Here's a breakdown of key aspects of virtual synchrony:

1. Reliable Group Communication:

- VS relies on a reliable multicast protocol to ensure messages are delivered to all group members, even if some processes fail or experience delays.
- This foundation guarantees that all members have a consistent view of the group's communication history.

2. Virtual View Synchrony:

- Processes maintain a virtual view of the group's membership and message delivery order.
- This view may differ from the actual physical events due to delays or failures, but it ensures that all members perceive a consistent state.

3. Membership Changes:

- VS gracefully handles membership changes (processes joining or leaving the group) without disrupting the virtual synchrony guarantees.
- This allows for dynamic and adaptable distributed systems.

4. Failure Detection:

- VS systems have mechanisms to detect process failures and exclude failed members from the group communication.
- This prevents inconsistencies and ensures the remaining members continue to operate correctly.

5. Virtual Synchrony Abstraction:

- VS provides a higher-level abstraction for application developers, hiding the complexities of underlying network issues and membership changes.
- This simplifies the development of distributed applications that require strong consistency guarantees.

Benefits of Virtual Synchrony:

- **Consistency:** Ensures all members have a consistent view of the system state, even in the face of delays and failures.
- **Fault Tolerance:** Makes systems more resilient to individual process failures.
- **Coordination:** Simplifies coordination among distributed processes for tasks like synchronization, replication, and consensus.

Applications of Virtual Synchrony:

- **Distributed Database Replication:** Maintaining consistency among replicated database nodes.
- **Transaction Processing Systems:** Ensuring atomicity and consistency of distributed transactions.
- **Collaborative Systems:** Supporting real-time collaboration among multiple users.
- **Fault-Tolerant Systems:** Building systems that can recover from failures without losing data or consistency.

Challenges of Virtual Synchrony:

- **Complexity:** Implementing VS protocols can be complex and require careful consideration of network conditions and failure scenarios.
- **Performance Overhead:** Maintaining virtual synchrony can introduce some overhead in terms of message complexity and processing time.

Virtual synchrony continues to be an active area of research, with advancements in algorithms and protocols aiming to improve efficiency, scalability, and fault tolerance in large-scale distributed systems.

The Consensus Problem

The Consensus Problem: Reaching Agreement in Distributed Systems

The consensus problem is a fundamental and well-known challenge in distributed systems. It involves a group of independent processes (agents, nodes) that need to agree on a single value despite uncertainties and potential failures. Imagine a group of friends trying to decide on a restaurant in a city they've never visited. They need to reach a consensus on a single choice, but each has their own preferences and might not receive everyone's suggestions due to network limitations or misunderstandings.

Formal Definition:

Given a set of n processes with initial values in some domain D , the consensus problem requires them to:

1. **Agree:** All processes eventually decide on the same value v in D .
2. **Validity:** If one process decides on a value v , then some other process proposed v .

3. Termination: All correct processes eventually decide on a value.

These properties ensure consistency, agreement, and progress within the group.

Challenges and Implications:

The consensus problem becomes particularly difficult in distributed systems because we face:

- Uncertainties: Processes might have different initial values or receive incomplete information.
- Failures: Processes can crash, become slow, or send incorrect messages.
- Asynchronous communication: Messages might arrive in unpredictable orders or experience delays.

These challenges make reaching consensus a complex task with significant implications for distributed systems design.

Different Types of Consensus Problems:

- Byzantine Fault Tolerance (BFT): Tolerates even malicious failures where processes can actively lie or crash.
- Crash Fault Tolerance: Deals with simpler crash failures where processes stop working unexpectedly.
- Value Agreement: Processes agree on a single value from a predefined set.
- Leader Election: Electing a single leader amongst the processes for coordinated actions.

Approaches to Solving the Consensus Problem:

- Synchronous Systems: Assume partially synchronized clocks and bounded message delays, allowing for deterministic algorithms like Paxos or Raft.
- Asynchronous Systems: More realistic but require probabilistic or randomized algorithms due to unpredictable timing and failures. Examples include Byzantine Fault Tolerance protocols or gossip-based approaches.

Applications of Consensus:

- Distributed databases: Maintaining data consistency across replicated copies.
- Transaction processing: Ensuring atomicity and consistency of financial transactions.
- Leader election: Choosing a single coordinator for distributed tasks.

- State machine replication: Keeping all nodes in a distributed system in sync.

Understanding the consensus problem and its complexities is crucial for anyone working with distributed systems. Choosing the right approach depends on the specific system requirements, fault tolerance needs, and performance considerations.

Consensus In Synchronous Systems

In synchronous systems with bounded delays and crash faults, achieving consensus (agreement on a single value) is simpler than in asynchronous systems. Here's a quick breakdown:

Advantages:

- Deterministic: Algorithms can guarantee reaching consensus due to the predictable environment.
- Simpler implementation: Protocols can be less complex compared to asynchronous settings.
- Faster operation: Bounded delays and synchronized clocks enhance message delivery and decision-making speed.

Common algorithms:

- Byzantine Fault Tolerance (BFT): Tolerates even malicious failures (processes can lie or crash).
- Paxos: A family of algorithms offering high efficiency and liveness guarantees.
- Raft: A popular leader-based algorithm known for its simplicity and fault tolerance.

Drawbacks:

- Idealized setup: Perfect synchrony is often unrealistic in real-world distributed systems.
- Scalability constraints: Maintaining clock synchronization can be challenging for large systems.
- Performance overhead: Deterministic algorithms may require additional communication compared to asynchronous approaches.

Overall, consensus in synchronous systems offers clear advantages in predictability and performance, but comes with limitations on scalability and realism.

Paxos

Paxos is a family of protocols used in distributed systems to solve the consensus problem, which involves a group of independent processes agreeing on a single value even in the face of uncertainties and failures. Think of it like a highly organized waiter ensuring everyone in a group decides on the same restaurant, even if some members are late or can't hear each other clearly.

Here's a simplified breakdown of Paxos:

The Participants:

- Proposer: Initiates the decision-making process by suggesting a value.
- Acceptors: Processes that vote on the proposed value.

The Phases:

1. Prepare phase: The proposer asks acceptors if they're free to accept a new value, based on their previous votes.
2. Accept phase: If enough acceptors are free, the proposer sends the chosen value for them to accept.
3. Commit phase: The proposer informs all acceptors that the value has been chosen.

Key Points:

- Agreement: All correct processes eventually agree on the same value.
- Liveness: Even if some processes fail, the system eventually makes a decision.
- Fault tolerant: Can handle crashes and network delays.

Different versions of Paxos exist, each with its own strengths and weaknesses. Choosing the right version depends on the specific needs of your distributed system.

Here are some additional things to know about Paxos:

- It's a complex protocol and can be difficult to understand in detail.
- It's typically used in situations where high consistency and fault tolerance are critical.
- It's not always the best choice for every distributed system problem.

Simply

The FLP Proof

The FLP (Fischer-Lynch-Paterson) Proof is a fundamental result in distributed computing that states the impossibility of achieving consensus in an asynchronous system with even one process failure. Let's break it down:

What is Consensus?

Consensus in distributed systems involves a group of processes agreeing on a single value despite uncertainties and failures. Imagine a group of friends deciding on a restaurant, needing to agree on one place even if some are late or their messages get lost.

Asynchronous Systems:

These systems lack guarantees on timing and message delivery. Messages can arrive in any order, take arbitrarily long, or even get lost. Think of your friends sending restaurant suggestions at random times over unreliable phone calls.

The FLP Proof:

This proof shows that in such a system, with just one process failure (e.g., a friend's phone dies), achieving consensus is impossible. No deterministic algorithm can guarantee all processes agree on the same value in all circumstances.

Why is it important?

The FLP Proof has significant implications for designing fault-tolerant distributed systems:

- Focus on partial solutions: Knowing consensus is impossible, developers focus on alternative solutions like Byzantine Fault Tolerance (BFT) protocols that offer strong consistency guarantees despite failures.
- Choose appropriate models: Understanding system assumptions (asynchronous vs. synchronous) is crucial for selecting suitable algorithms and techniques.
- Research directions: The FLP Proof motivates research in areas like probabilistic and randomized algorithms for consensus in asynchronous settings.

In short, the FLP Proof is a foundational result in distributed computing, highlighting the challenges of achieving consensus in unpredictable environments. It guides developers towards alternative solutions and motivates further research in fault tolerance for distributed systems.

Orientation Towards Cloud Computing Concepts: Some Basic Computer Science Fundamentals, Introduction

The Election Problem,

Ring Leader Election,

Election in Chubby and ZooKeeper,

Bully Algorithm,

Introduction and Basics,

Distributed Mutual Exclusion,

Ricart-Agrawala's Algorithm,

Maekawa's Algorithm and Wrap-Up