INF-3026

GAUHATI UNIVERSITY

Centre for Distance and Online Education

Third Semester

(Under CBCS)

M.Sc.-IT

Paper: INF 3026

DISTRIBUTED SYSTEM

Contents: **Page No.**

SLM Development Team:

HoD, Department of Computer Science, GU
Programme Coordinator, M.Sc.-IT, GUCDOE
Prof. Shikhar Kr. Sarma, Department of IT, GU
Dr. Khurshid Alam Borbora, Assistant Professor, GUCDOE
Dr. Swapnanil Gogoi, Assistant Professor, GUCDOE
Mrs. Pallavi Saikia, Assistant Professor, GUCDOE
Dr. Rita Chakraborty, Assistant Professor, GUCDOE
Mr. Hemanta Kalita, Assistant Professor, GUCDOE

Contributors:

Dr. Khurhid Alam Borbora Asstt. Prof., GUCDOE	(Block 1 : Units- 1,2,4,5 & 6) (Block 2 : Units- 1,2,3 & 4), (Block 3 : Unit- 6)
Mrs. Pinky Saikia Dutta Asstt. Prof., Dept. of Comp. & Engineering, G.C.U.	(Block 1 : Unit- 3)
Dr. Swapnanil Gogoi Asstt. Prof., GUCDOE	(Block 2 : Units- 5 & 6) (Block 3 : Unit- 5)
Dr. Nabanita Choudhury Asstt. Prof., Assam Down Town University	(Block 3 : Unit- 1)
Dr. Kshirod Sarmah Asstt. Prof., PDUAM	(Block 3 : Units- 2 & 4)
Dr. Dulumoni Das Asstt. Prof., Assam Down Town University	(Block 3 : Unit- 3)
Mr. Gunikhan Sonowal Asstt. Prof., Assam Down Town University	(Block 3 : Unit- 7)

Course Coordination:

Dr. Debahari Talukdar	Director, GUCDOE
Prof. Anjana Kakoti Mahanta	Programme Coordinator, GUCDOE Dept. of Computer Science, G.U.
Dr. Khurshid Alam Borbora	Assistant Professor, GUCDOE
Dr. Swapnanil Giogoi	Assistant Professor, GUCDOE
Mrs. Pallavi Saikia	Assistant Professor, GUCDOE
Dr. Rita Chakraborty	Assistant Professor, GUCDOE
Mr. Hemanta Kalita	Assistant Professor, GUCDOE
Mr. Dipankar Saikia	Editor SLM, GUCDOE

Content Editor:

Dr. Utpal Barman	Dean & Professor, Faculty of Computer Technology Assam Down Town University

Cover Page Designing:

Bhaskar Jyoti Goswami	GUCDOE
Nishanta Das	GUCDOE

ISBN: 978-81-982028-4-0
October, 2024

BLOCK- I

UNIT: 1

INTRODUCTION TO DISTRIBUTED SYSTEMS

Unit Structure:

1.1 INTRODUCTION

Distributed systems are collections of independent computers that appear to users as a single coherent system. These systems are designed to share resources, facilitate communication, and provide services seamlessly across multiple locations. They play a crucial role in various applications, from cloud computing to large-scale data processing.

Distributed systems are integral to modern computing, offering scalability, flexibility, and resilience. However, they also present unique challenges that require careful design and management to ensure effective operation. This unit will delve into these concepts, exploring the characteristics, challenges, and inherent limitations of distributed systems.

1.2 UNIT OBJECTIVES

After going through this unit, you will be able to:

- *understand* the Fundamentals of Distributed Systems.
- *understand* the characteristics of Distributed Systems.
- *analyse* Design Issues and Challenges.
- *understand* the various types of transparency in distributed systems.
- *identify* the Inherent Limitations.

1.3 WHAT IS DISTRIBUTED SYSTEM?

A distributed system is a collection of independent computers that communicate with each other over a network, and work together to accomplish a common task. In a distributed system, each computer node has its own processing power, memory, and storage, and the nodes communicate with each other to exchange data and coordinate their activities.

Distributed systems can be used to build large-scale applications that can handle a high volume of requests and provide high availability and fault tolerance. Examples of distributed systems include cloud computing platforms, peer-to-peer networks, and distributed databases.

One of the key challenges of building a distributed system is coordinating the activities of the individual nodes to ensure that they work together effectively. This can be accomplished through various techniques, such as distributed consensus algorithms, distributed locking, and distributed task scheduling. Additionally, designing a distributed system that is scalable, fault-tolerant, and secure requires careful planning and architecture.

1.4 CHARACTERISTICS OF DISTRIBUTED SYSTEMS

Distributed systems have several key characteristics that differentiate them from traditional centralized systems. Here are some of the most important characteristics:

- Concurrency: In a distributed system, multiple processes or nodes can execute simultaneously and independently of each other, which allows for increased processing power and efficiency.
- Scalability: Distributed systems can scale horizontally by adding more nodes to the network, which can handle increased traffic and load.
- Fault tolerance: Distributed systems can be designed to tolerate failures in individual nodes, ensuring that the system remains operational even if some nodes fail.
- Decentralization: Unlike centralized systems, where a single node or entity controls the system, distributed systems are decentralized, meaning that no single node has complete control over the system.
- Heterogeneity: Distributed systems can be composed of nodes with different hardware, software, and operating systems, allowing for flexibility and interoperability.
- Autonomy: Each node in a distributed system is autonomous, meaning that it can make independent decisions and act on its own behalf.
- Communication: Communication between nodes in a distributed system is typically done through messages or remote procedure calls, which can introduce additional latency and overhead compared to local communication.

Overall, the characteristics of distributed systems enable them to handle large volumes of data, support complex applications, and provide high availability and fault tolerance. However, designing and managing a distributed system can be challenging due to the complexity and potential for communication issues and other sources of failure.

1.5 BENEFITS OF DISTRIBUTED SYSTEMS

Scalability: ability to handle large amounts of data and users

Scalability is a crucial benefit of distributed systems, referring to the ability of a system to handle increasing amounts of work or data without sacrificing performance. There are several ways in which distributed systems provide scalability benefits:

Horizontal scalability: Distributed systems can be scaled horizontally by adding more machines to the system. This approach is also known as "scaling out." By adding more machines to the system, the workload can be distributed across multiple machines, allowing the system to handle more traffic or data.

Vertical scalability: Distributed systems can also be scaled vertically by adding more resources to each machine in the system. This approach is also known as "scaling up." By adding more CPU, memory, or storage to each machine, the system can handle more traffic or data.

Geographic scalability: Distributed systems can be deployed across multiple geographic locations, allowing the system to serve users in different regions. By distributing the workload across different geographic locations, the system can reduce latency and improve performance for users in different parts of the world.

Functional scalability: Distributed systems can be designed to scale horizontally or vertically for specific functions within the system. For example, a distributed database system might scale horizontally by adding more nodes to the cluster, while a distributed caching system might scale vertically by adding more memory to each node.

Fault-tolerance: ability to continue functioning even when parts of the system fail

Fault-tolerance is another key benefit of distributed systems, referring to the ability of a system to continue functioning even in the face of failures or errors. Distributed systems achieve fault-tolerance by using redundant components and data across multiple machines, so that if one machine fails, the system can continue to function without interruption. Here are some ways that distributed systems provide fault-tolerance benefits:

Redundancy: Distributed systems can replicate data across multiple machines, so that if one machine fails, the data can still be accessed from another machine. This approach is also known as "replication" or "mirroring." By replicating data, distributed systems can ensure that data is always available, even if one or more machines fail.

Load balancing: Distributed systems can distribute workloads across multiple machines, so that if one machine fails, the workload can be transferred to another machine. This approach is also known as "load balancing." By load balancing, distributed systems can ensure that workloads are always processed, even if one or more machines fail.

Self-healing: Distributed systems can be designed to detect and respond to failures automatically. For example, if a machine fails, the system can automatically transfer its workload to another machine and replicate its data. This approach is also known as "self-healing." By self-healing, distributed systems can ensure that they continue to function even in the face of failures.

Flexibility: ability to adapt to changing requirements and environments

Flexibility is another key benefit of distributed systems, referring to the ability of a system to adapt to changing requirements or environments. Distributed systems achieve flexibility by using modular components that can be deployed and scaled independently, allowing the system to be easily modified or extended. Here are some ways that distributed systems provide flexibility benefits:

Modularity: Distributed systems can be designed using modular components that can be deployed and scaled independently. This approach is also known as "micro services architecture." By using modular components, distributed systems can be easily modified or extended without impacting the entire system.

Interoperability: Distributed systems can be designed to support different programming languages, platforms, and protocols, allowing them to communicate with a wide variety of other systems. This approach is also known as "interoperability." By supporting interoperability, distributed systems can be integrated with other systems to provide additional functionality or data sources.

Adaptability: Distributed systems can be designed to dynamically adapt to changes in the environment. For example, if the system detects an increase in traffic, it can automatically scale up resources to handle the additional load. This approach is also known as "elasticity." By being adaptable, distributed systems can respond to changing requirements or conditions without requiring manual intervention.

Performance: ability to process requests quickly and efficiently

Performance is another key benefit of distributed systems, referring to the ability of a system to process large amounts of data or traffic quickly and efficiently. Distributed systems achieve high performance by using multiple machines to process workloads in parallel, allowing the system to scale up to handle large volumes of data or traffic. Here are some ways that distributed systems provide performance benefits:

Parallelism: Distributed systems can distribute workloads across multiple machines, allowing them to process workloads in parallel. This approach is also known as "parallelism." By using parallelism, distributed systems can process workloads faster and more efficiently than single machines.

Caching: Distributed systems can use caching to store frequently accessed data in memory, allowing it to be accessed quickly without needing to be retrieved from disk or network. This approach is also known as "caching." By using caching, distributed systems can reduce latency and improve performance.

Data partitioning: Distributed systems can partition data across multiple machines, allowing each machine to process a subset of the data. This approach is also known as "data partitioning." By using data partitioning, distributed systems can process large volumes of data quickly and efficiently.

Overall, performance is critical for ensuring that distributed systems can handle large volumes of data or traffic quickly and efficiently. By using parallelism, caching, and data partitioning techniques, distributed systems can provide a high-performance architecture that can scale up to meet the needs of modern applications.

Cost-effectiveness: ability to utilize resources more efficiently

Cost-effectiveness is another key benefit of distributed systems, referring to the ability of a system to provide high performance and fault-tolerance at a lower cost than traditional monolithic systems. Distributed systems achieve cost-effectiveness by using commodity hardware, which is less expensive than specialized hardware, and by allowing organizations to pay only for the resources they need, rather than investing in large, fixed infrastructure. Here are some ways that distributed systems provide cost-effectiveness benefits:

Commodity hardware: Distributed systems can use commodity hardware, such as off-the-shelf servers, instead of expensive specialized hardware. This approach is also known as "commodity hardware." By using commodity hardware, distributed systems can reduce the cost of infrastructure.

Pay-as-you-go pricing: Distributed systems can use pay-as-you-go pricing models, allowing organizations to pay only for the resources they need. This approach is also known as "pay-as-you-go" pricing. By using pay-as-you-go pricing, organizations can avoid investing in large, fixed infrastructure and can instead pay only for the resources they actually use.

Efficient resource utilization: Distributed systems can use resource allocation algorithms, such as load balancing and resource pooling, to efficiently allocate resources across multiple machines. This approach is also known as "efficient resource utilization." By using efficient resource utilization, distributed systems can minimize waste and optimize resource usage, reducing overall costs.

CHECK YOUR PROGRESS-I

1. State True or False:

a) In a distributed system, all nodes must share the same hardware and operating system.

b) Distributed systems can continue to function even if some nodes fail.

c) Distributed systems cannot be deployed across multiple geographic locations.

d) Fault tolerance in distributed systems is achieved through redundancy and load balancing.

e) Performance in distributed systems is improved through parallelism and caching

2. Fill in the Blanks:

a)In distributed systems, __________ allows multiple nodes to execute simultaneously and independently.

b)__________ scalability involves adding more machines to handle increased load in a distributed system.

c)__________ refers to the ability of a system to continue functioning even when parts of it fail.

d)The ability to serve users in different regions by deploying distributed systems across multiple locations is known as __________ scalability.

e)The approach of only paying for the resources used in distributed systems is known as __________ pricing.

1.6 DIFFERENCES BETWEEN CENTRALIZED AND DISTRIBUTED SYSTEMS

Centralized and distributed systems are two fundamentally different approaches to designing computing systems. Here are some of the key differences between centralized and distributed systems:

Control: In a centralized system, all control and decision-making authority is concentrated in a single central unit, whereas in a distributed system, control is distributed across multiple nodes that communicate and coordinate with each other.

Scalability: Centralized systems have limited scalability, as they are dependent on the capacity and processing power of the central unit, whereas distributed systems can be scaled up or down by adding or removing nodes.

Fault tolerance: Centralized systems are vulnerable to failures in the central unit, which can cause the entire system to fail, whereas distributed systems are designed to be resilient to failures in individual nodes or components.

Communication: Communication between nodes in a distributed system is typically done through messages or remote procedure calls, which can introduce additional latency and overhead compared to local communication in a centralized system.

Heterogeneity: Distributed systems can be composed of nodes with different hardware, software, and operating systems, whereas centralized systems typically have a uniform hardware and software architecture.

Complexity: Distributed systems are generally more complex to design and manage than centralized systems, due to the need to coordinate and manage communication between multiple nodes.

1.7 CHALLENGES IN DISTRIBUTED SYSTEMS

Distributed systems can present numerous challenges, including:

- Network failures
- Security
- Scalability

- Complexity
- Consistency
- Fault tolerance
- Interoperability

Network Failures:

Network failures are one of the most common and challenging issues in distributed systems. A distributed system is a collection of autonomous computers that work together to achieve a common goal. These systems rely on the network to communicate and share data between different nodes in the system. When the network fails, it can cause a range of problems that can impact the performance, availability, and consistency of the system.

Here are some ways network failures can pose challenges to distributed systems:

Communication failure: Distributed systems rely on communication between nodes to function. When a network failure occurs, nodes may not be able to communicate with each other, causing communication failure. This can result in data loss, inconsistent data, or even system crashes.

Increased latency: Network failures can cause delays in communication between nodes. Increased latency can slow down the system and cause delays in processing tasks, resulting in reduced system performance.

Partitioning: Network failures can result in the system being partitioned into separate segments, each of which is isolated from the others. This can cause issues such as data inconsistency, duplicate data, and lost data, which can lead to data corruption.

Reconfiguration: Network failures can also cause the system to reconfigure itself to adapt to the new network conditions. This can be a time-consuming process that can impact system performance and availability.

To overcome these challenges, distributed systems must be designed to be resilient to network failures. This can be achieved through the use of redundancy, fault tolerance, and load balancing mechanisms. These mechanisms can help ensure that the system continues to function even in the presence of network failures. Additionally, distributed systems should be designed to detect network failures and take appropriate actions to mitigate their impact.

Security:

Security is a critical challenge in distributed systems as they are often exposed to various types of attacks. A distributed system is a collection of interconnected nodes, and an attacker who gains control of one or more nodes can potentially compromise the entire system. In addition, distributed systems are often deployed across different geographic locations and networks, making them more vulnerable to attacks.

Here are some ways security can pose challenges to distributed systems:

Authentication and Authorization: Distributed systems must ensure that only authorized users can access the system and its resources. Authentication and authorization mechanisms must be implemented to verify the identity of users and enforce access control policies.

Confidentiality and Integrity: Distributed systems must ensure that data exchanged between nodes is kept confidential and not modified in transit. Encryption and digital signatures can be used to ensure data confidentiality and integrity.

Denial of Service (DoS) attacks: Distributed systems are vulnerable to DoS attacks that can overwhelm the system and make it unavailable to legitimate users. Distributed DoS attacks are particularly challenging as they involve multiple nodes, making it difficult to detect and mitigate the attack.

Malware and viruses: Distributed systems are vulnerable to malware and viruses that can infect one or more nodes and spread to other nodes. Such attacks can cause data loss, system downtime, and even complete system failure.

To address these security challenges, distributed systems must be designed with security in mind. This can be achieved through the use of secure communication protocols, strong authentication and authorization mechanisms, encryption, and digital signatures. In addition, distributed systems must be continuously monitored and updated to detect and mitigate security threats. Finally, education and training for system users can also help reduce the risk of security breaches caused by human error or social engineering attacks.

Scalability:

Scalability is a significant challenge in distributed systems as they are designed to handle a large number of users, transactions, and data. Scalability is the ability of the system to handle an increasing workload without sacrificing performance or stability. As the number of users or data grows, the system must be able to handle the load without becoming overloaded or crashing.

Here are some ways scalability can pose challenges to distributed systems:

Network bandwidth: Distributed systems must be designed to handle large amounts of data transferred over the network. As the number of users or data grows, the network bandwidth required to handle the load can become a bottleneck. This can cause delays, slow down the system, and impact system performance.

Load balancing: Distributed systems must be designed to distribute the workload evenly across nodes in the system. Load balancing mechanisms can be used to ensure that each node handles an equal share of the workload. As the number of users or data grows, load balancing can become a challenge, and the system may require additional resources to handle the load.

Data storage: Distributed systems must be designed to handle large amounts of data. As the amount of data grows, the system must be able to store and retrieve data efficiently. This can be challenging, and the system may require additional storage resources or more efficient data storage mechanisms.

System complexity: Distributed systems are often complex, with multiple nodes and components working together to achieve a common goal. As the system grows, the complexity of the system can increase, making it more challenging to manage and maintain.

To address these scalability challenges, distributed systems must be designed to scale horizontally and vertically. Horizontal scalability involves adding more nodes to the system to handle the increased workload, while vertical scalability involves adding more resources to existing nodes. Additionally, distributed systems must be designed to use resources efficiently, with load balancing mechanisms to distribute the workload evenly across nodes. Finally, distributed systems must be designed to be modular and scalable, with each component designed to work independently and communicate with other components in the system.

Complexity:

Complexity is a significant challenge in distributed systems due to the large number of components, nodes, and interactions involved in the system. Distributed systems are made up of multiple independent components that work together to achieve a common goal. Each component may have different requirements, interfaces, and communication protocols, making the system complex and challenging to design, test, and maintain.

Here are some ways complexity can pose challenges to distributed systems:

Interoperability: Distributed systems must be designed to work with other systems, including legacy systems, third-party components, and other distributed systems. Interoperability can be challenging, as different systems may have different interfaces, protocols, and data formats.

Integration: Distributed systems must be designed to integrate multiple components into a cohesive system. Integration can be challenging, as each component may have different requirements and interfaces, and the system must be able to handle failures and errors in individual components.

Testing: Distributed systems must be tested to ensure that each component works correctly and that the system as a whole performs as expected. Testing distributed systems can be challenging, as it involves testing interactions between components and nodes across different networks and geographic locations.

Maintenance: Distributed systems require ongoing maintenance to ensure that they remain secure, reliable, and performant. Maintenance can be challenging, as it involves updating components, managing configurations, and resolving issues that arise from the interaction between components.

To address these complexity challenges, distributed systems must be designed to be modular and loosely coupled, with each component designed to work independently and communicate with other components in the system. Standardized interfaces, protocols, and data formats can help facilitate interoperability and integration. Additionally, distributed systems should be designed to be testable, with automated testing mechanisms in place to test interactions between components and nodes. Finally, distributed systems should be designed to be maintainable, with tools and processes in place to manage configurations, update components, and resolve issues.

Consistency:

Consistency is a significant challenge in distributed systems due to the difficulty of ensuring that all nodes in the system have the same view of the data at any given time. Distributed systems are designed to handle large amounts of data across multiple nodes, and maintaining consistency across all nodes can be challenging.

Here are some ways consistency can pose challenges to distributed systems:

Data replication: Distributed systems often replicate data across multiple nodes for redundancy and improved performance. However, ensuring consistency across all nodes can be challenging, as each node may have a different view of the data at any given time.

Network delays: Distributed systems rely on network communication to maintain consistency between nodes. Network delays can cause inconsistencies, as nodes may receive updates at different times or in different orders.

Partitioning: Distributed systems must be designed to handle network partitions, where some nodes in the system are unable to communicate with each other. Partitioning can cause inconsistencies, as each partition may have a different view of the data.

Conflict resolution: Distributed systems must be designed to handle conflicts that arise when multiple nodes update the same data simultaneously. Conflict resolution can be challenging, as different nodes may have different views of the data and may have conflicting updates.

To address these consistency challenges, distributed systems must be designed to ensure that all nodes have the same view of the data at any given time. Techniques such as distributed consensus algorithms, two-phase commit, and vector clocks can be used to ensure consistency across nodes. Additionally, distributed systems should be designed to handle network delays and partitions, with mechanisms in place to detect and recover from failures. Finally, conflict resolution mechanisms should be in place to handle conflicting updates and ensure that the data remains consistent across all nodes.

Fault tolerance:

Fault tolerance is a significant challenge in distributed systems due to the high likelihood of component failures and network disruptions. Distributed systems rely on multiple independent components working together, and any failure in any of these components can cause the entire system to fail.

Here are some ways that fault tolerance can pose challenges to distributed systems:

Component failures: Distributed systems rely on multiple components, including nodes, databases, and network infrastructure. Any failure in any of these components can cause the entire system to fail.

Network disruptions: Distributed systems rely on network communication to exchange data and updates between nodes. Network disruptions, such as network outages, can cause nodes to become isolated from each other, leading to system failures.

Cascading failures: Faults in one component can cause failures in other components, leading to a cascading failure that affects the entire system.

Recovery time: Distributed systems must be able to recover from failures quickly to minimize downtime and maintain system availability. Recovery time can be challenging, as it may require complex coordination between components and may involve data loss or inconsistencies.

To address these fault tolerance challenges, distributed systems must be designed to be fault-tolerant, with mechanisms in place to detect and recover from failures. Redundancy and replication can be used to ensure that components and data are available even in the event of failures. Additionally, distributed systems should be designed to handle network disruptions, with mechanisms in place to detect and recover from network failures. Finally, recovery mechanisms should be in place to ensure that the system can recover from failures quickly and with minimal data loss or inconsistencies.

Interoperability:

Interoperability is a significant challenge in distributed systems due to the need for different components to communicate and work together seamlessly. Distributed systems typically consist of multiple independent components that need to interact with each other to achieve the system's overall objectives.

Here are some ways interoperability can pose challenges to distributed systems:

Heterogeneity: Distributed systems often consist of different components that use different hardware, software, and communication protocols. Ensuring that these components can work together seamlessly can be challenging, as each component may have its own requirements and interfaces.

Legacy systems: Distributed systems may need to integrate with legacy systems, which may have outdated hardware, software, and communication protocols. Integrating with legacy systems can be challenging, as these systems may not be designed with modern interoperability standards in mind.

Standardization: Distributed systems need to adhere to interoperability standards to ensure that different components can communicate with each other. However, there may be different standards for different components, and ensuring that all components adhere to the same standards can be challenging.

Scalability: Interoperability challenges can become more significant as the size and complexity of the distributed system increase. As the number of components and interactions between them grows, ensuring interoperability becomes increasingly challenging.

To address these interoperability challenges, distributed systems must be designed with interoperability in mind. Standardization of interfaces, protocols, and data formats can help ensure that different components can communicate with each other. Additionally, compatibility testing and certification programs can be put in place to ensure that components from different vendors work together seamlessly. Finally, distributed systems should be designed to be scalable, with mechanisms in place to handle the growing complexity and heterogeneity of the system as it expands.

1.8 TRANSPARENCY IN DISTRIBUTED SYSTEMS

Transparency is an essential characteristic of distributed systems that refers to the ability of the system to hide its distributed nature from users and applications, providing the illusion of a single, coherent system. Transparency issues arise when the distributed nature of the system becomes visible to users or applications, causing confusion or inconsistencies.

Here are some transparency issues that can arise in distributed systems:

Location transparency: Location transparency refers to the ability of users and applications to access resources without needing to know their physical location. If a distributed system is not location-transparent, users and applications may need to know the location of resources, leading to confusion and inconsistencies.

Access transparency: Access transparency refers to the ability of users and applications to access resources without needing to know how they are implemented or managed. If a distributed system is not access-transparent, users and applications may need to know the implementation details of resources, leading to confusion and inconsistencies.

Failure transparency: Failure transparency refers to the ability of users and applications to continue using the system in the event of failures or disruptions. If a distributed system is not failure-transparent, users and applications may need to be aware of failures or disruptions, leading to confusion and inconsistencies.

Performance transparency: Performance transparency refers to the ability of users and applications to access resources without needing to know their performance characteristics. If a distributed system is not performance-transparent, users and applications may need to know the performance characteristics of resources, leading to confusion and inconsistencies.

To address transparency issues, distributed systems must be designed to be transparent, with mechanisms in place to hide the distributed nature of the system from users and applications. Techniques such as virtualization, load balancing, and failover can be used to ensure that resources are location-transparent, access-transparent, and failure-transparent. Additionally, distributed systems should be designed to be scalable, with mechanisms in place to handle performance transparency issues as the system grows in complexity and size.

1.9 INHERENT LIMITATIONS OF DISTRIBUTED SYSTEMS

Distributed systems offer many benefits, but they also face several inherent limitations that pose challenges to their design and implementation:

- **Network Latency:** It refers to the delay in data transmission across the network. This can significantly affect performance, especially in applications requiring real-time data processing. Latency can lead to slower response times and decreased user satisfaction.
- **Partial Failures**:Unlike centralized systems, components in distributed systems can fail independently. And thus making failure detection and recovery more complex. Ensuring that the system continues to operate correctly despite partial failures requires robust fault tolerance mechanisms.
- **Concurrency and Synchronization**: It refers multiple processes may access shared resources simultaneously. But can lead to data inconsistency or race conditions if not managed properly. Synchronization mechanisms like locks or consensus algorithms are necessary but can introduce complexity and performance bottlenecks.
- **Data Consistency**: It refers maintaining a uniform view of data across distributed nodes is challenging.Inconsistencies can arise due to network delays, node failures, or concurrent updates.

Achieving strong consistency may require trade-offs with system availability and partition tolerance (as per the CAP theorem).

- **Complexity**:Distributed systems inherently involve multiple components, each potentially running on different hardware and software platforms.And thus Increases the complexity of system design, implementation, and maintenance. Debugging and troubleshooting distributed systems can be particularly challenging.
- **Scalability Challenges**:While distributed systems are designed to scale, there are limits to scalability due to bottlenecks in communication, resource contention, or centralized components. But beyond a certain point, adding more resources may not lead to proportional improvements in performance, requiring careful architectural planning.
- **Security Issues**:Ensuring security in a distributed system is more challenging due to multiple points of potential vulnerability.Threats such as data interception, unauthorized access, and distributed denial-of-service (DDoS) attacks must be mitigated with comprehensive security measures, including encryption, authentication, and access control.
- **Lack of a Global Clock**:There is no single, authoritative time source in a distributed system. This makes event ordering difficult, affecting the ability to maintain consistency and synchronization. Techniques like logical clocks or vector clocks are used, but they add complexity.
- **Difficulty in Guaranteeing QoS**:Quality of Service (QoS) guarantees can be hard to maintain across diverse network conditions and varying workloads.Service reliability and

performance can fluctuate, impacting user experience. Adaptive resource management and monitoring are essential.

Understanding these limitations is crucial for designing robust and efficient distributed systems. By anticipating these challenges, architects can implement strategies to mitigate their impact, ensuring that distributed systems meet their performance, reliability, and scalability requirements.

CHECK YOUR PROGRESS-II

3. State True or False:

a) In a centralized system, all control is concentrated in a single unit.

b) Distributed systems are less fault-tolerant than centralized systems.

c) Network latency can significantly impact the performance of distributed systems.

d) Distributed systems are easier to design and manage compared to centralized systems.

e) Scalability is not a challenge in distributed systems.

4. Fill in the Blanks:

a) In distributed systems, __________ mechanisms are necessary to prevent data inconsistency due to concurrent updates.

b) __________ transparency refers to the system's ability to hide failures from users.

c) A significant challenge in distributed systems is maintaining data __________ across all nodes.

d) __________ and __________ are essential for ensuring security in distributed systems.

e) The lack of a __________ clock makes event ordering difficult in distributed systems.

1.10 SUMMING UP

- A distributed system consists of independent computers connected via a network, collaborating to achieve common tasks with each node possessing its processing power, memory, and storage.
- The Characteristics of Distributed Systems are:
    - Concurrency: It allows multiple nodes to execute independently, enhancing processing efficiency.
    - Scalability: Scaling horizontally (adding nodes) or vertically (increasing resources) to handle increased workload.
    - Fault Tolerance: Distributed Systems are designed to tolerate failures in nodes, ensuring system continuity.
    - Decentralization: There is No single controlling node; control and data are distributed.
    - Heterogeneity: Here, nodes can differ in hardware, software, and OS, promoting flexibility.
    - Autonomy: Each node can operate independently, making decisions locally.
    - Communication: Nodes communicate via messages or remote procedure calls, with added latency compared to local systems.
- The Benefits of Distributed Systems are:
    - Scalability: Handles large data volumes and user traffic effectively.
    - Fault Tolerance: Continues functioning despite component failures.
    - Flexibility: Adapts to changing requirements and environments.
    - Performance: Processes requests quickly through parallelism, caching, and data partitioning.
    - Cost-effectiveness: Uses commodity hardware and pay-as-you-go pricing, optimizing resource utilization and reducing costs.

- Transparency in Distributed Systems mean:
    - Location Transparency: Users access resources without knowing physical locations.
    - Access Transparency: Users access resources without knowing implementation details.
    - Failure Transparency: Continues operation during failures without user awareness.
    - Performance Transparency: Users access resources without knowing performance characteristics.

1.11 ANSWERS TO CHECK YOUR PROGRESS

1. a) False b) True c) False d) True e) True

2. a) concurrency b) Horizontal c) Fault tolerance
d) geographic e) pay-as-you-go

3. a) True b) False c) True d) False e) False

4. a) synchronization b) Failure c) consistency
d) Authentication, encryption e) global

1.12 POSSIBLE QUESTIONS

Short Answer Type Questions:

1. What is a distributed system?
2. Give two examples of distributed systems.
3. What is concurrency in a distributed system?
4. What is meant by the term 'horizontal scalability'?
5. Define 'fault tolerance' in the context of distributed systems.
6. What is the role of redundancy in a distributed system?
7. Explain the term 'decentralization' in distributed systems.
8. What is the significance of heterogeneity in distributed systems?
9. How does a distributed system achieve fault tolerance through load balancing?
10. What is the benefit of modularity in distributed systems?

11. Explain the concept of network latency in distributed systems.
12. What does location transparency mean in distributed systems?
13. List two challenges in maintaining consistency in distributed systems.
14. Why is security a significant concern in distributed systems?
15. What are partial failures in distributed systems?
16. What is meant by the lack of a global clock in distributed systems?

Long Answer Type Questions:

17. Explain the concept of a distributed system and discuss its primary characteristics.
18. Discuss the various ways in which distributed systems can scale, and explain how each method contributes to the system's performance and efficiency.
19. How do distributed systems ensure fault tolerance? Provide examples of techniques used to achieve fault tolerance.
20. What are the benefits of using distributed systems in terms of flexibility and adaptability? Provide detailed examples.
21. Describe the performance benefits of distributed systems. How do parallelism, caching, and data partitioning contribute to these benefits?
22. Discuss the cost-effectiveness of distributed systems. How do commodity hardware and pay-as-you-go pricing models reduce overall costs?
23. What are some of the challenges involved in designing and managing distributed systems? How can these challenges be addressed?
24. Explain the concept of interoperability in distributed systems and its importance in modern applications.
25. How do distributed systems utilize efficient resource allocation to enhance cost-effectiveness? Provide detailed examples of resource allocation techniques.
26. What are the key differences between centralized and decentralized systems, and what advantages do decentralized systems offer over centralized ones?

27. Compare and contrast centralized and distributed systems, highlighting their main differences.
28. Describe the inherent limitations of distributed systems and discuss how they impact system design and implementation.
29. Explain the various types of transparency in distributed systems and why they are important for system design.
30. What are the main challenges in achieving fault tolerance in distributed systems? Provide examples of mechanisms used to address these challenges.
31. Discuss the issues of network failures in distributed systems and how they can be mitigated.
32. Explain the complexity challenges in distributed systems and suggest design principles to manage this complexity.
33. What is data consistency in distributed systems, and why is it challenging to achieve? Discuss the CAP theorem in this context.
34. Describe the challenges and strategies for ensuring scalability in distributed systems.
35. Analyze the security issues specific to distributed systems and propose measures to enhance security.
36. How do concurrency and synchronization issues arise in distributed systems, and what are the common strategies to address them?
37. Discuss the difficulties in guaranteeing Quality of Service (QoS) in distributed systems and suggest ways to manage them.

1.13REFERENCES AND SUGGESTED READINGS

1. "Distributed Systems: Concepts and Design" by George Coulouris
2. “Designing Data-Intensive Applications" by Martin Kleppmann
3. “Distributed Systems: Principles and Paradigms" by Andrew Tanenbaum and Maarten Van Steen

×××

UNIT: 2
HARDWARE CONCEPTS AND SYSTEM MODELS

Unit Structure:

2.1 INTRODUCTION

In modern computing, distributed systems and multiprocessor architectures have become integral to achieving high performance, scalability, and reliability. This unit discusses the fundamental concepts of multiprocessors, exploring both homogeneous and heterogeneous systems, and examines the role of middleware in distributed systems. Furthermore, we will explore fundamental system models that underpin the design and analysis of distributed systems, alongside architectural system models that provide blueprints for building and organizing distributed computing environments.

2.2 UNIT OBJECTIVES

After going through this unit, you will be able to:

- define and differentiate between homogeneous and heterogeneous multiprocessor system.
- describe the concept of middleware and its role in facilitating communication and resource management in distributed systems.
- identify different types of middleware and their specific applications in distributed environments.
- explore fundamental System Models.

2.3 WHAT IS MULTIPROCESSOR?

Multiprocessors, also known as parallel processing systems, are computer systems that use multiple processors or central processing units (CPUs) to perform tasks simultaneously. Multiprocessor systems are designed to improve performance by increasing processing power and reducing the time needed to complete a task.

There are two main types of multiprocessor systems: symmetric multiprocessing (SMP) and asymmetric multiprocessing (AMP).

Symmetric Multiprocessing (SMP):

In SMP systems, all processors have equal access to the system's resources and are considered to be homogeneous, meaning they have the same instruction sets and architectures. The processors communicate with each other through a shared memory, allowing them to work together to complete a task. SMP systems are designed for use in applications that require high processing power, such as scientific simulations and data processing. The advantages of SMP systems are:

- SMP systems can execute multiple processes simultaneously, which improves the system's processing power and speed.

- SMP systems can be cost-effective compared to single-processor systems because they use multiple processors on a single system.
- SMP systems can be easily scaled by adding more processors to the system, allowing for an increase in processing power.

Asymmetric Multiprocessing (AMP):

In AMP systems, different processors have different instruction sets and architectures, and each processor is assigned a specific task. These processors may communicate with each other through a network, rather than a shared memory, and are considered to be heterogeneous. AMP systems are designed for use in applications that require specialized processing power, such as graphics processing or artificial intelligence. The advantages of AMP systems are:

- AMP systems can be designed to perform specific tasks, such as graphics processing or artificial intelligence.
- AMP systems can be easily scaled by adding more processors to the system, allowing for an increase in processing power.

There are issues in multiprocessor systems, and they are:

Load balancing: Ensuring that tasks are distributed evenly among processors to optimize system performance.

Data consistency: Ensuring that shared data is consistent and accurate across all processors.

Communication overhead: The additional time and resources required for processors to communicate with each other in a multiprocessor system.

Multiprocessors can be further classified into two types: tightly-coupled and loosely-coupled systems. Tightly-coupled systems are characterized by having processors that share a common memory and I/O system, whereas loosely-coupled systems have processors that communicate with each other over a network.

2.4 TIGHTLY-COUPLED MULTIPROCESSOR SYSTEMS

Tightly-coupled multiprocessor systems are computer systems that use multiple processors or central processing units (CPUs) that share a common memory and I/O system. The processors are tightly-coupled because they are physically close to each other and communicate with each other through a shared bus, making it easier and faster for them to share data and resources. Tightly-coupled multiprocessor systems are commonly used in applications that require high performance and reliability, such as scientific simulations, real-time systems, and high-end servers.

Tightly-coupled multiprocessor systems offer several advantages over single-processor systems which include:

Increased Processing Power: Tightly-coupled multiprocessor systems can perform multiple tasks simultaneously, which improves the system's processing power and speed.

Improved Reliability: Tightly-coupled multiprocessor systems can continue to function even if one or more processors fail, making them more reliable than single-processor systems.

Scalability: Tightly-coupled multiprocessor systems can be easily scaled by adding more processors to the system, allowing for an increase in processing power.

Tightly-coupled multiprocessor systems also have some limitations and challenges and they are:

Increased Complexity: Tightly-coupled multiprocessor systems are more complex than single-processor systems, which can make them more difficult to design, implement, and maintain.

Memory Contention: Tightly-coupled multiprocessor systems may experience memory contention, where multiple processors try to access the same memory location simultaneously, leading to conflicts and delays.

Communication Overhead: The additional time and resources required for processors to communicate with each other in a tightly-coupled multiprocessor system can create communication overhead, which can reduce system performance.

2.5 LOOSELY-COUPLED MULTIPROCESSOR SYSTEMS

Loosely-coupled multiprocessor systems, also known as distributed systems, consist of a collection of independent computers interconnected by a network. Each computer in the system has its own memory and operates independently, communicating with other computers in the system through message passing.

In a loosely-coupled multiprocessor system, the communication between processors is slower than in tightly-coupled systems, due to the presence of the network. The network can introduce latency, bandwidth limitations, and other communication overheads that can impact the performance of the system.

Loosely-coupled systems are commonly used in distributed computing environments, such as cloud computing and grid computing, where a large number of independent computers are connected to provide a shared computing resource. In such environments, the use of message-passing interfaces and other distributed computing frameworks can help to manage the complexity of the system and minimize the impact of communication overheads.

Loosely-coupled multiprocessor systems offer several advantages over single-processor systems which include:

Flexibility: Loosely-coupled systems are highly flexible as nodes can be added or removed from the system without affecting the system's overall operation. This feature makes them highly scalable and ideal for use in dynamic computing environments where the size of the system may change over time.

Cost-Effective: Loosely-coupled systems are cost-effective, as each node can be a commodity computer, rather than a specialized processor. This approach makes it more affordable to build large-scale systems, which can offer significant computational power.

Fault Tolerance: Loosely-coupled systems are highly fault-tolerant, as the failure of one node does not necessarily impact the operation of the entire system. This feature is critical in systems where high reliability is necessary.

Loosely-coupled multiprocessor systems also have some limitations and challenges and they are:

Communication Overhead: Communication between nodes in a loosely-coupled system is slower than in a tightly-coupled system due to the presence of a network. This limitation can significantly impact the overall performance of the system.

Software Complexity: Designing software for loosely-coupled systems is more complicated than for tightly-coupled systems. It requires a more distributed computing framework, which can increase the complexity of the system.

Security: Loosely-coupled systems can be more vulnerable to security threats, as the network provides an attack surface for hackers. This challenge can be addressed by using security measures such as firewalls and encryption.

2.6 HOMOGENOUS MULTIPROCESSOR SYSTEMS

Homogenous systems are a type of multiprocessor system where all the processors are identical in terms of hardware and software. In other words, all processors in a homogeneous system have the same architecture and instruction set, and they operate under the same operating system.

In a homogenous system, processors share the same memory, and they communicate through shared memory or inter-processor communication mechanisms. Typically, the memory is organized in a way that each processor has its own local memory, and a shared memory space accessible to all processors.

One of the main advantages of homogeneous systems is that they are easier to program since all processors are identical. The programming model is relatively simple, and parallelism can be achieved by dividing a single task into smaller sub-tasks that can be executed simultaneously on different processors.

Homogeneous systems offer several benefits which include:

Improved Performance: Homogeneous systems can provide significant performance improvements over single-processor systems. By distributing tasks among multiple processors, the system can execute tasks in parallel, reducing the overall processing time.

Scalability: Homogeneous systems are highly scalable, as new processors can be added to the system as the workload increases.

This scalability allows the system to grow as demand increases, providing increased performance and capacity.

Fault Tolerance: Homogeneous systems are highly fault-tolerant, as they can continue to operate even if one or more processors fail. This is because tasks can be re-allocated to other processors, ensuring that the system continues to function.

2.7 HETEROGENOUS MULTIPROCESSOR SYSTEMS

Heterogeneous systems are a type of multiprocessor system where the processors are not identical in terms of hardware and software. In other words, processors in a heterogeneous system can have different architectures, instruction sets, and operating systems.

In a heterogeneous system, processors may communicate through shared memory, but they may also use other mechanisms such as message passing or remote procedure calls (RPCs). Each processor may have its own memory, and a shared memory space may be used to facilitate communication and data sharing among processors.

One of the main advantages of heterogeneous systems is that they can be optimized for specific workloads. Different processors can be chosen based on their strengths, such as high-performance graphics processing or specialized hardware for machine learning. This allows for the system to achieve better overall performance than a homogeneous system.

Heterogeneous systems offer several benefits which include:

Improved Performance: Heterogeneous systems can provide significant performance improvements over homogeneous systems since processors can be chosen for specific workloads. This allows for better utilization of the available hardware and can result in faster processing times.

Flexibility: Heterogeneous systems can be designed to meet specific requirements, such as high-performance computing or machine learning. This flexibility allows for better customization of the system and can lead to better performance and cost-effectiveness.

Energy Efficiency: Heterogeneous systems can be designed to be more energy-efficient than homogeneous systems. By using processors optimized for specific tasks, the system can consume less energy and reduce overall operating costs.

However, heterogeneous systems also have some limitations:

Complexity: Heterogeneous systems can be more complex to program and manage than homogeneous systems. The programming model must be carefully designed to ensure that tasks are distributed appropriately among processors, and that communication is efficient and reliable.

Hardware Compatibility: Heterogeneous systems can require specialized hardware, which can be expensive and difficult to acquire. Additionally, different processors may have different memory architectures or other requirements, which can complicate system design.

Interoperability: Heterogeneous systems may require specialized software to facilitate communication and data sharing among processors. Ensuring that different components of the system can work together seamlessly can be a challenge.

2.8 WHAT IS MIDDLEWARE?

Middleware is a layer of software that provides a bridge between different applications or software components in a distributed computing environment. It acts as a communication layer that enables applications to exchange data and communicate with each other, regardless of the programming languages, operating systems, or hardware platforms they are running on.

Middleware provides a set of standardized services and protocols that enable applications to communicate with each other and share data. These services can include message queuing, remote procedure calls (RPCs), transaction processing, object request brokers (ORBs), and web services.

One of the primary benefits of middleware is that it allows different applications to work together seamlessly without requiring them to be tightly coupled. Middleware enables applications to communicate with each other using standard protocols and interfaces, which simplifies the development and integration process.

2.8.1Types of Middleware

There are several types of middleware, including:

Message-oriented middleware (MOM): This type of middleware enables applications to communicate by exchanging messages. MOM provides a reliable messaging service that ensures messages are delivered in the correct order and without loss or duplication.

Object middleware: Object middleware provides an object-oriented approach to middleware. It enables objects to communicate with each other by passing messages using standard protocols such as CORBA or Java RMI.

Transaction middleware: Transaction middleware provides a set of services that enable multiple applications to participate in distributed transactions. These services ensure that transactions are executed in a reliable and consistent manner.

Web middleware: Web middleware provides a set of services that enable web applications to communicate with each other. These services include web services, XML, and REST.

2.8.2 Benefits and Limitations of Middleware

Middleware has several benefits which include:

Interoperability: Middleware enables applications to communicate with each other regardless of the programming languages, operating systems, or hardware platforms they are running on.

Scalability: Middleware can help to distribute the workload across multiple servers or nodes in a distributed computing environment, which can improve performance and scalability.

Flexibility: Middleware can help to simplify the development and integration process by providing a set of standardized services and protocols that enable applications to communicate with each other.

However, middleware can also have some limitations including:

Complexity: Middleware can be complex to design and implement, and can require specialized knowledge and skills.

Performance: Middleware can introduce additional overhead and latency, which can affect performance and throughput.

Cost: Middleware can be expensive to license and deploy, especially if specialized hardware or software is required.

2.9 SYSTEM MODELS IN DISTRIBUTED SYSTEMS

System models in distributed systems are crucial for understanding, designing, and analyzing the behavior of distributed applications. They help ensure that the systems are robust, reliable, scalable, secure, and performant, while also simplifying the inherent complexity of distributed computing.

Why System Models?

System models in distributed systems are important for several reasons:

- System models provide a structured framework that guides the design and development of distributed systems.
- System models help in analyzing the behavior and performance of distributed systems.
- System models aid in designing systems that can handle different types of failures gracefully..
- System models provide the foundation for implementing robust security measures.
- Standardized models enable different components and systems to work together seamlessly etc.

2.10 TYPES OF SYSTEM MODELS

There are three types of System Models in Distributed Systems. They are:

- Physical Model
- Architectural Model
- Fundamental Model

2.10.1 Physical Model

A physical model represents the hardware components of a distributed system. It shows how computers and devices are connected and helps in designing, managing, and improving the system's performance.

A physical model includes the following key parts:

Nodes:

- Nodes are devices that process data, run tasks, and communicate with each other. They can be user computers, servers, workstations, etc.
- Nodes provide an interface for users to interact with other backend devices for tasks like storage, processing, and web browsing.
- Each node has an operating system, execution environment, and middleware that enable communication and other essential tasks.

Links:

- Links are the communication channels between nodes and devices. They can be wired (using copper wires, fiber optic cables) or wireless.
- The choice of link depends on the environment and requirements. High-performance and real-time computing often need physical links.
- Types of connections include:
    - **Point-to-point links**: Connect two nodes directly.
    - **Broadcast links**: Allow one node to send data to multiple nodes at once.
    - **Multi-access links**: Multiple nodes share the same channel and need protocols to prevent interference.

Middleware:

- Middleware is software running on nodes, providing decentralized control and decision-making.
- It handles communication, resource management, fault tolerance, synchronization, and security.

Network Topology:

- This describes how nodes and links are arranged. Common topologies are bus, star, mesh, ring, and hybrid.

- The choice of topology depends on the specific needs and requirements.

Communication Protocols:

- Protocols are rules for transmitting data through links. Examples include TCP, UDP, HTTPS, and MQTT.
- They ensure nodes can communicate and understand each other.

2.10.2Architectural Model

An architectural model in a distributed computing system is the overall design and structure of the system. It shows how different components are organized to interact and provide the desired functions. This model gives an overview of development, deployment, and operations, ensuring efficient cost usage and improved scalability. There are different aspects of an architectural model. These are as follows:

Client-Server Model:

- A centralized approach where clients request services and servers provide them.
- Works on a request-response basis: the client sends a request, the server processes it, and then responds.
- Uses protocols like TCP/IP and HTTP.
- Commonly used in web services, cloud computing, and database management systems.

Peer-to-Peer Model:

- A decentralized approach where all nodes (peers) have equal capabilities and can request and provide services.
- Highly scalable as peers can join and leave dynamically, creating an ad-hoc network.
- Resources are distributed, and peers find them as needed.
- Direct communication between peers without intermediaries, following set rules.

Layered Model:

- Organizes the system into multiple layers, each providing a specific service.
- Each layer communicates with adjacent layers using defined protocols without affecting the overall system.
- Creates a hierarchical structure where each layer hides the complexity of the lower layers.

Microservices Model:

- Breaks a complex application into multiple independent services, each running on different servers.
- Each service performs a single function focused on a specific business capability.
- Makes the system more maintainable, scalable, and easier to understand.
- Services can be independently developed, deployed, and scaled without affecting other services.

2.10.3 Fundamental Model

A fundamental model in a distributed computing system is a basic framework that helps understand the key aspects of these systems. These models describe common properties in all architectural models and are essential to understand how a distributed system behaves. There are three fundamental models:

Interaction Model:

Distributed systems involve many processes interacting in complex ways. The interaction model helps us understand how these processes communicate and coordinate.

- Message passing sending messages that contain data, instructions, service requests, or process synchronization between computing nodes. This can be synchronous (immediate) or asynchronous (delayed).
- In this model, a process can publish a message on a topic, and other processes that subscribe to that topic can receive

and act on the message. This is common in event-driven architectures.

Failure Model:

This model deals with the faults and failures that can occur in a distributed system. It helps identify and fix faults using mechanisms like replication and error detection and recovery.

- **Crash Failures**: When a process or node stops working unexpectedly.
- **Omission Failures**: When a message is lost, leading to missed communication.
- **Timing Failures**: When a process takes longer than expected, causing delays or unsynchronized responses.
- **Byzantine Failures**: When a process sends malicious or unexpected messages that disrupt the system.

Security Model:

Distributed systems are vulnerable to attacks, unauthorized access, and data breaches. The security model helps understand security requirements, threats, vulnerabilities, and protection mechanisms.

- **Authentication**: Verifies the identity of users accessing the system to ensure only authorized and trusted entities can access it.
- **Encryption**: Converts data into a format that is unreadable without a decryption key, protecting sensitive information from unauthorized access.
- **Data Integrity**: Ensures data is not altered during storage, transmission, or processing, protecting it from unauthorized modifications or tampering.

CHECK YOUR PROGRESS-I

1. State True or False:

a) In modern computing, distributed systems and multiprocessor architectures are essential for achieving high performance, scalability, and reliability.
b) Symmetric multiprocessing (SMP) systems have processors with different instruction sets and architectures.
c) In asymmetric multiprocessing (AMP) systems, all processors share a common memory.
d) Tightly-coupled multiprocessor systems use multiple processors that share a common memory and I/O system.
e) Loosely-coupled multiprocessor systems consist of independent computers interconnected by a network.

2. Fill in the blanks:

a) Tightly-coupled multiprocessor systems are commonly used in applications that require high performance and reliability, such as __________ simulations and real-time systems.
b) In loosely-coupled multiprocessor systems, each computer in the system has its own __________ and operates independently.
c) One of the advantages of loosely-coupled multiprocessor systems is their high __________, as nodes can be added or removed without affecting the overall operation.
d) Designing software for loosely-coupled systems requires a more distributed computing __________, which can increase the complexity of the system.
e) Loosely-coupled systems can be more vulnerable to security threats, as the __________ provides an attack surface for hackers.

2.11 SUMMING UP

- Modern computing relies heavily on distributed systems and multiprocessor architectures to achieve high performance, scalability, and reliability.

- Multiprocessor systems use multiple CPUs to perform tasks simultaneously, enhancing processing power and efficiency. They are classified into:
    - Symmetric Multiprocessing (SMP): All processors share memory and have equal access to system resources, suitable for high-performance applications like scientific simulations.
    - Asymmetric Multiprocessing (AMP): Processors have different architectures and are assigned specific tasks, often used in specialized applications such as graphics processing.
- Tightly-Coupled Multiprocessor Systems: Multiple processors share a common memory and I/O system, communicating via shared buses for faster data exchange.
    - Advantages: Increased processing power, improved reliability (even with processor failures), and scalability through easy addition of processors.
    - Challenges: Increased system complexity, potential for memory contention, and communication overhead affecting performance.
- Loosely-Coupled Multiprocessor Systems: Independent computers interconnected via networks, each with its own memory, communicating through message passing.
    - Advantages: Flexibility to add or remove nodes without system disruption, cost-effectiveness with commodity computers, and high fault tolerance.
    - Challenges: Slower communication due to network latency, complexity in software design, and increased vulnerability to security threats.
- In Homogeneous Multiprocessor Systems
    - All processors are identical in hardware and software.
    - Share memory and communicate through shared memory or inter-processor mechanisms.
    - Advantages: Improved performance, scalability, and fault tolerance.
- In Heterogeneous Multiprocessor Systems
    - Processors differ in hardware, software, and architecture.
    - Communicate via shared memory or other mechanisms like message passing.
    - Advantages: Optimized for specific tasks, improved performance, and energy efficiency.

- Challenges: Complexity in programming, hardware compatibility, and interoperability.

- Physical Model:
  - Represents hardware components, nodes, links, middleware, and network topology.
  - Describes how computers and devices are interconnected and communicate.
- Architectural Model:
  - Defines overall structure and interaction of system components.
  - Types include client-server, peer-to-peer, layered, and microservices models.
- Fundamental Model:
  - Describes essential properties common to all architectural models.
  - Includes interaction model, failure model, and security model.

2.12 ANSWERS TO CHECK YOUR PROGRESS

1. a) True b) False c) False d) True e) True

2. a) scientific b) memory c) flexibility d) framework
e) network

2.13 POSSIBLE QUESTIONS

Short Answer Type Questions:

1. What are multiprocessors in modern computing?
2. Differentiate between symmetric multiprocessing (SMP) and asymmetric multiprocessing (AMP).
3. What is middleware, and what role does it play in distributed systems?
4. Explain the concept of load balancing in multiprocessor systems.
5. What are the advantages of loosely-coupled multiprocessor systems?

6. What is middleware and what role does it play in distributed systems?
7. Name and briefly explain two types of middleware.
8. What are the fundamental system models in distributed systems?

Long Answer Type Questions:

9. Describe the role of multiprocessor systems in achieving high performance, scalability, and reliability in modern computing. Discuss both homogeneous and heterogeneous systems.
10. Explain symmetric multiprocessing (SMP) and asymmetric multiprocessing (AMP) in detail. Compare their architectures, advantages, and applications in computing.
11. Discuss the concept of middleware in distributed systems. How does middleware facilitate communication and resource management among distributed computing nodes? Provide examples of different types of middleware and their applications.
12. Compare and contrast tightly-coupled and loosely-coupled multiprocessor systems. Discuss their architectures, advantages, limitations, and applications in various computing environments.
13. Examine the challenges and advantages of using loosely-coupled multiprocessor systems in distributed computing environments. Discuss factors such as communication overhead, scalability, fault tolerance, and security concerns.
14. Discuss the different types of system models in distributed systems. Explain each type and its significance in designing and analyzing distributed applications.
15. Explain the architectural model in distributed computing systems. Provide examples of different architectural models and their characteristics.
16. What are the key aspects of the fundamental system models in distributed systems? Discuss the interaction model, failure model, and security model in detail.

2.14 REFERENCES AND SUGGESTED READINGS

1. "Distributed Systems: Concepts and Design" by George Coulouris
2. “Designing Data-Intensive Applications" by Martin Kleppmann
3. “Distributed Systems: Principles and Paradigms" by Andrew Tanenbaum and Maarten Van Steen

×××

UNIT- 3
SYSTEM ARCHITECTURES

Unit Structure:

3.1 INTRODUCTION

- Web is a Service that allows computers to share and exchange data.
- Data can be images, video, audio, text, and documents, and that's why the web is referred to as Client-Server communication.
- A Client can be a machine or a program. A Client program is a program that allows the user to request the web. For example, the web browser is a user program that can make requests through the browser.
- A client, whether it is a machine or a program, is an appliance and a way to make requests through the web.
- A server is a computer program, NOT A DEVICE.
- High-performance computers are called servers because they run server programs.
- Servers provide functionality and serve other programs called clients.

3.2 OBJECTIVES

After going through this unit students will be able to learn

- The concept of Client and Server
- Variations in Client Server Model
- Client Server Architecture and Peer to Peer Architecture
- Application layering in a Distributed System
- Distributed operating system and its issues.

3.3 BASIC CONCEPT OF CLIENT SERVER MODEL

Server: A single server can serve multiple clients at the same time. Also, we can run multiple servers on one single machine, they are called virtual servers. There are several types of servers.

1.Web servers such as Apache that serve HTTP requests.
2.Database servers such as MYSQL that run database management systems.
A server can contain web resources, host web applications, store user and program data etc. It is used to serve hundreds or thousands of clients.
A server always listens for requests and as soon as it receives, it responds with a message.

3.4 CLIENT-SERVER MODEL

Now that we came to know what is client and what is server,we can define the client-server model in one sentence. Is an architecture on the web that splits computers into two sections, computers that require services are called clients, and computers that serveclients are called servers.The client-server model works through a request-response cycle through HTTP. The client-server model is just one way for the computers to communicate via the web.

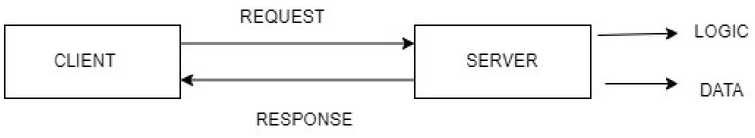
3.4.1 1-Tier Architecture

The 1-Tier architecture is also known as single-tier architecture. Here both client and server reside in one computer. This type of architecture is not suitable for web applications as data required by the application is available on the same computer or server. That means all the components required to run an application reside on the same computer.

3.4.2 2-Tier Architecture

In 2-tier architecture client request for services and the server responds. In the server side, both logic and data resides. Since some requests may need logical manipulation so after processing server responds. In this case client is called thin client since most of the processing done on server side. Client does not have many responsibilities in thin client. Eg. Hotstar, Netflix, E commerce sites, streaming applications.

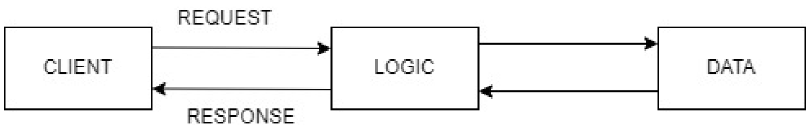
It not necessary that logic always sit on the server side. In some cases like Gaming app, Microsoft outlook, and video editing software where majority of the processing takes place in client side are called thick client.2 tier architecture mostly use in light weight websites or small businesses.



BASIC MODEL OF CLIENT SERVER ARCHITECTURE

3.4.3 3-Tier Architecture

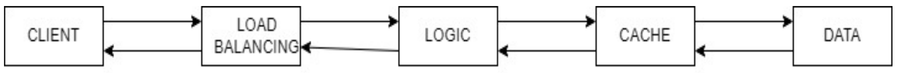
In some cases where large number of data is available, the load of server is very heavy. In that case extra layer is introduce. Client has presentation layer, the middle layer is application layer where logic resides and the third layer ,database layer where data resides. This is called 3 tier architecture. Example of 3 tier is Basic library management for school.



3 TIER ARCHITECTUTE

3.4.4 N-Tier Architecture

This above 3-Tier may not serve the purpose for a large complication application. In that case extra layer introduced between the client and logic layer such as the Load balancer and another layer cache layer between logic and data layer.Thus this type of architecture is known as N-tier architecture.



N-TIER ARCHITECTURE

STOP TO CONSIDER

1 tier architecture where client server database all resides in one machine.2 tier architecture where presentation layer run on client machine and database is store on server. In 2 tier if the number of client increases than the performance of the server degrades.3 tier architecture has three tiers, first tier contains client, second tier contains application layer and third tier contains database server. Application layer is also called business logic.

SELF ASKING QUESTIONS

1. How 1 tier is different from 2-tier and 3 –tier?
2. What is the concept of N-tier?

CHECK YOUR PROGRESS

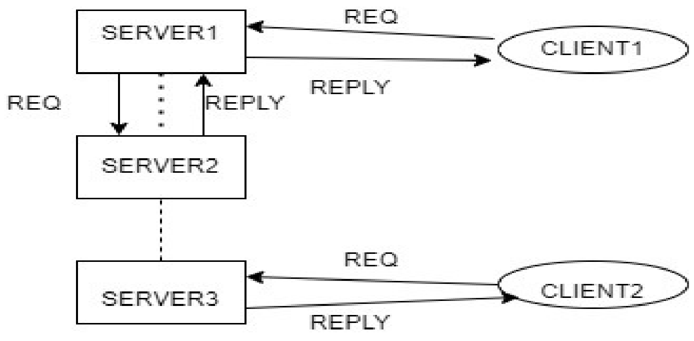
1. What is the advantage of the three-tier client-server database architecture against the two-tier alternative?
2. What are the limitations of two-tier architecture?

3.5 VARIATIONS IN CLIENT SERVER MODEL

3.5.1 Multiple Server Model

- Services may be provided by multiple server.

Server distribute resources among themselves. Thus communication among the servers takes place

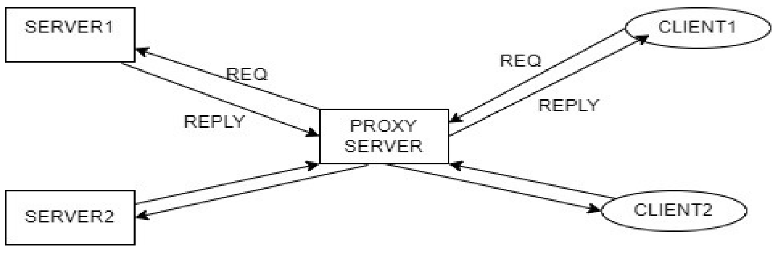


MULTIPLE SERVER MODEL

- In the above figure, Client1 requesting for a service to server1, that service may not available in server1,thus it request to server2. Server2 gives reply to server1 and thus to client1.

3.5.2 Proxy Server Model

- Proxy server provides copies(replication) of resources which are managed by other server.It acts as intermediate system between client and the server.
- Eg, Maintaining web resources cache. If client request resources ,then first it checks in the cache(proxy server) ,if not available then cache request to the server.

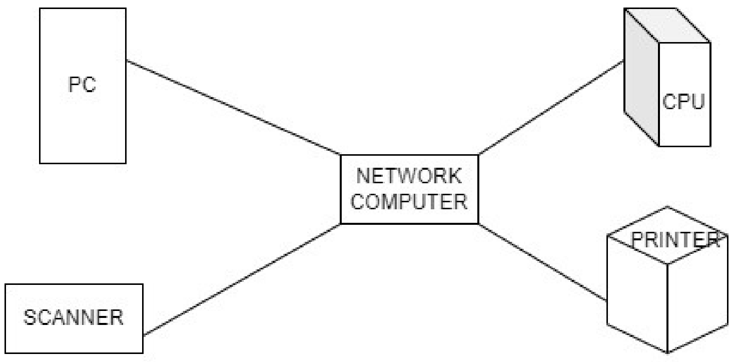


PROXY SERVER MODEL

- When a client makes a request, the request goes to the proxy server first, if the resources is available with the proxy server, itsends the response else it requests it to server1 or server2. Basically proxy server sits between client and the server. One advantage is that the load of the server reduces as it sits between client and server and also it enhances security as the proxy handles the request before going to the server so if there is any problem occurs it will affect the proxy server rather than the actual server.

3.5.3 Network Computers

- In network computers all the code is loaded from server and run locally.
- All files are stored and managed remotedly.
- Thus it is very simple and easy to managed.



NETWORK COMPUTER MODEL

- In the above figure, PC can access CPU, printer, scanner using network computer.

STOP TOCONSIDER

A proxy server is an intermediary between the client and the server. It acts as a load balancing, caching resources and anonymizing requests.

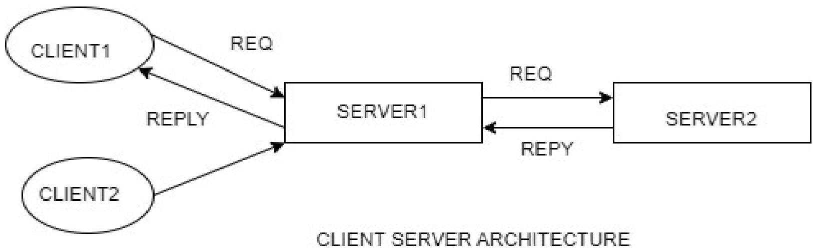
CHECK TO PROGRESS

3. What is the difference between a proxy server and a server?
4. Why do we need multiple servers?

3.6 ARCHITECTURE MODEL

3.6.1 Client Server Model

Two processes are available in client server architecture. Client has its own process and request for services or resources. Thus called requesting service. On the other hand, server has its own process ,provide the services requested by client. Thus it works on request/reply protocol. Remote Procodure calls or Remote method invocation(RMI) is used to invoke process for different computing device. Server can also request a service from another server.



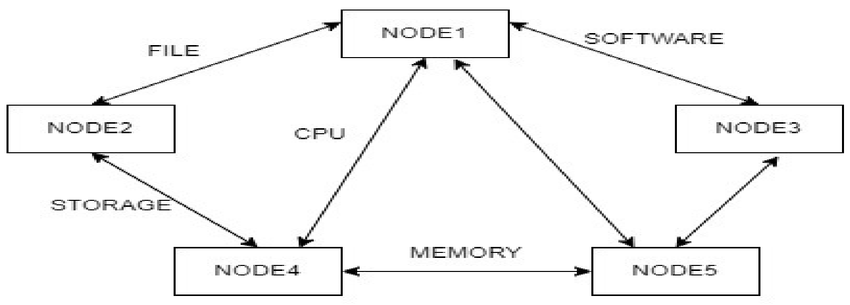
CLIENT SERVER ARCHITECTURE

In client server architecture, the server may act as a client also. For example, in the above, client1 asks requesting services from server1. Server1 may not have that particular information to reply back. In that case, it ask for information from another server say server2. Thus in that case server1 acts as a client. Whatever the information will be replied back by server2 to server1 and server1 response back to client1.

3.6.2 Peer-to-Peer Architecture

It composed of number of distributed, heterogenous, autonomous, dynamic peers in which participants share part of their own resources. Resources may be processing power, storage memory, software, files. All process/nodes play the same role.

In Peer-to-Peer model, there is no client and no server. Both computers act as requestors and response providers. In other words, each one can be a client and server. Each one is able to send and receive data with one another.



PEER TO PEER ARCHITECTURE

STOP TO CONSIDER

Architectural Model:

- It gives abstract view of distributed sysem
- .It helps in simplifying reasoning of system.

CHECK YOUR PROGRESS

5. Write advantages and disadvantages of Client server and Peer to Peer architecture.
6. What is the problem with peer-to-peer networks?
7. Why peer-to-peer is not safe?

.

3.7 APPLICATION LAYER IN DISTRIBUTED SYSTEM

It is the topmost layer of the internet model. Application layer programs are based on a client-server model. We can say that the application layer allows the user to use the internet. The application layer lies between the user and transport layer. The application layer provides security, different application programs, and addressing. The client cannot access the data or application directly, it is possible through the server only. Whenever a client requests services from the server, it has to include the address of the server, its own address as a source address.

3.7.1 Functions of Application Layer

a) **Identifying communication partners:** It identifies the availability of communication partners for an application with data to transmit.
b) **Determining resource availability:** It identifies whether sufficient network resources are available for the requested communication.
c) **Synchronizing communication:** All the communications that occur between the applications require cooperation which is managed by an application layer

3.7.2 Services of the application layer

a. Electronic mail
b. net news(Usenet)
c. WWW(world wide web)
d. Multimedia
e. Remote file transfer and access

STOP TO CONSIDER

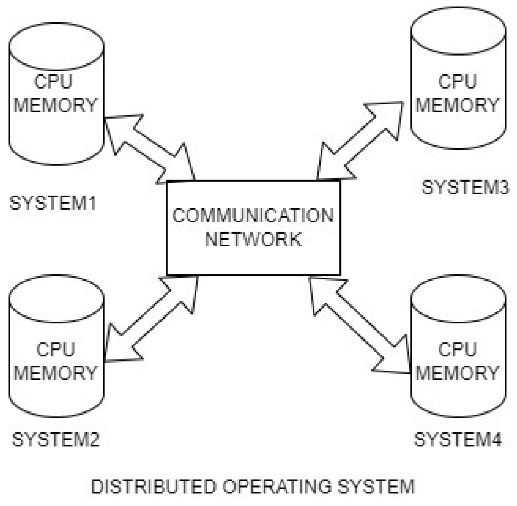
- The application layer and the end user can interact directly with the software application

CHECK TO PROGRESS

8. What is the main responsibility of the application layer?
9. What is the need of application layer?
10. What is the position of application layer?

3.8 DISTRIBUTED OPERATING SYSTEM(DOS) AND ITS ISSUES

A distributed operating system is a system in which a number of computers are connected to perform real-time applications. This multiple computers are connected by communication lines. Each of the system are said to be loosely connected as each of the system has its own applications, data and operating system. It allows to perform together as if there exists a big system. We can increase the performance of a real-time application by using distributed operating system.



3.8.1 Types of Distributed Operating system

3.8.1.1 Middleware

Also called message-oriented middle ware. It performed real-time applications by connecting two or more operating systems. It allows access of real-time information among the different systems by passing messages. It maintains data integrity among different systems. It helps to increase the growth of organizational efficiency and also streamlines business processes.

3.8.1.2 Client-server

Already discussed about client server model, where client requests for service and server response by replying with the required information.

3.8.1.3 Peer-to-Peer

Peer to Peer where there is no client and the server. Each one act as a client and the server.

3.8.2 Advantages and Disadvantages

3.8.2.1 Advantages

- It is very easy to share resources and to perform real-time applications in distributed operating systems as system are connected to each other via a network.
- Distributed operating system meet the needs of the business, making it more flexible and efficient.
- It is easier to control and monitor since the system can be managed centrally.
- Due to the rise of big data and the need for real-time applications, distributed operating systems became very popular

3.8.2.2 Disadvantages

- It is very difficult to administer and manage.
- Due to the increased number of systems, there is a greater possibility of risk of failure of the system.
- The cost of maintaining a distributed operating system is very high.

STOP TO CONSIDER

Distributed operating system in which several computers are connected through a common communication channel. Each one has its own individual processor and memory.

SELF ASKING QUESTION

1. What is the main purpose of middleware in distributed systems?
2. What are the functions of a middleware system?

3.9 SUMMING UP

This unit tells us about the Client-server model and its variations. How the client-server model is organized and how the organization is different from the Peer-to-Peer model. It describes the concept of the distributed operating system, its advantages and disadvantages. It tells about the application layer, how it works, its types and advantages and disadvantages.

3.10 REFERENCES AND SUGGESTED READING

1. https://unacademy.com/content/bank-exam/study-material/computer-knowledge/distributed-operating-system/

2. https://www.youtube.com/watch?v=ePvqvXEkVIk

3. Distributed System By Andrew Tanenbaum and Maarten van Steen

3.11 MODEL QUESTIONS

1. What are the challenges of the client-server model?

2. What is required by the client-server model?

3. Why is peer-to-peer network bad?

4. What are the limitations of client-server?

5. What are the features of peer to peer network?

6. What is the working principle of peer-to-peer networks?

7. What are the responsibilities of middleware?

8.What is a middleware issue?

9. What application layer protocol is commonly used?

10.What are the key challenges in building distributed systems?

3.12 ANSWER TO CHECK YOUR PROGRESS

ANSWER 1. Each tier can be worked separately by the development team and can be updated without impacting other tier .Each tier runs on its own infrastructure

ANSWER 2. In 2-tier performance decreases as the number of users increases. It is difficult to implement reliable security as users need to have login information for every database server.

ANSWER3. A user access websites by sending request to the server via web browser and the web server returns back the required information to the user. But the proxy server sits between the user and the web server that act as an intermediary.

ANSWER 4. Multiple server are used for balancing the load. Instead of allocating a single server for a specific function, we can allocate different servers for various functions. Thus reducing the load on a single server. Hence utilization of memory, CPU ,RAM and storage takes place efficiently.

ANSWER 5.ADVANTAGES OF CLIENT SERVER

a) If one system stops working ,it will not affect the other.
b) The size of the system can be set according to the requirement.

DISADVANTAGE:

a. Cost of set up is more.
b. If the central system fails,it will affect the whole system.

ADVANTAGES OF PEER-TO-PEER

a. No extra investment in server hardware and software
b. Easy setup then server-based network.

DISADVANTAGES OF PEER TO PEER

a) Additional load on the computer because of resource sharing.
b) Lack of central organization.

ANSWER 6**.** Lack of centralized control: Managing and coordinating network activities is difficult as the absence of a centralized network in peer-to-peer networks. Also difficult to coordinate complex activities, ensure data integrity and also difficult to enforce consistent policies.

ANSWER 7. Since in P2P, both acts a client and the server, so they are more easy for the attackers to gain access to each machine in the network.

ANSWER8. From the cloud-based storage or from a database, the user can access data using the application layer only. Within the same network or different network the transfer of files takes place using application layer. Also transfer of file from each other takes place using the application layer only.

ANSWER9.Web browsers and email clients are the end-user software that uses the application layer. It provides protocols that allow software to send and receive information and present meaningful data to users.

ANSWER 10**.**It is the topmost layer in the Open Systems Interconnection (OSI) model. It acts as the interface between the network and the end-user applications, and allows communication between software applications and lower-level network services.

×××

UNIT: 4
CLOCK SYNCHRONIZATION AND LOGICAL CLOCKS

Unit Structure:

4.1 INTRODUCTION

In distributed systems, multiple processes operate on different machines, each with its own local clock. These clocks often differ due to variations in hardware, environmental conditions, and inherent clock drift. Accurate and consistent timekeeping across these distributed components is crucial for ensuring the correct sequence of events, maintaining consistency, and coordinating actions among processes. This is where clock synchronization comes into play.

4.2 UNIT OBJECTIVES

After going through this unit, you will be able to:

- understand the importance of clock synchronization in distributed systems and explain why accurate timekeeping is essential for the coordination and consistency of operations.
- describe the concepts of external clock synchronization and the methods used to achieve it.
- explain internal clock synchronization techniques.
- discuss logical clocks and their role in distributed systems.
- identify and analyze the challenges and trade-offs involved in clock synchronization in distributed systems.

4.3 WHAT IS SYNCHRONIZATION?

In general, synchronization refers to the coordination or alignment of actions or events in time. In the context of computing, synchronization refers to the techniques and mechanisms used to coordinate the execution of multiple threads or processes in a way that ensures correctness and consistency.

When multiple threads or processes are running concurrently, they may access shared resources or data structures, and if their access is not properly coordinated, it can lead to various problems such as race conditions, deadlocks, and data corruption. Synchronization techniques provide a way to avoid such problems by ensuring that threads or processes access shared resources in a mutually exclusive and orderly manner.

Some common synchronization mechanisms include locks, semaphores, monitors, and barriers. Locks are used to enforce mutual exclusion, which means that only one thread or process can access a shared resource at a time. Semaphores are similar to locks, but they allow multiple threads or processes to access a shared resource with a specified limit. Monitors provide a higher-level abstraction for synchronization by combining locks with condition variables that allow threads or processes to wait for certain events to occur. Barriers are used to synchronize a group of threads or processes by ensuring that they all reach a certain point in their execution before continuing.

Overall, synchronization is an essential concept in concurrent programming, and it is crucial to ensure that shared resources and data structures are accessed in a coordinated and consistent manner to avoid problems and ensure correct program behavior.

4.3.1 Clock Synchronization

Clock synchronization is the process of ensuring that the clocks of different devices in a distributed system are aligned with each other. In a distributed system, devices may have their own clocks, which can drift over time due to factors such as temperature changes, aging of components, or differences in clock frequencies. If the clocks of different devices are not synchronized, it can lead to various problems such as incorrect time stamping of events, data inconsistencies, and even security vulnerabilities.

There are several approaches to clock synchronization in distributed systems, and some of the most common ones are:

NTP (Network Time Protocol): NTP is a protocol used to synchronize the clocks of devices over a network.

PTP (Precision Time Protocol): PTP is a protocol similar to NTP, but it is designed for high-precision clock synchronization in industrial and scientific applications.

GPS (Global Positioning System): GPS is a satellite-based navigation system that can be used to synchronize clocks in distributed systems.

There are several algorithms that can be used to synchronize clocks in a distributed system without relying on external time sources. One of the most common algorithms is the Cristian's algorithm, which uses a time server to estimate the clock offset between a client and the server. Another algorithm is the Berkeley algorithm, which uses a centralized server to synchronize the clocks of multiple devices in a LAN.

Clock synchronization is an important aspect of distributed systems, and it can have significant impacts on the correctness and performance of applications. Proper clock synchronization ensures that events are timestamped correctly, data is consistent across devices, and security protocols are effective. Different clock synchronization approaches have their own strengths and limitations, and the choice of approach depends on factors such as accuracy requirements, network topology, and available resources.

4.3.2 Need of Clock Synchronization

In a distributed system, multiple devices or processes may communicate with each other over a network. Each device or process has its own local clock, which can drift over time due to various factors such as temperature changes, aging of components, and differences in clock frequencies. This can result in time discrepancies between different devices, which can cause a range of problems such as:

Inconsistent data: If different devices record timestamps for the same event, but their clocks are not synchronized, it can be difficult to determine the order in which the events occurred. This can lead to inconsistencies in the data, making it difficult to analyze and process.

Incorrect results: In some applications, such as financial transactions or scientific experiments, accurate timing is critical. If the clocks of different devices are not synchronized, it can lead to incorrect results or even failures.

Security vulnerabilities: In many security protocols, accurate timing is critical for preventing attacks such as replay attacks or denial-of-service attacks. If the clocks of different devices are not synchronized, it can create vulnerabilities in the security protocols.

Coordination problems: In a distributed system, multiple devices may need to coordinate their actions or synchronize their operations. If the clocks of different devices are not synchronized, it can be difficult to ensure that they execute their operations in the correct order and at the correct time.

To address these problems, clock synchronization is needed in distributed systems. Clock synchronization ensures that the clocks of different devices are aligned with each other, reducing the time discrepancies between them. This enables consistent data recording, accurate timing, secure communication, and better coordination between devices. Clock synchronization is an essential aspect of distributed systems, and it is critical for ensuring correct and reliable operation of applications.

Example:

One example of the need for clock synchronization can be seen in a financial transaction system. Suppose that a financial institution operates a distributed system consisting of multiple servers that handle transactions from clients. Each server has its own clock, and transactions are recorded with timestamps based on the local clock of the server.

If the clocks of the servers are not synchronized, it can lead to discrepancies in the recorded timestamps. For example, suppose a client initiates a transaction at server A, and the transaction is recorded with a timestamp of 10:00:00 AM based on the local clock of server A. If the transaction is then forwarded to server B for processing, but the clock of server B is 5 minutes ahead of the clock of server A, the transaction will be recorded at server B with a timestamp of 10:05:00 AM. This can create confusion and errors when analyzing the transaction records, and it can also affect the correctness of financial reports and audits.

To avoid such problems, clock synchronization is necessary in a financial transaction system. The clocks of the servers need to be synchronized with each other to ensure that transactions are recorded with consistent and accurate timestamps, regardless of which server they are processed on. This enables the financial institution to maintain the integrity of its transaction records, ensure accurate billing and auditing, and provide reliable services to its clients.

This example illustrates how clock synchronization is crucial in distributed systems, especially in applications where accurate timing is critical. Without proper clock synchronization, inconsistencies and errors can arise, leading to incorrect results, security vulnerabilities, and coordination problems.

4.4 EXTERNAL CLOCK SYNCHRONIZATION

External clock synchronization refers to the process of synchronizing the clock of a device or system with an external reference clock. The reference clock is typically a highly accurate time source that is accessible over a network or through a dedicated hardware interface.

External clock synchronization is crucial in distributed systems for several reasons:

- **Event Ordering**: Ensures that events are recorded in the correct sequence.
- **Consistency**: Maintains data consistency across distributed databases.
- **Coordination**: Facilitates coordinated actions among distributed processes.
- **Security**: Enhances security protocols that rely on time-based authentication mechanisms

4.4.1 Network Time Protocol (NTP)

The Network Time Protocol (NTP) is a widely used protocol designed to synchronize the clocks of computers over a network. Developed by David L. Mills in 1985, NTP is essential for ensuring that all devices in a distributed system maintain a consistent and accurate time. It operates over the User Datagram Protocol (UDP) and is capable of synchronizing clocks to within a few milliseconds over the public Internet and even better over local area networks (LANs).

NTP plays a key role in achieving these objectives by providing a reliable method to synchronize the system clocks of devices within a network to a common reference time.

How NTP Works?

NTP operates in a hierarchical system of time sources. The hierarchy is organized into strata, where each stratum level indicates the distance from the reference clock:

- **Stratum 0**: High-precision timekeeping devices such as atomic clocks or GPS clocks.
- **Stratum 1**: Servers directly connected to Stratum 0 devices, serving as primary time servers.
- **Stratum 2 and below**: Servers that synchronize with servers in higher strata, creating a cascading effect of time synchronization.

NTP uses a client-server model where clients periodically query NTP servers to adjust their local clocks. The protocol involves exchanging timestamps between the client and the server, allowing the client to calculate the round-trip delay and offset.

Synchronization Process

1. **Timestamp Exchange**: NTP clients send requests to NTP servers, and servers respond with packets containing timestamps indicating when the request was received and when the response was sent.
2. **Delay and Offset Calculation**: The client uses the timestamps to calculate the round-trip delay and clock offset. This involves four key timestamps:
   - T1: Time when the client sends the request.
   - T2: Time when the server receives the request.
   - T3: Time when the server sends the response.
   - T4: Time when the client receives the response.

   The round-trip delay (d) and offset (θ) are calculated using the following formulas:

   $$d = (T4 - T1) - (T3 - T2)$$

   $$\theta = \frac{(T2 - T1) + (T3 - T4)}{2}$$
3. **Clock Adjustment**: The client adjusts its clock by the calculated offset (θ) to align with the server's time.

Features and Benefits of NTP:

- **Accuracy**: NTP can achieve synchronization accuracy within milliseconds over the Internet and microseconds in LAN environments.

- **Fault Tolerance**: NTP supports multiple servers for redundancy, improving reliability in case one or more servers fail.
- **Scalability**: The hierarchical structure of NTP allows it to scale efficiently across large networks.
- **Security**: NTP includes cryptographic mechanisms to prevent malicious attacks and ensure the integrity of the time synchronization process.

Challenges and Limitations:

- **Network Latency**: Variations in network latency can affect the accuracy of time synchronization.
- **Asymmetric Delays**: Differences in delay times between the client-server and server-client paths can introduce errors.
- **Server Load**: Heavy load on NTP servers can lead to delays and less accurate synchronization.

It is a robust and widely adopted protocol for external clock synchronization in distributed systems. By aligning the clocks of distributed devices to a common reference time, NTP ensures accurate event ordering, consistency, and coordination, which are essential for the correct functioning of distributed systems. Despite some challenges, the benefits of NTP in maintaining synchronized clocks far outweigh its limitations, making it a critical component in modern networked environments.

4.4.2 GPS Based Synchronization

Global Positioning System (GPS) is a satellite-based navigation system that provides highly accurate time and location information to GPS receivers on Earth. While GPS is primarily known for its navigation capabilities, it also plays a crucial role in providing precise time synchronization for various applications, including distributed systems. GPS-based synchronization is one of the most accurate methods for achieving external clock synchronization, offering time precision within nanoseconds.

How GPS-Based Synchronization Works?

GPS-based synchronization leverages the precise atomic clocks on GPS satellites to provide accurate time information to receivers on Earth. The process involves several key steps:

- **Satellite Signal Reception**: GPS receivers on the ground receive signals from multiple GPS satellites. Each satellite broadcasts its current time and position.
- **Time Calculation**: The receiver calculates the travel time of signals from each satellite by comparing the broadcast time with the reception time.
- **Position and Time Solution**: By receiving signals from at least four satellites, the GPS receiver can calculate its precise location and correct its internal clock. This process is known as trilateration.
- **Clock Adjustment**: The GPS receiver adjusts its local clock based on the highly accurate time information received from the satellites. This time information is typically derived from atomic clocks onboard the satellites, ensuring high precision.

Features and Benefits of GPS-Based Synchronization:

- **High Accuracy**: GPS provides time synchronization accuracy within nanoseconds, making it one of the most precise methods available.
- **Global Availability**: GPS signals are available worldwide, making it suitable for synchronization in geographically dispersed systems.
- **Independence from Network Conditions**: Unlike network-based protocols such as NTP, GPS-based synchronization is not affected by network latency or asymmetric delays.
- **Redundancy**: Multiple satellites ensure redundancy, enhancing the reliability of the synchronization process.

Applications of GPS-Based Synchronization:

- **Telecommunications**: Synchronizing base stations in cellular networks to ensure seamless handoffs and efficient spectrum usage.
- **Power Grids**: Coordinating operations of power plants and substations to maintain grid stability and prevent blackouts.

- **Financial Systems**: Ensuring accurate timestamps for financial transactions to meet regulatory requirements and prevent fraud.
- **Scientific Research**: Providing precise timing for experiments and data collection in fields such as astronomy and particle physics.

Challenges and Limitations

- **Signal Obstruction**: GPS signals can be obstructed by buildings, mountains, or other structures, limiting their effectiveness in certain environments.
- **Multipath Interference**: Reflections of GPS signals from surfaces like buildings and water can cause errors in time calculation.
- **Atmospheric Conditions**: Variations in the ionosphere and troposphere can affect signal travel time, introducing slight inaccuracies.
- **Dependency on Satellite Constellation**: The accuracy of GPS-based synchronization depends on the number of visible satellites and their positions relative to the receiver.

This technique is a very effective way to keep clocks accurate in distributed systems. By using the precise time from GPS satellites, devices in a network can stay in sync. This is important for keeping the order of events correct, ensuring data is consistent, coordinating actions, and maintaining security. Even though there are challenges like signal blockage and weather conditions, the benefits make GPS-based synchronization essential for many high-precision and time-sensitive tasks.

4.5 INTERNAL CLOCK SYNCHRONIZATION

Internal clock synchronization refers to the process of ensuring that all the clocks within a distributed system are synchronized with each other. Unlike external clock synchronization, which relies on an external time source (such as GPS or NTP), internal clock synchronization focuses on achieving consistency and agreement among the clocks of different nodes within the system itself. This is crucial because, in a distributed system, each node typically has its own clock, and these clocks can drift apart over time due to various factors, such as differences in clock hardware and environmental conditions.

The importance of Internal clock synchronization are as follows:

1. **Event Ordering Consistency:** Internal clock synchronization is essential for maintaining a consistent order of events across different nodes.
2. **Data Consistency:** Ensuring that all nodes have synchronized clocks helps maintain data consistency.
3. **Coordinated Actions:** Many applications in distributed systems require coordinated actions between nodes. For instance, in distributed control systems, synchronized clocks ensure that actions are taken at the correct time, allowing the system to function smoothly and predictably.
4. **Fault Tolerance:** Synchronized clocks can enhance fault tolerance in distributed systems.
5. **Security:** Time synchronization is crucial for security mechanisms such as authentication and authorization.

Internal clock synchronization is crucial for making sure distributed systems run accurately, efficiently, and safely. It helps solve problems caused by clocks getting out of sync and ensures that events happen in the right order, data stays consistent, actions are well-coordinated, system failures are handled better, performance is improved, and security is strong.

4.5.1 Cristian's Algorithm

Cristian's Algorithm is a method used in distributed systems to synchronize the clocks of different nodes with a time server. The goal is to minimize the difference between the server's clock and the clocks of the client nodes.

Steps of Cristian's Algorithm:

1. **Client Request**: A client node sends a request to the time server asking for the current time.
2. **Server Response**: The server receives the request and immediately sends back the current time (T_server) to the client.
3. **Client Adjustment**: Upon receiving the server's time, the client adjusts its own clock by considering the network delay.

To account for the network delay, the client measures the round-trip time (RTT) of the message and adjusts the server's time accordingly. The estimated time when the server sent its response can be calculated as:

$$T_{\text{client}} = T_{\text{server}} + \frac{\text{RTT}}{2}$$

where, RTT is the round-trip time measured by the client.

Example:

Let's consider an example to understand Cristian's Algorithm in action.

1. **Client Sends Request**:
   - At time $T_1=10$ (according to the client's clock), the client sends a request to the server.
2. **Server Responds**:
   - The server receives the request and immediately sends the time $T_{\text{server}}=15$.
3. **Client Receives Response**:
   - The client receives the server's response at time $T_2=18$ (according to the client's clock).
4. **Calculate RTT**:
   - The client calculates the round-trip time as:

     $\text{RTT}=T_2-T_1=18-10=8$
5. **Adjust Client's Clock**:
   - The client adjusts its clock based on the server's time and the estimated network delay:

$$T_{\text{client}} = T_{\text{server}} + \frac{\text{RTT}}{2} = 15 + \frac{8}{2} = 15 + 4 = 19$$

So, the client sets its clock to 19, aligning it closer to the server's time while accounting for the network delay.

Cristian's Algorithm is a popular method for clock synchronization in distributed systems for several reasons:

- **Simplicity and Ease of Implementation:** The algorithm is simple to understand and implement. It involves basic message exchanges between the client and the server, making it accessible for systems with limited resources.

  Here, the only requirement is a reliable time server and basic network communication capabilities, making it suitable for a wide range of environments.

- **Improved Accuracy over Simple Synchronization:**Unlike simple synchronization methods, Cristian's Algorithm accounts for network delay by measuring the round-trip time (RTT) of messages. This helps in reducing the synchronization error.

  The client adjusts its clock based on the server's time and the measured network delay, leading to more accurate synchronization.

- **Reduced Synchronization Error:**By calculating the RTT and adjusting the time accordingly, Cristian's Algorithm compensates for the delay in message transmission, which helps in achieving closer synchronization between client and server clocks.

- **Suitability for Various Network Conditions:**The algorithm is particularly effective in networks with relatively low delay and jitter. It can provide accurate synchronization in environments where network conditions are stable and predictable.

- **Wide Applicability:**Cristian's Algorithm can be applied in various distributed systems, such as distributed databases, real-time systems, and networked applications where synchronized time is crucial for operations.

Also, there are limitations this algorithm and they are:

- **Single Point of Failure**: The algorithm relies on a single time server, which can be a single point of failure. If the server becomes unavailable, clients cannot synchronize their clocks.

- Network Delay Variability: High variability in network delays can affect the accuracy of synchronization. The algorithm assumes a relatively stable network condition for accurate time adjustments.

Cristian's Algorithm is a practical and effective solution for clock synchronization in many distributed systems. Its simplicity, ease of implementation, and ability to reduce synchronization errors by accounting for network delays make it a valuable tool for achieving accurate time synchronization across networked devices.

4.5.2 Berkeley Algorithm

The Berkeley Algorithm is a way to synchronize clocks in distributed systems when there is no single, highly accurate time server. It works by choosing one node to act as the coordinator. This coordinator collects the current times from all the nodes, calculates the average time, and then tells each node how to adjust their clock to match this average. This method is particularly useful when no single node has a very accurate clock.

Steps of Berkeley Algorithm:

- **Coordinator Selection:** Choose one node to act as the coordinator.
- **Polling:** The coordinator asks all other nodes for their current time.
- **Time Collection:** Each node sends its current time back to the coordinator.
- **Average Calculation:** The coordinator calculates the average time from all the collected times, including its own.
- **Adjustment Calculation:** The coordinator figures out how much each node's time needs to change to match the average.
- **Time Adjustment:** The coordinator tells each node how much to adjust their clocks to sync with the average time.

Example:

Consider a distributed system with four nodes (A, B, C, and D). Node A is selected as the coordinator. The current times on each node are as follows:

- Node A: 10:00
- Node B: 10:05
- Node C: 09:58
- Node D: 10:02

Here’s how the Berkeley Algorithm would synchronize these clocks:

1. **Coordinator Polling**: Node A (the coordinator) polls nodes B, C, and D, asking for their current time.
2. **Time Collection**:
   - Node B reports 10:05
   - Node C reports 09:58
   - Node D reports 10:02
3. **Average Calculation**:
   - Node A collects these times: 10:00, 10:05, 09:58, and 10:02.
   - It calculates the average time:

     (10:00+10:05+09:58+10:02)/4=10:01.25

1. **Adjustment Calculation**:
   - Node A determines the difference from the average:
     - Node A 10:00−10:01.25=−1.25 minutes
     - Node B: 10:05−10:01.25=+3.75 minutes
     - Node C: 09:58−10:01.25=−3.25 minutes
     - Node D: 10:02−10:01.25=+0.75 minutes
2. **Time Adjustment**:
   - Node A sends each node the amount of time to adjust:
     - Node A adjusts by −1.25-1.25−1.25 minutes to match the average.
     - Node B adjusts by −3.75-3.75−3.75 minutes to match the average.
     - Node C adjusts by +3.25+3.25+3.25 minutes to match the average.
     - Node D adjusts by −0.75-0.75−0.75 minutes to match the average.

After these adjustments, all nodes will have their clocks synchronized to 10:01.25.

The benefits of using Berkeley algorithm for clock synchronization in distributed systems are:

- **Single Clock Independence:** Unlike some synchronization methods that rely on a single highly accurate time server, the Berkeley Algorithm does not require any node to have an extremely accurate clock. This makes it suitable for environments where no node can be guaranteed to have a perfectly accurate time source.
- **Fault Tolerance:** By using the average time from multiple nodes, the algorithm reduces the impact of any one node having an inaccurate clock. This approach increases the overall robustness and reliability of the system.
- **Coordinated Adjustments:** The Berkeley Algorithm ensures that all nodes adjust their clocks in a coordinated manner, preventing any significant time discrepancies between them. This is crucial for applications that require synchronized actions and consistent data states.

Also, there are limitations this algorithm and they are:

- **Coordinator Dependency:** The algorithm depends on the coordinator to compute the average time and distribute adjustments.
- **Communication Overhead:** Requires polling and communication with all nodes, which can introduce delays in large systems.
- **Synchronization Accuracy:** The accuracy depends on the stability of the network and the precision of the nodes' clocks.

The Berkeley Algorithm is a practical and efficient way to keep clocks synchronized in a distributed system without needing an external accurate time source.

CHECK YOUR PROGRESS-I

1. State True or False:

a) NTP stands for Network Time Protocol and is used for clock synchronization across computers over a network.

b) GPS-based synchronization can achieve time precision within microseconds.

c) The Berkeley Algorithm requires a highly accurate external time source for synchronization.

d) Clock synchronization is not important for financial transaction systems.

e) In NTP, the hierarchical structure is organized into strata, with Stratum 0 representing servers directly connected to atomic clocks.

2. Fill in the blanks:

a)______ is a satellite-based navigation system used for time synchronization.

b)Cristian's Algorithm estimates the clock ______ between a client and server.

c) In NTP, the client adjusts its clock based on the calculated ______ and offset.

d)GPS-based synchronization provides accuracy within ______.

e) Clock synchronization helps in achieving correct ______ and coordinated actions in distributed systems.

4.6 LOGICAL CLOCKS IN DISTRIBUTED SYSTEM

Logical clocks help organize events in distributed systems by assigning them logical times, which is useful when physical clocks are not perfectly in sync due to issues like network delays and clock drift.

In a distributed system, events can happen at the same time on different nodes, making it hard to determine the exact order of events. Logical clocks solve this problem by giving each event a logical time based on how it relates to other events, without depending on the actual time they occurred.

4.6.1 Lamport Clock

One example of a logical clock is the Lamport clock, which is a simple algorithm that assigns a unique timestamp to each event based on the timestamp of the preceding event and the messages exchanged between nodes. The Lamport clock algorithm operates as follows:

- Each node maintains a local counter that is incremented by 1 for each event.
- When an event occurs, the node assigns a timestamp to the event by appending the local counter value to the node's unique identifier.
- When a node sends a message to another node, it includes its own timestamp in the message.
- When a node receives a message from another node, it updates its own timestamp to the maximum of its current timestamp and the timestamp received in the message.

The Lamport clock algorithm ensures that events that are causally related, such as a message being sent and received, are assigned timestamps that reflect their causal ordering. However, events that are not causally related may be assigned the same timestamp or different timestamps, depending on the order in which they occur on different nodes.

4.6.2 Vector Clock

Another example of a logical clock is the vector clock, which extends the Lamport clock algorithm by maintaining a vector of counters instead of a single counter. Each entry in the vector corresponds to a node in the distributed system, and the value of each entry is the local counter value of the corresponding node. When an event occurs, the node increments its own counter and sends its vector clock along with the message. When a node receives a message, it updates its own vector clock by taking the maximum of its current vector and the vector received in the message.

Logical clocks are useful for various distributed system applications such as event ordering, synchronization, and debugging. By assigning logical timestamps to events, distributed systems can reason about the ordering of events and ensure that processes are executing consistently despite delays and failures in the network.

4.7 CHALLENGES AND TRADE-OFFS IN CLOCK SYNCHRONIZATION

The challenges in Clock Synchronization in distributed systems are as follows:

- **Clock Drift**: The Clocks in distributed systems drift due to factors such as temperature variations, aging components, and differences in oscillator frequencies. This drifting clocks lead to time discrepancies among nodes, affecting the correctness of event ordering and synchronization.
- **Network Latency**: Variations in network latency can affect the accuracy of time synchronization protocols like NTP or PTP. Higher latency can lead to synchronization errors and reduce the precision of synchronized clocks.
- **Fault Tolerance**: Synchronization protocols must be resilient to failures in network connectivity, hardware failures, or server downtime. Failure to handle faults can disrupt synchronization, leading to inconsistencies and potential system failures.
- **Security**: Ensuring the integrity and authenticity of time synchronization messages is crucial, especially in security-sensitive applications. Insecure synchronization can lead to vulnerabilities such as replay attacks or unauthorized access.
- **Scalability**: Synchronizing clocks in large-scale distributed systems with thousands of nodes poses scalability challenges.Scalability issues can affect synchronization accuracy and performance, especially as the network size increases.

The trade-offs in Clock Synchronization in distributed systems are as follows:

- **Accuracy vs. Overhead**: Increasing synchronization accuracy often requires more frequent clock updates and higher computational overhead.

Balancing accuracy with system performance is essential to meet application requirements without excessive resource consumption.

- **Centralized vs. Decentralized Approaches**: Centralized synchronization approaches (e.g., NTP) provide high accuracy but can introduce single points of failure.

  Decentralized approaches (e.g., PTP) distribute synchronization responsibility but require more complex coordination and management.

- **Real-time vs. Eventual Consistency**: Real-time consistency ensures immediate synchronization, whereas eventual consistency allows for gradual convergence over time.

  Choosing between these depends on application needs, such as real-time data processing or eventual data consistency in distributed databases.

- **Precision vs. Network Load**: Achieving higher precision in synchronization may increase network traffic and load.

  Optimizing synchronization protocols to minimize network overhead while maintaining acceptable precision is crucial, especially in bandwidth-constrained environments.

- **Security vs. Performance**: Implementing robust security measures (e.g., cryptographic authentication) adds overhead to synchronization protocols.

  Balancing security requirements with performance constraints ensures that clock synchronization does not compromise system responsiveness or scalability.

4.8 SUMMING UP

- Synchronization ensures coordinated actions or events in computing. It prevents issues like race conditions, deadlocks, and data corruption in concurrent processes.
- Clock Synchronization isEssential for aligning clocks in distributed systems to avoid time discrepancies.Common methods include NTP, PTP, and GPS-based synchronization.Algorithms like Cristian's and Berkeley's handle internal synchronization without external sources.

- Clock Synchronization is necessary Ensures consistent data, accurate timing, and secure communication.
- NTP is ahierarchical system with strata for time sources.
- NTP utilizes UDP for clock adjustment based on round-trip delay and offset calculations. It provides fault tolerance, scalability, and security features.
- **GPS-Based Synchronization** relies on satellite signals for precise time and location data. It offers high accuracy and global coverage, ideal for telecommunications, power grids, and financial systems.

4.9 ANSWERS TO CHECK YOUR PROGRESS

1. a) True b) False c) False d) False e) True

2. a) GPS b) offset c) delayd) nanoseconds
   e) event ordering

4.10 POSSIBLE QUESTIONS

Short Answer Type Questions:

1. What is clock synchronization in distributed systems?
2. Name two common protocols used for external clock synchronization.
3. What are the primary challenges in achieving accurate clock synchronization?
4. Explain the role of NTP in networked environments.
5. Why is GPS-based synchronization highly accurate?
6. Describe Lamport's Logical Clock and its fundamental principle.
7. What problem do Vector Clocks solve that Lamport's Logical Clocks cannot?
8. What are the key challenges in achieving accurate clock synchronization in distributed systems?
9. Explain one advantage and one limitation of using Cristian's Algorithm for clock synchronization.

Long Answer Type Questions:

10. Describe the process of internal clock synchronization in distributed systems. Why is it necessary?
11. Discuss the steps involved in Cristian's Algorithm for clock synchronization. Provide an example to illustrate its operation.
12. Explain how the Berkeley Algorithm works for clock synchronization in distributed systems. Provide an example scenario where the algorithm would be beneficial.
13. Compare and contrast Cristian's Algorithm and the Berkeley Algorithm in terms of their approach to clock synchronization, advantages, and limitations.
14. Discuss the challenges and considerations involved in achieving accurate internal clock synchronization in distributed systems. How do algorithms like Cristian's and Berkeley's address these challenges?
15. Discuss the operational principle of Lamport's Logical Clock with a step-by-step example involving two processes communicating through messages.
16. Explain the significance of logical clocks in ensuring causality tracking and event ordering in distributed systems. Provide examples of scenarios where this is crucial.
17. Describe the Berkeley Algorithm for clock synchronization in distributed systems. Include its steps, advantages, and limitations.
18. Discuss the challenges involved in achieving fault-tolerant clock synchronization in large-scale distributed systems. What strategies can be employed to mitigate these challenges?

4.11 REFERENCES AND SUGGESTED READINGS

1. "Distributed Systems: Concepts and Design" by George Coulouris
2. "Designing Data-Intensive Applications" by Martin Kleppmann
3. "Distributed Systems: Principles and Paradigms" by Andrew Tanenbaum and Maarten Van Steen

UNIT 5:
MESSAGE ORDERING AND CAUSAL ORDER

Unit Structure:

5.1 INTRODUCTION

Message ordering is a fundamental aspect of distributed systems, ensuring that messages exchanged between different components are processed in a consistent and predictable manner. Proper message ordering is crucial for maintaining system correctness, consistency, and reliability, especially in environments where components are geographically dispersed and operate concurrently.

5.2 UNIT OBJECTIVES

After going through this unit, you will be able to:

- *understand* the Concept of Message Ordering.
- *understand* how Lamport's Logical Clock is used to establish a partial order of events in distributed systems.
- *analyze* the use of Vector Clocks in relation with causal order of messages.
- *discuss* the concept of causal ordering of messages and its significance in maintaining consistency in distributed systems.

5.3 MESSAGE ORDERING AND ITS IMPORTANCE

In a distributed system, multiple processes or nodes operate concurrently and communicate by sending and receiving messages. These processes may execute at different speeds and may not have access to a global clock, leading to challenges in determining the correct sequence in which messages should be processed. Message ordering refers to the rules or protocols that determine the sequence in which messages are delivered and processed by the receiving nodes.

The following points emphasize the importance of Message Ordering in Distributed Systems.

- **Consistency and Correctness**

Making sure that messages are processed in the right order is essential for keeping a distributed system consistent.Correct message ordering prevents anomalies, such as processing a response before the corresponding request or applying updates out of sequence.

- **Synchronization of Operations**

In distributed systems, many tasks need to be coordinated across different nodes. Message ordering makes sure that these tasks, especially those that depend on each other, are carried out in the right order, keeping the system running smoothly and logically.

- **Causal Relationships**

In distributed systems, some events can influence others. Message ordering methods like causal ordering make sure these connections are respected, so that an effect isn't processed before its cause. This is crucial in systems where tasks rely on the outcomes of earlier actions.

- **Fault Tolerance and Recovery**

If the network breaks down or a node fails, message ordering protocols make sure that messages are delivered and processed in the right order once the system is back up. This keeps the system consistent and reliable even after failures. Having messages in the right order also helps with undoing operations correctly if something goes wrong, by reversing the actions in the order they happened.

- **Coordination and Consensus**

In distributed systems where nodes must agree on a shared state or make a group decision, message ordering is vital. Making sure all nodes process messages in the same order prevents situations where the system gets divided, ensuring everything stays consistent and unified.

- **Performance Optimization**

Well-planned message ordering can make distributed systems run more efficiently by cutting down on the need for extra synchronization and coordination. For instance, if some tasks can happen at the same time without needing strict order, the system can work faster and with less delay. However, it's important to balance this with the need for consistency, as being too relaxed with the order can cause problems with data accuracy.

5.4 MESSAGE ORDERING: LAMPORT'S LOGIAL CLOCK

In the previous unit, we talked about Lamport's Logical Clock in simple terms. Now, let's try to get the detail about it. The Lamport Logical Clock is a mechanism developed by Leslie Lamport in 1978 to order events in a distributed system where a global clock is not available. In distributed systems, different processes or nodes may operate independently and communicate through message passing, making it difficult to determine the order of events accurately.

5.4.1 How Lamport's Logical Clock Works?

Now, let's discuss how Lamport Logical Clock Ensures Message Ordering.

1. Event Timestamps

In a distributed system, each process maintains its own logical clock. Whenever an event occurs, the logical clock is incremented, and the event is tagged with the updated timestamp. This timestamp helps the system to track the order of events across multiple processes.

There are three types of events that Lamport's Logical Clock handles:

- **Internal events** (actions within a process)
- **Send events** (when a process sends a message to another process)
- **Receive events** (when a process receives a message from another process)

2. Message Send and Receive

- **Sending Messages:** When a process sends a message, it increments its logical clock and attaches the clock value to the message. This ensures that the timestamp reflects the order of the event when it was sent.
- **Receiving Messages:** When a process receives a message, it compares its own logical clock with the timestamp of the message. The receiving process sets its logical clock to be the higher of its current value or the message's timestamp,

and then increments it. This ensures that events are ordered properly, even if they occurred on different processes.

For example:

- Process A sends a message with timestamp 10 to Process B.
- Process B has a logical clock of 8 when it receives the message.
- Process B updates its clock to 11 (the max of its clock and the message timestamp, plus 1) before processing the message.

3. The "Happened-Before" Relation (→)

One of the key concepts of Lamport Logical Clock is the "happened-before" relation (denoted →), which helps define the order in which events should be processed.

- **Internal Order:** If an event A happens in the same process before event B, then A → B.
- **Message Ordering:** If an event A involves sending a message, and event B involves receiving that message, then A → B.
- **Transitivity:** If A → B and B → C, then it follows that A → C.

These rules ensure that messages which are causally related are processed in the correct order. For example, if Process A sends a message to Process B, the sending event should be ordered before the receiving event, preserving the logical sequence of events.

4. Partial Order vs. Total Order

Lamport Logical Clock provides a **partial order** of events. This means that it can order events that are causally related (i.e., where one event influences another), but it cannot definitively order events that are independent of each other.

For example, if Process A sends a message to Process B, those two events are causally linked and can be ordered. However, if Process C performs an action independently of A and B, its events are not directly related, and the clock may not provide a strict order between events on A, B, and C.

To obtain a **total order** of events (which is sometimes required in systems like distributed databases or consensus algorithms), additional mechanisms, such as attaching process IDs to timestamps, are needed.

Let’s walk through an example:

Step 1: Initial Setup

- Process A and Process B both start with a logical clock value of 0.

Step 2: Process A sends a message to Process B

- Process A increments its logical clock to 1, sends the message, and attaches the timestamp 1 to the message.

Step 3: Process B receives the message

- Process B receives the message when its clock is still 0. It compares the message timestamp (1) with its own clock (0), sets its clock to 2 (the higher value + 1), and then processes the message.

Now, the events have been ordered correctly:

- Process A’s send event has timestamp 1.
- Process B’s receive event has timestamp 2.

This ensures that B processes the message **after** it was sent, preserving the correct order of events.

5.4.2 Applications of Lamport’s Logical Clock in Message Ordering

Now let’s discuss the applications of Lamport’s Logical Clock in Message Ordering. Following are some of the application areas of Lamport’s Clock.

Distributed Databases:

- Ensures that updates to the database are applied in the correct order, even when they are processed by different nodes in the system. This maintains data consistency across the distributed system.

Event Logging and Debugging:

- Helps in tracking the sequence of events that occur across different processes, making it easier to debug and analyze system behavior.

Mutual Exclusion Algorithms:

- Used in algorithms that require processes to access shared resources without conflicts, ensuring that requests are ordered and processed correctly.

Consensus Protocols:

In distributed systems where nodes must agree on a particular state, Lamport Logical Clocks help maintain an agreed-upon order for decision-making.

5.5 MESSAGE ORDERING: VECTOR CLOCK

Vector clocks are an extension of Lamport's logical clocks, designed to overcome the limitations of partial ordering and better capture the causal relationships between events in distributed systems. While Lamport's clock can order events that are causally related, it can't detect independent events that happen simultaneously in different processes. Vector clocks provide a more detailed mechanism, allowing for the detection of concurrency, which is crucial in maintaining consistency and ordering in distributed systems.

5.5.1 How Vector Clock Works?

A vector clock is an array of logical clocks, one for each process in the system. Each process maintains its own vector clock, which keeps track of both its own events and the events of other processes it has communicated with.

Components of a Vector Clock:

- **Vector Array:** Each process maintains a vector (an array of integers), where each element in the array corresponds to a logical clock value of a particular process.
  - For example, in a system with three processes A, B, and C, process A's vector clock would be something like [Va, Vb, Vc], where Va is A's local clock, Vb is what A knows about B's clock, and Vc is what A knows about C's clock.
- **Clock Updates:**
  - **Internal Events:** When a process performs an internal action, it increments its own entry in the vector.

- **Send Events:** When a process sends a message, it increments its own clock entry in the vector and attaches the entire vector to the message.
- **Receive Events:** When a process receives a message, it compares its vector with the vector attached to the message. It updates each entry in its vector clock to be the maximum of its own value and the value in the received vector.

Vector clocks can ensure that messages are ordered correctly by comparing the vector timestamps of two events. The comparison works as follows:

Causal Order: If every entry in vector clock VC1 of event A is less than or equal to the corresponding entry in vector clock VC2 of event B, and at least one entry in VC1 is strictly less than VC2, then event A causally happened before event B. This is denoted as A → B.

Concurrent Events: If two events' vector clocks have entries that are neither completely less than nor greater than each other (i.e., some entries are greater and others are less), then the events are considered concurrent (independent of each other).

Equal Events: If two events have identical vector clocks, they are considered to have occurred at the same logical point in time.

Example of Message Ordering with Vector Clocks:

Let's consider a system with three processes, P1, P2, and P3, each maintaining a vector clock.

1. **Initial Setup:**
   - P1: [0, 0, 0]
   - P2: [0, 0, 0]
   - P3: [0, 0, 0]
2. **Internal Event at P1:**
   - P1 increments its own clock, so the new vector clock at P1 is [1, 0, 0].
3. **P1 Sends a Message to P2:**
   - P1 increments its clock before sending the message, making the vector [2, 0, 0].

    - P1 sends the message to P2 with the attached vector [2, 0, 0].
4. **P2 Receives the Message:**
    - P2 compares its vector [0, 0, 0] with the received vector [2, 0, 0].
    - P2 updates its clock to the maximum of the two vectors and increments its own clock: [2, 1, 0].
5. **P2 Sends a Message to P3:**
    - P2 increments its own clock to [2, 2, 0] and sends this vector to P3.
6. **P3 Receives the Message:**
    - P3 updates its vector clock to [2, 2, 1] after receiving the message.

5.5.2 Applications of Vector Clock in Message Ordering

Now let's discuss the applications of Vector Clock in Message Ordering. Following are some of the application areas of Vector Clock.

Causal Consistency in Distributed Databases:

Causal consistency is a type of rule used in distributed databases to make sure that actions that depend on each other are done in the right order. It ensures that if one action affects another, the system will keep that order across all parts of the network. However, actions that are not related or happen at the same time can be done in any order, giving the system more flexibility and better performance compared to stricter rules like sequential or linear consistency.

Version Control Systems:

In distributed databases, version control is essential for managing different versions of data that may be updated by multiple users at different locations. Since multiple copies of the same data can be updated simultaneously across nodes, vector clocks help track these updates, ensuring that versions are handled correctly and conflicts are resolved in a consistent manner.

Concurrency Control:

In distributed databases, concurrency control ensures that multiple operations, often happening simultaneously across different nodes, are executed in a manner that preserves consistency. Vector clocks are an essential tool in managing concurrency by tracking the causal relationships between events and maintaining the correct order of operations across distributed systems.

Event Logging and Debugging:

In distributed databases, event logging and debugging are crucial for tracking operations and identifying issues that may arise during the execution of distributed processes. Vector clocks are an effective tool for ordering events and identifying causal relationships, making them highly valuable for both logging and debugging in such systems.

5.6 CAUSAL ORDER OF MESSAGES

In distributed systems, different processes communicate by sending messages to each other. To keep the system working correctly, it's important that these messages are delivered in the right order. Causal message ordering is a method that ensures messages are delivered in the correct sequence when one message affects another. This is crucial in systems where actions on one node can impact actions on another. By respecting the order of events, causal message ordering helps maintain the proper flow and consistency of operations across the system.

5.6.1 Lamport's "Happened-Before" Relation

In the section 5.4.1, we discussed the "Happened-Before" Relation in a very brief manner. Now, let's discuss the same in detail.

The **"Happened-Before" relation** is a foundational concept in distributed systems, introduced by Leslie Lamport in his seminal 1978 paper "Time, Clocks, and the Ordering of Events in a Distributed System." This relation is critical for understanding the causal ordering of events in systems where multiple processes or nodes execute concurrently and independently.

In distributed systems, there is no global clock, and events may occur at different times across different processes. The "Happened-

Before" relation, often denoted as **A → B**, allows the system to infer causal relationships between events based on the notion of causality, rather than relying on absolute physical time.

The Key Concepts related to “happened-before” relation are:

- **Causal Ordering of Events:** The "Happened-Before" relation helps define the causal relationship between events in a distributed system. If an event **A** causally affects event **B**, we say that **A happened before B**, or **A → B**.
- **Partial Ordering:** The "Happened-Before" relation creates a **partial order** of events in the system. It ensures that events that are causally related are ordered, but it does not impose any ordering on events that are independent of each other (i.e., events that occur concurrently).

The "Happened-Before" relation (denoted →) is defined by the following three rules:

1. Within a Single Process:

If two events occur within the same process, the event that occurs earlier is said to have happened before the later event.

For example, let **A** and **B** be two events in a process. If **A** occurs before **B** in the same process, then **A → B**.

This is a simple linear ordering of events within a single process, which reflects the natural flow of time.

2. Message Passing Between Processes:

If one process sends a message to another process, the sending event happens before the receiving event.

For example, let **A** be the event of sending a message from **Process 1** to **Process 2**, and let **B** be the event of receiving the message in **Process 2**. In this case, **A → B**, since sending a message must logically occur before the message can be received.

This rule captures the causal relationship between message-sending and message-receiving events in distributed systems.

3. Transitivity:

The "Happened-Before" relation is **transitive**, meaning that if **A → B** and **B → C**, then **A → C**.

For example, if **A** is an event in **Process 1** that happened before **B**, and **B** is an event in **Process 2** that happened before **C**, then **A → C**.

Transitivity allows the system to infer causal relationships across multiple events and processes. This property is crucial for maintaining consistency in distributed systems, as it ensures that all events that are causally related are ordered correctly.

Now, let's understand the "happened-before" relation with the help of an example, Consider three processes in a distributed system: **P1**, **P2**, and **P3**. Events occur in each process, and some processes send messages to each other. Below is an illustration of how the "Happened-Before" relation works:

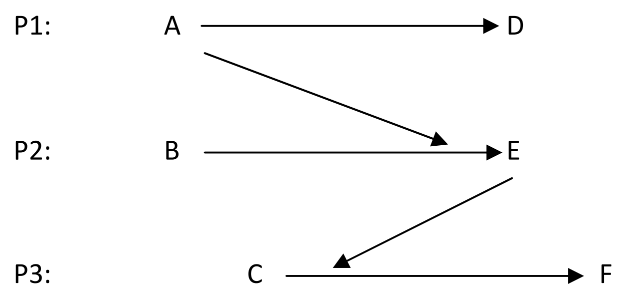


Fig. 5.1

- **A → D** because **A** happens before **D** in **Process P1** (Rule 1).
- **A → B** because **A** sends a message to **B** in **Process P2**, meaning **A** happened before **B** (Rule 2).
- **B → E** because **B** happens before **E** in **Process P2** (Rule 1).
- **C → F** because **C** happens before **F** in **Process P3** (Rule 1).
- **A → E** by transitivity, because **A → B** and **B → E** (Rule 3).

This partial ordering only captures events that have a causal relationship. Events **D** and **F**, for example, are concurrent and do not have a defined causal order because there is no message exchanged between them and no direct happened-before relationship.

The limitations of this relation are as follows:

- **Partial Order**: The "Happened-Before" relation only defines a partial order of events. Events that are concurrent are not ordered, which means that the relation cannot capture a total order of events in the system.

- **Performance Overhead**: Tracking causal relationships using vector clocks or other mechanisms introduces additional overhead in terms of communication and storage. Each process must maintain and exchange timestamp information, which can be costly in large-scale systems.

5.7 BIRMAN-SCHIPER-STEPHENSON (BSS) PROTOCOL

The **Birman-Schiper-Stephenson (BSS) protocol** is a method used in distributed systems to ensure **causal message ordering**. In distributed systems, messages are exchanged between different nodes or processes, and it is important to ensure that messages that are causally related are delivered in the correct order. The BSS protocol is designed to maintain this order.

5.7.1 Key Features

The key features of the BSS protocol are as follows:

- **Causal Message Ordering:** The BSS protocol guarantees that if one message causally affects another, the first message is delivered before the second. This ensures that messages are delivered in a sequence that respects their causal relationships, preventing inconsistencies in the system.
- **Vector Clocks for Tracking Causality:** Each process in the system maintains a **vector clock** that tracks the logical time of events. When a message is sent, it includes the sender's current vector clock value. The receiving process uses this information to determine the causal relationship between the received message and other events.
- **Decentralized Approach:** The BSS protocol operates without relying on a global clock. Instead, it uses a decentralized method where each process maintains its own vector clock to track causality. This is important in distributed systems, where a global clock is often not feasible due to network delays and system failures.
- **Concurrency Support:** The BSS protocol allows independent or concurrent messages (those without a causal relationship) to be delivered in any order. This flexibility improves system

performance by not imposing unnecessary ordering on messages that are not causally linked.

- **Message Buffering:** If a message arrives out of causal order (e.g., its causal predecessors haven't been received yet), it is temporarily buffered. The system delivers the message only after all causally preceding messages have been processed. This mechanism helps maintain causal consistency in the system.
- **Avoids Overhead of Total Ordering:** The BSS protocol imposes **partial ordering** of messages based on causality, which is more efficient than **total ordering** (where all messages must be ordered). This reduces overhead, especially in systems where messages are independent or occur concurrently.

5.7.2 How BSS Protocol Works?

Let's discuss a step-by-step explanation of how the BSS protocol works:

1. Initialization of Vector Clocks

- Each process in the system maintains a **vector clock**.
- For a system with **N** processes, each process **P_i** keeps a vector clock **VC_i** of size **N** (where **i** is the process ID).
- Initially, all entries in the vector clock are set to zero: **VC_i = [0, 0, 0, ..., 0]**.

2. Sending a Message

When a process sends a message, it needs to update its vector clock and attach it to the message.

Step 1: Before sending the message, the process increments its own entry in the vector clock. For example, if **P_1** is sending a message, it increments **VC_1[1]** by 1.

- o **VC_1 = [current values] → VC_1[1] = VC_1[1] + 1**.

Step 2: The updated vector clock is attached to the message and then sent to the receiving process.

- o **Message M** is sent with the timestamp **VC_1** attached.

3. Receiving a Message

When a process receives a message, it checks the vector clock attached to the message and compares it with its own vector clock to decide whether the message can be delivered immediately or must be delayed.

Step 1: The receiving process compares the vector clock from the message with its own vector clock. Specifically, it checks if the message respects the causal ordering by comparing each entry of the vector clocks.

Step 2: The message can be delivered only if all previous events that causally affect this message have been delivered. This is determined by checking the following condition:

- For each **P_j** (any other process), **VC_received[j] ≤ VC_receiver[j]**.
- This ensures that the process has received all necessary messages from other processes before processing the current message.

Step 3: If the condition holds, the message is delivered and processed. The process then updates its own vector clock by setting each entry to the maximum of its own vector clock and the received vector clock:

- **VC_receiver[k] = max(VC_receiver[k], VC_received[k])**, for each **k**.

Step 4: The process then increments its own vector clock entry to reflect the new event.

4. Message Buffering

If the condition from Step 2 fails (meaning that some previous causally related messages have not yet been received), the process cannot deliver the message immediately.

Step 1: The message is buffered (temporarily stored) until the missing causally related messages are received.

Step 2: The process periodically checks whether the buffered messages can now be delivered based on updated vector clocks.

5. Message Delivery and Causal Order

- The BSS protocol ensures that messages are delivered in a way that respects causal dependencies. This is crucial in distributed systems where the order in which events occur matters for system consistency.

- By maintaining and updating vector clocks and comparing them with incoming messages' timestamps, the protocol enforces causal message ordering without requiring a global clock.

6. Concurrency and Independent Messages

- If two messages are causally independent (i.e., they are not related by the "happened-before" relation), they can be delivered in any order.
- The BSS protocol does not impose strict ordering on messages that are not causally linked, improving system performance by allowing concurrency.

5.7.3 Advantages of BSS Protocol

The BSS Protocol is used in distributed systems to ensure causal message ordering. It provides several advantages in maintaining causal consistency across nodes. Here are the key advantages:

1. Causal Message Ordering: The BSS protocol ensures that messages are delivered in the order of their causal relationships. If one message causally affects another, it ensures that the first message is delivered before the dependent one, preserving the system's logical consistency.

2. Efficient Message Communication: The BSS protocol is designed to work efficiently in distributed systems where processes communicate via message passing. It ensures that messages are delivered in the correct order without excessive overhead, contributing to more reliable and organized communication across nodes.

3. Decentralized Control: BSS operates without the need for a centralized coordinator, which reduces the bottleneck that can be introduced by centralized systems. Instead, each process independently ensures that it respects the causal order of messages, leading to improved system scalability.

4. Handling of Concurrent Messages: The protocol allows for the concurrent delivery of messages that are not causally related. This means that independent messages can be processed in any order, which improves efficiency and throughput by allowing more parallelism in message processing.

5. Message Tagging with Timestamps: The BSS protocol uses timestamps to track causal dependencies. These timestamps are included with each message, enabling receiving processes to determine the correct order for processing. This approach is relatively simple to implement and ensures consistent ordering without requiring complex data structures.

6. Fault Tolerance: In the event of failures, the BSS protocol ensures that causal dependencies are respected once the system recovers. Messages are not processed until all causally related messages have been delivered, helping to maintain consistency even in the presence of failures or network delays.

7. Scalability: Due to its decentralized nature and the use of timestamps for causal ordering, the BSS protocol is scalable and can be applied to large distributed systems with many nodes. The lack of a centralized control mechanism helps it perform efficiently as the system grows.

8. Supports Asynchronous Systems: The BSS protocol works well in asynchronous environments where messages can arrive at different times. It ensures that causal relationships are preserved even when there is no global clock or synchronization between processes.

5.8 SCHIPER-EGGLI-SANDOZ (SES) PROTOCOL

The **Schiper-Eggli-Sandoz (SES) protocol** is a protocol used in distributed systems to ensure causal message delivery. Like the Birman-Schiper-Stephenson (BSS) protocol, the SES protocol ensures that messages are delivered in the correct causal order. However, it differs in its approach, as it is designed to work in fully asynchronous systems where there is no assumption about the speed or synchronization of message delivery between processes.

5.8.1 Key Features

Here are the key features of SES Protocol:

- **Causal Message Ordering**: The SES protocol ensures that if a message causally influences another, it is delivered before the dependent message. This preserves the causal relationships between events across different processes in the distributed system.

- **No Message Buffering**: Unlike other protocols like the BSS, the SES protocol does not rely on **buffering** undelivered messages. Instead, the system ensures that causally dependent messages are always sent in the correct order, so there is no need to delay message processing.
- **Piggybacking of Causal Information**: Causal information is piggybacked on every message sent between processes. Each message contains metadata (causal information) that informs the receiver about the causal dependencies of the message.
- **Reduced Overhead**: The SES protocol is designed to reduce the overhead associated with ensuring causal ordering. By eliminating message buffering and managing causal dependencies through metadata attached to messages, it simplifies the communication process between distributed nodes.
- **Causal History Tracking**: The SES protocol uses **causal histories** to track the causal relationships between messages. Instead of relying on vector clocks, it records the history of past messages that have been sent, allowing the receiving process to reconstruct the causal order.

5.8.2 How SES Protocol Works?

Let's discuss a step-by-step explanation of how the SESprotocol works. Here's a step-by-step explanation of how the SES protocol works:

1. Initialization of Causal History

- Each process in the distributed system maintains a **causal history** of the messages it has sent and received.
- This causal history contains information about the dependencies between messages, i.e., which messages must precede others to maintain causal order.

2. Sending a Message

When a process sends a message to another process, it attaches its causal history to the message.

Step 1: The process updates its causal history to reflect the current message being sent.

- The causal history is a record of all previous messages sent or received by this process that causally affect the current message.

Step 2: The updated causal history is piggybacked onto the message. This history provides the receiver with the information it needs to determine the causal dependencies of the message.

Step 3: The message is sent along with the piggybacked causal information to the recipient process.

3. Receiving a Message

When a process receives a message, it must check the causal history attached to the message to decide whether it can deliver the message immediately or if it must wait for any missing dependencies.

Step 1: The receiving process inspects the **causal history** piggybacked on the message. This history lists the messages that must have been delivered before the current message can be processed.

Step 2: The process verifies whether it has already received all the messages that are part of the causal history. If all messages from the causal history have already been delivered, the message can be processed immediately.

Step 3: If there are any missing causal dependencies (i.e., messages listed in the causal history that the receiving process has not yet received), the message is **not processed immediately**. Instead, the process waits until the missing causal messages are received.

4. Message Delivery

Once the receiving process has determined that all necessary causal messages have been received, it can proceed to deliver the message.

Step 1: The message is delivered to the application layer, and the process updates its own causal history to reflect the delivery of the message.

Step 2: The process updates its record of delivered messages, ensuring that future messages are delivered in the correct causal order.

5. Direct Message Delivery (No Buffering)

Unlike other protocols that use message buffering when a message arrives out of order, the SES protocol ensures that the sender only sends messages when the causal order can be respected. This eliminates the need for buffering at the receiver.

Step 1: Since the causal history is attached to every message, and each message is sent only when it can be delivered in the correct order, the receiver does not need to buffer undelivered messages.

Step 2: Messages are delivered immediately as soon as all necessary causal dependencies are met, improving efficiency.

6. Concurrency and Independent Messages

Concurrent messages (messages that are independent of each other and have no causal relationship) can be delivered in any order. The SES protocol does not enforce ordering on messages that are not causally related, which allows for greater concurrency and efficiency in distributed systems.

5.8.3 Advantages of SES Protocol

The SES Protocol offers several advantages in distributed systems, particularly in environments that require causal message ordering. Here are the key benefits:

1. No Message Buffering: Unlike some other protocols that require messages to be buffered until all causal dependencies are satisfied, the SES protocol avoids buffering altogether. This simplifies the implementation and reduces memory overhead, as messages are delivered directly when received.

2. Efficient Causal Message Ordering: The SES protocol ensures that messages are delivered in the correct causal order by piggybacking causal information on each message. This guarantees that messages with dependencies are processed in the proper sequence, ensuring system consistency without the need for complex reordering.

3. Designed for Asynchronous Systems: The SES protocol is particularly suited for fully **asynchronous systems** where there is no assumption about the speed or timing of message delivery. It ensures causal consistency even in environments where messages may arrive unpredictably or out of order.

4. Reduced Communication Overhead: By piggybacking causal information directly onto messages, the SES protocol minimizes the need for additional control messages or synchronization steps. This reduces the communication overhead compared to protocols that require constant coordination or acknowledgment messages.

5. Concurrency Support: The SES protocol allows for **concurrent message delivery** when messages are independent or have no causal relationships. This improves system performance by enabling more parallelism in message processing.

6. Simpler Causal Dependency Management: The use of causal history (attached to messages) simplifies the tracking and management of dependencies between messages. Processes can easily determine the causal order by examining the history attached to incoming messages, without needing to maintain complex data structures like vector clocks.

7. Improved Scalability: The protocol's efficient handling of causal relationships and its ability to eliminate message buffering make it well-suited for large-scale distributed systems. By reducing the need for centralized control or excessive synchronization, the SES protocol can scale more easily across multiple nodes.

CHECK YOUR PROGRESS-I

1. State True or False:

a) Causal consistency ensures that only causally related operations are executed in the correct order.

b) Lamport's Logical Clock can establish a total order of events in a distributed system.

c) Vector clocks can detect concurrent events that occur in different processes.

d) In distributed systems, message ordering is only necessary in fault-tolerant environments.

e) A message sent with a lower timestamp must always be received before a message with a higher timestamp.

2. Fill in the blanks:

a) In a distributed system, ensuring that messages are delivered in the correct order is important for maintaining _______ and _______.

b) The Lamport Logical Clock is used to order _______ in a distributed system where a global clock is not available.

c) Vector clocks provide more detailed mechanisms than Lamport's clock by allowing for the detection of _______.

d) Causal consistency ensures that actions that are causally related are executed in the _______ order across all nodes.

e) Vector clocks are commonly used in distributed databases to manage different _______ of data and ensure consistency.

5.9 SUMMING UP

- The importance of Message Ordering in Distributed Systems are:
    - Consistency and Correctness: Ensures correct processing order to prevent anomalies.
    - Synchronization of Operations: Coordinates dependent tasks across nodes.

    - Causal Relationships: Preserves the cause-effect sequence of events.
    - Fault Tolerance and Recovery: Maintains order after network or node failures.
    - Coordination and Consensus: Helps achieve agreement on shared states.
    - Performance Optimization: Improves efficiency by balancing consistency and flexibility.
- The mechanism of Lamport's Logical Clock is as follows:
    - Event Timestamps: Processes increment clocks for internal, send, and receive events.
    - Message Send/Receive: Sender increments and attaches the timestamp; receiver updates its clock to ensure proper ordering.
    - Happened-Before Relation (→): Defines causal relationships using internal order, message ordering, and transitivity.
    - Partial vs. Total Order: LLC provides partial ordering, and additional mechanisms are needed for total order.
- The mechanism of Vector Clock is as follows:
    - Vector Array: Each process maintains a vector clock tracking events of itself and others.
    - Clock Updates: Internal events, send, and receive events update the vector clock.
    - Causal Order: Events are ordered by comparing vector entries.
    - Concurrency Detection: VC detects concurrent (independent) events.
- The operation of BSS Protocol is as follows:
    - Vector Clocks: Each process maintains a vector clock to track events.
    - Message Sending: A process increments its vector clock and attaches it to the message.

    - Message Receiving: The receiver checks vector clocks to determine if the message can be delivered or should be delayed.
    - Message Buffering: If causality is violated, messages are buffered until dependencies are resolved.
    - Concurrency: Independent messages are delivered in any order.
- The operation of BSS Protocol is as follows:
    - Causal History: Each process maintains a causal history of past messages.
    - Message Sending: Causal history is attached to the message.
    - Message Receiving: The receiver checks causal history to ensure all dependencies are met before delivery.
    - Direct Delivery: Messages are delivered immediately when causal dependencies are satisfied.
    - Concurrency: Independent messages are delivered without enforcing an order.

5.10 ANSWERS TO CHECK YOUR PROGRESS

1. a) True b) False c) True d) False e) False

2. a) consistency, correctness b) events c) concurrency

d) correct e) versions

5.11 POSSIBLE QUESTIONS

Short Answer Type Questions:

1. What role does Lamport's Logical Clock play in distributed systems?
2. How are causal relationships between events maintained in distributed systems?
3. How does vector clock differ from Lamport's Logical Clock?
4. Why is causal consistency important in distributed databases?

5. What does the "happened-before" relation mean in Lamport's Logical Clock?
6. How do vector clocks help in version control systems?
7. What is the main limitation of Lamport's Logical Clock?

Long Answer Type Questions:

8. Discuss the importance of message ordering in distributed systems.
9. Discuss how Lamport’s Logical Clock ensures message ordering in distributed systems.
10. Explain how vector clocks improve over Lamport's Logical Clock in detecting concurrent events.
11. Explain how vector clocks are used in event logging and debugging in distributed systems.
12. What are the trade-offs between using total order and partial order message delivery in distributed systems?

5.12 REFERENCES AND SUGGESTED READINGS

1. "Distributed Systems: Concepts and Design" by George Coulouris
2. “Designing Data-Intensive Applications" by Martin Kleppmann
3. “Distributed Systems: Principles and Paradigms" by Andrew Tanenbaum and Maarten Van Steen

×××

UNIT: 6
DISTRIBUTED SNAPSHOT AND TERMINATION DETECTION

Unit Structure:

6.1 INTRODUCTION

A Distributed Snapshot is a way to take a picture of the global state of a distributed system without stopping it. It helps record the status of all processes and the messages being sent between them, allowing us to understand the system's behavior or detect problems.

While Termination Detection is used to figure out when all the processes in a distributed system have finished their work and no more messages are being sent. This is important to know when the entire system has completed its tasks, especially when processes are waiting for messages.

6.2 UNIT OBJECTIVES

After going through this unit, you will be able to:

- understand the Concept of Global State in Distributed Systems.
- study the Chandy-Lamport Snapshot Algorithm.
- explore the Message-Driven Approach.
- understand how the Dijkstra-Scholten algorithm detects termination in distributed computations.
- understand Termination Detection in Distributed Systems.

6.3 GLOBAL STATE IN DISTRIBUTED SYSTEMS

Global State in Distributed Systems refers to the collective state of all processes and communication channels at a particular instant in time across a distributed system. Since distributed systems consist of multiple, independent processes that run concurrently and communicate through messages, capturing a consistent global state is a complex task.

6.3.1 Importance of Capturing Global State in Distributed Systems

Capturing the Global State in Distributed Systems is important for several reasons, as it plays a crucial role in ensuring correct and efficient operation in scenarios where distributed systems need coordination, consistency, or failure detection. Below are the key reasons why capturing global state is important:

1. Consistency and Correctness:

- Capturing a consistent global state helps in recovering from failures. By recording the global state, the system can resume operation from a known, valid checkpoint in case of failures (check pointing).
- Debugging a distributed system is challenging due to the concurrent and independent execution of processes. Capturing global states helps in analysing the behavior of the system and identifying bugs or abnormal behaviours.
- A consistent global state is needed to detect deadlocks in distributed systems, where two or more processes are waiting on each other indefinitely.

2. Snapshot Algorithms:

- Global states are used in algorithms like the Chandy-Lamport snapshot algorithm to capture a "snapshot" of the system's state without stopping the distributed computation. This snapshot can be used for monitoring, checkpointing, and rollback recovery.
- Some algorithms for detecting global termination rely on capturing the global state to determine when all processes have finished their tasks and no messages are in transit.

3. Distributed Consistency and Coordination:

- To make meaningful decisions in distributed systems, it is important to capture a consistent cut of the system's global state. This is essential for ensuring that actions taken on the basis of the global state are correct.
- Capturing global states is crucial in coordination protocols like distributed transactions, where the consistency of the system state needs to be guaranteed across multiple processes.

4. Performance Monitoring and Optimization:

- Capturing global states allows for performance monitoring by providing insights into the load distribution, resource utilization, and bottlenecks across the distributed system.

This information can be used to optimize resource allocation and system efficiency.

- Understanding the global state can help in dynamically adjusting the distribution of tasks across nodes to ensure even load and prevent some nodes from becoming overwhelmed.

5. Event Ordering and Causal Consistency:

- In distributed systems, understanding the cause-and-effect relationships between events is important. Capturing the global state helps in determining which events are causally related and ensures that these relationships are respected when reasoning about system behavior.
- Global state information is essential for implementing different consistency models in distributed databases and systems where processes operate on shared data.

6. Failure Detection and Recovery:

- By capturing a consistent global state, the system can "rollback" to a previous state if an error or inconsistency is detected, ensuring that the system can continue operating correctly.
- **State Synchronization:** Global state capture helps in synchronizing the system's state after failures or partitions, ensuring that all processes eventually converge to a consistent view of the system.

7. Security and Fault Diagnosis:

- Capturing a global state can assist in detecting security breaches or anomalous behavior, allowing for quick identification of compromised processes.
- It helps in diagnosing faults or abnormal states in the system, leading to faster resolution and preventing widespread system failures.

6.3.2 Challenges in Capturing Global State In Asynchronous Systems

Capturing the Global State in asynchronous distributed systems presents several challenges due to the nature of these systems, where processes operate independently, communicate by message passing, and do not share a global clock. Unlike synchronous systems, where processes can assume certain timing guarantees, asynchronous systems have no such assumptions. This makes it difficult to capture a consistent global state at a single point in time.

The key challenges in capturing global state in asynchronous systems are as follows:

- **Lack of a Global Clock:** In asynchronous systems, there is no global clock or a shared notion of time across processes. As a result, each process maintains its own local clock, and messages between processes can experience arbitrary delays. This lack of synchronized time makes it impossible to capture a snapshot of the entire system at the same physical moment.

  Due to the absence of a global clock, determining the causal relationship between events across different processes is complex. It's difficult to establish a consistent global view where events are properly ordered according to cause and effect.

- **Concurrent Message Passing:** At any given time, there may be messages in transit between processes. These messages have been sent but not yet received, creating ambiguity about whether they should be included in the global state. If a message is in flight, the sender considers it part of its past state, while the receiver may not yet be aware of it.

  Message delivery is unpredictable in asynchronous systems. A message might take a long time to reach its destination or arrive out of order. This variability complicates the process of determining a consistent global state, as messages may arrive after the snapshot has been initiated, introducing inconsistencies.

- **Inconsistent Local States:** Each process in a distributed system maintains its own **local state**, which evolves independently of other processes. Since processes execute concurrently, their local states may not be consistent with each other when captured at arbitrary points in time. Capturing the local state of each process does not automatically yield a consistent global state due to these differences.

In an asynchronous system, there’s no guarantee that the snapshot will be initiated at the same time across all processes. As a result, processes may capture their local states at different points in their execution, leading to inconsistencies when combining the snapshots into a global state.

- **Difficulty in Ensuring Consistency:** To ensure a consistent global state, it is necessary to capture the state of processes and the messages in transit in a way that reflects a coherent and meaningful snapshot of the system. A global state is considered consistent if it reflects a valid "cut" of the system’s execution, where no messages are counted both in the sender’s past state and in the receiver’s future state. However, achieving such consistency is difficult in an asynchronous system.

  An inconsistent cut occurs when part of the snapshot reflects events from the future and another part from the past (e.g., the sender thinks a message has been sent, but the receiver hasn’t received it yet). Handling such inconsistent cuts is challenging and requires special mechanisms like marking channels or delaying snapshot collection until certain conditions are met.

- **State Coordination Among Processes:** Asynchronous systems are typically decentralized, meaning that there is no global controller to coordinate when and how snapshots should be taken. Processes need to coordinate among themselves to agree on capturing a consistent global state, but in an asynchronous environment, coordinating this activity is difficult without introducing further delays or inconsistencies.

  Ensuring that processes capture their states at roughly the same logical point in time without halting the entire system is complex. Processes must be able to continue their normal operations while capturing snapshots, making it harder to coordinate the snapshot process.

- **Overhead and Performance Costs:**Capturing global states, especially in large-scale systems, can incur significant performance overhead. The communication required to propagate control messages (like snapshot initiation markers) and the delays introduced to ensure consistency can reduce system performance. Balancing the frequency and efficiency of snapshot capture with system throughput is a key challenge.

Storing and managing the state of all processes and channels, particularly for large systems with many components, can consume a lot of memory and computational resources. This can limit the feasibility of frequent snapshot captures in resource-constrained environments.

- **Handling Non-Deterministic Events:**In distributed systems, non-deterministic events such as message loss, message duplication, or variable message delays introduce further uncertainty when capturing a global state. Processes might behave differently based on these events, and replicating a consistent global state becomes difficult if such non-deterministic events are not properly accounted for.

  If a global state is captured inconsistently due to non-deterministic behavior, rolling back to such a state after a failure can lead to further inconsistencies or even system corruption.

- **Capturing Channel State:**In asynchronous systems, capturing the state of communication channels (i.e., the messages in transit) is especially challenging. Since messages may be delayed, it is difficult to determine which messages should be considered part of the snapshot. Including or excluding messages in transit may lead to inconsistencies if not handled properly.

  Techniques like **channel marking** (used in the **Chandy-Lamport snapshot algorithm**) are required to capture the state of communication channels. However, the process of marking channels and ensuring that all processes adhere to the snapshot protocol adds complexity to the global state capture process.

- **Dynamic System Changes:**Asynchronous systems are often dynamic, where processes or nodes can join or leave the system, and failures can occur at any time. Capturing a global state in the presence of failures is difficult, as the system topology can change during the snapshot process, leading to incomplete or outdated snapshots.

  If new processes are created or existing processes terminate during the snapshot process, the global state must account for these changes. This dynamic behavior complicates the process of capturing a coherent global state.

6.4 CHANDY-LAMPORT SNAPSHOT ALGORITHM

This algorithm is widely used for capturing consistent global snapshots in asynchronous systems. It solves many of the challenges by introducing control messages (markers) that propagate through the system to help capture both the local states of processes and the state of communication channels.

6.4.1 Assumptions and Prerequisites for Chandy-Lamport Algorithm

The assumptions and prerequisites those ensures the Chandy-Lamport algorithm works effectively in asynchronous distributed systems, by allowing a consistent global snapshot to be captured without halting the system, are as follows:

- The algorithm assumes that the system is asynchronous. Messages sent between processes can have arbitrary delays, but they will eventually be delivered.
- Messages are reliably delivered, but the delivery may be delayed.
- The algorithm assumes that communication channels between processes are reliable and that messages sent between processes will eventually arrive, although the order of arrival may be unpredictable.
- Channels are **FIFO** (First-In-First-Out). This assumption ensures that messages from a process arrive at their destination in the same order they were sent, simplifying the tracking of message states.
- Every communication link (channel) between two processes is unidirectional, meaning each channel allows messages to flow in only one direction. If two processes need to communicate both ways, there are two channels: one for each direction.
- Processes can continue their normal operations while the snapshot is being captured. There is no requirement for processes to pause or synchronize with others during the snapshot collection.

- Each process can send and receive messages as usual during the snapshot algorithm execution.
- The algorithm is typically initiated by a single process known as the initiator. The initiator process triggers the snapshot procedure by sending marker messages to all other processes with which it shares a communication channel.
- Once a process receives a marker message for the first time, it records its local state and begins the snapshot procedure for its outgoing channels.
- The algorithm assumes that no process fails during the execution of the snapshot. All processes are expected to remain functional and participate fully in the snapshot-taking procedure.
- If a process crashes or fails during the snapshot process, the global state capture could be incomplete or inconsistent.
- Each process knows the set of processes with which it directly communicates and the channels connecting them.

6.4.2 Steps Involved in Chandy-Lamport Algorithm

Following is the step-by-step explanation of Chandy-Lamport Algorithm.

1. **Initiation by a Process:**
   - A process (called the initiator) begins the snapshot process.
   - The initiator records its local state (this includes process variables, resources, etc.).
   - The initiator sends a marker message to all processes with which it has outgoing communication channels.
2. **Receiving a Marker Message for the First Time:**
   - When a process receives a marker for the first time from any incoming channel, it:
     - Records its local state immediately.
     - Marks the incoming channel on which the marker was received as empty (no messages are in transit on that channel).
     - Sends a marker message to all its neighbouring processes with outgoing channels.

        - Starts recording the messages arriving on its other incoming channels (those where it has not yet received a marker).

3. **Receiving Subsequent Markers on Other Channels:**
    - When a process receives a marker on an incoming channel after recording its local state, it:
        - Stops recording the state of that incoming channel.
        - The messages received on that channel after the local state was recorded but before the marker arrived are considered part of the channel's state (i.e., in-transit messages).
4. **Propagation of Marker Messages:**
    - After a process records its local state, it propagates marker messages to all its neighbouring processes (i.e., processes with which it shares an outgoing communication channel).
    - The propagation continues until all processes receive marker messages on all their incoming channels.
5. **Completion of Snapshot:**
    - The snapshot is complete for a process when:
        - The process has recorded its local state.
        - It has received a marker on all its incoming channels and recorded the state of each channel.
    - The global snapshot is complete when all processes in the system have completed this process.
6. **Collecting the Global State:**
    - Once the snapshot is complete, the global state consists of:
        - The local state of each process.
        - The state of all communication channels, which includes messages that were in transit when the snapshot was initiated.

Now, let's understand the above steps with the help of an example. Suppose three processes: **P1, P2, and P3**, connected by communication channels. P1 initiates the snapshot.

1. **P1 initiates the snapshot:**
    - P1 records its local state.
    - P1 sends a **marker** message to P2 and P3.

2. **P2 receives the marker from P1:**
   - P2 records its local state.
   - P2 records the channel state from P1 to P2 as empty, as the marker indicates that no new messages are in transit on that channel.
   - P2 sends marker messages to its outgoing neighbours (for simplicity, assume P2 sends a marker back to P1 or to other processes, if connected).
   - P2 continues to record any incoming messages from channels where it hasn't received a marker.
3. **P3 receives the marker from P1:**
   - P3 records its local state.
   - P3 records the channel state from P1 to P3 as empty.
   - P3 sends marker messages to any outgoing channels (for example, it may send a marker to P2).
   - P3 also starts recording incoming messages from other channels, such as messages from P2 to P3, until it receives a marker from P2.
4. **P2 receives the marker from P3:**
   - P2 stops recording the state of the channel from P3 to P2. Any messages that arrived before the marker was received are recorded as part of the channel state.

Once every process has received markers on all its incoming channels and recorded its local state, the snapshot is complete.

6.5 TERMINATION DETECTION IN DISTRIBUTED SYSTEMS

Termination detection in distributed systems refers to the process of determining whether a distributed computation or algorithm has completed its execution, meaning that all processes have finished their tasks and no messages are in transit. In distributed environments, where processes operate independently and communicate through message-passing, it can be challenging to know when the entire system has terminated because there is no centralized control or global clock.

6.5.1 Importance of Termination Detection in Distributed Systems

Termination detection in distributed systems is crucial because it allows the system to determine when a distributed computation has completed, ensuring that all processes have finished their tasks and no messages are in transit.

Below are the key reasons why Termination Detection is important in Distributed Systems.

- **Efficient Resource Utilization:** Termination detection allows for the timely release of resources (e.g., CPU, memory, network bandwidth). Once the computation is complete, system resources can be reallocated or freed, preventing resource wastage and improving system efficiency.
- **Synchronization in Distributed Algorithms:** Many distributed algorithms require a well-defined point of termination before moving to the next stage. Detecting when all tasks are finished ensures that distributed processes remain synchronized, which is crucial for algorithms like distributed consensus, leader election, and distributed sorting.
- **Fault Tolerance and Recovery:** In systems designed for fault tolerance, termination detection helps identify when a computation has finished, allowing the system to enter a recovery or check pointing phase. It prevents premature check pointing or incorrect fault recovery when processes are still active.
- **Task Completion in Distributed Systems:** Distributed computations often involve multiple processes running concurrently across different machines. Without termination detection, it is difficult to determine when all tasks have finished, especially in asynchronous systems where messages may be delayed or lost. Termination detection provides a clear signal that the distributed task has completed.
- **Consistency in Results:** Correct termination detection ensures that no part of the distributed system is prematurely stopped or ignored, leading to consistent and correct results. This is especially important in critical applications like financial transactions, distributed databases, and scientific computations.

- **Coordination for System Maintenance:** In environments such as cloud or cluster computing, it is necessary to know when a distributed task is complete so that maintenance activities like system upgrades, backups, or scaling can be performed without disrupting active computations.
- **Deadlock and Livelock Detection:** Termination detection can also be used as a tool to detect deadlocks or livelocks in distributed systems. If processes remain active indefinitely without making progress, the termination detection mechanism can identify potential issues with the computation or communication patterns.

6.5.2 Challenges in Termination Detection in Distributed Systems

Termination detection in distributed systems is challenging due to asynchrony, lack of global visibility, the presence of in-transit messages, and potential network failures. These factors make it difficult to determine when all processes have completed their tasks and no messages are pending, necessitating sophisticated algorithms to ensure accurate detection and coordination.

The key challenges in Termination Detection in distributed systems are as follows:

- **Asynchrony:** In distributed systems, processes and message passing are typically asynchronous, meaning that there is no global clock to coordinate actions. Processes may execute at different speeds, and messages may be delayed or arrive out of order, making it difficult to determine when all processes have become idle and no messages are in transit.
- **No Global State Visibility:**Sincedistributed systems are decentralized, no single entity has a complete view of the entire system. Each process only knows about its own state and its direct interactions with other processes. This makes it hard to detect when all processes have finished their tasks and when the system as a whole has terminated.
- **Messages in Transit:**Even if all processes are in a passive state (i.e., not performing any tasks), there could still be messages in transit between processes. These in-transit messages can activate a passive process, meaning the system hasn't truly terminated.

Accurately detecting and accounting for in-transit messages is a major challenge.

- **False Termination Detection:**If termination is detected prematurely (e.g., assuming no messages are in transit when some still are), the system may conclude that the computation has finished when in fact, some tasks remain. This can lead to incorrect results or unfinished tasks.
- **Distributed Coordination:**Detecting termination requires coordination across multiple processes. Ensuring that all processes correctly communicate their state without errors or delays, while accounting for network latency and failures, adds complexity to termination detection protocols.
- **Network Failures and Partitions:**Distributed systems are prone to network failures, including message loss, delays, and network partitions (where parts of the system become disconnected). These failures can make it difficult to detect the termination of processes or to ensure that all processes have been accounted for in the detection process.
- **Scalability:**As the number of processes and communication channels in the system increases, the complexity of termination detection also grows. More processes mean more communication and coordination, which increases the likelihood of delays, miscommunication, or failures in detecting termination accurately.
- **Process and Message State Changes:**Processes may switch between active and passive states multiple times during computation, making it challenging to track the exact state of all processes. Additionally, if a process is falsely assumed to be passive, any remaining activity could cause an incorrect termination signal.

6.6 DIJKSTRA-SCHOLTEN ALGORITHM

The Dijkstra-Scholten Algorithm is a method used in distributed systems to check when all processes have finished their work and no messages are being sent between them. This algorithm is important because it helps determine when everything is done without needing a central authority to oversee the process. It ensures that the detection of completion is done efficiently and correctly.

6.6.1 Assumptions and Prerequisites for Dijkstra-Scholten Algorithm

The Dijkstra-Scholten Algorithm relies on certain assumptions and prerequisites for its correct functioning in a distributed system. These conditions are necessary to ensure that the algorithm can effectively detect when the system has terminated.

The assumptions for the algorithm are:

- **Reliable Communication Channels:** The algorithm assumes that the communication between processes is reliable. Messages must be delivered without being lost, duplicated, or corrupted. Each message should eventually reach its destination.
- **Asynchronous System:** The system is asynchronous, meaning processes do not operate in lockstep or have synchronized clocks. Each process can execute at its own pace, and message delivery times can vary, but no process waits for a global clock.
- **Spanning Tree Structure:** A spanning tree exists over the distributed system. This structure is necessary because the algorithm passes tokens (or markers) up the tree to detect termination. The root of the tree (initiator) can then decide when the system has finished.
- **Processes Have Two States:**Each process can be either active (performing some task or waiting for a message) or passive (idle or waiting). A passive process can only become active again upon receiving a message from another process.
- **Finite Number of Messages:**The algorithm assumes that the system will eventually stop sending messages. If messages are continually exchanged without termination, the algorithm won't work as there would always be messages in transit.
- **No Failure during Execution:**The algorithm assumes that no process or communication channel fails during its execution. It does not handle situations where processes crash or the network breaks down.

The prerequisites for the algorithm are:

- **Initiator Process:**A specific process, known as the initiator, must start the termination detection procedure. It is responsible for triggering the detection by passing tokens down the spanning tree to other processes.

- **Tracking of Messages:**Each process must track the messages it sends and receives. This tracking is necessary to ensure that it knows when it has finished sending all of its messages and can safely declare itself passive.
- **Knowledge of Parent and Children in the Tree:**Each process in the system must know its parent and children in the spanning tree structure. This information is vital for propagating the termination signals correctly.
- **Passive Process Sends Termination Signal:**Once a process becomes passive and has no pending messages, it should be capable of sending a signal (marker) to its parent, indicating that it is done.

6.6.2 Steps Involved in Dijkstra-Scholten Algorithm

The algorithm works by organizing processes into a spanning tree and using a token-passing mechanism to signal the completion of work. Below is a step-by-step explanation of how the algorithm works:

1. **Initialization:**
    - One process (known as the **initiator**) begins the termination detection process.
    - The system is represented as a spanning tree, with the initiator as the root.
2. **Process States:**
    - Each process can be in one of two states:
        - **Active:** The process is doing some work or waiting for a message.
        - **Passive:** The process is idle and has no work to do.
3. **Sending a Message (Establish Dependency):**
    - When an active process sends a message to another process, it marks that process as its child (dependent) and itself as the parent.

   - The sender process keeps track of how many messages it has sent to each child.

4. **Becoming Passive:**
   - When a process finishes its work and has no more messages to send, it transitions to a passive state.
   - It does not immediately inform its parent; instead, it waits until it receives acknowledgments from all its children (i.e., all processes it has sent messages to must become passive first).
5. **Receiving a Termination Signal (Marker):**
   - Once a process receives a termination signal from all its children, it knows that all its dependents have finished.
   - The process can now send a termination signal (marker) to its own parent, indicating that it has finished, and all its dependents are passive.
6. **Sending Termination Signals to Parent:**
   - When a passive process has sent termination signals to all its parents and received acknowledgments from all children, it forwards the termination signal to its own parent.
   - This signal propagation continues until it reaches the initiator.
7. **Termination Detection at the Initiator:**
   - The initiator (root process) collects termination signals from all its children.
   - When the initiator has received confirmation from all its children that they are passive, it concludes that the entire system has terminated (i.e., all processes are passive, and no messages are in transit).
8. **Global Termination:**
   - Once the initiator determines that all processes are done, the termination is declared, and the distributed system can move on to other tasks, release resources, or shut down.

Let's understand the workflow with the help of an example:

1. Active Process A sends messages to processes B and C, making them its children.

2. Process A then becomes passive after finishing its work and receiving acknowledgments from B and C.
3. B and C finish their work, become passive, and send termination signals to A.
4. Once A receives termination signals from B and C, it sends its own termination signal to its parent (say, Process D).
5. The termination signal propagates up the tree until it reaches the root (initiator), which then declares that the system has terminated.

6.7 HUANG'S ALGORITHM FOR TERMINATION DETECTION

Huang's Algorithm is a popular method for detecting when all processes in a distributed system have finished their tasks and no messages are still being exchanged. It works by using a system of credit distribution and collection, where credits are passed along with messages. This algorithm is especially useful in asynchronous systems, where processes don't have to run in sync with each other.

The Concepts of Huang's Algorithm are as follows:

- A process (initiator) starts with an initial amount of credit (often set to 1). As it sends messages to other processes, it distributes fractions of its credit along with the messages.
- Each process can be either active (performing computations) or passive (idle, with no pending tasks).
- A process collects credit as it receives messages and completes tasks. When a process becomes passive, it passes its accumulated credit back to the initiator.
- Termination is detected when the initiator regains its original amount of credit, and all processes are in a passive state.

6.7.1 Steps Involved in Huang's Algorithm

Huang’s Algorithm uses a credit distribution and collection system to track the state of processes. When the initiator process regains all of its original credit and all processes are passive, the system can safely declare that termination has occurred.Below is a step-by-step explanation of how the algorithm works:

1. **Initialization:**
   - One process is designated as the initiator and starts the algorithm with 1 unit of credit.
   - All other processes start with zero credit.
2. **Message Sending with Credit:**
   - When an active process sends a message to another process, it divides its current credit and sends a fraction of the credit along with the message.
   - For example, if Process A sends a message to Process B, it might send 0.5 units of credit to B and keep the remaining 0.5 units for itself.
3. **Receiving a Message:**
   - When a process receives a message, it adds the credit that came with the message to its own credit.
   - If the receiving process was passive and now has some credit, it becomes active and begins performing its task.
4. **Becoming Passive:**
   - Once a process finishes its task and has no more messages to send, it becomes passive.
   - A passive process sends any remaining credit back to the initiator or its parent (if a parent-child structure is used).
5. **Credit Collection:**
   - As processes finish and become passive, they return their remaining credit (if any) to the initiator.
   - Credit flows back along the path through which it was originally distributed.
6. **Termination Detection:**
   - The initiator continuously collects credit from passive processes.
   - Termination is detected when the initiator has regained the full 1 unit of credit, and all processes in the system are passive.
   - This signals that all tasks are completed, and there are no messages left in transit.

CHECK YOUR PROGRESS-I

1. State True or False:

a) The FIFO property of communication channels is not required for the Chandy-Lamport algorithm to work.
b) The global state snapshot can be used for rollback recovery in case of system failure.

c) The lack of causal relationships between events makes capturing global state easier in asynchronous systems.

d) Capturing global state is essential for debugging distributed systems.
e) In asynchronous distributed systems, all processes share a common global clock.

2. Fill in the blanks:

a) The Chandy-Lamport Algorithm is used to capture a consistent __________ in asynchronous distributed systems.

b) In distributed systems, the lack of a __________ makes capturing a global state more complex.

c) A process becomes aware of an incoming message in the Chandy-Lamport algorithm after receiving a __________ message.

d) In the Chandy-Lamport algorithm, once a process records its local state, it sends a __________ to all neighbouring processes.

e) The global state of a system includes the state of all processes and __________ between them.

6.8 SUMMING UP

- Global State refers to the collective state of all processes and communication channels at a particular point in time in a distributed system.
- Capturing a consistent global state is complex due to the concurrent and independent execution of processes.

- Capturing global state helps in failure recovery and check pointing.
- Global State helps to detect deadlocks in distributed systems.
- Global State used in algorithms like Chandy-Lamport for capturing system states without halting.
- Global State helps determine causal relationships between events.
- Few of the challenges in Capturing Global State in Asynchronous Systems are:
    - Difficult to capture a snapshot across processes with no synchronized time.
    - Messages in transit create ambiguity in the global state.
    - Achieving a coherent global state that reflects a valid system cut is complex.
    - Coordinating state capture without halting processes is difficult.
    - Difficult to track messages in transit accurately.
    - Changes like process failures make capturing global states challenging etc.
- Chandy-Lamport Algorithm is widely used for capturing consistent global snapshots in asynchronous systems.
- Control messages (markers) help capture local states and communication channel states without stopping the system.
- Termination detection ensures that all processes in a distributed system have completed their tasks and no messages are in transit.
- Importance of Termination Detection are:
    - Frees up system resources once tasks are completed.
    - Allows correct progression to the next phase in algorithms like consensus and leader election.
    - Ensures accurate recovery or check-pointing.
    - Provides a clear signal when distributed computations finish.

    - Prevents premature termination, ensuring accurate results.
    - Helps schedule tasks like upgrades once all work is done.
    - Identifies potential issues like deadlocks or livelocks.
- Dijkstra-Scholten Algorithm is a distributed termination detection algorithm using a spanning tree to track process dependencies.
- Huang's Algorithm uses credit distribution and collection to detect termination.

6.9 ANSWERS TO CHECK YOUR PROGRESS

1.a) False b) True c) False d) True e) False

2.a) global state b) global clock c) marker

d) markere) communication channels

6.10 POSSIBLE QUESTIONS

Short Answer Type Questions:

1. What is the purpose of the Chandy-Lamport algorithm in distributed systems?
2. How do marker messages work in the Chandy-Lamport algorithm?
3. What are the two main components of the global state in a distributed system?
4. Why is capturing a global state in asynchronous systems more challenging than in synchronous systems?
5. What role does a global state play in detecting deadlocks in distributed systems?

Long Answer Type Questions:

6. Explain the significance of termination detection in distributed systems.
7. What are the primary challenges in termination detection in distributed systems?

8. How does the Dijkstra-Scholten algorithm detect termination in a distributed system?
9. What are the key challenges that Huang's algorithm addresses in asynchronous systems, and how does it solve them?
10. Discuss the importance of resource utilization in distributed systems in the context of termination detection.

6.11REFERENCES AND SUGGESTED READINGS

1. "Distributed Systems: Concepts and Design" by George Coulouris
2. “Designing Data-Intensive Applications" by Martin Kleppmann
3. “Distributed Systems: Principles and Paradigms" by Andrew Tanenbaum and Maarten Van Steen

×××

BLOCK- II

UNIT: 1
INTRODUCTION TO MUTUAL EXCLUSION AND PERFORMANCE METRICS

Unit Structure:

1.1 INTRODUCTION

In distributed systems, multiple processes often need to access shared resources like files, data structures, or devices. To avoid conflicts and ensure data consistency, it is important to manage access to these resources. This management is done through mutual exclusion, which makes sure that only one process can use a shared resource at a time. Mutual exclusion is a key part of distributed computing, helping with synchronization and maintaining consistency between processes in a network.

In this unit, we will cover the basics of mutual exclusion, the requirements of mutual exclusion algorithms, and the metrics used to measure their performance. Understanding these topics is crucial for designing distributed systems that can efficiently handle shared resources and maintain stability.

1.2 UNIT OBJECTIVES

After going through this unit, you will be able to:

- *define* mutual exclusion and explain its significance in distributed systems;
- *understand* the requirements that must be met by mutual exclusion algorithms;
- *describe* the various performance measurement metrics used to evaluate mutual exclusion in distributed systems;
- *identify* and explain different classifications of mutual exclusion algorithms.

1.3 MUTUAL EXCLUSION IN DISTRIBUTED SYSTEMS

Mutual exclusion in distributed systems is a way to ensure that only one process can access a shared resource at a time. This is important to prevent issues like inconsistencies or conflicts when multiple processes try to modify the same resource at once. Mutual exclusion guarantees that if a process is in the critical section (the part of the code that accesses the shared resource), no other process can enter this section until the first process is done.

Achieving mutual exclusion in distributed systems is harder than in centralized systems because there is no global clock, there are communication delays, and we have to consider network partitions and failures. Ensuring effective mutual exclusion is vital for keeping data consistent and synchronized in distributed applications like databases, file systems, and network services.

1.3.1 Importance of Mutual Exclusion in Distributed Systems

Mutual exclusion is a fundamental requirement in distributed systems to ensure proper coordination among processes and to maintain data consistency and integrity. The following points highlight the importance of mutual exclusion in distributed systems:

- **Ensuring Access to Shared Resources**: In distributed systems, different processes on various nodes may need to use shared resources like databases, files, or hardware. Without mutual exclusion, these processes might try to access the same resource at the same time, leading to problems like data corruption or loss. Mutual exclusion ensures that only one process can access the shared resource at a time, helping to keep everything consistent.
- **Preventing Race Conditions**: A race condition happens when the outcome of a process depends on the timing or order of other processes' actions. This issue is more common in distributed environments, where processes run in parallel on different nodes. Mutual exclusion helps prevent race conditions by controlling access to critical sections, making sure that operations happen in a predictable order.
- **Maintaining Data Integrity**: Distributed systems often use databases with data replicated across multiple nodes. To keep data integrity during updates, it’s essential to enforce mutual exclusion. If several processes try to change the same data at the same time without proper control, it can lead to conflicting updates and inconsistencies, which are hard to fix in a distributed setting.
- **Avoiding Deadlocks and Starvation**: When multiple processes or nodes coordinate in distributed systems, improper management of shared resource access can cause deadlocks or starvation. Deadlocks happen when processes wait for each other endlessly, while starvation occurs when a process can't access a resource despite trying repeatedly. Mutual exclusion mechanisms help prevent these problems by ensuring fair and controlled resource allocation.
- **Supporting Consistent State of Distributed Applications**: Many distributed applications, like databases or banking systems, require a consistent state across various nodes. Mutual exclusion is vital in making sure that updates are done in order, keeping a consistent overall state. Without mutual exclusion, simultaneous updates could lead to inconsistencies that might disrupt the entire application.
- **Coordination and Synchronization**: In distributed systems, processes often need to work together to ensure correct results.

Mutual exclusion acts as a coordination tool, enforcing a sequence for how processes execute. This is particularly important in situations like leader elections, distributed transactions, and consensus algorithms, where the order and timing of actions are crucial for system stability.

- **Facilitating Fault Tolerance**: Mutual exclusion mechanisms also help make distributed systems more fault-tolerant. By properly managing access to critical sections, the system can handle issues like node failures, lost messages, or network problems more effectively. For example, quorum-based algorithms ensure that even if some nodes fail, the system can still maintain mutual exclusion.

1.4 REQUIREMENTS OF MUTUAL EXCLUSION ALGORITHMS

Mutual exclusion algorithms must satisfy several requirements to ensure the proper functioning of the distributed system. They are:

- **Safety Requirement:**The safety requirement ensures that only one process can be in the critical section at any given time. This is the primary condition of mutual exclusion. If multiple processes access the shared resource simultaneously, data inconsistencies and race conditions can occur, leading to unpredictable and erroneous system behavior.

  The key aspects of the safety requirement are:

  - **Mutual Exclusion Property**: At most one process is allowed to enter the critical section at any point in time.
  - **Resource Consistency**: Safety guarantees the integrity and consistency of shared data, as concurrent access is strictly regulated.
  - **Avoidance of Conflict**: By ensuring that only one process can access a shared resource, the safety requirement prevents conflicts between processes.

- **Liveness Requirement:**The liveness requirement focuses on the ability of the system to make progress. It ensures that every process requesting access to the critical section will eventually be able to enter it, thereby preventing the system from getting stuck in an inactive state.

Liveness requirements are defined by:

  - **No Deadlock**: Deadlock occurs when two or more processes are indefinitely waiting for each other to release resources, leading to a system standstill. Mutual exclusion algorithms must ensure that no group of processes remains permanently blocked.
  - **No Starvation**: Starvation occurs when a process is indefinitely denied access to the critical section while other processes continue to enter and exit it. The liveness requirement ensures fairness, making sure that each process eventually gets its turn to access the resource.
  - **Progress**: The system must guarantee that processes requesting the critical section will be granted access within a finite amount of time, allowing the system to make progress without unnecessary delays.

- **Performance Requirement:**Performance considerations involve evaluating the efficiency of the mutual exclusion algorithm in a distributed system. A well-designed algorithm should minimize communication overhead, reduce delays, and be scalable to handle an increasing number of processes or nodes.

  Performance considerations include:

  - **Message Complexity**: The number of messages exchanged between processes to achieve mutual exclusion is a critical factor. The lower the number of messages, the more efficient the algorithm is. Minimizing message overhead helps to reduce network congestion and enhances system responsiveness.
  - **Synchronization Delay**: This refers to the time required for a process to enter the critical section after the previous process has exited. Lower synchronization delay improves the throughput of the system, allowing more efficient use of resources.
  - **Scalability**: Distributed systems can involve a large number of processes or nodes, and the mutual exclusion algorithm must be scalable to accommodate this. The performance of the algorithm should not degrade significantly as the number of processes grows.

- **Fault Tolerance**: Distributed systems are prone to failures, such as process crashes or network partitions. A mutual exclusion algorithm should handle these failures gracefully to ensure that the system can continue functioning without violating mutual exclusion properties.
- **Load Balancing**: If the algorithm involves centralized control, such as in a coordinator-based approach, the performance can suffer due to the single point of control becoming a bottleneck. Therefore, mutual exclusion algorithms should aim to distribute the load effectively across multiple nodes to avoid such bottlenecks.

1.5 PERFORMANCE MEASUREMENT METRICS IN DISTRIBUTED SYSTEMS

In distributed systems, performance measurement metrics are used to evaluate how efficiently the system functions under different conditions and workloads. These metrics are essential for comparing different system designs, identifying bottlenecks, and optimizing performance. The key performance metrics in distributed systems include response time, throughput, scalability, resource utilization, availability, fault tolerance, and consistency.

1. Response Time

Response time is the time interval between a user request and the system's response to that request. It is one of the most critical performance metrics, particularly for time-sensitive applications. Response time includes:

- **Request Propagation Delay**: Time taken for a request to reach the server or resource.
- **Processing Time**: Time spent by the server in executing the request.
- **Response Propagation Delay**: Time taken for the response to return to the requester.

Minimizing response time is important for providing better user experience, especially in real-time applications like video conferencing or online gaming.

2. Throughput

Throughput refers to the number of requests or tasks processed by the distributed system within a given time frame. It indicates the system's ability to handle workload and is usually expressed as the number of operations per second or transactions per second.

Factors affecting throughput include:

- **Network Bandwidth**: Higher bandwidth allows more data to be transferred, improving throughput.
- **System Bottlenecks**: Bottlenecks like slow processors, limited memory, or overloaded nodes reduce throughput.
- **Task Scheduling Efficiency**: Proper load balancing and efficient task scheduling can enhance system throughput.

3. Scalability

Scalability is the ability of a distributed system to maintain its performance levels when additional resources (such as nodes or servers) are added to handle increased workload. A well-designed distributed system should be able to scale horizontally (by adding more nodes) or vertically (by upgrading the capacity of existing nodes).

Scalability can be assessed through:

- **Horizontal Scaling**: Adding more nodes to the system without significantly affecting performance.
- **Vertical Scaling**: Upgrading hardware components of existing nodes.
- **Elasticity**: The ability of a system to handle workload spikes by dynamically scaling resources up or down.

4. Resource Utilization

Resource utilization measures how effectively the distributed system utilizes its hardware and software resources. It includes CPU usage, memory utilization, disk space, and network bandwidth usage. The goal is to achieve high utilization without overloading the system.

- **CPU Utilization**: Percentage of CPU resources used by processes. Efficient scheduling is essential for optimal CPU utilization.
- **Memory Utilization**: Memory allocation must be managed to avoid underutilization or excessive swapping, which can degrade performance.
- **Network Utilization**: Measures bandwidth usage and is crucial in determining the efficiency of data transfer between nodes.

5. Availability

Availability measures the proportion of time that the distributed system is operational and accessible to users. High availability is crucial for distributed systems that provide critical services, such as online banking or cloud computing.

- **Mean Time to Failure (MTTF)**: The average time between system failures.
- **Mean Time to Repair (MTTR)**: The average time required to repair the system and bring it back online.
- **Redundancy**: Use of redundant nodes and data replication to improve availability.

High availability can be achieved through techniques like replication, failover mechanisms, and redundancy in infrastructure.

6. Fault Tolerance

Fault tolerance is the ability of a distributed system to continue functioning correctly even in the presence of faults or failures. A fault-tolerant system can detect, isolate, and recover from faults without significantly affecting performance.

Fault tolerance is measured by:

- **Reliability**: Probability that a system will function correctly over a specified time.

- **Redundancy**: Adding redundant components to prevent failure.
- **Recovery Time**: Time taken to detect and recover from a fault.

Techniques such as data replication, consensus protocols, and check pointing can be employed to enhance fault tolerance.

7. Consistency

Consistency ensures that all nodes in a distributed system have the same view of data at any given point in time. It is an essential metric for systems that rely on data replication, as inconsistencies can lead to erroneous operations.

Consistency metrics include:

- **Latency of Consistency**: The time taken to propagate changes across replicas to achieve a consistent state.
- **Consistency Level**: Different levels of consistency, such as strong, eventual, or causal consistency, depending on the application's requirements.
- **Staleness**: The difference between the actual state of data and the last-known consistent state, indicating how out-of-date a copy might be.

Consistency must be balanced with availability and partition tolerance, often involving trade-offs governed by the CAP theorem.

8. Latency

Latency is the delay experienced in the communication between nodes in a distributed system. It can be caused by several factors, such as network congestion, physical distance between nodes, and processing overhead. Latency directly affects the responsiveness of the system and is critical for real-time applications.

- **Propagation Delay**: Time taken for data to travel from the sender to the receiver.

- **Queuing Delay**: Time spent waiting in queues at the network interface due to congestion.

- **Processing Delay**: Time taken by a node to process the data packet before forwarding or responding.

Reducing latency often involves optimizing communication protocols, efficient routing, and minimizing the number of hops between nodes.

9. Load Balancing Efficiency

Load balancing refers to distributing the workload evenly across nodes to ensure that no single node becomes a bottleneck. Load balancing efficiency is measured by how well the system distributes tasks and maintains uniform resource utilization.

- **Uniform Task Distribution**: Even distribution of tasks ensures that no node is overloaded.
- **Node Utilization Metrics**: Measuring the utilization of individual nodes can help identify imbalances in workload distribution.
- **Task Migration Cost**: The cost of migrating tasks between nodes for balancing purposes, which should be minimized for efficient load balancing.

10. Network Bandwidth Utilization

Network bandwidth utilization measures how effectively the available bandwidth is used by the distributed system. It is important for applications with heavy data transfers, such as video streaming or data analytics.

- **Effective Utilization**: The proportion of the total available bandwidth that is effectively used for productive communication.
- **Congestion Control**: Mechanisms to avoid overloading the network, which can degrade performance.
- **Minimizing Overhead**: Efficient communication protocols to reduce message overhead and improve bandwidth usage.

1.6 CLASSIFICATION OF MUTUAL EXCLUSION ALGORITHMS

Mutual exclusion algorithms are fundamental in distributed systems to ensure that shared resources are accessed in a mutually exclusive manner, preventing race conditions and data inconsistencies. Various algorithms have been proposed for achieving mutual exclusion, and they can be broadly classified into several categories based on different factors, such as the type of communication, the coordination strategy, or whether they require a central coordinator. There are two main classifications of mutual exclusion algorithms and they are - **Centralized Algorithms and Distributed/Decentralized Algorithms.**

1.6.1 Centralized Algorithms

Centralized algorithms are one of the primary classifications of mutual exclusion algorithms in distributed systems. In a centralized approach, a designated coordinator (a single central node or server) is responsible for managing access to the shared resource. This coordinator acts as an authority that decides which process can enter the critical section, ensuring that only one process has access to the resource at a time. Now, let's discuss how a centralized algorithm works:

- In a centralized mutual exclusion algorithm, all processes that need access to a critical section must send a request to the coordinator.
- The coordinator maintains a queue of incoming requests for the critical section.
- When a process sends a request, the coordinator decides if the critical section is available. If it is available, the coordinator grants permission to the requesting process. Otherwise, the request is queued until the critical section is free.
- Once the process finishes its execution in the critical section, it sends a release message to the coordinator, allowing the next process in the queue to be granted access.

Advantages of Centralized Algorithms:

Simplicity: Centralized algorithms are relatively simple to implement because only one node needs to handle all the coordination.

Guaranteed Mutual Exclusion: The use of a central coordinator ensures that only one process can access the critical section at a time, making mutual exclusion straightforward.

Low Message Overhead: Compared to distributed algorithms, centralized algorithms usually require fewer messages to manage access. A request, a grant, and a release message are all that is needed for each access, resulting in a total of 3 messages per critical section entry.

Disadvantages of Centralized Algorithms:

Single Point of Failure: The coordinator is a single point of failure. If the coordinator crashes, no process can enter the critical section until a new coordinator is chosen, this can disrupt the system.

Scalability Issues: As the number of processes grows, the coordinator can become a bottleneck, leading to longer waiting times for processes and degraded performance. This approach is not suitable for systems that require high scalability.

Coordinator Overload: All requests are handled by the central coordinator, which can lead to overload and reduced efficiency if the number of requests is very high.

1.6.2 Distributed/Decentralized Algorithms

Decentralized algorithms are a type of mutual exclusion algorithms used in distributed systems. Unlike centralized algorithms, where one node manages access to shared resources, decentralized algorithms spread this responsibility across multiple nodes. This approach helps prevent problems like having a single point of failure or a bottleneck, which are common issues with centralized methods. The decentralized algorithms are of two types and they are namely – **Token-Based Algorithms** and **Non Token-Based Algorithms** (or **Permission-Based Algorithms**). The two types of the decentralized algorithms will be discussed in the next Unit.

CHECK YOUR PROGRESS-I

1. State True or False:

a) Mutual exclusion in distributed systems ensures that only one process can access a shared resource at a time.

b) In a centralized mutual exclusion algorithm, a single coordinator manages access to shared resources.

c) Liveness requirements ensure that a process requesting access to the critical section may remain blocked indefinitely.

2. Fill in the blanks:

a) The __________ requirement focuses on the ability of the system to make progress.

b) The main communication method used in permission-based algorithms is __________ passing.

c) A race condition occurs when the outcome of a process depends on the __________ of other processes' actions.

d) __________ algorithms are designed to ensure that every process requesting access will eventually be able to enter the critical section.

e) In distributed systems, __________ measures the proportion of time that the system is operational and accessible to users.

1.7 SUMMING UP

- Mutual Exclusion ensures that only one process accesses a shared resource at a time to prevent inconsistencies and conflicts.
- The challenges for Mutual Exclusion in Distributed Systems include the lack of a global clock, communication delays, and network failures.
- Importance of Mutual Exclusion:
    - Prevents data corruption by ensuring exclusive access,
    - Controls access to prevent outcomes dependent on timing,

    - Essential for ensuring data consistency during updates,
    - Ensures fair and controlled resource allocation,
    - Ensures ordered updates for a consistent system state,
    - Helps in coordination for leader elections, transactions, etc.,
    - Improves system resilience in case of failures.
- Performance Measurement Metrics in Distributed Systems are: Response Time, Throughput, Scalability, Resource Utilization, Availability, Fault Tolerance, Consistency, Latency, Load Balancing Efficiency andNetwork Bandwidth Utilization.
- In Centralized Algorithms, single coordinator manages access to shared resources.
- Simplicity, guaranteed mutual exclusion, low message overhead are the advantages of Centralized Algorithms.

1.8 ANSWERS TO CHECK YOUR PROGRESS

1.a) True b) True c) False

2.a) liveness b) message c) timing or order

d) Liveness e) availability

1.9 POSSIBLE QUESTIONS

Short Answer Type Questions:

1. What is mutual exclusion in distributed systems?

2. What is the main function of a coordinator in centralized mutual exclusion algorithms?

3. What does scalability mean in the context of mutual exclusion algorithms?

Long Answer Type Questions:

4. Discuss the importance of mutual exclusion in distributed systems.

5. Explain the safety and liveness requirements of mutual exclusion algorithms.

6. Compare centralized and decentralized mutual exclusion algorithms.

7. How does mutual exclusion contribute to maintaining data integrity in distributed systems?

1.10 REFERENCES AND SUGGESTED READINGS

1. "Distributed Systems: Concepts and Design" by George Coulouris
2. “Designing Data-Intensive Applications" by Martin Kleppmann
3. “Distributed Systems: Principles and Paradigms" by Andrew Tanenbaum and Maarten Van Steen

×××

UNIT: 2

TOKEN-BASED AND NON-TOKEN-BASED MUTUAL EXCLUSION ALGORITHMS

Unit Structure:

2.1 INTRODUCTION

In distributed systems, Mutual Exclusion is a key part of distributed computing, helping with synchronization and maintaining consistency between processes in a network. As discussed in the previous unit, the algorithms for mutual exclusion are classified under two categories namely - Centralized Algorithms and Distributed/Decentralized Algorithms.In this unit, we will cover the Distributed/Decentralized Algorithms in detail.

2.2 UNIT OBJECTIVES

After going through this unit, you will be able to:

- *understand* token-based decentralized algorithms.
- *understandnon* token-based decentralized algorithms.

2.3 TOKEN-BASED DECENTRALIZED MUTUAL EXCLUSION ALGORITHMS

A classification of decentralized mutual exclusion algorithms in distributed systems where a special token is used to manage access to shared resources. The token acts as a unique key, and only the process holding the token is allowed to enter the critical section and access the shared resource. This approach ensures mutual exclusion since there is only one token, preventing multiple processes from accessing the resource simultaneously.

2.3.1 How Token-Based Algorithms Work?

In this approach, all the processes are arranged in a logical ring, and the token is passed around the ring in one direction. Each process knows the identity of the next process to which it must send the token. If a process needs to enter the critical section, it waits for the token to arrive, and once done, it passes the token to the next process. If the process does not need the critical section, it simply forwards the token. This method is straightforward, but if the token is lost or a process fails, additional mechanisms are required for token.

2.3.2 Advantages of Token-Based Algorithms

Following are the advantages of token-based mutual exclusion algorithms:

- **Guaranteed Mutual Exclusion**: Since only the process holding the token can access the critical section, mutual exclusion is guaranteed.

- **Fairness**: Token-based algorithms ensure fairness as the token is passed in a predefined sequence, giving all processes an equal chance to access the critical section.
- **Avoidance of Deadlock**: These algorithms avoid deadlocks because only one process holds the token at any time, and others simply wait for it to be passed.

2.3.3 Disadvantages of Token-Based Algorithms

Following are the disadvantages of token-based mutual exclusion algorithms:

- **Loss of Token**: If the token is lost or a process holding the token fails, the entire system might be blocked. Extra mechanisms are required to regenerate the lost token.
- **Token Circulation Delay**: In large systems, the time taken for the token to circulate among all processes can lead to delays, especially if only a few processes require the critical section.
- **Fault Tolerance**: Token-based systems are susceptible to node failures. If a node fails while holding the token, other nodes cannot access the critical section until the token is regenerated.

2.4 NON TOKEN-BASED DECENTRALIZED MUTUAL EXCLUSION ALGORITHMS

Non-token-based or permission-based algorithms are a type of decentralized classification mutual exclusion algorithm used in distributed systems where access to a critical section is managed without using a token. Instead, these algorithms use message exchanges between processes to control access to shared resources.

2.4.1 How Non Token-Based Algorithms Work?

The core idea is to get permission from other processes before entering the critical section, ensuring that no two processes can access it simultaneously. This coordination is done through request and reply messages, often using logical clocks, timestamps, or voting mechanisms. While these algorithms solve the challenges related to token management, they come with their own issues, such as higher communication overhead, vulnerability to deadlocks, and reliance on all processes responding properly. Choosing a permission-based algorithm depends on factors like scalability, fault tolerance, and overall system efficiency.

The main characteristics of Non Token-Based/ Permission-Based Algorithms are:

- **Message Passing**: Processes request permission from other processes and wait for replies before entering the critical section.
- **Logical Clocks**: Requests are often times-tamped to determine the order of access, helping to prevent conflicts.
- **Decentralization**: There is no central coordinator or token; instead, multiple nodes are involved in granting permissions.

2.4.2 Advantages of Non Token-Based Algorithms

Following are the advantages of non token-based mutual exclusion algorithms:

- There is no single point of failure since permission is obtained from multiple nodes.
- Most permission-based algorithms use timestamps to ensure fair access to the critical section, preventing starvation.
- Since no token is used, problems like token loss, duplication, or token regeneration are avoided.

2.4.3 Disadvantages of Non Token-Based Algorithms

Following are the disadvantages of non token-based mutual exclusion algorithms:

- In systems with a large number of processes, the number of messages required to obtain permission can become significant.
- The algorithms are sensitive to process failures, as a process may need replies from all other processes (or quorum members). A failure may lead to indefinite waiting.
- Some permission-based algorithms can suffer from deadlock if multiple processes are waiting for each other’s replies, or

starvation if some processes are unable to get permission due to repeated denials.

CHECK YOUR PROGRESS-I

1. State True or False:

a)In token-based algorithms, processes request permission from other processes to enter the critical section.

b)Token-based algorithms guarantee mutual exclusion because only one token exists in the system.

c) Token-based algorithms do not require message passing between processes.

d)One disadvantage of non-token-based algorithms is that they are insensitive to process failures.

2. Fill in the blanks:

a)In token-based decentralized mutual exclusion algorithms, a special ______ is used to manage access to shared resources.

b) Token-based algorithms avoid ______ because only one process can hold the token at any time.

c)Non-token-based algorithms often rely on ______ clocks or timestamps to determine the order of access.

d)In token-based systems, if the ______ is lost, mechanisms are required to regenerate it.

2.5 SUMMING UP

- In token-based algorithms, a unique token is used to grant access to shared resources, ensuring mutual exclusion as only the token-holder can access the critical section.
- In token-based algorithms, Processes are arranged in a logical ring.

- Disadvantages of token-based algorithms are:
    - Lost tokens block the system; regeneration mechanisms are needed.
    - Inefficiency in large systems when token circulates unnecessarily.
    - Node failures disrupt token passing, requiring recovery protocols.
- In non token-based algorithms, access to the critical section is managed through permission-based mechanisms, avoiding token dependency.
- Characteristics of non token-based algorithms are:
    - Message Passing: Processes exchange request and reply messages.
    - Logical Clocks: Timestamps ensure proper access order.
    - Decentralization: No single point of control.
- Disadvantages of non token-based algorithms are:
    - Requires many messages, especially in large systems.
    - Node failures or missing replies can cause indefinite waiting.
    - Deadlocks can occur due to waiting cycles; starvation may happen with repeated denials.

2.6 ANSWERS TO CHECK YOUR PROGRESS

1.a) False b) True c) False d) False

2.a) token b) deadlocks c) logicald) token

2.7 POSSIBLE QUESTIONS

Short Answer Type Questions:

1. What is Token-Based Decentralized Mutual Exclusion Algorithm?
2. What is token in Non Token-Based Decentralized Mutual Exclusion Algorithm?
3. How do decentralized algorithms avoid single points of failure?

Long Answer Type Questions:

4. What are the advantages and disadvantages of Token-Based Decentralized Mutual Exclusion Algorithms in Distributed Systems?
5. What are the advantages and disadvantages of Non Token-Based Decentralized Mutual Exclusion Algorithms in Distributed Systems?
6. Discuss the characteristics of Non Token-Based Decentralized Mutual Exclusion Algorithms.

2.8 REFERENCES AND SUGGESTED READINGS

1. "Distributed Systems: Concepts and Design" by George Coulouris
2. "Designing Data-Intensive Applications" by Martin Kleppmann
3. "Distributed Systems: Principles and Paradigms" by Andrew Tanenbaum and Maarten Van Steen

×××

UNIT: 3
ELECTION ALGORITHMS

Unit Structure:

3.1 INTRODUCTION

Distributed algorithms are specialized algorithms designed to operate in a distributed system, where a collection of independent computers, each with its own memory, work collaboratively without sharing memory. Communication between these computers occurs over a network, where processes running on different machines exchange information to achieve a common goal. In many distributed algorithms, the presence of a coordinator is essential, as it takes on specific roles like managing resources or coordinating the tasks of other processes. Election algorithms play a key role in this context, as they are used to select such a coordinator or leader to manage tasks centrally.

Election algorithms are vital in distributed systems to determine which process among several should assume the role of a leader or coordinator. These algorithms aim to elect a process that can take on centralized responsibilities, such as managing access to shared resources or making important decisions for the entire system.

3.2 UNIT OBJECTIVES

After going through this unit, you will be able to:

- *understand* the purpose and importance of election algorithms in distributed systems.
- *describe* how the Bully Algorithm and Ring Algorithm operate to elect a coordinator.
- *compare* the Bully and Ring Algorithms in terms of efficiency, complexity, and use cases.
- *identify* the advantages and limitations of different election algorithms.
- *explain* the role of election algorithms in ensuring the reliability and coordination of distributed systems.

3.3 ELECTION ALGORITHM AND ITS ROLE

Election algorithms play a crucial role in distributed systems by providing a mechanism to elect a coordinator or leader among distributed processes. In a distributed system, multiple processes operate independently, and the absence of shared memory necessitates coordinated decision-making to avoid conflicts and maintain consistency. A coordinator often needs to be selected to centralize decision-making, assign responsibilities, or facilitate efficient task execution. Election algorithms are used to determine which process should act as the coordinator when the system starts or when the existing coordinator fails.

The importance of election algorithms lies in their ability to dynamically manage the system in cases where processes may fail and recover unpredictably. The election process ensures that there is always a unique, active coordinator, which is crucial for maintaining control over shared resources and ensuring system reliability. Two popular election algorithms are the Bully Algorithm and the Ring Algorithm, each of which addresses the leader selection problem in different configurations of distributed systems.

Role of Election Algorithms:

Election algorithms play an important role in distributed systems by ensuring that all independent processes work together smoothly and reliably. In a distributed system, different processes run on separate nodes, each with its own memory, and they need to coordinate to achieve common goals. Since there is no shared memory, and nodes can fail unexpectedly, a central point of control is needed for decision-making and efficient management of tasks. Election algorithms solve this problem by selecting a leader or coordinator among the processes. This coordinator manages resources, maintains consistency, and helps the system recover from failures. By electing a leader, election algorithms make it easier to coordinate actions, reduce conflicts, and keep the system running smoothly, even when problems arise.

Leader Selection isthe primary roles of election algorithms is to select a leader or coordinator among the processes. The coordinator is responsible for managing specific tasks, such as resource allocation, synchronization, or decision-making. The leader is chosen based on specific criteria, such as process priority or unique identifier, ensuring that only one process is responsible for coordinating activities at any given time. By designating a leader, election algorithms prevent conflicts, as processes no longer need to compete for control over shared tasks.

Assumptions for Election Algorithms:

The assumptions are:

Assumption 1: Each process in the system has a unique identifier, such as a network address (assuming one process per machine for simplicity). Generally, election algorithms aim to identify the process with the highest identifier and designate it as the coordinator.

Assumption 2: Every process is aware of the identifiers of all other processes in the system.

Assumption 3: Processes do not have information about which other processes are currently active (up) or inactive (down).

Examples of election algorithms include the **Bully Algorithm** and the **Ring Algorithm**, each with unique approaches to selecting a leader.

3.4 BULLY ALGORITHM

The Bully Algorithm is a popular election algorithm in distributed systems, designed to select a coordinator among multiple processes. Its primary objective is to choose the process with the highest unique identifier (ID) as the coordinator, ensuring that there is always a designated leader to manage coordination tasks effectively. This algorithm is particularly beneficial in environments where processes may fail and recover, as it helps maintain stability by always electing a new leader when needed. The Bully Algorithm assumes reliable communication between processes and requires that each process can directly communicate with others. When a process detects that the current coordinator has failed, it initiates an election, thereby ensuring that the system continues to operate efficiently.The algorithm is initiated when a process detects that the current coordinator has failed or is not responding. The process that initiates the election is called the “initiator”.

STEPS INVOLVED IN BULLY ALGORITHM:

1. **Election Initiation:**
   - When a process detects that the current coordinator has failed, it starts an election process by sending “election” messages to all processes with higher IDs.
   - If no process responds, the initiator assumes itself to be the new coordinator and broadcasts a “coordinator” message to all other processes.
2. **Response to Election:**
   - When a process with a higher ID receives the “election” message, it responds with an “OK” message to indicate that it is alive and capable of becoming the coordinator.
   - The higher-ID process then starts its own election process by sending “election” messages to processes with even higher IDs.
3. **Coordination Announcement:**
   - Eventually, the process with the highest ID among those participating in the election will not receive any response to its “election” messages. This process declares itself the new

coordinator by sending a "coordinator" message to all other processes.

- All other processes update their records to recognize the new coordinator.

4. **Failure Handling:**
   - If the new coordinator fails, the election process is restarted by the next process that detects the failure.

Advantages of Bully Algorithm:

- The Bully Algorithm ensures that the process with the highest ID always becomes the coordinator, providing a clear and deterministic outcome.
- It is simple to implement and works well in systems where processes have unique identifiers and can directly communicate with each other.

Disadvantages of Bully Algorithm:

- The algorithm generates a significant amount of message traffic, especially if multiple processes initiate elections simultaneously.
- It can be inefficient in large systems, as the number of messages grows rapidly with the number of processes.
- The failure of multiple processes during the election process can lead to delays in selecting a new coordinator.

3.5 RING ALGORITHM

The Ring Algorithm is an election algorithm used in distributed systems to select a coordinator among multiple processes arranged in a logical ring. The main goal of this algorithm is to ensure that a single process is elected as the coordinator to manage coordination tasks and shared resources. In this algorithm, the processes are organized in a ring structure, where each process has a direct communication link with its successor in the ring. The algorithm ensures that the process with the highest unique identifier (ID) becomes the coordinator.

In addition to the other assumptions mentioned earlier, the Ring Algorithm assumes that the processes are arranged in a specific order, either physically or logically, so that each process knows who comes next in the sequence.The Ring Algorithm is well-suited for systems where processes are naturally organized in a circular structure and communication overhead must be minimized by using only local interactions.

STEPS INVOLVED IN RING ALGORITHM:

1. **Logical Ring Formation:**

All processes in the system are arranged in a logical ring. Each process knows the identity of the process that comes right after it, allowing communication to flow around the ring.

2. **Election Initiation:**

When a process, say P, notices that the current coordinator has failed, it starts an election. It sends an election message with its own ID to the next process in the ring.

3. **Message Passing:**

The election message moves around the ring. Each process that receives the message compares its ID with the one in the message:

- If its ID is higher, it replaces the ID in the message with its own and sends it to the next process.
- If its ID is lower, it simply passes the message along without changing it.

4. **Election Completion:**

Eventually, the election message returns to the process that started it, carrying the highest ID found among all the processes in the ring. The process with this highest ID becomes the new coordinator.

5. **Coordinator Announcement:**

Once the new coordinator is chosen, a message is sent around the ring to let all processes know who the new coordinator is.

Advantages of Ring Algorithm:

- The Ring Algorithm is simple to implement and requires only local communication between neighbours in the ring.
- It guarantees that a coordinator is eventually elected, even if multiple processes initiate elections simultaneously.

Disadvantages of Ring Algorithm:

- The algorithm can be slow, especially in large systems, since messages must traverse the entire ring.
- If the ring is disrupted (e.g., due to a process or communication link failure), the election process can be delayed.

3.6 BULLYVS RING ALGORITHM

Let's try to analyze the Bully and Ring algorithm in terms of different aspects.

Basic Approach:

- Bully algorithm selects the process with the highest ID as the coordinator through direct communication among processes.
- Ring Algorithm organizes processes in a logical ring, with each process passing an election message around the ring to determine the coordinator.

Assumptions:

- In Bully Algorithm it is assumed that each process knows the ID of every other process and can communicate directly with them.
- In Ring Algorithm it is assumed that processes are arranged in a logical ring and each process knows its immediate successor.

Communication Complexity:

- In Bully Algorithm, the complexity is High. Involves direct communication between all active processes, resulting in $O(n^2)$ messages in the worst case (for n processes).
- In Ring Algorithm, the complexity is Moderate. Only involves communication along the ring, resulting in O(n) messages for an election.

Coordinator Selection:

- In Bully Algorithm, the process with the highest ID is elected as the coordinator.
- In Ring Algorithm, the process with the highest ID among those that receive the election message is elected as the coordinator.

Initiation of Election:

- In Bully Algorithm, any process can initiate an election when it detects that the coordinator has failed.
- In Ring Algorithm, any process can initiate an election when it detects a failure. The message travels along the ring to find a new coordinator.

Failure Handling:

- In Bully Algorithm, when a process detects a coordinator failure, it starts an election by sending messages to all higher-numbered processes.
- In Ring Algorithm, when a coordinator failure is detected, an election message is passed around the ring to find the new coordinator.

Scalability:

- In Bully Algorithm, Scalability is limited due to high communication costs for larger networks.
- In Ring Algorithm, Scalability is better since each process only communicates with its immediate successor.

Overall, he Bully Algorithm works well for small systems where processes can easily communicate directly with each other, but it becomes less efficient in larger systems because of the high number of messages involved. On the other hand, the Ring Algorithm is more efficient in terms of the number of messages and can handle larger systems better, but it may take longer to elect a new coordinator since the message has to pass through every process in the ring.

CHECK YOUR PROGRESS-I

1. State True or False:

a) Election algorithms select a coordinator among distributed processes.

b) Election algorithms are only used when the system starts, not during coordinator failure.

c) The Bully Algorithm requires each process to know the ID of every other process.

d) In the Ring Algorithm, processes are organized in a logical ring.

e) The Ring Algorithm involves communication along the ring to determine the new coordinator.

2. Fill in the blanks:

a) In the Ring Algorithm, each process knows the identity of its immediate _______.

b) The Bully Algorithm has a high communication complexity of ______ in the worst case.

c) The process that initiates an election in the Bully Algorithm is called the _______.

d) In the Ring Algorithm, the process with the ______ ID in the ring becomes the new coordinator.

e) The ______ Algorithm is more suitable for large systems due to better scalability.

3.7 SUMMING UP

- Election Algorithms provide a mechanism to elect a coordinator or leader among distributed processes and Ensure coordinated decision-making in the absence of shared memory.
- The Role of election algorithm is select a leader to manage tasks, allocate resources, and handle failures.
- The assumptions for election algorithms are as follows:
    - Each process has a unique identifier,

- Every process knows the identifiers of other processes,
- Processes do not know which others are active or inactive.

- **Bully Algorithm**
  - Chooses the process with the highest unique identifier as coordinator.
  - Works in environments where processes may fail and recover.
  - Initiates election when a process detects coordinator failure.
- **Ring Algorithm**
  - Arranges processes in a logical ring; each knows its successor.
  - Elects a coordinator through message passing along the ring.

3.8 ANSWERS TO CHECK YOUR PROGRESS

1. a) True b) False c) True d) Truee) True

2. a) successor b) O(n^2) c) initiatord) higheste) Ring

3.9 POSSIBLE QUESTIONS

Short Answer Type Questions:

1. What is the primary purpose of election algorithms in distributed systems?
2. How does the Ring Algorithm arrange processes?
3. What happens when a coordinator fails in the Bully Algorithm?
4. What type of communication does the Ring Algorithm rely on?
5. How does the Bully Algorithm determine the new coordinator?

Long Answer Type Questions:

6. Explain the importance of election algorithms in distributed systems.
7. Describe the steps involved in the Bully Algorithm.
8. Discuss the advantages and disadvantages of the Ring Algorithm.

9. Compare the Bully Algorithm and the Ring Algorithm in terms of scalability and efficiency.
10. Describe the assumptions made by the Bully and Ring Algorithms.

3.10 REFERENCES AND SUGGESTED READINGS

1. "Distributed Systems: Concepts and Design" by George Coulouris
2. “Designing Data-Intensive Applications" by Martin Kleppmann
3. “Distributed Systems: Principles and Paradigms" by Andrew Tanenbaum and Maarten Van Steen

×××

UNIT: 4

DISTRIBUTED SCHEDULING AND LOAD DISTRIBUTION

Unit Structure:

4.1 INTRODUCTION

Distributed Scheduling involves deciding which node in a distributed system should execute a particular task, with the aim of achieving optimal system performance, load balancing, and minimizing response time. It requires making decisions about job assignments based on factors such as resource availability, task priority, and current system load.

Load Distribution focuses on dividing the workload evenly across all nodes in the system to prevent some nodes from being overloaded while others are underutilized. It involves transferring tasks from heavily loaded nodes to lightly loaded ones, which can improve overall system performance and ensure fairness in resource usage.

4.2 UNIT OBJECTIVES

After going through this unit, you will be able to:

- understand the concept of distributed scheduling and its significance in distributed systems.
- identify the challenges and issues involved in distributed load distribution.
- describe the components of load distribution algorithms.
- explain various task migration strategies used in load distribution.
- analyze the factors that affect the stability of distributed scheduling.

4.3 DISTRIBUTED SCHEDULING

Distributed Scheduling means organizing and managing tasks across multiple independent computers in a distributed system. In a distributed environment, tasks are spread across different computers (nodes) to make the best use of resources, improve system performance, and increase fault tolerance. The main goal of distributed scheduling is to decide which tasks should be handled by which nodes, considering factors like load balancing, communication delays, and available resources.

In distributed systems, each node works independently, but they cooperate to complete shared tasks. Scheduling in these systems not only involves assigning tasks to nodes but also considering the network structure, communication between nodes, and the changing state of nodes (e.g., availability and load levels).

Let's discuss the importance of Distributed Scheduling. Below are some key reasons why distributed scheduling is important:

- Distributed scheduling makes sure that all nodes in the system are used effectively by balancing the workload among them. This prevents some nodes from being idle while others are overloaded, leading to better resource utilization.
- By distributing tasks intelligently, distributed scheduling reduces the processing time, leading to faster execution of tasks and improved performance. It helps ensure that no single node

becomes a bottleneck, allowing the system to handle larger workloads more efficiently.

- Distributed scheduling is essential for scaling distributed systems. As the number of nodes or tasks grows, scheduling helps distribute the increased workload without compromising system performance, making it easier to expand the system.
- In distributed systems, nodes can fail unpredictably. Distributed scheduling provides a mechanism to reassign tasks from failed nodes to active ones, thus maintaining system reliability and ensuring that the system can continue to function smoothly even if some nodes fail.
- Distributed systems are often dynamic, with nodes joining or leaving the network and workloads fluctuating over time. Distributed scheduling helps adapt to these changes in real time, ensuring that tasks are reassigned efficiently based on the current state of the system.
- Effective distributed scheduling ensures that the workload is evenly distributed across all nodes, preventing any single node from being overwhelmed. This leads to improved system stability and prevents performance degradation due to overloading.

Also, Distributed scheduling faces several challenges, including:

- Heterogeneity: Nodes may have different processing power and available resources. Scheduling needs to adjust according to the capabilities of each node.
- Communication Overhead: Assigning and moving tasks between nodes requires communication over a network, which can cause delays and increase overhead.
- Dynamic Nature: Distributed systems are always changing, with nodes joining or leaving the network and their load levels varying. Scheduling algorithms must adapt to these changes in real time.
- Load Balancing: Ensuring tasks are evenly distributed among nodes is essential to prevent any one node from becoming overloaded and to maximize efficiency. In the next section, we will discuss it in detail.

Distributed scheduling algorithms can be broadly categorized into three types:

1. **Static Scheduling:** The tasks are assigned to nodes before execution begins, based on predefined criteria. This approach is simpler but less flexible in adapting to dynamic changes in the system.
2. **Dynamic Scheduling:** The assignment of tasks is done during runtime, allowing the system to adapt to changes in workload and resource availability.
3. **Hybrid Scheduling:** Combines features of both static and dynamic scheduling to achieve a balance between simplicity and adaptability.

4.4 LOAD DISTRIBUTION

Load Distribution in distributed scheduling is about assigning tasks or workloads to different nodes (computers) in a distributed system. The main goal of load distribution is to use resources effectively, improve system performance, and make sure no single node gets too much work while others are underused. The key concepts related to Load Distribution are:

1. **Load Balancing:** This means distributing the workload among nodes so that each one gets a fair share of tasks, based on its ability to handle them. Load balancing prevents any one node from being overloaded, which keeps the system working smoothly.
2. **Load Monitoring:** To distribute tasks effectively, the system needs to monitor how busy each node is. This includes checking things like CPU usage, memory usage, and the number of tasks each node is handling. Based on this information, tasks can be assigned to keep the workload balanced.
3. **Task Assignment:**
   - Centralized Approach: One central scheduler decides which tasks go to which nodes based on their current load. This is easy to implement but can become a problem if the scheduler fails.
   - Distributed Approach: Each node decides for itself whether to take on new tasks or pass tasks to others. This prevents a

single point of failure but requires good communication between nodes to avoid conflicts.

4. **Task Migration:** If one node gets too much work, tasks can be moved to nodes with less work. This helps keep the workload evenly distributed.

Now, let's discuss the why of load distribution is important in distributed systems. Following are the importance of load distribution:

- Load distribution ensures that all nodes are being used, so no node is idle while others are overloaded.
- By distributing tasks evenly, they get done faster, reducing the total time and improving system responsiveness.
- Load distribution helps the system grow more easily as new nodes are added, allowing it to handle more work without slowing down.
- If a node fails, its tasks can be reassigned to other nodes. This backup ensures the system keeps running smoothly even if there are failures.
- Without good load distribution, some nodes might become bottlenecks and slow everything down. Load distribution prevents this by spreading tasks evenly.

4.5 LOAD DISTRIBUTION ALGORITHMS

Load distribution algorithms are methods used in distributed systems to assign tasks or workloads to different computers or servers. The main aim of these algorithms is to make sure that all computers are used effectively, so that no single machine gets overloaded while others are not fully used. This helps improve how well the system performs, makes better use of resources, increases reliability, and allows the system to grow.

In distributed systems, tasks can be assigned either dynamically (adjusting in real-time) or statically (based on fixed rules), depending on factors like the processing power of each computer, network conditions, current workloads, and the need to complete tasks efficiently.

4.5.1 Challenges of Load Distribution Algorithms

Load distribution algorithms in distributed systems face several challenges that can affect their efficiency and performance. Some of the key challenges include:

Heterogeneity of Nodes: Distributed systems often consist of nodes with different hardware configurations, processing power, memory, and network capabilities. Designing a load distribution algorithm that takes these differences into account can be difficult, as tasks may need to be allocated in a way that suits each node's capacity.

Dynamic Workload and Resource Availability: The workload in distributed systems is not constant; it can fluctuate over time as tasks are added or completed. Similarly, nodes may join or leave the system, and their resource availability can change (e.g., due to failures or resource contention). Load distribution algorithms must adapt in real time to these changes to prevent system overloads or underutilization.

Communication Overhead: Effective load distribution often requires frequent communication between nodes to monitor system states and transfer tasks. This communication introduces network overhead, which can reduce system performance, especially in large distributed systems.

Task Migration Complexity: Moving tasks from one node to another (task migration) is crucial for dynamic load balancing, but it can be complex. The task's state, including data, memory, and dependencies, must be transferred without errors. Migration also incurs overhead in terms of time and network bandwidth, which can impact the overall system's efficiency if not handled properly.

4.5.2 Types of Load Distribution Algorithms

Load distribution algorithms can be broadly classified into two main types: **Static load distribution**, **Dynamic load distribution**and **Adaptive load distribution**. Each type has its own set of strategies and characteristics, which are outlined below.

STATIC LOAD DISTRIBUTION ALGORITHMS

In static load distribution, tasks are assigned to nodes based on pre-defined rules or policies that do not change during runtime. The allocation is determined before the system starts executing tasks, and it remains fixed regardless of any changes in the system's state, such as node failures or load variations. The key characteristics of this type of algorithms are:

- Tasks are assigned to nodes based on fixed parameters like node capacity, number of tasks, or historical performance.
- Since task assignments are made in advance, there is no need for continuous monitoring or real-time adjustments, leading to lower communication overhead.
- Static algorithms are not responsive to dynamic changes, such as node failures or fluctuating workloads, which can result in poor load balancing if the system's state changes.

DYNAMIC LOAD DISTRIBUTION ALGORITHMS

In dynamic load distribution, task assignment is made in real-time based on the current state of the system. The algorithm continuously monitors the status of nodes (e.g., CPU utilization, memory availability, network load) and adjusts the distribution of tasks accordingly. The key characteristics of this type of algorithms are:

- Decisions about which node should handle a task are made during runtime, based on current system conditions.
- Dynamic algorithms can adapt to changes in the system, such as varying workloads, node failures, or resource availability.
- Dynamic load distribution requires continuous monitoring and communication between nodes, which can increase system overhead.

ADAPTIVE LOAD DISTRIBUTION ALGORITHMS

Adaptive Load Distribution Algorithms in distributed systems are dynamic strategies that adjust to changing system conditions in real-time to distribute the workload efficiently among multiple nodes. These algorithms continuously monitor system resources and workload distribution and adapt to fluctuations in load, node availability, or network performance. The goal is to achieve optimal system performance, prevent bottlenecks, and ensure the fair utilization of resources across the network. The key characteristics of this type of algorithms are:

- Adaptive algorithms respond to changes in system conditions such as node failures, varying workloads, and new nodes joining the network. These algorithms make real-time decisions on how to assign or migrate tasks to maintain a balanced load.

- The system regularly monitors each node's resource utilization (e.g., CPU, memory, task queue length) to determine if nodes are overloaded or underutilized. Based on this information, the algorithm can redistribute tasks to balance the load across all nodes.
- These algorithms make decisions on-the-fly, meaning they analyze system conditions and adjust task allocations in real-time to maintain efficiency. As workloads change dynamically, so does the task allocation strategy.

There are different types of Adaptive load distribution algorithms. They are:

Sender-Initiated Adaptive Algorithms: In this approach, nodes that are heavily loaded initiate the process of distributing tasks. When a node's workload exceeds a certain threshold, it looks for underloaded nodes to transfer tasks to.

Receiver-Initiated Adaptive Algorithms: n this strategy, underloaded nodes initiate the process. They request tasks from overloaded nodes or the central system to balance their workload.

Symmetric Adaptive Algorithms: This is a combination of sender-initiated and receiver-initiated approaches. Both heavily loaded and underutilized nodes can initiate load balancing by communicating with each other.

Centralized Adaptive Algorithms: A central node or controller monitors the system and decides how to distribute the workload among the nodes. It dynamically adjusts allocations based on real-time system conditions.

Decentralized Adaptive Algorithms:In a decentralized approach, individual nodes collaborate and make decisions locally about workload distribution. Nodes exchange information with neighboring nodes and adjust their tasks accordingly.

4.6 TASK MIGRATION

Task migration is a process in distributed systems where a task (or a process) that is already running on one node is transferred to another node. This transfer occurs to balance the system load, improve performance, or handle system changes like node failures or resource unavailability.

The main goal of task migration is to ensure that workloads are distributed more efficiently across the nodes in a distributed system. By moving tasks from overloaded nodes to underloaded ones, the system can avoid performance bottlenecks, reduce response times, and optimize resource utilization.Below are the key concepts that form the foundation of task migration:

Load Balancing: The main reason for task migration is to balance the load. This means making sure that no computer or node is overwhelmed with too many tasks while others are underused. By moving tasks from overloaded nodes to less busy ones, the system works more efficiently and performs better. The goal is to:

- Prevent any node from becoming a bottleneck.
- Improve the speed at which tasks are completed.
- Ensure tasks are shared fairly among all nodes based on their abilities.

Resource Optimization: In distributed systems, different nodes have different capacities, like CPU power and memory. Task migration helps make the best use of resources by moving tasks to the nodes that have more available power. This prevents powerful nodes from being idle while weaker ones struggle with a heavy load, ensuring that resources are used as efficiently as possible.

Fault Tolerance: Task migration helps keep the system running smoothly, even when something goes wrong. If a node fails or its performance drops, the tasks on that node can be moved to other active nodes. This ensures that tasks keep running without interruption, helping the system recover quickly from problems.

Dynamic Adaptation: Distributed systems are constantly changing—nodes may join or leave, and workloads may shift. Task migration allows the system to adapt to these changes by regularly monitoring the state of the system and moving tasks as needed. It adjusts tasks in real time based on the availability and workload of each node to keep the system balanced.

Task State Transfer: When a task is moved to another node, its current state (like memory usage, open files, and active connections) must also be transferred. The task should be able to continue on the new node without losing progress. This can be complicated, especially if the task is connected to other processes, but it’s important to transfer all relevant data correctly and efficiently.

Network Overhead: Moving tasks between nodes requires communication over the network, which can slow things down. This extra communication is called network overhead. The goal of task migration is to keep this overhead as low as possible so that the benefits of moving the task outweigh the cost of transferring it.

Consistency: Maintaining consistency during task migration is crucial. If a task interacts with other processes or services, the system needs to make sure no errors or inconsistencies occur due to the migration. Consistency management ensures that everything runs smoothly and all interactions are handled correctly, so the task can continue without issues after the move.

TYPES OF TASK MIGRATION IN DISTRIBUTED SYSTEMS

There are different types of task migration strategies, each designed to address specific challenges and scenarios in distributed systems. These types can be categorized based on the timing of migration, the approach taken, and the triggering conditions.

Static Task Migration:

Static task migration refers to a fixed, predefined migration strategy that does not consider the real-time state of the system. Once tasks are assigned to nodes, they do not change, regardless of variations in load or node status. The key characteristics of this kind are:

- Static migration is easy to implement but lacks flexibility, as it does not adapt to changes in the system’s state.
- This approach works well in environments with predictable workloads where tasks do not need to be reallocated dynamically.

Dynamic Task Migration:

In contrast to static task migration, dynamic task migration continuously monitors the system's state and redistributes tasks based on real-time conditions such as node load, resource availability, or network conditions. Tasks can be migrated dynamically as needed. The key characteristics of this kind are:

- Dynamic migration responds to changing conditions in the system, making it more flexible and efficient.

- The system must constantly monitor resource usage and task status, which adds complexity.
- This is the most suitable approach for highly dynamic distributed systems where workloads and resources change frequently.

Preemptive Task Migration:

Preemptive migration involves transferring a task from one node to another while it is still in progress. This type of migration is more complex because the task's current state, including variables, memory, and other resources, must be preserved during the transfer to ensure that it can resume execution seamlessly on the new node. The key characteristics of this kind are:

- The system must transfer the entire state of the task, including data and intermediate results.
- Preemptive migration incurs higher overhead due to the complexity of transferring the task’s state across nodes.
- Preemptive migration is useful in dynamic systems where workloads change frequently and tasks need to be moved in real-time to prevent overloads on certain nodes.

Non-Preemptive Task Migration:

Non-preemptive migration involves moving a task only after it has completed execution or when it is in a waiting state (e.g., when waiting for input/output operations to finish). Unlike preemptive migration, there is no need to transfer the task's state during execution, making it simpler and less resource-intensive. The key characteristics of this kind are:

- The task is either finished or paused, so there is no need to transfer its current state.

- Since no active state needs to be transferred, non-preemptive migration incurs less overhead.
- Non-preemptive migration is suitable for systems where tasks have relatively short execution times or can be deferred to migrate at natural points in the workflow.

Incremental Task Migration:

In Incremental Task Migration only the incremental changes or updates in a task's state are transferred from one node to another, rather than migrating the entire task's state at once. This approach is used to reduce the overhead and downtime typically associated with migrating a task, allowing the system to maintain high performance and minimize service interruptions. The key characteristics of this kind are:

- Only the changes or deltas in the task's state (e.g., modified memory, variables, or file handles) are transferred instead of migrating the entire task state. This helps in reducing the amount of data that needs to be sent across the network.
- Task state is transferred in small increments over time, which reduces network load and minimizes the performance impact on the system. This prevents bottlenecks and ensures a smoother migration process.
- Distributed systems often experience changes in workload or resource availability. Incremental migration allows tasks to move between nodes efficiently while adapting to these changes in real time.

CHECK YOUR PROGRESS-I

1. State True or False:

a) Distributed Scheduling aims to balance the workload evenly across nodes.

b) Static scheduling can easily adapt to dynamic changes in the system.

c) Dynamic load distribution assigns tasks based on the real-time state of the system.

d) Task migration involves moving completed tasks from one node to another.

e) Heterogeneity of nodes is a challenge in load distribution algorithms.

2. Fill in the blanks:

a) The goal of ____________________ is to decide which tasks should be handled by which nodes.

b) ____________________ ensures that each node gets a fair share of tasks based on its capabilities.

c) ____________________allows the system to adapt to changes in workload and resource availability in real-time.

d) ________________________ transfers only the changes or updates in a task's state.

e) ____________________ task migration occurs after a task has finished or is in a waiting state.

4.7 SUMMING UP

- Distributed scheduling decides which node in a distributed system should execute tasks for load balancing, system performance, and minimized response time.
- The factors related to distributed scheduling are resource availability, task priority, system load, communication delays, and node states.

- Types of distributed scheduling are namely – Static, Dynamic and Hybrid.
- Importance of Distributed Scheduling are:
    - Ensures efficient resource utilization by preventing node overloading.
    - Improves task execution time and system scalability.
    - Increases fault tolerance by reassigning tasks from failed nodes.
    - Adapts to system changes (e.g., node availability, workload fluctuations).
- Load Distribution is to distribute workload evenly across nodes to prevent overloading and underutilization.
- Load Balancing snsures fair task distribution based on each node's capacity.
- Load Monitoring tracks node resource usage (CPU, memory) to guide task assignment.
- Task Migration transfers tasks between nodes to balance load, optimize resources, and improve fault tolerance.
- Types of Task Migration are namely – Static, Dynamic, Preemptive, Non-preemptive and Incremental.
- Challenges related to Load Distribution Algorithms are:
    - Heterogeneity: Nodes with different capabilities require tailored task assignments.
    - Dynamic Workloads: System load fluctuates, requiring real-time adaptation.
    - Communication Overhead: Task migration and monitoring introduce network load.
    - Task Migration Complexity: Transferring task state without disrupting execution.

4.8 ANSWERS TO CHECK YOUR PROGRESS

1. a) True b) False c) True d) Falsee) True

2. a) distributed scheduling b) Load balancing c) Dynamic schedulingd) Incremental task migratione) Non-preemptive

4.9 POSSIBLE QUESTIONS

Short Answer Type Questions:

1. What is the primary goal of distributed scheduling?

2. What is load balancing in distributed systems?

3. Why is task migration important in distributed systems?

4. What is a centralized task assignment approach?

5. What is heterogeneity in distributed systems?

Long Answer Type Questions:

6. Explain the key components of load distribution in distributed systems.

7. Discuss the challenges of load distribution algorithms in distributed systems.

8. Describe different types of task migration in distributed systems.

9. Discuss the significance of adaptive load distribution algorithms in distributed systems.

10. What are the challenges of task migration in distributed systems, and how can they be addressed?

4.10 REFERENCES AND SUGGESTED READINGS

1. "Distributed Systems: Concepts and Design" by George Coulouris
2. "Designing Data-Intensive Applications" by Martin Kleppmann
3. "Distributed Systems: Principles and Paradigms" by Andrew Tanenbaum and Maarten Van Steen

×××

UNIT: 5

DEADLOCKS IN DISTRIBUTED SYSTEMS

Unit Structure:

5.1 INTRODUCTION

In one of the earlier unit, we have learnt about the use of locks to prevent race conditions, maintain data integrity and implement an efficient concurrency control mechanism for accessing shared resources in case of distributed systems. So, use of lock is very significant in distributed systems. But deadlock may occur due to the use of locks in distributed systems. Deadlock can be defined as a state where a set of transactions is unable to complete as each transaction is waiting for a resource which is currently allocated to another transaction available in that set. For example, consider A and B are two transactions and R and S are two resources. Currently R is allocated to A and A is waiting for S. On the other hand S is allocated to B and B is waiting for R. At this point, A is waiting for B to release a lock on S and B is waiting for A to release a lock on

R. In such a situation, both transactions will be in waiting mode indefinitely and this type of state is called a deadlock. In distributed system, due to deadlock, the system may become unresponsive. In this chapter we will learn about different conditions to occur deadlock and different ways to handle it.

5.2 OBJECTIVES

After going through this chapter, we will be able to learn:

- About the basic conditions to occur deadlocks in distributed systems.
- How to prevent deadlock?
- How to avoid deadlock?
- About Deadlock Detection and Resolution.
- About different issues in Deadlock Detection and Resolution.
- About Deadlock Detection Algorithms.
- What is Communication deadlock?
- About the differences between Resource deadlock and Communication deadlock.

5.3 BASIC CONDITIONS OF DEADLOCKS

We have already learnt that deadlock is a situationin which a set of transactions are not able to complete or proceed as they are indefinitely waiting for resources.Now there are four basic conditions that must be presentall together so that deadlocks may occur. These conditions are presented as follows.

- **Mutual exclusion condition: In** Mutual exclusion condition, each resource can be allocated by only one process or transaction at a time. For example, if a database table is being accessed by a transaction, T, then no other transaction is allowed to access it until the transaction, T releases it.

- **Hold and wait condition:**In the Hold and Wait condition, if a transaction is already accessing a resource then it can request for more resources to complete its job. For example, if a transaction is currently accessing a database table and it requires the access of another database table to complete its job then it can request for that table.
- **No-Preemption condition:** In the No-Preemption condition, if a transaction is accessing a resource then the resource cannot be released forcibly from that transaction before completion of its job with that resource. For example, if a transaction is accessing a database table then the table cannot be forcibly released from the transaction for any other transaction that is waiting for that resource. Only the transaction can release the table allocated to it.
- **Circular Wait condition:**In the Circular Wait condition, a circular chain is formed among two or more transactionsin such a way that each transaction is waiting for a resource that is allocated to the next transaction in the chain. For example, let us consider that transaction T1 is waiting for a resource that is allocated to the transaction T2,transaction T2 is waiting for a resource that is allocated to the transaction T3 and finally transaction T3 is waiting for a resource that is allocated to the transaction T1. As a result, a circular chain of transactions(T1→T2→T3→T1) is created.

5.4 DEADLOCK HANDLING

In general, deadlocks can be handled by using three approaches that areDeadlock Prevention, Deadlock Avoidance, Deadlock Detection and Resolution.

5.4.1 Deadlock Prevention

Earlier, we have learned about four basic conditions that must be fulfilled all together for a deadlock occurrence in a system. So, deadlocks can be prevented by providing a mechanism to stop from satisfying at least any one of these conditions. But it is not possible to stop from satisfying Mutual Exclusion condition and No-Preemption condition in case of all kind of resources. For example, we have already learnt in earlier chapters that Mutual Exclusion condition can be prevented by allowing read operation on a file by multiple processes simultaneously but in case of write operation, it cannot be allowed. Similarly, if a process is performing write operation on a file then it cannot be taken away forcibly from the process so that another waiting process can perform write operation on that file. Now, possible strategies to stop from satisfying, Hold and Wait Condition andCircular Wait Condition, are discussed in the following points.

- The first possible strategy to break Hold and Wait condition is to allow all the processes or transactions to lock all the required resourcesbefore beginning of their executions.This locking process must be performed as one atomic step. If a transaction cannot lock all its required resources as one or more of them are currently locked by other transactions then the transaction will not lock any resources and it will be in waiting mode until the availability of all the required resources.As a result, each transaction will complete its job without waiting for any resource and deadlock will never occur in the system.But two issues are observed in this strategy as presented in the following points.
    - ➢ In case of some transactions, it is impossible to determine all the resources required for their job before beginning of their execution.
    - ➢ In this approach, efficient utilization of shared resources cannot be achieved.
- The second possible strategy to break Hold and Wait condition is related to the temporary release of resources.In this strategy, if a transaction requires more resources which are currently allocated to the other transactions then the transaction must temporarily unlock all the resources currently allocated to it before requesting other required resources.After that, it will attempt to allocate all the necessary resources together.

- The first possible strategy to break Circular Wait condition is to apply a restriction on all the transactions or processes such that each transaction or process can allocate only one resource at a time and if it require another resource then it must release the allocated one.But practically this approach cannot be used for all types of transactions because some transactions may require more than one resource together to complete their jobs.
- A better approach than the first one is available to break the Circular Wait condition. In this approach, a global ordering is assigned to the all resources available in a distributed system. Now, if a transaction is currently accessing a resource then it can request only for those resources which are placed after the already allocated resource in the global order of resources.If the transaction require a resource which is placed beforethe already allocated resource in that global order then at first it must release the already allocated one.Due to this restriction, Circular Wait condition will never be satisfied in any situation.So, in this approach, transactions can request for resources depending upon a global numerical order. For example, let us consider process P1 holds resource T and process P2 holds resource U.The global order of T is n1 and the global order of U is n2.Now if n1 is greater than n2 then the process P1 cannot be able to request for the resource U but process P2 can request for the resource T. On the other hand if n1 is smaller than n2 then the process P1 can request for the resource U but the process P2 cannot request for the resource T. In both situations, Circular Wait condition will never be satisfied.Issues with this approach are presented in the following points.
    - ➢ The major issue with this approach is to implement an efficient global ordering for resources that will work efficiently in every condition.
    - ➢ Due to this approach, resource utilization may not be efficient as concurrency will be reduced.

5.4.2 Deadlock Avoidance

We have already learnt how deadlocks can be prevented in the earlier section and observed some issues in that approach.In this section, we are going to learn how deadlocks can be avoided without preventing the four basic conditions.Deadlock avoidance is based on two factors that are careful resource allocation and maintaining a safe state in the system. In this context,a safe state in a system means that the system can allocate available resources to different processes in some order as per their requirements without causing a deadlock. So, deadlocks can be avoided if a careful resource allocation strategy is applied so that the system remains in safe state after each resource allocation process. It means, thesystem will allow a resource request of a process only if it will not cause a deadlock situation.On the other hand, if there is a possibility of a deadlock occurrence in the system after a sequence of resource allocations to different processes then it means that the system is in unsafe state.

Now, to implement Deadlock avoidance approach, the system must have four important information that are: (a) information about presently available resources, (b) information about the resources that are already locked by each process, (c) information about the resources thatwill be required by different processes in future, and (d) information about the resources that will be released by the processes in future.Theseinformation will be updated after each movement of the system from one safe state to another safe state.In case of a distributed system, each server must maintain these information and communicate with each other to maintain a safe state in the whole system by carefully allocating resources to different processes.

In 1965, Edsger Dijkstraproposed a scheduling algorithm to implement Deadlock avoidance. This algorithm is referred as Banker's algorithm because it is based on the idea that a banker might apply to grant funds to a set of customers.The main concept of this algorithm is to find out the state of a system if it will allow a resource request of a process. If the state is a safe state then the system may allow the resource request. Otherwise, the system will delay the request. In case of distributed systems, the Banker's algorithm for multiple resources is applied to avoid deadlock.

The Banker's algorithm for multiple resources: In case of distributed systems, multiple types of resources are required by different processes from different servers to complete their jobs.Now to apply Banker's algorithm, the number of resources of each type must be fixed and the maximum number of resources required by each process must be stated in advance.Let us consider, the current number of processes is A and the number of resource types available in the system is B.The execution of the Banker's algorithm requires three data structures that are presented in the following points.

- **Resource_Available:** Resource_Available is a data structure that stores the number of available resources of each type in the system.So, let us consider,Resource_Available[t] stores the number of available resources of resource type $R_{t.}$'Resource_Available[t] = N' means that the number of available R_t type resources in the system is N. The maximum value of t can be B as per our assumption.In Figure 11.1(a), an example of Resource_Available is presented.
- **Resource_Allocated:** Resource_Allocated is a [A × B] matrix that stores the number of already allocated resources of each type to each process in the system.So, 'Resource_Allocated[s][t] = M' meansthat M number of resources of resource type,R_t is currently allocated to the process,$P_{s.}$In Figure 11.1(b), an example of Resource_Allocated is presented.
- **Resource_Required:** Resource_Required is also a [A × B] matrixthat stores the number of resources of each type that is still required by each process to complete its job. So, 'Resource_Required[s][t] = M' means that M number of resources of resource type,R_t is still required by the process,$P_{s.}$In Figure 11.1(c), an example of Resource_Required is presented.

	R_1	R_2	R_3	R_4
	1	2	3	4
Resource_Available	3	4	3	2

Figure 11.1(a) Resource_Available

		R_1	R_2	R_3	R_4
		1	2	3	4
P_1	1	0	2	1	0
P_2	2	1	1	0	1
P_3	3	0	2	1	1
P_4	4	2	0	1	2
P_5	5	2	1	3	0

Figure 11.1(b) Matrix Resource_Allocated

		R_1	R_2	R_3	R_4
		1	2	3	4
P_1	1	3	0	1	0
P_2	2	3	2	1	0
P_3	3	2	0	1	2

P_4	4	3	1	2	1
P_5	5	2	3	1	1

Figure 11.1 (c) Matrix Resource_Required

Now, the Banker's algorithm to avoid deadlockscan be explained with the following points.

1. When a process try to request for a resource then before allowing it, the system must check the possible state of the system if the process request is allowed. If the state is found to be safe then only the process request is allowed. In Banker's algorithm, a search operation is performed to find a row, I, in the Resource_Requiredmatrix (figure 11.1 (c))whereResource_Required[I][J] <= Resource_Available[J] for all values of J. If the search operation could not finda row with that condition then it means that the system is in unsafe state. So, due to a process request, if the state of the system will become unsafe then the system delays the process request. Otherwise, the system will allow the process request to allocate a required resource and update related information in Resource_Available,Resource_Allocated and Resource_Required.
   For example, let us consider, the process, P_1 try to request for a resource of type R_1. Now, if this request is allowed then information in Resource_Required related to P_1 and information inResource_Available related to R_1 will be changed as shown in figure 11.2(a) and 11.2(b).

	R_1	R_2	R_3	R_4
	1	2	3	4
Resource_Available	2	4	3	2

Figure 11.2(a): Updated Resource_Available If Request of P_1 is Allowed

		R_1	R_2	R_3	R_4
		1	2	3	4
P_1	1	**2**	0	1	0
P_2	2	3	2	1	0
P_3	3	2	0	1	2
P_4	4	3	1	2	1
P_5	5	2	3	1	1

Figure 11.2(b): Updated Resource_Required If Request of P_1 is Allowed

From figure 11.2(a) and 11.2(b), following statement can be stated.

- 'Resource_Required[1][J] <= Resource_Available[J]' is true for all values of J(1 to 4).

From the above statement, it means that the system will be in safe state if request of the process, P_1 for a resource of type, R_1 is allowed by the system. So, at this point, a resource of type R_1 is allocated to the process, P_1 and the value in Resource_Allocated[1][1] is updated to 1. Now, let us consider, the process, P_2 try to request for a resource of type R_1. If this request is allowed then information in Resource_Available related to $R_1$and information inResource_Required related to P_2will be changed as shown in figure 11.3(a) and 11.3(b)

	R_1	R_2	R_3	R_4
	1	2	3	4
Resource_Available	**1**	4	3	2

Figure 11.3(a): Updated Resource_Available If Request of P_2 is Allowed

		R_1	R_2	R_3	R_4
		1	2	3	4
P_1	1	2	0	1	0
P_2	2	**2**	2	1	0
P_3	3	2	0	1	2
P_4	4	3	1	2	1
P_5	5	2	3	1	1

Figure 11.3(b): Updated Resource_Required If Request of P_2 is Allowed

From figure 11.3(a) and 11.3(b), following statement can be stated.

- Resource_Required[1][J] <= Resource_Available[J] is false for J = 1
- Resource_Required[2][J] <= Resource_Available[J] is false for J = 1
- Resource_Required[3][J] <= Resource_Available[J] is false for J = 1
- Resource_Required[4][J] <= Resource_Available[J] is false for J = 1
- Resource_Required[5][J] <= Resource_Available[J] is false for J = 1

From the above statements, it means that the system will be in unsafe state if request of the process, P_2 for a resource of type, R_1 is allowed by the system. So, the system will delay the request of the process, P_2.

2. If a process is able to allocate all the required resources and complete its job then the process is marked as finished and

all its resources are declared as available resources by updating corresponding information in Resource_Available.
3. Above steps are repeated until all processes are marked finished.

Issues associated with Deadlock avoidance approach are stated in the following points.

- To apply Deadlock avoidance approach, the number of resources must be fixed in the system and the system must have the information about the resource requirements for each processin advance. In a large distributed system, it may not be possible to fulfill these two requirements of this approach.
- Another issue of Deadlock avoidance approach is that the execution of processes may not fulfill the synchronization requirements to improve system performance.
- Due to this approach, resource utilization may be decreased as resource requests from processes may be delayed to maintain safe state in the system although required resources are available for allocation.
- In case of a large system with a huge number of processes and resources, this approach may increases computational overhead considerably in the system to avoid deadlocks.
- Due to this approach, it may be possible that resource requests of some processes are continuously delayed to maintain safe state of the system which may degrade the system performance.

5.4.3 Deadlock Detection and Resolution

From the earlier sections, we can observe that deadlock occurrences can be stopped by using Deadlock Prevention and Deadlock Avoidance approaches. In this section, we are going to discuss about Deadlock Detection and Resolution technique to remove a deadlock after its occurrence in a distributed system. In this approach, at first occurrence of a deadlock is detected and after detection of the

deadlock, it is removed from the system by using deadlock resolution strategies.
Detection of deadlocks can be performed by maintaining a wait-for graph. The wait-for graph is a directed graph that represents which processis waiting for which processes to release their one or more resources.Each node in a wait-for graph representsa process or a transaction. Let us consider, P and Q are two nodes available in the wait-for graph. Now, if there is a directed edge from P to Q then it means that process P is waiting for some resource that is currently allocated to the process Q. If a cycle of processes is formed in the wait-for graph then it indicates the occurrence of a deadlock.For example, from the wait-for graph presented in thefigure (11.4), it is observed that a cycle of processes is formed (P→R→S→Q→P) and it indicates a possible occurrence of a deadlock. So, detection of deadlocks can be performed by searching for cycles in the wait-for graph.

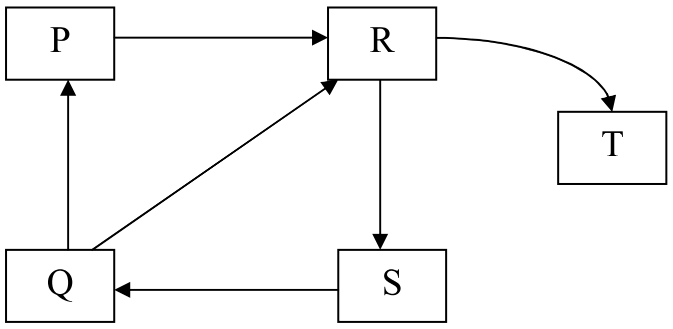


Figure 11.4 : Wait-for Graph

The lock manager of a system contains the software to detect deadlocks. This software must be able to access a representation of the wait-for graph so that it can detect deadlocks by searching the graph for cycles. On the other hand, the wait-for graph is continuously updated whenever new edges are added to it or old edges are removed from it depending upon ‘setLock’ and ‘unLock’ operations of the lock manager.

In a distributed system, several servers are accessed by multiple processes or transactions. As a result, multiple wait-for graphs are available in a distributed system. At each server, the lock manager constructs a local wait-for graph. So, to perform deadlock detection in a distributed system, a global wait-for graph may be created from the local wait-for graphs associated with the servers of the system. Detection of a distributed deadlock can be performed by searching for cycles in the global wait-for graph. In this process, communication among the servers available in the distributed system is necessary. In the later sub-section, distributed deadlock detection algorithms will be discussed.

After detection of a deadlock, a resolution strategy is applied to remove the deadlock situation from the system.Different deadlock detection and resolution strategies will be discussed in the next chapter.

5.4.3.1 Issues in Deadlock Detection and Resolution

Before implementing Deadlock detection and resolution strategy to solve deadlock problem in a distributed system, we have to consider different issues related to this approach. These issues are presented in the following points.

- In distributed systems, deadlock detection approaches may require a large amount of transmissions of information between servers. The cost of these transmissionsmay be very high and it may degrade the performance of the system.
- It is hard to maintain a consistent global wait-for graph in a distributed system due to different reasons like server failures, information losses, network delays etc. As a result, deadlock detections may be inconsistent.
- In distributed systems, detection of deadlock may require considerable time as it may require large amount of information and analysis from multiple servers. As a result, deadlocks will be active for longer time periods in a system which will affect the performance of the system.
- The complexity of deadlock detection and resolution is enhanced when the distributed system becomes greater in size.
- Sometimes one or more servers in a distributed system may become inactive or temporarily not reachable.Deadlock detection and resolution approach must have the ability to handle such situations so that consistent deadlock detections can be maintained in the system.

- After deadlock detection, it is a difficult to choose appropriate resolution strategy to break the deadlock condition from the system.
- In case of distributed systems, resources may be dynamically allocated and released. In such a dynamic environment, it is a very complex process in real-time to constantly track and update different modifications related to dynamic resource allocations and releases so that the deadlock detections can be performed consistently by the system.

5.5 RESOURCE DEADLOCKS VERSUS COMMUNICATION DEADLOCKS

We have already discussed about deadlocks that are occurred when in a group of transactions, each transaction is holding resources and waiting for some other resources which are currently acquired by other transactions. This type of deadlocks is referred as resource deadlocks in distributed systems.We have already discussed in the earlier section about the four basic conditions that must be present all together to occur a Resource deadlock.

Now in this section, we are going to discuss about Communication deadlock in distributed systems.Communication deadlocks occur in a situation where two or more than two processes are waiting for communication through messages from each other.Communication deadlock is particularly connected to inter-process communication.Let us consider a situation to understand Communication deadlock. Let P1, P2 and P3 are three processes in a distributed system. P1 sends a message to P2 and then it is waiting for a message from P2. At the same time, P2 sends a message to P3 and then it is waiting for a message from P3. Similarly, P3 sends a message to P1 and it is waiting for a message from P1. At this point, a circular wait condition is occurred where each process is waiting for a message from the next process in this cycle of three processes.As a result, deadlock occurs and it is termed as Communication deadlock. So, two conditions must be present together so that Communication deadlocks may occur and these are presented as follows.

- **Waiting forsend and receive operation**: In this condition, processes are in block state until messages are sent or received before progressing. So, due to this condition, a process may be waiting for a message from another process indefinitely as the other process is also waiting for a message indefinitely.

- **Circular wait condition**: In this condition, a set of processes forms a cycle where each process is waiting for a message from the next process in that cycle.

The differences between Resource deadlock andCommunicationdeadlock are presented in the following points:

- Communication deadlock is connected to inter-process communication but Resource deadlock is associated withlogical or physical resources of a distributed system.
- Mutual Exclusion, Hold and Wait, No-Preemption and Circular Wait are the four basic conditionsthat must be satisfied all together to take place a Resource deadlock. On the other hand, Waiting for send and receive operation and Circular Wait are the two conditions that must be satisfied so that a Communication deadlock may occur.

5.5.1 Communication Deadlock Handling

Communication deadlocks can be handled in two ways that are:

- Deadlock Detection and Recovery
- Deadlock Prevention

Deadlock Detection and Recovery: In this approach, at first, Communication deadlock is detected and then recovery procedure is applied to remove the deadlock condition. Detection of Communication deadlock is a difficult job in distributed system. In general, it can be performed by using the following techniques.

- A Time-out Mechanism can be implemented which can generate deadlock detection alert if a process is waiting for a message from a long duration of time.
- Dependency Graphs can be maintained to monitor which process is waiting for messages from which other processes.So, if a cycle of processes is formed in this graph then it means that a Communication deadlock is occurred.

- Processes can send Heartbeat Messages at regular intervals to specify that the processes are in active mode. So, if Heartbeat Message is not received from a process from particular duration of time then it means that the process is not in active mode due to deadlock or failure.

After successful detection of a Communication deadlock, recovery from that deadlock can be performed by using the following approaches.

- One or more than one processes associated with the deadlock can be terminated and then restarted again to recover from the deadlock situation. In this approach, most important part is to find out which process or processes should be terminated and then restarted first. This decision is dependent on different conditions. For example, priority value of the processes and number of messages sent or received by them can be considered to decide which process or processes should be terminated and then restarted first.
- Processes involved in a Communication deadlock can be rollback to their previous safe states so that they can continue to perform their jobs without having a deadlock situation. In this approach, states of the processes must be saved at regular intervals.

Deadlock Prevention: Communication deadlocks can also be prevented by using following approaches.

- If processes are not in block state indefinitely while sending or receiving messages from other processes then Communication deadlock can be prevented. It means to prevent Communication deadlock, processes can use non-blocking send and receive operations and check for messages from other processes at regular intervals.
- If a process is waiting for a message from another process and it do not received it within a particular duration of time then the process can send the request for message again to the other process or it can rollback to a previous safe state from where it can continue again. As a result, a possible Communication deadlock can be prevented.
- Message ordering protocols can be implemented so that a definite order to send and receive messages by the processes can be maintained in such a way that no circular wait condition can be formed. So, if no circular wait condition is formed then no Communication deadlock will be occurred.

5.6 SUMMING UP

- Deadlock can be defined as a state where a set of transactions is unable to complete as each transaction is waiting for a resource which is currently allocated to another transaction available in that set.
- Four basic conditions to occur deadlocks are Mutual exclusion condition, Hold and wait condition, No-Preemption condition and Circular Wait condition.
- Deadlocks can be handled by three approaches that are (a) Deadlock prevention, (b) Deadlock avoidance and (c) Deadlock detection and resolution.
- If at least any one of the basic conditions to occur deadlocks is prevented then the occurrence of deadlocks can be prevented.
- Deadlocks can be avoided by carefully allocating resources to processes or transactions. Banker's algorithm is a scheduling algorithm to implement Deadlock avoidance and it was proposed by Edsger Dijkstra in 1965.
- Detection of deadlocks can be performed by maintaining a wait-for graph. The wait-for graph is a directed graph that represents which process is waiting for which processes to release their one or more resources. Each node in a wait-for graph represents a process or a transaction.
- The lock manager of a system contains the software to detect deadlocks.
- After deadlock detection, resolution strategy is used to break the deadlock condition in the system.
- Communication deadlocks occur in a situation where two or more than two processes are waiting for communication through messages from each other.Communication deadlock is particularly connected to inter-process communication.
- Communication deadlocks can be handled in two ways that are: Deadlock detection and recovery and Deadlock prevention.

CHECK YOUR PROGRESS

1. Fill in the blanks

(a) The four basic condition to occur deadlock are ____ , _____, ______ and _____.

(b) Communication deadlock is connected to _____.

(c) The ______of a system contains the software to detect deadlocks.

(d) The _____ is a directed graph that represents which processis waiting for which processes to release their one or more resources.

(e) ______is a scheduling algorithm used to avoid deadlocks.

5.7 ANSWERS TO CHECK YOUR PROGRESS

1. (a) Mutual exclusion condition, Hold and wait condition, No-Preemption condition, Circular Wait condition
   (b)inter-process communication
   (c) lock manager.
   (d) wait-for graph
   (e) Banker's algorithm

5.8 POSSIBLE QUESTIONS

1. Define deadlock. Write down the basic conditions to occur deadlocks.
2. Explain how deadlock can be prevented. Give examples.
3. Explain deadlock avoidance with examples.
4. Explain Banker's algorithm for multiple resources to implement Deadlock avoidance.
5. What is Communication deadlock? Write down the differences between Resource deadlock and Communication deadlock.
6. Write down the issues associated with Deadlock avoidance approach.
7. How distributed deadlocks can be detected?

8. What are the main issues of Deadlock Detection and Resolution approach to handle deadlocks?

5.9 REFERENCES AND SUGGESTED READINGS

- Coulouris, George, Jean Dollimore, and Tim Kindberg. "Distributed Systems: Concepts and Design Edition 4." (2005).
- Tanenbaum, Andrew S., and Maarten Van Steen. “*Distributed systems:Principles and Paradigms Edition 2.” (2007)*.

×××

UNIT: 6

DEADLOCK DETECTION AND RESOLUTION ALGORITHMS

Unit Structure:

6.1 INTRODUCTION

In the earlier chapter, we have already learnt that deadlock is a condition that occurs in a system when a set of transactions could not be able to complete their jobs as each transaction is waiting for at least one resource that is presently locked by another transaction available in that set. There are four basic conditions that must be satisfied all together so that deadlock may occur in a system. These conditions are Mutual exclusion condition, Hold and wait condition, No-Preemption condition and Circular Wait condition. We have also learnt that deadlocks can be handled by applying three approaches that are Deadlock prevention, Deadlock avoidance and Deadlock detection and resolution. In this unit, we are going to discuss about different deadlock detection algorithms and their issues. Different deadlock resolution strategies will also be discussed in this unit.

6.2 OBJECTIVES

- After going through this chapter, we will be able to learn:
- About Centralized deadlock detection algorithm.
- About Distributed deadlock detection algorithm.
- About Hierarchical deadlock detection algorithm.
- Different deadlock resolution strategies.

6.3 DEADLOCK DETECTION ALGORITHM

We have already learnt in the earlier chapter that a deadlock can be resolved after its detection. Deadlock detection can be performed by maintaining a wait-for graph. We already know that if a cycle of transactions or processes is developed in the wait-for graph then it represents a deadlock occurrence. So, in a single server system, deadlock detection can be performed by searching for cycles in the wait-for-graph. But a distributed system consists of multiple servers and each of these servers maintains one local wait-for graph. So, deadlock detection process requires extra effort in distributed systems. In distributed systems, a global wait-for graph may be developed for deadlock detections.In general, three approaches are available that can be used to detect deadlocks in distributed systems. These are (a) Centralized deadlock detection approach, (b)

Distributed Approach to detect deadlocks, and(c) Hierarchical deadlock detection approach.

6.4 CENTRALIZED DEADLOCK DETECTION

In the centralized deadlock detection approach, one server of a distributed system is given the responsibility to detect distributed deadlocks in the system. This server is referred as global deadlock detector. Each of the other servers available in the system sends regularly its recently updated local wait-for graph (Figure 12.1(a) and 12.1(b)) to the global deadlock detector. The global deadlock detector builds a global wait-for graph (Figure 12.1(c)) by combining all the information received from the local wait-for graphs and regularly updates it with any updated information received at any time from the local wait-for graphs. In the next step, the global deadlock detector, regularly searches for cycles in the global wait-for graph and if it discovers any cycle then it informs the servers about a best possible deadlock resolution strategy. For example, a cycle (P→A→C→R→P) is detected in the global wait-for graph as shown in figure 12.1(c).

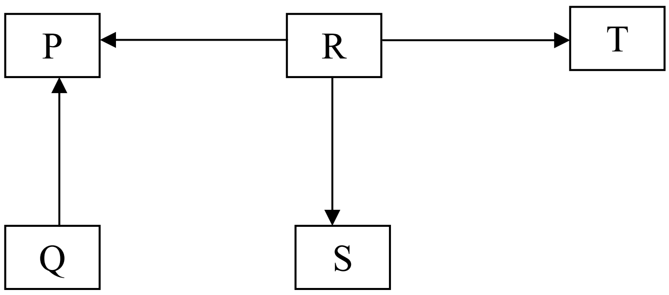


Figure 12.1(a) : Local Wait-For Graph in Server S1

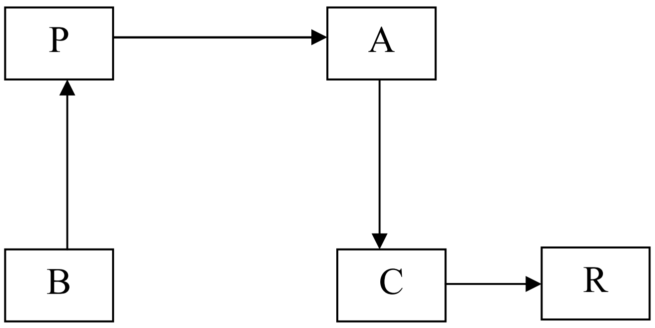


Figure 12.1(b) : Local Wait-For Graph in Server S2

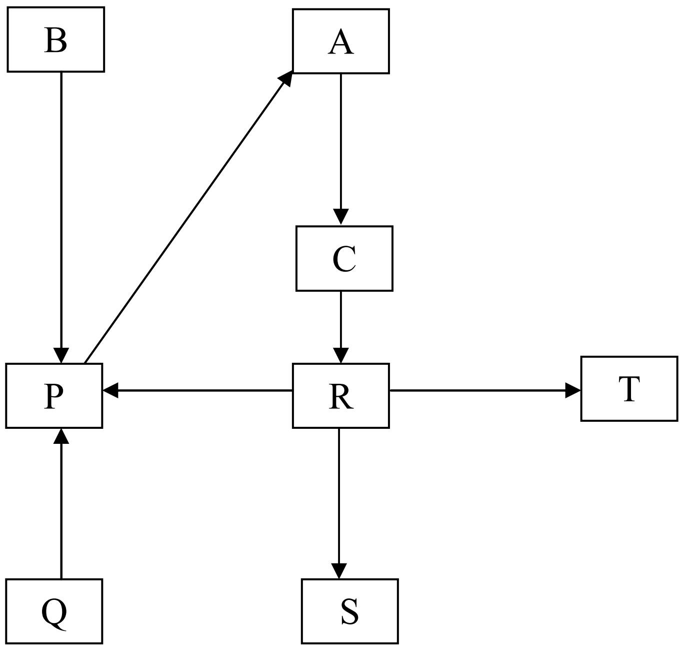


Figure 12.1(c) : Global Wait-For Graph in Global Deadlock Detector

6.4.1 Issues in Centralized Deadlock Detection

Now issues with this approach are presented in the following points.

- Fault tolerance is very poor in this approach as only one server controls the deadlock detection process. If the global deadlock detector crashes due to some reason then whole deadlock detection mechanism will be failed.
- If the local wait-for graphs are sent to the global deadlock detector very frequently then the cost of these transmissions will be very high. On the other hand, if the frequency of these transmissions is reduced then more time will be required to detect deadlocks and it may degrade the system performance.
- This approach does not have the capacity to scale.
- Phantom deadlocks may occur in the system due to this approach. If a deadlock is detected but in reality it is not a deadlock then that deadlock is referred as a Phantom deadlock. In case of the Centralized deadlock detection approach, all updated information from local wait-for graphs are regularly sent from different servers to the Global deadlock detector. After receiving updated information, the Global deadlock detector will detect a deadlock if it finds any cycle in the Global wait-for graph. This process requires some time. So, sometimes it may be happened that one process from the detected cycle releases its resources in that time duration and as a result the detected deadlock is not available in the system. But the Global deadlock detector will not have any idea about it and it will announce deadlock detection which is not actually available in reality. Lamport's algorithm can be used to avoid the occurrence of Phantom deadlocks.

STOP TO CONSIDER

In Lamport's algorithm, a global timing is maintained for distributed systems. When the Global deadlock detector receives new information from a server and detects a possible deadlock after updating the global wait-for graph then according to Lamport's algorithm, it immediately sends a message to all the servers that it just got a message with a timestamp, T^s that leads to a deadlock and if any server has a message with an timestamp before T^s for the global detector then send it to the detector instantly. After receiving reply from all the servers, the Global deadlock detector can find out the reality of the detected deadlock.The issue with this algorithm is that a global timing has to be maintained for the system which is costly.

6.5 DISTRIBUTED APPROACH TO DETECT DEADLOCKS

In Distributed approach to detect deadlocks, no specific server is made responsible for deadlock detection by forming a global wait-for graph. Instead of a global deadlock detector, every server participates in the process of deadlock detection in this approach. So, no global wait-for graph is required to form for deadlock detection by a specific server or node in a distributed system. Each server maintains a local wait-for graph for deadlock detection. Path-pushing and Edge-chasing are two important Distributed approaches for deadlock detection discussed in the following subsections.

6.5.1 Path-Pushing Algorithm

In Path-pushing algorithm, distributed deadlock detection process is performed by developing a global wait-for graph in each server of the distributed system. According to this algorithm, each server maintains a local wait for graph. When a server detects an external process in its local wait-for graph then it sends the graph to all the adjacent servers. Then each adjacent server updates its local wait-for graph with the new information and sends again this updated local wait-for graph to its adjacent sites. This process is performed repeatedly till any server receives enough information so that it can declare deadlock detection or prove that there is no deadlock available in the system.

Deadlock detection using Obermarck's Path Pushing deadlock detection algorithm is explained with the following figures (Figure 12.2(b), 12.2(c), 12.2(d) and 12.2(e)). In figure 12.2(a), a possible scenario of process dependencies in a distributed system with four servers (Server1, Server2, Server3 and Server4) is presented. Here, Server1 has three processes represented by '1', '2' and '3' where '3' is currently dependent on an external process and an external process is dependent on ‘1’. Server2 has two processes represented by ‘4’ and ‘5’ where ‘5’ is dependent on an external process and an external process is dependent on ‘4’. Server3 has three processes represented by ‘6’, ‘7’ and ‘8’ where ‘8’ is dependent on an external process and an external process is dependent on ‘6’. Finally, Server4 has only one process represented by ‘9’ and ‘9’ is dependent on an external process and an external process is dependent on ‘9’.

In this example, let us consider that the deadlock detection process is started in Server1. As shown in Figure 12.2 (b), the first step of the deadlock detection algorithm is started with Server1 where Server1 sends the path information from its local wait-for graph to Server2 through a string, “E_n 123 E_n”. E_n represents external process. Similarly, Server2 sends the path information from its local wait-for graph to Server3 through a string “E_n45 E_n”. Server3 sends the path information from its local wait-for graph to Server4 with a string “E_n68 E_n”. Finally, Server4 sends the path information from its local wait-for graph to Server1 with a string “E_n9 E_n”. After completion of the first step, it is observed that the string received by Server1 from Server4 does not include the path information of Server1’s local wait-for graph. It means that at this moment Server1 cannot declare the occurrence of deadlock in the system.

As shown in Figure 12.2 (c), in the second step of the deadlock detection algorithm, Server2 concatenate the path information received from Server1 in the first step with its current path information and the updated path information in Server2 is “E_n 12345 E_n”. Server3 concatenate the path information received from Server2 in the first step with its current path information and the updated path information in Server3 is “E_n 4568 E_n”. Finally, Server4 concatenate the path information received from Server3 in the first step with its current path information and the updated path information in Server4 is “E_n 689 E_n”. Each server sends its path information to its adjacent server. After completion of the second step, it is observed that the string received by Server1 from Server4 does not include the path information of its local wait-for graph. It means that at this moment Server1 cannot declare the occurrence of deadlock in the system.

Figure 12.2(d) presents the third step of the deadlock detection algorithm. In this step, Server3 concatenate the path information received from Server2 in the second step with its current path information and the updated path information in Server3 is “E_n 1234568 E_n”. Server4 concatenate the path information received from Server3 in the second step with its current path information and the updated path information in Server4 is “E_n 45689 E_n”. Each server sends its path information to its adjacent server. After completion of the third step, it is again observed that the string received by Server1from Server4 does not include its path information of its local wait-for graph. It means that at this moment Server1 cannot declare the occurrence of deadlock in the system.

Figure 12.2(e) presents the fourth step of the deadlock detection algorithm. In this step, Server4 concatenate the path information received from Server3 in the third step with its current path information and the updated path information in Server4 is “E_n 12345689 E_n”. Each server sends its path information to its adjacent server. After completion of the fourth step, it is observed that the string received by Server1 from Server4 include its path information of its local wait-for graph. It means that a cycle is formed in the system. As a result, at this moment Server1 can declare the occurrence of deadlock in the system.

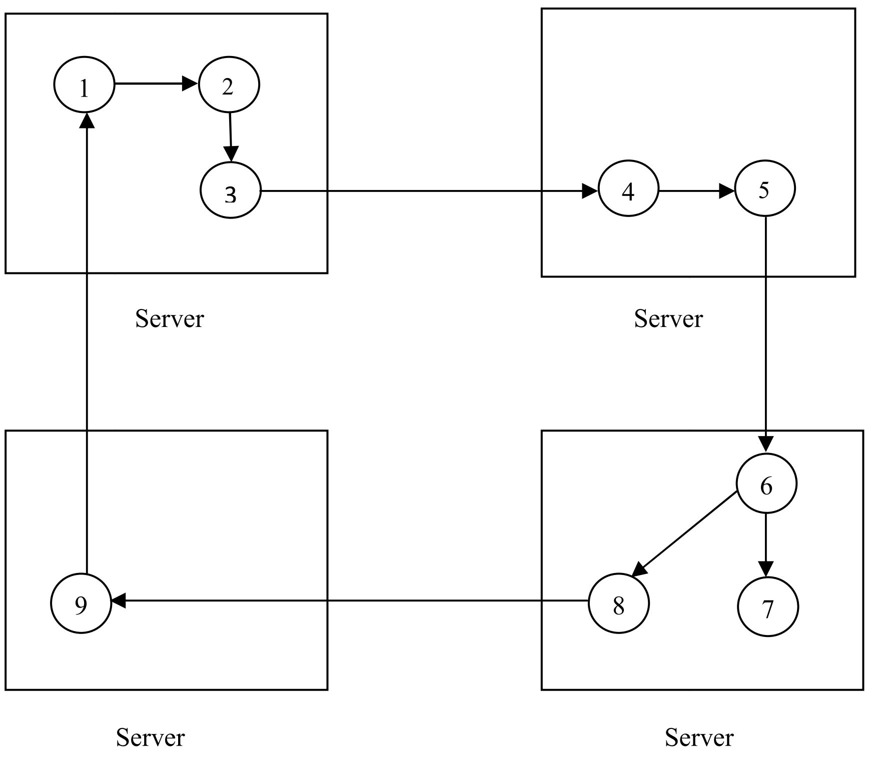


Figure 12.2 (a) Example of Process Dependencies in a Distributed System with Four Servers

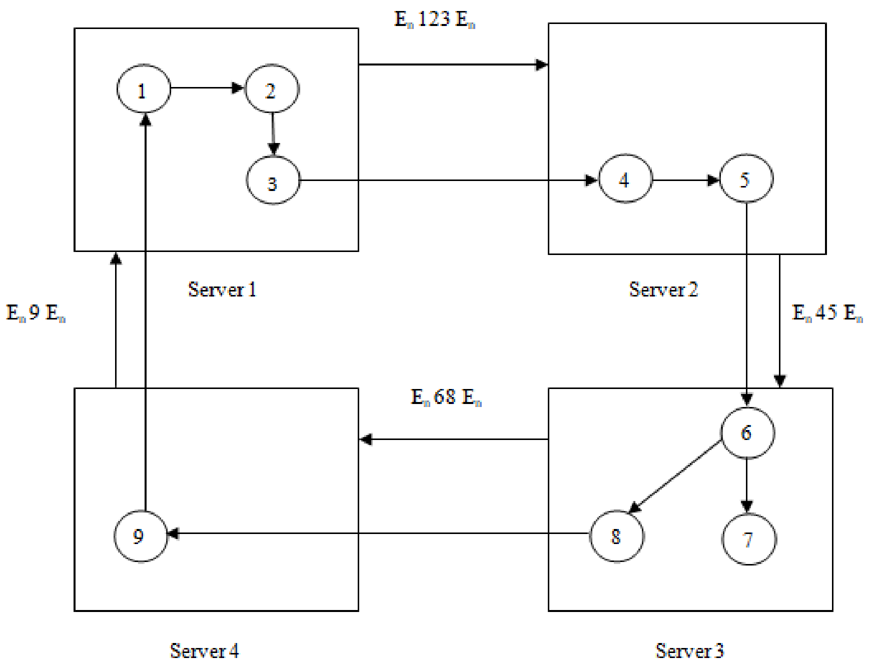


Figure 12.2 (b) First Step of Deadlock Detection using Path-Pushing Algorithm

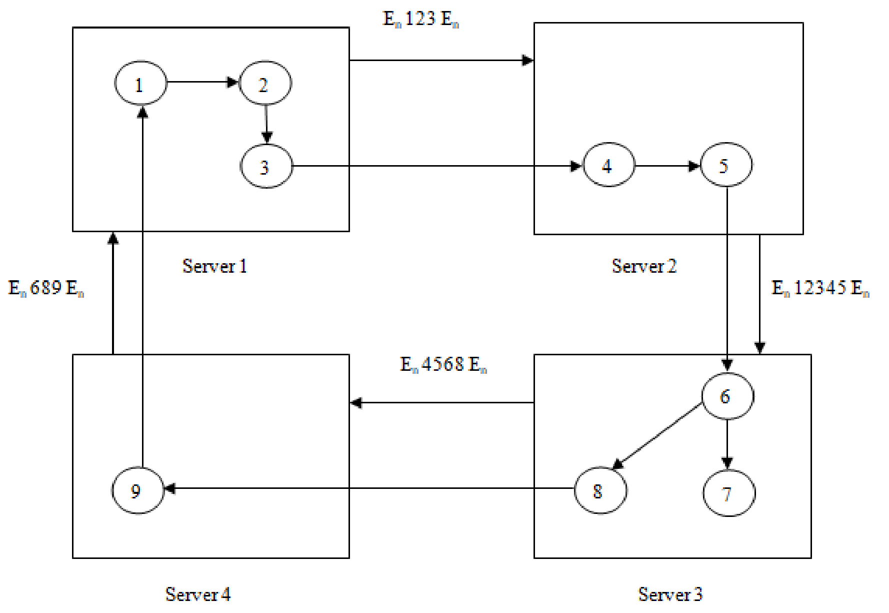


Figure 12.2 (c) Second Step of Deadlock Detection using Path-Pushing Algorithm

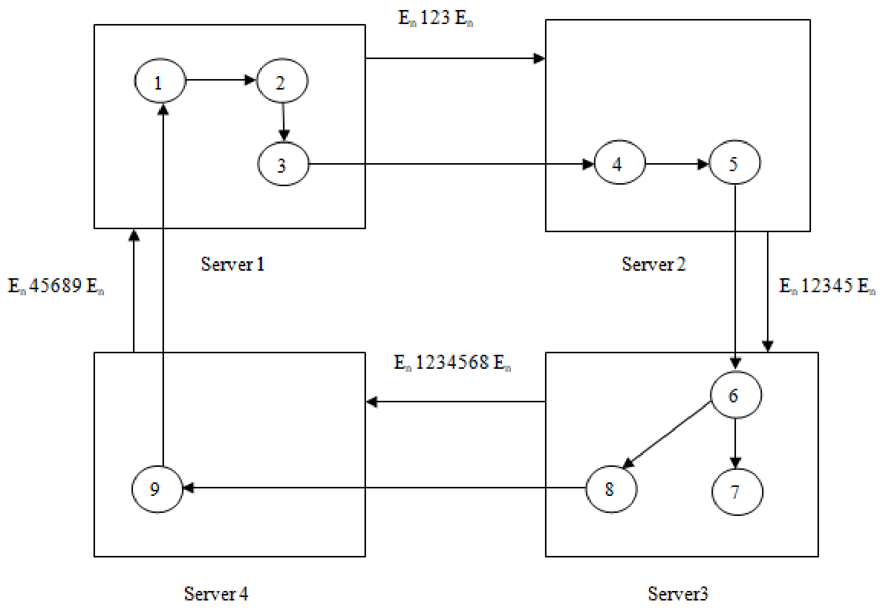


Figure 12.2 (d) Third Step of Deadlock Detection using Path-Pushing Algorithm

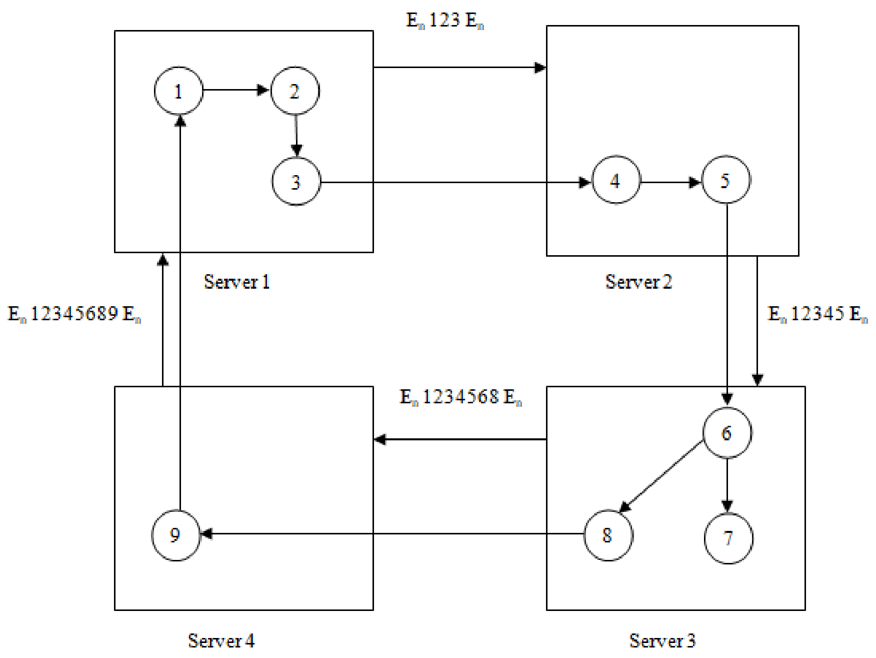


Figure 12.2 (e) FourthStep of Deadlock Detection using Path-Pushing Algorithm

6.5.2 Edge Chasing Algorithm

Edge chasing is another approach to detect distributed deadlocks where no global wait-for graph is formed. In this approach, no Global deadlock detector is available to detect distributed deadlocks in the system. Instead of a Global deadlock detector, each of the servers is involved in the deadlock detections. Each server maintains a local wait-for graph. According to this algorithm, the servers of the distributed system try to find cycles by forwarding messages to other servers. These messages are referred as probes and each probe contains three values. Let us consider, (a, b, c) is a probe where a refers the process P_a , b refers the process P_b and c refers the process P_c . Now, P_a is the generator of the probe, P_b is the sender of the probe and P_c is the receiver of the probe. It means that P_b sends the probe (a, b, c) to P_c because P_a is directly or indirectly dependent on P_b and P_b is waiting for a resource that is currently locked by $P_{c.}$ When a process or transaction in a server of a distributed system is waiting for a resource from specific time duration and still the resource request is not successful then it indicates a possible deadlock situation in the system. In this situation, the process generates a probe and forwards it to those processes which are currently holding the resources that are currently requested by it. Then each process which receives the probe will update the probe and again forward it to those processes which are currently holding the resources that are currently requested by it. This procedure continues and the probe is forwarded from one server to another server in the distributed system through different processes. When a probe is returned back to its generator process then it indicates the formation of a cycle that means a deadlock is detected.One important point is that if P_b sends the probe, (a, b, c) to P_c then P_c can discard the probe if it is currently not dependent on any other process. This algorithm is also known as Chandy Misra Haas distributed deadlock detection algorithm.

Figure 12.3 present the working of Chandy Misra Haas distributed deadlock detection algorithm on the scenario of process dependencies in a distributed system with four servers (Server1, Server2, Server3 and Server4) as presented in figure 12.2 (a). Let us consider, an integer value x refers the process, P_x. In this example, the process, P_1 is the generator of the probe and it sends the probe, (1, 1, 2) to the process, P_2. Then the process P_2, forward the probe (1, 2, 3) to the process P_3 and P_3 send the probe, (1, 3, 4) to P_4. This process continues and finally, the probe is returned back to the generator process, P_1 as (1, 9, 1). It indicates the formation of cycle in the distributed system that means a deadlock situation is developed in the system. From this example, it is also observed that the process, P_7 is not dependent on any other process and so, it discards the probe, (1, 6, 7) which is sent by the process, P_6.

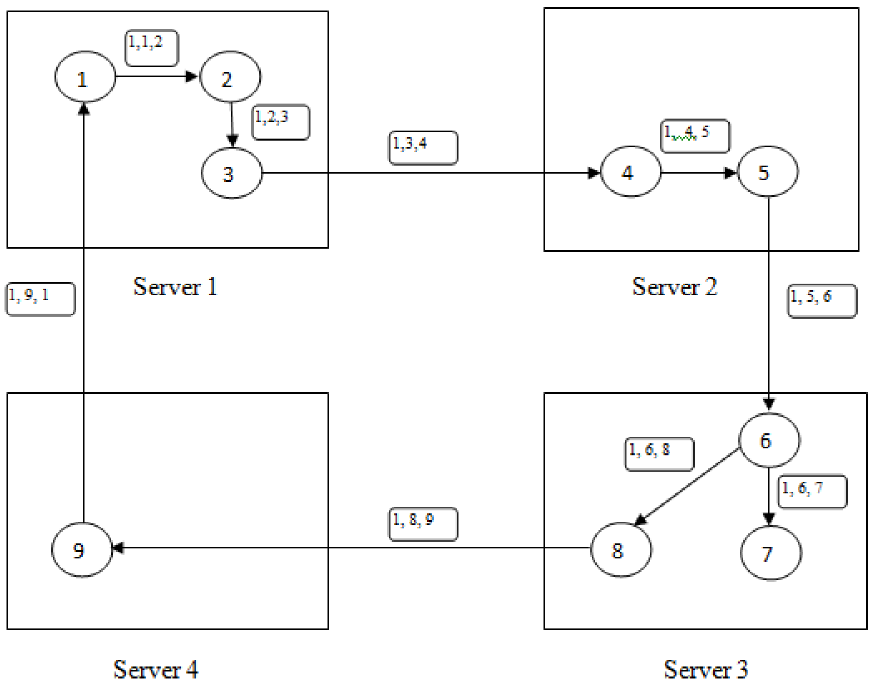


Figure 12.3: Distributed Deadlock Detection using Chandy Misra Haas distributed deadlock detection algorithm.

6.5.1 Issues in Distributed Approach to Detect Deadlocks

Different issues associated with the Distributed approach for deadlock detection are discussed in the following points.

- One of the major issues associated with distributed approach to detect deadlocks is its design complexity. It is a complex process to design a distributed approach which will be efficient and consistent in every possible situation that may be developed in a distributed system.
- Distributed algorithms may require large amount of information transfer among different servers. As a result, considerable amount of communication overhead may be observed in case of a large and complex distributed systems to detect deadlocks efficiently.
- Accuracy of deadlock detection may be affected by using Distributed approach due to inconsistent states of different servers available in the system. Inconsistent states from servers may be viewed due to different reasons like

information loss, network delays, variability in information delivery, variability in processing power among them etc.

- Due to this approach, false deadlock may be detected and sometimes the system may fail to detect an actual deadlock. For example, in case of Edge Chasing algorithm, if deadlock related probes are discarded then it may be possible that a deadlock will be occurred but it will not be detected. On the other hand if outdated probes are forwarded by the processes then it may be possible that deadlocks are detected that actually not exist in the system.
- The distributed approach must be able to handle the situation where one or more servers of the system are crashed due to some reason so that deadlock detection process can be continued without significant degradation of its performance.
- In case of Edge Chasing algorithm, probes may be lost or delayed due to network problems and it will affect the accuracy of the deadlock detection process. Additionally, in case of a large and complex distributed system, tracking probes of different processes may increase the system overhead.

CHECK YOUR PROGRESS

1. Fill in the blanks

(a) In Centralized deadlock detection approach, the server responsible for deadlock detection is referred as ________.

(b) If deadlock is detected in a system but actually it is not currently exist then that deadlock is referred as a _______.

(c) Path-Pushing algorithm is a ______ approach to detect deadlocks.

(d) In case of___algorithm, the servers of the distributed system try to find cycles by forwarding probes to other servers.

(e) In Edge chasing algorithm, process, P_b sends the probe (a, b, c) to the process______.

6.6 HIERARCHICAL DEADLOCK DETECTION

In the earlier sections, we have discussed the centralized approach and distributed approach to detect distributed deadlocks. It has been observed that both approaches are useful to detect distributed deadlocks but different issues are also associated with both of these approaches. In this section, we are going to discuss about Hierarchical approach for distributed deadlock detection.

Hierarchical deadlock detection approach is a hybrid approach where the concepts of both centralized approach and distributed approach are combined to detect distributed deadlocks in a distributed system. In this approach, servers are arranged in a hierarchical structure where one server is responsible for detection of deadlocks occurred in a cluster of servers and this server is the local deadlock detector of that cluster. There may be multiple clusters of servers in a system and deadlock detection of each cluster is performed by a local deadlock detector by using centralized or distributed approach. A global deadlock detector is responsible for managing the local deadlock detectors and detects deadlocks associated with multiple clusters. Within a cluster, each server detects deadlocks that are local to that server by using its local wait-for graph. Each server also transmits its local wait-for graph to the local deadlock detector for deadlock detection occurred within its cluster. Each local deadlock detectors transmit information related to the path dependencies of its cluster to the global deadlock detector. The global deadlock detector forms a higher level wait-for graph with these information and search cycles to detect inter-cluster deadlocks.

6.6.1 Issues in Hierarchical Deadlock Detection

Issues associated with the Hierarchical deadlock detection approach are presented in the following points.

- Implementation of Hierarchical deadlock detection approach is more complex than other deadlock detection approaches due to its requirement of hierarchical arrangement of servers and the coordination between different levels of the hierarchical arrangement.
- Maintenance of a Hierarchical structure in a dynamic environment is a complex process. So, maintenance complexity is increased in case of Hierarchical deadlock detection approach.
- Deadlock detection process may be slower in case of Hierarchical deadlock detection approach as in this approach, information are required to be combined and processed at different levels to detect distributed deadlocks.
- In this approach, failure of a server that is responsible for deadlock detection of a cluster of servers can have a negative impact on the deadlock detection process. So, servers available in the upper layers of the hierarchical arrangement are very crucial in the deadlock detection process. Failure of such servers may degrade the performance of the deadlock detection process significantly.

6.7 COMPARATIVE ANALYSIS OF DEADLOCK DETECTION ALGORITHMS

We have discussed the three approaches to detect distributed deadlocks in the earlier sections. Now, the most important point is to find out the best approach among them to detect distributed deadlocks. But there is no clear result to this query. It is observed that selection of an appropriate deadlock detection approach for a distributed system is dependent upon different factors like size and complexity of the system, requirement of resources, possible frequency of deadlock occurrence etc.

Implementation of Centralized deadlock detection is simple as only one server is responsible for deadlock detection process. Deadlock detection can be efficiently performed by using this approach in case of small or medium sized distributed systems with low deadlock occurrences. But in case of large and complex distributed systems, this approach may not be appropriate one. The global deadlock detector plays a very crucial role in this approach and failure of this server may stop the whole deadlock detection process. We have already discussed other issues associated with this approach in the earlier section.

Distributed approaches to detect deadlocks are more suitable in case of large and complex distributed systems where the frequency of deadlock occurrence is high. In this approach, every server may participate in the deadlock detection process. As a result, in this approach, the fault tolerance is better than the Centralized deadlock detection approach. This approach is more scalable and reliable. But designing an efficient distributed approach to detect distributed deadlocks is a complex process. Due to the requirement of large amount of communications among servers, system overhead may be increased in this approach.

The concepts of both Centralized deadlock detection and Distributed deadlock detection approach are used in Hierarchical deadlock detection approach to detect distributed deadlocks. In this approach, servers are arranged in a hierarchical structure where upper layer servers are responsible for deadlock detection associated with its descendant lower layer servers. Communication overhead can be reduced in this approach as the system is divided into different clusters of servers. Deadlock detection in each cluster is controlled by an upper layer server. If work load can be balanced between local and global deadlock detectors then this approach may perform better than the other two approaches. The main drawback of this approach that its implementation and maintenance is more complex than the other two approaches. Additionally, deadlock detection process may be slower in case of Hierarchical deadlock detection approach.

6.8 DEADLOCK RESOLUTION APPROACHES

In the following points, possible resolution strategies are discussed.

- If one or more than one processes associated with a deadlock are terminated then the cycle of processes as detected in the wait-for graph may be removed. As a result, the deadlock situation may be removed from the system. But it is a difficult job to find out which process or transaction associated in a deadlock should be terminated to resolve the deadlock situation. To find out a best possible solution for this problem, different factors may be considered. For example: priority of the process, age of the process, number of cycles associated with the process in the wait-for graph, number of resources allocated to the process etc. When Edge chasing algorithm is used to detect distributed deadlocks then it may be possible that multiple transactions associated with a cycle start deadlock detection process at the same time by forwarding probes. As a result, detection of a deadlock may be announced at different servers and it may be possible that multiple transactions will be terminated to resolve the same deadlock. To solve this problem, transactions or processes can be arranged in an order by assigning each transaction with a priority value. If deadlock occurs then the transaction with lowest priority and involved in that deadlock is terminated.

Another problem with this deadlock resolution approach is that the system may not work efficiently and data consistency issues may be occurred due to the termination of processes.

- Rollback one or more processes or transactions to a previous safe state can be performed instead of terminating processes to resolve deadlock situation in a system. In this approach, when a deadlock is detected then one or more processes involved in that deadlock are rollback to one of their previous safe state which exist before allocation of resources to them. As a result, it may break the cycle of processes available in the deadlock condition and deadlock condition will be removed from the system. The main challenge in this approach is that safe state of each process or transaction must be saved and managed at regular intervals.
- Lock timeouts can also be applied to remove deadlock condition from a system. In this approach, each lock is assigned with a certain amount of time. When a resource is locked then two cases may be observed. The first case is that no other processes are waiting for that resource and secondly, one or more processes are waiting for that resource. If the first case is observed then the resource can be remain locked even after the time period assigned with it.

But if the second case is observed then the resource is unlocked after the time period assigned with it. It is referred as the lock timeout. As a result of a lock timeout, a waiting process can resume its job and it may resolve a deadlock condition. Different issues observed in this approach are presented in the following points.

- Sometimes, lock timeout may happen even if there is no actual deadlock.
- In case of an overloaded system, some processes or transactions may require long time to perform their jobs and so, these processes may be badly affected by lock timeouts.
- It is very difficult to estimate a proper time period for lock timeouts.

CHECK YOUR PROGRESS

2. Choose the correct option

(a) Which of the following deadlock detection approach is not suitable for large and complex distributed system?
(i) Centralized deadlock detection
(ii) Distributed approach to detect deadlocks
(iii) Hierarchical deadlock detection
(iv) Both (i) and (ii)

(b) Which of the following deadlock detection approach is suitable for large and complex distributed system?
(i) Centralized deadlock detection
(ii) Distributed approach to detect deadlocks
(iii) Hierarchical deadlock detection
(iv) Both (ii) and (iii)

(c) Which of the following deadlock detection approach is suitable for small distributed system?
(i) Centralized deadlock detection
(ii) Distributed approach to detect deadlocks
(iii) Hierarchical deadlock detection
(iv) None of the above

(d) Which of the following is not a way to resolve deadlocks?
(i) Terminate one or more processes involved in a deadlock.
(ii) Using lock timeouts.
(iii) Prevent any one of the basic conditions to occur deadlocks.
(iv) Rollback one or more processes to a previous safe state.

(e) In ______, both the concepts of centralized approach and distributed approach are used to detect deadlocks.
(i) Hierarchical deadlock detection approach
(ii) Edge Chasing algorithm
(iii) Path-Pushing algorithm
(iv)None of the above

6.9 SUMMING UP

- The three approaches to detect deadlocks in distributed systems are (a) Centralized deadlock detection approach,(b) Distributed Approach to detect deadlocks, and(c) Hierarchical deadlock detection approach.
- In the centralized deadlock detection approach, one server of a distributed system is given the responsibility to detect distributed deadlocks in that system. This server is referred as global deadlock detector. It detects distributed deadlock by forming a global wait-for graph.
- If a deadlock is detected in a system but in reality it is not a deadlock then that deadlock is referred as a Phantom deadlock.
- In Distributed approach, every server participates in the process of deadlock detection. Path-pushing and Edge-chasing are two important Distributed approaches for deadlock detection in distributed systems.
- In Path-pushing algorithm, distributed deadlock detection process is performed by developing a global wait-for graph in each server of the distributed system. When a server detects an external process in its local wait-for graph then it sends the graph to all its adjacent servers.
- In Edge chasing algorithm, the servers of the distributed system try to find cycles by forwarding messages to other servers. These messages are referred as probes.

- In Hierarchical deadlock detection approach, the concepts of both centralized approach and distributed approach are used to detect distributed deadlocks. In this approach, servers are arranged in a hierarchical structure.
- Deadlock detection can be efficiently performed by using Centralized deadlock detection approach in case of small or medium sized distributed systems with low deadlock occurrences.
- Distributed approaches to detect deadlocks are more suitable in case of large and complex distributed systems where the frequency of deadlock occurrence is high.
- Deadlock can be resolved by three ways that are (a) Terminating one or more processes involved in a deadlock cycle, (b) Rollback one or more processes to a previous safe state, and (c) Using lock timeouts.

6.10 ANSWERS TO CHECK YOUR PROGRESS

1.

(a) Global deadlock detector
(b) Phantom deadlock
(c) Distributed
(d) Edge chasing
(e) P_c

2.

(a) (i) Centralized deadlock detection
(b) (iv) Both (ii) and (iii)
(c) (i) Centralized deadlock detection
(d) (iii) Prevent any one of the basic conditions to occur deadlocks.
(e) (i) Hierarchical deadlock detection approach

6.11 POSSIBLE QUESTIONS

1. Explain Centralized Deadlock Detection approach. Write down the issues in this approach.
2. Explain Distributed Deadlock Detection approach. Write down the issues in this approach.
3. Explain Hierarchical Deadlock Detection approach. Write down the issues in this approach.

4. Write down the deadlock resolution strategy that can be applied without detecting deadlocks.
5. Write down different deadlock resolution strategies that can be applied after detection of deadlocks.

6.12 REFERENCES AND SUGGESTED READINGS

- Chandy, K. Mani, Jayadev Misra, and Laura M. Haas. "Distributed deadlock detection." *ACM Transactions on Computer Systems (TOCS)* 1.2 (1983): 144-156.
- Tanenbaum, Andrew S. "*Distributed Operating Systems*" *(1995).*
- Coulouris, George, Jean Dollimore, and Tim Kindberg. "Distributed Systems: Concepts and Design Edition 4." (2005).
- Tanenbaum, Andrew S., and Maarten Van Steen. "*Distributed systems:Principles and Paradigms Edition 2." (2007).*

×××

BLOCK- III

UNIT: 1
AGREEMENT PROBLEMS AND PROTOCOLS

Unit Structure:

1.1 Introduction

Imagine you are working with a group of people, who arebased at different locations, each member contributing to a project. How do you ensure that everyone agrees on the project's direction and maintains consistency, especially when some team members might face communication issues or misunderstandings? This scenario is similar to what happens in distributed computing systems. Agreement problems are the binding factors that hold these systems together, ensuring that all nodes, or team members, reach a consensus despite potential failures or communication delays.

The most crucial task in distributed systems is to achieve consensus across multiple nodes, that is a must to ensure consistency, reliability, and coordination (Lamport, Shostak, & Pease, 1982). Distributed systems are intrinsically complex because of the lack of a central coordinating entity, the potential for node failures, and the variability in communication delays (Coulour is *et al.*, 2011). Agreement problems and protocols are mechanisms that help these systems function smoothly by enabling nodes to agree on a common value or decision despite failures and asynchronous communication (Castro & Liskov, 1999). This section introduces the fundamental concepts of agreement problems, highlighting their importance in maintaining the integrity of distributed systems, and sets the stage for a deeper exploration of their classifications and solutions.

To start, we will dive into the objectives of understanding these problems. You will see why grasping these concepts is essential for designing and operating reliable distributed systems. Think of it as knowing why having a meeting agenda is crucial for a productive team meeting. You will explore various types of agreement problems, learning how each type addresses specific challenges in distributed environments (Cachin, Guerraoui, & Rodrigues, 2011).After exploring the classification of agreement problems, we will examine solutions to the Byzantine Agreement Problem. Finally, we will look at real-world applications of agreement algorithms. You willobserve how these algorithms are used in blockchain technology to confirm the security and consistency of distributed ledgers, in distributed databases to maintain data consistency, in multi-agent systems to facilitate coordination, and in cloud computing to ensure reliable and consistent service delivery. These examples will show you the practical importance of agreement protocols in modern distributed systems.

By the end of this unit, you will have a comprehensive understanding of agreement problems and protocols, their significance, and their applications in real-world distributed environments. This knowledge will be invaluable in designing and operating robust distributed systems capable of handling faults and maintaining consistency, ensuring reliable and efficient performance.

1.2 Objectives

This unit explores various agreement problems and protocols in distributed systems. By the end of this unit, you should be able to -

- *understand* the fundamental concepts of agreement problems and why they are crucial in distributed systems. Think of it as knowing why having a meeting agenda is essential for a productive team meeting.

- *explore* the different types of agreement problems. This is like understanding the different ways to ensure everyone on your team is on the same page.
- *analyze* the Byzantine Agreement Problem in detail. It is a bit like dealing with a team member who might intentionally try to mislead the group.
- *examine* solutions and protocols like Byzantine Fault Tolerance (BFT) and Practical Byzantine Fault Tolerance (PBFT). These are strategies to ensure that your team can still reach a consensus, even under challenging conditions.
- *identify* real-world applications of agreement algorithms in systems like blockchain and cloud computing.
- *gain* a comprehensive overview of agreement protocols and their importance.

1.3 Classification of Agreement Problems

In distributed systems, agreement problems in can be classified based on the type of the faults they address and the mechanisms used to achieve consensus. Understanding these classifications helps in identifying the appropriate protocols and solutions for different scenarios. Let us explore some of the main types of agreement problems and their unique characteristics.

1.3.1 Byzantine Agreement

Let us dive into a tricky situation: imagine one of your team members isn't just misunderstanding but is intentionally trying to disrupt the project. This is akin to the Byzantine Agreement Problem in distributed systems. It requires nodes to agree on a value in the situation where some nodes are behaving arbitrarily or maliciously. To achieve Byzantine Agreement, certain conditions must be met:

- **Termination**: At some point, each working node must choose a value.
- **Agreement**: The non-faulty nodes must be agreeing on the same value.
- **Validity**: The agreed-upon value must be the starting value if it is shared by all non-faulty nodes.

The challenge here is filtering out the "noise" from those disruptive nodes and still reaching a common decision (Lamport, Shostak, & Pease, 1982).

STOP TO CONSIDER

- Why is Byzantine Agreement important?
  - ✓ It ensures system reliability despite faulty or malicious nodes.
  - ✓ It is critical for applications like blockchain and secure communicationsystems.
- What makes achieving Byzantine Agreement challenging?
  - ✓ Handling arbitrary faults and misleading information.
  - ✓ High communication overhead and complexity.

1.3.2 Consensus Problem

Now, think about a situation where all your team members are trying to agree on the next step of the project. The consensus problem in distributed systems is similar. It requires all non-faulty nodes to agree on a single value, depending on their initial values. For consensus to be achieved, the following conditions must be met:

- **Termination**: At some point, each non-faulty node needs to choose a value.
- **Agreement**: At some point, each non-faulty node needs to choose a value.
- **Validity**: The agreed-upon value must be the starting value if it is shared by all non-faulty nodes.

The challenge lies in ensuring that all team members reach the same decision, even if some face issues (Pass & Shi, 2017).

1.3.3 Interactive Agreement

Imagine an ongoing conversation within your team where each member shares their thoughts until everyone agrees. This is similar to Interactive Agreement in distributed systems, which involves a series of communications between nodes to reach consensus. For interactive agreement to be achieved:

- **Multiple Rounds**: Nodes participate in several rounds of message exchanges.
- **Convergence**: The system must ensure that the values

proposed by non-faulty nodes converge to a single decision.

- **Fault Tolerance**: The system should tolerate a certain number of faulty nodes and still reach an agreement.

The complexity here is managing these interactions efficiently (Cachin, Guerraoui, & Rodrigues, 2011).

STOP TO CONSIDER

- Byzantine Agreement deals with arbitrary or malicious behaviour.
- The goal of the Consensus Problem is for every non-faulty node to concur on a single value.
- Interactive Agreement involves multiple rounds of communication to achieve consensus.
- Fault Tolerance: The ability of a system to continue operating properly in the event of the failure of some of its components.
- Convergence: Ensuring that the proposed values by non-faulty nodes converge to a single decision.
- Communication Overhead: The extra communication required to achieve consensus, especially in Byzantine Agreement.

Check Your Progress

CYP1. Define the Byzantine Agreement Problem. What are the conditions required for achieving Byzantine Agreement?

CYP2. Why is Byzantine Agreement particularly challenging in distributed systems, and what are some real-world scenarios where it is essential?

CYP3. What is consensus problem in the context of distributed systems? Describe the three key conditions that must be encountered for consensus.

CYP4. Discuss the main challenges in achieving consensus in a distributed system having potentially faulty nodes.

CYP5. Explain what Interactive Agreement is and how it differs from the standard Consensus Problem.

CYP6. Identify the unique challenges associated with Interactive Agreement and how they impact the communication between nodes.

1.4 Solutions to the Byzantine Agreement Problem

The Byzantine Agreement Problem is one of the most challenging issues in distributed systems because of the presence of faulty or malicious nodes. To address this problem, numerous protocols have been developed. These protocols aim to ensure that non-faulty nodes can still reach a consensus in situations where some nodes are behaving arbitrarily or maliciously. In this section, we will explore two key solutions: Byzantine Fault Tolerance (BFT) and Practical Byzantine Fault Tolerance (PBFT).

Key Concepts in Byzantine Agreement Solutions

Before exploring specific solutions, it is important to understand some of the fundamental concepts that are common to these protocols:

- **Fault Models:** Different fault models, including crash faults and Byzantine faults, dictate the complexity of achieving consensus. Byzantine faults are the most severe, as they include any arbitrary behaviour by faulty nodes.
- **Message Complexity:** The number of messages exchanged between nodes is a critical factor, as high message complexity can result in inefficiencies.
- **Cryptographic Techniques:**In order to confirm the integrity and authenticity of messages, many Byzantine agreement protocols rely on cryptographic techniques, *viz.*digital signatures and hash functions.
- **Redundancy and Replication:** These techniques are frequently used to make sure that even if some nodes fail, the system can still reach consensus.

1.4.1 Byzantine Fault Tolerance

Think of Byzantine Fault Tolerance (BFT) as a strategy to ensure your team can come to an agreement even if some members are trying to disrupt the process. BFT protocols use a combination of message exchanges and cryptographic techniques to filter out the influence of Byzantine nodes and ensure that non-faulty nodes can agree on a common value (Castro & Liskov, 1999).BFT protocols are designed to handle Byzantine faults by ensuring that non-faulty nodes can arrive at a consensus despite of the faulty nodes. The fundamental idea is to use redundant computations and message exchanges to filter out the influence of faulty nodes.

Overview of BFT

The BFT protocol requires nodes to exchange a series of messages to agree on a value. All nodes are in communication with one another, and through a process of majority voting and redundancy, the system can achieve consensus. This protocol typically tolerates up to $\frac{n-1}{3}$ faulty nodes, where n is the overall number of nodes present in the system.

Phases of BFT

1. Pre-Vote Phase: Each node proposes a value based on its initial state.
2. Vote Phase: Nodes exchange their proposed values with each other.
3. Commit Phase: Nodes decide on the final value based on the majority of received votes.

Advantages of BFT

- Robustness: BFT can tolerate a significant number of faulty nodes, making it extremely robust against a range of failures.
- Security: Cryptographic techniques are used to ensure that messages cannot be tampered with, providing a high level of security.

Challenges of BFT

- Communication Overhead: The protocol requires a high number of message exchanges, leading to substantial communication overhead.
- Scalability: Due to its high communication complexity, BFT can become inefficient in large-scale systems.

STOP TO CONSIDER

- ➢ Byzantine Fault Tolerance (BFT) uses message exchanges and cryptographic techniques to achieve consensus despite malicious nodes.
- ➢ BFT is highly robust and secure but can face challenges in terms of communication overhead and scalability.
- ➢ Message Exchanges: The communication among nodes required to achieve consensus.

1.4.2 Practical Byzantine Fault Tolerance

Practical Byzantine Fault Tolerance (PBFT) is designed for practical applications, making it more efficient and suitable for real-world scenarios. It uses a three-phase protocol (pre-prepare, prepare, commit) to achieve consensus with fewer message exchanges. Imagine organizing a team meeting where you propose an idea, discuss it, and then finalize the decision once everyone agrees (Castro & Liskov, 1999).

Overview of PBFT

PBFT enhances the traditional BFT approach by organizing the protocol into three distinct phases: pre-prepare, prepare, and commit. This structure increases the efficiency of the protocol by reducing the number of message exchanges needed to reach consensus.

Phases of PBFT

1. **Pre-Prepare Phase:** The leader node (or the primary node) proposes a value and broadcasts it to all the other nodes (replicas).
2. **Prepare Phase:** Each replica verifies the proposal and broadcasts a prepare message to all other replicas if it finds the proposal valid.
3. **Commit Phase:** Once a replica receives a majority of prepare messages, it sends a commit message. When a majority of commit messages is received, the replica commits the value.

Advantages of PBFT

- **Efficiency:** By reducing the number of required message exchanges, PBFT is more efficient and scalable compared to traditional BFT protocols.
- **Practicality:** PBFT is designed for real-world applications, making it suitable for use in systems like blockchain and distributed databases.

Challenges of PBFT

- **Leader Selection:** The protocol relies on a leader to propose values, making it susceptible to performance degradation if the leader fails or becomes slow.
- **Complexity:**Though being more efficient than BFT, PBFT is still complex to implement and manage.

STOP TO CONSIDER

- Practical Byzantine Fault Tolerance (PBFT) is optimized for real-world applications, making it more efficient and scalable.
- PBFT reduces message complexity by organizing the protocol into three phases: pre-prepare, prepare, and commit.
- Leader Selection: The process of choosing a primary node to propose values.
- Efficiency: The ability of the protocol to achieve consensus with fewer message exchanges.

Check Your Progress

CYP7. Describe the Byzantine Fault Tolerance (BFT) protocol. How does it achieve consensus in the presence of Byzantine faults?

CYP8. What are the strengths and weaknesses of BFT, and in what scenarios is it most effectively applied?

CYP9. Outline the Practical Byzantine Fault Tolerance (PBFT) protocol and its three-phase process. What role does each stage have in reaching a consensus?

CYP10. Compare PBFT to traditional BFT. List the practical advantages of PBFT in real-world applications?

1.5 Applications of Agreement Algorithms in Distributed Systems

Agreement algorithms are fundamental to the operation of various distributed systems. Their ability to ensure consensus among nodes, even when faulty nodes are present, makes them indispensable in many real-world applications. This section explores how these algorithms are applied in blockchain technology, distributed databases, multi-agent systems, and cloud computing.

1.5.1 Blockchain Technology

Blockchain technology relies heavily on agreement algorithms to maintain the security and integrity of a distributed ledger. In a blockchain, each block of transactions must be agreed upon by the network before it is added to the chain. By ensuring that every node has the same version of the ledger, this agreement helps to avoid problems like double-spending.

Key Concepts in Blockchain Consensus:

- **Proof of Work (PoW):** An algorithm where nodes (miners) solve complex cryptographic puzzles to propose a new block. The first node to solve the puzzle gets to add the block to the chain and is rewarded. This process, used in Bitcoin, ensures that adding new blocks requires significant computational effort, making it difficult for malicious actors to alter the blockchain (Nakamoto, 2008).

- **Proof of Stake (PoS):** Instead of mining, nodes are chosen to propose new blocks depending on the number of coins they hold and are willing to "stake" as collateral. This method, used in Ethereum 2.0, lowers energy usage in comparison to PoW and aligns the incentives of participants with the network's security (Buterin, 2014).
- **Practical Byzantine Fault Tolerance (PBFT):**PBFT is used in permissioned blockchains, PBFT allows nodes to reach consensus through a series of message exchanges in three phases: pre-prepare, prepare, and commit. This method provides low-latency finality and is efficient in environments with a known set of participants, such as Hyperledger Fabric (Castro & Liskov, 1999).

Applications in Blockchain:

- **Bitcoin:** Uses PoW to secure its network, making it resistant to tampering and attacks.
- **Ethereum:** Transitioning from PoW to PoS to improve scalability and reduce environmental impact.
- **Hyperledger Fabric:** Employs PBFT for fast and reliable consensus in enterprise blockchain solutions.

1.5.2 Distributed Databases

Distributed databases ensure the consistency and dependability of data across several nodes through consensus protocols. These protocols help maintain a single version of the truth, in situations when some nodes fail or become unreachable.

Key Concepts in Distributed Database Consensus:

- **Two-Phase Commit (2PC):** A protocol used to ensure all nodes present in a distributed database either commit to a transaction or abort it, preventing partial updates. The coordinator node sends a prepare message to all participant nodes and waits for their acknowledgment. If all nodes agree, the coordinator sends a commit message; otherwise, it sends an abort message (Gray, 1978).

- **Three-Phase Commit (3PC):** An extension of 2PC that introduces an additional phase to handle coordinator failures, reducing the chances of a system-wide deadlock (Skeen, 1981).
- **Paxos:** A consensus algorithm that is intended to get distributed nodes agree on a single value. It tolerates node failures and asynchronous communication, making it suitable for large-scale distributed databases (Lamport, 1998).
- **Raft:** Similar to Paxos but designed to be more understandable and easier to implement. It divides the consensus process into leader election, log replication, and safety (Ongaro & Ousterhout, 2014).

Applications in Distributed Databases:

- **Google Spanner:** Uses Paxos for distributed consensus, providing strong consistency and global distribution.
- **Amazon DynamoDB:** Employs a version of Paxos for eventual consistency and high availability.
- **CockroachDB:** Implements Raft to ensure consistency and fault tolerance across distributed nodes.

1.5.3 Multi-Agent Systems

Agreement algorithms help multi-agent systems coordinate and make decisions. Imagine a team of robots working together, each relying on consensus protocols to synchronize their actions and achieve a common goal (Cachin, Guerraoui, & Rodrigues, 2011).In multi-agent systems, agreement algorithms facilitate coordination and decision-making among autonomous agents. These systems often involve robots, drones, or software agents that work together to reach a common goal.

Key Concepts in Multi-Agent Consensus:

- **Consensus-Based Control:**It makes certain that every agent in a network is in agreement on a single state or choice. This is vital for tasks like formation control, where robots must maintain specific positions relative to each other (Olfati-Saber, Fax, & Murray, 2007).

- **Distributed Task Allocation:** Assigns tasks to agents in a way that balances the workload and optimizes performance. Consensus algorithms help ensuring the agreement of all the agents on task assignments without central coordination (Gerkey & Mataric, 2004).
- **Swarm Intelligence:** Models the collective behaviour of decentralized, self-organized systems, such as ant colonies or bird flocks. Consensus algorithms enable swarms to coordinate their actions so that they can adapt to the changes in the environment (Beni & Wang, 1993).

Applications in Multi-Agent Systems:

- **Robotic Swarms:** Use consensus algorithms to coordinate movements and tasks, enabling applications like search and rescue or environmental monitoring.
- **Autonomous Vehicles:** Vehicles communicate with one another to maintain safe distances and optimize traffic flow, using consensus protocols to agree on routes and speeds.
- **Distributed AI:** Multi-agent systems in AI research use consensus algorithms to combine the outputs of different models or agents, improving decision-making and performance.

1.5.4 Cloud Computing

Cloud computing systems rely on agreement algorithms to ensure reliable and consistent service delivery across distributed resources. These protocols manage data replication, load balancing, and fault tolerance in cloud environments.

Key Concepts in Cloud Computing Consensus:

- **Data Replication:** Ensures that data are kept in many copies on several nodes, to improve reliability and access speed. Consensus algorithms help synchronize these copies and ensure consistency (Birman & Joseph, 1987).
- **Load Balancing:** Distributes workloads across multiple servers to optimize resource use and minimize response times. Consensus protocols help maintain a balanced state across the cloud infrastructure (Lu *et al.*, 2011).
- **Fault Tolerance:** Enables cloud services to keep operating even during the failure of some of the components. Consensus algorithms guarantee that the system can recover

and maintaining consistency despite failures (Cachin *et al.*, 2011).

Applications in Cloud Computing:

- **Kubernetes:** Uses consensus protocols to manage container orchestration, ensuring that applications run reliably across distributed nodes.
- **Amazon Web Services (AWS):** Implements consensus algorithms in its data replication and load balancing services to provide high availability and fault tolerance.
- **Google Cloud Platform (GCP):** Uses consensus protocols in its distributed storage and computing services to ensure data consistency and service reliability.

CheckYourProgress

CYP11. Give an example of how agreement algorithms are used in blockchain technology. What role do these algorithms play in maintaining a secure and consistent ledger?

CYP12. Describe how consensus protocols are applied in cloud computing. How do they ensure reliable and consistent service delivery?

1.6 Summing Up

In this unit, we explored the fundamental concepts and importance of agreement problems and protocols in distributed systems. We began by comprehendingthe significance of consensus in preserving consistency, dependability, and coordination among nodes in the face of malfunctions or communication delays. We then examined different types of agreement problems, including Byzantine Agreement, Consensus Problem, and Interactive Agreement, highlighting their unique challenges and conditions for achieving consensus.

We delved into solutions for the Byzantine Agreement Problem, focusing on Byzantine Fault Tolerance and Practical Byzantine Fault Tolerance. BFT ensures consensus by filtering out the influence of the nodes that are faulty, through extensive exchanges of messages and cryptographic techniques. PBFT, optimized for practical applications, reduces message complexity, and improves efficiency.

Finally, we discussed the real-world applications of agreement algorithms in blockchain technology, distributed databases, multi-agent systems, and cloud computing. These applications demonstrate how consensus protocols enhance security, consistency, and fault tolerance in various distributed environments.
Understanding and implementing these protocols is essential for designing robust and resilient distributed systems capable of withstanding faults and ensuring reliable operation.

Self-AskingQuestions

SAQ1. In what ways do agreement problems shape the overall reliability and integrity of distributed systems, and how might different types of faults impact the consensus process?

SAQ2. How do the Byzantine Agreement and Consensus Problems differ in terms of their challenges and solutions, and what practical scenarios can you think of where each would be applied?

SAQ3. Considering the high communication overhead in Byzantine Fault Tolerance (BFT), what strategies might you employ to balance robustness and efficiency in a large-scale distributed system?

SAQ4. Reflect on how PBFT optimizes the consensus process for real-world applications. What are the key advantages of PBFT, and how does it mitigate the limitations of traditional BFT protocols?

SAQ5. How do consensus algorithms like PBFT and Paxos enhance the functionality and reliability of systems such as blockchain, distributed databases, and cloud computing? Can you identify potential challenges in implementing these algorithms in different distributed environments?

1.7 References and Suggested Readings

1. Beni, G., & Wang, J. (1993). Swarm Intelligence in Cellular Robotic Systems. *Proceedings of the NATO Advanced Workshop on Robots and Biological Systems*, 102, 1-8.
2. Birman, K. P., & Joseph, T. A. (1987). Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems (TOCS)*, 5(1), 47-76.

3. Buterin, V. (2014). A Next-Generation Smart Contract and Decentralized Application Platform. Retrieved from https://ethereum.org/en/whitepaper/
4. Cachin, C., Guerraoui, R., & Rodrigues, L. (2011). Introduction to Reliable and Secure Distributed Programming. Springer.
5. Castro, M., & Liskov, B. (1999). Practical Byzantine Fault Tolerance. Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI).
6. Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2011). Distributed Systems: Concepts and Design (5th ed.). Addison-Wesley.
7. Gray, J. (1978). Notes on Data Base Operating Systems. In R. Bayer, R. M. Graham, & G. Seegmüller (Eds.), *Operating Systems: An Advanced Course* (pp. 393-481). Springer-Verlag.
8. Gerkey, B. P., & Mataric, M. J. (2004). A Formal Analysis and Taxonomy of Task Allocation in Multi-Robot Systems. *The International Journal of Robotics Research*, 23(9), 939-954.
9. Lamport, L., Shostak, R., & Pease, M. (1982). The Byzantine Generals Problem. ACM Transactions on Programming Languages and Systems (TOPLAS), 4(3), 382-401.
10. Lamport, L. (1998). The Part-Time Parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2), 133-169.
11. Lu, R., Lin, X., Zhu, H., Ho, P.-H., & Shen, X. (2011). A Novel Energy-Efficient Data Gathering Protocol in Wireless Sensor Networks. *IEEE Transactions on Vehicular Technology*, 60(7), 3443-3458.
12. Nakamoto, S. (2008). Bitcoin: A Peer-to-Peer Electronic Cash System. Retrieved from https://bitcoin.org/bitcoin.pdf
13. Olfati-Saber, R., Fax, J. A., & Murray, R. M. (2007). Consensus and Cooperation in Networked Multi-Agent Systems. *Proceedings of the IEEE*, 95(1), 215-233.
14. Ongaro, D., & Ousterhout, J. (2014). In Search of an Understandable Consensus Algorithm (Extended Version). *USENIX Annual Technical Conference*, 305-319.
15. Pass, R., & Shi, E. (2017). The Sleepy Model of Consensus. Advances in Cryptology – ASIACRYPT 2017.
16. Skeen, D. (1981). Nonblocking Commit Protocols. *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, 133-142.
17. https://www.geeksforgeeks.org/what-is-a-distributed-system/

1.8 Model Questions

1. Explain the importance of achieving consensus in distributed systems. How does it affect system reliability and coordination?
2. Define the Byzantine Agreement Problem. What are the main conditions that must be satisfied to achieve Byzantine Agreement?
3. Discuss the main challenges in solving the consensus problem in distributed systems. How do these challenges impact the termination, agreement, and validity conditions?
4. Differentiate between Byzantine Agreement and the Consensus Problem in the context of distributed systems. Provide examples of scenarios where each is applicable.
5. Describe the Interactive Agreement process. What are the unique challenges associated with achieving Interactive Agreement in distributed systems?
6. Outline the mechanism of Byzantine Fault Tolerance (BFT). How does it ensure consensus in the presence of Byzantine faults?
7. Explain the Practical Byzantine Fault Tolerance (PBFT) protocol. Describe its three-phase process and how it contributes to achieving consensus.
8. Compare and contrast Byzantine Fault Tolerance (BFT) and Practical Byzantine Fault Tolerance (PBFT). What are the strengths and weaknesses of each protocol?
9. How do agreement algorithms like PBFT enhance the security and reliability of blockchain technologies? Provide specific examples.
10. Discuss the role of consensus protocols in maintaining data consistency in distributed databases. How do these protocols ensure reliable transactions?
11. Explain how agreement algorithms facilitate coordination and decision-making in multi-agent systems. Provide an example to illustrate your explanation.
12. Describe how consensus protocols are applied in cloud computing environments. What are the key benefits of using these protocols in such scenarios?
13. What are the implications of failing to achieve consensus in a distributed system? Discuss with reference to potential faults and failures.

14. Evaluate the impact of cryptographic techniques on the effectiveness of Byzantine Fault Tolerance (BFT) protocols.
15. How does the iterative communication process in Interactive Agreement contribute to achieving consensus in distributed systems?
16. Discuss the significance of the validity condition in the context of the Consensus Problem. Why is it crucial for ensuring correct outcomes?
17. What are the practical challenges of implementing PBFT in large-scale distributed systems? How can these challenges be mitigated?
18. In what ways can understanding agreement problems and protocols influence the design of more resilient distributed applications?
19. Explain the role of message exchanges in Byzantine Fault Tolerance. How do they help in filtering out the influence of Byzantine nodes?
20. Summarize the key takeaways from studying agreement problems and protocols. How can this knowledge be applied to future advancements in distributed computing?

1.9 Answer to check your progress questions

CYP1. Define the Byzantine Agreement Problem. What are the conditions required for achieving Byzantine Agreement?

Answer: The Byzantine Agreement Problem requires nodes in a distributed system to agree on a single value even if some of the nodes (Byzantine nodes) exhibit arbitrary or malicious behaviour. The conditions required for achieving Byzantine Agreement are:

1. **Termination:**At some point, each non-faulty node needs to choose a value.
2. **Agreement:**Every non-faulty node needs to concur on the same value
3. **Validity:**The agreed-upon value must be the starting value if it is shared by all non-faulty nodes.

CYP2. Why is Byzantine Agreement particularly challenging in distributed systems, and what are some real-world scenarios where it is essential?

Answer: Byzantine Agreement is particularly challenging because it must account for nodes that can act arbitrarily or maliciously, sending conflicting or misleading information. This makes it difficult to achieve consensus as nodes cannot rely on the trustworthiness of their peers. Real-world scenarios where Byzantine Agreement is essential include:

- **Blockchain Technology:** Ensuring all nodes agree on the state of the distributed ledger despite potential malicious actors.
- **Military Communication Systems:** Securing command and control systems against potential sabotage or misinformation.
- **Financial Systems:** Protecting against fraudulent transactions or data tampering.

CYP3. What is consensus problem in the context of distributed systems? Describe the three key conditions that must be encountered for consensus.

Answer: The consensus problem in distributed systems requires all non-faulty nodes to agree on a single value based on their initial values. The three key conditions that must be met for consensus are:

1. **Termination:** Every non-faulty node must eventually agree on a value.
2. **Agreement:** All non-faulty nodes must chose the same value.
3. **Validity:** If all non-faulty nodes have the same initial value, the agreed-upon value must be that initial value.

CYP4. Discuss the main challenges in achieving consensus in a distributed system having potentially faulty nodes.

Answer: The main challenges in achieving consensus in a distributed system with potentially faulty nodes include:

- **Communication Delays:** Asynchronous communication can cause delays, making it difficult to synchronize decisions.
- **Faulty Nodes:** Nodes may fail or behave maliciously, sending incorrect or conflicting information.
- **Network Partitions:** Temporary network failures can isolate nodes, preventing them from participating in the consensus process.

- **Scalability:** As the number of nodes increases, the complexity and overhead of achieving consensus also increase.

CYP5. Explain what Interactive Agreement is and how it differs from the standard Consensus Problem.

Answer: Interactive Agreement involves a series of communications between nodes to reach consensus, often requiring multiple rounds of message exchanges. It differs from the standard Consensus Problem in that it typically involves more complex interactions and iterative communication to ensure convergence and fault tolerance. While the standard Consensus Problem focuses on a single decision round, Interactive Agreement uses iterative processes to refine and validate decisions.

CYP6. Identify the unique challenges in Interactive Agreement and how they impact the communication between nodes.

Answer: Unique challenges associated with Interactive Agreement include:

- **High Communication Overhead:** Multiple rounds of message exchanges can lead to significant communication overhead.
- **Latency:** The iterative process can introduce delays, impacting the overall speed of reaching agreement.
- **Fault Tolerance:** Managing faults across multiple rounds of communication can be complex and resource-intensive.
- **Synchronization:** Ensuring all nodes participate in each round of communication and maintain synchronization is challenging.

CYP7. Describe the Byzantine Fault Tolerance (BFT) protocol. How does it achieve consensus in the presence of Byzantine faults?

Answer: The Byzantine Fault Tolerance (BFT) protocol achieves consensus by using a combination of message exchanges and cryptographic techniques to filter out the influence of Byzantine nodes. Each node communicates with every other node, and through a process of majority voting and redundancy, the system can achieve consensus. BFT typically tolerates up to n−13\frac{n-1}{3}3n−1 faulty nodes, where nnn is the total number of nodes.

CYP8. What are the strengths and weaknesses of BFT, and in what scenarios is it most effectively applied?

Answer:Strengths:

- **Robustness:** BFT can tolerate a significant number of faulty nodes, making it highly robust.
- **Security:** Cryptographic techniques ensure message integrity and authenticity.

Weaknesses:

- **Communication Overhead:** High number of message exchanges lead to substantial communication overhead.
- **Scalability:** BFT can become inefficient in large-scale systems due to high communication complexity.

Scenarios where BFT is most effectively applied:

- **Blockchain Technology:** Ensuring secure and consistent ledgers.
- **Secure Communication Systems:** Protecting against malicious interference.
- **Financial Systems:** Ensuring reliable transaction processing.

CYP9. Outline the PBFT protocol and its three-phase process. How does each phase contribute to achieving consensus?

Answer: The Practical Byzantine Fault Tolerance (PBFT) protocol is organized into three phases: pre-prepare, prepare, and commit.

1. **Pre-Prepare Phase:** The leader node proposes a value and broadcasts it to all other nodes.
2. **Prepare Phase:** Each node verifies the proposal and broadcasts a prepare message to all other nodes if the proposal is valid.
3. **Commit Phase:** Once a node receives a majority of prepare messages, it sends a commit message. When a majority of commit messages is received, the node commits the value.

Each phase helps ensure synchronization among nodes and filters out any misleading information from faulty nodes, thus achieving consensus efficiently.

CYP10. Compare PBFT to traditional BFT. List the practical advantages of PBFT in real-world applications?

Answer:Comparison to Traditional BFT:

- **Message Complexity:** PBFT reduces the number of message exchanges compared to traditional BFT, making it more efficient.
- **Phases:** PBFT uses a structured three-phase process, whereas traditional BFT involves more complex interactions.

Practical Advantages of PBFT:

- **Efficiency:** Lower message complexity makes PBFT more scalable and suitable for real-world applications.
- **Latency:** PBFT provides faster consensus with lower latency.
- **Applicability:** Suitable for permissioned blockchain systems and distributed databases where node identities are known.

CYP11. Give an example of how agreement algorithms are used in blockchain technology. What role do these algorithms play in maintaining a secure and consistent ledger?

Answer: In block chain technology, agreement algorithms like PBFT are used to ensure that all nodes must come to an agreement on the state of the distributed ledger. For example, Hyperledger Fabric employs PBFT to achieve consensus among known participants. These algorithms maintain a secure and consistent ledger by making sure that all nodes validate and agree on each block of transactions before it is added to the chain, preventing issues like double-spending and tampering.

CYP12. Describe how consensus protocols are applied in cloud computing. How do they ensure reliable and consistent service delivery?

Answer: In cloud computing, consensus protocols are used to manage data replication, load balancing, and fault tolerance through distributed resources. For example, Kubernetes uses consensus algorithms to orchestrate container deployment and scaling. These protocols ensure reliable and consistent service delivery by synchronizing the state and configuration of all nodes, enabling seamless scaling, and maintaining service availability even when node failures exist.

×××

UNIT- 2

IPC AND COMMUNICATION PROTOCOLS

Unit Structure:

2.1 Introduction

Inter-Process Communication (IPC) in distributed systems refers to the mechanisms and techniques used for communication and data exchange between different processes running on different machines within a network. In a distributed system, processes are not limited to a single machine; they can span across multiple machines, which adds complexity to communication and coordination. It is used for exchanging data between multiple threads in one or more processes or programs. The Processes may be running on single or multiple computers connected by a network. The full form of IPC is Inter-process communication.

In another words IPC, is set of interfaces, which is usually programmed in order for the programs to communicate between series of processes. This allows running programs concurrently in an Operating System. It is a set of programming interface which allow a programmer to coordinate activities among various program processes which can run concurrently in an operating system. This allows a specific program to handle many user requests at the same time.

Since every single user request may result in multiple processes running in the operating system, the process may require to communicate with each other. Each IPC protocol approach has its own advantage and limitation, so it is not unusual for a single program to use all of the IPC methods.

2.2 Objectives

After going through this unit you will be able to:

- Understand the basic concepts of Inter-Process Communication (IPC) in distributed systems
- Know about the importance of IPC.
- Know about different approaches of IPC.
- Know about some Common IPC mechanisms used in distributed systems
- Understand about Application Programming Interfaces (API) for UDP and TCP
- Know about the Request-Reply Protocol for Communication in distributed system
- Idea about basics of Remote Procedure Call (RPC) in distributed system
- Know about some examples of RPC Usage

2.3 Importance of Inter-Process Communication (IPC)

Here, are the reasons for using the inter-process communication protocol for information sharing:

- It helps to speedup modularity
- Computational
- Privilege separation
- Convenience
- Helps operating system to communicate with each other and synchronize their actions.

2.3.1 Characteristics of Inter-Process Communication

There are mainly five characteristics of inter-process communication in a distributed environment/system.

- **Synchronous System Calls:** In the synchronous system calls both sender and receiver use blocking system calls to transmit the data which means the sender will wait until the acknowledgment is received from the receiver and receiver waits until the message arrives.

- **Asynchronous System Calls:** In the asynchronous system calls, both sender and receiver use non-blocking system calls to transmit the data which means the sender doesn't wait from the receiver acknowledgment.
- **Message Destination:** A local port is a message destination within a computer, specified as an integer. Aport has exactly one receiver but many senders. Processes may use multiple ports from which to receive messages. Any process that knows the number of a port can send the message to it.
- **Reliability:** It is defined as validity and integrity.
- **Integrity:** Messages must arrive without corruption and duplication to the destination.
- **Validity:** Point to point message services are defined as reliable, If the messages are guaranteed to be delivered without being lost is called validity.
- **Ordering:** It is the process of delivering messages to the receiver in a particular order. Some applications require messages to be delivered in the sender order i.e the order in which they were transmitted by the sender.

2.4 Approaches for Inter-Process Communication

Here, are few important methods for inter-process communication:

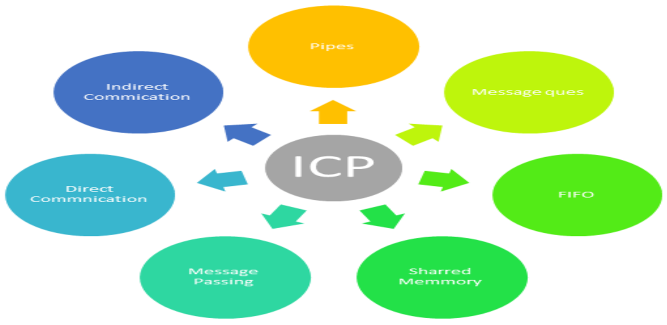


Figure 1. Inter-Process Communication Approaches

Pipes

Pipe is widely used for communication between two related processes. This is a half-duplex method, so the first process communicates with the second process. However, in order to achieve a full-duplex, another pipe is needed.

Message Passing

It is a mechanism for a process to communicate and synchronize. Using message passing, the process communicates with each other without resorting to shared variables.

IPC mechanism provides two operations:

- Send (message)- message size fixed or variable
- Received (message)

Message Queues

A message queue is a linked list of messages stored within the kernel. It is identified by a message queue identifier. This method offers communication between single or multiple processes with full-duplex capacity.

Direct Communication

In this type of inter-process communication process, should name each other explicitly. In this method, a link is established between one pair of communicating processes, and between each pair, only one link exists.

Indirect Communication

Indirect communication establishes like only when processes share a common mailbox each pair of processes sharing several communication links. A link can communicate with many processes. The link may be bi-directional or unidirectional.

Shared Memory

Shared memory is a memory shared between two or more processes that are established using shared memory between all the processes. This type of memory requires to protected from each other by synchronizing access across all the processes.

FIFO

Communication between two unrelated processes. It is a full-duplex method, which means that the first process can communicate with the second process, and the opposite can also happen.

The important terms used in IPC are Semaphores and Signals which are defined below:

Semaphores: A semaphore is a signaling mechanism technique. This Operating System method either allows or disallows access to the resource, which depends on how it is set up.

Signals: It is a method to communicate between multiple processes by way of signaling. The source process will send a signal which is recognized by number, and the destination process will handle it.

2.5 Some Common IPC mechanisms used in distributed systems

2.5.1 Message Passing

This is a fundamental IPC mechanism where processes communicate by sending and receiving messages. In a distributed system, messages are typically sent over the network using protocols like TCP/IP or UDP. Message passing can be synchronous or asynchronous, and it's often used for communication between processes running on different machines.

2.5.2 Remote Procedure Calls (RPC)

RPC allows a process to invoke a procedure or function in another process as if it were a local procedure call. The RPC mechanism hides the complexities of network communication, making it easier for developers to build distributed applications. However, developers need to be mindful of issues like network latency, failures, and data consistency.

2.5.3 Remote Method Invocation (RMI)

RMI is a Java-specific form of RPC that enables communication between Java objects across different Java Virtual Machines (JVMs). RMI allows objects in one JVM to invoke methods on objects in another JVM, making it easier to build distributed Java applications.

2.5.4 Shared Memory

Shared memory IPC allows processes to communicate by accessing shared regions of memory. In a distributed system, shared memory can be implemented using distributed shared memory (DSM) techniques, where the memory is distributed across multiple machines but appears as a single address space to processes. However, DSM introduces challenges related to consistency, coherence, and synchronization.

2.5.5 Publish/Subscribe

In this model, processes (or components) publish messages to specific topics or channels, and other processes subscribe to receive messages from those topics. Publish/subscribe mechanisms are often used in distributed systems for event-driven architectures, where components need to react to events generated by other components.

2.5.6 Socket Programming

Sockets provide a low-level IPC mechanism for communication between processes over a network. In a distributed system, processes can communicate using sockets by establishing connections and exchanging data streams. Socket programming allows for flexibility and customization but requires developers to manage details like connection establishment, data serialization, and error handling.

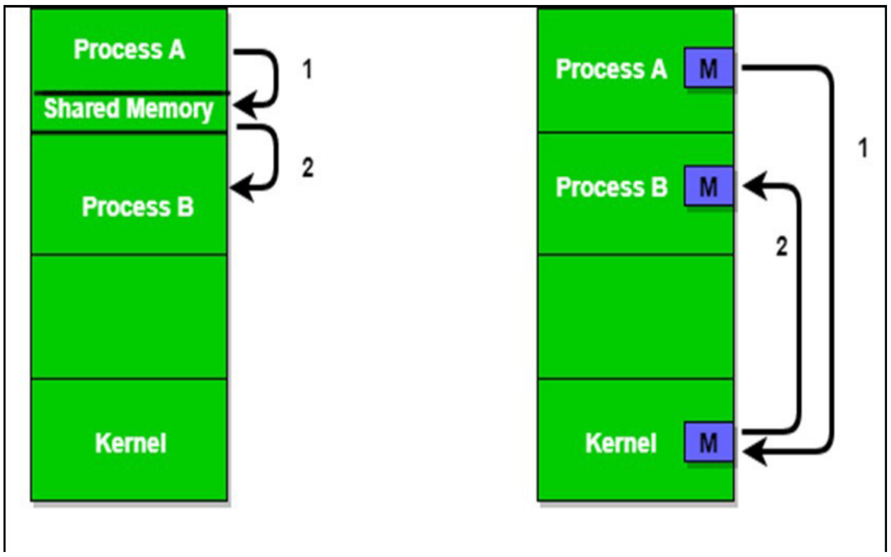


Figure 2. Shared Memory and Message Passing

Each IPC mechanism has its advantages and disadvantages, and the choice depends on factors such as performance requirements, programming language, scalability, fault tolerance, and ease of implementation. In distributed systems, designers often use a combination of IPC mechanisms to meet the specific needs of the application.

2.6 Application Programming Interfaces (API) for UDP and TCP

As we know that UDP (User Datagram Protocol) and TCP (Transmission Control Protocol) are two widely used transport layer protocols in computer networks. APIs (Application Programming Interfaces) for UDP and TCP provide programmers with the necessary functions and methods to create, send, receive, and manage network communications using these protocols. Below, we explain the basic concepts of API for UDP and TCP:

2.6.1 API for TCP

Socket Creation: The API provides functions to create a TCP socket.

Binding and Listening: For server applications, the TCP socket needs to be bound to a specific IP address and port and set to listen for incoming connections. The API provides functions for these purposes.

Connection Establishment: TCP is a connection-oriented protocol, so before data exchange can occur, a connection needs to be established between the client and server. The API includes functions to initiate connections from the client side and accept connections on the server side.

Data Transmission: Programmers can use the API to send and receive data over an established TCP connection. Data is transmitted reliably and in order, and the API provides functions to handle large data transfers, buffering, and flow control.

Connection Termination: TCP connections need to be properly terminated once data exchange is complete. The API includes functions to gracefully close connections from both the client and server sides.

Error Handling: TCP provides reliable, error-checked delivery of data, but errors such as connection timeouts or broken connections can still occur. The API includes mechanisms for error detection and recovery.

2.6.2 API for UDP

Socket Creation: It is similar to TCP. The API provides functions to create a UDP socket, which is a communication endpoint that allows data to be sent and received over UDP.

Binding: Before using a socket, it needs to be bound to a specific network interface and port. The API provides functions to bind a socket to a particular IP address and port number.

Sending Data: Programmers can use the API to send data over UDP. This involves specifying the destination IP address, port number, and the data to be sent.

Receiving Data: The API provides functions to receive data on a UDP socket. Programmers can specify the maximum size of the data buffer and retrieve the sender's IP address and port number along with the received data.

Error Handling: UDP is a connectionless protocol, so errors like packet loss or duplication may occur. The API includes error-handling mechanisms to handle such situations.

In summary, the API for UDP and TCP provides programmers with the necessary tools to implement network communication using these protocols, including socket creation, data transmission, error handling, and connection management. Programmers can use these APIs to develop a wide range of networked applications, from simple client-server interactions to complex distributed systems.

2.7 The Request-Reply Protocol for Communication in distributed system

The Request-Reply Protocol is a fundamental communication pattern used in distributed systems where one component (the client) sends a request message to another component (the server), and the server responds with a corresponding reply message. This protocol is widely used in various distributed systems scenarios, including client-server architectures, micro services, and remote procedure calls (RPC).

Here's how the Request-Reply Protocol works:

Request Message: The client initiates communication by sending a request message to the server. The request message typically contains information about the action the client wants the server to perform. This could be a query for data, a request to execute a specific function, or any other operation that the server is capable of handling.

Server Processing: Upon receiving the request message, the server processes the request based on its functionality. This may involve executing the requested operation, accessing data, performing calculations, or any other task required to fulfill the client's request.

Reply Message: After processing the request, the server generates a reply message containing the result of the operation or the requested data. The reply message is then sent back to the client as a response to the original request.

Client Handling: Upon receiving the reply message, the client processes the response to extract the information it needs. Depending on the application logic, the client may take further actions based on the contents of the reply message, such as displaying data to the user, performing additional processing, or sending subsequent requests to the server.

Error Handling: In addition to successful responses, the Request-Reply Protocol also includes mechanisms for handling errors and exceptions. If an error occurs during request processing on the server side, the server can generate an error response indicating the nature of the problem. The client then needs to handle these error responses appropriately, which may involve retrying the request, notifying the user, or taking other corrective actions.

2.7.1 Key characteristics of the Request-Reply Protocol

Synchronous Communication: Request-reply communication is typically synchronous, meaning that the client waits for a response from the server before proceeding with further actions. This synchronous nature simplifies the programming model, as the client can assume that a response will be received in a predictable manner.

Reliability: The protocol ensures reliable communication between the client and server by requiring that each request be acknowledged with a corresponding reply. This ensures that both parties are aware of the outcome of the communication and can take appropriate actions based on the response.

Statelessness: The Request-Reply Protocol is often designed to be stateless, meaning that each request-reply interaction is independent of previous interactions. This simplifies the design and scalability of distributed systems by allowing servers to handle requests from multiple clients concurrently without maintaining client-specific state between requests.

Overall, the Request-Reply Protocol provides a straightforward and reliable communication mechanism for building distributed systems, allowing components to interact seamlessly across network boundaries while ensuring that communication is predictable and robust.

2.8 Basics of Remote Procedure Call (RPC) in distributed system

Remote Procedure Call (RPC) is a protocol that enables a program to execute procedures or functions on a remote system as if they were local, abstracting away the details of network communication. It allows distributed applications to communicate and invoke procedures across different systems transparently, making it appear as if the remote procedure is a local function call.

Here the basics of Remote Procedure Call (RPC) in a distributed system are given below:

Invocation: The client program calls a procedure or function on the remote system as if it were a local function call. From the client's perspective, there's no distinction between local and remote procedures. The client specifies the procedure name and provides the necessary parameters for the remote invocation.

Marshalling: Before the request is sent over the network, the parameters of the procedure call need to be converted into a format that can be transmitted. This process is called marshalling or serialization. Complex data structures and objects are serialized into a byte stream that can be transmitted over the network.

Communication: The client sends the serialized request message containing the procedure name and parameters to the server over the network. This communication typically occurs using a transport layer protocol such as TCP/IP.

Unmarshalling: Upon receiving the request, the server needs to deserialize or unmarshal the incoming message to extract the procedure name and parameters. This process reconstructs the original data structures from the byte stream received over the network.

Execution: Once the server has extracted the procedure name and parameters, it invokes the corresponding procedure or function locally, using the provided parameters. From the server's perspective, the invocation is a local function call, and it executes the requested operation as if it were initiated locally.

Result Marshalling: After executing the procedure, the server may return a result or response to the client. The result, along with any output parameters, needs to be serialized or marshalled into a format suitable for transmission over the network.

Response: The server sends the serialized response message back to the client over the network.

Result Unmarshalling: Upon receiving the response, the client unmarshals or deserializes the message to extract the result and any output parameters returned by the remote procedure call.

Completion: Finally, the client receives the result of the remote procedure call and can continue with its execution based on the returned values.

RPC frameworks and libraries, such as gRPC, Apache Thrift, and Java RMI, provide tools and APIs to simplify the implementation of RPC-based communication in distributed systems. These frameworks handle many of the underlying details, such as marshalling, communication, and error handling, allowing developers to focus on defining the interface and implementing the procedures to be invoked remotely.

2.9 RPC Operations, Parameter Passing in RPC in distributed system

Remote Procedure Call (RPC) operations facilitate the seamless execution of functions or procedures across distributed systems, abstracting away the complexities of network communication. The RPC model typically involves several operations and mechanisms for parameter passing, ensuring smooth interaction between client and server components. Here's a concise explanation of RPC operations and parameter passing:

2.9.1 RPC Operations

Bind: In the bind operation, the client initiates communication with the server by establishing a connection. This involves identifying the server's network address and establishing a communication channel, often using TCP/IP or another transport protocol. The bind operation sets up the foundation for subsequent RPC operations.

Call: The call operation is the heart of RPC, where the client invokes a remote procedure on the server. The client specifies the name of the procedure to be executed, along with any required parameters. The RPC runtime system then marshals the procedure name and parameters into a message format suitable for transmission over the network.

Execute: Upon receiving the RPC call request, the server executes the specified procedure locally. The server locates the appropriate procedure based on the name provided in the RPC call and executes it with the provided parameters. The execution may involve complex computations, database queries, or other operations, depending on the functionality of the remote procedure.

Return: After executing the procedure, the server generates a response containing the result of the operation, along with any output parameters. This response message is then sent back to the client over the network. The return operation completes the remote procedure call cycle initiated by the client.

Unbind: In the unbind operation, the client or server terminates the RPC connection, releasing any allocated resources and closing the communication channel. This operation is typically performed after all required RPC calls have been completed or when the client or server no longer requires communication.

2.9.2 Parameter Passing in RPC

Marshalling: Before transmitting parameters over the network, the RPC system serializes or marshals the parameters into a format suitable for transmission. This involves converting data structures, objects, and primitive values into a byte stream that can be transmitted over the network. Marshalling ensures that parameters are represented consistently across different systems and programming languages.

Transmission: Once marshalled, the RPC system transmits the parameter data along with the procedure name to the server over the network. This transmission may occur using a reliable transport protocol such as TCP/IP to ensure the integrity and reliability of data delivery.

Unmarshalling: Upon receiving the RPC call request, the server unmarshals or deserializes the incoming parameters from the network message. This process reconstructs the original data structures and values from the byte stream received over the network, making the parameters accessible for procedure execution.

Execution: With the parameters unmarshalled, the server executes the specified procedure locally using the provided parameters. The execution may involve accessing databases, performing computations, or interacting with other components, depending on the functionality of the remote procedure.

Result Marshalling: After executing the procedure, the server marshals the result and any output parameters into a response message format suitable for transmission back to the client. Marshalling ensures that the result data is represented consistently for transmission over the network.

Result Transmission and Unmarshalling: Finally, the server transmits the response message containing the result back to the client over the network. The client then unmarshals the result and any output parameters from the response message, making them available for further processing or display.

In summary, RPC operations enable the seamless execution of remote procedures across distributed systems, while parameter passing mechanisms ensure the consistent transmission and representation of data between client and server components. By abstracting away the complexities of network communication, RPC facilitates efficient and transparent interaction between distributed components, enabling the development of robust and scalable distributed applications.

2.10 Some Examples of RPC Usage

RPC is widely used in various distributed systems and networked applications for seamless communication between client and server components. Here are some examples of RPC usage in different domains:

2.10.1 Client-Server Applications

RPC is commonly used in client-server architectures for communication between client applications and server-side services. For example:

Web services: A client application can invoke remote procedures exposed by a web service using RPC, such as retrieving data from a database or performing authentication.

Remote administration: Systems administrators can use RPC to remotely execute administrative tasks on servers, such as starting or stopping services, managing files, or monitoring system health.

2.10.2 Microservices Architecture

In microservices-based systems, RPC is often used for communication between individual microservices. Each microservice exposes a set of RPC endpoints, allowing other microservices to invoke their functionality. For example:

User service: A microservice responsible for managing user accounts might expose RPC endpoints for operations like user authentication, profile management, and access control.

Payment service: Another microservice handling payment processing might provide RPC endpoints for processing payments, generating invoices, and managing payment methods.

2.10.3 Distributed Computing

RPC is essential for distributed computing environments where computation is distributed across multiple machines or nodes. Examples include:

Grid computing: Large-scale scientific or computational tasks can be parallelized and distributed across a grid of interconnected computers. RPC facilitates communication between grid nodes to coordinate task execution and data exchange.

Map Reduce frameworks: Distributed data processing frameworks like Apache Hadoop and Apache Spark utilize RPC for communication between master and worker nodes. RPC is used to distribute computation tasks, exchange intermediate results, and coordinate job execution.

2.10.4 Inter-process Communication (IPC)

Within a single machine or operating system, RPC can be used for communication between different processes or threads. Examples include:

Remote method invocation (RMI) in Java: Java applications can use RMI, a form of RPC, to invoke methods on remote objects running in different Java Virtual Machines (JVMs) within the same network or on different machines.

Named pipes on Windows: RPC can be used for inter-process communication on Windows systems using named pipes, allowing processes to communicate and exchange data within the same machine.

2.10.5 Embedded Systems

RPC can also be utilized in embedded systems for communication between microcontrollers, sensors, and other devices. For example:

IoT applications: Internet of Things (IoT) devices often communicate with backend servers or cloud services using RPC protocols like MQTT or CoAP. RPC enables devices to send sensor data, receive commands, and interact with remote services.

Automotive systems: In-vehicle communication networks use RPC for communication between electronic control units (ECUs) responsible for functions such as engine control, braking, and infotainment.

Overall, RPC is a versatile communication mechanism used in a wide range of distributed systems and applications to enable seamless interaction between remote components, services, and devices.

2.11 Summing Up

- **Definition:** Inter-process communication is used for exchanging data between multiple threads in one or more processes or programs.
- Inter-Process Communication (IPC) in distributed systems refers to the mechanisms and techniques used for communication and data exchange between different processes running on different machines within a network.
- Message Passing is a fundamental IPC mechanism where processes communicate by sending and receiving messages. In a distributed system, messages are typically sent over the network using protocols like TCP/IP or UDP. It is a mechanism for a process to communicate and synchronize.
- Shared memory IPC allows processes to communicate by accessing shared regions of memory.
- Sockets provide a low-level IPC mechanism for communication between processes over a network. In a distributed system, processes can communicate using sockets by establishing connections and exchanging data streams.
- Pipe is widely used for communication between two related processes.
- A message queue is a linked list of messages stored within the kernel.
- Direct process is a type of inter-process communication process, should name each other explicitly.
- Indirect communication establishes like only when processes share a common mailbox each pair of processes sharing several communication links.
- Shared memory is a memory shared between two or more processes that are established using shared memory between all the processes.
- Inter Process Communication method helps to speedup modularity.
- A semaphore is a signaling mechanism technique.
- Signaling is a method to communicate between multiple processes by way of signaling.
- The Request-Reply Protocol is a fundamental communication pattern used in distributed systems where one component (the client) sends a request message to another component (the server), and the server responds with a corresponding reply message.
- Remote Procedure Call (RPC) is a protocol that enables a program to execute procedures or functions on a remote system as if they were local, abstracting away the details of network communication.

- Like FIFO follows FIFO method whereas Unlike FIFO use method to pull specific urgent messages before they reach the front.

2.12 References and Suggested Readings:

- Tanenbaum, A. S., & Van Steen, M. (2006). Distributed systems: principles and paradigms. Prentice Hall.
- Stallings, William (2009).Computer Communication: Architecture Protocols and Standards.
- Stallings, William, Tenth Edition, Data and Computer Communications, Pearson

2.13 Model Questions

1. What do you mean by Inter-Process Communication (IPC)?Explain its various characteristics.
2. Explain the importance of Inter-Process Communication (IPC).
3. What are different approaches for Inter-Process Communication?
4. Explain the basics of Remote Procedure Call (RPC) in distributed system.
5. Explain RPC Operations, Parameter Passing in RPC in distributed system.
6. What are the key characteristics of the Request-Reply Protocol?
7. What do you mean by Distributed Computing?

2.14 Answer to check your progress/Possible Answers to SAQ

1. Select the correct options of the following questions

(i). IPC Stands for

(a) Inter-Process Communication. (b) Inter-Program Communication

(c) Interface-Process Communication (d) None of above.

(ii) A fundamental IPC mechanism where processes communicate by sending and receiving messages is known as

(a) Pipe (b) Message Passing

(c) Shared Memory (d) None of Above

(iii) Which of the following is the Transport Layer Protocol

(a) HTTP (b) FTP

(c)UDP (d) SMTP

(iv) A low-level IPC mechanism for communication between processes over a network is provided by

(a) Sockets (b) Pipe

(c)Message (d) None of above

(v) Which operation is used by the client initiates communication with the server by establishing a connection.

(a) call (b) bind

(c) execute (d) exit

×××

UNIT: 3

REMOTE OBJECT INVOCATION AND DISTRIBUTED OBJECTS

Unit Structure:

3.1 Introduction

The world of computing is becoming increasingly interconnected. Gone are the days of standalone applications; today, programs often collaborate across networks, spanning multiple machines. This is the realm of distributed systems, where components work together to achieve a common goal, even if they're physically separated. This chapter introduces you to a powerful concept in distributed systems: **Remote Method Invocation (RMI)**. But before diving into RMI, we need to understand its key players i.e. **Distributed Objects**. These are special objects that reside on remote machines, yet can be accessed and interacted with just like local objects. Imagine a program on your computer seamlessly interacting with a database residing on a server across the network, that's the magic of distributed objects. RMI acts as the bridge between clients (programs initiating requests) and these remote objects. It allows clients to invoke methods on remote objects as if they were local, making development for distributed systems much simpler and more intuitive. This chapter will equip you with the knowledge of RMI and distributed objects. We'll explore how they work together, delve into different approaches to RMI, and discover how parameters are passed between clients and objects. Finally, we'll see how RMI is used in real-world applications, showcasing its power in building robust and scalable distributed systems. So, buckle up and get ready to unlock the potential of a distributed world.

3.2 Objectives

This unit is an attempt to equip you with a solid understanding of Remote Method Invocation (RMI) and its role in distributed systems. After going through this unit you will be able to-

- Understand how distributed objects enable resource sharing, location transparency, and scalability in distributed systems.
- Discover how RMI bridges this gap by providing a mechanism for transparent remote method invocation.
- Differentiate between static and dynamic approaches to RMI.
- Explain how parameters are transmitted during remote method invocations using RMI.
- Identify real-world applications of RMI in various distributed system scenarios.

3.3 Remote Object Invocation in Distributed Systems

3.3.1 Distributed System

Distributed system is a collection of autonomous computers cooperating to achieve a common goal through message passing. A distributed system isn't a single giant computer, but rather a network of independent computers called nodes. These nodes can be anything from laptops and desktops to powerful servers. Each node has its own CPU, memory, and operating system, and they can run programs independently. Since nodes are separate; they don't directly access each other's memory. Instead, they communicate by sending messages back and forth. These messages contain information and instructions that allow nodes to work together. Message passing protocols define how messages are formatted, sent, received, and handled by different nodes. One of the main benefits of distributed systems is the ability to share resources across multiple nodes. This can include hardware resources like storage and processing power, software resources like databases and applications, and even data itself. By sharing resources, distributed systems can handle larger workloads and provide services to more users. Ideally, a well-designed distributed system appears to the user as a single, coherent system. Users shouldn't need to be aware of the underlying complexity of the distributed architecture. They should be able to interact with the system as if it were a single computer, accessing resources and services seamlessly. Examples of distributed systems: World Wide Web, Cloud computing platforms (Google Cloud, Amazon Web Services), Cluster computing for scientific simulations, Multiplayer online games etc. Some key characteristics of distributed system are:

- It is a collection of independent computers (nodes).
- It communicates via message passing.
- Resource sharing across multiple nodes.
- Transparency of distribution (appears as a single system).
- The nodes are independent of one another and the failure of one does not impact the others.

The essential components of a distributed system are:

- **Nodes:** Individual computers or devices.
- **Network:** The communication backbone connecting nodes.
- **Middleware:** Software that facilitates communication and interaction between nodes.
- **Distributed Algorithms:** Protocols for coordination and decision-making.

A distributed system's design describes how several separate computers work together to accomplish a single objective. It includes defining how these nodes exchange information, distribute resources, and deal with errors. The architecture of a distributed system can be client-server, peer-to-peer and hybrid. Client-server architecture is a classic model where clients request services from servers. In peer-to-peer nodes can act as both clients and servers. Hybrid architecture combines elements of both client-server and peer-to-peer. Some Challenges of distributed system are:

- **Concurrency:** Managing multiple processes accessing shared resources.
- **Consistency:** Ensuring data integrity across multiple nodes.
- **Fault Tolerance:** Handling node failures and network partitions.
- **Security:** Protecting data and systems from unauthorized access.
- **Latency:** Dealing with communication delays between nodes.

STOP TO CONSIDER

A distributed system is a network of independent computers (nodes) that work together to appear as a single, unified system to users. These nodes communicate through message passing, sharing resources like hardware, software, and data.

Check Your Progress

Question1. What is a distributed system?
Question2. How do nodes communicate in a distributed system?
Question3. What is the significance of resource sharing in distributed systems?

3.3.2 Communication in Distributed System

➢ In order to comprehend the basic components of a distributed system, two questions must be taken into consideration: In a distributed system, what are the entities that are communicating?

➢ What paradigm of communication is employed, or more precisely, how do they communicate?

The answers to the two questions above are essential for understanding distributed systems; the distributed systems developer has a rich design space to choose from depending on what entities are communicating and how they communicate with one other. The answer is often extremely clear from a system perspective for **communicating entities** because distributed systems are typically made up of processes that communicate with one another. This has led to the usual understanding of distributed systems as processes combined with suitable inter process communication paradigms. In most distributed system environments, processes are augmented by threads; thus, threads are the endpoints of communication. In some primitive environments, such as sensor networks, the underlying operating systems may not support process abstractions, and hence the entities that communicate in such systems are nodes. This is adequate to describe a distributed system at a basic level. From the standpoint of programming, this is insufficient, and further problem-oriented abstractions have been suggested like objects and components.

Objects, which include both object-oriented design and object-oriented programming languages, were introduced to facilitate and promote the application of object-oriented techniques in distributed systems. A computation in distributed object-based techniques is made up of several interacting objects that serve as the problem domain's natural units of decomposition. Interfaces are used to access objects, and the methods defined on an object are specified by the interface description language (IDL) that is associated with the object. **Components** resemble objects in that they offer problem-oriented abstractions for building distributed systems and are also accessed through interfaces. The key difference is that components specify not only their interfaces but also the assumptions they make in terms of other components/interfaces that must be present for a component to fulfill its function – in other words, making all dependencies explicit and providing a more complete contract for system construction.

Communication paradigms, how entities communicate in a distributed system, commonly we have two types of communication paradigm: **interprocess communication** and **remote invocation**. Interprocess Communication in a distributed system is a process of exchanging data between two or more independent processes in a distributed environment.

Some types of interprocess communication (IPC) commonly used in distributed systems are:

- **Message Passing**: Message passing involves processes communicating by sending and receiving messages. Messages can be structured data packets containing information or commands. It is a versatile method suitable for both synchronous and asynchronous communication. Message passing can be implemented using various protocols such as TCP/IP, UDP, or higher-level messaging protocols like AMQP (Advanced Message Queuing Protocol) or MQTT (Message Queuing Telemetry Transport).
- **Remote Procedure Calls (RPC)**: RPC allows one process to invoke a procedure (or function) in another process, typically located on a different machine over a network. It abstracts the communication between processes by making it appear as if a local procedure call is being made. RPC frameworks handle details like parameter marshalling, network communication, and error handling.
- **Sockets**: Sockets provide a low-level interface for network communication between processes running on different computers. They allow processes to establish connections, send data streams (TCP) or datagrams (UDP), and receive responses. Sockets are fundamental for implementing higher-level communication protocols.
- **Message Queuing Systems**: Message queuing systems facilitate asynchronous communication by allowing processes to send messages to and receive messages from queues. They decouple producers (senders) and consumers (receivers) of messages, providing fault tolerance, scalability, and persistence of messages.

Remote invocation represents the most common communication paradigm in distributed systems, covering a range of techniques based on a two-way exchange between communicating entities in a distributed system and resulting in the calling of a remote operation, procedure or method.

STOP TO CONSIDER

Processes are often considered the primary communicating entities in distributed system along with threads, nodes, objects, and components.

Check Your Progress

Question1. What are the common entities that communicate in a distributed system?

Question2. Differentiate between interprocess communication and remote invocation.

Question3. What is the role of an interface in distributed object-based systems?

3.3.3 Distributed Objects

A distributed object is essentially a software component that resides on a remote machine but can be accessed and interacted with as if it were local. This abstraction is crucial for building complex, scalable, and flexible systems. The principles of encapsulation, inheritance and polymorphism remain the same as in traditional object-oriented programming. Data and methods are bundled together within the object, providing a clear interface and protecting internal state. However, in a distributed context, encapsulation becomes even more important as it helps to manage the complexity of distributed systems. Distributed objects can inherit properties and methods from other objects, just like their local counterparts. This promotes code reuse and promotes a hierarchical structure for object relationships. However, inheritance in distributed systems can introduce challenges related to object location and network communication. The ability of objects to take on multiple forms is equally valuable in distributed systems. It allows for flexible and dynamic interactions between objects, enabling different implementations of the same interface to be used interchangeably. **Distribution** is the core characteristic of distributed objects. They can reside on different machines, connected by a network. This enables load balancing, fault tolerance, and scalability. However, it also introduces challenges related to communication, synchronization, and consistency.

Properties of Distributed Object

- **Location Transparency:** Ideally, a distributed object should appear to be local to the client, regardless of its actual location. This simplifies development and improves system flexibility.

- **Concurrency:** Distributed objects often need to handle multiple concurrent requests. This requires careful synchronization and resource management to prevent data inconsistencies and race conditions.
- **Fault Tolerance:** Distributed systems are inherently prone to failures. Distributed objects should be designed with fault tolerance in mind, using techniques like replication, redundancy, and error handling.
- **Security:** Protecting distributed objects and their data is crucial. This involves authentication, authorization, encryption, and other security measures to prevent unauthorized access and data breaches.

Role of Distributed Object in Distributed Computing

- **Modularity:** By breaking down a system into distributed objects, you create smaller, more manageable components that can be developed, tested, and deployed independently. This improves code maintainability and facilitates collaboration among development teams.
- **Flexibility:** Distributed objects can be replaced or upgraded without affecting the entire system, as long as they adhere to the same interface. This allows for incremental improvements and adaptation to changing requirements.
- **Scalability:** Distributed objects can be deployed on multiple machines to handle increasing workloads. This enables systems to grow gracefully as demand increases, without compromising performance.
- **Reusability:** Well-designed distributed objects can be reused in different applications, reducing development time and effort. This promotes code reuse and improves overall system efficiency.

Distributed objects are applicable in a wide range of domains, including **e-commerce, enterprise applications, cloud computing, real-time systems etc**.

STOP TO CONSIDER

Distributed objects are software components that can be accessed remotely as if they were local. They offer encapsulation, inheritance, polymorphism, and distribution as core properties

Check Your Progress

Question1. What are the core properties of a distributed object?
Question2. In what domains are distributed objects commonly used?
Question3. What are the challenges in managing distributed objects?

3.3.4 Remote Method Invocation

Remote Method Invocation (RMI) is a mechanism that allows an object on one machine to invoke a method on an object located on another machine. This enables distributed computing, where applications can be divided into components that run on different systems, communicating and collaborating seamlessly. To understand how RMI works, imagine you have two computers, Computer A and Computer B. Computer A has a program with a method that Computer B wants to use as shown in figure 1. With RMI, Computer B can call that method on Computer A as if it were a local method.

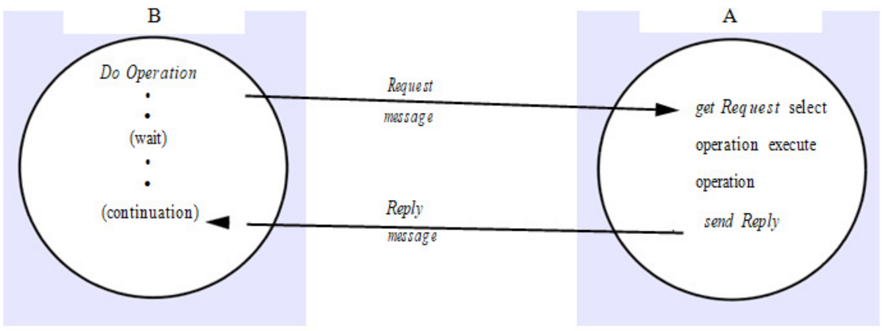


Figure 1. Request reply communication between B and A

This remote call is managed by the RMI system, which handles the communication details, making the process seamless for the programmer. Essentially, RMI allows methods to be executed on a remote server, and the results are returned to the client.RMI system architecture typically involves the following components:

- **Client:** The initiating process that makes the remote method call.
- **Stub:** A local proxy object on the client side that represents the remote object. The client interacts with the stub as if it were the actual object.

- **Skeleton:** A local helper object on the server side that receives incoming requests, unmarshals parameters, and forwards them to the actual remote object.
- **Registry:** A service that maps object names to their network addresses.

Working of RMI:
In the client side**,** the client obtains a reference to the remote object's stub. The client then invokes a method on the stub and the stub marshals (serializes) the method arguments and sends them to the server. On the server-side, the skeleton receives the request, unmarshals the arguments, and invokes the corresponding method on the remote object. The remote object executes the method and returns the result to the skeleton. The skeleton then marshals the result and sends it back to the client. The client side stub receives the result from the server, unmarshals it, and returns it to the client application.

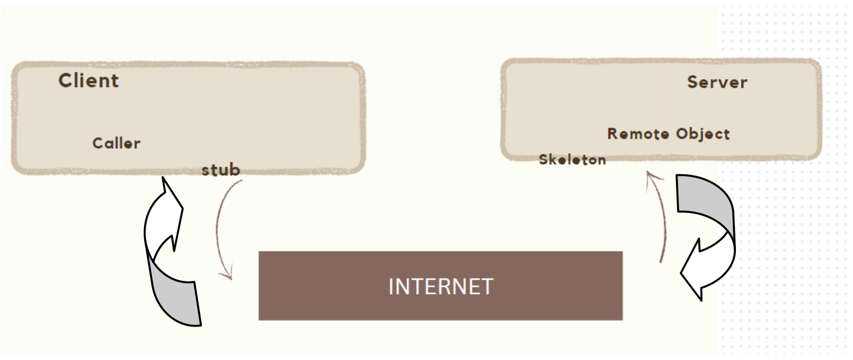


Figure 2: Working of RMI

RMI systems face significant security challenges. Ensuring robust security is paramount for protecting sensitive data and maintaining system integrity. This necessitates implementing measures for verifying the **authenticity** of both clients and servers to prevent unauthorized access. Additionally, controlling access to specific system functions based on user privileges is crucial. To safeguard data transmitted over the network, robust **encryption** mechanisms are essential. Maintaining **data integrity** during transmission is another critical aspect, requiring measures to detect and prevent tampering. Finally, **non-repudiation** mechanisms are necessary to prevent parties from denying their actions, ensuring accountability and trust within the system.

STOP TO CONSIDER

Remote Method Invocation (RMI) is a mechanism that allows objects to communicate and interact across different machines. It involves a client, stub, skeleton, and registry.

Check Your Progress

Question1. What is the role of a stub in RMI?
Question2. What is the difference between a stub and a skeleton in RMI?
Question3. What are the primary security concerns in RMI?

3.4 Integrating Clients and Objects in a Distributed Environment

3.4.1 Client-Server Model in Distributed Systems

The client-server model is the cornerstone of distributed systems, dividing applications into two distinct roles: clients and servers. Clients, typically the user interface or application logic, handle user interaction and initiate requests. Servers, the workhorses of the system, manage shared data and resources, responding to client requests and providing services. This distributed processing approach offers several key advantages. Firstly, tasks are split between client and server, allowing for efficient resource allocation. The client focuses on user experience while the server handles complex calculations and data management. Secondly, network communication becomes the backbone of interaction. Clients and servers can reside on separate machines, enabling scalability – adding more servers can handle increased workloads. Finally, the model offers flexibility, adapting to diverse application domains. Whether it's web servers delivering content, email servers managing communication, or database servers storing data, the client-server model underpins a vast range of distributed systems.

STOP TO CONSIDER

The client-server model is a fundamental architecture for distributed systems that divides applications into clients and servers, offering advantages in task distribution, scalability, and adaptability.

Check Your Progress

Question1. What are the two main components of a client-server model?
Question2. Give an example of a client-server application.
Question3. What is the role of a server in a client-server model?
Question4. What is the role of a client in a client-server model?

3.4.2 Object-Oriented design for Distributed Systems

Object-oriented design (OOD) principles are fundamental to constructing efficient and adaptable distributed systems. By encapsulating data and behavior within objects, OOD promotes modularity, making code more manageable and maintainable. Inheritance facilitates code reuse, accelerating development and reducing redundancy. Polymorphism allows objects to take on various forms, enhancing system flexibility and adaptability to changing requirements. To achieve seamless integration, distribution transparency is crucial, masking the complexities of remote object interaction. OOD also supports concurrency, enabling objects to handle multiple requests simultaneously, improving system responsiveness. Furthermore, fault tolerance, a cornerstone of reliable distributed systems, can be enhanced through OOD by encapsulating error handling and recovery mechanisms within objects. However, challenges persist. Managing the distribution of objects, including their location and state, requires careful consideration to ensure consistency and availability. Network latency and bandwidth limitations can impact performance, necessitating efficient communication protocols and object design. Additionally, robust security measures are essential to protect sensitive data and prevent unauthorized access. By effectively addressing these challenges, developers can leverage OOD to build scalable, resilient, and secure distributed systems.

STOP TO CONSIDER

Object-Oriented Design (OOD) is a powerful approach for building distributed systems. By encapsulating data and behavior, using inheritance and polymorphism, OOD promotes modularity, reusability, and flexibility.

Check Your Progress

Question1. What are the key OOD principles beneficial for distributed systems?

Question2. What is the importance of distribution transparency in OOD for distributed systems?

Question3. What are the primary challenges of applying OOD to distributed systems?

3.4.3 Distributed Object Naming and Discovery Services

In distributed systems, locating and accessing objects across different machines is a fundamental challenge. To address this, naming and discovery services play a crucial role. **Naming services** act as a directory or mapping system that associates human-readable names with the network addresses of objects. This abstraction layer simplifies the process of referencing objects, as users can employ meaningful names instead of complex network locations. Examples of naming services include the Domain Name System (DNS), which maps domain names to IP addresses, and the Lightweight Directory Access Protocol (LDAP), which provides a hierarchical structure for storing and accessing directory information.

Discovery services go beyond simple name-to-address mapping by enabling the search for objects based on attributes or services they provide. This dynamic approach allows for flexible and adaptable system designs. Unlike naming services, which rely on predefined names, discovery services facilitate finding objects based on their characteristics. For instance, a discovery service might locate all printers available on a network or identify services that meet specific criteria.

The core functionalities of naming and discovery services encompass object registration, lookup, and binding. **Object registration** involves adding objects to the service's database, associating them with their respective names or attributes. **Object lookup** enables clients to find objects by querying the service with a name or specific criteria. Once an object is located, **object binding** establishes a communication channel between the client and the object, facilitating interaction and data exchange. By providing efficient mechanisms for locating and accessing distributed objects, naming and discovery services are essential components of modern distributed systems. They contribute to system scalability, flexibility, and ease of management by abstracting away the complexities of network-based object interactions.

STOP TO CONSIDER

Naming and discovery services are essential components of distributed systems that facilitate the location and access of objects across different machines.

Self Asking Questions

How does object binding contribute to the overall functionality of distributed systems? (50 words)

…………………………………………………………………
…………………………………………………………………
…………………………………………………………………
…………………………………………………………………

3.4.4 Object Lifecycle Management in Distributed Environments

Object lifecycle management is a critical aspect of distributed systems, encompassing the entire journey of an object from creation to destruction. It involves a complex interplay of various operations and considerations.

Object Creation and Activation: The lifecycle of a distributed object begins with its creation. This involves instantiating the object on a suitable remote machine. However, not all objects need to be actively running at all times. To optimize resource utilization, inactive objects can be placed in a suspended state. When required, these objects can be activated, bringing them back to a ready state to serve client requests. This process, known as object activation, involves loading the object's state from persistent storage and initializing it.

Object Passivation and Persistence: To conserve system resources and handle failures gracefully, objects can be passivated. This involves saving the object's state to persistent storage and temporarily removing it from memory. Passivation is essential for objects that are infrequently accessed or have large memory footprints. Object persistence, on the other hand,

is concerned with storing object data for extended periods, independent of the object's runtime state. This is crucial for data that needs to be preserved even after the system is shut down.
Object Migration: In dynamic distributed environments, it may be necessary to move objects between different machines. Object migration involves transferring an object's state and identity to a new location. This can be done for various reasons, such as load balancing, fault tolerance, or data locality. However, migrating objects can be complex and requires careful coordination to avoid data inconsistencies and disruptions.

Object Garbage Collection: Similar to traditional programming languages, distributed systems need mechanisms to reclaim resources occupied by objects that are no longer in use. Object garbage collection identifies and removes these objects to prevent memory leaks and improve performance. In distributed environments, garbage collection becomes more challenging due to the distributed nature of objects and the potential for network partitions.

Managing the lifecycle of distributed objects presents several challenges. **Coordination** is crucial to ensure consistency across multiple nodes. For example, updating an object's state on one node while other nodes have outdated information can lead to inconsistencies. **Fault tolerance** is another critical aspect. Objects and their associated data must be protected from failures, and the system should be able to recover from such events. **Performance** is also a significant concern. Object lifecycle operations, such as activation, passivation, and migration, should be optimized to minimize overhead and maximize system responsiveness. To address the challenges of object lifecycle management, several key concepts are employed. **Object persistence** ensures that object data survives system failures and restarts. **Object replication** creates multiple copies of an object to enhance availability and performance. By distributing object data across different nodes, the system can tolerate failures and provide faster access to data. **Object leasing** is a mechanism for controlling object lifetimes. Objects are granted leases, which specify a time period during which they are considered active. If an object is not renewed before the lease expires, it can be passivated or garbage collected.

Effective object lifecycle management is essential for building reliable, scalable, and efficient distributed systems. By carefully considering the various factors involved and employing appropriate techniques, developers can manage the complex lifecycle of distributed objects and optimize system performance.

Self Asking Questions

How does object migration impact system performance? (50 words)

…………………………………………………………………
…………………………………………………………………
…………………………………………………………………
…………………………………………………………………

3.5 STATIC VERSUS DYNAMIC REMOTE METHOD INVOCATION

3.5.1 Static Remote Method Invocation

Static Remote Method Invocation, also known as compiled RMI is the traditional approach where method calls are resolved at compile time. In this method, the client-side code is aware of the remote interfaces and the methods they provide. This means that the method signatures are fixed and checked during the compilation of the client application. Static RMI is straightforward and offers performance advantages because the method calls are resolved at compile time, reducing the overhead during runtime.

Characteristics of Static RMI:

Compile-Time Binding: In static RMI, method invocations are bound to their respective implementations during the compile time. This results in faster execution as the method calls do not need to be resolved at runtime.

Predictable Performance: Since the method signatures are known in advance, the performance is more predictable and generally better compared to dynamic RMI.

Less Flexibility: Static RMI can be less flexible as any change in method signatures requires recompiling the client application. This can lead to higher maintenance efforts when changes are made to the remote interfaces.

3.5.2 Dynamic Remote Method Invocation

Dynamic Remote Method Invocation, on the other hand, involves resolving method calls at runtime. In this approach, the client does not have fixed knowledge of the method signatures in advance. Instead, the method calls are dynamically resolved using reflection or similar mechanisms. This allows for greater flexibility as the client application does not need to be recompiled when remote interfaces change.

Characteristics of Dynamic RMI:

Runtime Binding: Methods are bound at runtime, which provides flexibility as the client application does not need to be updated when remote interfaces change. This makes dynamic RMI suitable for environments where interfaces evolve frequently.

Increased Overhead: The runtime binding introduces additional overhead because method calls must be resolved dynamically, which can affect performance compared to static RMI.

Greater Flexibility: Dynamic RMI is more adaptable to changes in the remote interfaces. This can simplify development and maintenance, especially in complex distributed systems where interfaces are subject to frequent changes.

STOP TO CONSIDER

Static RMI involves pre-compiled method calls, offering performance benefits but limited flexibility. **Dynamic RMI** resolves method calls at runtime, providing greater flexibility but potentially sacrificing performance. The choice between the two depends on the specific requirements of the distributed system, such as performance needs, frequency of interface changes, and development complexity.

Both static and dynamic Remote Method Invocation approaches have their advantages and trade-offs. Static RMI offers better performance and simplicity when method signatures are stable, while dynamic RMI provides flexibility and adaptability in environments where remote interfaces are frequently updated. The choice between static and dynamic RMI depends on the specific requirements of the distributed application and the nature of the remote interactions.

Check Your Progress

Question1. What is the primary difference between static and dynamic RMI?

Question2. How does flexibility differ between static and dynamic RMI?

3.6 Parameter Passing in RMI

Parameter passing is a fundamental aspect of remote method invocation. It involves transferring data between the client and server during method calls. Understanding how different data types are handled and the implications of passing by value or reference is crucial for effective RMI programming.

RMI supports a wide range of data types for parameter passing, including:

- **Primitive data types:** These include basic data types like int, double, boolean, etc. They are passed by value, meaning a copy of the value is sent to the remote method.

- **Strings:** Strings are treated as immutable objects in Java and are passed by reference. However, since strings are immutable, the remote method cannot modify the original string.

- **Arrays:** Arrays can be passed as parameters. The behavior depends on the element type. If the elements are primitive types, the array is passed by value, meaning a copy of the array is sent. If the elements are objects, the array is passed by reference, but the objects themselves are still passed by value.

- **Objects:** Objects can be passed as parameters, but they are passed by reference. This means that the remote method receives a reference to the same object that exists on the client side. Any changes made to the object on the server side will be reflected on the client side.

Passing Primitives and Objects by Value and Reference

Understanding the difference between passing by value and passing by reference is essential in RMI. In **passing by value,** a copy of the data is sent to the remote method. Any modifications made to the data on the server side do not affect the original data on the client side. This is the case for primitive data types and arrays of primitive data types. In case of **passing by reference,** a reference to the original object is sent to the remote method. Any changes made to the object on the server side will be reflected on the client side. This is the case for objects and arrays of objects. It's important to note that while objects are passed by reference in RMI, the object itself is still serialized and deserialized during the transfer. This means that a copy of the object's state is sent to the server, but the reference to the original object is retained on the client side. Object serialization is the process of converting an object's state into a byte stream, which can be transmitted over the network. Deserialization is the reverse process, reconstructing the object from the byte stream. RMI uses object serialization and deserialization to transfer objects between the client and server. For an object to be serializable, it must implement the Serializable interface. This interface is a marker interface that indicates that the object's state can be serialized. When an object is passed as a parameter or returned from a remote method, it is automatically serialized and sent over the network. The object is then deserialized on the receiving end to recreate the object. It's important to consider the performance implications of object serialization and deserialization. Large objects can take significant time to serialize and deserialize, affecting overall RMI performance. Several factors influence the performance of parameter passing in RMI, **Data type;** primitive data types are generally faster to pass than objects. **Object size;** larger objects take longer to serialize and deserialize. **Network latency;** the time it takes for data to travel between the client and server affects overall performance. **Serialization/deserialization overhead;** the process of converting objects to byte streams and vice versa adds overhead. To optimize performance, it's essential to carefully consider the data types used in parameter passing, minimize object size, and use efficient serialization techniques.

STOP TO CONSIDER

Parameter passing in RMI involves transferring data between client and server. Primitive data types are passed by value, while objects are passed by reference, though the object itself is serialized and deserialized.

Check Your Progress

Question1. What is the difference between passing a primitive and an object as a parameter in RMI?

Question2. Can you modify an object passed by reference in a remote method?

Self Asking Questions

How does the choice of data types impact the overall efficiency of an RMI application? (80 words)

…………………………………………………………………………
…………………………………………………………………………
…………………………………………………………………………
…………………………………………………………………………

3.7 Examples of RMI Usage in Distributed Systems.

RMI has found widespread application in various domains due to its ability to distribute functionalities across multiple systems. **E-commerce;** RMI is extensively used in e-commerce applications for managing product catalogs, processing orders, and handling inventory management. For instance, a distributed e-commerce system might have separate servers for product information, order processing, and payment gateways, all communicating through RMI. **Distributed Databases;** RMI plays a crucial role in distributed database systems by enabling data access and manipulation across multiple nodes. It can be used for load balancing, data replication, and fault tolerance. For example, a distributed database system might employ RMI to distribute query processing among multiple servers. In other applications such as **financial systems;** for real-time stock quotes, trading, and risk management, **Collaborative applications;** for sharing documents, whiteboarding, and real-time communication, **Enterprise resource planning (ERP) systems;** for integrating different modules and managing complex business processes use RMI.

RMI is a form of middleware, which is software that facilitates communication and interaction between different software components. It provides a layer of abstraction over underlying network protocols and operating systems. **CORBA (Common Object Request Broker Architecture)** is a more complex and heavyweight middleware platform compared to RMI, CORBA offers a broader range of features, including language interoperability and distributed transaction support. However, it is also more complex to implement and deploy. **DCOM (Distributed Component Object Model)** is a Microsoft-specific technology, similar to RMI but is tightly integrated with the Windows operating system. While it offers some advantages in terms of performance and integration with other Microsoft technologies, it is less portable than RMI.

RMI, while powerful, faces certain challenges in terms of performance, security, reliability and complexity. Network latency and serialization/deserialization overhead can impact performance. RMI inherently involves network communication, which introduces latency. This can significantly impact the perceived performance of applications, especially for operations that require frequent remote method invocations. The process of converting objects into byte streams for transmission (serialization) and reconstructing them at the receiving end (deserialization) adds overhead. Large objects or complex data structures can exacerbate this issue. Protecting data and preventing unauthorized access is crucial in RMI. Verifying the identity of clients and servers is crucial to prevent unauthorized access. Weak authentication mechanisms can lead to security breaches. Controlling access to specific methods or resources based on user privileges is essential to protect sensitive data.

Protecting sensitive data transmitted over the network requires encryption and other security measures to prevent eavesdropping. Ensuring that data is not tampered with during transmission is vital for maintaining data integrity. RMI systems can be vulnerable to Denial of Service attacks, which can render the system unresponsive. Ensuring fault tolerance and recovery is essential for mission-critical applications. RMI systems must be able to handle failures such as network outages, server crashes, or unexpected exceptions. Implementing mechanisms for detecting and recovering from failures is crucial. Coordinating transactions across multiple nodes can be complex and error-prone. Ensuring data consistency and atomicity in distributed environments is challenging. Developing and managing distributed systems using RMI can be complex. Developing distributed applications using RMI can be more complex than traditional monolithic applications. Debugging issues across multiple machines can be time-consuming. Deploying and managing distributed systems requires careful planning and coordination. Scaling and maintaining RMI-based systems can be challenging.

Example of RMI Usage in Distributed Systems

An example shared whiteboard application to illustrate RMI. This application allows users to draw shapes on a shared canvas, with a server maintaining the state of the drawing.

Interfaces:
Shape: Represents a drawable shape on the whiteboard.
ShapeList: Manages a collection of Shape objects.

```
import java.rmi.*;
import java.util.Vector;

public interface Shape extends Remote {
   int getVersion() throws RemoteException;
   GraphicalObject getAllState() throws RemoteException;
}

public interface ShapeList extends Remote {
   Shape newShape(GraphicalObject g) throws
RemoteException;
   Vector<Shape> allShapes() throws RemoteException;
```

```
    int getVersion() throws RemoteException;
}
```

In this example:

Shape interface has methods to get the version and state of a shape.

ShapeList interface has methods to add new shapes, retrieve all shapes, and get version information.

Implementation Details

Serialization: GraphicalObject must implement Serializable to allow its instances to be passed by value.

Method Invocation: The newShape method in ShapeList accepts a GraphicalObject and returns a Shape reference. The getAllState method in Shape returns a GraphicalObject by value.

Practical Considerations

Concurrency: When designing RMI applications, consider thread safety and concurrency, as remote objects might be accessed by multiple clients simultaneously.

Exceptions: Remote method calls must handle Remote Exception to manage communication failures.

STOP TO CONSIDER

RMI is a powerful tool for building distributed systems, enabling communication and interaction between objects on different machines. It has found applications in various domains, such as e-commerce, distributed databases, and financial systems.

Check Your Progress

Question1. What are the main advantages of using RMI in distributed systems?

Question2. What are the common performance challenges in RMI?

Question3. What is the difference between RMI and CORBA?

3.8 Summing Up

1. **Distributed systems** involve multiple computers working together to achieve a common goal.
2. **Remote Method Invocation (RMI)** enables objects on different machines to interact as if they were local.
3. **Key components** of RMI include client, stub, skeleton, and registry.
4. **Distributed objects** are the foundation of RMI, allowing for location transparency and object-oriented programming in distributed environments.
5. **Client-server model** is a common architecture for distributed systems, with clients requesting services from servers.
6. **Object-oriented design (OOD)** principles enhance distributed system development through encapsulation, inheritance, polymorphism, and distribution transparency.
7. **Naming and discovery services** help locate distributed objects.
8. **Object lifecycle management** includes creation, activation, passivation, migration, and garbage collection.
9. **Static RMI** involves compile-time binding, offering performance but less flexibility.
10. **Dynamic RMI** uses runtime binding, providing flexibility but potentially lower performance.
11. **Primitive data types** are passed by value in RMI.
12. **Objects** are passed by reference, but the object itself is serialized and deserialized.
13. RMI faces challenges in performance, security, reliability, and complexity.

3.9 References and Suggested Readings

1. Coulouris, G., Dollimore, J., Kindberg, T., Blair, G., (2012). *Distributed Systems Concepts and Design* (Fifth Edition).Pearson.
2. https://www.geeksforgeeks.org/remote-method-invocation-in-java/

3.10 Model Questions

1. Define a distributed system.
2. What are the key characteristics of a distributed system?

3. Differentiate between a distributed system and a parallel system.
4. Explain the concept of transparency in distributed systems.
5. What are the common challenges in building distributed systems?
6. Describe the role of middleware in distributed systems.
7. What is the significance of fault tolerance in distributed systems?
8. Define Remote Method Invocation (RMI).
9. What are the core components of an RMI system?
10. Explain the role of a stub in RMI.
11. Differentiate between a stub and a skeleton.
12. How does object serialization work in RMI?
13. What are the advantages of using RMI for distributed computing?
14. What are the performance implications of passing large objects in RMI?
15. Explain the concept of distributed objects.
16. How does object-oriented design support distributed systems?
17. What is the role of a naming service in distributed systems?
18. Describe the client-server model.
19. What are the key components of a client-server architecture?
20. What are the challenges of managing distributed objects?
21. How does object replication improve system availability?
22. Explain the concept of object leasing.
23. What is the role of garbage collection in distributed systems?
24. What are the common security threats in RMI?
25. How can you ensure data integrity in RMI?
26. Explain the concept of distributed transactions.
27. What are the performance optimization techniques for RMI?
28. How does RMI compare to other distributed computing technologies like CORBA and DCOM?

29. Discuss the role of middleware in RMI.
30. Differentiate between static and dynamic RMI.
31. When would you choose static RMI over dynamic RMI?
32. What are the performance implications of static and dynamic RMI?
33. How does flexibility differ between static and dynamic RMI?
34. Discuss the use cases for static and dynamic RMI.
35. Explain the difference between passing by value and passing by reference in RMI.
36. How are primitive data types handled in RMI?
37. What are the performance implications of object serialization in RMI?
38. How can you optimize parameter passing in RMI?
39. What are the potential security risks associated with parameter passing in RMI?

3.11 Answer to check your progress/Possible Answers to SAQ

What is a distributed system?

- A distributed system is a collection of independent computer systems that communicate and coordinate their actions to appear as a single, coherent system to the user. These systems are geographically dispersed and connected through a network. They work together to achieve a common goal by sharing resources and processing tasks.

How do nodes communicate in a distributed system?

- Nodes in a distributed system communicate through various methods. Nodes exchange data by sending and receiving messages. Nodes can communicate using remote **procedure calls.** One node can invoke a procedure on another node as if it were a local call. **Sockets,** a lower-level mechanism for network communication, providing direct control over data transmission. Using **message queuing systems** also nodes can communicate for asynchronous communication.

What is the significance of resource sharing in distributed systems?

- Resource sharing is a cornerstone of distributed systems. It **increases efficiency** by distributing workloads across multiple nodes; systems can handle larger workloads and improve performance. If one node fails, other nodes can take over its tasks, preventing system failure. Systems can grow by adding more nodes, increasing capacity and handling increased demand. By sharing resources, organizations can optimize hardware and software utilization. Data can be replicated across multiple nodes for improved accessibility and reliability.

What are the common entities that communicate in a distributed system?

- In distributed systems, the primary entities that communicate are **processes** or **threads**. These are the fundamental units of execution within a system. While processes are heavier-weight entities with their own memory space, threads are lighter-weight and share the same memory space within a process. Both can serve as endpoints for communication in a distributed environment.

Differentiate between interprocess communication and remote invocation.

- **Interprocess Communication (IPC)** is a general term for any mechanism that allows processes to exchange data and synchronize their actions. It encompasses a wide range of techniques, including message passing, shared memory, and pipes. IPC can occur within a single system or across multiple systems. **Remote Invocation** is a specific type of IPC where a process on one machine invokes a procedure or method on another machine. It provides a higher-level abstraction than raw IPC, making it easier to interact with remote components. Examples include Remote Procedure Calls (RPC) and Remote Method Invocation (RMI).

What is the role of an interface in distributed object-based systems?

- In distributed object-based systems, an interface defines the contract between a client and a distributed object. It specifies the methods that a client can invoke on the object, along with their parameters and return types. The interface acts as a blueprint for interaction, promoting modularity, encapsulation, and code reusability. By defining the interface separately from the object's implementation, it allows for different implementations of the same interface to be used interchangeably, enhancing flexibility and maintainability.

What are the core properties of a distributed object?

- The core properties of a distributed object are Location transparency, Concurrency, Fault tolerance and Security.

In what domains are distributed objects commonly used?

- E-commerce, enterprise applications, cloud computing, real-time systems etc**.**

What are the challenges in managing distributed objects?

- Managing distributed objects presents several challenges:
    - Ensuring consistency and synchronization of object state across multiple nodes.
    - Handling failures and recovering object state.
    - Optimizing object access and communication to achieve acceptable performance.
    - Protecting distributed objects from unauthorized access and data breaches.
    - Dealing with network partitions and ensuring system resilience.
    - Managing objects on different platforms and with different programming languages.

What is the role of a stub in RMI?

- A stub is a local proxy object on the client side that represents a remote object. It acts as a placeholder for the actual remote object. When a client invokes a method on the stub, the stub marshals the method arguments and sends them to the server-side skeleton. It then waits for the response from the server and returns the result to the client. Essentially, the stub hides the network communication details from the client, making it appear as if the method is being invoked locally.

What is the difference between a stub and a skeleton in RMI?

- Stub Resides on the client side. Acts as a local representative of the remote object. Marshals method arguments and sends them to the server. Skeleton resides on the server side. Receives incoming requests, unmarshals parameters, and forwards them to the actual remote object. Returns the result back to the client through the stub.

What are the primary security concerns in RMI?

- The primary security concerns in RMI includes
  - **Authentication:** Verifying the identity of clients and servers to prevent unauthorized access.
  - **Authorization:** Controlling access to specific methods or resources based on user privileges.
  - **Confidentiality:** Protecting sensitive data transmitted over the network using encryption.
  - **Integrity:** Ensuring data is not tampered with during transmission.
  - **Non-repudiation:** Preventing parties from denying their actions.
  - **Denial of Service (DoS) attacks:** Protecting against malicious attempts to overload the system.

What are the two main components of a client-server model?

- **Client and server** are the two main components of a client-server model.

Give an example of a client-server application.

- **Email** is a common example of a client-server application.

What is the role of a server in a client-server model?

- The **server** in a client-server model is responsible for:
  - Managing shared resources and data
  - Providing services to clients
  - Responding to client requests
  - Storing and processing information

What is the role of a client in a client-server model?

- The **client** in a client-server model is responsible for:
  - Initiating requests to the server
  - Interacting with the user
  - Displaying information received from the server
  - Sending data to the server for processing

What are the key OOD principles beneficial for distributed systems?

- The key OOD principles that are particularly beneficial for distributed systems are:
  - **Encapsulation:** This helps to create well-defined, modular components that can be distributed across different systems.
  - **Inheritance:** Supports code reuse and the creation of hierarchical relationships between distributed objects.
  - **Polymorphism:** Enables flexible interactions between distributed objects, allowing for different implementations of the same interface.
  - **Distribution transparency:** This principle aims to hide the complexities of distributed computing, making remote objects appear as local ones.

What is the importance of distribution transparency in OOD for distributed systems?

- Distribution transparency is crucial in OOD for distributed systems as by making remote objects appear local, developers can focus on application logic rather than network communication details. As the system grows, new components can be added without significantly impacting existing code. Changes to the underlying distribution infrastructure are less likely to affect the application which improves maintainability**.**

What are the primary challenges of applying OOD to distributed systems?

- The primary challenges of applying OOD to distributed systems include:
  - Tracking object locations, states, and lifecycles across multiple systems.
  - Dealing with network latency, bandwidth limitations, and potential failures.
  - Managing concurrent access to shared objects and preventing data inconsistencies.
  - Protecting distributed objects and their data from unauthorized access.
  - Designing systems that can continue to function in the face of failures.
  - Dealing with different hardware, software, and network environments.

How does object binding contribute to the overall functionality of distributed systems?

- **Object binding** is the crucial step that transforms a theoretical connection between a client and a remote object into a functional interaction. It establishes a communication channel, allowing data and method calls to be exchanged. Without binding, even if an object is discovered, it remains inaccessible. This process ensures seamless interaction between distributed components, enabling tasks like remote procedure calls, data transfer, and synchronization. In essence, object binding is the bridge that connects the dots in distributed systems, making them operational and effective.

How does object migration impact system performance?

- **Object migration** can significantly impact both system performance and consistency. It can improve performance by moving objects closer to their frequently accessing clients, reducing network latency. However, the migration process itself can be resource-intensive, temporarily impacting system performance.

What is the primary difference between static and dynamic RMI?

- The primary difference between static and dynamic RMI lies in the timing of method resolution. In static RMI, method calls are resolved at compile time, while in dynamic RMI, they are resolved at runtime. This fundamental difference impacts performance, flexibility, and development complexity.

How does flexibility differ between static and dynamic RMI?

- Dynamic RMI offers significantly more flexibility than static RMI. In static RMI, the structure of remote objects is fixed at compile time, requiring changes to the client code if the remote interface evolves. In contrast, dynamic RMI allows clients to adapt to changes in remote interfaces at runtime without recompilation, making it more suitable for systems with frequently changing requirements.

What is the difference between passing a primitive and an object as a parameter in RMI?

- **Primitives are passed by value**, meaning a copy of the primitive value is sent to the remote method. Any modifications made to the primitive within the remote method do not affect the original value on the client side. **Objects are passed by reference**, meaning a copy of the object reference is sent to the remote method. While this might seem like pass-by-reference behavior, it's important to note that the object itself is serialized and sent over the network. Any changes made to the object's state on the server side will be reflected on the client side when the object is deserialized back.

Can you modify an object passed by reference in a remote method?

- When an object is passed by reference in RMI, the remote method receives a reference to the same object that exists on the client side. Any changes made to the object's state within the remote method will be reflected in the original object on the client side when the method returns.

How does the choice of data types impact the overall efficiency of an RMI application?

- The choice of data types significantly influences the efficiency of an RMI application. **Primitive Data Types:** Generally more efficient as they are passed by value, requiring less overhead compared to objects. **Objects:** Less efficient due to the overhead of serialization and deserialization. Larger objects can significantly impact performance, especially over slower networks. Complex object hierarchies can further increase overhead. **Arrays:** Efficiency depends on the element type. Primitive arrays are generally more efficient than object arrays. Large arrays can impact performance due to their size.

What are the main advantages of using RMI in distributed systems?

- RMI aligns seamlessly with object-oriented programming principles, making it natural to distribute object-based applications. Compared to other distributed computing technologies, RMI provides a relatively straightforward approach for developing distributed applications, especially within the Java ecosystem. RMI supports communication between Java applications running on different platforms, enhancing portability. Automatic garbage collection simplifies memory management in distributed environments.

What are the common performance challenges in RMI?

- The time taken for data to travel between machines can significantly impact performance, especially for frequent interactions. Converting objects to byte streams and vice versa for network transmission incurs processing costs. The garbage collection process can sometimes interfere with application performance, particularly in high-throughput systems.

What is the difference between RMI and CORBA?

- RMI and CORBA are both middleware technologies for distributed computing, but they have key differences:
  RMI is primarily designed for Java, while CORBA

supports multiple programming languages. RMI is generally simpler to use and implement compared to CORBA, which offers a more complex and feature-rich approach. RMI often offers better performance than CORBA for simpler applications due to its focus on Java and reduced overhead.

×××

UNIT: 4

NAMING ENTITIES AND DOMAIN NAME SYSTEM (DNS)

Unit Structure:

4.1 Introduction

Naming entities and managing the Domain Name System (DNS) are crucial aspects of designing and maintaining distributed systems. This abstract explores the significance of naming entities in distributed environments and the pivotal role of DNS in translating human-readable domain names into machine-readable IP addresses.

Naming entities in distributed systems involves assigning unique identifiers to components, services, and resources to facilitate communication and interaction. With the dynamic nature of distributed environments, challenges such as scalability, consistency, and fault tolerance arise. Effective naming schemes provide a foundation for seamless communication, enabling components to discover and interact with each other transparently.

Naming entities in distributed systems involves assigning unique identifiers to various components, services, and resources within the system. A well-designed naming scheme simplifies communication, enables resource discovery, and facilitates scalability and flexibility in distributed environments.

In distributed systems, naming entities involves assigning unique identifiers to various components, services, and resources. Challenges such as scalability, consistency, and fault tolerance arise due to the dynamic nature of distributed environments. Effective naming schemes simplify communication, enable resource discovery, and support scalability and flexibility.

The Domain Name System (DNS) serves as the backbone of the Internet's naming infrastructure, providing a hierarchical naming structure and facilitating name resolution. DNS operates through a distributed system of servers, including authoritative name servers and recursive resolvers, to translate domain names into IP addresses and vice versa.

Advanced topics in DNS, such as DNS Security (DNSSEC), Anycast DNS, and Content Delivery Networks (CDNs), enhance the security, performance, and resilience of distributed systems. DNSSEC adds cryptographic signatures to DNS records to prevent tampering and spoofing, while Anycast DNS and CDNs optimize routing and content delivery.

This introduction sets the stage for exploring the intricacies of naming entities and DNS management in distributed systems. Throughout this discussion, we will delve into the challenges of naming entities, the principles of DNS operation, advanced topics in DNS management, and best practices for designing robust and scalable distributed systems. By understanding the significance of naming entities and leveraging DNS effectively, organizations can build resilient and efficient distributed systems capable of meeting the demands of modern computing environments.

In a distributed system, naming entities and managing them efficiently is crucial for seamless communication and interaction between distributed components. This chapter delves into the importance of naming entities in distributed systems and explores the Domain Name System (DNS) as a fundamental component for translating human-readable names into machine-readable IP addresses.

4.2 Objectives

After going through this unit you will be able to:

- Understand the basic concepts of Naming entities and managing the Domain Name System (DNS)
- Know about the importance of Naming entities and DNS.
- Know about Uniform Resource Names (URNs) and its Importance
- Know about Directory Services and Distributed Name Services
- Understand about Domain Name System (DNS) and its Architecture and Hierarchy
- Know about Resource Records (RRs), Name Resolution Process of DNS

- Idea about DNS Spoofing, Cache Poisoning and DNS Anycast.

4.3 The Importance of Naming entities and DNS

Understanding the principles of naming entities and DNS management is essential for building robust and efficient distributed systems. By employing scalable naming schemes and leveraging DNS effectively, organizations can ensure seamless communication, resource discovery, and reliable operation in distributed environments. In distributed systems, where computing resources are spread across multiple networked machines, efficient communication and interaction among distributed components are essential for system functionality and performance. Central to this communication is the ability to uniquely identify and address various entities within the distributed environment. This introduction explores the critical role of naming entities and the Domain Name System (DNS) in facilitating communication and resource discovery in distributed systems.

At the heart of the Internet's naming infrastructure lies the Domain Name System (DNS), a distributed hierarchical naming system that translates human-readable domain names (e.g., www.example.com) into machine-readable IP addresses (e.g., 192.0.2.1) and vice versa. DNS plays a fundamental role in enabling users to access websites and services using memorable domain names, abstracting away the complexities of IP address management.

4.4 Key Components of DNS

DNS operates through a distributed system of servers, including authoritative name servers, recursive resolvers, and caching servers. These servers work collaboratively to resolve domain names to their corresponding IP addresses, providing efficient and reliable name resolution services to clients.

4.5 Names, Identifiers, and Addresses and their distinctions

In distributed systems, naming entities involves the assignment of unique identifiers to various components, services, and resources to facilitate communication, interaction, and resource management. This process encompasses names, identifiers, and addresses, each serving distinct roles in identifying entities within the distributed environment. Understanding the differences between these concepts is crucial for designing effective naming schemes and managing distributed systems efficiently.

4.5.1 Names

Names are human-readable labels used to refer to entities within a distributed system. They are typically chosen to be meaningful and intuitive, making it easier for users, developers, and administrators to identify and reference specific entities. Names are often hierarchical, allowing for organization and categorization of entities into logical groupings.

In distributed systems, names can represent a wide range of entities, including:

Computers and Servers: Hostnames, such as "server.example.com" or "workstation1.local", identify individual machines within a network.

Services and Applications: Service names, such as "database-service" or "payment-gateway", identify specific services or applications running on distributed systems.

Resources and Objects: Object names, such as "file.txt" or "user123", identify resources or objects within the system, such as files, documents, or users.

Names provide a level of abstraction that shields users and applications from the underlying details of network addressing and topology. However, names alone are not sufficient for communication and resource access in distributed systems; they need to be translated into machine-readable identifiers and addresses.

4.5.2 Identifiers

Identifiers are unique, system-assigned labels used to unambiguously identify entities within a distributed system. Unlike names, identifiers are typically not chosen by users or administrators but are generated or assigned by the system itself. Identifiers are used internally by the system to reference and manage entities efficiently.

In distributed systems, identifiers may take various forms, including:

UUIDs (Universally Unique Identifiers): Globally unique identifiers generated using algorithms that ensure uniqueness across distributed systems. UUIDs are commonly used to identify resources, transactions, or sessions within distributed applications.

Object IDs: Unique identifiers assigned to objects or resources within a system's data model. Object IDs are used internally by applications and databases to reference and manipulate data objects.

Process IDs (PIDs): Identifiers assigned to individual processes or threads within a distributed system. PIDs are used by the operating system for process management and resource allocation.

Identifiers provide a low-level mechanism for uniquely identifying entities within a distributed system. They are essential for efficient resource management, concurrency control, and coordination across distributed components. However, identifiers are often system-specific and may not be meaningful or human-readable.

4.5.3 Addresses

Addresses are machine-readable labels used to locate entities within a distributed system. Unlike names and identifiers, which focus on identification, addresses specify the physical or network location of entities, enabling communication and data exchange between distributed components.

In distributed systems, addresses may include:

IP Addresses: Numeric labels assigned to network interfaces and used to identify and locate devices within a network. IPv4 and IPv6 addresses are commonly used in distributed systems for network communication.

URLs (Uniform Resource Locators): Uniform resource locators that specify the protocol, host, and path to access web resources. URLs enable clients to locate and retrieve web pages, files, and services from remote servers.

Addresses serve as the foundation for communication and data transfer in distributed systems. They enable entities to send messages, access resources, and establish connections across networks. Addresses are essential for establishing communication channels, routing data packets, and ensuring reliable delivery within distributed environments.

4.6 Distinctions between Names, Identifiers, and Addresses

While names, identifiers, and addresses all play important roles in naming entities within distributed systems, they serve distinct purposes and have different characteristics:

Names provide human-readable labels for entities and are used for identification and reference by users and applications.

Identifiers are system-assigned labels that uniquely identify entities within the system and are used for internal management and manipulation.

Addresses specify the physical or network location of entities and are used for communication and data exchange between distributed components.

By understanding the distinctions between names, identifiers, and addresses, architects and developers can design effective naming schemes and communication protocols that meet the requirements of distributed systems, balancing human readability with system efficiency and scalability.

4.7 Uniform Resource Names (URNs) and its Importance

Uniform Resource Names (URNs) are a type of Uniform Resource Identifier (URI) used to uniquely identify resources in a persistent and location-independent manner. Unlike Uniform Resource Locators (URLs), which specify the location of a resource, URNs provide a consistent and permanent name for a resource regardless of its location or access method.

Structure of URNs: URNs follow a specific syntax defined by the Internet Engineering Task Force (IETF) in RFC 8141. They consist of three main components:

URN Scheme: The URN scheme specifies the namespace to which the URN belongs. Common URN schemes include "urn:isbn" for identifying books by their International Standard Book Number (ISBN) and "urn:uuid" for universally unique identifiers (UUIDs).

Namespace Identifier (NID): The NID uniquely identifies the namespace to which the URN belongs. It is typically a hierarchical string, such as "isbn" or "uuid", that defines the context or type of resource being identified.

Namespace-Specific String (NSS): The NSS is the portion of the URN that provides the specific identifier within the namespace. It can vary in format and content depending on the rules and conventions established by the namespace authority.

Importance of URNs

Persistent Identification: URNs provide a persistent and stable identifier for resources, even if their location or access method changes over time. This makes URNs suitable for referencing resources in scholarly publications, citations, and digital archives, ensuring that the identifier remains valid and functional indefinitely.

Location Independence: Unlike URLs, which may change if a resource is moved to a different location or domain, URNs are location-independent. They do not contain information about the resource's location, making them suitable for referencing resources that may be accessed through different protocols or network paths.

Global Uniqueness: URNs are designed to be globally unique identifiers, ensuring that no two resources within the same namespace have the same URN. This prevents naming conflicts and ambiguity, allowing for unambiguous identification and reference to resources across distributed systems and networks.

Decentralized Naming: URNs support decentralized naming schemes, allowing different organizations and communities to define their own namespaces and assign URNs to resources within those namespaces. This promotes interoperability and flexibility in naming resources across diverse domains and contexts.

Interoperability: URNs are part of the broader framework of Uniform Resource Identifiers (URIs), which also includes URLs and Uniform Resource Characteristics (URCs). This interoperability enables URNs to be used in conjunction with other URI schemes, facilitating seamless integration with existing web technologies and protocols.

Overall, URNs play a vital role in providing persistent, location-independent, and globally unique identifiers for resources in distributed systems. By offering a standardized mechanism for naming resources, URNs contribute to improved resource management, interoperability, and long-term accessibility in digital environments.

4.8 The Name Resolution Mechanisms and Strategies

Name resolution mechanisms or strategies are techniques used to translate human-readable names into machine-readable identifiers or addresses within a distributed system. These mechanisms are essential for enabling communication and resource access by resolving names to their corresponding entities. Several name resolution strategies exist, each with its characteristics and suitability for different distributed system architectures:

4.8.1 Local Name Resolution

In this approach, each node within the distributed system maintains a local mapping of names to identifiers or addresses. When a name resolution request is received, the local node consults its mapping table to find the corresponding identifier or address. Local name resolution is simple and efficient but may lack scalability and consistency in larger distributed systems.

4.8.2 Centralized Name Resolution

A centralized name resolution system employs a central server or service responsible for maintaining a global mapping of names to identifiers or addresses. When a name resolution request is received, it is forwarded to the central server, which performs the resolution and returns the result to the requesting node. Centralized name resolution simplifies management and ensures consistency but may introduce single points of failure and scalability limitations.

4.8.3 Distributed Hash Tables (DHTs)

DHTs distribute the responsibility for name resolution across multiple nodes in the distributed system using a decentralized approach. Each node in the DHT is responsible for a portion of the name space, and name resolution requests are routed through the network based on a distributed hash function. DHTs provide scalability, fault tolerance, and load balancing but may suffer from increased latency and complexity in routing.

4.8.4 Hierarchical Name Resolution

Hierarchical name resolution structures names in a hierarchical manner, with each level representing a different scope or domain. Name resolution proceeds recursively through the hierarchy, starting from the root and descending to the specific entity. Hierarchical name resolution is commonly used in domain name systems (DNS) and directory services, providing scalability, organization, and delegation of naming authority.

4.8.5 Iterative Name Resolution

In iterative name resolution, the name resolution process involves multiple iterative steps, with each step querying a different node or service for resolution. The requesting node iteratively contacts authoritative servers or services until it receives a definitive resolution. Iterative name resolution is flexible and fault-tolerant but may incur higher latency and complexity due to multiple round-trip queries.

4.9 Structuring and Organizing Name Spaces

Name spaces are logical namespaces used to organize and structure names within a distributed system. Effective organization of name spaces facilitates efficient naming, management, and resolution of entities. Several approaches are used to structure and organize name spaces within distributed systems:

4.9.1 Flat Name Space

In a flat name space, all names exist within a single, global namespace without hierarchical structure. Each name is unique within the namespace, and resolution is based on exact matching. Flat name spaces are simple and easy to implement but may suffer from naming conflicts and scalability limitations in large distributed systems.

4.9.2 Hierarchical Name Space

Hierarchical name spaces organize names in a hierarchical tree-like structure, with each level representing a different scope or domain. Names are composed of multiple components separated by delimiters, such as dots (.) in DNS. Hierarchical name spaces support delegation of naming authority, scalability, and efficient resolution through hierarchical traversal.

4.9.3 Partitioned Name Space

Partitioned name spaces divide the global namespace into smaller partitions or subdomains, each managed independently. Partitioning enables distributed management and delegation of naming authority, allowing different organizations or administrative domains to control their portion of the namespace. Partitioned name spaces support scalability, autonomy, and administrative flexibility.

4.9.4 Distributed Name Space

Distributed name spaces distribute naming authority and resolution across multiple nodes or servers within the distributed system. Each node is responsible for managing a portion of the namespace, and resolution requests are routed dynamically based on the distributed structure. Distributed name spaces provide scalability, fault tolerance, and load balancing but require robust coordination and synchronization mechanisms.

4.10 Directory Services and Distributed Name Services

Directory services and distributed name services are specialized components within distributed systems responsible for managing naming information and facilitating name resolution. These services provide centralized or distributed repositories for storing and retrieving naming data, enabling efficient and scalable name resolution across the distributed environment.

4.10.1 Directory Services

Directory services centralize naming information and provide a unified directory or database for storing and querying naming data. They support features such as search, query, and access control, allowing clients to retrieve information about entities based on various attributes or criteria. Directory services are commonly used in enterprise environments for managing user identities, resources, and access permissions.

4.10.2Distributed Name Services

Distributed name services distribute naming information across multiple nodes or servers within the distributed system. They employ distributed data structures, such as DHTs or replicated databases, to store and replicate naming data across the network. Distributed name services provide scalability, fault tolerance, and decentralized management of naming information, suitable for large-scale distributed systems and peer-to-peer networks.

In summary, effective name resolution mechanisms, structured name spaces, and specialized directory services are essential components of distributed systems, enabling efficient naming, resolution, and management of entities across diverse network environments. By employing appropriate naming strategies and services, organizations can build robust and scalable distributed systems capable of meeting the demands of modern computing environments.

4.11 Challenges in Naming Entities

Scalability: As distributed systems grow in size and complexity, managing a large number of named entities becomes challenging. Scalable naming schemes and resolution mechanisms are necessary to handle the increasing number of entities.

Consistency: Ensuring consistency in naming across distributed components is essential to avoid ambiguity and confusion. Distributed naming systems must support mechanisms for name resolution and synchronization to maintain consistency.

Fault Tolerance: Distributed naming systems should be resilient to failures and network partitions. Redundancy, replication, and fault-tolerant algorithms are employed to ensure the availability and reliability of naming services.

4.12 Domain Name System (DNS)

The Domain Name System (DNS) is a fundamental part of how the internet works, serving as a distributed directory that translates human-readable domain names into numerical IP addresses. This translation process allows users to access websites, send emails, and utilize various internet services using familiar domain names rather than having to remember complex IP addresses. One of the most common and important uses of DNS is connecting your network to the global Internet. To connect to the Internet, your network IP address must be registered with whomever is administering your parent domain.

Name-to-Address Resolution

Though it supports the complex, worldwide hierarchy of computers on the Internet, the basic function of DNS is actually very simple: providing name-to address resolution for TCP/IP-based networks. Name-to-address resolution, also referred to as mapping, is the process of finding the IP address of a computer in a database by using its host name as an index. Name-to-address mapping occurs when a program running on your local machine needs to contact a remote computer. The program most likely will know the host name of the remote computer but might not know how to locate it, particularly if the remote machine is in another company, miles from your site. To get the remote machine's address, the program requests assistance from the DNS software running on your local machine, which is considered a DNS client. Your machine sends a request to a DNS name server, which maintains the distributed DNS database. The files in the DNS database bear little resemblance to the NIS+ host or ipnodes Table or even the local /etc/hosts or /etc/inet/ipnodes file, though they maintain similar information: the host names, the ipnode names, IPv4 and IPv6 addresses, and other information about a particular group of computers. The name server uses the host name your machine sent as part of its request to find or "resolve" the IP address of the remote machine. It then returns this IP address to your local machine if the host name is in its DNS database.

If the host name is not in that name server's DNS database, this indicates that the machine is outside of its authority, or, to use DNS terminology, outside the local administrative domain. Thus, each name server is spoken of as being "authoritative" for its local administrative domain. Fortunately, the local name server maintains a list of host names and IP addresses of root domain name servers, to which it will forward the request from your machine. These root name servers are authoritative for huge organizational domains, as explained fully in DNS Hierarchy and the Internet. These hierarchies resemble UNIX file systems, in that they are organized into an upside down tree structure. Each root name server maintains the host names and IP addresses of top level domain name servers for a company, a university, or other large organizations. The root name server sends your request to the top-level name servers that it knows about. If one of these servers has the IP address for the host you requested, it will return the information to your machine. If the top-level servers do not know about the host you requested, they pass the request to second level name servers for which they maintain information. Your request is then passed on down through the vast organizational tree. Eventually, a name server that has information about your requested host in its database will return the IP address back to your machine.

4.12.1 Role of Domain Name System (DNS)

Overview: DNS is a distributed hierarchical naming system used to translate human-readable domain names (e.g., www.example.com) into IP addresses (e.g., 192.0.2.1) and vice versa. It plays a crucial role in the Internet's architecture by enabling users to access websites and services using memorable domain names.

Hierarchy: DNS organizes domain names into a hierarchical structure, with the root domain at the top, followed by top-level domains (TLDs), second-level domains, and sub-domains. This hierarchical structure allows for efficient name resolution and delegation of authority.

Name Resolution: DNS operates through a distributed system of DNS servers, including authoritative name servers, recursive resolvers, and caching servers. When a client requests the IP address of a domain name, the DNS resolver recursively queries authoritative name servers until it obtains the corresponding IP address.

Resource Records: DNS uses resource records (RRs) to store information about domain names, including mapping records (e.g., A records for IPv4 addresses, AAAA records for IPv6 addresses), alias records (CNAME), mail exchange records (MX), and others. These records provide essential metadata for resolving domain names.

4.12.2 DNS Administrative Domains

From a DNS perspective, an administrative domain is a group of machines which are administered as a unit. Information about this domain is maintained by at least two name servers, which are "authoritative" for the domain. The DNS domain is a logical grouping of machines. The domain groupings could correspond to a physical grouping of machines, such as all machines attached to the Ethernet in a small business. Similarly, a local DNS domain could include all machines on a vast university network that belong to the computer science department or to university administration. For example, suppose the Ajax company has two sites, one in San Francisco and one in Seattle. The Retail.Sales.Ajax.com. domain might be in Seattle and the Wholesale.Sales.Ajax.com. domain might be in San Francisco. One part of the Sales.Ajax.com. domain would be in one city, the other part in the second city. Each administrative domain must have its own unique sub-domain name. Moreover, if you want your network to participate in the Internet, the network must be part of a registered administrative domain. The section Joining the Internet has full details about domain names and domain registration.

There are three types of DNS name servers which are Master server, Slave server and Stub server. Each domain must have one master server and should have at least one slave server to provide backup

4.13 Domain Name System (DNS): Architecture, Hierarchy and Zones

The Domain Name System (DNS) is like the phonebook of the internet, translating human-readable domain names (like google.com) into IP addresses (like 172.217.12.174) that computers use to communicate with each other. It's a hierarchical decentralized naming system, organized into a structure of domains and zones.

4.13.1 Architecture

DNS operates on a client-server model. When a user types a domain name into their web browser, the browser sends a DNS query to a DNS resolver (typically operated by the user's Internet Service Provider or ISP). If the resolver already has the IP address for the domain in its cache, it returns the result immediately. Otherwise, it forwards the query through a series of DNS servers until it reaches a server that can provide the IP address for the requested domain. Once the IP address is obtained, the resolver returns it to the user's device, allowing the device to establish a connection with the desired website or service.

At its core, DNS operates on a client-server model, where DNS servers work together to fulfill requests from clients (such as web browsers or email clients) to resolve domain names to IP addresses. This process involves several types of DNS servers:

Root DNS Servers: These are the top-level DNS servers in the DNS hierarchy, managing the root zone. There are 13 sets of root DNS servers distributed worldwide, each represented by a letter from A to M. These servers provide information about the authoritative DNS servers for top-level domains (TLDs).

Top-Level Domain (TLD) Name Servers: These servers are responsible for managing the DNS records associated with specific top-level domains (like .com, .org, .net). They maintain information about domain names registered within their respective TLDs and direct queries to the authoritative name servers for the next level in the domain hierarchy.

Authoritative Name Servers: These servers store the authoritative DNS records for specific domains or zones. They are responsible for providing authoritative answers to DNS queries related to the domains they oversee. Authoritative name servers can be operated by domain registrars, internet service providers (ISPs), organizations, or hosting providers.

Recursive DNS Resolvers: These are the DNS servers that most internet users interact with indirectly through their ISPs or network providers. When a client makes a DNS query, the recursive resolver handles the request on behalf of the client, recursively querying other DNS servers until it obtains the IP address associated with the requested domain name.

4.13.2 Hierarchy

DNS has a hierarchical structure composed of multiple levels, with each level separated by a dot. The highest level is the root domain, represented by a dot (.), followed by top-level domains (TLDs) like .com, .org, .net, and country code top-level domains (ccTLDs) like .uk, .de, .jp, etc. Below TLDs are second-level domains (SLDs), such as google.com or wikipedia.org. Subdomains can be further specified, resulting in a structure like subdomain.example.com.

DNS follows a hierarchical structure that organizes domain names into a tree-like system. Each level of the hierarchy represents a different level of specificity in the domain name. The hierarchy begins with the root domain, represented by a single dot (.), followed by subsequent levels of domains, separated by dots. For example:

Root Domain: .

Top-Level Domain (TLD): .com, .org, .net

Second-Level Domain (SLD): google.com, wikipedia.org

Subdomains: www.google.com, blog.wikipedia.org

The hierarchical structure allows for efficient and scalable DNS resolution by dividing the responsibility for managing different parts of the DNS namespace among various DNS servers.

4.13.3 Zones

Zones are portions of the DNS namespace that are managed by a single entity, typically an organization or a domain registrar. Each zone corresponds to a portion of the domain name space and is administered independently. Zones are delineated by domain boundaries and are responsible for managing the domain's DNS records, including mapping domain names to IP addresses (A records), mapping domain names to mail servers (MX records), establishing domain aliases (CNAME records), and configuring DNSSEC security settings.

There are two primary types of zones:

Forward Lookup Zones: These zones translate domain names to IP addresses. When a user types a domain name into their browser, the DNS resolver looks up the corresponding IP address in the forward lookup zone.

Reverse Lookup Zones: These zones perform the opposite function, translating IP addresses to domain names. They are commonly used for logging, troubleshooting, and security purposes.

Each zone is administered by one or more DNS servers responsible for hosting and distributing the zone's DNS records. These servers are often categorized as authoritative servers, which store authoritative DNS records for specific domains or zones, and caching servers, which temporarily store DNS records to speed up subsequent DNS queries.

Each DNS zone contains various types of DNS records, including:

A Records (Address Records): Map domain names to IPv4 addresses.

AAAA Records (IPv6 Address Records): Map domain names to IPv6 addresses.

MX Records (Mail Exchange Records): Specify mail servers responsible for receiving email for a domain.

CNAME Records (Canonical Name Records): Alias one domain name to another (canonical) domain name.

NS Records (Name Server Records): Identify authoritative name servers for the zone.

By organizing DNS information into zones, DNS administrators can efficiently manage and update DNS records for specific domains without affecting the resolution of other domains.

Example:

Let's consider the process of resolving the domain name "www.google.com" to its corresponding IP address using the DNS hierarchy:

The user's web browser sends a DNS query to a recursive DNS resolver, asking for the IP address of "www.google.com."

The recursive resolver begins the resolution process by querying the root DNS servers to find the authoritative name servers for the ".com" TLD.

The root DNS servers respond with the IP addresses of the TLD name servers responsible for the ".com" TLD.

The recursive resolver then queries one of the ".com" TLD name servers, asking for the authoritative name servers for the "google.com" domain.

The ".com" TLD name server responds with the IP addresses of the authoritative name servers for the "google.com" domain.

The recursive resolver selects one of the authoritative name servers for "google.com" and sends a query for the IP address of "www.google.com."

The authoritative name server for "google.com" responds with the IP address(es) associated with "www.google.com."

The recursive resolver returns the IP address(es) to the user's web browser, allowing the browser to establish a connection with the Google website.

In this example, multiple DNS servers work together hierarchically to resolve the domain name "www.google.com," demonstrating the distributed nature of the DNS architecture.

In summary, the Domain Name System (DNS) is a hierarchical and decentralized naming system that translates domain names into IP addresses, enabling users to access internet resources using human-readable names. Through its architecture, hierarchy, and zone-based management, DNS facilitates efficient and scalable resolution of domain names across the internet.DNS is a crucial component of the internet's infrastructure, providing a decentralized system for translating domain names into IP addresses. Its hierarchical architecture and zone-based management enable efficient and scalable resolution of domain names across the internet.

4.14 Resource Records (RRs) in Domain Name System (DNS)

Resource Records (RRs) are the building blocks of the Domain Name System (DNS), containing various types of information associated with domain names. Each RR consists of several fields, including the domain name it pertains to, a class identifier (usually IN for internet), a time-to-live (TTL) value indicating how long the record can be cached, and type-specific data.

Here are some common types of Resource Records and their purposes:

A Records (Address Records): These records map domain names to IPv4 addresses. For example, an A record for "example.com" might contain the IPv4 address "192.0.2.1".

AAAA Records (IPv6 Address Records): Similar to A records but used for mapping domain names to IPv6 addresses.

MX Records (Mail Exchange Records): MX records specify the mail servers responsible for receiving email for a domain. Each MX record has a priority value indicating the order in which mail servers should be used. For example, an MX record for "example.com" might specify "mail.example.com" as the mail server with priority 10.

CNAME Records (Canonical Name Records): CNAME records alias one domain name to another (canonical) domain name. They are often used to create aliases for existing domains or to implement load balancing across multiple servers. For example, a CNAME record for "www.example.com" might point to "example.com".

NS Records (Name Server Records): NS records identify the authoritative name servers for a domain. These records specify which DNS servers are authoritative for answering queries related to a particular domain. For example, NS records for "example.com" might specify "ns1.example.com" and "ns2.example.com" as authoritative name servers.

PTR Records (Pointer Records): PTR records are used in reverse DNS lookups to map IP addresses to domain names. They are the reverse equivalent of A and AAAA records. For example, a PTR record might map the IP address "192.0.2.1" to the domain name "example.com".

TXT Records (Text Records): TXT records contain arbitrary text information associated with a domain. They are often used for adding human-readable notes, SPF (Sender Policy Framework) records for email authentication, or other types of metadata. For example, a TXT record might contain a message like "This domain is managed by Example Corp".

SOA Records (Start of Authority Records): SOA records are fundamental to each DNS zone and contain essential information about the zone, such as the primary name server for the zone, the email address of the zone administrator, the serial number of the zone, and various timing parameters (e.g., refresh interval, retry interval, expiry time, minimum TTL).

These are just a few examples of the many types of Resource Records used in DNS. Each RR serves a specific purpose in facilitating the resolution of domain names to IP addresses and providing essential information about domain configuration and services. By combining different types of Resource Records, DNS administrators can configure the behavior of domain names and manage internet services effectively.

4.15 Name Resolution Process in the Domain Name System (DNS)

The name resolution process in the Domain Name System (DNS) is the mechanism by which domain names are translated into IP addresses. This process involves multiple steps and components working together to provide the correct IP address for a given domain name. Here's a brief overview of the name resolution process:

DNS Query Initiation: The name resolution process begins when a user or application requests the IP address associated with a domain name. For example, when a user enters a domain name into a web browser, the browser initiates a DNS query to resolve the domain name.

Local DNS Cache Lookup: The DNS resolver on the user's device first checks its local cache to see if it has recently resolved the requested domain name. If the IP address is found in the cache and has not expired (based on the Time-to-Live or TTL value), the resolver can immediately return the cached IP address to the user.

Recursive DNS Query: If the IP address is not found in the local cache or has expired, the resolver initiates a recursive DNS query. The resolver sends the query to a recursive DNS resolver, typically operated by the user's Internet Service Provider (ISP) or network provider.

Root DNS Servers: If the recursive resolver does not have the requested domain name cached, it starts the resolution process by querying the root DNS servers. The root servers provide information about the authoritative name servers for the top-level domains (TLDs) based on the domain name's TLD (e.g., .com, .org, .net).

TLD Name Servers: The recursive resolver then queries one of the TLD name servers to obtain information about the authoritative name servers for the specific domain name's TLD. For example, if the requested domain name is "example.com," the resolver queries the .com TLD name servers.

Authoritative Name Servers: With the information obtained from the TLD name servers, the recursive resolver sends a query to one of the authoritative name servers for the requested domain name. These authoritative name servers are responsible for storing DNS records specific to the domain, such as A (Address) records, MX (Mail Exchange) records, etc.

Response from Authoritative Name Server: The authoritative name server responds to the recursive resolver's query with the IP address associated with the requested domain name, along with other relevant DNS records (such as MX records for email servers).

Response to Client: Finally, the recursive resolver returns the IP address obtained from the authoritative name server to the client that initiated the DNS query. The client can then use the IP address to establish a connection with the desired website, server, or service.

Throughout the name resolution process, DNS queries and responses are exchanged between different DNS servers, including recursive resolvers, root servers, TLD name servers, and authoritative name servers. By following this hierarchical resolution process, DNS efficiently translates domain names into IP addresses, allowing users to access internet resources using human-readable domain names.

4.16 Caching and Performance Optimization of DNS

Caching is a crucial aspect of DNS that significantly improves performance by reducing the time required to resolve domain names to IP addresses. When a DNS resolver receives a response to a query, it caches the response for a specified period (TTL) determined by the authoritative name server. Subsequent queries for the same domain name can be answered directly from the resolver's cache, eliminating the need to perform the entire resolution process again.

Reduced Latency: Caching helps reduce latency by providing faster responses to DNS queries. Instead of querying authoritative name servers every time, resolvers can retrieve cached responses locally, resulting in quicker resolution times.

Improved Scalability: Caching reduces the load on authoritative name servers by distributing query traffic across various levels of the DNS hierarchy. This improves the overall scalability of the DNS infrastructure, ensuring efficient resolution even during periods of high query volume.

Enhanced User Experience: Faster DNS resolution times lead to a better user experience, as websites and services can be accessed more quickly. This is particularly important for web applications and services that rely on low latency to deliver content and functionality to users.

To optimize performance further, DNS servers may employ techniques such as prefetching, which involves proactively caching DNS records before they expire based on usage patterns or anticipated queries.

4.17 Security and Challenges of DNS

While DNS plays a critical role in enabling communication on the internet, it also presents several security challenges that need to be addressed:

DNS Spoofing and Cache Poisoning: Attackers may attempt to manipulate DNS responses to redirect users to malicious websites or intercept sensitive information. DNS spoofing involves forging DNS responses or injecting malicious records into DNS caches to redirect users to fraudulent websites. Cache poisoning attacks exploit vulnerabilities in DNS resolvers to corrupt their cache with false information.

DNS Amplification Attacks: In DNS amplification attacks, attackers exploit open DNS resolvers to amplify the volume of traffic directed towards a victim's server. By sending DNS queries with spoofed source IP addresses to open resolvers, attackers cause the resolvers to send large amounts of traffic to the victim's server, overwhelming its network capacity.

DNSSEC (DNS Security Extensions) Implementation: DNSSEC is a suite of extensions to DNS designed to provide authentication and data integrity for DNS responses. By digitally signing DNS records, DNSSEC helps prevent DNS spoofing and cache poisoning attacks. However, DNSSEC implementation can be complex, and not all DNS servers support it.

DDoS (Distributed Denial of Service) Attacks: DNS infrastructure is susceptible to DDoS attacks, which aim to disrupt DNS resolution by flooding DNS servers with an overwhelming volume of traffic. DDoS attacks can cause service outages, impacting the availability of websites and online services.

To mitigate these security challenges, organizations employ various security measures such as deploying firewalls and intrusion detection/prevention systems, implementing DNS filtering and monitoring solutions, regularly updating DNS software to patch known vulnerabilities, and adopting DNSSEC to enhance DNS security.

In summary, while DNS caching and performance optimization enhance the efficiency of DNS resolution, security remains a critical concern. By addressing security challenges and implementing appropriate safeguards, organizations can ensure the integrity, availability, and reliability of their DNS infrastructure.

4.18 DNSSEC in Domain Name System (DNS)

DNSSEC, or Domain Name System Security Extensions, is a set of protocols and cryptographic techniques designed to enhance the security and integrity of the Domain Name System (DNS). DNSSEC addresses vulnerabilities in the DNS that can be exploited by attackers to perform various types of attacks, such as DNS cache poisoning and DNS spoofing.

How DNSSEC Works:

Digital Signatures: DNSSEC uses digital signatures to verify the authenticity and integrity of DNS data. Each DNS record is signed with a digital signature generated using public-key cryptography.

Chain of Trust: DNSSEC establishes a chain of trust, starting from the root DNS zone and extending down to individual domain names. The root zone is signed with a root zone signing key (ZSK), and each subsequent zone signs its zone data using its own zone signing key (ZSK). DNS resolvers use these cryptographic signatures to validate DNS responses received from authoritative name servers.

Public-Key Cryptography: DNSSEC relies on public-key cryptography to generate key pairs consisting of a public key and a private key. The private key is used to sign DNS records, while the corresponding public key is published in DNSKEY records as part of the DNS zone's DNSSEC configuration.

Chain of Delegation: DNSSEC ensures the authenticity of DNS data throughout the chain of delegation, from the root zone to the authoritative name servers for individual domain names. By verifying the digital signatures associated with DNS records, DNS resolvers can validate that the DNS data has not been tampered with or forged.

Benefits of DNSSEC:

Data Integrity: DNSSEC ensures the integrity of DNS data by detecting and preventing unauthorized modification or tampering of DNS records. This helps prevent DNS cache poisoning attacks and DNS spoofing attacks.

Authentication: DNSSEC provides authentication of DNS responses, allowing DNS resolvers to verify that the data they receive originates from legitimate authoritative name servers and has not been altered in transit.

Trustworthiness: DNSSEC enhances the trustworthiness of DNS data by enabling DNS resolvers to verify the authenticity of DNS responses and validate the chain of trust from the root zone down to individual domain names.

Security: By mitigating DNS-related vulnerabilities, DNSSEC helps protect against various types of DNS-based attacks, including man-in-the-middle attacks, DNS redirection attacks, and DNS amplification attacks.

In summary, DNSSEC is a critical security measure that strengthens the security and integrity of the Domain Name System. By using digital signatures and cryptographic techniques, DNSSEC helps prevent DNS-related attacks and enhances the trustworthiness of DNS data, contributing to a more secure and reliable internet infrastructure.

4.19 DNS Spoofing, Cache Poisoning and DNS Anycast in Domain Name System (DNS)

4.19.1 DNS Spoofing

DNS spoofing, also known as DNS cache poisoning, is a type of attack where an attacker manipulates DNS responses to redirect users to malicious websites or servers. The goal of DNS spoofing is to compromise the integrity of the DNS resolution process, leading users to unwittingly visit fraudulent websites or disclose sensitive information.

How it works: In a DNS spoofing attack, the attacker sends falsified DNS responses to DNS resolvers or caches, containing forged or malicious IP addresses mapped to legitimate domain names. When a user or application queries the DNS resolver for a particular domain name, the resolver may cache the falsified response. Subsequent queries for the same domain name are then redirected to the malicious IP address specified by the attacker.

Implications: DNS spoofing can lead to various security risks, including phishing attacks, malware distribution, and data theft. By redirecting users to counterfeit websites that mimic legitimate ones, attackers can trick users into entering sensitive information such as login credentials, credit card numbers, or personal details.

4.19.2 DNS Cache Poisoning

DNS cache poisoning is a specific type of DNS spoofing attack that targets DNS caches maintained by DNS resolvers. In a cache poisoning attack, the attacker exploits vulnerabilities in the DNS resolver's caching mechanism to inject falsified DNS records into its cache, thereby corrupting the integrity of the DNS data stored in the cache.

How it works: The attacker sends a flood of DNS queries containing requests for non-existent domain names or legitimate domain names with spoofed source IP addresses. By overwhelming the DNS resolver with a large volume of queries, the attacker increases the likelihood of the resolver accepting and caching falsified DNS responses containing malicious IP addresses.

Implications: Once the DNS resolver's cache is poisoned with falsified DNS records, all subsequent DNS queries for the affected domain names are redirected to the malicious IP addresses specified by the attacker. This can result in widespread DNS resolution errors, leading users to unintentionally access malicious websites or services.

4.19.3 DNS Anycast

DNS Anycast is a networking technique used to improve the performance, availability, and resilience of DNS infrastructure by routing DNS queries to the nearest or best-performing DNS server within a group of geographically distributed servers. With Anycast, multiple DNS servers advertise the same IP address for a given domain name, and network routing protocols ensure that DNS queries are directed to the nearest or most optimal server based on network conditions.

How it works: DNS Anycast involves deploying multiple DNS servers in different locations around the world and assigning them the same IP address for a specific domain name. When a DNS resolver sends a query to the Anycast IP address, the query is routed to the nearest DNS server in terms of network latency or routing distance.

Benefits: DNS Anycast improves DNS resolution performance by reducing latency and network congestion, as queries are directed to nearby DNS servers. It also enhances the availability and resilience of DNS infrastructure by distributing query traffic across multiple redundant servers, minimizing the impact of server failures or network outages.

In summary, DNS spoofing and cache poisoning are malicious techniques used to manipulate DNS resolution and redirect users to malicious websites or servers. DNS Anycast, on the other hand, is a networking technique that enhances the performance, availability, and resilience of DNS infrastructure by routing queries to the nearest or best-performing DNS server within a group of geographically distributed servers.

4.20 Summing Up

- Naming entities in distributed systems involves assigning unique identifiers to components, services, and resources to facilitate communication and interaction.
- The Domain Name System (DNS) serves as the backbone of the Internet's naming infrastructure, providing a hierarchical naming structure and facilitating name resolution. DNS operates through a distributed system of servers, including authoritative name servers and recursive resolvers, to translate domain names into IP addresses and vice versa.
- DNS operates through a distributed system of servers, including authoritative name servers, recursive resolvers, and caching servers.
- Name-to-address resolution, also referred to as mapping, is the process of finding the IP address of a computer in a database by using its host name as an index. Name-to-address mapping occurs when a program running on your local machine needs to contact a remote computer.

- DNS has a hierarchical structure composed of multiple levels, with each level separated by a dot. The highest level is the root domain, represented by a dot (.), followed by top-level domains (TLDs) like .com, .org, .net, and country code top-level domains (ccTLDs) like .uk, .de, .jp, etc. Below TLDs are second-level domains (SLDs), such as google.com or wikipedia.org.
- DNS operates on a client-server model. When a user types a domain name into their web browser, the browser sends a DNS query to a DNS resolver (typically operated by the user's Internet Service Provider or ISP).
- Resource Records (RRs) are the building blocks of the Domain Name System (DNS), containing various types of information associated with domain names.
- Uniform Resource Names (URNs) are a type of Uniform Resource Identifier (URI) used to uniquely identify resources in a persistent and location-independent manner.
- Caching is a crucial aspect of DNS that significantly improves performance by reducing the time required to resolve domain names to IP addresses. When a DNS resolver receives a response to a query, it caches the response for a specified period (TTL) determined by the authoritative name server.
- DNSSEC, or Domain Name System Security Extensions, is a set of protocols and cryptographic techniques designed to enhance the security and integrity of the Domain Name System (DNS).
- DNS spoofing, also known as DNS cache poisoning, is a type of attack where an attacker manipulates DNS responses to redirect users to malicious websites or servers. The goal of DNS spoofing is to compromise the integrity of the DNS

resolution process, leading users to unwittingly visit fraudulent websites or disclose sensitive information.

- DNS cache poisoning is a specific type of DNS spoofing attack that targets DNS caches maintained by DNS resolvers. In a cache poisoning attack, the attacker exploits vulnerabilities in the DNS resolver's caching mechanism to inject falsified DNS records into its cache, thereby corrupting the integrity of the DNS data stored in the cache.
- DNS Anycast is a networking technique used to improve the performance, availability, and resilience of DNS infrastructure by routing DNS queries to the nearest or best-performing DNS server within a group of geographically distributed servers.

4.21 References and Suggested Readings

Tanenbaum, A. S., & Van Steen, M. (2006). Distributed systems: principles and paradigms. Prentice Hall.

Mockapetris, P. (1987). RFC 1034: Domain names—concepts and facilities.

RFC 4033, RFC 4034, RFC 4035: DNS Security Introduction and Requirements, Protocol Modifications, and Threat Analysis of the Domain Name System.

Albitz, P., & Liu, C. (2017). DNS and BIND (5th ed.). O'Reilly Media.

4.22 Model Questions

1. Explain the Importance of Naming entities and DNS.
2. What do you means by the terms Names, Identifiers, and Addresses and their distinctions?
3. How does the chapter "Naming Entities" delve into the conceptual framework of assigning names to various entities within a system?

4. What are the key principles discussed in the chapter regarding the naming of entities, and how do they contribute to system organization and clarity?
5. Could you outline the key components of the Domain Name System (DNS) as discussed in the chapter, and how they contribute to the efficient resolution of domain names to IP addresses?
6. In what ways does the chapter explore the historical development and evolution of naming conventions, particularly in relation to the emergence of the internet and digital communication?
7. How does the chapter address the challenges and complexities inherent in managing naming systems, both at the level of individual entities and within the context of global networks?
8. Could you provide examples from the chapter illustrating the practical implications of effective naming strategies for enhancing user experience, system reliability, and security?
9. What insights does the chapter offer into the design principles underlying the DNS, including considerations related to scalability, redundancy, and fault tolerance?
10. How does the chapter examine the interplay between human-readable domain names and their corresponding machine-readable IP addresses, and the mechanisms by which DNS resolves this mapping?
11. What discussions or case studies are presented in the chapter regarding the governance and regulation of naming systems, particularly within the context of internet governance bodies and standards organizations?
12. In what ways does the chapter contextualize the significance of naming entities and the DNS within broader discussions of digital infrastructure, cybersecurity, and the socio-technical implications of networked communication?

4.23 Answer to check your progress/Possible Answers to SAQ

Choose the correct option from the following questions

1. Which of the following best describes the function of the Domain Name System (DNS)?

A) A protocol used for transferring files between networked computers.

B) An encryption standard for securing internet communications.

C) A hierarchical decentralized naming system for computers, services, or any resource connected to the internet.

D) A method for compressing large data files for efficient storage.

2. What is the primary purpose of the Domain Name System (DNS)?

A) Encrypting internet traffic for enhanced security. B) Transferring files between networked computers. C) Resolving domain names to IP addresses. D) Managing email communication protocols.

3. Which of the following best describes a domain name?

A) A numerical label assigned to each device connected to a computer network.

B) An alphanumeric string that represents the location of a website on the internet.

C) A protocol used for transferring files over the internet.

D) An encryption standard for securing online transactions.

4. What is the hierarchical structure used in domain names called?

A) Binary tree B) Directory structure C) Domain tree D) Domain hierarchy

5. Which organization oversees the management of the global Domain Name System?

A) World Wide Web Consortium (W3C)

B) Internet Corporation for Assigned Names and Numbers (ICANN)

C) Internet Engineering Task Force (IETF)

D) International Organization for Standardization (ISO)

6. What is the purpose of a top-level domain (TLD)?

A) Identifying the geographic location of a website.

B) Designating the type or category of an organization.

C) Encoding sensitive information for secure transmission.

D) Optimizing search engine ranking for a website.

7. Which of the following is an example of a generic top-level domain (gTLD)?

A) .com B) .us C) .gov D) .uk

8. What is the purpose of a DNS resolver?

A) Encrypting DNS queries for privacy protection.

B) Translating domain names into IP addresses.

C) Managing DNS servers for domain registration.

D) Analyzing network traffic for security threats.

9. Which protocol is commonly used for communication between DNS clients and servers?

A) HTTP B) SMTP C) UDP D) TCP

10. What is the significance of the root DNS servers in the DNS hierarchy?

A) They store all domain name records.

B) They manage top-level domain registries.

C) They provide the initial point of contact for DNS queries.

D) They regulate access to restricted websites.

11. What is the purpose of DNS caching?

A) Speeding up DNS resolution by storing previously resolved mappings.

B) Encrypting DNS queries to prevent eavesdropping.

C) Reducing the load on authoritative DNS servers.

D) Blocking access to malicious websites.

Answer Keys

1(C), 2(C), 3(B), 4(C), 5(B), 6(B),7(A),8(B), 9(C), 10(C),11(A)

×××

UNIT: 5

DISTRIBUTED TRANSACTIONS

Unit Structure:

5.1 INTRODUCTION

We have already learnt that a transaction in a system is a single logical unit of work. It consists of one or more operations that must be executed in such a way that either all of themare successful to achieve their completionor none of them is allowed to reach its completion and in that situation, the transaction is rollback to its earlier safe state. Transaction management is very important to maintain data integrity and consistency in a system.Transaction recovery and concurrency control are the two major parts of transaction management. In this unit, we are going to discuss about different concepts related to the transactions that are executed in distributed environments.

5.2 OBJECTIVES

After going through this chapter, we will be able to learn:

- About the requirements and characteristics of distributed transactions.
- About ACID properties in distributed environment.
- Different approaches to achieve ACID properties in distributed environment.
- About Two-phase commit (2PC) protocol.
- About Three-phase commit (3PC) protocol.
- About different Isolation levels.
- About concurrency control in distributed transactions.
- About handling durability in distributed systems.

5.3 INTRODUCTION TO DISTRIBUTED TRANSACTIONS

In distributed environments, a transaction is associated with multiple servers located in different physical sites. As a result,a distributed transaction becomes more complex than a transaction that is local to a particular server. A distributed transaction consists of multiple processes where each process is responsible to perform an operation of the transaction on a particular server in the distributed system. A distributed transaction can access and update multiple objects that may be managed by different servers. A distributed system must be able to identify the processes of a particular transaction.Atomicity of transactions in distributed environment must be maintained by ensuring that either all the processes of a transaction successfully complete their jobs in the associated servers or all are aborted by the associated servers.One of the servers in a distributed system is responsible for coordinating all other servers to maintain this atomicity property of distributed transactions.If a coordinator in a server opens or starts a distributed transaction then it serves as the coordinator of that transaction. Concurrency control and transaction recovery in distributed environment requires more effort than in case of a single server system. Maintaining data integrity and consistency and handling different system failures are the major objectives of distributed transactions.

Distributed transactions are prepared either asflat transactions or as nested transactions.Flat transactions are simple distributed transaction wheremultiple objects are requested that are managed at multiple servers. A flat distributed transaction completes only one request foran object at a time. It means after completing the current request, the next request will be completed in case of flat transactions. As a result, a flat transaction accesses objects managed at multiple serverone after another.

On the other hand, in case of nested distributed transactions, a top-level transaction is available which can open nested subtransactions. Again each subtransaction can open the next level of subtransactions. This process can be continued as per requirement. Subtransactions available at asimilar level can execute simultaneously.

5.3.1 Need for Distributed Transactions

Need for distributed transactions are discussed in the following points.

- Distributed transactions are required to maintain data consistency and integrity in all servers available in a distributed system.
- Distributed transactions are crucial when a database is distributed among several servers in a distributed system.
- Distributed transactions may improve the fault tolerance and scalability in a distributed system.
- Distributed transactions offer a way for coordination among different servers available in the system. It will

improve the overall system performance. So, distributed transactions are very crucial to implement complex processes in a large and complex distributed system where large amount of coordination among different servers are required.

- Error handling mechanism is provided by Distributed transactions. In case of any system failure, distributed transactions may play an important role to recover the system so that the system can again attain a safe state.

5.3.2 Characteristics of Distributed Transactions

Characteristics of distributed transactions are presented in the following points.

- Operations of a distributed transaction may be executed in multiple servers and a distributed transaction may access multiple objects that are managed by different servers.
- In case of a distributed transaction, either all operations of the transactioncomplete successfully in different servers or all operations are aborted by corresponding servers.
- Execution of each distributed transaction is isolated from other transactions so that concurrent distributed transactions in a system don’t interfere with each other’s operations.
- Successful completion of a distributed transaction moves the system to a new safe state and it assure that the changes occurred in the system due to the transaction will not be lost in any type of system failure.

5.4 ACID PROPERTIES IN DISTRIBUTED ENVIRONMENT

ACID (Atomic Consistent Isolated Durable) properties are followed by distributed transactions to maintain consistency and reliability in a distributed system. It is a set of four properties discussed as follows.

- **Atomicity:**According to this property, either all operations of each distributed transaction complete successfully or the transaction is aborted and no change will be permitted in the system. The goal of this property is to prevent partial completion of distributed transactions so that consistency of the system can be maintained.

- **Consistency:**According to this property, the system will move to a consistent state from a consistent state after the competition of each distributed transaction. The goal of this property is to maintain the data integrity of the system after each successful completion of distributed transactions.
- **Isolation:**According to this property, the execution of each distributed transaction is isolated from other concurrent distributed transactions so that concurrent transactions are executed without interfering each other’s operations. The goal of this property is to avoid race conditions among distributed transactions.
- **Durability:**According to this property, the changes happened to a distributed system after the successful completion of a distributed transaction is permanent. These changes will not be lost even in case of any type of system failures. The goal of this property is to save the changes that are happened to the system after each successful completion of distributed transactions so that these will not be lost in any situation.

5.4.1 Challenges in Achieving ACID Properties in Distributed Environment

We have already learnt about the importance of implementing ACID properties in case of distributed transactions. But several challenges are also available in distributed environment in achieving ACID properties.The major challenges are presented in the following points.

- We have already learnt that a distributed system is a group of multiple independent servers that are located in different physical sites but they are connected by a centrally controlled computer network. Sometimes it may be possible that one or more servers are disconnected from the corresponding distributed systemdue to network failures. In that situation, achieving atomicity and consistency in related distributed transactions will be difficult.

- Achieving durability in distributed transactions require efficient mechanismsto handle server failures, server crashes, system errors and data corruption. Complexity in system design and cost of the system will be increased to include such mechanisms.
- Achieving isolation in distributed transactions require efficient concurrency control mechanism. Communication delay among servers in a distributed system may increase the difficulty to achievetransaction isolation in distributed environment.
- Using locks to achieve isolation in distributed transactions may create distributed deadlocks in the system.
- Difficulty in achieving ACID properties in distributed transactions may be increased when the corresponding distributed system try to increase its size by including more servers and data.
- According to the CAP theorem, distributed systems can provide two out of three desired features. These features are consistency, availability, and partition Tolerance.

STOP TO CONSIDER

Availability is a desired feature of distributed system which means that each active server will receive response forall its requests for resources even if one or more servers are disconnected or failed due to some reason.

Partition toleranceis another desired feature of distributed systems which means that a distributedsystem can be able to perform its functions even if one or more servers are temporarily disconnected or failed due to some reason like network failures or any type of system errors.

5.4.2 Techniques for achieving ACID Properties

We have already discussed the challenges in achieving ACID properties in case of distributed transactions.Different techniques to achieve ACID properties in distributed transactions are introduced in the following points.

- Atomic commit protocols are used to maintain atomicity and consistency in case of distributed transactions. In general, two types of Atomic commit protocols are available and these are Two-Phase Commit (2PC) protocol and Three-Phase Commit (3PC) Protocol. Detailed discussions of these protocols are provided in the later part of this unit.

- Concurrency control mechanisms are provided to achieve transaction isolation in case of concurrent distributed transactions. Locking mechanism, Concurrency control based on Timestamp ordering and Optimistic concurrency control are the general concurrency control approaches. Detailed discussion about concurrency control is provided in the later part of this unit.
- Recovery mechanisms and permanent storage can be utilized to maintain durability in distributed transactions.

Check Your Progress

1. Fill in the blanks

(a) ______ transactions provide a top-level transaction to open sub-transactions.

(b) The four ACID properties are _____, ______, ______, and ______.

(c) Atomicity in distributed transactions can be achieved by applying _____ protocols.

(d) _____ can be achieved in distributed system by applying Recovery mechanisms.

(e) _______ transaction completes only one request for an object at a time.

5.5 ATOMIC COMMIT PROTOCOLS

Atomicity of ACID properties can be achieved by atomic commit in case of distributed transactions. Atomic commit means either all operations within each of the distributed transactions will be successfully committed in different servers or the entire transaction will be aborted.Atomic commit can be implemented in distributed systems by using a transaction coordinator and an atomic commit protocol. A transaction coordinator is a specific server available in a distributed system. Its job is to coordinate execution of the distributed transactions and to achieve atomicity property by following an atomic commit protocol. The simplest atomic commit protocol is the One-Phase commit protocol where the transaction coordinatorsends the commit or abort request repeatedly to all of the servers that are involved in a distributed transaction. If all servers acknowledged that they can commit their parts related to that transaction then the entire transaction will be committed on all the corresponding servers. On the other hand if one or more servers are not able to commit their parts then the entire transaction will be aborted.The advantage of this protocol is its simplicity. But practically it cannot be used in case of distributed transactions because when failure occurs then it may be possible that one or more servers may not be able to communicate with the coordinator to acknowledge the commit request.In some situations, one or more server may not able to perform their parts related to a distributed transactionbut according to this protocol, it is not possible to send this information to the coordinator.So, In general, atomic commit can be implemented in distributed systems by using one of the two atomic commit protocols that are Two-Phase Commit (2PC) protocol and Three-Phase Commit (3PC) protocol.

5.5.1 Two-Phase Commit (2PC) Protocol

The Two-phase commit protocol consists of two phases. The first phase is referred as the Voting phase and in the second phase, commit or abort transaction is performed depending upon the result of the Voting phase. Steps of the each phase are presented as follows.

Phase I:

- **Step 1:** The transaction coordinator sends a request to all participating servers of a distributed transaction to vote for commit or vote for aborttheir parts of the transaction.
- **Step 2:** When a participating server of the transaction receive the request from the transaction coordinator then either it inform the coordinator that it is ready to commit its part of the transaction or it replies to the coordinator with the vote for abort the transaction and locally abort the transaction right away.

Phase II:

- **Step 1:** The transaction coordinator gathers all the messages from the participating servers of the transaction. If the coordinator finds that all participants are ready to commit their parts of the transactions then it decides to commit the transaction and sends a global commit request to all the participants. On the other hand if the coordinator finds that one or more participants are voted for abort the transaction then it decides to abort the entire transaction and sends global abort request to all participants.

- **Step 2:**In this step, all participating servers ready to commit the transaction wait for the global commit or global abort request from the coordinator. If they receive global commit request then they locally commit the transaction and confirm the coordinator about their commit operation. On the other hand, if they receive global abort requests then they locally abort the transaction.

Two-phase atomic commit protocol is a simple and flexible protocol to implement atomicity property in case of distributed systems. Communication overhead is increased with the use of this protocol due to the requirement of communications between the coordinator and all the participants. A possible situation during the execution of Two-phase commit protocol is that one or more participating servers of a transaction may be crashed during the communications with the coordinator. Such type of situation can fail the Two-phase atomic protocol. As a solution to this problem, each of the participants saves all information related to the Two-phase protocol in permanent storage so that it can be used to replace crashed participants. It may also be possible that the communication between a participant and the coordinator may be lost due to network fail or server crash. As a result, some processes may be blocked indefinitely. This situation can be prevented by using timeouts.

The major disadvantage of this protocol is the possible occurrence of the blocking problem. Let us consider a possible scenario to understand the blocking problem where a participating server of a transaction is ready to commit and waiting for the global commit or global abort request from the coordinator. At this point, unfortunately the coordinator crashes and the participant cannot be able to proceed as it is continuously waiting for the decision (commit or abort) from the coordinator. Now the participant is in an uncertain state and it blocks the system. This uncertain state of the participant will be continued till the replacement of the crashed coordinator.This possible problem is referred as the blocking problem. A possible solution to this problem is to provide a mechanism so that the participants can also receive the decision (commit or abort) cooperatively. But in a possible situation, if all the participants are in uncertain state then this strategy also fails to solve the blocking problem.

5.5.2 Three-Phase Commit (3PC) Protocol

Three-phase commit protocol is actually the upgraded version of Two-phase commit protocol. The blocking problem of Two-phase commit protocol may be solved by using Three-phase commit protocol and reliability of the commit operation can be improved. Three-phase commit protocol consists of three phases that are presented as follows.

Phase I: Phase I of Three-phase commit protocol is similar to the phase I of Two-phase commit protocol. So, in this phase, the transaction coordinator sends a request to all participating servers of a distributed transaction to vote for commit or vote for abort their parts of the transaction.When a participating server of the transaction receive that request from the transaction coordinator then either it inform the coordinator that it is ready to commit its part of the transaction or it replies to the coordinator with the vote for abort the transaction and locally abort the transaction right away.

Phase II: The coordinator gather all votes received from all the participants. If it finds that all participants vote for commit operation then it sends a pre-commit request to all the participants. Otherwise, it aborts the transaction and send transaction abort request to all the participants that voted for commit operation. When a participant receivesthe pre-commit request from the coordinator then it acknowledges it. On the other hand, when a participant receivesthe transaction abort request from the coordinator then it locally abort its part of the transaction.

Phase III: In this phase, the coordinator gathers all the acknowledgments for the pre-commit request received from the participants. When the coordinator receives all the acknowledgments then it sends a global commit request to all the participants. All participants wait for the global commit request from the coordinator. When a participant receives the global commit request then it commits its part of the transaction.

In the Three-phase commit protocol, an additional phase (Pre-commit phase) is included to decrease the possibility of occurrence of the blocking problem. In this case, this protocol is better than the Two-phase commit protocol. But on the other hand, due to the additional phase, the implementation complexity and communication overhead will be increased in the system when the Three-phase protocol is used to implement atomic commit operation.

5.5.3 Trade-offs of Atomic Commit Protocols

We have learnt that atomic commit protocols are very important in case of distributed transactions to maintain consistency and reliability in the distributed systems. But these protocols alsoincludedifferent trade-offs. In general, following trade-offs are involved in atomic commit protocols.

- Atomic commit protocols ensure system consistency and reliability but communication overhead is increased in the system due to these protocols because they require several communications among the coordinator and the participating servers. This overhead may impact the overall performance of the distributed system.
- Atomic commit protocol implement the atomicity property in case of distributed systems but in this process the system performance is compromised because sometimes atomic commit operation of a transaction may have to wait for slow participating servers or controlling network failures.

- Implementation of atomic commit protocolsin a distributed system may require additional resources like storage media, computational resources, network facilities etc.
- We have already learnt that the Two-phase commit protocol is a simple and flexible atomic commit protocol. But the blocking problem may occur when this approach is used to perform atomic commit operation in a distributed system. Now, to deal with the blocking problem, we can use the Three-phase commit protocol. But this approach is more complex than the Two-phase commit protocol and due to this approach; communication overhead is also increasedin the system.
- Atomic commit protocols can be used to implement atomicity in case of distributed transactions that involve any number of participating servers. But if the number of participating servers of a distributed transaction is increased then the communication overhead in the system will also be increased.

5.6 ISOLATION LEVELS

Isolation is one of the ACID properties of distributed transactions. We have already learnt that according to this property, operations of a distributed transaction should be performed in isolation from the operations of other simultaneous distributed transactions. Now, the degree of this isolation is referred as isolation level. An isolation level is termed as low when multiple concurrent distributed transactions can be able to access the same resource at the same time. As a result, consistency in the system may be lost. On the other hand, an isolation level is termed as high when only one of the multiple concurrent transactions can be able to access a resource at a time. In this case, consistency in the system will be high but the system requires more number of resources.

There are four basic isolation levels defined in case of transactions and these are presented in the following points.

- The lowest isolation level is Read Uncommitted. This isolation level allows multiple concurrent distributed transactionsto read the uncommitted changes that are performed by other concurrent transactions. Dirty read, non-repeatable read and phantom read may be occurred due to this isolation level.

- Read Committed is the next higher isolation level. This isolation level allows multiple concurrent distributed transactions to read only committed changes that are performed by other concurrent transactions. As a result, dirty read does not occur in case of this isolation level. But non-repeatable read and phantom read may be occurred due to this isolation level.
- Repeatable Read is the higher isolation level than Read Committed.This isolation level allows only one distributed transaction at a time to readcommitted changes performed by other concurrent transactions. It means, in this level, a distributed transaction holds both read and write lock on a resource. So,both dirty read and non-repeatable read do not occur due to this isolation level.But phantom read is possible in case of this isolation level.
- The highest isolation level is serializable. In this isolation level, the execution of distributed transactions istotally isolated to each other. All transactions are executed serially by maintaining an order. So, dirty reads, non-repeatable reads, and phantom reads can be avoided in case of this isolation level.

STOP TO CONSIDER

Dirty read: Let us consider a situation where a transaction, R updates some data but it does not perform the commit operation till now. Now, another transaction, S is allowed to read that uncommitted data. At this moment, if the transaction,R aborts its operations and rollback the modifications that are performed by its operations then it leads to a situation where the transaction,Sreadsdata that is not exist in the system at present. This situation is termed as Dirty read.

Non Repeatable read: It is a situation where a transaction is allowed to read the same data two times and it obtains different valuesin each time.

Phantom Read: It is a situation where two identical queries are executed but the data obtained by the second query is different from the data retrieved by the first query.

5.7 CONCURRENCY CONTROL IN DISTRIBUTED TRANSACTIONS

Concurrency control in distributed systems is a mechanism that is utilized to control the execution of multiple concurrent distributed transactions when they are required to access shared resources so that consistency and integrity can be maintained in the system. Isolation property can be achieved by using concurrency control in case of distributed transactions. In general, following techniques are used for concurrency control in distributed systems.

5.7.1 Locking Mechanism

Locking mechanism can be used to implement concurrency control in case of distributed transactions. In this approach, when a transaction is accessing an object then that object is locked so that other concurrent transactions cannot be able to access it. In distributed systems, locks are managed locally. Each server in a distributed system uses a lock manager to handle locks on its resources independently. The lock manager is responsible to grant a lock on an object of the server. If an object required by a transaction is already locked then the lock manager is responsible to keep the resource request in waiting mode. In a distributed system, a lock manager can release the lock from an object only when the corresponding transaction is committed in all its participating servers. On the other hand, if the transaction is aborted then the lock is released after the phase one of atomic commit protocol. Other concurrent transactions can try to access the object after the release of lock on it. Due to locking mechanism, distributed deadlock may be developed in the system.

5.7.2 Concurrency Control Based on Timestamp Ordering

In this approach, each distributed transaction is assigned with a global unique timestamp. When a transaction begins its execution in a distributed system then the global timestamp is assigned to it by the coordinator inthe first server where it is opened. This timestamp is passed to each coordinator in the participating servers of the transaction. The distributed transactions are executed serially depending on their timestamps. It means older transactions are executed before younger transactions.All participating servers of different transactions in a distributed system have to work together so that transactions are executed serially depending on their timestamps.

5.7.3 Optimistic Concurrency Control

In 1981, H. T. Kung and John T. Robinson had proposed the Optimistic concurrency control approach to control concurrent transactions so that consistency and integrity can be maintained in a system.In the Optimistic concurrency control approach,concurrent distributed transactions within a distributed system are controlled without using locks on resources. In this approach, when a transaction started, it is supposed that it will notconflictwith other concurrent transactions when shared resources will beaccessed by them. When the transaction completes its operations then the system checks for any conflict that may occurred duringits operations. If it is found that a conflict has occurred then some transaction is aborted to resolve the conflict.The Optimistic concurrency control approach consists of the following phases.

- **Phase I:**In the phase I, eachtransaction is provided a tentative version of each of the resources that it requires to access.A tentative version of a resource is the latest committed copy of that resource. At first each transaction performs read operation on the tentative versions of the corresponding resources.Then as per requirement write operations are performed by each transaction to update the tentative version of the corresponding resources. When multiple concurrent distributed transactionsare required to perform write operation on the same resource then in this phase, multiple tentative values for that same resource may be created. Two records are maintained for each concurrent transaction in this phase. One record contains the resources read by the transaction and the other record contains the resources updated by the transaction.
- **Phase II:**The phase II is referred as the validation phase. The validation phase starts when a transaction completes its operations and the request to close that transaction is received.In this phase, the system validated that the operations of the transaction do not conflict with the operations of other concurrent transactions when they are accessing the same resource. When a distributed transaction enters the validation phase then a global transaction number

is assigned to it. So, all the concurrent transactions available in the validation phase are serialized depending upon the order of their transaction numbers. In case of distributed transactions, multiple independent servers are responsible for the validation of a transaction. The servers whose resources are accessed by a distributed transaction are responsible for the validation of that transaction. The validation process at all the servers of a distributed system is performed during the phase I of the Two-phase atomic commit protocol.
The validation process is performed by satisfying three read-write conflict rules. These three rules are based on the conflicts between the operations of two concurrent transactions. Let us consider, the validation test will be performed on the transaction, S and on the other hand, T_i is one of the concurrent transactions. Then the validation will be successful for the pair of transactions, (S, T_i) if the following read-write rules are satisfied.

- ➢ If S has performed write operation on some resources then T_i must not perform read operation on those resources.
- ➢ If $T_i$has performed write operation on some resources then S must not perform read operation on those resources.
- ➢ T_i must not perform write operation on the resources that are written by S and S must not perform write operation on the resources that are written by T_i.

If the validation is successful then the transaction goes to the third phase. Otherwise, a conflict resolution approach is used. In general, one of the transactions involved in the conflict is aborted to resolve that conflict.

- **Phase III:** If the validation of a distributed transaction is successful then all updated values available in the tentative versions are permanently written to the original resources. Then the transaction is committed. A distributed transaction that performs only read operation is committed immediately after the validation phase.

Optimistic concurrency control approach can be used instead of locking mechanism to reduce the overhead that is introduced in the system due to lock maintenance. We already know that the use of locks may lead to deadlock situations in a system. So, occurrence of deadlocks may be reduced by using Optimistic concurrency control approach. Finally, if a few conflicts are possible among transactions in a distributed system then Optimistic concurrency control approach can provide better concurrency and performance in the system.

Implementation of Optimistic concurrency control approach is more complex than the other concurrency control mechanisms. This approachmay increase the transaction aborts in a systemwhere a large number of conflicts among transactions are developed. It can significantly degrade the system performance.

5.8 HANDLING DURABILITY IN DISTRIBUTED SYSTEM

We have already learnt about Durability which is one of the ACID properties. Durability in distributed systems confirms that the changes generated by a transaction in the system will be permanent after successful commit operation of the transaction. It means these changes will not be lost from the system in case of any type of failures, server crashes and system errors.So, permanent storage media is required to record objects so that Durability can be achieved. When a distributed transaction is committed then the changes made by it are saved in the permanent storage. Durability in case of distributed transactions is crucial for the maintenance of accuracy and reliability in the system. Durability can be handled in distributed system by using different recovery mechanisms. The recovery mechanisms are used to restore a crashed server or an erroneous server with the most recent saved versions of its objects from permanent storage.Two types of recovery approaches are available in distributed systems that are backward recovery and forward recovery. In case of backward recovery if the current state of a system become inconsistent then the recovery process will bring the system to its previous consistent state. On the other hand, in case of forward recovery, the recovery approach tries to bring the system with inconsistent state to a new consistent state from where the system can continue to perform its tasks.

General approaches to maintain Durability in a distributed system are discussed in the following points.

- Multiple copies of data can be saved in multiple servers so that data can be protected from server failures or crash situations. Regularly data backups can also be performed to prevent data loss from the system.
- To implement backward recovery, a distributed system must regularly record its consistent global states in permanent storage. This process is termed as Check pointing. A consistent global state of a distributed system is termed as a distributed snapshot. Distributed snapshots are constructed from the local consistent states recorded in multiple servers of a distributed system.
- Each server in a distributed system maintains a log that contains the records of all transactions that are executed by the server. Record of a transaction in a log consists of objects' values, entries of the transaction status and the references and the values of all the objects that are changed by the transaction.The order of the records in the log of a server is dependent upon the order in which the transactions are executed at the server. When a server is crashed or failed then recovery of the system to a consistent state can be performed by utilizing the log of that server. In this process, all committed transactions can be recovered by using the records available in the log.

Check Your Progress

2. Choose the correct option

(a) Which of the following is performed at the first phase of the Two-phase commit protocol?

(i) Participating servers vote for commit or abort operation.

(ii) The transaction coordinator send request to all participating servers to vote for commit or abort operation.

(iii) The transaction coordinator vote for commit or abort operation.

(iv) None of the above

(b) Which of the following problem may be solved by using Three-phase commit protocol?
(i) The blocking problem.
(ii) Deadlock.
(iii) Communication delay.
(iv) All of the above.

(c) Which of the following will not occur in case of Repeatable Read isolation level?
(i) Dirty read
(ii) Non-repeatable read
(iii) Phantom read
(iv) Both (i) and (ii)

(d) Which of the following is not used to control concurrency?
(i) Locking mechanism.
(ii) Timestamp ordering
(iii) Transaction recovery
(iv) All of the above

(e) Multiple copies of data can be saved in multiple servers to achieve____.
(i) Atomicity
(ii) Integrity
(iii) Efficiency
(iv) Durability

5.9 SUMMING UP

- A distributed transaction is associated with multiple servers located in different physical sites. A distributed transaction may access multiple objects that are managed by different servers. Either all operations of a distributed transaction complete successfully in different servers or all operations are aborted by corresponding servers.
- Flat distributed transactions are simple distributed transaction where multiple objects are requested that are managed at multiple servers. A flat distributed transaction completes only one request for an object at a time.

- In case of a Nested distributed transaction, a top-level transaction is available which can open nested sub-transactions. Then each sub-transaction can open the next level of sub-transactions. This process can be continued as per requirement. Sub-transactions available at a similar level can execute concurrently.
- According to the Atomicity property, either all operations of each distributed transaction complete successfully or the transaction is aborted and no change will be permitted in the system. Atomic commit protocols are used to maintain atomicity and consistency in case of distributed transactions. In general, two types of Atomic commit protocols are available and these are Two-Phase Commit (2PC) protocol and Three-Phase Commit (3PC) Protocol.
- According to the Consistency property, the system will move to a consistent state from a consistent state after the successful competition of each distributed transaction.
- According to the Isolation, the execution of each distributed transaction is isolated from other concurrent distributed transactions so that concurrent transactions are executed without interfering each other's operations.
- According to Durability property, the changes happened to a distributed system after the successful completion of a distributed transaction is permanent.
- The Two-phase commit protocol consists of two phases. The first phase is referred as the Voting phase and in the second phase, commit or abort transaction is performed depending upon the result of the Voting phase.
- Three-phase commit protocol is actually the upgraded version of Two-phase commit protocol. The blocking problem of Two-phase commit protocol may be solved by using Three-phase commit protocol and reliability of the commit operation can be improved.
- An isolation level is termed as low when multiple concurrent distributed transactions can be able to access the same resource at the same time. On the other hand, an isolation level is termed as high when only one of the multiple concurrent transactions can be able to access a resource at a time.

- Four basic isolation levels are Read Uncommitted, Read Committed, Repeatable Read and Serializable.
- Concurrency control in distributed systems is a mechanism that is utilized to control the execution of multiple concurrent distributed transactions when they are required to access shared resources so that consistency and integrity can be maintained in the system. Concurrency control mechanisms are provided to achieve transaction isolation in case of concurrent distributed transactions. Locking mechanism, Concurrency control based on Timestamp ordering and Optimistic concurrency control are the general concurrency control approaches.
- Durability can be handled in distributed system by using different recovery mechanisms. The recovery mechanisms are used to restore a crashed server or an erroneous server with the most recent saved versions of its objects from permanent storage.
- In case of any server failure, backward recovery will bring the system to its previous consistent state.
- In case of forward recovery, the recovery process tries to bring the system with inconsistent state to a new consistent state from where the system can continue to perform its tasks.

5.10 ANSWERS TO CHECK YOUR PROGRESS

1.

(a) Nested distributed
(b) Atomicity, Consistency, Isolation, Durability
(c) Atomic commit
(d) Durability
(e) A flat distributed

2.

(a)(ii) The transaction coordinator send request to all participating servers to vote for commit or abort operation.
(b)(i) The blocking problem.
(c)(iv) Both (i) and (ii)
(d)(iii) Transaction recovery
(e)(iv) Durability

5.11 POSSIBLE QUESTIONS

1) Write down the characteristics of distributed transactions.
2) How ACID properties can be achieved in distributed transactions?
3) Explain Two-phase atomic commit protocol.
4) Write down the difference between Two-phase commit protocol and Three-phase commit protocol.
5) How concurrency control can be implemented in distributed systems?
6) Write a short note on Isolation levels in distributed transactions.
7) How Durability can be handled in distributed systems?

5.12 REFERENCES AND SUGGESTED READINGS

- Tanenbaum, Andrew S. "*Distributed Operating Systems*" *(1995).*
- Coulouris, George, Jean Dollimore, and Tim Kindberg. "Distributed Systems: Concepts and Design Edition 4." (2005).
- Tanenbaum, Andrew S., and Maarten Van Steen. "*Distributed systems:Principles and Paradigms Edition 2." (2007).*
- https://www.ibm.com/topics/cap-theorem. Accessed on: 07-08-2024
- https://www.eecs.harvard.edu/~htk/publication/1981-tods-kung-robinson.pdf Accessed on: 18-08-2024

×××

UNIT: 6

REPLICATION AND CONSISTENCY IN DISTRIBUTED SYSTEMS

Unit Structure:

6.1 INTRODUCTION

Replication in distributed systems means making multiple copies of data or objects and storing them on different servers or locations. This helps improve system availability, performance, and fault tolerance. If some servers fail or go offline, the system can still work because other servers have the same data.

Consistency in distributed systems means, making sure that all servers or replicas show the same data at any given time, so users always see the same information. However, ensuring strong consistency is tricky because there's a trade-off with availability, especially if some servers are temporarily disconnected or the network has issues (as explained by the CAP theorem).

In short, replication and consistency are key for keeping distributed systems reliable and efficient, but there's often a balance between having consistent data and keeping the system running smoothly when problems arise.

6.2 UNIT OBJECTIVES

After going through this unit, you will be able to:

- understand the need for Data Replication.
- identify Types of Replication Strategies.
- explore Object Replication Techniques.
- learn Different Consistency Models.
- understand Trade-offs between Consistency and Availability.

6.3 DATA REPLICATION IN DISTRIBUTED SYSTEMS

Data replication means making and storing copies of data on different servers or locations in a distributed system. The main reason for doing this is to make the system more reliable, faster, and able to keep working even if some servers fail or cannot be reached. With multiple copies of data, the system can continue running smoothly. There are different ways to replicate data, like primary-backup, where one server has the main copy, or multi-master, where many servers have equal copies and can update them. While replication makes the system more reliable, it can be tricky to make sure all copies of the data stay the same across servers.

6.3.1 Need for Data Replication

The need for data replication in distributed systems centers around the following points:

Increased Availability: By keeping multiple copies of data on different servers, the system can keep running even if some servers fail. This ensures users can still access their data despite problems like hardware failures or network issues.

Fault Tolerance: Replication means there are backup copies. If one server goes down, another copy can take over, lowering the chance of losing data and keeping the system working.

Better Performance: Having data stored closer to users helps them get information faster, reducing delays. This is especially useful in systems spread across large areas.

Load Balancing: Multiple data copies allow the system to share the workload between different servers. This prevents any one server from getting overloaded, helping the system run smoothly.

Data Safety: With several copies, there's a lower risk of permanently losing data. If one copy gets damaged or lost, the others can still provide the correct information.

6.3.2 Types of Data Replication

In distributed systems, data replication can be implemented in several ways, depending on how data is synchronized and managed across different servers. The common types of data replication are:

- Primary-Backup Replication,
- Multi-Master Replication,
- Chain Replication,
- Distributed Replication.

Now, let’s discuss each of the types one by one.

6.3.2.1 Primary-Backup Replication

Primary-Backup Replication is a common method used in distributed systems to make sure the system is always available, reliable, and consistent with data. In this method, one main server, called the primary (or leader), stores the main version of the data. One or more other servers, called backups (or followers), store copies of this data. The primary server handles all changes or updates to the data, while the backup servers simply keep their copies updated. If the primary server stops working, one of the backups can take over to keep the system running.

The working of Primary-Backup Replication is follows:

- The primary server handles all read and write operations from clients. Whenever there is a write or update request, it modifies its local copy of the data and sends the updated information to all backup servers. It ensures that changes are propagated to backups to maintain consistent copies across the system.
- Backup servers receive updates from the primary and keep their copies in sync. They do not handle client write operations directly; they simply mirror the state of the primary. In the event of the primary server failing, one of the backups can be promoted to the new primary, ensuring the system remains operational.
- If the primary server crashes or becomes unavailable, the system detects this failure and promotes one of the backups to take over as the new primary. The new primary continues handling client requests and updating the remaining backup servers. Thus, it process ensures that the system remains available even in the event of failures, but there might be some delay during the switch from the primary to the backup (failover).

Now, consider a banking application where customer accounts and transactions are stored across multiple servers to provide fault tolerance and ensure high availability. In this setup:

- The primary server handles all transaction processing, such as deposits, withdrawals, and balance updates. It records all the latest account information and ensures data integrity.
- Multiple backup servers store replicas of the account data, which are updated by the primary server. These backup servers are

passive—they do not process transactions but receive updates from the primary.

- Now, if the primary server fails (e.g., due to hardware failure or a network issue), one of the backup servers is promoted to the new primary. It takes over transaction processing, ensuring that customers can continue using the banking system without interruption. The remaining backups continue to receive updates from the new primary.

Advantages of Primary-Backup Replication:

- The system remains available even when the primary server fails, as backups can quickly take over.
- Since only the primary handles write operations, the system ensures a consistent state across all replicas, provided that synchronous replication is used.
- The primary-backup model is relatively easy to understand and implement compared to more complex replication schemes like multi-master.
- Write operations are handled by a single server (primary), preventing issues like conflicting updates or race conditions between replicas.

Disadvantages of Primary-Backup Replication:

- In case of a primary server failure, there is a delay while the system detects the failure and promotes a backup to the new primary. This failover process, although automatic, can cause a temporary service outage.
- Since all write operations go through the primary server, it can become a bottleneck when handling a large number of updates. This model may not scale well for systems with high write loads.
- If asynchronous replication is used, backups may lag behind the primary, meaning they might not have the most up-to-date data. In the event of a primary failure, the newly promoted primary could have slightly out-dated information.
- The primary server represents a single point of failure for write operations. If the primary fails, even though a backup can take over, there is still a brief interruption in service.

6.3.2.2 Multi-Master Replication

Multi-Master Replication is a replication strategy used in distributed systems where multiple servers, known as masters, can accept write operations and maintain copies of the same data. This model enhances availability and performance by allowing updates to occur on any of the master nodes, rather than relying on a single primary server. However, it comes with its own set of challenges, particularly around data consistency and conflict resolution. By carefully designing the replication strategy and conflict-handling mechanisms, organizations can leverage the benefits of this model while mitigating its drawbacks.

The working of Multi-Master Replication is follows:

- In this model, several servers act as masters. Each master can accept read and write requests from clients. Because of this, the system can continue to operate and serve requests even if one or more masters fail.
- When a master server receives a write request, it processes the update and then replicates this change to other master servers. This can happen in various ways, such as:

  Synchronous Replication: The update is sent to all masters, and they must acknowledge receipt before the operation is considered complete. This ensures data consistency but can increase latency.

  Asynchronous Replication: The update is sent to other masters without waiting for their acknowledgment. This approach improves performance but may lead to temporary inconsistencies between replicas.
- Since multiple masters can accept write operations, conflicts may arise if two or more masters attempt to update the same piece of data simultaneously. To handle this, the system must implement a conflict resolution strategy, which could include:
  - The most recent write operation takes precedence.
  - Each write includes a version number, and the system uses this number to determine the most recent update.
  - The application logic decides how to resolve conflicts based on specific business rules.

Advantages of Multi-Master Replication:

- The system remains operational even if one or more master servers go down. This redundancy enhances overall system reliability.
- With multiple masters available to handle requests, the system can distribute the load more evenly, reducing response times for users.
- In scenarios where users are spread across different regions, having multiple masters can bring the data closer to users, further reducing latency.

Challenges of Multi-Master Replication:

- **Complexity**: Managing multiple masters and ensuring data consistency can be complex. This requires sophisticated conflict resolution and synchronization mechanisms.
- **Data Inconsistency**: The asynchronous nature of updates may lead to temporary inconsistencies among the replicas, making it challenging to guarantee a uniform view of data at all times.
- **Overhead**: The need for synchronization and conflict resolution can add overhead to the system, potentially affecting performance.

6.3.2.3 Chain Replication

Chain replication is a structured method used in distributed systems to manage the replication of data across multiple nodes, ensuring strong consistency. It is particularly effective for systems that require high availability while maintaining strict order and consistency of data. In chain replication, the nodes are organized in a sequence or chain, and data is propagated through this chain from start to end.

The working of Multi-Master Replication is follows:

The Architecture:

- Nodes are arranged in a linear sequence, often referred to as a chain.
- There is a head node, intermediate nodes, and a tail node.

- Write operations always start at the head and move sequentially through the chain to the tail.

Writing Operations:

- A client that wants to update the data sends a write request to the head node.
- The head node processes the write and then forwards it to the next node in the chain.
- This process continues until the tail node is updated.
- Once the tail node is updated, it sends an acknowledgment back through the chain, confirming that the update has been fully replicated.
- The acknowledgment to the client is only sent when the tail has processed the write, ensuring that all nodes in the chain have consistent data.

Reading Operations:

- Typically, read requests are handled by the tail node.
- This ensures that clients always read the most up-to-date data because the tail node receives all updates last, after they have been processed by every other node in the chain.

Handling Failure:

- If a node in the chain fails, the chain is reconfigured to bypass the failed node.
- The system can add new nodes to the chain to replace the failed one, maintaining redundancy and ensuring consistency.

Now, let's try to understand this with the help of an example. Consider a distributed system where multiple copies of a data object, say "X," are maintained across three nodes: Node A as **Head node**, Node B as **Intermediate Node**, and Node C as **Tail Node**.

Now, the steps are as follows:

- **Write Operation**:
    - A client wants to update the value of "X" to 100.
    - The client sends a write request to **Node A**.

  - **Node A** updates "X" to 100 and forwards the update to **Node B**.
  - **Node B** updates "X" to 100 and forwards it to **Node C**.
  - **Node C** updates "X" to 100 and sends an acknowledgment back to **Node B**, then to **Node A**, and finally to the client.
  - Only when the update is confirmed at **Node C** (the tail) is the operation considered successful, ensuring all nodes have consistent data.
- **Read Operation**:
  - A client requests to read the value of "X."
  - The read request is directed to **Node C**, which provides the value 100.
  - Since **Node C** is the last node in the chain to be updated, it always has the most recent value of "X."
- **Failure Handling**:
  - If **Node B** fails, the system reconfigures to bypass **Node B**, and the chain becomes **Node A → Node C**.
  - A new node, **Node D**, could be added later, making the new chain **Node A → Node D → Node C**.
  - This ensures that redundancy is maintained and that the system can continue functioning even if nodes fail.

Advantages of Chain Replication:

- Since all updates are processed sequentially through the chain, all nodes have the same version of data once an acknowledgment is sent, ensuring strong consistency.
- Reads are served from the **tail node**, ensuring that the most recent data is always provided to clients.
- The chain can be reconfigured in the event of a node failure, and new nodes can be added without interrupting the service.

Disadvantages of Chain Replication:

- Since write operations must propagate through each node in the chain before being acknowledged, the write latency can be higher compared to other replication methods.

- The **head** and **tail nodes** can become bottlenecks. The **head** handles all incoming writes, while the **tail** handles all reads.
- If a node fails, the reconfiguration of the chain might take some time, potentially affecting the availability of the system.

6.3.2.3 Distributed Replication

Distributed replication means keeping multiple copies of data on different nodes or locations in a distributed system. Replication helps improve data availability, fault tolerance, reliability, and performance. By having multiple copies, the system can handle failures more effectively, provide faster data access, and ensure that data is still available even if some nodes fail.

There are types of Distributed Replication:

Synchronous Replication:

- All replicas must be updated before a write operation is considered complete.
- Guarantees that all replicas have consistent data, but may increase latency because of waiting for acknowledgments from all replicas.
- Example: A banking system where transaction consistency is critical.

Asynchronous Replication:

- The primary node acknowledges a write operation immediately, and the updates are propagated to replicas in the background.
- This approach offers better performance but might lead to temporary inconsistencies between replicas.
- Example: Social media applications where immediate consistency is not critical.

Partial Replication:

- Only a subset of the nodes in the distributed system contains copies of the data.
- Helps reduce storage costs and network overhead.
- Example: In large-scale distributed databases, only frequently accessed data might be replicated across all nodes, while less frequently accessed data is only partially replicated.

Full Replication:

- Every node in the distributed system contains a copy of the entire dataset.
- This ensures high availability but comes at the cost of increased storage and replication overhead.
- Example: Distributed ledger systems like block-chain where each node contains the entire ledger.

The working of Multi-Master Replication is follows:

Writing Operations:

- When a write request is made, the data must be updated on all or some of the replicas.
- In **synchronous replication**, the system waits until all replicas have been updated before acknowledging the write to the client.
- In **asynchronous replication**, the primary replica is updated first, and then the changes are propagated to secondary replicas.

Reading Operations:

- **Read** requests can be handled by any replica, which helps distribute the load and improves read performance.
- The system may use a **load balancer** to distribute read requests among replicas.

Consistency Models:

- **Strong Consistency**: Guarantees that all reads will return the latest value.
- **Eventual Consistency**: Guarantees that, given enough time, all replicas will eventually become consistent.
- **Causal Consistency**: Ensures that causally related updates are seen in the same order by all replicas.

Now, let’s try to understand this with the help of an example. Consider an online e-commerce platform with users distributed across the world. The platform wants to ensure that product information (e.g., availability, price) is accessible with minimal latency and remains highly available even during a network partition or failure. The system could use distributed replication as follows:

Nodes Setup:

- The system has several nodes: **Node A** (in North America), **Node B** (in Europe), **Node C** (in Asia).
- Each node contains a replica of the product database.

Write Operation:

- Suppose a seller in North America updates the price of a product.
- The update is first made to **Node A**.
- In **synchronous replication**, the update will then be propagated to **Node B** and **Node C**, and only after all replicas are updated will the seller receive an acknowledgment.
- In **asynchronous replication**, the update will be immediately acknowledged to the seller after it is made to **Node A**, while **Node B** and **Node C** are updated in the background.

Read Operation:

- If a customer in Asia wants to check the price of the product, the read request is directed to **Node C**, which is geographically closer.
- This reduces latency and ensures a faster response to the customer.

Advantages of Distributed Replication:

- By having multiple replicas, data remains accessible even if one or more nodes fail.
- Replication provides redundancy, allowing the system to continue operating in the event of hardware or network failures.
- Read requests can be distributed among different replicas, improving system scalability and reducing bottlenecks.
- Users can access the closest replica, reducing the response time for read operations.

Challenges of Distributed Replication

- **Consistency Management:** Maintaining consistency across replicas is challenging, especially in asynchronous replication, where temporary inconsistencies may occur.

- **Network Overhead:** Replicating data across nodes incurs additional network traffic, which can be a bottleneck in high-volume systems.
- **Conflict Resolution:** When multiple replicas accept write operations, conflicts can arise those needs to be detected and resolved.

6.4 OBJECT REPLICATION IN DISTRIBUTED SYSTEMS

Object replication in distributed systems refers to the practice of creating and maintaining multiple copies (or replicas) of an object (such as files, databases, or application components) across different nodes in the network. The main goal of object replication is to enhance system availability, fault tolerance, scalability, and performance. Replicating objects allows a distributed system to continue operating smoothly even in the presence of hardware failures or network issues.

6.4.1 Types of Object Replication

Object replication in distributed systems can be categorized into the following types based on how updates to replicas are handled:

Active Replication or Synchronous Replication:

- In active replication, all replicas of an object are kept identical by processing all requests in parallel.
- Every replica processes the same request, and the system ensures that they remain consistent with one another.
- This type is generally used in systems that need high availability and consistency, such as financial services.
- Example: In a distributed online banking application, every transaction can be processed at all replicas simultaneously to ensure that account balances are always consistent across nodes.

Passive Replication or Primary-Backup Replication:

- In passive replication, a primary replica processes all client requests and updates. After processing, the primary sends updates to the backup replicas.

- If the primary replica fails, one of the backups takes over as the new primary.
- This approach is often used where lower update latency is preferred, and it's acceptable for the replicas to be slightly out-of-date.
- Example: In a distributed file storage system, a primary server stores user files, and changes are replicated to backup servers after a write operation. If the primary server fails, a backup server becomes the new primary.

Lazy or Asynchronous Replication:

- In lazy replication, updates are made to a primary replica and propagated to other replicas asynchronously, meaning changes may take some time to reach all replicas.
- This can lead to temporary inconsistencies across replicas, but it improves system performance since nodes do not need to wait for acknowledgments from all replicas.
- Example: In content distribution networks (CDNs), when a new video is uploaded, the update is initially made on one server and then propagated to other servers gradually. This may result in some users getting the latest content before others, but overall system performance is improved.

6.4.2 Replication Consistency Models

Ensuring consistency among replicas is a key challenge in object replication. The different levels of consistency are:

Strong Consistency:

- All replicas must reflect the latest update immediately.
- This guarantees that any read operation returns the most recent write.
- Example: A distributed database that ensures every read returns the latest value, even if it means delaying the read request until all replicas are updated.

Eventual Consistency:

- Replicas will become consistent over time, but they may be temporarily out of sync.

- This is used when the system favours availability and partition tolerance over immediate consistency.
- Example: DNS servers use eventual consistency to update their records. Changes to DNS records might take time to propagate to all DNS servers.

Causal Consistency:

- Causal relationships between updates are preserved, meaning that if update A happens before update B, any node that sees update B must also see update A.
- This allows for a more relaxed consistency model without sacrificing the logical flow of updates.
- Example: Social media feeds can maintain causal consistency, ensuring that users see a comment only after they have seen the original post.

6.5 DATA-CENTRIC VS. PROCESS-CENTRIC REPLICATION

In distributed systems, replication can be broadly categorized into two types based on the focus of what is being replicated: data-centric replication and process-centric replication. Both approaches aim to improve reliability, availability, and performance, but they differ in terms of what they replicate and how they handle replication. Let's discuss both approaches in detail:

6.5.1 Data-Centric Replication

Data-centric replication involves replicating the data across multiple nodes or servers in a distributed system. The main goal is to ensure that data is accessible, consistent, and available even if some nodes fail. The characteristics of data-centric replication are**:**

- Data-centric replication aims to ensure that the same piece of data is available in multiple locations, thus enhancing data availability and fault tolerance.
- This approach may use various consistency models, such as strong consistency, eventual consistency, or causal consistency.

  - Guarantees that all replicas have the same value at any point in time.
  - Ensures that replicas eventually converge to the same value, even if there may be temporary discrepancies.

- Data is replicated in a way that reduces latency and optimizes read performance. This is especially useful in distributed databases or content delivery networks (CDNs) to provide fast access to data for users across different regions.

- Conflicts may arise during concurrent updates to the same data at different nodes. Conflict resolution mechanisms, such as version vectors or timestamps, are used to ensure consistency.

For example, in a distributed database, such as Cassandra or MongoDB, data-centric replication ensures that copies of data are stored in multiple nodes. When a client requests data, it is served from the nearest replica, improving response time. If an update is made, the changes are propagated to all replicas to maintain consistency.

6.5.2 Process-Centric Replication

Process-centric replication involves replicating the processes or computations across multiple nodes. In this approach, the main focus is on ensuring that the processing of requests can continue even if some nodes or processes fail. The characteristics of process-centric replication are as follows:

- Process-centric replication is used to enhance **fault tolerance** and **high availability** of services or processes, allowing the system to continue functioning even when individual processes fail.

- **Active &. Passive Replication**:

  - **Active Replication**: All replicas run the same process and receive the same input, ensuring that they produce the same output simultaneously. This method is suitable for achieving strong fault tolerance.

    - **Passive Replication (Primary-Backup)**: One process (primary) handles requests, and the state is replicated to backups. If the primary fails, one of the backups takes over.

- The replicas of the process need to maintain a synchronized state. This may involve periodically sending state updates from the primary process to the backup processes.
- Replicated processes ensure that, if one node handling a request fails, another replica can take over and continue processing without interrupting the overall service.

For example, in a distributed web service, a process-centric replication approach could involve having multiple replicas of a web server that handle client requests. If one web server fails, another server can seamlessly take over the requests, ensuring uninterrupted service.

6.5.3 Data-Centric vs.Process-Centric Replication

The comparison between data-centric and process-centric replication are presented in table 6.1.

Aspect	Data-Centric Replication	Process-Centric Replication
Focus	Replicates data across multiple nodes	Replicates processes or computations across nodes
Goal	Enhance data availability and consistency	Improve fault tolerance and service availability
Consistency Model	Strong, eventual, or causal consistency	Typically uses active or passive replication models
Access	Optimizes data access and read performance	Ensures that services remain available in case of failure
Conflict Resolution	Involves techniques like timestamps or version vectors	Ensures state consistency between replicated processes

Each approach addresses different aspects of distributed system design, and choosing between them depends on whether the goal is to provide data availability or service continuity. In practice, a combination of both types of replication is often used to achieve a robust and highly available distributed system.

6.6 CONSISTENCY MODELS IN DISTRIBUTED SYSTEMS

In earlier sections we had a very brief introduction to Consistency Models. Now, let's try to get a bigger picture.

In distributed systems, consistency models define the rules that govern how data replicas are kept in sync to maintain a coherent view across the entire system. These models establish the conditions under which different parts of the system can communicate, and determine how changes made by one process become visible to others. By setting specific guarantees on the visibility and propagation of updates, consistency models address distributed computing challenges like network delays, partial failures, and concurrency. The choice of a consistency model plays a crucial role in balancing consistency, availability, and performance, ensuring the system's reliability and predictable behavior even in the presence of faults.

In distributed systems, there are several types of consistency models, each suited to different requirements and use cases. Each model offers a different balance of consistency, availability, and performance, with its own advantages and limitations. The main types of consistency models are: **Strong Consistency, Sequential Consistency, Causal Consistency, Weak Consistency** etc. The choice of a specific consistency model depends on the needs of the system, such as the level of data consistency required and the tolerance for delays or failures.

6.6.1 Strong Consistency

Strong consistency means that every read operation always returns the latest value that has been written, no matter which copy of the data you access. To the user, it looks like there is only one version of the data, and all the changes are seen instantly. In a system with strong consistency, all nodes agree on the order of operations, meaning that everyone sees updates in the same sequence. This ensures that reads always provide the latest version, and writes are visible to all nodes immediately. However, achieving this level of consistency can lead to performance and availability challenges.

Let’s discuss the Principles of strong consistency and they are:

Single Copy: To users and applications, the distributed system behaves as if there is only one copy of the data. This means that any changes made to the data are immediately visible to all users.

Immediate Visibility: When a write operation is completed, all subsequent read operations will reflect that change. There is no delay or inconsistency in seeing the latest data.

Global Ordering: All operations (reads and writes) are executed in a total order that is agreed upon by all nodes in the system. This guarantees that all nodes see operations in the same sequence.

Synchronous Operations: Strong consistency often requires that write operations are completed before subsequent read operations can proceed, ensuring that all processes are up to date.

Advantages of Strong Consistency:

- Clients can always expect to see the most recent data, which simplifies application logic and improves user experience.
- Strong consistency helps ensure that the system maintains data integrity across distributed nodes, preventing issues like stale reads.
- Developers can write simpler code because they don’t need to account for inconsistencies that could arise from concurrent operations.

Disadvantages of Strong Consistency:

- Achieving strong consistency typically requires more communication between nodes, which can introduce latency and reduce throughput, particularly in large distributed systems.
- During network partitions or node failures, maintaining strong consistency can lead to reduced availability. If some nodes are unreachable, the system may block operations to ensure consistency.
- The need for synchronization can result in higher response times for operations, especially in geographically distributed systems where communication delays are significant.

6.6.2 Sequential Consistency

Sequential Consistency is a key concept in distributed systems that helps maintain a clear and consistent view of operations across different nodes. This model finds a middle ground between strict consistency and more relaxed models by ensuring that operations seem to happen in a specific order, even if they are actually processed at the same time on different nodes. According to this model, the results will always look like the operations (both read and write) were executed one after the other in a sequence, respecting the order in which each process performed them. It was introduced by Leslie Lamport and is considered less strict than stronger consistency models, making it easier to implement while still providing a reliable framework for understanding how operations relate to each other in a distributed system.

Let’s discuss the Principles of sequential consistency and they are:

Global Order: The results of operations must be consistent with some global ordering, meaning that even if operations are performed in parallel, they can be viewed as part of a single, orderly sequence.

Local Order: The local order of operations for each process must be preserved. If a process issues a read after a write, any subsequent reads must see the effects of that write.

Client View: From the client's perspective, the system behaves as if all operations were executed in some sequential order, which provides a straightforward understanding of the system's state.

Advantages of Sequential Consistency:

- Sequential consistency is easier for programmers to reason about than stronger models because it maintains a simple, intuitive view of the order of operations.
- Compared to strong consistency, sequential consistency allows for some level of concurrency, potentially improving performance and throughput.
- It offers a middle ground between strong consistency (which can be too restrictive) and weaker models (which can lead to confusion).

Disadvantages of Sequential Consistency:

- While it allows some concurrency, maintaining a sequential order can still introduce latency, especially in systems with high contention for resources.
- Implementing sequential consistency can be complex, especially in a distributed environment where nodes may have different latencies and failure rates.
- In cases of network partitions or failures, achieving sequential consistency may require delaying some operations, which can impact system availability.

6.6.3 Casual Consistency

Causal Consistency is an important model in distributed systems that helps keep track of the logical connections between different operations. It ensures that if one operation affects another, the first one will be visible before the second. This model finds a middle ground between high performance and keeping things coherent, making it useful for various applications. Introduced by Hutto and Ahamad in 1990, causal consistency ensures that all processes see related operations in the same order, but it’s less strict than models like sequential consistency, which require a complete order of operations. By focusing on the cause-and-effect relationships, causal consistency allows systems to run efficiently while still providing a clear view of the data, making it popular for collaborative and real-time applications where the order of actions is important.

Let’s discuss the Principles of casual consistency and they are:

Causal Relationship: In causal consistency, operations are seen as causally related if one operation can influence the outcome of another. For example, if Process A sends a message to Process B, any subsequent operations by B that depend on that message must see the effects of A's operation.

Event Ordering: The model ensures that if an operation O1 causally affects another operation O2, then all processes must see O1 before they see O2. However, operations that are not causally related can be seen in different orders by different processes.

Visibility: Each process has its own view of the operations, which may differ from other processes, but the visibility of operations respects causal relationships. This allows for more flexibility and improved performance compared to stronger consistency models.

Advantages of Causal Consistency:

- By allowing operations to be executed concurrently as long as they are not causally related, causal consistency enhances performance and scalability in distributed systems. This leads to lower latency and better resource utilization.
- Causal consistency aligns closely with how users typically reason about dependencies in their operations, making it easier for developers to build applications that behave predictably.
- Causal consistency provides more flexibility than strict or sequential consistency, allowing systems to tolerate delays and network partitions while still maintaining a coherent view of operations.

Disadvantages of Causal Consistency:

- Implementing causal consistency can be complex due to the need to track causal relationships and maintain metadata like vector clocks. This may introduce overhead and increase system complexity.
- Because operations are not globally ordered, processes may read stale data if they do not have visibility into all preceding operations. This can lead to scenarios where users see outdated information.
- Causal consistency does not guarantee that all operations will be seen in the same order by all processes, which can be problematic in certain applications requiring stronger guarantees.

6.6.4 Weak Consistency

Weak consistency is a model used in distributed systems that allows changes to data to be visible at different times across different nodes. Unlike stronger consistency models that require all copies of data to be in sync right away, weak consistency accepts that there may be temporary differences between data replicas. This approach helps improve performance and availability. When creating systems that use weak consistency, developers need to think carefully about what the application needs, balancing consistency, availability, and performance to meet user expectations. Applications that can handle short periods of inconsistency, like social media updates, messaging apps, and collaboration tools, are a good fit for this model. In summary, weak consistency is beneficial for distributed systems as it supports better performance and availability while allowing for some data differences, even though it can lead to challenges with data accuracy and complexity in development.

The key features of weak consistency are as follows:

- In weak consistency models, there are no guarantees that all replicas of data will reflect the most recent updates immediately. This means that a read operation may return stale data if it occurs before updates propagate to all nodes.
- Weak consistency allows for operations to be visible in different orders across different processes. Thus, different nodes may see the results of operations in varying sequences.
- Although weak consistency permits temporary inconsistencies, it often includes mechanisms that ensure that all replicas will converge to the same state eventually, given sufficient time without new updates.

Advantages of Weak Consistency:

- Weak consistency models typically allow for more efficient use of resources since they do not require synchronization among all nodes for every operation, leading to faster read and write operations.
- By not enforcing strict consistency, distributed systems can remain operational even during network partitions or failures, improving overall system availability.
- Weak consistency models are well-suited for large-scale distributed systems, where strict consistency can be challenging to maintain due to the sheer volume of data and interactions.

Disadvantages of Weak Consistency:

- Developers must handle the complexities introduced by potential stale data and ensure that applications can tolerate inconsistency, which can lead to increased programming effort.

- Weak consistency may lead to scenarios where conflicting updates occur, and resolving these conflicts can be challenging, especially in collaborative applications.
- Users may experience unexpected results since the data might not be immediately consistent across nodes, which can affect the user experience.

6.7 CONSISTENCY VS. AVAILABILITY

In distributed systems, the trade-offs between consistency and availability are central to system design and operation. These two concepts often compete with each other, especially in the context of the CAP theorem, which states that in the presence of a network partition, a distributed system can only guarantee either consistency or availability, but not both. Below is a detailed discussion of these trade-offs, their implications, and examples. But first just refresh what does consistency and availability means.

In distributed systems, **consistency** means that all nodes see the same data at the same time. When a write operation is performed, it should be visible to all subsequent read operations across the system. Strong consistency ensures that all replicas reflect the latest updates immediately, while eventual consistency allows temporary discrepancies between replicas, with the guarantee that they will eventually converge.

Availability refers to the system's ability to respond to requests, even in the face of failures or network partitions. An available system guarantees that every request receives a response, whether it is a success or an error, as long as a node is reachable.

6.7.1 Trade-offs between Consistency and Availability

Lets' discuss the trade-offs between Consistency and Availability.

1) **Consistency vs. Availability**:
   - **High Consistency, Low Availability**: Systems that prioritize strong consistency often require coordination among nodes to ensure that updates are synchronized. This can lead to reduced availability, especially during network partitions or node failures. For example, a banking application that requires all transactions to be immediately

reflected across all servers may temporarily deny access if some nodes are unreachable.

- **High Availability, Low Consistency**: Conversely, systems that prioritize availability might allow different nodes to serve different versions of the data temporarily. For instance, social media platforms often accept that users may see outdated information briefly while ensuring that the system remains operational. Users can still post updates even if some nodes are behind in syncing.

2) **CAP Theorem**:

- The CAP theorem asserts that it is impossible for a distributed data store to simultaneously provide all three guarantees: Consistency, Availability, and Partition tolerance (the system's ability to continue operating despite network partitions). When a partition occurs, systems must choose between maintaining consistency or availability. For example, during a partition, a system may either block writes (ensuring consistency) or allow writes to proceed at the risk of creating conflicts later (ensuring availability).

3) **Implications**:

- **User Experience**: The choice between consistency and availability can significantly impact user experience. In applications where immediate accuracy is critical (e.g., financial transactions), consistency is essential. However, for applications where user engagement is a priority (e.g., messaging apps), availability may take precedence.
- **Conflict Resolution**: When prioritizing availability over consistency, developers need to implement conflict resolution strategies to handle scenarios where different nodes have diverging data. For example, in a distributed database, if two nodes accept updates simultaneously, a merge process must determine which update is the "correct" one.
- **Latency**: Systems that emphasize strong consistency often experience higher latency because they must wait for all replicas to acknowledge the update before proceeding. On the other hand, systems that favor availability may have

lower latency, as they do not need to wait for all nodes to sync.

The trade-offs between consistency and availability in distributed systems are complex and depend on the situation. Designers need to carefully consider the specific needs of their applications, balancing what users expect with what the system can handle. Choosing between strong consistency and high availability often means making compromises, and understanding these trade-offs is crucial for creating effective distributed systems.

CHECK YOUR PROGRESS-I

1. State True or False:

a) Data replication in distributed systems improves availability and fault tolerance.

b) In Primary-Backup replication, the backup servers can directly handle client write operations.

d) Multi-Master Replication allows multiple servers to accept write requests, enhancing performance.

e) Chain replication always ensures the latest data is available for read requests.

2. Fill in the blanks:

a) In Multi-Master Replication, _________ is required to handle conflicts that may arise from concurrent updates.

b) _________ replication allows data to be written and read from any of the master servers.

c) In _________ replication, the update operation is considered complete only after all nodes have been updated.

d) In chain replication, the _________ node handles write requests.

e) Distributed replication provides _________ by having multiple replicas to handle read requests.

6.8 SUMMING UP

- Data Replication means Storing data copies across multiple servers for reliability, speed, and fault tolerance.
- Need for data replication are:
    - System remains operational despite server failures,
    - Backup copies reduce data loss risk,
    - Data closer to users for faster access,
    - Distributes workload across servers,
    - Reduced risk of permanent data loss.
- The types of Data Replication are namely Primary-Backup Replication, Multi-Master Replication, Chain Replication and Distributed Replication.
- Object Replication means maintaining multiple copies of an object across different nodes to enhance availability, fault tolerance, scalability, and performance.
- Type of Object Replications are:
    - Active Replication: All replicas process requests in parallel; ensures high availability and consistency.
    - Passive Replication: Primary replica processes requests; backups receive updates and take over on failure.
    - Lazy Replication: Updates propagated asynchronously; temporary inconsistencies allowed improving performance.
- In case of Strong Consistency, all replicas reflect the latest update immediately.
- In Weak Consistency, replicas eventually converge to the same state.
- In Causal Consistency, causal relationships are preserved between updates.
- In Data-centric Replication, data is replicated across nodes to improve availability and reduce latency. Uses consistency models like strong, eventual, or causal.
- In Process-centric Replication, processes are replicated to enhance fault tolerance. Can use active (all replicas run in parallel) or passive (primary-backup) replication.

- The Consistency Models are namely:
  - Strong Consistency: Ensures every read returns the latest value; high latency and availability challenges.
  - Sequential Consistency: Operations are seen in a specific order, maintaining local order of each process.
  - Causal Consistency: Maintains the order of causally related operations; improves performance and flexibility.
  - Weak Consistency: Allows temporary inconsistencies across nodes; improves performance and availability but requires applications to handle stale data.

6.9 ANSWERS TO CHECK YOUR PROGRESS

1. a) True b) False c) True d) True

2. a) conflict resolution b) Multi-Master c) synchronous

d) heade) load balancing

6.10 POSSIBLE QUESTIONS

Short Answer Type Questions:

1. What is the primary goal of data replication in distributed systems?
2. How does Multi-Master replication improve system availability?
3. What is the main advantage of chain replication regarding read operations?
4. How does synchronous replication differ from asynchronous replication?
5. What is a key disadvantage of chain replication in terms of write latency?

Long Answer Type Questions:

6. Explain the difference between active replication and passive replication in distributed systems.
7. Describe the advantages and disadvantages of strong consistency in distributed systems.

8. How does causal consistency differ from strong consistency and sequential consistency?
9. Explain how weak consistency can improve system performance and availability in distributed systems.
10. Compare and contrast data-centric replication and process-centric replication in distributed systems.

6.11 REFERENCES AND SUGGESTED READINGS

1. "Distributed Systems: Concepts and Design" by George Coulouris
2. “Designing Data-Intensive Applications" by Martin Kleppmann
3. “Distributed Systems: Principles and Paradigms" by Andrew Tanenbaum and Maarten Van Steen

×××

UNIT: 7

DISTRIBUTED FILE SYSTEMS

Unit Structure:

7. 1. Introduction

A Distributed File System (DFS) is a vital infrastructure component that allows data to be stored and accessed from numerous file servers and locations. It provides a seamless approach for programs to handle isolated data as if it were stored locally, while also letting users to access and exchange files across the network. The primary goals of DFS are to improve data availability, improve performance, and ensure data redundancy. This document describes the essential characteristics, difficulties, and needs of DFS, including scalability, fault tolerance, security, and access controls. It also provides an overview of the file service architecture, including the structural design and component interactions inside a DFS. It also covers alternative file access models and protocols, such as Server Message Block (SMB) and Andrew File System (AFS), which are critical for efficient and safe file sharing in remote contexts. This comprehensive reference is intended to provide a detailed understanding of DFS, its architectural foundation, and the protocols that enable its functionality.

7. 2. Objectives

1. Understanding the basic concepts and principles of distributed file systems.
2. Exploring different architectures and designs of distributed file systems.
3. Learning about the challenges and trade-offs in designing and implementing distributed file systems.
4. Studying the various techniques and algorithms used in distributed file systems for fault tolerance, scalability, and performance.
5. Analyzing real-world examples of distributed file systems and their use cases.
6. Discussing future trends and advancements in distributed file systems technology.

7. 3. Characteristics and their key features

A distributed file system uses multiple servers to store files that are accessible over networks. A distributed file system's primary features offer the benefits, and they can be summed up as follows by using a few example distributed file systems, like Hadoop, Andrew File System, Coda, GFS, Sun NFS, and many more. Characteristics and key features of distributed file systems include[1]:

1. **Transparency:** This security mechanism conceals details of one file system from others and from users. There are four types:
    a. **Structure Transparency:** Users are unaware of the DFS's actual structure, including the number of file servers and storage devices.
    b. **Access Transparency:** Users can access their file resources securely regardless of their location, following the correct login process.

[1] https://www.techtarget.com/searchstorage/tip/Key-features-of-a-distributed-file-system

   c. **Replication Transparency:** Replicated files in different nodes are hidden from other nodes within the system.
   d. **Naming Transparency:** File names do not indicate their location and remain consistent even when files move among storage nodes.
2. **Performance:** This measures the time required to process user file access requests, including processor time, network transmission time, and storage access time. The DFS performance should be comparable to a local file system.
3. **Scalability:** The DFS should support the addition of storage resources seamlessly, maintaining performance levels as storage capacity scales up.
4. **High Availability:** The DFS must remain operational despite issues like node failures or drive crashes. It should quickly reconfigure to alternative storage resources to maintain uninterrupted operations. Disaster Recovery (DR) plans must include provisions for backing up and recovering DFS servers and storage devices.
5. **Data Integrity:** The DFS must manage multiple access requests to the same file storage systems without causing disruptions or damage to file integrity.
6. **High Reliability:** To ensure data availability and survivability during disruptions, the DFS should create backup copies of files. This complements high availability and ensures that files and databases are accessible when needed.
7. **Security:** Data must be protected from unauthorized access and cyber attacks. Encrypting data at rest and in transit enhances security and protection.
8. **User Mobility:** This feature directs a user's file directory to the node where the user logs in, ensuring seamless access to their resources.
9. **Namespaces:** A namespace defines a repository of commands and variables for specific activities. In DFS, namespaces collect the necessary commands and actions for the system's functionality. A single namespace supporting multiple file systems creates a unified interface, making all systems appear as a single file system to the user. This reduces the likelihood of interference with the contents of other namespaces.

STOP TO CONSIDER

Transparency:

- **Structure Transparency:** Users are unaware of the DFS's actual structure.
- **Access Transparency:** Secure access to file resources regardless of location.
- **Replication Transparency:** Replicated files in different nodes are hidden from each other.
- **Naming Transparency:** File names remain consistent and do not indicate location.

Performance: Measures time for processing user file access requests, including processor, network, and storage access times. Comparable to local file system performance.

Scalability: Supports seamless addition of storage resources while maintaining performance levels as capacity scales up.

High Availability: Remains operational despite node failures or drive crashes, with quick reconfiguration to alternative resources. Disaster Recovery (DR) plans must cover backups and recovery of DFS servers and storage devices.

Data Integrity: Manages multiple access requests without causing disruptions or damage to file integrity.

High Reliability: Creates backup copies to ensure data availability and survivability during disruptions, complementing high availability.

Security: Protects data from unauthorized access and cyber attacks through encryption at rest and in transit.

User Mobility: Routes user's file directory to the node where they log in, ensuring seamless access to resources.

Namespaces: Collects commands and actions for DFS functionality, creating a unified interface that makes all systems appear as a single file system to the user, reducing interference with other namespaces.

Check Your Progress

Question 1. What is structure transparency in a distributed file system, and why is it important?

Question 2. How does access transparency benefit users in a distributed file system?

Question 3. Explain replication transparency and its significance in a distributed file system.

Question 4. What is naming transparency, and how does it maintain file name consistency?

Question 5. List the components that contribute to the performance of a distributed file system.

Question 6. Describe how scalability is achieved in a distributed file system.

Question 7. What mechanisms ensure high availability in a distributed file system during node failures or drive crashes?

Question 8. How does a distributed file system maintain data integrity when multiple users access the same files?

Question 9. Explain the role of high reliability and how backup copies contribute to it in a distributed file system.

Question 10. What security measures are implemented in a distributed file system to protect data from unauthorized access and cyber attacks?

Question 11. How does user mobility enhance the user experience in a distributed file system?

Question 12. Define namespaces in the context of a distributed file system and their role in providing a unified user interface.

Question 13. Why is it essential for a distributed file system to have performance comparable to a local file system?

Question 14. How do disaster recovery plans complement high availability in a distributed file system?

Question 15. What are the benefits of having a single namespace support multiple file systems in a distributed file system?

7. 4. Challenges and Requirements

The challenges of a distributed system is shown in Figure 1.

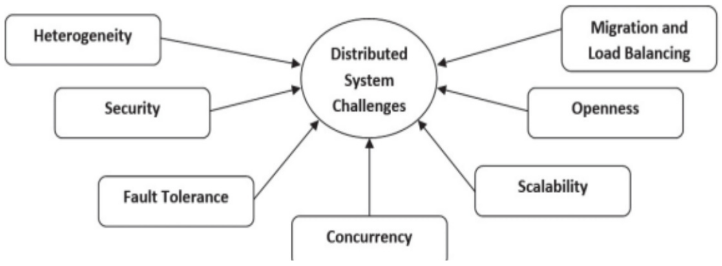


Figure 1 The challenges of a distributed system(1)

Heterogeneity: Distributed systems encompass a variety of services and applications that can operate across different computers and networks accessible over the Internet. This includes hardware devices of various types (PCs, tablets, etc.) running different operating systems (Windows, iOS, etc.), which need to communicate and exchange information. Programs written in different languages can interact only when these differences are addressed. Agreed-upon standards, similar to Internet protocols, are necessary. Middleware, a software layer that provides programming abstraction, helps mask the heterogeneity of networks, programming languages, operating systems, and hardware. An example of mobile code that can be relocated and executed on different computers is a Java applet.

STOP TO CONSIDER

- **Diverse Environments:** Operates across various hardware and operating systems.
- **Middleware:** Masks differences in networks, languages, and hardware. Example: Java applet.

Security: Security is a critical concern in distributed environments, especially when using public networks. Security encompasses confidentiality (protection against unauthorized access), integrity (protection against data tampering), and availability (protection against service disruption). Encryption techniques, such as those used in cryptography, can help address these concerns, but they are not foolproof. Distributed systems are vulnerable to threats like data leakage, integrity breaches, denial-of-service (DoS) attacks, and unauthorized access. DoS attacks, often conducted by botnets, overwhelm a server with fake requests.

STOP TO CONSIDER

- **Core Aspects:** Confidentiality, integrity, and availability.
- **Protection Measures:** Encryption enhances security, but vulnerabilities like DoS attacks remain.

Fault Tolerance and Handling: Distributed systems must be fault-tolerant, continuing to function normally even when some components fail. This involves detecting failures (e.g., using checksums), masking failures (e.g., retransmitting data upon failure), recovering from failures (e.g., rolling back to a safe state), and building redundancy (e.g., replicating data to prevent loss). Failures are inevitable, and the system must be equipped to handle them without significant disruption.

STOP TO CONSIDER

- **Failure Management:** Detect, mask, and recover from failures while maintaining functionality.
- **Redundancy:** Data replication to prevent loss and ensure continuous operation.

Concurrency: Concurrency issues arise when multiple clients request a shared resource simultaneously. The results may depend on the order of completion, necessitating synchronization. Distributed systems lack a global clock, making synchronization essential for the proper functioning of all components.

STOP TO CONSIDER

- **Synchronization:** Essential for managing simultaneous access to shared resources.

Scalability: Scalability challenges occur when a system cannot handle a sudden increase in resources or users. Efficient architecture and algorithms are crucial. Scalability has three primary dimensions:

- **Size:** The number of users and resources. Overloading can be a problem.
- **Geography:** The distance between users and resources. Communication reliability is a concern.
- **Administration:** Managing a growing number of nodes. Administrative chaos can occur.

STOP TO CONSIDER

- **Efficient Growth:** Manages increases in resources and users.
- **Key Dimensions:**
    - **Size:** Number of users and resources.
    - **Geography:** Distance between users and resources.
    - **Administration:** Managing a growing number of nodes.

Openness and Extensibility: Distributed systems should have well-defined interfaces that are openly available, facilitating easy addition of new components or features. Openness issues arise when previously published content is retracted. There is often no central authority in open distributed systems, with different systems having their own mediators. For example, platforms like Facebook and Twitter allow developers to create interactive software through their APIs.

STOP TO CONSIDER

- **Interfaces:** Well-defined and openly available for easy additions.
- **API Integration:** Enables interactive software development.

Migration and Load Balancing: Tasks and applications should operate independently to allow for seamless migration within the system without affecting others. To optimize performance, the system should distribute the load among available resources effectively.

STOP TO CONSIDER

- **Seamless Migration:** Tasks and applications move without disrupting others.
- **Optimal Performance:** Balances load across resources.

Check Your Progress

Question 1. What are the challenges associated with heterogeneity in distributed systems?
Question 2. How does middleware help manage heterogeneity in distributed systems?
Question 3. What are the three core aspects of security in distributed systems?
Question 4. How do encryption techniques enhance security in distributed systems?
Question 5. What methods are used to detect failures in a distributed system?
Question 6. How does redundancy improve fault tolerance in distributed systems?
Question 7. What is the importance of having recovery mechanisms in a distributed system?
Question 8. What role does synchronization play in managing concurrency?
Question 9. How do distributed systems handle multiple simultaneous access requests?
Question 10. What are the key dimensions of scalability in distributed systems?
Question 11. What are common challenges faced when scaling up a distributed system?
Question 12. How do well-defined interfaces benefit a distributed system?
Question 13. What are the challenges associated with openness in distributed systems?
Question 14. What is the significance of task migration in distributed systems?
Question 15. How do distributed systems ensure load balancing across resources?
Question 16. What are the benefits of effective load balancing in a distributed environment?
Question 17. How does middleware function as a bridge in distributed systems?
Question 18. How do namespaces contribute to the functioning of a distributed file system?

7. 5. Overview of File Service Architecture

In distributed system, File Service Architecture is an essential component that enables users to access and manipulate files remotely. It allows multiple users to access a shared file system over a distributed network. The design of file service architecture is based on the client server model. The client sends a request to server and server process request and send back requested data. The client Server model provides a scalable, fault-tolerance and reliable file service architecture. Examples of File Service Architecture are Network File System (NFS), Amazon S3, GlusterFS. The file service architecture of distributed file system is shown in Figure 2[2].

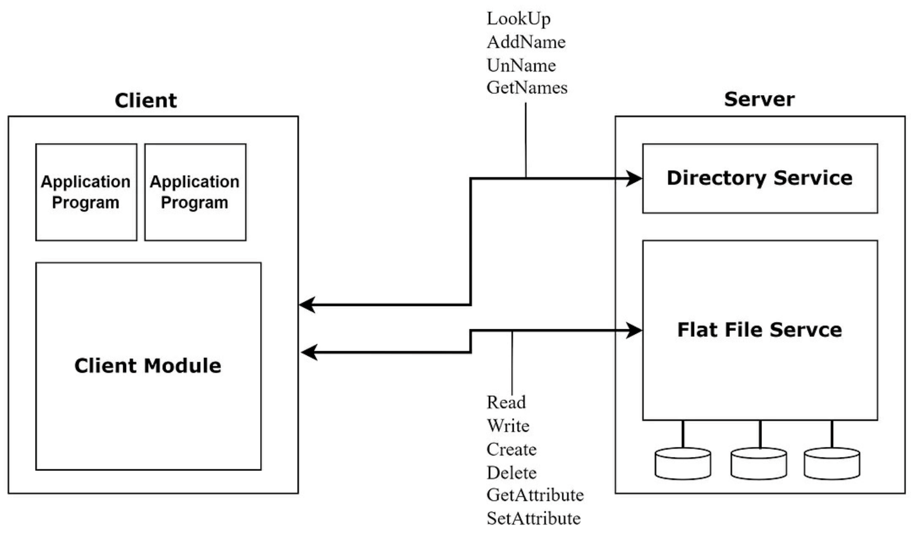


Figure 2 File Service Architecture of Distributed File System

The File Service Architecture consist of three primary component

1. Flat File Service: The flat file service is responsible for implementing the operations on the content of file. A Unique File Identification (UFIDs) are used to refer all requests for the flat file service operations. The falt file service operations includes Read, write, Create, Delete, GetAttribute and SetAttribute.

2. Directory Service: Directory Service provides mapping between file names and their UFIDs. Users and client

[2] https://medium.com/@ak_gaur/file-service-architecture-0b85d2051cba

modules can use the directory service to navigate the file system, search for files, and determine their location. Directory service operations include Lookup, AddName, UnNme and GetName.

3. Client Module: The client module is the software that runs on user devices and interacts with the file service architecture. It acts as an intermediary between users the file services. The client module initiates requests for file operations, accesses the directory service to locate files, and communicates with the flat file service to read or write data.

STOP TO CONSIDER

- **Flat File Service:**
    - Responsible for file content operations like Read, Write, Create, Delete, GetAttribute, and SetAttribute.
    - Uses Unique File Identifications (UFIDs) to manage file requests efficiently.
- **Directory Service:**
    - Provides mapping between file names and UFIDs.
    - Enables navigation, file searching, and location determination within the file system.
    - Supports operations such as Lookup, AddName, UnName, and GetName.
- **Client Module:**
    - Software running on user devices that interacts with the file service architecture.
    - Acts as an intermediary between users and file services.
    - Initiates file operations requests, accesses directory services for file location, and communicates with the flat file service for data operations.

Check Your Progress

Question 1. What are the key components of a distributed file system architecture?
Question 2. How does middleware help in managing heterogeneity in distributed systems?
Question 3. What are the core security concerns in distributed systems, and how can they be addressed?
Question 4. Explain the concept of fault tolerance in distributed systems. What strategies are commonly used?
Question 5. How does concurrency control work in distributed file systems?
Question 6. What are the scalability challenges in distributed systems, and how can they be mitigated?
Question 7. Discuss the importance of APIs in enabling interaction between different components of distributed systems.
Question 8. What role does synchronization play in ensuring data consistency across distributed systems?
Question 9. How do distributed systems handle load balancing and resource allocation?
Question 10. Explain the concept of namespaces in distributed file systems and their significance.

7. 6. Distributed File System Requirements

Distributed File System Requirements refer to the set of criteria that a distributed file system must meet in order to effectively store, manage, and access files across multiple nodes in a network.

7.6.1. Needs and goals

A distributed file system (DFS) must meet several fundamental needs and goals to effectively manage and provide access to files across distributed environments. The primary objective is to ensure efficient and reliable storage and retrieval of data. This involves enabling seamless access to files from multiple locations while prioritizing data integrity and security. The goal is to support diverse applications and user needs by offering a unified and scalable platform for storing and accessing data across distributed networks.

STOP TO CONSIDER

Needs and Goals

- **Efficient and Reliable Storage and Retrieval**: Ensuring data is stored and retrieved effectively.
- **Seamless Access**: Providing smooth access to files from multiple locations.
- **Data Integrity and Security**: Prioritizing the safety and correctness of data.
- **Support for Diverse Applications**: Catering to various user needs and applications.
- **Unified and Scalable Platform**: Offering a cohesive system that can grow with data demands.

7.6.2. Scalability

Scalability is a critical requirement for distributed file systems to handle increasing data volumes and user demands effectively. The system should be able to scale horizontally by adding nodes or storage resources dynamically as the workload grows. This scalability ensures that the DFS can accommodate expanding data storage requirements and maintain performance levels without disruptions, supporting the growth and evolution of organizational data needs over time.

STOP TO CONSIDER

Scalability:

- **Horizontal Scalability**: Adding nodes or storage resources as needed.
- **Dynamic Scaling**: Adjusting resources based on workload growth.
- **Maintaining Performance**: Ensuring consistent performance levels despite increasing data volumes.
- **Supporting Growth**: Accommodating expanding data storage requirements over time.

7.6.3. Fault tolerance

Fault tolerance is essential to maintain system availability and data integrity in the face of node failures or network issues. The distributed file system must implement robust redundancy mechanisms such as data replication and failover procedures. These mechanisms ensure that data remains accessible and consistent even if individual nodes fail, thereby minimizing downtime and ensuring continuous operation of critical services and applications.

STOP TO CONSIDER

Fault Tolerance

- **System Availability**: Keeping the system operational despite failures.
- **Data Integrity**: Preserving data accuracy and consistency.
- **Redundancy Mechanisms**: Implementing data replication and failover procedures.
- **Minimizing Downtime**: Ensuring continuous operation of services and applications.

7.6.4. Data Consistency

Ensuring data consistency across distributed nodes is another crucial requirement for a DFS. It involves managing concurrent access and updates to files across multiple users and locations while maintaining a consistent and accurate view of data. Distributed locking mechanisms and transaction protocols are employed to synchronize data access and updates, preventing conflicts and maintaining data integrity throughout the distributed file system.

STOP TO CONSIDER

Data Consistency

- **Concurrent Access Management**: Handling multiple users and locations accessing and updating files.
- **Consistent Data View**: Maintaining an accurate view of data across the system.
- **Synchronization**: Using distributed locking mechanisms and transaction protocols.
- **Conflict Prevention**: Preventing data access and update conflicts.

7.6.5. Security and Access controls

Security is paramount in distributed file systems to protect data from unauthorized access, tampering, and breaches. Robust authentication, encryption, and access control mechanisms are required to enforce data security policies effectively. These measures ensure that only authorized users and applications can access and manipulate data within the DFS, safeguarding sensitive information and maintaining regulatory compliance.

STOP TO CONSIDER

Security and Access Controls

- **Data Protection**: Safeguarding against unauthorized access and breaches.
- **Authentication**: Verifying user identities.
- **Encryption**: Protecting data during storage and transmission.
- **Access Control**: Enforcing policies to control data access and manipulation.
- **Regulatory Compliance**: Ensuring adherence to data protection regulations.

7. 7. File Service Architecture

File Service Architecture in a distributed file system refers to the way in which the file system is designed and structured to provide file services across a network of computers. In the previous section, it has been explained that this architecture typically involves a set of components that work together to manage and store files in a distributed manner.

7.7.1 Client-server vs. peer-to-peer models

Client-server and peer-to-peer are two common architectural models used in distributed systems, including file systems. Here is a comparison of the two models:

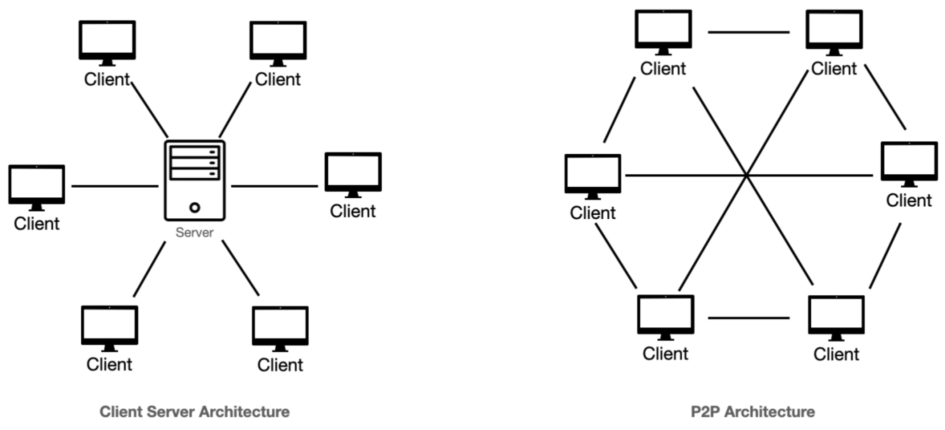


Figure 3 Client-server and peer-to-peer Model

(Source: https://systemdesignschool.io/blog/peer-to-peer-architecture)

Client-Server Model: In the client-server model, there are two types of entities: clients and servers.

- ❖ Clients request services or resources from servers, which respond to these requests.
- ❖ Servers are typically centralized and dedicated to providing specific services or resources to clients.
- ❖ Clients have limited responsibilities and rely on servers for most tasks.

Examples of client-server architectures include web servers serving web pages to web browsers, file servers providing file access to client machines, and database servers handling database queries from client applications.

Peer-to-Peer Model: In the peer-to-peer model, all participating nodes (peers) in the network have the ability to act as both clients and servers.

- ❖ Peers can directly communicate and share resources with each other without the need for a centralized server.
- ❖ Peers collaborate and contribute resources to the network, such as files, processing power, or bandwidth.
- ❖ The peer-to-peer model is decentralized, allowing for more scalability and fault tolerance compared to client-server architectures.

Examples of peer-to-peer architectures include file-sharing networks like BitTorrent, decentralized cryptocurrency networks like Bitcoin, and collaborative applications like Skype.

Table 1 Client Server vs. Peer-to-Peer model

Client-Server Model	**Peer-to-Peer Model**
Client-server model is centralized, with dedicated servers providing services to clients,	Peer-to-peer model is decentralized, with peers sharing resources directly with each other.
servers have specialized roles and responsibilities,	All peers have equal status and can act as both clients and servers.
easier to manage and control	offers more scalability and fault tolerance
commonly used for services that require centralized control and management	Suitable for distributed applications that benefit from decentralization and collaboration.

Both client-server and peer-to-peer models have their advantages and limitations, and the choice between them depends on the specific requirements of the distributed system being designed.

Check Your Progress

Question 1. What are the key characteristics of the client-server model in distributed systems?

Question 2. Describe how clients and servers interact in the client-server architecture. Provide examples of applications that use this model.

Question 3. What distinguishes the peer-to-peer model from the client-server model in distributed file systems?

Question 4. Compare and contrast the centralized nature of the client-server model with the decentralized nature of the peer-to-peer model.

Question 5. Explain why the peer-to-peer model offers more scalability and fault tolerance compared to the client-server model.

Question 6. Give examples of applications or networks that utilize the peer-to-peer architecture. How do these applications benefit from decentralization?

Question 7. Discuss the advantages and disadvantages of using a client-server architecture in distributed file systems.

Question 8. In what scenarios would you prefer to use a client-server model over a peer-to-peer model, and vice versa?

Question 9. How does each model handle resource sharing and management differently?

Question 10. What are the implications of choosing between a client-server and a peer-to-peer architecture for ensuring security and data integrity in distributed systems?

7.7.2 Role of metadata and data servers

Metadata and data servers play crucial roles in distributed file systems and storage systems.

Metadata Servers:

❖ Metadata servers store metadata information about files and directories in a distributed file system.
❖ Metadata includes attributes such as file names, sizes, timestamps, permissions, file locations, and directory structures.
❖ Metadata servers maintain the namespace of the file system and track the mapping between logical file names and physical file locations.
❖ Clients in the distributed system contact metadata servers to perform operations like file lookups, file creation, file deletion, and directory listing.
❖ Metadata servers help coordinate access to data stored across multiple data servers in the system.

STOP TO CONSIDER

Function: Store metadata information about files and directories in a distributed file system.
Metadata Includes: Attributes such as file names, sizes, timestamps, permissions, file locations, and directory structures.
Responsibilities: Maintain the namespace of the file system. Track the mapping between logical file names and physical file locations.
Client Interaction: Clients contact metadata servers to perform operations like file lookups, file creation, file deletion, and directory listing.
Coordination: Help coordinate access to data stored across multiple data servers in the system.

Check Your Progress

Question 1. What is the primary function of metadata servers in a distributed file system?

Question 2. List some key attributes that are typically stored in metadata by metadata servers. Why are these attributes important?

Question 3. Explain the role of metadata servers in maintaining the namespace of a distributed file system. Why is namespace management crucial?

Question 4. How do clients interact with metadata servers in a distributed file system? Provide examples of operations that clients perform via metadata servers.

Question 5. Discuss the coordination role of metadata servers in accessing data stored across multiple data servers. Why is coordination necessary in distributed file systems?

Data Servers:

- ❖ Data servers store the actual data contents of files in a distributed storage system.
- ❖ Data servers are responsible for storing and retrieving data blocks or chunks that make up files.
- ❖ Data servers handle read and write requests from clients, providing access to the data stored on disk or in memory.
- ❖ Data servers may replicate data blocks for fault tolerance and performance optimization.
- ❖ Data servers work in conjunction with metadata servers to ensure data consistency and availability in the distributed system.

Check Your Progress

Question 1. What is the primary role of data servers in a distributed storage system?

Question 2. Explain how data servers manage the storage and retrieval of file data in distributed file systems.

Question 3. What are data blocks or chunks, and how do data servers handle them in distributed storage systems?

Question 4. Discuss the responsibilities of data servers in handling read and write requests from clients. How do they ensure data availability and reliability?

Question 5. Why is data replication important for data servers in distributed file systems? How does replication contribute to fault tolerance and performance optimization?

Question 6. Describe the collaboration between data servers and metadata servers in ensuring data consistency across a distributed system.

Key Roles of Metadata and Data Servers:

1. Namespace Management: Metadata servers manage the namespace of the file system, while data servers store the actual data contents of files.
2. Metadata Access: Clients interact with metadata servers to access file metadata information, while data servers handle read and write operations for file data.
3. Coordination: Metadata servers coordinate access to data stored across multiple data servers in the distributed system.
4. Fault Tolerance: Data servers may replicate data blocks for fault tolerance, while metadata servers help maintain consistency and availability of metadata information.
5. Performance Optimization: Data servers optimize data access by storing and retrieving data efficiently, while metadata servers help optimize file system operations.

In summary, metadata servers handle metadata information about files and directories in a distributed file system, while data servers store and manage the actual data contents of files.

Together, they play essential roles in ensuring efficient and reliable access to data in distributed storage systems.

STOP TO CONSIDER

Namespace Management:

- **Metadata Servers:** Manage the namespace of the file system.
- **Data Servers:** Store the actual data contents of files.

Metadata Access:

- **Metadata Servers: Clients interact with these servers to access file metadata information.**
- **Data Servers:** Handle read and write operations for file data.

Coordination:

- **Metadata Servers:** Coordinate access to data stored across multiple data servers in the distributed system.

Fault Tolerance:

- **Data Servers:** May replicate data blocks for fault tolerance.
- **Metadata Servers:** Help maintain consistency and availability of metadata information.

Performance Optimization:

- **Data Servers:** Optimize data access by storing and retrieving data efficiently.
- **Metadata Servers:** Help optimize file system operations.

Check Your Progress

Question 1. What is the primary function of metadata servers in a distributed file system?

Question 2. How do clients interact with metadata servers, and what type of information do they typically request from them?

Question 3. Explain the role of data servers in fault tolerance within distributed file systems. How does data replication contribute to this?

Question 4. What is namespace management, and why is it important in the context of distributed file systems?

Question 5. Discuss the coordination role of metadata servers in accessing data across multiple data servers. Why is coordination necessary?

7.7.3 Communication protocols

A distributed system consists of interconnected computers, known as nodes, each performing specific tasks and collaborating to achieve a common objective. Such systems are designed to continue functioning even if some nodes fail, and these systems are referred to as fault-tolerant distributed systems. The primary types of faults in these systems are: crash (a node stops working), omission (a node fails to perform an expected action or to send/receive data), and Byzantine failure (a node exhibits malicious behavior). To achieve their goals, nodes in a distributed system communicate with one another, exchanging information through methods like messaging or shared memory, and coordinate their activities based on the shared information. This communication can be synchronous, asynchronous, or a hybrid of both[3].

[3] https://www.linkedin.com/pulse/communication-protocols-distributed-systems-arthur-sergeyan/

STOP TO CONSIDER

Types of Faults:

- **Crash:** A node stops working.
- **Omission:** A node fails to perform an expected action or to send/receive data.
- **Byzantine Failure:** A node exhibits malicious behavior.

1. **Synchronous Communication**: In this type of communication, nodes interact in a tightly coordinated manner. They wait for each other to send and receive messages before proceeding. This ensures that all nodes are always on the same page, but it can be slower due to the waiting times.

STOP TO CONSIDER

- Nodes interact in a tightly coordinated manner.
- Nodes wait for each other to send and receive messages before proceeding.
- Ensures all nodes are synchronized, but can be slower due to waiting times.

2. **Asynchronous Communication**: Nodes operate more independently, sending and receiving messages without waiting for responses. This can lead to faster overall performance, but it requires more complex mechanisms to ensure that all nodes stay synchronized and that the system remains consistent.

STOP TO CONSIDER

- Nodes operate independently, sending and receiving messages without waiting for responses.
- Can lead to faster overall performance.
- Requires complex mechanisms to ensure synchronization and system consistency.

3. **Hybrid Communication**: Many distributed systems use a mix of synchronous and asynchronous communication to balance the need for coordination with the need for speed and efficiency.

Check Your Progress

Question 1. What are the primary types of faults that can occur in fault-tolerant distributed systems? Explain each type briefly.

Question 2. Describe synchronous communication in distributed systems. What are the advantages and disadvantages of this communication model?

Question 3. Explain asynchronous communication in distributed systems. What are the benefits and challenges associated with this approach?

Question 4. Why do many distributed systems opt for a hybrid communication approach? Provide examples of scenarios where hybrid communication might be advantageous.

Question 5. How do nodes in a distributed system coordinate their activities through communication protocols?

7.7.4 Data Transfer Mechanisms

Data transfer mechanisms in distributed file systems (DFS) are crucial for ensuring efficient, reliable, and scalable access to files across multiple nodes. These systems distribute data storage and access across a network of computers, enabling high availability and performance. Here is some key data transfer mechanisms used in distributed file systems,

1. **Chunk-Based Data Storage and Transfer:** Distributed file systems often break files into fixed-size chunks or blocks. These chunks are stored on different nodes, and data transfer involves reading/writing these chunks. Example: Google File System (GFS)**(2)**.

STOP TO CONSIDER

- Files are broken into fixed-size chunks or blocks.
- These chunks are stored on different nodes.
- Data transfer involves reading/writing these chunks.

2. **Replication**: To ensure reliability and fault tolerance, chunks are often replicated across multiple nodes. The replication mechanism ensures data availability even if some nodes fail. Example: Hadoop Distributed File System (HDFS) (3)

STOP TO CONSIDER

- Chunks are replicated across multiple nodes to ensure reliability and fault tolerance, maintaining data availability even if some nodes fail.

3. Data Placement and Balancing: DFS uses algorithms to decide where to place chunks to optimize for load balancing and network traffic minimization. Example: Ceph (4).

STOP TO CONSIDER

- DFS uses algorithms to decide where to place chunks to optimize for load balancing and network traffic minimization.

4. Metadata Management: Efficient metadata management is crucial for tracking the locations of chunks and ensuring quick access to files. Metadata servers handle this information and respond to client queries. Example: Lustre File System, (5)

STOP TO CONSIDER

- Efficient metadata management is crucial for tracking the locations of chunks and ensuring quick access to files.
- Metadata servers handle this information and respond to client queries.

5. Client Caching: Clients often cache frequently accessed data locally to reduce network traffic and improve access speed.

Caching mechanisms ensure consistency and coherency of data. Example: AFS (Andrew File System). (6)

STOP TO CONSIDER

- Clients often cache frequently accessed data locally to reduce network traffic and improve access speed.
- Caching mechanisms ensure consistency and coherency of data.

6. Data Striping: Data striping distributes chunks of data across multiple disks or nodes to improve throughput by parallelizing read/write operations. Example: IBM GPFS (General Parallel File System) (7)

STOP TO CONSIDER

- Data striping distributes chunks of data across multiple disks or nodes to improve throughput by parallelizing read/write operations.

7. Consistency Protocols: Ensuring data consistency across replicas is a key challenge. Distributed file systems implement various consistency protocols to handle concurrent access and updates. Example: Coda File System. (8)

STOP TO CONSIDER

- Ensuring data consistency across replicas is a key challenge.
- Distributed file systems implement various consistency protocols to handle concurrent access and updates.

8. Erasure Coding: Erasure coding is used to provide fault tolerance with lower storage overhead compared to replication. It divides data into fragments, expands it with redundant data pieces, and stores these pieces across different locations. Example: Microsoft Azure Storage. (9)

STOP TO CONSIDER

- Erasure coding is used to provide fault tolerance with lower storage overhead compared to replication.
- It divides data into fragments, expands it with redundant data pieces, and stores these pieces across different locations.

Check Your Progress

Question 1. What are the primary types of faults that can occur in fault-tolerant distributed systems? Explain each type briefly.

Question 2. Describe synchronous communication in distributed systems. What are the advantages and disadvantages of this communication model?

Question 3. Explain asynchronous communication in distributed systems. What are the benefits and challenges associated with this approach?

Question 4. Why do many distributed systems opt for a hybrid communication approach? Provide examples of scenarios where hybrid communication might be advantageous.

Question 5. How do nodes in a distributed system coordinate their activities through communication protocols?

7. 8. File Accessing Models:

File Accessing Models refer to the different methods or approaches used to control and manage access to files in a file system. These models define the rules and permissions governing how users or processes can interact with files, read or write data, and perform operations on files stored in the file system. The file accessing model essentially depends on:

- The unit of data access/transfer.
- The method utilized for accessing remote files.

Based on the unit of data access, the following file access models may be used to access specific files:

1. File-Level Transfer Model: In the file-level transfer model, the entire file is transferred whenever a specific action requires the file data. The entire document is sent across the distributed computing network between the client and server. This model has better scalability and is efficient.
2. Block-Level Transfer Model: In the block-level transfer model, file data is transferred between the client and server in units of file blocks. The unit of data transfer in this model is file blocks. This model may be utilized in a distributed computing environment with several diskless workstations.
3. Byte-Level Transfer Model: In the byte-level transfer model, file data is transferred between the client and server in units of bytes. The unit of data transfer in this model is bytes. The byte-level transfer model offers greater flexibility compared to other file transfer models because it allows the recovery and management of an inconsistent subset of a file. The major disadvantage of this model is the complexity in cache management due to variable-length data for different access requests.
4. Record-Level Transfer Model: The record-level transfer model is used in scenarios where file contents are organized as records. In this model, file data is transferred between the client and server in units of records. The unit of data transfer in the record-level transfer model is records.

STOP TO CONSIDER

File-Level Transfer Model: In this model, the entire file is transferred whenever a specific action requires the file data. The entire document is sent across the distributed computing network between the client and server. This model has better scalability and is efficient.

Block-Level Transfer Model: Here, file data is transferred between the client and server in units of file blocks. This model may be utilized in a distributed computing environment with several diskless workstations.

Byte-Level Transfer Model: In this model, file data is transferred between the client and server in units of bytes. The byte-level transfer model offers greater flexibility because it allows the recovery and management of an inconsistent subset of a file. However, it has the disadvantage of complexity in cache management due to variable-length data for different access requests.

Record-Level Transfer Model: This model is used in scenarios where file contents are organized as records. File data is transferred between the client and server in units of records.

Check Your Progress

Question 1. Compare and contrast the file-level transfer model with the block-level transfer model. What are the advantages and disadvantages of each?

Question 2. Explain the flexibility offered by the byte-level transfer model compared to other file transfer models. What challenges does it pose for cache management?

Question 3. In which scenarios would the record-level transfer model be most beneficial? Discuss its advantages in specific use cases.

Question 4. How does the choice of data access unit (file, block, byte, or record) impact the performance and efficiency of file access in distributed computing environments?

7.8.1. Remote access

In the remote service model, handling a client's request is performed at the server's hub. The client's request for file access is passed across the network as a message to the server. The server machine performs the access request, and the result is sent back to the client.

Advantages:

1. Simplifies consistency management by keeping a single authoritative copy at the server.
2. Useful when the client's main memory is limited.

Disadvantages:

1. Increases server load and network traffic, potentially compromising performance.
2. Remote access handling across the network is inherently slower.
3. Transmitting a series of responses to specific requests results in higher network overhead.

This model is essentially an extension of the local file system interface across the network, ensuring that the server maintains a consistent copy of the data.

STOP TO CONSIDER

- In the remote service model, handling a client's request is performed at the server's hub.
- The client's request for file access is passed across the network as a message to the server.
- The server machine performs the access request, and the result is sent back to the client.
- **Advantages:** Simplifies consistency management by keeping a single authoritative copy at the server, and is useful when the client's main memory is limited.
- **Disadvantages:** Increases server load and network traffic, potentially compromising performance; remote access handling across the network is inherently slower; transmitting a series of responses to specific requests results in higher network overhead.
- This model is essentially an extension of the local file system interface across the network, ensuring that the server maintains a consistent copy of the data.

Check Your Progress

Question 1. How does the remote service model simplify consistency management in distributed file systems?
Question 2. What are the primary disadvantages of handling remote access across a network?
Question 3. Explain the concept of network overhead in the context of remote service models.
Question 4. What advantages does the remote service model offer when the client's main memory is limited?
Question 5. Compare the performance implications of remote access handling versus local file system operations.

7.8.2. Caching and Sharing

The data-caching model reduces network traffic by caching data obtained from the server. This exploits the locality aspect observed in file accesses. A replacement policy, such as Least Recently Used (LRU), is employed to keep the cache size limited.

Advantages:

1. Remote access can be served locally, making access faster.
2. Reduces network traffic and server load, improving scalability.
3. Network overhead is less significant when transmitting large amounts of data compared to the remote service model.

- **Disadvantages**:

1. Maintaining consistency can be challenging. Performance is better with fewer writes and worse with more frequent writes.
2. Caching is more effective for machines with disks or large main memory.
3. The lower-level machine interface is different from the upper-level user interface.

STOP TO CONSIDER

- The data-caching model reduces network traffic by caching data obtained from the server, exploiting the locality aspect observed in file accesses.
- A replacement policy, such as Least Recently Used (LRU), is employed to keep the cache size limited.
- **Advantages:** Remote access can be served locally, making access faster; reduces network traffic and server load, improving scalability; and network overhead is less significant when transmitting large amounts of data compared to the remote service model.
- **Disadvantages:** Maintaining consistency can be challenging, with performance being better with fewer writes and worse with more frequent writes; caching is more effective for machines with disks or large main memory; and the lower-level machine interface is different from the upper-level user interface.

Check Your Progress

Question 1. How does the data-caching model reduce network traffic in distributed systems?

Question 2. What role does the Least Recently Used (LRU) policy play in caching?

Question 3. List two advantages of using data caching in distributed systems.

Question 4. What are two challenges associated with maintaining consistency in a caching system?

Question 5. Explain why caching may be more effective for machines with disks or large main memory.

7.8.3. Benefit of Data-Caching Model over the Remote Service Model

The data-caching model offers the potential for improved performance and greater system scalability. It reduces network traffic, contention for the network, and contention for the file servers. Consequently, almost all distributed file systems implement some form of caching.

Example: NFS primarily uses the remote service model but adds caching for better performance.

STOP TO CONSIDER

- The data-caching model offers the potential for improved performance and greater system scalability.
- It reduces network traffic, contention for the network, and contention for the file servers.
- Consequently, almost all distributed file systems implement some form of caching.
- **Example:** NFS primarily uses the remote service model but adds caching for better performance.

Check Your Progress

Question 1. What potential advantages does the data-caching model offer over the remote service model?

Question 2. How does caching reduce network traffic in distributed file systems?

Question 3. Name one specific benefit of caching mentioned in the text.

Question 4. Why do almost all distributed file systems implement some form of caching?

Question 5. Which distributed file system primarily uses the remote service model but incorporates caching for performance enhancement?

7. 9. File Access Protocols:

File access protocols are essential communication protocols that define how data is accessed, transferred, and managed between clients and servers in a networked file system environment. These protocols enable users to access and manipulate files stored on remote servers as if they were stored locally. Here are five common file access protocols used in networking:

7.9.1. Network File System (NFS)

NFS, or Network File System, is a protocol for a distributed file system developed by Sun Microsystems in 1984. It operates on a client/server architecture, consisting of a client program, a server program, and a protocol that facilitates communication between the client and server.

NFS allows users to access data and files remotely over a network, making it possible for users to manipulate files as if they were stored locally. It is an open standard, enabling easy implementation by any user. The protocol is built on the ONC RPC system.

NFS is particularly useful in computing environments where centralized management of resources and data is essential. It uses both the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) to access and deliver data and files.

NFS operates on all IP-based networks and is implemented in client/server applications where the NFS server handles authorization, authentication, and client management. This protocol is commonly used with operating systems such as Apple Mac OS, Unix, and Unix-like systems including Solaris, Linux, FreeBSD, and AIX.

STOP TO CONSIDER

- NFS, developed by Sun Microsystems in 1984, is a protocol for distributed file systems operating on a client/server architecture.
- It allows remote access to data and files over a network, facilitating manipulation as if stored locally.
- NFS is an open standard built on the ONC RPC system, supporting both TCP and UDP for data access.
- It's widely used in environments requiring centralized resource management and is compatible with various operating systems like Apple Mac OS, Unix, Solaris, Linux, FreeBSD, and AIX.

Check Your Progress

Question 1. When was NFS developed, and by whom?
Question 2. What is the basic architecture of NFS?
Question 3. Name two protocols used by NFS for data access and delivery.
Question 4. What role does the NFS server play in the client/server setup?
Question 5. Which computing environments find NFS particularly useful, and why?
Question 6. List two operating systems compatible with NFS.

7.9.2. Common Internet File System (CIFS)

The Common Internet File System (CIFS) is a network file-sharing protocol that provides shared access to files, printers, serial ports, and various communications between nodes on a network. Originally developed by IBM and later enhanced by Microsoft, CIFS is an extension of the Server Message Block (SMB) protocol, which allows programs to request files and services from servers over a network. CIFS operates over TCP/IP, making it suitable for use both over the internet and within local area networks (LANs).

CIFS operates on a client-server model where clients request file access and other services from servers. It supports essential features such as file sharing, printer sharing, and named pipes for inter-process communication. The protocol includes mechanisms for secure access, including user authentication and permission management, ensuring data integrity and coherence when multiple users access and update the same file.

One of the primary advantages of CIFS is its interoperability across different operating systems, including Windows, Linux, and Unix. This cross-platform compatibility simplifies resource sharing within networks. Additionally, CIFS is well-documented and standardized, supporting a wide range of implementations. However, CIFS can be slower than some other file-sharing protocols, especially over wide area networks (WANs), and earlier versions had security vulnerabilities. Managing permissions and shares can also be complex in larger environments.

CIFS is widely used in enterprise networks for sharing files and printers and facilitates cross-platform file sharing between different operating systems. While CIFS is often used interchangeably with SMB, it specifically refers to the version of SMB used in Windows NT 4.0 and later. Modern implementations of SMB, such as SMB2 and SMB3, offer enhanced performance and security features, improving upon CIFS.

An example implementation of CIFS is mapping a network drive in Windows, allowing users to access shared folders and files on a remote server as if they were on their local machine. Another example is Samba, an open-source implementation of the SMB/CIFS protocol, enabling Unix and Linux systems to interact with Windows clients and servers. In summary, CIFS is a robust and versatile protocol widely used for network file and resource sharing across different operating systems and network configurations, with continued evolution and integration into modern SMB versions ensuring its relevance in today's networked environments.

STOP TO CONSIDER

- CIFS is a network file-sharing protocol developed by IBM and enhanced by Microsoft, extending the Server Message Block (SMB) protocol.
- It enables shared access to files, printers, and other resources across nodes on a network, operating over TCP/IP for use in LANs and over the internet.
- CIFS operates on a client-server model, supporting features like file sharing, printer sharing, and secure access with authentication and permission management.
- It offers interoperability across Windows, Linux, and Unix systems, simplifying resource sharing.
- Despite earlier security vulnerabilities and potential performance issues over WANs, CIFS remains widely used in enterprise networks for its robust file-sharing capabilities and cross-platform support.

Table 2. Difference between NFS and CIFS

NFS	CIFS
1. NFS is an abbreviation of the Network File System.	1. CIFS is an abbreviation of the Common Internet File system.
2. This protocol is used for sharing the files by Unix and Linux Operating systems.	2. This protocol is used for sharing the files by Windows Operating systems.
3. It is highly scalable.	3. It is low scalable.
4. The speed of communication is fast.	4. The speed of communication is medium.
5. The network File system is not a secure protocol.	5. Common Internet File System is more secure than the Network File System.
6. NFS is not a reliable protocol.	6. CIFS is a reliable protocol.
7. This protocol does not provide the session.	7. This protocol provides the sessions.
8. This protocol is easy to	8. Its implementation is

implement and set up.	complex.
9. This protocol uses 111 port for both TCP and UDP.	9. This protocol uses 139 and 445 TCP ports and 137 and 138 UDP ports.

STOP TO CONSIDER

- Name and Purpose: NFS: Network File System, used by Unix and Linux whereas CIFS: Common Internet File System, used by Windows.
- Scalability: NFS: Highly scalable whereas CIFS: Low scalability.
- Speed of Communication: NFS: Fast communication speed. whereas CIFS: Medium communication speed.
- Security: NFS: Not a secure protocol. whereas CIFS: More secure than NFS.
- Reliability: NFS: Not considered highly reliable. whereas CIFS: Considered a reliable protocol.
- Session Support: NFS: Does not provide sessions. whereas CIFS: Provides sessions.
- Ease of Implementation: NFS: Relatively easy to implement and set up. whereas CIFS: Implementation can be complex.
- Port Usage: NFS: Uses port 111 for both TCP and UDP. whereas CIFS: Uses TCP ports 139 and 445, and UDP ports 137 and 138.

Check Your Progress

Question 1. What does CIFS stand for, and what protocol is it an extension of?

Question 2. Describe the client-server model used by CIFS.

Question 3. What are two primary advantages of CIFS?

Question 4. How does CIFS ensure data integrity and coherence when multiple users access the same file?

Question 5. Name one disadvantage of CIFS mentioned in the text.

Question 6. Differentiate between NFS and CIFS based on scalability.

Question 7. Which modern implementations of SMB have enhanced upon CIFS, and how?

7.9.3. Server Message Block protocol (SMB protocol)

The Server Message Block (SMB) protocol is a client-server communication protocol used for sharing access to files, printers, serial ports, and other resources on a network. It also supports transaction protocols for interprocess communication. Initially developed by IBM in the 1980s, SMB has become a widely implemented solution, particularly in Windows environments, but it is also supported by Linux and macOS.

SMB allows applications and users to access files on remote servers and connect to other resources like printers and named pipes. It provides secure and controlled methods for opening, reading, moving, creating, and updating files on remote servers. As a response-request protocol, SMB enables network communications where the client sends a request and the server responds, establishing a two-way communication channel.

Originally, SMB ran on top of Network Basic Input/Output System over TCP/IP (NetBIOS over TCP/IP, or NBT) and used ports 137, 138, and 139. Nowadays, SMB runs directly over TCP/IP using port 445. Systems that do not support SMB directly over TCP/IP require NetBIOS over a transport protocol like TCP/IP.

List of SMB Protocol Dialects:

Over the years, SMB has evolved through several dialects, each introducing improvements in capabilities, scalability, security, and efficiency:

- **SMB 1.0 (1984)**: Introduced by IBM for DOS, featuring opportunistic locking to reduce network traffic.
- **CIFS (1996)**: Microsoft's SMB dialect introduced in Windows 95, supporting larger file sizes and direct transport over TCP/IP.
- **SMB 2.0 (2006)**: Released with Windows Vista and Windows Server 2008, enhancing performance and scalability.
- **SMB 2.1 (2010)**: Introduced with Windows Server 2008 R2 and Windows 7, featuring improved caching and energy efficiency.
- **SMB 3.0 (2012)**: Came with Windows 8 and Windows Server 2012, adding significant upgrades like SMB Multichannel, SMB Direct, and SMB Encryption.
- **SMB 3.02 (2014)**: Included performance updates and the option to disable CIFS/SMB 1.0.
- **SMB 3.1.1 (2015)**: Released with Windows 10 and Windows Server 2016, featuring advanced encryption and protection against man-in-the-middle attacks.

Despite its widespread use, SMB has faced security challenges. For instance, the WannaCry and Petya ransomware attacks in 2017 exploited a vulnerability in SMB 1.0. Microsoft released a patch, and experts recommended disabling SMB 1.0/CIFS. Newer versions like SMB 3.0 and 3.1.1 offer enhanced security features such as end-to-end encryption and pre-authentication integrity.

STOP TO CONSIDER

- The Server Message Block (SMB) protocol is a client-server communication protocol used for sharing access to files, printers, and other resources on a network.
- Initially developed by IBM and widely implemented in Windows environments, SMB also has support in Linux and macOS.
- It facilitates secure and controlled access to remote resources, allowing applications to perform operations like opening, reading, and updating files over a network.
- SMB operates over TCP/IP, primarily using port 445, and has evolved through various dialects (SMB 1.0 to SMB 3.1.1), each introducing improvements in capabilities, security, and efficiency.

Check Your Progress

Question 1. What is the Server Message Block (SMB) protocol used for?

Question 2. Which company initially developed SMB, and in what decade?

Question 3. What is the current port used by SMB over TCP/IP?

Question 4. Name two improvements introduced in SMB 3.0 over previous versions.

Question 5. What were the security vulnerabilities associated with SMB 1.0 that led to significant cyberattacks?

Question 6. How does SMB facilitate network communications between clients and servers?

Question 7. What is the significance of SMB 2.0 in terms of enhancements?

7.9.4. Andrew File System (AFS)

The Andrew File System (AFS), created by Morris et al. in 1986 at Carnegie Mellon University (CMU), is a distributed computing environment designed to facilitate campus-wide computer and information system usage. AFS allows client workstations in various locations to access server files easily, providing a consistent and location-independent file namespace through a network of reliable servers.

Key Features and Functionality

- **Location-Independent Namespace**: AFS offers a homogeneous file namespace that is transparent to the location, accessible to all client workstations.
- **Distributed Computing Infrastructure (DCI)**: Users log into workstations within the DCI to share data and applications.
- **Client-Server Communication**: AFS reduces the frequency of client-server communications by transferring entire files between servers and clients and caching them locally until updates are available.
- **Local Caching**: Servers respond to requests by storing data in the client's local cache, improving speed and efficiency in distributed networks.

Architecture of AFS

Vice

- **Role**: Vice refers to the group of trustworthy servers that provide the homogeneous, location-transparent file namespace.
- **Implementation**: Uses the Berkeley Software Distribution (BSD) of Unix on both clients and servers.
- **Operation**: Each workstation's operating system intercepts file system calls and redirects them to a user-level process known as Venus.

Venus

- **Role**: Venus is the mechanism that caches files from Vice and updates the server with new versions of those files.
- **Operation**: Venus communicates with Vice only when files are opened or closed. It reads and writes individual bytes directly on the cached copy, bypassing Vice for most operations.
- **Scalability**: Venus performs most of the work to minimize the load on Vice, which focuses on maintaining the file system's integrity, availability, and security.

Components of AFS Networks

- **Clients**: Any computer that requests files from AFS servers on the network.
- **Local Cache**: Once a server responds to a file request, the file is saved in the client's local cache and displayed to the user.
- **Callback Mechanism**: The client sends modifications to the server via callbacks, and frequently accessed files are stored in the local cache for rapid access.

Implementation of AFS

- **Client and Server Communication**: Client processes interact with the UNIX kernel through standard system calls. The kernel is modified to identify and route requests to Vice files to the workstation's Venus client process.
- **File Retrieval and Caching**: Venus contacts the server for missing volumes in the cache, retrieves the file or directory, and caches a copy on the local disk.
- **Security and Authentication**: Establishing a secure connection is essential for accessing files. Venus returns the cached file to the kernel, which then opens it for the client process.
- **Local Directory for Cache**: The client cache is a local directory on the workstation's disk, containing placeholder files for cache entries.

Advantages of AFS

- **Longevity of Shared Files**: Files that are not frequently updated can be cached for a long time.

- **Ample Caching Storage**: AFS allocates significant storage for caching files.
- **Efficient Working Set**: Ensures that a user's frequently accessed files remain in the cache for quick access.

In summary, the Andrew File System is a scalable and efficient distributed file system that uses local caching and a transparent file namespace to streamline file access and reduce network communication overhead.

STOP TO CONSIDER

- **Creation and Purpose:** Created in 1986 at Carnegie Mellon University (CMU) to support campus-wide computing and information system usage.
- **Location-Independent Namespace:** AFS provides a homogeneous file namespace accessible to client workstations regardless of location.
- **Distributed Computing Infrastructure (DCI):** Users within the DCI log into workstations to share data and applications.
- **Client-Server Communication:** AFS transfers entire files between servers and clients, minimizing client-server communications and caching files locally until updates are available.
- **Architecture Components:**
  - **Vice:** Group of trustworthy servers providing a location-transparent file namespace.
  - **Venus:** Client-side mechanism caching files from Vice and updating server copies.

- **Components of AFS Networks:**
  - **Clients:** Computers requesting files from AFS servers.
  - **Local Cache:** Stores requested files locally for efficient access.
  - **Callback Mechanism:** Ensures server updates for modified files via callbacks.

- **Implementation Details:**
  - **Client and Server Communication:** Uses UNIX system calls, routing requests to Venus client processes.
  - **File Retrieval and Caching:** Venus retrieves files from servers and caches them locally.
  - **Security and Authentication:** Establishes secure connections for file access.
  - **Local Cache Directory:** Stores cached files on the workstation's disk.

- **Advantages:**
  - **Longevity of Shared Files:** Caches files not frequently updated for extended periods.
  - **Ample Caching Storage:** Allocates significant storage for file caching.
  - **Efficient Working Set:** Ensures frequently accessed files remain cached for quick access.

7. 10. Summing Up

Distributed File Systems (DFS) have emerged as a critical infrastructure for modern computing environments, enabling seamless data storage and access across numerous servers and locations. These systems provide the foundation for programs to handle isolated data as if it were stored locally while facilitating secure and efficient file sharing over networks. The primary objectives of DFS are to enhance data availability, improve performance, and ensure data redundancy. These goals are achieved through the system's key characteristics, which include transparency, high performance, scalability, high availability, data integrity, reliability, security, user mobility, and unified namespaces. Transparency in DFS, divided into structure, access, replication, and naming transparency, hides the complexities of the underlying system architecture from users, allowing for a more user-friendly experience. High performance in DFS ensures that the time taken to process user file access requests remains comparable to that of local file systems. Scalability is another crucial feature, enabling the system to seamlessly integrate additional storage resources while maintaining optimal performance levels. High availability ensures that the system remains operational despite potential hardware or software failures, employing disaster recovery plans to back up and recover servers and storage devices. Data integrity is maintained through mechanisms that manage multiple access requests without disrupting or damaging file contents. Reliability is achieved by creating backup copies of files, ensuring data availability even during disruptions.

Security in DFS is paramount, involving robust measures to protect data from unauthorized access and cyber threats. Encryption techniques, both for data at rest and in transit, enhance this security.

User mobility is another essential feature, allowing users to access their file directories from any node within the network seamlessly. Unified namespaces provide a single interface for multiple file systems, making the entire system appear as a single file system to the user and reducing the risk of interference. However, the implementation of DFS is not without challenges. Heterogeneity, which involves the variety of services and applications operating across different hardware and networks, requires standardized protocols and middleware to mask differences and facilitate communication. Security concerns are significant, especially when using public networks, necessitating encryption and robust access controls to protect against data breaches, tampering, and denial-of-service attacks. Fault tolerance is essential to maintain system functionality in the face of component failures, employing techniques such as data replication and failover procedures. Concurrency issues arise from multiple clients accessing shared resources simultaneously, requiring synchronization to ensure proper operation. Scalability challenges involve handling increasing resources and users efficiently, with considerations for size, geography, and administration. Openness and extensibility are also critical, allowing easy addition of new components and features without compromising the system's integrity.

The file service architecture of DFS, typically based on the client-server model, includes components like the flat file service, directory service, and client module. The client-server model provides a scalable, fault-tolerant, and reliable architecture, with servers processing client requests and managing data. Alternatively, the peer-to-peer model offers a decentralized approach, allowing peers to share resources directly, providing greater scalability and fault tolerance. The architecture must support seamless data access, robust security measures, and efficient data management across distributed networks. Real-world examples of DFS include Network File System (NFS), Amazon S3, and GlusterFS. As organizations continue to generate and rely on vast amounts of data, the importance of DFS will only grow, necessitating advancements in scalability, fault tolerance, data consistency, and security to meet evolving data management needs. Understanding the principles and challenges of DFS is essential for developing robust, efficient, and secure distributed systems that can handle the demands of modern data-driven environments.

The data-caching model enhances performance and scalability in distributed file systems by locally caching data obtained from servers. This reduces network traffic, server load, and access times. Caching exploits data locality, employing policies like Least Recently Used (LRU) to manage cache size. While it improves read performance, maintaining consistency can be challenging, especially with frequent writes. Caching is most effective with machines having disks or ample memory. In contrast to the remote service model, which handles requests directly at the server, data-caching minimizes network overhead and server contention, offering a more efficient and scalable solution. Example: NFS with added caching.

File access protocols like NFS, CIFS, and SMB facilitate seamless file sharing across networks. NFS, developed by Sun Microsystems, offers open standard access primarily used in Unix and Linux environments. CIFS, an extension of SMB, supports file sharing, printer access, and inter-process communication across TCP/IP networks, widely used in Windows environments. SMB, evolving through versions like SMB 3.0, enhances performance with features like multichannel support and encryption, supporting Windows, Linux, and macOS. These protocols ensure secure, efficient, and scalable file access, crucial for collaborative and distributed computing environments.

7. 11. Model Questions

Question 1. What is the primary function of a distributed file system (DFS)?

Question 2. Explain the difference between a centralized and distributed file system.

Question 3. Describe the components typically found in a distributed file system architecture.

Question 4. How do clients interact with servers in a distributed file system?

Question 5. What role does metadata play in a distributed file system?

Question 6. Draw a diagram illustrating the architecture of a distributed file system and label its components.

Question 7. How does data access differ between a distributed file system and a traditional file system?

Question 8. List three main objectives of a distributed file system.

Question 9. Explain how a distributed file system addresses the scalability challenge.

Question 10. Discuss the importance of fault tolerance in distributed file systems.

Question 11. Why is data consistency critical in distributed file systems?

Question 12. How does a distributed file system enhance data availability compared to traditional file systems?

Question 13. What are the security objectives that a distributed file system aims to achieve?

Question 14. Describe the goals of caching mechanisms in distributed file systems.

Question 15. Define transparency in the context of distributed file systems.

Question 16. Discuss the role of concurrency control in distributed file systems.

Question 17. What scalability challenges might distributed file systems face as they grow?

Question 18. How does fault tolerance contribute to the reliability of a distributed file system?

Question 19. Explain the concept of metadata management in distributed file systems.

Question 20. Describe the role of caching in improving performance in distributed file systems.

Question 21. Compare the performance of read and write operations in distributed file systems.

Question 22. What are the main challenges in ensuring data consistency across distributed file systems?

Question 23. How does network latency affect the performance of distributed file systems?

Question 24. Discuss the security challenges specific to distributed file systems.

Question 25. Explain the impact of metadata scalability on the overall performance of a distributed file system.

Question 26. Describe the challenges associated with maintaining fault tolerance in distributed file systems.

Question 27. What are the requirements for effective load balancing in distributed file systems?

Question 28. How does data fragmentation impact the efficiency of distributed file systems?

Question 29. Compare and contrast file-level and block-level access models in distributed file systems.

Question 30. Explain the benefits of using a record-level transfer model for specific types of data.

Question 31. Describe the role of access control lists (ACLs) in managing file access permissions.

Question 32. Discuss the advantages of using erasure coding over traditional replication in distributed file systems.

Question 33. What is the significance of data striping in improving throughput in distributed file systems?

Question 34. How do consistency protocols ensure data integrity across replicas in distributed file systems?

Question 35. Compare NFS and SMB/CIFS protocols in terms of their design and use cases.

Question 36. Explain how distributed file systems handle file locking to ensure data consistency.

Question 37. Describe the role of caching in improving performance in distributed file systems.

Question 38. What are the common authentication mechanisms used in distributed file systems?

Question 39. Explain how encryption is used to secure data in transit and at rest in distributed file systems.

Question 40. Describe the challenges of implementing secure access controls in distributed file systems.

Question 41. How does auditing contribute to maintaining security in distributed file systems?

Question 42. Discuss the role of firewalls in protecting distributed file system infrastructures.

Question 43. What are the best practices for securing data stored in distributed file systems?

Question 44. Explain the concept of role-based access control (RBAC) and its implementation in distributed file systems.

Question 45. Describe the chunk-based data storage and transfer mechanism used in distributed file systems.

Question 46. How does data replication contribute to fault tolerance in distributed file systems?

Question 47. Discuss the challenges associated with maintaining data consistency in systems using data replication.

Question 48. Explain the concept of data placement and balancing in distributed file systems.

Question 49. Describe the advantages and disadvantages of client-side caching in distributed file systems.

Question 50. How does data striping improve performance in distributed file systems?

Question 51. Compare and contrast the use of synchronous and asynchronous communication in distributed file systems.

Question 52. What is the role of metadata servers in managing file access in distributed file systems?

Question 53. Explain how distributed file systems manage data transfer across different network topologies.

Question 54. Describe the scalability challenges that distributed file systems face as the number of clients increases.

Question 55. How does data partitioning contribute to scalability in distributed file systems?

Question 56. Discuss the trade-offs between strong consistency and eventual consistency in distributed file systems.

Question 57. Explain the impact of network bandwidth on the performance of distributed file systems.

Question 58. Describe the mechanisms used to optimize data access performance in distributed file systems.

Question 59. How does load balancing improve the overall performance of distributed file systems?

Question 60. Define fault tolerance and explain its importance in distributed file systems.

Question 61. Describe the role of redundancy in achieving fault tolerance in distributed file systems.

Question 62. Explain how distributed file systems handle node failures and maintain data availability.

Question 63. Discuss the challenges of achieving fault tolerance in geographically distributed file systems.

Question 64. Compare the strategies for handling Byzantine faults in distributed file systems.

Question 65. What is the role of consensus algorithms in maintaining data consistency in distributed file systems?

Question 66. Describe the challenges of recovering data after a catastrophic failure in distributed file systems.

Question 67. Provide examples of industries or applications where distributed file systems are commonly used.

Question 68. Describe the architecture and features of the Google File System (GFS) and its impact on distributed computing.

Question 69. How does the Hadoop Distributed File System (HDFS) address the storage and processing needs of big data applications?

Question 70. Discuss the evolution of distributed file systems in cloud computing environments.

Question 71. Describe the use of distributed file systems in supporting real-time data analytics applications.

Question 72. Explain the role of distributed file systems in enabling collaborative work environments.

Question 73. What are the emerging trends in distributed file systems for edge computing applications?

Question 74. Discuss the role of machine learning in optimizing performance and scalability in distributed file systems.

Question 75. How are blockchain technologies influencing the development of distributed file systems?

Question 76. Describe the potential impact of quantum computing on distributed file system architectures.

Question 77. What advancements are being made in the area of metadata management in distributed file systems?

Question 78. Explain how distributed ledger technologies (DLTs) can enhance security in distributed file systems.

Question 79. Discuss the challenges and opportunities of integrating distributed file systems with IoT platforms.

Question 80. How does cloud storage differ from traditional distributed file systems?

Question 81. Explain the concept of hybrid cloud storage and its benefits for distributed file systems.

Question 82. Describe the role of data migration strategies in distributed file systems.

Question 83. How does regulatory compliance impact the design and implementation of distributed file systems?

Question 84. What are the environmental considerations associated with deploying large-scale distributed file systems?

Question 85. Discuss the challenges of data sovereignty and localization in distributed file systems.

Question 86. Explain the concept of self-healing architectures and their relevance to distributed file systems.

7.12 References and Suggested Readings

1. *"Cyber-Security Techniques in Distributed Systems, SLAs and other Cyber Regulations." Cyber Security in Parallel and Distributed Computing: Concepts, Techniques, Applications and Case Studies (2.* **Ghosh, Soumitra, Anjana Mishra, and Brojo Kishore Mishra.**

2. . *"The Google File System."* . **Ghemawat, S., Gobioff, H., & Leung, S.-T. ().** 2003, Proceedings of the 19th ACM Symposium on Operating Systems Principles, , pp. 29-43.

3. *"The Hadoop Distributed File System."*. **Shvachko, K., Kuang, H., Radia, S., & Chansler, R.** 2010, Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1-10.

4. *"Ceph: A scalable, high-performance distributed file system."* . **Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., & Maltzahn, C.** 2006, Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI), , pp. 307-320.

TEXT BOOK

1. Distributed Systems, Concepts and Design, George Coulouris, J Dollimore and Tim Kindberg,Pearson Education, Edition. 2009.

REFERENCE BOOKS

1. Distributed Systems, Principles and Paradigms, Andrew S. Tanenbaum, Maarten Van Steen, 2nd Edition, PHI.
2. Distributed Systems, An Algorithm Approach, Sukumar Ghosh, Chapman&Hall/CRC, Taylor & Fransis Group, 2007

×××