TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

THAPATHALI CAMPUS

Old Question Solution

Of

Distributed System (2078 Bhadra)

Submitted By:

Nandan Singh (THA076BCT020)

Ngawang Sopha Tamang (THA076BCT021)

Pratima Dawadi (THA076BCT030)

Santosh Pandey (THA076BCT041)

Submitted To:

Department of Electronics and Computer Engineering

Thapathali Campus

Kathmandu, Nepal

1) What do you mean by Distributed System? Explain various models of distributed Computing Systems.

A distributed system refers to a collection of interconnected computing devices (such as computers, servers, or devices) that work together to achieve a common goal or provide a service. These devices communicate and coordinate their actions through a network, enabling them to collaborate even if they are physically separate. The primary objective of a distributed system is to improve performance, reliability, scalability, and fault tolerance by distributing tasks and data across multiple devices.

Architectural Models of a distributed system:

Client-server model

The client-server model is a fundamental architecture in distributed computing, where the system is divided into two main components: clients and servers. This model is widely used in various applications and services, ranging from web browsing to email, database management, and more. Here's a closer look at the client-server model:

1. Clients: Clients are devices or software applications that request services, resources, or data from servers. Clients are typically end-user devices such as personal computers, smartphones, tablets, or even embedded systems like IoT devices. Clients initiate requests to servers to perform specific tasks, retrieve information, or access services.

2. Servers: Servers are specialized devices or software applications that provide resources, services, or data to clients. Servers are designed to handle multiple client requests simultaneously, process data, and deliver responses. They have greater processing power, memory, and storage capabilities compared to clients. Servers are responsible for managing resources, maintaining data integrity, and ensuring secure communication.

The client-server model operates through a series of interactions:

- Request: A client initiates a request by sending a message to a server. The message contains the details of the requested service or data.

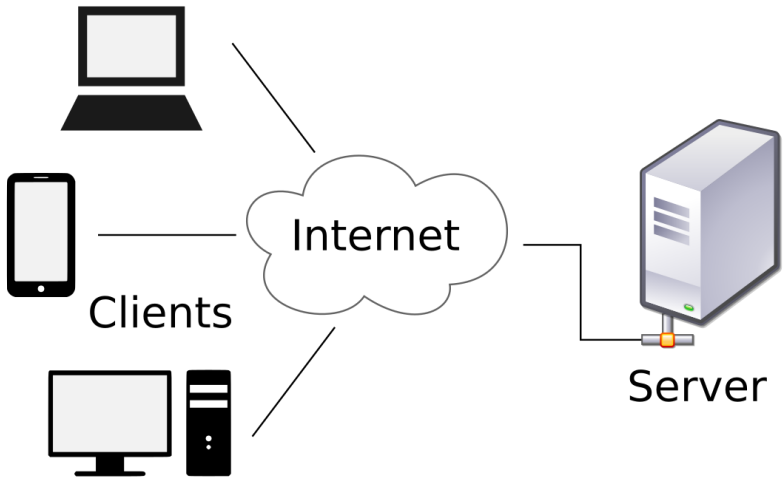
- Processing: The server receives the request, processes it, and performs the necessary computations or accesses the required data.
- Response: The server generates a response based on the request and sends it back to the client.
- Reception: The client receives the response and processes the information as needed.

Key features and advantages of the client-server model include:

- Centralized Management: Servers act as centralized points of control, making it easier to manage and maintain data, security, and resources.
- Scalability: Servers can be scaled up or down to accommodate changes in demand. Additional servers can be added to distribute the load and ensure performance under heavy traffic.
- Resource Sharing: Servers can efficiently manage shared resources such as databases, files, and services, ensuring data consistency and minimizing redundancy.
- Security: By centralizing data and resources on servers, it's possible to implement robust security measures, including access controls, authentication, and encryption.
- Maintenance and Updates: System updates, patches, and maintenance can be applied to servers, which then propagate changes to clients as needed.

However, the client-server model also has limitations:

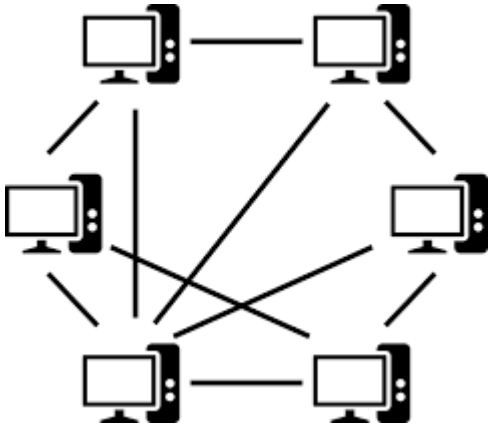
- Single Point of Failure: If a server fails, it can disrupt service for multiple clients relying on it.
- Network Dependency: Both clients and servers require a functional network connection for communication. Network issues can impact the availability of services.
- Scalability Challenges: While server scalability is possible, it requires careful planning and management to avoid bottlenecks and performance issues.
- Resource Overhead: Servers may require more resources (hardware and maintenance) than clients, leading to higher costs



Peer to peer(P2P)

Peer-to-Peer (P2P) networking is a decentralized model in the realm of distributed computing, where devices connect directly with each other to share resources, data, or services without the need for a central server. In a P2P network, every device, known as a "peer," acts both as a client and a server, contributing and consuming resources interchangeably.

P2P networks have gained popularity for various applications, including file sharing, communication, and collaborative computing. Unlike traditional client-server architectures, where a central server manages and distributes resources, P2P networks distribute the load and responsibilities across the participating peers.



Key characteristics and advantages of P2P networking include:

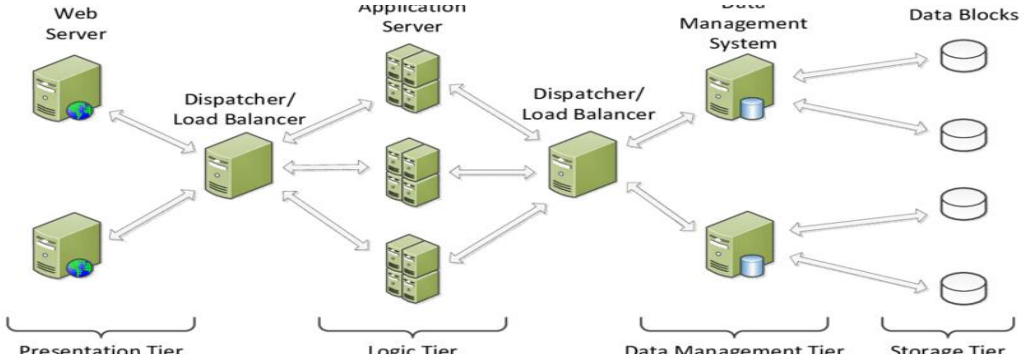
- Decentralization: P2P networks distribute authority and control, eliminating single points of failure and reducing the risk of bottlenecks.

- Scalability: P2P networks can scale easily as more peers join the network, often improving performance and resource availability as the network grows.
- Resource Sharing: Peers can share files, data, processing power, and other resources directly, promoting efficient use of available assets.
- Redundancy: Since data is distributed across multiple peers, P2P networks are inherently fault-tolerant. If one peer fails, the data is still accessible from others.
- Anonymity: P2P networks can offer a degree of anonymity as peers interact directly with each other, reducing the visibility of individual actions to central authorities.
- Resilience: P2P networks can withstand network failures or attacks more effectively, as they don't rely on a single point of control.

However, P2P networking also presents some challenges:

- Security:
- Quality of Service:
- Coordination: Without central control, managing and coordinating tasks and resources in P2P networks can be more intricate.
- Efficiency: Some P2P networks can suffer from inefficiencies due to peers being unevenly distributed or some peers consuming more resources than they contribute.

Multitiered architecture



A multi-tiered architecture in the context of a distributed system refers to a design pattern that divides a distributed application into multiple tiers or layers, each responsible for specific functions. This architectural approach helps manage the complexity of distributed systems by promoting modularity, scalability, and maintainability. In a multi-tiered distributed system

architecture, the tiers are distributed across different physical or virtual machines, which can be geographically dispersed. Here's an overview of the typical tiers found in such an architecture:

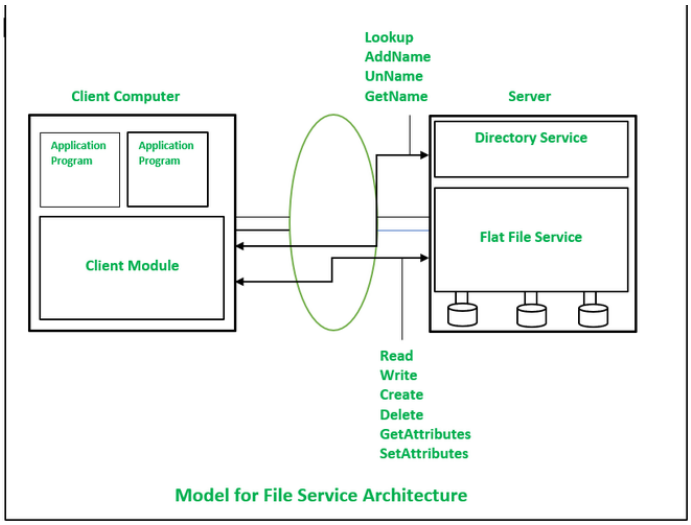
- Presentation Tier:
    - This tier is responsible for handling the user interface and user interactions.
    - It presents information to users and receives user inputs, typically through web browsers, mobile apps, or other client-side interfaces.
    - In a distributed system, the presentation tier might involve multiple user interfaces hosted on various devices.
- Application Logic Tier (Business Logic):
    - This tier contains the core business logic and application processing logic.
    - It processes user requests, implements business rules, and orchestrates interactions between different components.
    - The application logic tier might be distributed across multiple nodes to handle the computational load.
- Application Services Tier:
    - In some architectures, the application logic tier is further divided into an application services tier.
    - This tier hosts specialized services that encapsulate specific business functionalities.
    - The services in this tier can be accessed by various components in the presentation tier.
- Data Processing Tier:
    - This tier is responsible for processing and transforming data before it's sent to the presentation tier.
    - It can include components such as data analytics, data transformation, and data aggregation services.
    - In a distributed system, data processing tasks can be distributed across different nodes for parallel processing.
- Data Storage Tier:
    - This tier manages the storage and retrieval of data required by the application.

    - It can involve various storage solutions such as databases, data warehouses, and distributed file systems.
    - In a distributed system, data storage can be distributed across multiple nodes for fault tolerance and scalability.
- Communication Tier:
    - This tier handles communication between different components and tiers of the distributed system.
    - It ensures that data and messages are exchanged efficiently and reliably across the network.
    - The communication tier can include message queues, communication protocols, and network infrastructure.
- Infrastructure Tier:
    - This tier provides the underlying infrastructure and resources required to host and manage the distributed system.
    - It includes hardware resources, operating systems, virtualization technologies, and cloud services.

2) Elaborate the differences between RMI and RPC. Describe the File Service Architecture of DFS.

S.NO	RPC	RMI
1.	RPC is a library and OS dependent platform.	Whereas it is a java platform.
2.	RPC supports procedural programming.	RMI supports object oriented programming.
3.	RPC is less efficient in comparison of RMI.	While RMI is more efficient than RPC.
4.	RPC creates more overhead.	While it creates less overhead than RPC.
5.	The parameters which are passed in RPC are ordinary or normal data.	While in RMI, objects are passed as parameter.
6.	RPC is the older version of RMI.	While it is the successor version of RPC.
7.	There is high Provision of ease of programming in RPC.	While there is low Provision of ease of programming in RMI.
8.	RPC does not provide any security.	While it provides client level security.
9.	It's development cost is huge.	While it's development cost is fair or reasonable.

File service architecture of DFS



Let's discuss the functions of these components in file service architecture in detail.

1. Flat file service: A flat file service is used to perform operations on the contents of a file. The Unique File Identifiers (UFIDs) are associated with each file in this service. For that long sequence of bits is used to uniquely identify each file among all of the available files in the distributed system. When a request is received by the Flat file service for the creation of a new file then it generates a new UFID and returns it to the requester.

Flat File Service Model Operations:

- Read(FileId, i, n) -> Data: Reads up to n items from a file starting at item 'i' and returns it in Data.
- Write(FileId, i, Data): Write a sequence of Data to a file, starting at item I and extending the file if necessary.
- Create() -> FileId: Creates a new file with length 0 and assigns it a UFID.
- Delete(FileId): The file is removed from the file store.
- GetAttributes(FileId) -> Attr: Returns the file's file characteristics.
- SetAttributes(FileId, Attr): Sets the attributes of the file.

2. Directory Service: The directory service serves the purpose of relating file text names with their UFIDs (Unique File Identifiers). The fetching of UFID can be made by providing the text

name of the file to the directory service by the client. The directory service provides operations for creating directories and adding new files to existing directories.

Directory Service Model Operations:

- Lookup(Dir, Name) -> FileId : Returns the relevant UFID after finding the text name in the directory. Throws an exception if Name is not found in the directory.
- AddName(Dir, Name, File): Adds(Name, File) to the directory and modifies the file's attribute record if Name is not in the directory. If a name already exists in the directory, an exception is thrown.
- UnName(Dir, Name): If Name is in the directory, the directory entry containing Name is removed. An exception is thrown if the Name is not found in the directory.
- GetNames(Dir, Pattern) -> NameSeq: Returns all the text names that match the regular expression Pattern in the directory.

3. Client Module: The client module executes on each computer and delivers an integrated service (flat file and directory services) to application programs with the help of a single API. It stores information about the network locations of flat files and directory server processes. Here, recently used file blocks hold in a cache at the client-side, thus, resulting in improved performance.

3.a) What are the key differences between Network OS and Distributed OS.

Key	Network OS	Distributed OS
Objective	It provides local services to remote clients.	It manages the hardware resources.
Communication	Communication is file-based, shared folder based.	Communication is message-based or shared memory-based.
Scalability	Network OS is highly scalable. A new machine can be added very easily.	Distributed OS is less scalable. The process to add new hardware is complex.
Fault tolerance	Less fault tolerance as compared to distributed OS.	It has very high fault tolerance.
Autonomy	Each machine can acts on its own thus autonomy is high.	It has a poor rate of autonomy
Implementation	Network OS-based systems are easy to build and maintain.	It is difficult to implement a Distributed OS.
Operating System	Network OS-based systems have their own copy of operating systems.	Distributed OS-based nodes have the same copy of the operating system.

3.b) Explain Various roles of middleware in DS.

Middleware plays a crucial role in distributed systems by acting as a layer of software that facilitates communication and interaction between different components or applications across various nodes within the system. It abstracts the complexities of low-level network communication and provides a set of services and tools that help in building and managing distributed applications efficiently. Here are some key roles of middleware in distributed systems:

- Communication Management: Middleware enables seamless communication between distributed components by providing standardized protocols and interfaces. It abstracts away the details of network communication, such as addressing, data serialization, and message routing. This allows developers to focus on the application logic rather than the intricacies of networking.
- Interoperability: Distributed systems often comprise diverse hardware, operating systems, programming languages, and software components. Middleware bridges the gap between these heterogeneous elements, ensuring that they can communicate and work together effectively. It provides a common interface that abstracts the differences between various platforms.
- Data Transformation and Serialization: Middleware assists in converting data formats between systems. It helps in serializing complex data structures into a format that can be easily transmitted over the network and then deserializing them at the receiving end.
- Security and Authentication: Middleware can implement security mechanisms to protect data and resources within a distributed system. This includes encryption, authentication, authorization, and access control, ensuring that only authorized parties can access and manipulate sensitive information.
- Load Balancing: In distributed systems with multiple nodes, middleware can manage the distribution of incoming requests across these nodes to ensure balanced workloads. This helps prevent overload on certain nodes and ensures optimal resource utilization.
- Transaction Management: Middleware provides mechanisms for managing distributed transactions, ensuring that multiple operations across different nodes can be coordinated and maintained in a consistent manner, even in the presence of failures.
- Fault Tolerance and Error Handling: Middleware can implement fault tolerance mechanisms, such as replication and redundancy, to ensure that the system remains operational even in the face of hardware or software failures. It can also handle error detection, reporting, and recovery.
- Caching and Data Distribution: Middleware can facilitate data caching and distribution, allowing frequently accessed data to be stored closer to the application, reducing latency and improving overall performance.

- Scalability: Middleware can enable easy scalability of distributed systems by providing tools and techniques for adding or removing nodes dynamically, without significant disruptions to the system.
- Monitoring and Management: Middleware often includes tools for monitoring the health and performance of the distributed system. This includes features such as logging, monitoring dashboards, and resource management.
- Message Brokering and Publish/Subscribe: Middleware can provide message brokering capabilities, allowing different components to communicate via a publish/subscribe mechanism, where publishers send messages to specific topics and subscribers receive messages based on their interests.

4) What are design Issues of NFS? Show the complete working of DNS.

Network File System (NFS) is a distributed file system protocol that allows remote access to files over a network. While NFS offers many benefits for sharing and accessing files across different machines, there are several design issues and challenges associated with its implementation. Here are some important design issues related to NFS:

- Security and Authentication: NFS was originally designed with a focus on ease of use and performance, often at the expense of strong security measures. This has led to security concerns, as data transferred over the network can be susceptible to eavesdropping and unauthorized access. Implementing strong authentication and encryption mechanisms is crucial to address these security issues.
- Access Control: NFS traditionally relies on a simple permission model, based on the user and group IDs of the client and server systems. This model can lead to access control problems, as user and group IDs might not be consistent across different machines. More advanced access control mechanisms, such as Access Control Lists (ACLs), have been added in later versions of NFS to address this issue.
- Performance and Latency: NFS performance can be affected by factors such as network latency, server load, and the size of files being transferred. The design of NFS should consider optimizing read and write operations, caching strategies, and minimizing the impact of network delays to ensure good performance.

- Concurrency and File Locking: NFS should handle concurrent access to files by multiple clients correctly. This requires effective file locking mechanisms to prevent conflicts and maintain data consistency. Poorly implemented file locking can lead to data corruption and unexpected behavior.
- State Management: NFS has both stateful and stateless components. State management becomes a challenge when clients need to recover from server failures or network interruptions. Proper mechanisms for handling disconnected clients and maintaining consistent states are essential.
- Client Failures: When a client machine fails or loses network connectivity, it can lead to issues like stale file handles and inconsistent data. Effective recovery mechanisms and techniques for handling client failures are important to ensure data integrity.
- Versioning and Compatibility: NFS has evolved over time, resulting in multiple versions of the protocol. Ensuring compatibility between different versions of NFS, especially when upgrading systems, can be complex. Proper version negotiation and fallback mechanisms are crucial to maintaining interoperability.
- Data Caching and Consistency: NFS employs caching to improve performance, but this can lead to issues related to data consistency. Maintaining a balance between caching to enhance performance and ensuring that clients always have access to the most up-to-date data requires careful design and implementation.
- Scalability: As the number of clients and the amount of data increase, NFS must be able to scale to handle the growing load. Designing NFS to be scalable requires considerations such as load balancing, distributing metadata, and optimizing server resources.
- Data Integrity: Ensuring the integrity of data stored and transferred via NFS is essential. Issues like data corruption during transmission, disk failures, and server crashes need to be addressed to maintain reliable data storage and access.
- Data Migration and Backup: Designing NFS systems that support data migration, backups, and disaster recovery is important for maintaining data availability and reliability.

A domain name serves as a distinctive identification for a website. To make it simpler for consumers to visit websites, it is used in place of an IP address.

A device connected to the internet is given an IP address, which is a numerical identity. It's used to pinpoint where on the internet a website or gadget is.
An application called a DNS resolver is in charge of translating domain names into IP addresses. The DNS resolver contacts a DNS server to seek the IP address associated with a domain name when a user types it into their web browser.

DNS Server: A DNS server is a piece of software that keeps track of domain names and IP addresses. It answers with the appropriate IP address to requests from DNS resolvers.
The initial point of contact in the DNS system is a DNS root server. It offers details on the DNS servers in charge of top-level domains (TLDs), including.com,.org, and.net.
TLD Server: A TLD server is a DNS server that is in charge of keeping track of data on domain names that fall under a certain top-level domain, such as.com or.org.

DNS stands for a Domain Name System. DNS resolves names to numbers, to be more specific it resolves domain names to IP addresses. So if you type in a web address in your web browser, DNS will resolve the name to a number because the only thing computers know are numbers.

Working: If you wanted to go to a certain website you would open up your web browser and type in domain name of that website. Let us use google.com. Now technically you really do not have to type in google.com to retrieve Google web page, you can just type in IP address instead if you already know what google's IP address is, but since we are not accustomed to memorizing and dealing with numbers, especially when there are millions of websites on Internet, we can just type in domain name instead and let DNS convert it to an IP address for us.

So back to our example, when you type google.com on your web browser DNS server will search through its cache to find a matching IP address for that domain name, and when it finds it it will resolve that domain name to IP address of Google web site, and once that is done then your computer is able to communicate with a Google web server and retrieve the webpage.

5) Write Implementation rule of Lamport Clock. State the limitations of Lamport Logical clock.

Lamport's Logical Clock was created by Leslie Lamport. It is a procedure to determine the order of events occurring. It provides a basis for the more advanced Vector Clock Algorithm. Due to the absence of a Global Clock in a Distributed Operating System Lamport Logical Clock is needed.

Algorithm:

- **Happened before relation(->):** a -> b, means 'a' happened before 'b'.
- **Logical Clock:** The criteria for the logical clocks are:
  [C1]: Ci (a) < Ci(b), [Ci -> Logical Clock, If 'a' happened before 'b', then time of 'a' will be less than 'b' in a particular process.]
  [C2]: Ci(a) < Cj(b), [Clock value of Ci(a) is less than Cj(b)]

Reference:

Process: Pi

Event: Eij, where i is the process in number and j: jth event in the ith process.

tm: vector time span for message m.

Ci vector clock associated with process Pi, the jth element is Ci[j] and contains Pi's latest value for the current time in process Pj.

d: drift time, generally d is 1.

Implementation Rules[IR]:

1) [IR1]: If a -> b ['a' happened before 'b' within the same process] then, Ci(b) =Ci(a) + d
2) [IR2]: Cj = max(Cj, tm + d) [If there's more number of processes, then tm = value of Ci(a), Cj = max value between Cj and tm + d]

For Example:

Take the starting value as 1, since it is the 1st event and there is no incoming value at the starting point:

e11 = 1

e21 = 1

The value of the next point will go on increasing by d (d = 1), if there is no incoming value i.e., to follow [IR1].

e12 = e11 + d = 1 + 1 = 2

e13 = e12 + d = 2 + 1 = 3

e14 = e13 + d = 3 + 1 = 4

e15 = e14 + d = 4 + 1 = 5

e16 = e15 + d = 5 + 1 = 6

e22 = e21 + d = 1 + 1 = 2

e24 = e23 + d = 3 + 1 = 4

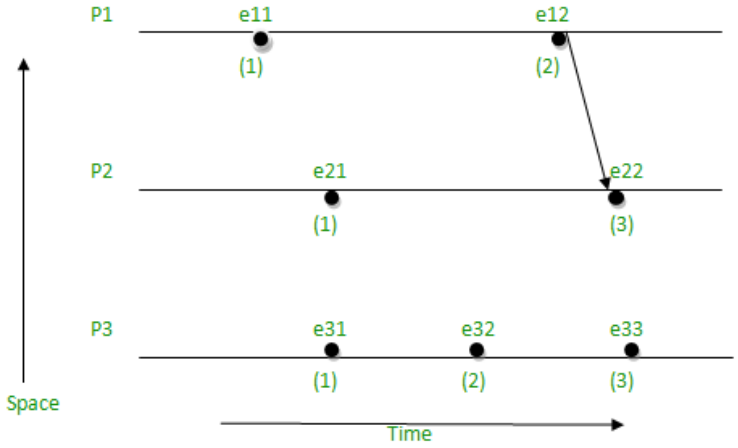
e26 = e25 + d = 6 + 1 = 7

When there will be incoming value, then follow [IR2] i.e., take the maximum value between Cj and Tm + d.

e17 = max(7, 5) = 7, [e16 + d = 6 + 1 = 7, e24 + d = 4 + 1 = 5, maximum among 7 and 5 is 7]

e23 = max(3, 3) = 3, [e22 + d = 2 + 1 = 3, e12 + d = 2 + 1 = 3, maximum among 3 and 3 is 3]

e25 = max(5, 6) = 6, [e24 + 1 = 4 + 1 = 5, e15 + d = 5 + 1 = 6, maximum among 5 and 6 is 6]



Limitation:

In case of [IR1], if a -> b, then C(a) < C(b) -> true.

In case of [IR2], if a -> b, then C(a) < C(b) -> May be true or may not be true.

6) Compare and contrast token based mutual exclusion algorithm and non-token based mutual exclusion algorithm. Explain the various state of Ricart Agrawala token based mutual exclusion.

A distributed system is a system in which components are situated in distinct places, these distinct places refer to networked computers which can easily communicate and coordinate their tasks by just exchanging asynchronous messages with each other. These components can communicate with each other to conquer one common goal as a task. There are many algorithms are used to achieve Mutual Exclusion ,where there are multiple processes(or sites) requesting access for a single shared resource(often called as Critical Section) ,and these are broadly divided into 2 categories: Token-Based Algorithms and Non-Token Based Algorithms.

Difference between Token based and Non-Token based Algorithms in Distributed Systems:

S.No.	Token Based Algorithms	Non-Token Based Algorithms
1.	In the Token-based algorithm, a unique token is shared among all the sites in Distributed Computing Systems.	In Non-Token based algorithm, there is no token even not any concept of sharing token for access.
2.	Here, a site is allowed to enter the Critical Section if it possesses the token.	Here, two or more successive rounds of messages are exchanged between sites to determine which site is to enter the Critical Section next.
3.	The token-based algorithm uses the sequences to order the request for the Critical Section and to resolve the conflict for the simultaneous requests for the System.	Non-Token based algorithm uses the timestamp (another concept) to order the request for the Critical Section and to resolve the conflict for the simultaneous requests for the System.
4.	The token-based algorithm produces less message traffic as compared to Non-Token based Algorithm.	Non-Token based Algorithm produces more message traffic as compared to the Token-based Algorithm.
5.	They are free from deadlock (i.e. here there are no two or more processes are in the queue in order to wait for messages that will actually can't come) because of the existence of unique token in the distributed system.	They are not free from the deadlock problem as they are based on timestamp only.
6.	Here, it is ensured that requests are executed exactly in the order as they are made in.	Here there is no surety of execution order.
7.	Token-based algorithms are more scalable as they can free your server from storing session state and also they contain all the necessary information which they need for authentication.	Non-Token based algorithms are less scalable than the Token-based algorithms because server is not free from its tasks.

S.No.	Token Based Algorithms	Non-Token Based Algorithms
8.	Here the access control is quite Fine-grained because here inside the token roles, permissions and resources can be easily specifying for the user.	Here the access control is not so fine as there is no token which can specify roles, permission, and resources for the user.
9.	Token-based algorithms make authentication quite easy.	Non-Token based algorithms can't make authentication easy.
10.	Examples of Token-Based Algorithms are: **(i)** Singhal's Heuristic Algorithm **(ii)** Raymonds Tree Based Algorithm **(iii)** Suzuki-Kasami Algorithm	Examples of Non-Token Based Algorithms are: **(i)** Lamport's Algorithm **(ii)** Ricart-Agarwala Algorithm **(iii)** Maekawa's Algorithm

The Ricart-Agrawala algorithm is a mutual exclusion algorithm used in distributed systems to ensure that only one process at a time can access a shared resource. It uses a token-based approach where processes request and release a token to enter the critical section. The algorithm is designed to prevent conflicts and ensure safe access to the critical section while minimizing contention.

In the context of the Ricart-Agrawala algorithm, there are two main states that processes can be in: the Requesting State and the CriticalSection State. Let's explore these states:

1) Requesting State:
   - When a process wants to enter the critical section, it transitions to the Requesting State.
   - The process sends a request message to all other processes in the system, indicating its intention to enter the critical section.
   - Upon sending a request, the process waits for replies from all other processes.
   - The process can only enter the CriticalSection State when it has received replies from all other processes and has the token (permission) to enter.
   - If another process has the token and is not currently in the critical section, it will reply positively to the requesting process's request. Otherwise, it will queue the request.

2) Critical Section State:

- Once a process has received replies from all other processes and has the token, it transitions to the CriticalSection State.
- In this state, the process is granted exclusive access to the critical section and can perform its critical section tasks.
- Other processes that wish to enter the critical section must wait until the current process releases the token and transitions out of the CriticalSection State.
- It's important to note that while the algorithm ensures mutual exclusion, it also prioritizes processes that have requested access earlier. This means that if multiple processes are requesting access, the process that made the request earlier will be granted access first when the token becomes available.

7) Mention the Requirements and challenges of replication. Explain active replication model in fault tolerance. How is it different than passive model?

Requirements of Replication:

- Data Availability and Reliability
- Performance Improvement
- Load Balancing
- Scalability
- Geographic Distribution

Challenges of Replication:

- Data Consistency
- Conflict Resolution
- Replica Synchronization
- Consistency Models
- Fault Tolerance
- Concurrency Control
- Performance Overhead
- System Complexity

- Consensus Protocols: (Many replication systems use consensus protocols (e.g., Paxos, Raft) to agree on updates across replicas. Implementing and managing these protocols introduces challenges related to correctness and performance.)
- Versioning and Garbage Collection

The active replication model is a fault tolerance technique used in distributed systems to enhance system reliability and availability. It involves maintaining multiple copies (replicas) of a service or data and ensuring that these replicas are continuously updated and synchronized in real-time. This model aims to provide uninterrupted service even in the presence of failures by allowing one of the replicas to take over in case of a fault.

Let's break down the sequence of events when a client requests an operation to be performed in a distributed system that utilizes the active replication model:

Request:

The process begins when a client sends a request to the distributed system. This request can be a specific operation that the client wants to be performed, such as retrieving data, updating a resource, or executing a task.

Coordination:

Upon receiving the request, the distributed system needs to coordinate the execution of the operation. This involves determining which replica (primary or backup) should handle the request. The coordination process ensures that the request is directed to the appropriate replica.

Execution:

The designated primary replica receives the request and begins the execution of the operation. It processes the request, which may involve reading or modifying data, performing calculations, or any other necessary actions based on the nature of the operation.

Agreement:

In an active replication model, maintaining synchronization among replicas is crucial. Once the primary replica completes the execution of the operation, it sends updates (changes resulting from the operation) to the backup replicas. The replicas need to agree on these updates to ensure consistency.

Response:

After the operation is executed and the updates are propagated to the backup replicas, the primary replica generates a response. This response contains the result of the operation or an acknowledgment of its completion. The response is then sent back to the client who initiated the request.

Failover (if applicable):

If the primary replica experiences a failure before responding to the client, a failover mechanism may be triggered. In this case, one of the backup replicas is promoted to become the new primary replica, ensuring that the system remains operational and can continue serving client requests.

Recovery and Consistency:

The system works to recover the failed primary replica and restore its state to match the state of the other replicas. This ensures that consistency is maintained across all replicas, even after a failure and failover.

Active replication and passive replication are two different approaches to achieving fault tolerance in distributed systems. They have distinct characteristics and mechanisms for handling failures and maintaining system availability. Here's how they differ:

S.N.	Active Replication	Passive Replication
1	Primary replica handles requests, backup replicas stay synchronized.	All replicas maintain independent states.
2	Real-time updates maintain synchronization.	Replicas execute operations independently.
3	Immediate failover to back up on primary replica failure.	No immediate failover mechanism.
4	Quick recovery, high availability.	Potential replication delays, minor inconsistencies.
5	Overhead due to synchronization, risk of inconsistency.	Better performance, acceptable for minor inconsistencies.

8) Discuss how Consensus can be achieved in Distributed System. Explain the check pointing approach for distributed recovery.

Consensus in a distributed system refers to the process of reaching an agreement or making a collective decision among a group of nodes, even in the presence of failures or network delays. Achieving consensus is a fundamental challenge in distributed systems, as nodes may have

differing information, experience failures, or exhibit varying behavior due to factors like network latency. The overview of consensus process are as follows:

a. **Proposal:** A participant (often referred to as a proposer) suggests a value or decision that needs to be agreed upon by the group.
b. **Voting:** Other participants (acceptors) vote on whether to accept or reject the proposal. Participants can vote "yes" or "no."
c. **Majority Rule:** For consensus to be achieved, a majority of participants must agree on the proposal. This ensures that the agreed-upon value is chosen in a decentralized manner.

Checkpoint is a point of time at which a record is written onto the database from the buffers. As a consequence, in case of a system crash, the recovery manager does not have to redo the transactions that have been committed before checkpoint. Periodical checkpointing shortens the recovery processes.

The two types of checkpointing techniques are:

- Consistent checkpointing
- Fuzzy checkpointing

a. Consistent Checkpointing

Consistent checkpointing creates a consistent image of the database at checkpoint. During recovery, only those transactions which are on the right side of the last checkpoint are undone or redone. The transactions to the left side of the last consistent checkpoint are already committed and needn't be processed again. The actions taken for checkpointing are:

i. The active transactions are suspended temporarily.
ii. All changes in main-memory buffers are written onto the disk.
iii. A "checkpoint" record is written in the transaction log.
iv. The transaction log is written to the disk.
v. The suspended transactions are resumed.

If in step (iv), the transaction log is archived as well, then this checkpointing aids in recovery from disk failures and power failures, otherwise it aids recovery from only power failures.

b. Fuzzy Checkpointing

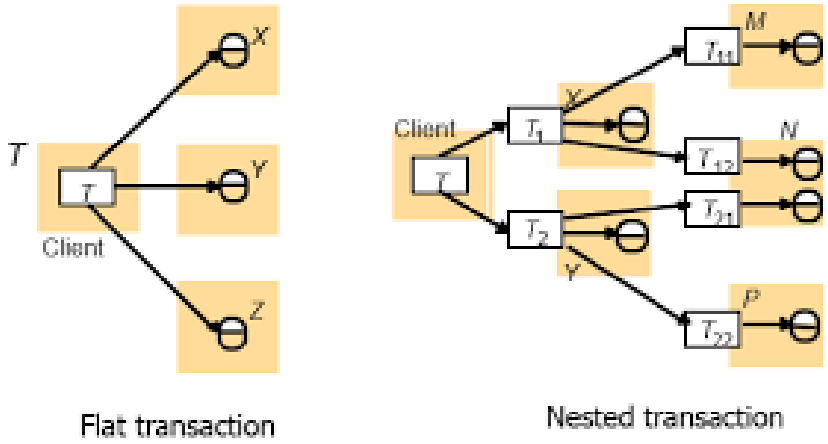
In fuzzy checkpointing, at the time of checkpoint, all the active transactions are written in the log. In case of failure, the recovery manager processes only those transactions that were

active during checkpoint and later. The transactions that have been committed before checkpoint are written to the disk and hence need not be redone.

9) What are the Flat and Nested Transactions? Explain how the problems of 2PC protocols are solved by 3PC.

In flat transaction, a client makes requests to more than one server. It completes each of its requests before going on to next one, accessing server objects sequentially.

In nested transaction, the top-level open sub transactions, each of which can further open sub transactions. The sub transactions at same level run concurrently.



The two-phase commit protocol allows a server to decide to abort unilaterally. It includes timeout actions to deal with delays due to servers crashing. The two-phase protocol can take an unbounded amount of time to complete but is guaranteed to complete eventually. Big Problem with Two-Phase Commit:

a. It can lead to both the coordinator and the group members blocking, which may lead to the dreaded deadlock.

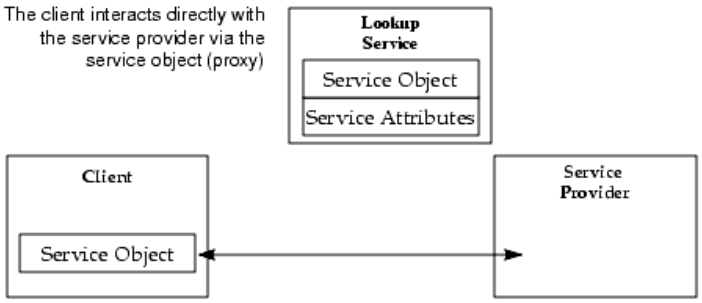
b. If the coordinator crashes, the group members may not be able to reach a final decision, and they may, therefore, block until the coordinator recovers. Hence, Two- Phase Commit is known as a blocking-commit protocol for this reason.

c. The solution to this problem is given by The Three-Phase Commit Protocol.

The 3-Phase Commit (3PC) protocol addresses the limitations of the 2-Phase Commit (2PC) algorithm by incorporating additional phases that enhance fault tolerance and reliability in distributed systems. In contrast to 2PC's potential for indefinite blocking when the coordinator fails after collecting "yes" votes but before finalizing the commit decision, 3PC introduces a "pre-commit" phase. This phase ensures that, even if the coordinator crashes after sending "pre-commit" messages, participants will eventually time out and proceed to commit if no further communication is received. Furthermore, the 3PC protocol resolves the issue of blocking in the face of participant failure by introducing a "can-commit" phase prior to the "pre-commit" phase. In this step, participants acknowledge their ability to commit or abort along with their votes, allowing the coordinator to make informed decisions even in the presence of participant failures. Additionally, to mitigate availability problems where the coordinator becomes unavailable, 3PC adds a "pre-prepare" phase, during which the coordinator proposes a transaction to participants, who acknowledge receipt. This enhances readiness for commit or abort, ensuring that participants are prepared to take action when the actual decision is made. These added phases in 3PC collectively bolster fault tolerance, reduce blocking scenarios, and enhance the overall reliability of distributed consensus processes.

10) Write short notes on: (Any Two)

a) JINI



JINI is a service-oriented architecture that defines a programming model that both exploits and extends Java technology. This programming model enables the construction of secure, distributed systems consisting of federations of well-behaved network services. Jini helps to build networks that are scalable and flexible, which are required attributes in distributed computing scenarios. Jini's main objective is to shift the focus of distributed computing from

a disk-drive-oriented approach to a network-adaptive approach by developing scalable, evolvable and flexible dynamic computing environments. Jini makes resources over a network look like local resources. The basic components of Jini are:

1. **Service:** A service in Jini refers to an entity that offers a specific functionality or capability that can be utilized by other entities, such as people, devices, or other services. Services can encompass a wide range of functionalities, from printing documents to displaying videos, and more. They play a crucial role in enabling collaboration and resource sharing within a networked environment.
2. **The Lookup Service:** The Lookup Service is a fundamental concept in the Jini framework. It acts as a centralized registry or directory where services can be registered and discovered by clients. Clients use the Lookup Service to find and locate services that fulfill specific requirements. This decouples clients from having to know the exact network location or details of each service they need to interact with, making the system more flexible and dynamic.
3. **Proxy Object and Its Attributes:** In Jini, proxy objects serve as intermediaries between clients and remote services. These proxy objects implement all the interfaces provided by the remote service, allowing clients to interact with the service as if it were a local object. Attributes are used to distinguish one proxy object from other objects of the same type. These attributes can include information like service version, security settings, or other characteristics that help clients make informed decisions about which service instance to use.
4. **Client:** Clients in the Jini framework are entities that request services provided by other entities in the network. Clients leverage the Lookup Service to discover and locate the desired services based on their requirements. Once a suitable service is located, the client interacts with it through proxy objects, making remote service access appear seamless and similar to local object interactions.

b) Distributed Debugging:

Distributed debugging is a process of identifying, diagnosing, and resolving issues in software applications that run across multiple machines or nodes in a distributed or networked environment. As applications become more complex and are distributed across various components, servers, and devices, debugging becomes challenging due to the increased potential for interactions, dependencies, and communication issues.

Key Aspects of Distributed Debugging:

Remote Debugging: Distributed debugging involves debugging code that runs on remote machines. Developers use specialized tools and techniques to remotely monitor and analyze code execution, even on different servers or devices.

Interactions and Dependencies: Debugging distributed systems requires considering interactions between various components. A bug in one component might manifest as an issue in another component due to interdependencies.

Log Analysis: Logs generated by distributed components play a crucial role in identifying issues. Analyzing logs helps pinpoint errors, communication failures, or performance bottlenecks across the system.

Message Tracing: Distributed systems often communicate via messages or API calls. Tracing messages allows developers to track their path through different components, identifying where errors occur.

Replication and Consistency: Debugging distributed databases and storage systems involves ensuring data consistency across multiple replicas, detecting synchronization issues, and resolving conflicts.

Latency and Timing: Debugging distributed systems requires accounting for network latency and timing issues that can impact communication and coordination between components.

Concurrency and Parallelism: Identifying race conditions, deadlocks, and other concurrency-related issues becomes more complex in distributed environments.

Collaboration: Distributed teams may work on different parts of a system. Effective collaboration tools and practices are vital to share findings and resolve issues collectively.

Failure Analysis: Distributed systems are prone to failures, such as node crashes or network disruptions. Debugging involves understanding how failures propagate and affect the system.

Debugging distributed systems demands specialized skills and tools to navigate the complexities introduced by remote components, communication, and concurrency. Techniques like logging, tracing, remote debugging tools, and monitoring dashboards are essential to diagnose and fix issues efficiently in these complex environments.