2071 Shawan-New Back

1) **Discuss the properties of Distributed System(DS). How interaction model addresses the relevant issues in DS?**

Ans: A distributed system is a collection of independent computers at networked locations such that they communicate and interact only through message passing that is viewed as a single coherent system by its users. Example: Internet, Mobile Network, DNS:Distributed Database System.

The properties of Distributed System are:

- Concurrency: In a distributed system, multiple nodes can execute their tasks simultaneously, resulting in concurrent operations. This property enables efficient resource utilization and better performance but also introduces the need for synchronization mechanisms to manage shared resources and prevent data inconsistency.
- No Global Clock: Distributed systems lack a global clock due to differences in clock speeds and network delays between nodes. This makes it challenging to establish a consistent global time reference, which impacts synchronization and coordination of events across the system.
- Partial Failure: Since distributed systems consist of multiple nodes, the likelihood of individual node failures is higher than in centralized systems. Therefore, distributed systems need to be designed with fault tolerance in mind, ensuring that the system can continue to operate even when some nodes fail.
- Communication: Communication between nodes in a distributed system can involve various communication channels, such as message passing or remote procedure calls. Network communication introduces latency, potential message loss, and order-of-delivery issues, which must be addressed to ensure reliable communication.
- Heterogeneity: Nodes in a distributed system can have different hardware, operating systems, and software components. Managing heterogeneity is crucial for interoperability and achieving system-wide goals.
- Scalability: Distributed systems should be able to scale horizontally (adding more nodes) or vertically (upgrading individual nodes) to accommodate changing workloads and user demands. Scalability can be challenging due to issues like load balancing and maintaining consistent performance.

Interaction models are patterns or protocols that guide how nodes in a distributed system communicate and coordinate with each other. These models help address the relevant challenges in distributed systems:

- Remote Procedure Call (RPC): This model allows a node to invoke procedures or methods on remote nodes as if they were local, abstracting the complexities of network communication. RPC helps with communication and coordination while hiding the underlying message passing details.
- Message Passing: In this model, nodes communicate by sending messages to each other. This approach helps address communication challenges but requires careful consideration of issues like message ordering, reliability, and potential failures.
- Publish-Subscribe: This model facilitates communication between nodes by allowing publishers to send messages to specific topics or channels, and subscribers can receive messages from these topics. It supports flexible and loosely coupled communication patterns.
- Distributed Shared Memory (DSM): DSM abstracts the memory of multiple nodes as a single shared memory space, enabling processes to read and write to this space as if it were local. This helps manage concurrency and data sharing but requires mechanisms for consistency and synchronization.
- Coordination Models (e.g., Two-Phase Commit, Paxos, Raft): These models provide protocols for achieving consensus and coordination among distributed nodes. They address the challenges of maintaining consistency and fault tolerance in distributed systems.
- MapReduce and Stream Processing: These models handle large-scale data processing by dividing tasks into smaller subtasks that can be executed in parallel on different nodes. They address scalability and data processing challenges in big data scenarios.

In conclusion, distributed systems possess unique properties such as concurrency, lack of a global clock, partial failure, communication challenges, heterogeneity, and scalability issues. Interaction models provide protocols and patterns to address these challenges by facilitating communication, synchronization, and coordination among distributed nodes. The choice of interaction model depends on the specific requirements and goals of the distributed system being designed.

2) What is importance of IDL in RMI? Write the operation of static RMI.

Ans: IDL stands for Interface Definition Language. It is a standardized language used to define the interface or contract between software components, especially in distributed systems. IDL provides a way to describe the methods, data structures, and communication protocols that components will use to interact with each other.

IDL (Interface Definition Language) plays a crucial role in RMI (Remote Method Invocation) by providing a standardized and language-neutral way to define the remote interfaces that clients and servers use to communicate. The importance of IDL in RMI includes:

- Language Neutrality: IDL allows developers to define remote interfaces in a language-independent manner. This is particularly useful when RMI is used to enable communication between systems implemented in different programming languages.
- Interface Contract: IDL defines the contract between the client and the server. It specifies the methods that can be invoked remotely, their parameters, and return types. Both the client and the server must adhere to this contract, ensuring that communication is consistent and reliable.
- Code Generation: From the IDL definition, tools can generate the necessary code for both the client-side and server-side stubs and skeletons. Stubs and skeletons are intermediary components that handle the marshaling and unmarshaling of parameters and results during remote method calls.
- Consistency: IDL helps ensure that the remote interfaces are consistent between different components of the distributed system. Changes to the interface can be made in a controlled manner by updating the IDL definition and regenerating the code.
- Abstraction: IDL abstracts the complexity of network communication, serialization, and parameter passing. Developers can focus on defining the interface and its methods without needing to delve into the low-level details of distributed communication.
- Interoperability: IDL facilitates interoperability between systems that use different programming languages. As long as the systems can generate compatible stubs and skeletons from the IDL, they can communicate seamlessly.

Static RMI is a variation of the traditional RMI approach that simplifies the setup and configuration process. In a traditional RMI setup, objects are registered with the RMI registry, and clients retrieve these objects to invoke methods remotely. In static

RMI, this registration step is eliminated, and objects are directly available for remote invocation without explicitly registering them. Here's how static RMI operates:

1. Server-side Setup:
- Define the remote interface using IDL or Java interface.
- Implement the remote interface in a Java class.
- Compile the remote interface and implementation class.
- Generate stub and skeleton classes using the rmic tool (for Java RMI) or appropriate tools for other languages.
2. Server Execution:
- Start the RMI registry, if necessary.
- Create an instance of the server implementation class.
- Export the remote object using a specific mechanism provided by the RMI framework (e.g., UnicastRemoteObject.exportObject() in Java RMI).
3. Client-side Setup:
- Define the remote interface in the client's code (using the same IDL or Java interface).
- Compile the client code.
4. Client Execution:
- Obtain a reference to the remote object using a factory method or lookup method provided by the RMI framework.
- Invoke methods on the remote object as if it were a local object.

In static RMI, the server object is automatically made available for remote invocation when it's created, without explicitly registering it with an RMI registry. This approach simplifies the deployment and configuration process but may have limitations in terms of flexibility and dynamic management of remote objects compared to traditional RMI.

3) What are the characteristics of SUN-NFS? Discuss with its architecture

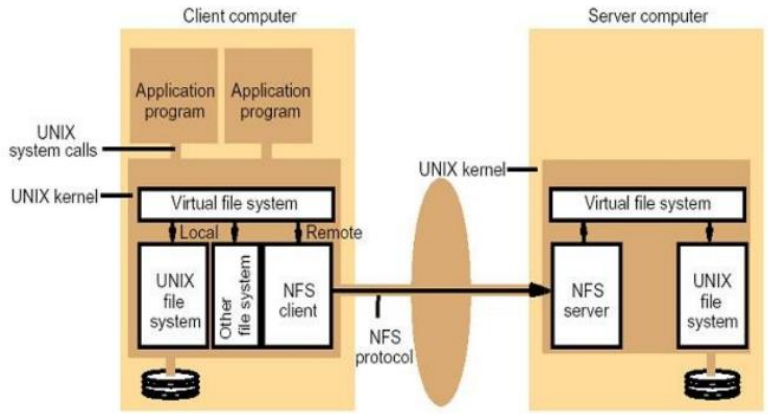
Ans: SUN Network File System (NFS) is a distributed file system protocol developed by Sun Microsystems (now Oracle) that allows users and applications to access files over a network as if they were local. It was designed to provide seamless access to remote files and storage resources in a transparent and efficient manner. Here are the key characteristics of SUN-NFS along with its architecture:

- Transparency: NFS aims to provide transparency to users and applications. Remote files are accessed using standard file system operations, such as open, read, write, and close, without requiring users to be aware of the fact that the files are located on remote servers.
- Location Independence:NFS abstracts the physical location of files, allowing users to access files regardless of where they are stored on the network.
- Caching: To improve performance, NFS supports caching of file data on the client side. This reduces the need to repeatedly fetch data from the server, especially for read-heavy workloads.
- Stateless Server:NFS servers are designed to be stateless, meaning they do not maintain information about client sessions. This simplifies server design and enhances fault tolerance.
- Interoperability: NFS promotes interoperability across different operating systems and platforms. Clients and servers can be based on various operating systems while still communicating effectively.
- File Locking: NFS supports file locking mechanisms for coordinating access to files among multiple clients. This helps prevent conflicts and data corruption when multiple clients attempt to modify the same file simultaneously.
- Security: While early versions of NFS lacked robust security mechanisms, modern versions incorporate features like Kerberos authentication and secure communication protocols to enhance data protection.
- Scalability: NFS is designed to scale across a network of servers and clients, making it suitable for large-scale distributed environments.

The architecture of SUN-NFS involves several components that work together to enable remote file access:

- NFS Client: The client is the system that wants to access remote files. It issues file system calls to the local operating system, which are then translated into RPC (Remote Procedure Call) requests sent to the NFS server.
- NFS Server: The server hosts the remote files that the client wants to access. It listens for RPC requests from clients and responds with the appropriate data or metadata.
- RPC: NFS relies on Remote Procedure Calls for communication between the client and server. The client sends RPC requests to the server, specifying the requested operation (e.g., read, write, open) and the necessary parameters.

- File Handle: Each file on the server is uniquely identified by a file handle, which is an opaque identifier that clients use to reference files in subsequent requests.
- File System Operations: Clients issue file system operations like open, read, write, and close. These operations are translated into RPC requests with the file handle and necessary parameters.
- NFS Protocol Versions: NFS has undergone multiple protocol versions, each with improvements and enhancements. The most well-known versions include NFSv2, NFSv3, and NFSv4, with each version introducing new features and addressing limitations of previous versions.



1. Protocol:

   - It uses SUN RPC mechanism and SUN external data representation (XDR) standard.

   - The protocol is stateless. It enhances crash recovery.

   - Each procedure call must contain all the information necessary to complete the call.

2. Server Side:

   - It provides file handle consisting of:

   a) Filesystem id (identify disk partition)

   b) I-node number (identify file)

   c) Generation number

   - File system id is stored in super block.

- Generation number is stored in I-node.

3. Client Side:

- It provides transparent interface to NFS.

- Mapping between remote file name and remote file address is done at server boot time

through remote mount.

4) What are the common problems of physical clock synchronization algorithm? Write Chandy-Lamport's algorithm for recording global states in Distributed System

Ans: Physical clock synchronization, also known as time synchronization or clock synchronization, refers to the process of aligning the timekeeping of different devices or nodes in a distributed system to a common reference time. In a distributed system, where multiple machines or nodes work together to achieve a common goal, maintaining synchronized clocks is crucial for various tasks such as event ordering, data consistency, scheduling, and coordination. However, there are several challenges and problems associated with these algorithms:

- Clock Drift: Physical clocks are not perfect and can experience drift over time, leading to inaccuracies in synchronization.
- Clock Skew: Clock skew refers to the difference in time rates between different clocks. This can lead to uneven time progress among nodes, even if they are initially synchronized.
- Network Delays: Communication delays between nodes can lead to inaccuracies in clock synchronization. Variability in network delays can make it difficult to maintain precise synchronization.
- Fault Tolerance: Clock synchronization algorithms need to account for nodes that may fail or experience transient issues. Recovery mechanisms must be in place to handle such scenarios.
- Complexity: Some clock synchronization algorithms are complex and require careful tuning and configuration to achieve accurate synchronization.
- Dependency on Infrastructure: Certain algorithms may rely on specific hardware or infrastructure features, making them less adaptable in different environments.

- Scalability: Ensuring accurate synchronization in large-scale distributed systems with numerous nodes can be challenging due to the potential for high communication overhead.
- Security: Some algorithms might be vulnerable to attacks that aim to manipulate the synchronization process for malicious purposes.

The Chandy-Lamport algorithm is used to record consistent global states in distributed systems, which is essential for debugging and analyzing distributed algorithms. The algorithm is based on the principle that a consistent global state can be captured by observing the local states and the communication events (messages) between processes.

Here's a simplified overview of the Chandy-Lamport algorithm:

Assumptions of the algorithm:

- There are finite number of processes in the distributed system and they do not share memory and clocks.
- There are finite number of communication channels and they are unidirectional and FIFO ordered.
- There exists a communication path between any two processes in the system
- On a channel, messages are received in the same order as they are sent.

Algorithm:

- **Marker sending rule for a process *P* :**
    - Process **p** records its own local state
    - For each outgoing channel **C** from process **P**, **P** sends marker along **C** before sending any other messages along **C**.
      (**Note:** Process Q will receive this marker on his incoming channel C1.)
- **Marker receiving rule for a process *Q* :**
    - If process **Q** has not yet recorded its own local state then
        - Record the state of incoming channel C1 as an empty sequence or null.
        - After recording the state of incoming channel C1, process **Q** Follows the marker sending rule
    - If process **Q** has already recorded its state
        - Record the state of incoming channel **C1** as the sequence of messages received along channel C1 after the state of **Q** was

recorded and before **Q** received the marker along C1 from process P.

5) **Measure the performance issue of non-token based Ricart- Agrawal Algorithm. Write alternate algorithm to address those performance issues.**

Ans: The non-token-based Ricart-Agrawal algorithm is a mutual exclusion algorithm used in distributed systems to ensure that only one process can access a critical section at a time. However, this algorithm can suffer from certain performance issues:

- High Message Overhead: The non-token-based approach requires each process to send and receive messages to/from all other processes. This results in high message overhead as the number of processes increases.
- Contention: Processes may request access to the critical section concurrently, leading to contention for resources and potential delays due to message transmission, processing, and replies.
- Blocking: If a process receives multiple requests simultaneously, it might block all requests until it can enter the critical section, leading to further delays.
- Deadlocks: Under certain conditions, the algorithm can result in deadlock situations where processes are stuck waiting for each other to release resources.

An alternate approach to address the performance issues of the non-token-based Ricart-Agrawal algorithm is to use a distributed queue-based approach. This approach maintains a queue of processes requesting access to the critical section. Here's how the algorithm works:

- Request Queue: Each process maintains a local queue of requests to enter the critical section.
- Request Propagation: When a process wants to enter the critical section, it sends a request message to all other processes. The request message includes the requesting process's ID and timestamp.
- Queue Update: Upon receiving a request, a process adds the requesting process's information to its local queue, along with the timestamp.
- Priority Queue: Processes prioritize requests based on timestamps. The process with the lowest timestamp gets priority.

- Access Decision: When a process wants to enter the critical section, it checks if its request is at the front of its local queue. If it is, and no higher-priority requests are pending, the process can enter the critical section.
- Release and Dequeue: After a process exits the critical section, it removes its request from its local queue and notifies other processes by sending release messages.

By using a distributed queue-based approach, the performance issues of the non-token-based algorithm can be mitigated:

- Reduced Message Overhead: Processes send fewer messages, only updating queues instead of sending messages to all processes.
- Contention and Blocking Mitigation: The distributed queue helps manage contention and reduces the chances of blocking since processes are prioritized based on timestamps.
- Deadlock Prevention: The queue approach helps prevent deadlocks because processes can be serviced in an orderly manner based on timestamps.
- Optimized Resource Utilization: Processes only send messages when they have a legitimate request, reducing unnecessary message transmission.

6) How to come to consensus in DS? Discuss with an approach, How do you make the distributed system service highly available?

Ans: Coming to consensus in a distributed system means that all nodes in the system agree on a particular value or decision. Achieving consensus is challenging due to the potential for network delays, message losses, node failures, and the inherent asynchrony of the distributed environment. One well-known approach to achieve consensus is the Paxos algorithm.

The Paxos algorithm was introduced by Leslie Lamport to achieve consensus in a distributed system. It uses a multi-round approach where nodes exchange messages to reach agreement. Here's a high-level overview of the Paxos algorithm:

- **Phase 1 - Prepare:** A node (proposer) sends a "prepare" message to other nodes (acceptors) containing a proposal number. Each acceptor checks if the proposal number is higher than any proposal it has seen before. If it is, the acceptor replies with a "promise" message that includes the highest proposal it has accepted.

- **Phase 2 - Accept:** If the proposer receives promises from a majority of acceptors, it chooses a value to propose and sends an "accept" message to the acceptors. The acceptors will only accept the proposal if the value has not been accepted in a previous proposal.
- **Phase 3 - Learn:** Once a value is accepted by a majority of acceptors, the proposer broadcasts a "learn" message to all nodes, informing them of the agreed-upon value.

Paxos ensures that only one value can be agreed upon and that the value is chosen from the proposals that were presented. It is designed to work even in the presence of failures and network delays.

Achieving high availability in a distributed system involves designing the system to minimize downtime and provide continuous service even in the face of failures. Here are some key approaches to making a distributed system highly available:

- **Redundancy:** Deploy multiple instances of services and components across different nodes or data centers. If one instance fails, another can take over to ensure continuity.
- **Load Balancing:** Distribute incoming requests evenly across multiple servers to prevent any single server from becoming a bottleneck. This ensures that resources are utilized optimally.
- **Failover Mechanisms:** Implement failover mechanisms that automatically detect node failures and redirect traffic to backup nodes. This minimizes downtime and maintains service availability.
- **Replication:** Replicate data across multiple nodes to ensure that if one node fails, data is still accessible from other nodes. Replication can be achieved through techniques like master-slave or multi-master replication.
- **Health Monitoring:** Continuously monitor the health of nodes and services. If a node or service becomes unresponsive or experiences performance degradation, it can be taken out of rotation until it recovers.
- **Isolation and Microservices:** Isolate different components of the system into separate microservices. This way, failures in one component do not affect the entire system, and individual services can be scaled independently.
- **Caching and Content Delivery Networks (CDNs):** Use caching and CDNs to serve frequently accessed content closer to the user, reducing the load on the main servers and improving response times.

- **Auto-Scaling:** Implement auto-scaling mechanisms that automatically adjust resources based on demand. This ensures that the system can handle varying workloads without manual intervention.
- **Stateless Services:** Design services to be stateless as much as possible. Stateless services are easier to scale and recover, as they don't rely on maintaining specific server states.

By combining these strategies and tailoring them to the specific requirements of the distributed system, high availability can be achieved, ensuring that the system remains operational and responsive even in the presence of failures and challenges.

7) What are the relationship between parent and child transaction in DS? Write the problems of locking with the solutions to avoid it.

Ans: In a distributed database or distributed system, transactions often involve multiple operations that need to be executed together in a coordinated manner. These operations might be related hierarchically, leading to the concept of parent and child transactions.

Parent and child transactions refer to a hierarchical relationship between transactions. A parent transaction is a higher-level transaction that encapsulates one or more child transactions. Child transactions are nested within the parent transaction. The parent transaction can be thought of as a wrapper that groups and manages the execution of its child transactions.

For example, in a distributed database scenario, a parent transaction might involve updating an account balance and transferring funds between accounts. The child transactions within this parent transaction could be updating the balance of the sender's account and the receiver's account.

The relationship between parent and child transactions is essential for maintaining data integrity and consistency in a distributed environment. If a parent transaction fails or is rolled back, it should ensure that the associated child transactions are also rolled back to maintain data consistency.

Problems of Locking in Distributed Systems and Solutions:

Locking is a common technique used to manage concurrent access to shared resources in distributed systems. However, locking can lead to various problems that need to be addressed to ensure data consistency and efficient system operation:

1. Deadlocks: Deadlocks occur when two or more transactions are waiting for resources held by each other, leading to a circular waiting condition. Deadlocks can cause transactions to stall indefinitely.

Solution: Implement deadlock detection and resolution mechanisms. Techniques like timeout-based approaches or resource allocation graph algorithms can help identify and break deadlocks.

2. Lock Contention: High lock contention occurs when multiple transactions are competing for the same resources, leading to performance degradation due to frequent lock acquisitions and releases.

Solution: Use fine-grained locking and avoid locking large portions of data. Optimistic concurrency control techniques, such as timestamp-based ordering, can help reduce lock contention.

3. Locking Overhead: Locking introduces additional overhead due to the need to acquire and release locks, leading to reduced system performance.

Solution: Use lock-free data structures and algorithms wherever possible. Opt for optimistic concurrency control mechanisms that allow transactions to proceed independently until they need to be synchronized.

4. Isolation Levels: Different isolation levels (e.g., Read Uncommitted, Read Committed, Serializable) dictate the level of locking and visibility of changes across transactions. Choosing the wrong isolation level can lead to problems like dirty reads, non-repeatable reads, and phantom reads.

Solution: Select the appropriate isolation level based on the application's requirements. In some cases, relaxing isolation levels can improve concurrency and performance.

5. Lock Escalation: Lock escalation occurs when a database system converts fine-grained locks to coarser locks to reduce overhead. This can lead to reduced concurrency and performance.

Solution: Employ dynamic lock escalation mechanisms that consider the workload and resource usage patterns. Avoid overly aggressive lock escalation strategies.

6. Lock Durations: Holding locks for an extended period can lead to contention and decrease system throughput.

Solution: Minimize the duration for which locks are held. Use techniques like deadlock detection to release locks held by stalled transactions.

While locking is essential for ensuring data consistency in distributed systems, it introduces challenges such as deadlocks, lock contention, and overhead. Solutions involve a combination of well-designed locking strategies, appropriate isolation levels, and concurrency control mechanisms to mitigate these issues and maintain system performance and data integrity.

8) **How do you avoid faults in DS? Compare independent checkpointing with coordinated checkpointing approach.**

Ans: Avoiding faults in distributed systems is a complex task that involves various strategies and techniques to ensure system reliability, availability, and fault tolerance. While complete avoidance of faults is not always possible, minimizing their impact and ensuring system recovery are essential. Two common approaches to enhance fault tolerance in distributed systems are independent checkpointing and coordinated checkpointing.

Independent Checkpointing:

Independent checkpointing is a fault tolerance technique where individual processes or nodes in a distributed system take periodic checkpoints of their own state independently without coordinating with other processes. Each process saves its state to stable storage at its own pace.

Coordinated Checkpointing:

Coordinated checkpointing involves a coordinated effort among processes to take checkpoints simultaneously. Processes coordinate their checkpointing activities to ensure that they reach a consistent global state.

Comparison:

- Consistency: Coordinated checkpointing ensures a consistent global state, while independent checkpointing can lead to an inconsistent state during recovery.
- Rollback: Coordinated checkpointing minimizes rollback propagation, whereas independent checkpointing can result in larger-scale rollbacks.

- Coordination Overhead: Independent checkpointing has lower coordination overhead, but coordinated checkpointing has higher coordination and communication overhead.
- Recovery Time: Coordinated checkpointing leads to controlled recovery with shorter rollback propagation, whereas independent checkpointing might have longer recovery times due to inconsistent states.
- Implementation Complexity: Independent checkpointing is simpler to implement, while coordinated checkpointing requires synchronization mechanisms.

9) Write short notes on:

a) Monolithic and Micro-Kernel:

Monolithic Kernel:

A monolithic kernel is a traditional operating system architecture where the entire operating system and its components are tightly integrated into a single, large executable. In a monolithic kernel, all system services, device drivers, file systems, memory management, and other functionalities run in the same address space as the kernel itself.

Advantages:

- **Performance:** Monolithic kernels typically have lower overhead because they avoid the inter-process communication (IPC) required in micro-kernel architectures.
- **Simplicity:** Developing and maintaining a monolithic kernel can be simpler since all components are tightly integrated.
- **Efficient Communication:** Communication between kernel components is direct and efficient due to their close proximity.

Disadvantages:

- **Limited Modularity:** Changes in one part of the kernel can affect other parts, leading to potential bugs and complexities.
- **Scalability:** Monolithic kernels can become less efficient as the system scales, leading to potential performance bottlenecks.
- **Reliability:** A bug in one part of the kernel can crash the entire system, reducing system reliability.

Micro-Kernel:

A micro-kernel architecture is an alternative to the monolithic approach. In a micro-kernel system, the core functionalities such as memory management, basic IPC, and scheduling are kept in the kernel space, while other services and drivers are moved to user space as separate processes.

Advantages:

- **Modularity:** Micro-kernels offer higher modularity as individual services and drivers run in user space, allowing easier updates and maintenance.
- **Reliability:** Faults in user-level components are less likely to crash the entire system, improving system reliability.
- **Scalability:** Micro-kernels can be more scalable as many services run in user space, reducing kernel complexity.
- **Customizability:** Users can choose and load only the required services, reducing the attack surface and optimizing resource usage.

b)Services provided by CORBA with the functions of Object Adapter:

CORBA is a middleware technology that enables communication between distributed objects across different programming languages and platforms. It provides a set of services to facilitate remote method invocation and interaction between objects in a distributed system. Some of the key services provided by CORBA are:

- Object Request Broker (ORB): The ORB is the core component of CORBA that handles communication between distributed objects. It marshals and unmarshals method calls and parameters, manages object references, and routes requests to the appropriate objects.
- Interface Definition Language (IDL): IDL is a language-neutral way to define the interfaces of objects in a distributed system. It allows developers to specify the methods, data types, and communication protocols that objects will use to interact.
- Object Services: CORBA provides a range of standard services that can be used by distributed objects, such as naming, trading, event notification, and persistence.
- Life Cycle Services: These services manage the creation, deletion, and activation of objects in a distributed environment.

- Naming Service: The naming service allows objects to be registered with and looked up by name, providing a way to locate objects dynamically.
- Trading Service: The trading service enables clients to locate objects based on their properties and interfaces, making it easier to find and use distributed components.

The Object Adapter is a component within the CORBA ORB that provides an interface between the client and the server objects. It is responsible for the following functions:

- Object Activation and Deactivation: The Object Adapter manages the activation and deactivation of objects in response to client requests. It activates objects when they are needed and deactivates them when they are no longer in use, optimizing resource usage.
- Object Reference Handling: The Object Adapter handles object references, allowing clients to transparently invoke methods on remote objects as if they were local. It marshals and unmarshals parameters and results during remote method calls.
- Implementation Hiding: The Object Adapter hides the implementation details of objects from clients. Clients interact with objects through their interfaces defined in the IDL, and the Object Adapter handles the mapping of method calls to the actual object implementations.
- Request Forwarding: When a client invokes a method on a remote object, the Object Adapter forwards the request to the appropriate object instance, managing the routing of requests and responses.

c)Two Phase Distributed Commit:

Two-Phase Commit (2PC) is a distributed algorithm used to achieve consensus among multiple participants (nodes or processes) in a distributed system regarding whether to commit or abort a transaction. It ensures that all participants agree on the outcome of a distributed transaction, ensuring data consistency and integrity across the system.

The 2PC algorithm consists of two phases: the preparation phase and the commitment phase. Here's how the algorithm works:

Preparation Phase:

- The transaction coordinator (usually the initiator of the transaction) sends a prepare message to all participants, asking them to vote on whether they can commit the transaction or not.
- Participants receive the prepare message and internally decide whether they can commit the transaction. If everything is in order, they vote "YES." If there's an issue (e.g., resource unavailability), they vote "NO."
- Participants reply to the coordinator with their votes.

Commitment Phase:

- Based on the received votes, the coordinator determines the final decision for the transaction. If all participants voted "YES," the coordinator sends a commit message to all participants. If any participant voted "NO," the coordinator sends an abort message.
- Participants receive the commit or abort message and act accordingly:
- If they received a commit message, they commit the transaction and release any locks held.
- If they received an abort message, they roll back the transaction and release any locks held.

d) Distributed Debugging:

Distributed debugging is the process of identifying, diagnosing, and resolving issues in a distributed system. It involves identifying the root causes of failures, anomalies, or unexpected behaviors that arise due to the complexity of interactions among multiple components across different nodes and networks. Distributed debugging tools and techniques provide insights into the system's internal state, message flows, and resource usage to help developers locate and fix errors. Techniques such as distributed logging, distributed tracing, and snapshot capture aid in collecting and analyzing information across multiple nodes. While distributed debugging can be challenging due to the lack of a single execution context and the potential for non-deterministic behavior, it plays a critical role in maintaining the reliability and performance of complex distributed systems.

e) RPC communication semantics:

Remote Procedure Call (RPC) communication semantics define how remote calls between distributed components behave in terms of order, reliability, and error handling. There are three primary RPC communication semantics:

- **At Most Once (AMO):** In this semantics, the RPC system guarantees that a remote procedure will be executed at most once. If a caller sends an RPC request, the system ensures that the request is executed on the remote server. If a response is lost or there's a network error, the system will retry the request. This approach guarantees that a remote procedure will not be executed more than once, ensuring idempotence. However, there is a possibility that some valid requests might be missed if a response is lost.
- **At Least Once (ALO):** In this semantics, the RPC system guarantees that a remote procedure will be executed at least once. If a caller sends an RPC request and does not receive a response, it will retry the request. The remote server needs to handle duplicate requests and ensure idempotence to avoid executing the same operation multiple times. This approach ensures that no valid request is lost, but it can lead to duplicate executions.
- **Exactly Once (EO):** This is the most stringent semantics, ensuring that a remote procedure is executed exactly once, even in the presence of failures or retries. Achieving exactly once semantics can be complex, especially in distributed systems. It requires mechanisms for detecting duplicates and maintaining the state to prevent redundant executions.