TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

THAPATHALI CAMPUS

Old Question Solution

Of

Distributed System

Submitted By:

Aayush Regmi **(THA076BCT002)**

Anuj Rayamajhi **(THA076BCT007)**

Arun Subedi **(THA076BCT010)**

Kaustuv Karki **(THA076BCT017)**

Submitted To:

Department of Electronics and Computer Engineering

Thapathali Campus

Kathmandu, Nepal

August, 2023

2079 BHADRA (REGULAR)

Q1. Transparency is one of the important design goals of distributed system. Justify it. List out the advantages of distributed system over centralized system.

Solution:

Transparency refers to the idea that the complexities and intricacies of a distributed system are hidden from users and applications, making the system appear as if it were a single, cohesive entity. There are several types of transparency that contribute to the overall effectiveness and usability of distributed systems:

1. User Transparency

2. Location Transparency

3. Access Transparency

4. Concurrency Transparency

5. Replication Transparency

6. Failure Transparency

7. Migration Transparency

Justifying the importance of transparency in distributed systems:

1. **Simplicity and Usability:** Transparency simplifies the interaction with complex distributed systems, making them more approachable for developers and end-users. This leads to better usability and reduced learning curves.

2. **Flexibility and Scalability**: Transparent systems can be easily scaled and adapted without requiring extensive changes to applications. Users can interact with resources and services regardless of their physical location or underlying changes.

3. **Maintenance and Evolution**: Transparent systems are easier to maintain and evolve over time. Changes to the underlying infrastructure can be made without impacting users, leading to better system reliability and adaptability.

4. **Reduced Development Complexity**: By abstracting away many of the distributed system's intricacies, developers can focus more on the application logic and less on handling low-level networking and synchronization details.

5. **Consistency and Reliability**: Transparency helps maintain consistency and reliability by providing mechanisms to handle failures, replication, and migration without burdening users or applications.

6. **Interoperability**: Transparency enables interoperability between different components and services, as they can communicate seamlessly without needing to understand each other's internal workings.

Distributed systems offer several advantages over centralized systems. Here are some key advantages of distributed systems:

1. Scalability: Distributed systems can be easily scaled out by adding more nodes to the network. This allows them to handle increased workloads and user demands effectively. In contrast, centralized systems might encounter performance bottlenecks as they grow.

2. Performance: Distributed systems can achieve better performance by distributing tasks across multiple nodes, enabling parallel processing. This leads to faster response times and improved throughput compared to centralized systems that might struggle to manage heavy workloads.

3. Fault Tolerance: Distributed systems are inherently more fault-tolerant. If a single node fails, the system can continue to function by redistributing tasks and resources to other nodes. This is in contrast to centralized systems, where a failure can lead to a complete system outage.

4. Reliability: Distributed systems can achieve higher levels of reliability because data and services can be replicated across multiple nodes. This redundancy ensures that if one node fails, another can take over seamlessly, minimizing disruptions.

5. Resource Utilization: Distributed systems allow better utilization of resources since tasks can be distributed based on the available computing power and capacity of each node. This contrasts with centralized systems, where unused resources on one node cannot benefit other tasks.

6. Geographical Distribution: Distributed systems can span multiple geographical locations, enabling better service delivery to users in different regions. This leads to reduced latency and improved user experience.

7. Load Balancing: Distributed systems can distribute workloads across nodes to ensure even resource utilization and prevent overloading of specific nodes. Load balancing enhances system performance and responsiveness.

QN2. What is recursive and iterative query? Describe working mechanism of DNS with suitable example. Mention the role of distributed file system.

Solution:

Recursive Query:

In a recursive query, the DNS client (usually a user's computer or a network device) sends a query to its local DNS resolver, requesting the resolver to perform all the necessary steps to obtain the final answer. The resolver then contacts various DNS servers on the user's behalf until it receives a complete answer or determines that the domain name cannot be resolved. The resolver is responsible for chasing down the information and returning the final result to the client.

Iterative Query:

In an iterative query, the DNS client sends a query to a DNS server, and the server provides the best available answer it currently has. If the server doesn't have the answer, it will refer the client to another DNS server that might have more information about the domain. This process continues until the client receives a complete answer or until the chain of referrals

ends. The client is responsible for performing subsequent queries to other DNS servers if needed.

Working of DNS

1. A user opens a web browser, enters www.example.com in the address bar, and presses Enter.
2. The request for www.example.com is routed to a DNS resolver, which is typically managed by the user's Internet service provider (ISP), such as a cable Internet provider, a DSL broadband provider, or a corporate network.
3. The DNS resolver for the ISP forwards the request for www.example.com to a DNS root name server.
4. The DNS resolver for the ISP forwards the request for www.example.com again, this time to one of the TLD name servers for .com domains. The name server for .com domains responds to the request with the names of the four Amazon Route 53 name servers that are associated with the example.com domain.
5. The DNS resolver for the ISP chooses an Amazon Route 53 name server and forwards the request for www.example.com to that name server.

Role of Distributed File System:

A distributed file system is a network-based file system that allows multiple users or applications to access and manage files stored across multiple servers or nodes. In the context of DNS, a distributed file system might play a role in providing redundancy and fault tolerance for DNS zone files, which contain DNS records mapping domain names to IP addresses. These files need to be consistent and available across multiple DNS servers to ensure the reliable functioning of the DNS system.

For example, if a domain name is associated with multiple IP addresses for load balancing or fault tolerance, the DNS records corresponding to that domain need to be stored and synchronized across multiple servers. A distributed file system can ensure that updates to

these records are propagated to all authoritative DNS servers in a consistent and timely manner. This helps maintain the accuracy and availability of DNS information, even in the face of server failures or updates.

Qn3. Define distributed object and remote interface. How RMI helps in distributed programming model? Explain with architecture.

Distributed Object:

A distributed object is an object-oriented programming concept that extends the idea of objects beyond a single process or machine. It involves creating objects that can be accessed and manipulated across different nodes or machines within a distributed system. Distributed objects encapsulate both data and behavior, allowing them to communicate and collaborate seamlessly over a network. Each distributed object has a unique identity, and interactions with it can involve method calls and data exchange.

Remote Interface:

A remote interface defines a contract or set of methods that a distributed object provides for remote access and communication. It serves as a way to specify how other objects or components can interact with the distributed object from a remote location. Remote interfaces usually define the methods that can be invoked on the object from a different process or machine, as well as the parameters and return values for those methods. By adhering to the remote interface, the distributed object ensures that its behavior is accessible and consistent across distributed boundaries.

Remote Method Invocation (RMI) is a technology in Java that facilitates distributed programming by allowing objects in different Java Virtual Machines (JVMs) to communicate and invoke methods on each other as if they were local objects. RMI provides a mechanism for creating distributed applications using an object-oriented paradigm, making it easier for developers to design and build distributed systems. Here's how RMI helps in achieving a distributed programming model, along with its architecture:

How RMI Helps in Distributed Programming Model:

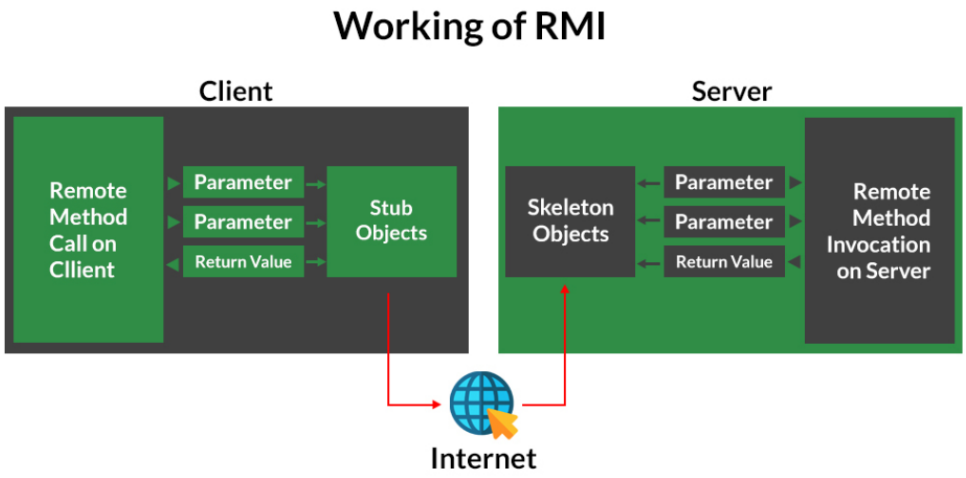
1. **Abstraction of Network Communication:** RMI abstracts the complexities of network communication, making it easier for developers to focus on designing and implementing the application's logic rather than dealing with low-level networking details.
2. **Seamless Method Invocation:** RMI enables developers to invoke methods on remote objects in a manner similar to invoking methods on local objects, abstracting away the fact that the objects are actually located on different machines.
3. **Object-Oriented Approach:** RMI follows an object-oriented approach, allowing developers to work with distributed objects that encapsulate both data and behavior. This is consistent with the principles of object-oriented programming and promotes code reusability and maintainability.
4. **Location Transparency:** With RMI, developers do not need to be concerned about the physical location of objects. The underlying infrastructure handles the communication and remote method invocation transparently.
5. **Dynamic Class Loading:** RMI supports dynamic class loading, which means that objects and classes can be loaded and used on-demand during runtime, allowing for flexible and efficient distribution of code.

RMI Architecture:

The RMI architecture involves several components working together to enable remote communication between objects:

1. **Remote Interface:** This defines the methods that a remote object provides. It specifies the methods that can be invoked remotely and the data types of parameters and return values. Clients use the remote interface to communicate with the remote object.
2. **Remote Object:** A remote object implements the remote interface and provides the actual implementation of the methods defined in the interface. It extends the **java.rmi.server.UnicastRemoteObject** class to enable remote communication.

3. **Stubs and Skeletons:** Stubs on the client side and skeletons on the server side are generated by the RMI compiler. The stub acts as a proxy for the remote object on the client side, while the skeleton receives incoming requests on the server side and forwards them to the actual remote object.
4. **Registry:** The RMI registry acts as a centralized directory service that maps remote object references to their locations. It provides a way for clients to locate remote objects by looking up their references in the registry.
5. **RMI Compiler (rmic):** The RMI compiler generates stub and skeleton classes based on the remote interface. These classes are responsible for marshaling and unmarshaling method calls and results for network transmission.
6. **Java Virtual Machine (JVM):** RMI enables objects in different JVMs to communicate with each other. Each JVM hosts the execution environment for a distributed object.



Q4: Define cuts of a distributed computation along with its types. How is casual ordering of message realized using vector clocks?

Solution:

In the context of distributed computing, a "cut" refers to a partitioning of events within a distributed system's execution history. It separates events into two distinct sets: the "past"

and the "future." A cut helps to analyze the distributed computation by dividing events based on their causal relationships and their occurrence in time.

Types of Cuts:

1. **Global State Cut:** A global state cut is a division of events into two sets: those that occurred before the cut and those that occurred after the cut. It represents a snapshot of the entire distributed system's state at a specific point in time. This type of cut provides a view of the distributed computation's state and event relationships at a particular moment.
2. **Computation Cut:** A computation cut divides events into two sets: those that are causally related to the events outside the cut and those that are not. It captures the causal relationships among events within the distributed system. Computation cuts help analyze how events are ordered in terms of causality.

Casual Ordering of Messages using Vector Clocks:

Vector clocks are a technique used in distributed systems to establish causal relationships among events and messages. Each process maintains a vector clock, which is a data structure containing a vector of counters. Each counter corresponds to a process, and its value indicates the number of events that have occurred in that process's history.

The process of achieving causal ordering of messages using vector clocks involves the following steps:

1. **Initialization:** Each process initializes its vector clock with zeros.
2. **Event Increment:** Whenever a process performs an internal event (like processing data or executing a local computation), it increments its own counter in the vector clock.
3. **Sending Messages:** When a process sends a message, it includes its current vector clock in the message.
4. **Receiving Messages:** Upon receiving a message, the receiving process updates its vector clock component for the sending process to the maximum of its own counter

and the received counter. This reflects the fact that the receiving process knows about the sending process's past events up to the point when the message was sent.

5. **Causal Order Determination:** A message is considered causally ordered with respect to another message if the sender's vector clock is less than or equal to the receiver's vector clock for all processes, and at least one component is strictly less. This ensures that the sender's events are in the past of the receiver's events, establishing a causal relationship.

Q6. Explain with example how can you achieve consensus in distributed system. explain ricart-agrawala token-based algorithm.

Solution:

Achieving consensus in a distributed system refers to the process of having all nodes or processes in the system agree on a certain value or decision, even in the presence of failures and network delays. Consensus is a fundamental problem in distributed systems and is often achieved through protocols like the Paxos algorithm or the Raft algorithm.

Example: Consensus for Choosing a Leader

Imagine a distributed system with multiple nodes that need to choose a leader node among them. The leader will be responsible for coordinating actions and making decisions on behalf of the whole system. To achieve consensus on the leader, the system can use a basic two-phase voting protocol.

1. **Phase 1: Nomination**
    - Each node initiates the process by sending a "nominate yourself as leader" message to all other nodes.
    - Each node collects the nominations from other nodes and determines which node has the most nominations.
    - If a node receives a majority of nominations, it moves to the next phase. Otherwise, it retries the nomination process.

2. **Phase 2: Confirmation**
    - The nominated node sends a "confirm leadership" message to all other nodes, indicating its willingness to become the leader.
    - Each node waits for confirmations from the nominated node and other nodes.
    - If a node receives confirmations from a majority of nodes, it acknowledges the nominated node as the leader.
3. **Achieving Consensus**
    - Once a node receives acknowledgments from a majority of nodes, it becomes the leader.
    - All nodes recognize the leader based on the acknowledgments received and start interacting with the leader for making decisions.

In this example, the consensus is achieved by ensuring that a leader is chosen only when a majority of nodes agree on the same candidate. The two-phase protocol helps prevent the possibility of having multiple leaders or a lack of agreement on the leader.

The Ricart-Agrawala algorithm is a distributed mutual exclusion algorithm used to manage access to critical sections among multiple processes in a distributed system. This algorithm ensures that only one process can execute the critical section at a time while preserving safety and liveness properties. It uses a token-based approach to achieve mutual exclusion.

Here's how the Ricart-Agrawala algorithm works:

1. **Initialization**:
    - Each process maintains a local logical clock that represents its progression in time. Clock values are unique and are used to determine the order of requests.
    - Each process also maintains two flags: a request flag and a reply flag, both initially set to false.
2. **Requesting Critical Section**:

- When a process wants to enter the critical section, it first sets its request flag to true.
- It then sends a request message to all other processes in the system. The message contains its logical clock value.

3. **Handling Request Messages**:
    - Upon receiving a request message, a process compares the logical clock value in the message with its own clock.
    - If the receiving process is not in its critical section and the request flag is false, it sends a reply message to the requesting process.
    - If the receiving process is already in its critical section or has a pending request of its own, it compares the logical clocks to decide which process should reply first.
4. **Receiving Replies**:
    - When a process receives a reply message, it updates its reply flag to true.
5. **Entering Critical Section**:
    - A process can enter the critical section only if it has received replies from all other processes and its request flag is true.
    - Once inside the critical section, the process performs its critical section code.
6. **Exiting Critical Section**:
    - After exiting the critical section, the process resets its request flag to false and sends a release message to all other processes.
7. **Handling Release Messages**:
    - Upon receiving a release message, a process clears its reply flag for the corresponding process.

By following this protocol, the Ricart-Agrawala algorithm guarantees that processes can enter the critical section in a way that ensures mutual exclusion and respects the order of request messages. However, the algorithm has some limitations, such as potential deadlocks and performance issues in a highly loaded system.

Q6. List the challenges of replication in DS. Explain how passive replication model supports in fault tolerance. How it is differ than active replication mode.

Solution:

Challenges of replication in distributed systems:

1. **Consistency and Synchronization**: Ensuring uniform data across replicas in the presence of concurrent updates.
2. **Data Integrity**: Preventing data corruption and maintaining accurate information in all replicas.
3. **Conflict Resolution**: Managing conflicts arising from conflicting updates in different replicas.
4. **Network Constraints**: Dealing with latency and bandwidth limitations when replicating data.
5. **Consensus and Coordination**: Achieving agreement among replicas and coordinating updates.
6. **Performance Overhead**: Handling additional communication and coordination costs due to replication.
7. **Replica Placement**: Choosing optimal locations for replicas to balance performance and fault tolerance.
8. **Failure Handling**: Detecting and recovering from replica failures while maintaining consistency.

9. **Maintenance and Updates**: Updating and maintaining replicas without causing disruptions.
10. **Economic Factors**: Balancing costs and benefits of replication in terms of resources and storage.
11. **Security and Access Control**: Ensuring secure access to replicated data across various replicas.

Passive Replication Model: In passive replication, multiple identical copies of a resource are maintained across different nodes. However, only one of these replicas, the primary replica, actively handles client requests and updates. The other replicas, called backup replicas, remain passive and do not participate in processing client requests.

Support for Fault Tolerance: The passive replication model provides fault tolerance through the following steps:

1. **Primary Replica Operation**: The primary replica actively serves client requests, processing updates and responding to queries.
2. **Backup Replicas**: Backup replicas maintain a copy of the data or service but do not process client requests directly.
3. **Fault Detection**: The system monitors the primary replica for failures. If the primary replica fails or becomes unresponsive, the system detects the failure.
4. **Failover**: When a primary replica failure is detected, one of the backup replicas is promoted to the role of the new primary replica. This process is known as failover.
5. **Client Redirection**: After failover, the system redirects client requests to the new primary replica.
6. **Recovery**: Once the failed primary replica is restored, it becomes a backup replica, and the data is synchronized between the restored replica and the current primary replica.

Differences from Active Replication: Passive replication differs from active replication in the following ways:

1. **Resource Usage**: In passive replication, backup replicas are mostly idle and do not process requests. This saves computing resources compared to active replication, where all replicas process incoming requests concurrently.
2. **Latency**: Passive replication can introduce higher latency during failover, as clients need to be redirected to a backup replica, which then becomes the new primary.
3. **Consistency**: Active replication maintains strong consistency, as all replicas process requests simultaneously. Passive replication usually maintains weaker consistency, as only the primary replica processes requests.
4. **Complexity**: Passive replication is generally simpler to implement since backup replicas remain passive most of the time. Active replication requires more complex mechanisms to ensure synchronization and consistency among active replicas.
5. **Failover Trigger**: In passive replication, failover is triggered by detecting the failure of the primary replica. In active replication, failover can be triggered by client load balancing algorithms or performance degradation.

Q7. Define flat and nested transactions. Discuss the approach of optimistic concurrency control in distributed transactions.

Flat Transactions:

Flat transactions, also known as single-level transactions, are a type of transaction model used in databases and distributed systems. In a flat transaction model, a transaction is a unit of work that consists of a sequence of operations or actions that are executed as a single, indivisible unit. Once a transaction begins, it must complete all its operations successfully to be considered as committed. If any operation within the transaction fails, the entire transaction is rolled back, and the system returns to its previous consistent state.

Flat transactions are simpler to implement and understand, making them suitable for many scenarios. However, they might not be suitable for complex systems where finer-grained control over transactions is required.

Nested Transactions:

Nested transactions, also known as multi-level transactions, are a more advanced transaction model that allows a transaction to be subdivided into smaller, nested transactions. Each nested transaction can have its own set of operations and commits or rollbacks independently of its parent transaction. Nested transactions provide more flexibility and finer-grained control over transaction management.

In a nested transaction model:

- A parent transaction can initiate multiple child transactions.
- Child transactions can commit or roll back without affecting the state of the parent transaction.
- A child transaction's changes are visible only to its parent transaction until the parent commits.
- If the parent transaction rolls back, all changes made by the child transactions are also rolled back.

Nested transactions are particularly useful in scenarios where a complex task can be divided into smaller subtasks, each requiring its own transactional behavior. They provide a way to manage the granularity of transactions more precisely, improving modularity and error recovery.

Optimistic Concurrency Control is a strategy used in distributed transactions to handle concurrent access to shared resources while minimizing contention and enhancing performance. This approach is based on the assumption that conflicts among transactions are infrequent, and it allows transactions to proceed without locking resources immediately.

Here's how the approach of optimistic concurrency control works in distributed transactions:

1. **Read Phase**:
    - When a transaction wants to access a resource, it reads the resource's current state without acquiring any locks.
    - The transaction records a "snapshot" of the resource's state, capturing the data it has read.
2. **Execution Phase**:
    - The transaction proceeds with its operations and manipulates the local copies of resources as needed.
    - During this phase, the transaction does not lock any resources or prevent other transactions from accessing them.
3. **Validation Phase**:
    - Before committing, the transaction validates whether its local changes conflict with updates made by other transactions.
    - This involves comparing the current state of the resources with the snapshots taken during the read phase.
4. **Conflict Detection and Resolution**:
    - If conflicts are detected during the validation phase (e.g., another transaction modified a resource the current transaction also modified), the system needs to resolve them.
    - Different strategies can be employed, such as rolling back one of the transactions or merging conflicting changes.
5. **Commit Phase**:

- If no conflicts are detected during validation, the transaction is allowed to commit its changes.
- Otherwise, the transaction might need to be rolled back, or appropriate corrective measures are taken to resolve conflicts.

Q8. Explain snapshot algorithm used for backward recovery in distributed system. Explain three phase commit protocol with state diagram.

Solution:

Snapshot Algorithm for Backward Recovery:

The snapshot algorithm is used for backward recovery in distributed systems to capture a consistent global snapshot of the system's state. This snapshot is useful for restoring the system to a consistent state after a failure has occurred. The algorithm involves taking a snapshot of the local state of each process and the messages in transit. These snapshots are then used to reconstruct the state of the system just before the failure occurred.

The algorithm proceeds as follows:

1. **Marker Message**: A special marker message is injected into the system. The marker message acts as a trigger to initiate the snapshot process. When a process receives the marker message for the first time, it records its local state and the state of any incoming messages.
2. **Recording Local State**: Each process records its local state, which includes its local variables, data structures, and any messages it has sent but not yet acknowledged.
3. **Recording Incoming Messages**: When a process receives the marker message for the first time, it starts recording incoming messages and their states. This ensures that the state of messages in transit is captured.

4. **Marker Propagation**: Upon receiving the marker message, a process continues to propagate the marker message to its neighbors. This triggers the snapshot process in those processes as well.
5. **Termination Condition**: The snapshot process terminates when all processes have received and propagated the marker message.

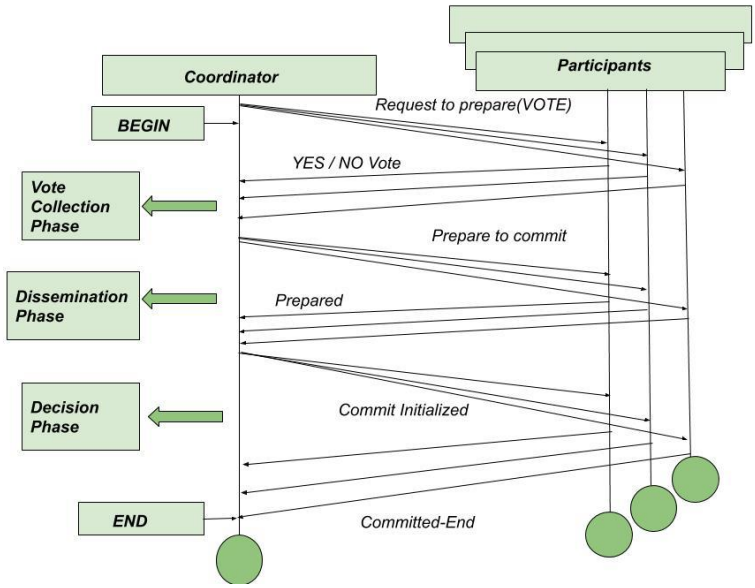
Once the snapshot algorithm has completed, the recorded states can be used to reconstruct the system's state just before the failure occurred. This information is crucial for backward recovery and ensuring the system returns to a consistent state after a failure.

Three-Phase Commit Protocol

Three-Phase Commit (3PC) Protocol is an extension of the Two-Phase Commit (2PC) Protocol that avoids blocking problem under certain assumptions. In particular, it is assumed that no network partition occurs, and not more than k sites fail, where we assume 'k' is predetermined number. With the mentioned assumptions, protocol avoids blocking by introducing an extra third phase where multiple sites are involved in the decision to commit.

Instead of directly noting the commit decision in its persistent storage, the ***coordinator*** first ensures that at least 'k' other sites know that it intended to commit transaction.

In a situation where coordinator fails, remaining sites are bound to first select new coordinator. This new coordinator checks status of the protocol from the remaining sites. If the coordinator had decided to commit, at least one of other 'k' sites that it informed will be up and will ensure that commit decision is respected. The new coordinator restarts third phase of protocol if any of rest sites knew that old coordinator intended to commit transaction. Otherwise, new coordinator aborts the transaction.



Q9. Write Short notes on:

a) Process and Threads in a Distributed System:

- **Process:** In a distributed system, a process is an independent program with its own memory space and resources. It represents an execution unit that can run concurrently with other processes. Processes in a distributed system can communicate and cooperate through inter-process communication (IPC) mechanisms like message passing or remote procedure calls.
- **Thread:** Threads are lighter-weight units of execution within a process. Multiple threads can exist within a single process and share the same memory space. Threads within a process can run concurrently, enabling better utilization of resources. Threads can communicate more efficiently compared to processes because they share the same memory context.
- **Advantages of Threads in Distributed Systems:**
    - Threads can provide better concurrency and parallelism.

- Threads within the same process can communicate more efficiently than separate processes.
- Threads can be managed with less overhead compared to full processes.
- Threads are particularly useful for tasks that require frequent communication and coordination.

b) Monolithic and Microkernel:

Monolithic Kernel: In a monolithic kernel architecture, the entire operating system runs as a single, large, and interconnected program. It provides all the necessary services, including device drivers, memory management, file systems, and networking, within the kernel. Monolithic kernels are efficient in terms of performance but can be complex and less modular.

Microkernel: In a microkernel architecture, the core operating system functions are kept minimal, and additional services are implemented as separate user-level processes or servers. Microkernels aim for simplicity, modularity, and easy maintenance. Communication between different components is usually done through message passing. This approach makes the system more reliable and extensible but can introduce some performance overhead due to message passing.

c) MACH:

MACH is a microkernel-based operating system that was developed at Carnegie Mellon University in the 1980s. It is known for its modularity and innovative design principles. Key features of MACH include:

- **Microkernel Architecture:** MACH's microkernel provides only essential services, such as memory management, inter-process communication, and scheduling. Additional services are implemented as separate user-level processes.

- **Message Passing:** MACH heavily relies on message passing for communication between different components. This design choice emphasizes modularity and isolation.
- **Virtual Memory:** MACH introduced advanced virtual memory management techniques, including support for demand paging and memory protection.
- **Distributed Environment:** MACH was designed to support distributed computing and had features like remote procedure calls (RPCs) to enable communication between processes running on different nodes.
- **Influence:** MACH's design principles and concepts have influenced subsequent operating systems and research projects, including Apple's macOS (which was initially based on MACH) and various research endeavors in distributed systems and operating systems.

2080 Baisakh (BACK)

Q1: Explain the design goals and challenges in distributed system. What is fundamental model?

Solution:

The design of distributed systems is guided by several key goals and challenges:

1. Scalability: Distributed systems should be able to handle an increasing number of users, tasks, or data without sacrificing performance. Scalability can be achieved through horizontal scaling (adding more machines) or vertical scaling (upgrading existing machines).

2. Reliability: Distributed systems should continue to function correctly even in the presence of hardware failures, software errors, or network problems. This goal is often achieved through redundancy, fault tolerance mechanisms, and error detection and recovery techniques.

3. Resource Sharing: Resource sharing in distributed systems involves efficient utilization of hardware, software, and data across multiple nodes to provide better performance, availability, and cost-effectiveness. This goal addresses the challenge of managing and distributing resources effectively in a distributed environment.

4. Openness: Openness in distributed systems refers to the principle of allowing easy integration, communication, and interaction between different systems, regardless of their origins, architectures, or technologies. An open distributed system is designed to be accessible and adaptable, enabling seamless collaboration between diverse components.

5. Transparency: Distributed systems should appear as a single, coherent system to users and applications, even though they are composed of multiple interconnected components.

6. Consistency: Maintaining data consistency across distributed nodes is a significant challenge. Ensuring that all nodes see the same data or reach a consensus on shared data is a complex issue, especially in the presence of network delays and failures.

7. Transparency: Distributed systems should hide the complexities of their underlying architecture from users and applications. This transparency can involve hiding network

details, providing location-independent access to resources, and ensuring a seamless user experience.

8. Security: Distributed systems must protect data and resources from unauthorized access, attacks, and breaches. Security mechanisms such as authentication, encryption, and access control are essential to ensure the system's integrity.

Fundamental Model

The fundamental model in distributed systems provides an abstraction that simplifies the understanding and analysis of complex distributed systems. It helps researchers and designers to reason about the behavior, interactions, and limitations of distributed systems.

The fundamental model defines the basic building blocks and assumptions of distributed systems. These can include concepts such as processes (units of execution), communication channels (through which processes exchange messages), and synchronization mechanisms.

Q2: Explain the role of middleware in Distributed System. Explain the operation of dynamic RMI.

Solution:

Middleware is a software layer that sits between the operating system and application software, providing a set of services and abstractions that facilitate communication, coordination, and interaction between distributed components in a networked environment. The primary roles of middleware in distributed systems include:

1. **Communication Abstraction:** Middleware hides the intricacies of network communication by offering high-level communication abstractions, such as remote procedure calls (RPC), message queues, and publish-subscribe mechanisms. This enables developers to focus on application logic without dealing with low-level network protocols.

2. **Location Transparency:** Middleware provides location-independent access to resources and services. This means that applications can access remote resources as if they were local, regardless of their physical location in the network.
3. **Interoperability:** Middleware enables different components, often developed using various programming languages and technologies, to communicate and collaborate seamlessly. This promotes system interoperability by providing a common communication framework.
4. **Scalability:** Middleware often incorporates load balancing, replication, and distributed caching mechanisms that help distribute the workload across multiple nodes and enhance system scalability.
5. **Concurrency and Synchronization:** Middleware may offer synchronization and concurrency control mechanisms that allow distributed processes to coordinate and manage concurrent access to shared resources.
6. **Security:** Middleware can provide authentication, encryption, and access control mechanisms to ensure secure communication and data protection in a distributed environment.
7. **Fault Tolerance:** Many middleware systems include features for handling failures, such as automatic failover, data replication, and error detection and recovery.
8. **Resource Management:** Middleware can assist in managing resources like distributed databases, remote services, and computational nodes by providing tools for resource discovery, allocation, and monitoring.
9. **Message Queues and Publish-Subscribe:** Middleware often includes message-oriented middleware (MOM) that facilitates asynchronous communication using message queues or the publish-subscribe pattern. This is useful for decoupling components and managing asynchronous workflows.
10. **Distributed Transactions:** Middleware can provide support for distributed transactions, ensuring that multiple operations across distributed resources are either completed successfully or fully rolled back in case of failure.

11. **Service Composition:** Middleware can aid in composing complex applications from distributed services, allowing developers to build applications by combining existing services.

Operation of Dynamic RMI (Remote Method Invocation):

Dynamic RMI is a mechanism that allows objects in different Java Virtual Machines (JVMs) to invoke methods on each other as if they were local objects. It's a key feature for building distributed applications in Java. Here's how dynamic RMI works:

1. **Stub Generation:** When a client wants to access a remote object's methods, the Java RMI system generates a stub for that remote object. The stub is a client-side proxy that mimics the interface of the remote object.
2. **Client Invocation:** The client-side stub provides the same methods as the remote object's interface. When a client invokes a method on the stub, it appears as if the client is calling a local method.
3. **RMI Registry:** The RMI registry acts as a service locator that maintains a registry of remote objects and their corresponding stubs. The registry helps clients locate the stubs of remote objects.
4. **Server Setup:** On the server side, the remote object's implementation registers itself with the RMI registry using a unique name.
5. **RMI Invocation:** When the client invokes a method on the stub, the RMI system is responsible for intercepting the method call, serializing the method parameters, and sending them to the remote JVM.
6. **Server-side Processing:** The remote JVM receives the method invocation request, deserializes the parameters, and routes the call to the corresponding remote object's implementation.
7. **Return Value:** The remote object's implementation processes the method call and returns the result to the client JVM.

8. **Serialization:** All parameters and return values must be serializable, as they need to be transmitted between JVMs. If objects are not serializable, they need to be converted into a serializable form.
9. **Dynamic Class Loading:** Dynamic RMI supports class loading on demand. If the client doesn't have the class definitions for the remote object's interface or implementation, the necessary classes can be loaded from the server JVM dynamically.

Q3: Compare stateful and stateless services. Explain the operation and architecture of any one modern distributed file system.

Solution:

Aspect	Stateful Services	Stateless Services
State Management	Maintain client-specific state and context	Do not store client-specific state
Scalability	More complex scaling due to state	Highly scalable due to independent state
Fault Tolerance	Prone to failures due to state replication	More fault-tolerant due to statelessness
Data Consistency	Challenging to maintain consistent state	Easier to achieve data consistency

One modern distributed file system that I can explain is the **Hadoop Distributed File System (HDFS)**, which is commonly used in big data processing. HDFS is designed to store and manage large amounts of data across a cluster of commodity hardware.

Operation of Hadoop Distributed File System (HDFS):

1. **Data Splitting and Distribution:**

- Large files are split into fixed-size blocks (default 128 MB or 256 MB).
- These blocks are then distributed across the nodes in the HDFS cluster.

2. **Master-Slave Architecture:**
    - HDFS follows a master-slave architecture. There are two main components: NameNode (master) and DataNodes (slaves).
    - NameNode stores metadata about files and directories, including block locations and namespace hierarchy.
    - DataNodes store the actual data blocks.
3. **Data Replication:**
    - HDFS replicates data blocks for fault tolerance. By default, each block is replicated three times across different DataNodes.
    - Replicas are placed on different racks for data locality and improved fault tolerance.
4. **Write Operation:**
    - When a client wants to write a file to HDFS, it communicates with the NameNode to create metadata entries for the new file.
    - Data is written in blocks to available DataNodes. As each block is written, it is replicated to other nodes.
5. **Read Operation:**
    - When a client wants to read a file, it contacts the NameNode to get block locations.
    - The client then directly communicates with DataNodes hosting the required blocks to retrieve the data.
6. **Data Locality:**

- HDFS emphasizes data locality, meaning processing is done near the data to minimize network overhead.
- Computation tasks are scheduled on nodes where data is already present to reduce data transfer time.

Architecture of Hadoop Distributed File System (HDFS):

1. **NameNode:**
    - Single point of failure (SPOF) in the HDFS architecture.
    - Manages the metadata, maintains the file system namespace, and tracks the blocks' locations.
    - Keeps track of DataNodes and their health status.
    - Serves read and write requests from clients and provides block location information.
2. **DataNodes:**
    - Responsible for storing the actual data blocks.
    - Periodically send heartbeat signals to the NameNode to report their status and availability.
    - DataNodes respond to read and write requests from clients, as well as block replication commands from the NameNode.
3. **Secondary NameNode:**
    - Misleading name; it does not act as a standby NameNode.
    - Periodically checkpoints the metadata from the NameNode and compacts the transaction logs to prevent them from becoming too large.
4. **Client:**

- Interacts with the HDFS for reading and writing data.
- Communicates with the NameNode for metadata information and directly with DataNodes for data retrieval or storage.

Q4: What is physical and logical clock? Explain the importance of VECTOR clock with its implementation rules and example.

Solution:

Physical Clock: A physical clock is a hardware-based timekeeping device that uses oscillations of a quartz crystal to measure time intervals accurately.

Logical Clock: A logical clock is a concept used in distributed systems to track the order of events or actions, even in the absence of synchronized physical clocks. It helps establish a partial ordering of events based on causal relationships.

Importance of Vector Clock:

Vector clocks are a crucial concept in distributed systems for tracking the causal ordering of events across multiple processes or nodes. They help maintain a consistent view of the order in which events occur, which is essential for tasks like distributed data consistency, conflict resolution, and detecting concurrent updates.

Implementation Rules for Vector Clocks:

1. **Initialization:**
    - Each process/node initializes its vector clock with zeros.
2. **Local Event:**
    - When a process/node initiates an event, it increments its own entry in the vector clock.
3. **Message Sending:**

- Before sending a message, the sender increments its own entry in the vector clock.
- The sender includes its vector clock in the message.

4. **Message Receiving:**
    - Upon receiving a message, the receiver compares its own vector clock with the sender's vector clock component-wise.
    - For each component, it takes the maximum value between the sender's and its own value and increments it by one.

Example of Vector Clocks:

Consider a distributed system with three processes: A, B, and C. Each process has a vector clock initialized to [0, 0, 0].

1. Process A initiates an event: Its vector clock becomes [1, 0, 0].
2. Process B initiates an event: Its vector clock becomes [0, 1, 0].
3. Process A sends a message to B: The message includes [1, 0, 0].
4. Process B receives the message and updates its vector clock to [1, 1, 0].
5. Process C initiates an event: Its vector clock becomes [0, 0, 1].
6. Process B sends a message to C: The message includes [1, 1, 0].
7. Process C receives the message and updates its vector clock to [1, 1, 1].

At any point, the vector clock for each process represents the number of events that have happened in that process and other processes up to that point. Comparing vector clocks between processes allows determining the causal ordering of events and identifying potential conflicts or concurrent updates.

Q5: Compare non token and token based mutual exclusion. Why election is important in DS? Explain the Ring Based election algorithm with rules and example.

Solution:

Aspect	Non-Token-Based Mutual Exclusion	Token-Based Mutual Exclusion
Mechanism	Uses flags, messages, or variables for coordination	Uses a circulating token to control access
Complexity	Can be complex with synchronization challenges	Generally simpler and easier to implement
Starvation Prevention	Prone to starvation, as some processes may never get access	Prevents starvation, ensures equitable access
Scalability	Complexity increases with the number of processes	Can be more scalable due to inherent fairness

Elections are crucial in distributed systems for selecting leaders, ensuring fault tolerance by replacing failed nodes, maintaining load balancing, and enabling coordinated decision-making. They establish a single point of control, prevent split-brain scenarios, and adapt to changing network conditions, enhancing stability and performance in distributed environments.

Ring Based election algorithm

The Ring-Based Election Algorithm is a simple approach to electing a leader in a distributed system. It's based on the concept of nodes forming a logical ring structure, where each node communicates with its neighbors to determine the leader. Here's how the algorithm works:

Algorithm Rules:

1. Each node in the distributed system has a unique identifier.

2. Nodes are organized in a logical ring structure.
3. A node initiates an election if it suspects that the leader is unavailable (failed).

Algorithm Steps:

1. **Election Initiation:**
    - A node that wishes to initiate an election starts by sending an "election" message containing its own identifier.
2. **Message Propagation:**
    - Upon receiving the "election" message, each node compares the received identifier with its own.
    - If the received identifier is greater, the node forwards the message to its neighbor in the ring. If it's smaller, the node discards the message.
    - If the received identifier is equal to its own, the node becomes the leader and sends a "leader" message to announce its victory.
3. **Leader Announcement:**
    - The "leader" message travels through the ring, and each node updates its state to acknowledge the new leader.

Example:

Let's consider a scenario where nodes A, B, C, and D are arranged in a ring. Node B initiates an election because it suspects that the leader (Node A) has failed.

1. Node B sends an "election" message containing its identifier.
2. Node C receives the message and compares identifiers: B < C, so C forwards the message to Node D.
3. Node D receives the message and compares identifiers: B < D, so D forwards the message to Node A.

4. Node A receives the message and compares identifiers: B < A, so A discards the message.
5. Nodes B, C, and D receive no "leader" message from Node A, so they all assume Node B is the new leader.

In this example, Node B wins the election as it has the highest identifier among the participating nodes. The "leader" message eventually circulates the ring, and each node acknowledges Node B as the leader.

Q6: What are the different consistency models applicable in a distributed system? How does a primary-backup model work?

Solution:

Consistency models define the ordering and visibility of updates to shared data in a distributed system. They play a crucial role in maintaining data correctness and ensuring that operations on shared data occur in a predictable manner. Here are some different consistency models applicable in a distributed system:

1. **Strict Consistency:**
    - All operations on shared data appear as if they are executed instantaneously and in the same order across all nodes.
    - Provides the strongest guarantee of consistency but can lead to high latency and limited scalability due to synchronization requirements.
2. **Sequential Consistency:**
    - All operations from a single process are seen by all other processes in the same order.
    - Operations from different processes might be interleaved in an unpredictable manner.

- Provides a compromise between strict consistency and performance.

3. **Causal Consistency:**
    - Ensures that causally related operations are seen by all processes in the same order.
    - Doesn't require a strict order for unrelated operations, allowing greater concurrency.
    - Provides a balance between ordering guarantees and scalability.
4. **Eventual Consistency:**
    - Guarantees that if no updates are made to a piece of data, eventually all replicas of that data will converge to the same state.
    - Allows temporary inconsistencies and focuses on availability and partition tolerance.
    - Suited for systems that prioritize availability and tolerate short-lived inconsistencies.
5. **Read-your-Writes Consistency:**
    - Guarantees that a process will see its own writes in subsequent reads.
    - Provides a strong guarantee for maintaining the order of reads and writes made by the same process.
6. **Monotonic Read Consistency:**
    - If a process reads a value, it will not read a previous version of that value in the future.
    - Ensures that subsequent reads will reflect at least the same value or a more recent value.
7. **Monotonic Write Consistency:**

- If a process writes a value, all future reads will reflect that value or a more recent value.
- Ensures that writes to shared data are reflected in subsequent reads.

8. **PRAM (Parallel Random-Access Machine) Consistency:**
    - Provides a theoretical model for consistency in parallel and distributed systems.
    - Defines a strict order of operations, but often not practical to achieve in real-world distributed systems due to synchronization and performance overhead.

The primary-backup model is a replication strategy used in distributed systems to enhance fault tolerance and ensure high availability of services.

1. **Primary Node:**
    - The primary node is responsible for handling client requests and performing the desired computations or services.
    - It maintains the most up-to-date state and responds to clients in real-time.
2. **Backup Nodes:**
    - Backup nodes replicate the state of the primary node. They keep copies of the data and state information, ensuring redundancy.
    - Backup nodes remain passive and do not process client requests directly.
3. **Heartbeat Mechanism:**
    - To detect the primary node's failure, the backup nodes use a heartbeat mechanism. They periodically send heartbeat signals to the primary node.
    - If a backup node detects that the primary node has stopped responding, it assumes that the primary has failed.
4. **Promotion of a Backup:**

- When a backup node detects the primary node's failure, it initiates a process to promote itself to become the new primary.
- The backup node updates its state to reflect the most recent state of the failed primary node.

5. **Client Redirection:**
    - After the promotion, the new primary node starts accepting client requests and responding to them.
    - Clients need to be redirected to the new primary node to ensure continuous service availability.
6. **Synchronization of Backups:**
    - Backup nodes need to remain synchronized with the primary node's state to ensure consistency when one of them is promoted.
    - This synchronization can be achieved through mechanisms like periodic state updates or using techniques like quorum-based approaches.
7. **Failure Recovery:**
    - Once the failed primary node is restored, it becomes a backup node and synchronizes its state with the new primary node.
    - The system returns to its normal state with the new primary node handling client requests.

Q7: Briefly explain different methods for concurrency control. What are the drawbacks of 2PL and how does strict 2PL, overcomes those?

Solution:

Methods for concurrency control

1. **Locking:**
    - Locking involves acquiring locks on resources (e.g., data items) to prevent other processes from accessing or modifying them concurrently.
    - Two common types of locks are shared locks (read locks) and exclusive locks (write locks).
    - Ensures data consistency by allowing only one process at a time to modify shared data.
2. **Optimistic Concurrency Control:**
    - Processes are allowed to perform their operations without obtaining locks initially.
    - When a process wants to commit its changes, the system checks if any conflicts occurred during its operation.
    - If no conflicts are detected, the changes are committed; otherwise, the process's changes are rolled back, and it needs to retry.
3. **Timestamp Ordering:**
    - Each transaction is assigned a unique timestamp based on its start time.
    - Transactions are executed in timestamp order to maintain consistency.
    - Older transactions take precedence over newer ones to prevent anomalies.
4. **Two-Phase Locking (2PL):**
    - Transactions follow a strict protocol where they first acquire locks and then release them in two phases: the growing phase and the shrinking phase.
    - Guarantees serializability by ensuring that transactions do not release any locks until they have acquired all they need.
5. **Multiversion Concurrency Control (MVCC):**

- Creates multiple versions of data items to support different transactions simultaneously.
- Each transaction reads from a consistent snapshot, preventing read and write conflicts.
- Widely used in databases to improve read concurrency.

Drawbacks of Two-Phase Locking (2PL):

1. **Deadlocks:** 2PL can lead to deadlocks if processes hold locks but are waiting for other locks to be released.
2. **Blocking:** Transactions might need to wait for a long time, causing blocking and reduced system throughput.
3. **Conservative:** Some resources might be locked for longer than needed, limiting concurrency.

Strict Two-Phase Locking (Strict 2PL) Overcomes:

Strict 2PL overcomes these drawbacks by requiring transactions to hold all their locks until they reach the commit point, ensuring serializability. This eliminates deadlocks and reduces blocking, enhancing system concurrency.

Q8: Define faults, error and failures. How reliable client server communication can be achieved in DS?

Solution:

1. **Fault:**
   - A fault is a defect or imperfection in the hardware or software of a system that can lead to errors or failures.
   - It can be caused by design flaws, manufacturing defects, software bugs, or external factors.
2. **Error:**
   - An error is a deviation from the expected behavior or correct result in a computation, process, or system operation.

- Errors occur when a system encounters a fault during its execution and may lead to undesired outcomes.

3. **Failure:**
    - A failure is a situation where a system or component ceases to provide the expected service or produces incorrect results.
    - Failures occur due to errors that propagate to a point where the system's functionality is compromised.

Reliable client-server communication in distributed systems involves several techniques to ensure that data is transmitted accurately and consistently. It includes:

1. **Error Detection and Correction:**
    - Techniques like checksums, cyclic redundancy checks (CRC), and parity bits are used to detect and correct errors in transmitted data.
    - Redundant information is added to the data, allowing the receiver to verify if the data has been corrupted during transmission.
2. **Acknowledgment and Timeout:**
    - After sending a message, the sender waits for an acknowledgment from the receiver.
    - If the sender doesn't receive an acknowledgment within a certain time (timeout), it assumes the message was lost or corrupted and retransmits it.
3. **Automatic Repeat reQuest (ARQ):**
    - ARQ protocols (e.g., Stop-and-Wait, Go-Back-N, Selective Repeat) manage retransmission of lost or corrupted packets.
    - The receiver acknowledges correctly received packets, and the sender retransmits only the missing ones.
4. **Sequence Numbers:**

- Sequence numbers are assigned to messages to ensure their correct order at the receiver.
- This prevents out-of-order delivery and helps maintain data consistency.

5. **Flow Control:**
    - Flow control mechanisms regulate the rate of data transmission to avoid overwhelming the receiver or network.
    - Techniques like sliding window protocols ensure efficient data transfer while preventing congestion.
6. **Reliable Transport Protocols:**
    - Transport protocols like TCP (Transmission Control Protocol) provide reliable communication by ensuring data integrity, sequence, and flow control.
    - They handle acknowledgment, retransmission, and error recovery, offering a robust solution for reliable communication.

Q9: Write short notes on: (Any Three)

a) MACH

b) Strongly Consistent CUT in DS

c) Multithreading in DS

c) CORBA services

Answer:

a) MACH: MACH is a microkernel-based operating system that emphasizes modularity and extensibility. It separates the core functions of an operating system into small, well-defined

modules, or microkernels, while delegating other functionalities to user-level processes. MACH's design enables better flexibility, scalability, and security by reducing the kernel's complexity. One of its notable implementations is the GNU Hurd, which uses MACH's microkernel approach alongside other components.

b) Strongly Consistent CUT in DS: A Cut (Consistent Global State) in a distributed system is a snapshot of the system at a particular moment. A Strongly Consistent Cut is a cut that includes all events up to a certain time and ensures that no event in the cut depends on events outside the cut. This guarantees a clear view of the global state without causality violations. Strongly Consistent Cuts are essential for accurate analysis, debugging, and visualization of distributed systems.

c) Multithreading in DS: Multithreading involves executing multiple threads within a single process to improve concurrency and responsiveness. In distributed systems, multithreading is crucial for handling multiple tasks or requests simultaneously. However, managing shared resources, synchronization, and avoiding race conditions become more complex in distributed multithreaded environments. Proper synchronization mechanisms like locks, semaphores, and barriers are used to ensure safe and efficient execution of threads in a distributed context.

d) CORBA Services: CORBA (Common Object Request Broker Architecture) is a middleware technology that facilitates communication and interaction between distributed objects across different programming languages and platforms. CORBA services provide a standardized way to access common functionalities in distributed systems. Some of the key CORBA services include:

- **Naming Service:** Provides a directory service for locating objects by their names.
- **Event Service:** Enables asynchronous communication and event notifications between objects.
- **Transaction Service:** Offers support for distributed transactions to ensure data consistency.
- **Query Service:** Allows querying distributed databases and repositories.

- **Concurrency Control Service:** Ensures proper synchronization and access to shared resources.
- **Security Service:** Provides authentication, authorization, and data protection mechanisms.

CORBA services abstract the complexity of distributed systems, allowing developers to focus on building and connecting distributed components without worrying about the underlying network and platform differences.

THANK YOU