



**TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
THAPATHALI CAMPUS**

DS Assignment:

2072 Old Questions

Submitted By:

**Prabin Bohara (THA076BCT026)
Prabin Sharma Poudel (THA076BCT027)
Raj Kumar Dhakal (THA076BCT033)
Sajjan Acharya (THA076BCT038)**

Submitted To:

Department of Electronics and Computer Engineering
Thapathali Campus
Kathmandu, Nepal

July, 2023

2072 Chaitra [Regular]

33 TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
Examination Control Division
2072 Chaitra

Exam.	Regular		
Level	BE	Full Marks	80
Programme	BCT	Pass Marks	32
Year / Part	IV / I	Time	3 hrs.

Subject: - Distributed System (CT703)

- ✓ Candidates are required to give their answers in their own words as far as practicable.
 - ✓ Attempt All questions.
 - ✓ The figures in the margin indicate Full Marks.
 - ✓ Assume suitable data if necessary.
1. Differentiate between centralized and distributed system? Explain the design issues related to distributed system. [2+6]
 2. Discuss the importance of Distributed File System (DFS). Describe the operations of SUNNFS with its properties. [2+6]
 3. Explain RMI with suitable diagram. How RMI is superior to RPC? [8+2]
 4. What is the role of middleware in DS? Explain about CORBA and its services. [2+8]
 5. Differentiate between physical clock and logical clock. Why it is difficult to synchronize physical clock? Describe a method for physical clock synchronization. [2+2+6]
 6. What are the basic requirements for mutual exclusion in distributed system? Explain the non-token based distributed mutual exclusion algorithm and compare it with token based algorithm. [2+8]
 7. What are the reasons for Replication? Explain active replication model with its advantages and disadvantages. [3+5]
 8. What do you mean by nested transactions? Explain optimistic concurrency control method with its advantages over other concurrency control methods. [2+4+2]
 9. Write short notes on: [4+4]
 - i) Distributed OS
 - ii) JINI

Answers:

Q. n. 1. Answer:

S.N.	Centralized System	Distributed System
1.	Uses client/server architecture where one or more client nodes are directly connected to a central server.	Utilizes computational resources across multiple, separate computation nodes to achieve a common, shared goal
2.	Network resources are placed and managed from a main location, by administrators.	Resources are on separate nodes, which need to be communicated and synchronized over a common network.
3.	Characteristics: <ul style="list-style-type: none"> ➤ Client clock needs to be synchronized with the global clock of the central server. ➤ Central node failure causes the entire system to fail. 	Characteristics: <ul style="list-style-type: none"> ➤ Lack of global clock, each node function according to its local clock. ➤ Failure at one node still allows the functioning of the whole system.
4.	Architecture: <ul style="list-style-type: none"> ➤ Client-Server Architecture 	Architecture: <ul style="list-style-type: none"> ➤ Peer to Peer ➤ N-tier Architecture
5.	Advantages: <ul style="list-style-type: none"> ➤ Easy to physically secure. ➤ Dedicated resources (memory, CPU cores, etc.) ➤ Easy detachment of a node from the system. ➤ More cost-efficient for small systems up to a certain limit. 	Advantages: <ul style="list-style-type: none"> ➤ Distributed systems are highly scalable. ➤ Fault tolerance ➤ Distribution of heavy workloads ➤ Reliable, Cost effective and improved system performance due to parallel processing.
6.	Disadvantages: <ul style="list-style-type: none"> ➤ Abrupt failure of entire system ➤ Difficulty in server maintenance. ➤ Lack of Transparency, Scalability and innovation. 	Disadvantages: <ul style="list-style-type: none"> ➤ Difficult to achieve consensus. ➤ Synchronization issues. ➤ High development and maintenance cost.
7.	Applications/Use cases: <ul style="list-style-type: none"> ➤ Personal computing, Single-gaming. ➤ Data Analysis 	Applications/Use cases: <ul style="list-style-type: none"> ➤ Cluster computing, Grid computing. ➤ Multi-player online games

<p>8. Wikipedia. Consider a massive server to which we send our requests and the server responds with the article that we requested. Suppose we enter the search term 'junk food' in the Wikipedia search bar. This search term is sent as a request to the Wikipedia servers (mostly located in Virginia, U.S.A) which then responds back with the articles based on relevance. In this situation, we are the client node, Wikipedia servers are the central server.</p>	<p>Example: Google search system. Each request is worked upon by hundreds of computers that crawl the web and return the relevant results. To the user, Google appears to be one system, but it actually is multiple computers working together to accomplish one single task (return the results to the search query).</p>
---	---

There are several design issues in distributed system, which are given below:



1. Heterogeneity:

- Distributed computers consist of many different sorts of networks, and their differences are masked by the fact that all the computers attached use Internet

protocols to communicate. Computer on one network need an implementation of the IP for another network for it to communicate with the nodes of another network.

- Data types such as integers are represented in different ways in different sorts of hardware. These differences must be dealt with if messages are to be exchanged between programs running on different hardware.
- Even if an Operating system provides an implementation of the internet protocols, they may not provide the same API to these protocols. For example, the calls for message exchange in UNIX and Windows are different.
- Different programming languages use different representation for characters and data structures. These differences also should be addressed.

✚ Solutions:

- Middleware such as CORBA: They deal with the differences in the OS and the hardware. They also provide uniform computational models used by programmers.
- Mobile code: It's a program code that can be transferred from one computer to another and run at the destination. Example: JAVA applets.
- Virtual machines: They provide a way of making code executable on variety of host computers, as the compiler for a particular programming language generates code for a virtual machine instead of a particular hardware order code.

2. Openness:

- The openness means if the system can be extended or reimplemented by developers. Openness cannot be achieved unless the specifications and the documentation of the key software interfaces of the system is made available or published.
- In distributed systems many components are engineered by different people, thus developers also have that challenges even after acquiring the published documentation.

✚ Solution:

- Requests for Comments (RFC) introduced by the designers of the internet protocols. This practice has continued and forms the basis for technical documentation to this day.

3. Security:

The challenge is to send a sensitive information over a network without any security issues.

Security encapsulates 3 areas:

- Confidentiality: Protection against disclosure to unauthorized individuals.

- Integrity: Protection against alteration or corruption.
- Availability: Protection against interference with the means to access the resources.

✚ Solutions:

- Firewall: Use of firewall can restrict the traffic that can enter and leave. But this doesn't ensure the appropriate use of the resources by the user that is already in.
- Encryption: Ensures 2 things- conceal the contents of the message, and correctly identify the recipient of the sensitive message.

✚ Serious challenges with no optimum solutions:

- Denial of Service Attacks: When the service is bombarded with pointless requests to make serious users unable to use it.
- Fishy Mobile codes: Mobile codes need to be handled with care. For example, an executable program sent through an Email can have unpredictable consequences on the receiver's PC when he/she tries to run it.

4. Scalability:

- Controlling the cost of the physical resources is difficult when scaling. It's so especially because as the number of users increase, the unitarily scaled number of servers may not be enough to handle in practical scenarios due to bottleneck and latency. For example, if 1 server handles 20 users, 2 servers may not be able to handle 40 users!
- Controlling the performance loss is difficult as the nodes are added. This is due to the performance bottleneck. Although hierarchical structures scale better than the linear algorithms, the time taken to access them is $O(\log n)$.
- Preventing the software resources is a major challenge in the internet. The supply of the available internet address is running out. For this reason, a new version of protocol with 128-bit internet address is being adopted, and this will require modifications to many software components. But this will also bring out another challenge- larger internet addresses will occupy extra space in messages and in computer storage.
- Avoiding the performance bottleneck is difficult to achieve.

✚ Solution:

- Decentralize the algorithms to avoid bottleneck in performance,
- Data replication.
- Associated techniques of caching in the located server computers.

- Deployment of multiple servers.

5. Failure handling:

- Failures in distributed system are partial. Therefore, handling the failures might prove difficult.
- To detect failures, checksum can be used, but that technique is not enough to detect other failures such as remote crashed server in the internet. The main challenge is to manage in the presence of failures that can't be detected but are suspected.
- Masking of failures may not be efficient every time as for the worst cases, the replicated disks can be corrupted too and the message may not get through the reasonable time however often it is retransmitted.
- Roll back operation may cause incomplete operation and may need to be restarted again.

✚ Solution:

- Two different routes in minimum between any two routers in the internet.
- Failure tolerance by informing users of the current failures can be helpful to not cause wastage of time of the users.
- The servers can be designed so as to detect faults in their peers. The clients can be redirected to the remaining peers.

6. Concurrency:

- To achieve better throughput, services allow multiple client requests in parallel but this can cause inconsistency if not executed in managed concurrent threads.
- The synchronization of clocks is done in some concurrency protocols such as Timestamp ordering protocol. But the synchronization itself is a very challenging task in itself.

✚ Solution:

- Any programmer who takes an implementation of an object that was not intended for use in distributed system must do what is necessary to include the concurrency tackling mechanism.
- Use of Semaphores in OS.
- Use of various concurrency control mechanisms such as 2PL, Timestamp Ordering protocol, multi-version control,
- Use of Data segmentation and apply various concurrency techniques in different segments separately depending upon the nature of the data.

- Employ hybrid concurrency control manager in the server.

7. Transparency:

Transparency is the concealment of resources that are not directly relevant to the task for users and application programmers. ANSA defines 8 forms of transparency:

- Access Transparency:
 - This refers to the illusion of accessing local and remote resources in a uniform manner, using identical operations.
 - API for files use the same operations to access both local and remote files.
 - Achieving it is difficult because hiding differences in resource location, data representation, and access mechanisms requires sophisticated middleware.
- Location Transparency:
 - This involves concealing the actual location of resources from users and applications.
 - URLs are location transparent. (But not mobility transparent)
 - Location and Access Transparency are combinedly called as Network Transparency.
 - Ensuring location transparency is challenging because it demands mechanisms that can manage and resolve the dynamic changes in resource locations.
- Migration (Mobility) Transparency:
 - This form of transparency allows resources to be moved without affecting ongoing computations.
 - Achieving migration transparency is complex due to the need for maintaining consistent states across distributed nodes during resource movement.
- Replication Transparency:
 - This involves providing multiple copies of resources while concealing the presence of duplicates.
 - In case of mobile phone call, the caller is a client and the callee is called to be a resource. In this example, the 2 users are making a call unaware of the mobility of their phones between calls.
 - Achieving replication transparency is difficult due to the need for synchronization and consistency maintenance among replicas.

- Concurrency Transparency:
 - Concurrency transparency ensures that multiple users can access and modify resources simultaneously without conflicts.
 - It's challenging because handling concurrent access, ensuring data consistency, and avoiding race conditions require complex synchronization mechanisms.
- Failure Transparency:
 - This form of transparency conceals failures in the system from users and applications.
 - The electronic mails are eventually delivered even if there is a failure at the time of delivery. The faults are masked by attempting to retransmit the messages until they are successfully delivered.
 - Ensuring failure transparency is challenging due to the need for fault detection, fault tolerance mechanisms, and graceful degradation.
- Performance Transparency:
 - Performance transparency hides variations in performance caused by changes in load and resource availability.
 - Achieving performance transparency is difficult because it requires efficient load balancing, resource monitoring, and dynamic adaptation mechanisms.
- Scaling transparency:
 - Hiding the complexities of system scaling from users and applications.
 - Ensuring seamless expansion or contraction of resources while maintaining performance and availability is challenging.

8. Quality of Service:

QoS (Quality of Service) captures the following aspects, and the reasons for the difficulty in achieving them are also written below:

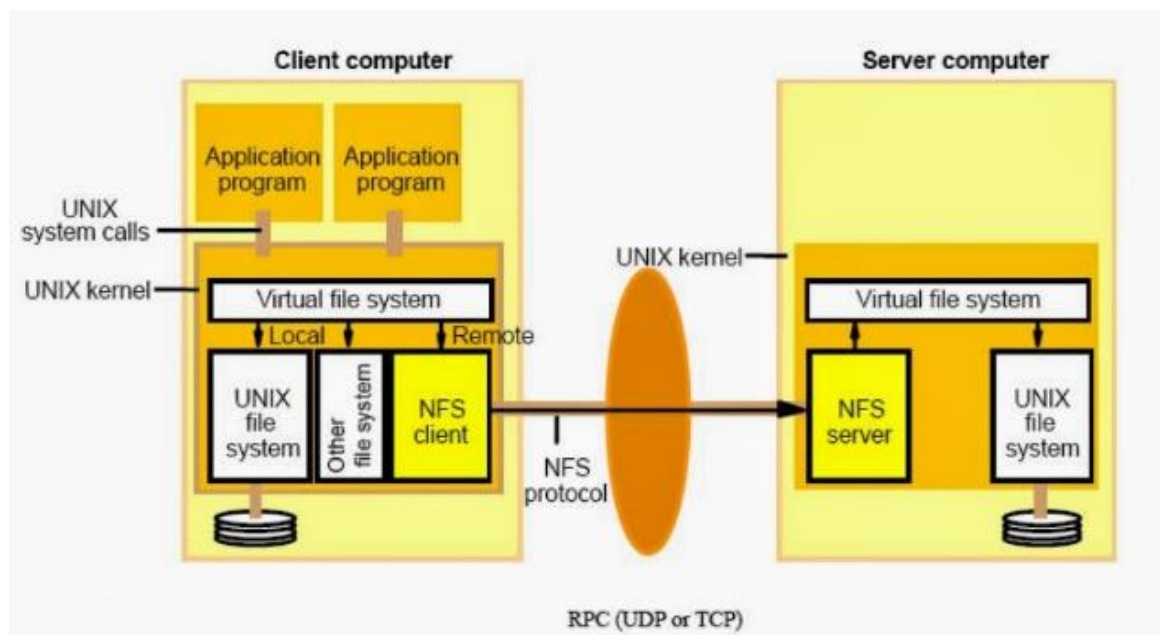
- Reliability and Availability:
 - Ensuring consistent and reliable access to resources across distributed nodes is complex.
 - Difficult to achieve due to potential network failures, node crashes, and varying communication latencies.
- Response Time:

- Providing predictable response times for user requests across distributed components is a challenge.
- Challenging to achieve due to network congestion, resource contention, and varying processing speeds.
- Bandwidth Management:
 - Allocating and managing network bandwidth fairly among competing applications is difficult.
 - Achieving fairness is complex due to dynamic traffic patterns, varying data sizes, and diverse application requirements.
- Prioritization:
 - Assigning appropriate priorities to different applications and users based on their needs is complex.
 - Difficult to achieve due to conflicts between high-priority and low-priority tasks during peak loads.
- Resource Reservation:
 - Reserving resources to meet specific QoS requirements for critical applications is challenging.
 - Challenging due to contention for resources and the need to guarantee reserved capacity.
- Adaptability:
 - Adapting to changing network conditions and adjusting QoS parameters dynamically is difficult.
 - Achieving dynamic adaptation is complex due to the need for real-time monitoring and decision-making.
- Security and QoS Trade-offs:
 - Balancing the need for stringent security measures with maintaining desired QoS levels is challenging.
 - Difficult due to potential overhead introduced by security protocols and potential impact on performance.

Q. n. 2. Answer:

The importance of DFS (Distributed File System) are:

- **Scaling:** DFS allows storage and retrieval of files across multiple servers, enabling seamless scaling of storage capacity to accommodate growing data needs.
- **Fault tolerance:** DFS replicates files across distributed nodes, ensuring data availability even if a server or node fails, enhancing system reliability.
- **Redundancy:** By storing multiple copies of files, DFS ensures data redundancy, minimizing the risk of data loss due to hardware failures or disasters.
- **Load balancing:** DFS distributes file requests across servers, preventing resource overload and optimizing performance by balancing the system's load.
- **Collaboration and Accessibility:** DFS provides a centralized point of access to files, facilitating collaboration among geographically dispersed users while maintaining consistent file versions.
- **Data Integrity:** Through replication and data consistency mechanisms, DFS safeguards against data corruption, ensuring the accuracy and reliability of stored information.



Operation:

- SUNNFS operates in a client-server architecture where clients request file operations and servers provide file services.
- **File access request:** Clients initiate file access requests such as read, write, and execute through system calls.
- **Mounting:** Clients use the 'mount' command to establish a connection to the NFS server and mount remote directories as if they were local.

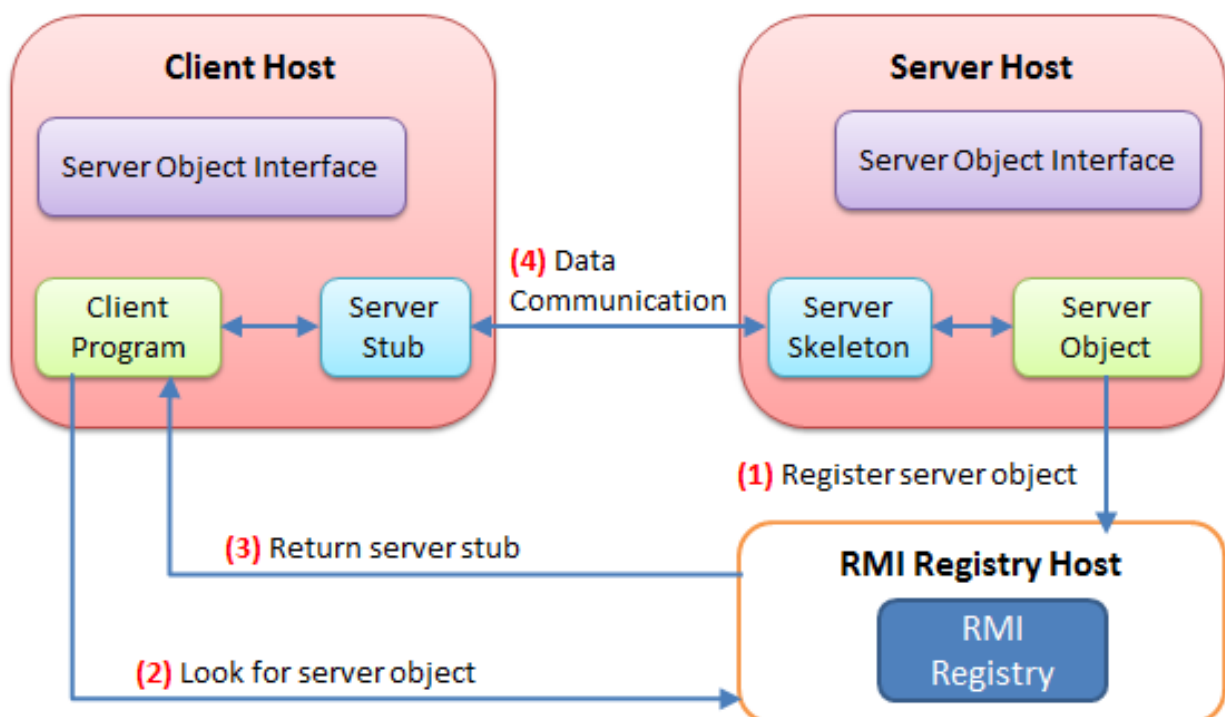
- Clients send RPCs to the server using the appropriate protocol to request file operations.
- A unique file handle representing the file's attributes and location is used to identify files on the server.
- SUNNFS employs a caching mechanism on the client side to store frequently accessed data, reducing the need for repeated server requests. The client caches file attributes like size and modification time to minimize attribute retrieval calls to the server. Caching data in the client's memory enhances read performance by avoiding repetitive data transfer from the server.
- SUNNFS implements coherency mechanisms to ensure data consistency between the client's cache and the server.
- When a client modifies a cached file, it marks the data as "dirty". The server is notified to invalidate other clients' caches and update its own copy.
- Write operations are buffered in the client's cache before being written to the server, enhancing write efficiency.
- Servers maintain statelessness, treating each request independently, which simplifies server design and enables easier fault tolerance.
- Clients and applications interact with remote files as if they were local, thanks to the transparency of the NFS protocol.

The properties of SUNNFS are:

- **Transparency:** SUNNFS provides network transparency, allowing clients to access remote files as if they were local. For example, a user can manipulate a file on a remote server just like a local file.
- **Location Independence:** Clients can access files on remote servers regardless of their physical location. This property enables seamless file access across geographically distributed systems.
- **Heterogeneity:** SUNNFS supports heterogeneous environments, allowing clients and servers to have different operating systems and hardware architectures. For instance, a client running on Linux can access files on a server running Windows.
- **Concurrent Access:** Multiple clients can access the same file simultaneously without conflicts. For example, several users can read and edit a shared document concurrently.

- **Caching Mechanism:** SUNNFS employs caching to store frequently accessed data on the client side. For instance, a client can cache a file's contents, reducing the need for repeated server requests.
- **Statelessness:** Servers maintain statelessness, treating each request independently. This property simplifies server design and enhances fault tolerance. For instance, a server can serve multiple clients without keeping track of their individual states.
- **Remote Procedure Calls (RPCs):** SUNNFS utilizes RPCs to enable communication between clients and servers. Clients invoke remote procedures on servers to perform file operations, such as reading or writing.
- **File Handle Mechanism:** A file handle uniquely identifies files on the server. Clients use this handle to reference files. For example, a client uses a file handle to perform read operations on a specific file.
- **Coherency Mechanisms:** SUNNFS ensures data consistency through coherency mechanisms. When a client modifies a file, it informs the server to invalidate other clients' caches, maintaining data integrity.
- **Asynchronous Write:** Clients can perform asynchronous writes, where data is first stored in the client's cache and then written to the server. This enhances write efficiency and reduces network overhead.

Q. n. 3. Answer:



Explanation:

- JAVA RMI extends the JAVA object model to provide support for distributed objects in the JAVA language. It allows objects to invoke methods on remote objects using the same syntax for local invocation, thus capturing the design goal/challenge of 'Access Transparency'. However, the object making the remote invocation is aware that its target is remote and not local because it must handle '*Remote Exceptions*'. Similarly the implementor of remote object is also aware because it must handle '*Remote Interface*'.
- The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. It does the following:
 - It initiates a connection with remote Virtual Machine (JVM),
 - It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
 - It waits for the result
 - It reads (unmarshals) the return value or exception, and
 - It finally, returns the value to the caller.
- The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:
 - It reads the parameter for the remote method
 - It invokes the method on the actual remote object, and
 - It writes and transmits (marshals) the result to the caller.
- Before sending method calls over the network, parameters are marshalled (converted to a transferable format). On the server, unmarshalling converts parameters back to their original form.
- Objects sent across the network must be serializable, meaning they can be converted into a byte stream for transmission.
- RMI provides security checks through the Java Security Manager, ensuring safe communication between distributed components.
- Remote methods can throw exceptions. RMI transparently handles remote exception propagation and rethrows them on the client side.

RMI is superior to RPC (Remote Procedure Call) because of the following:

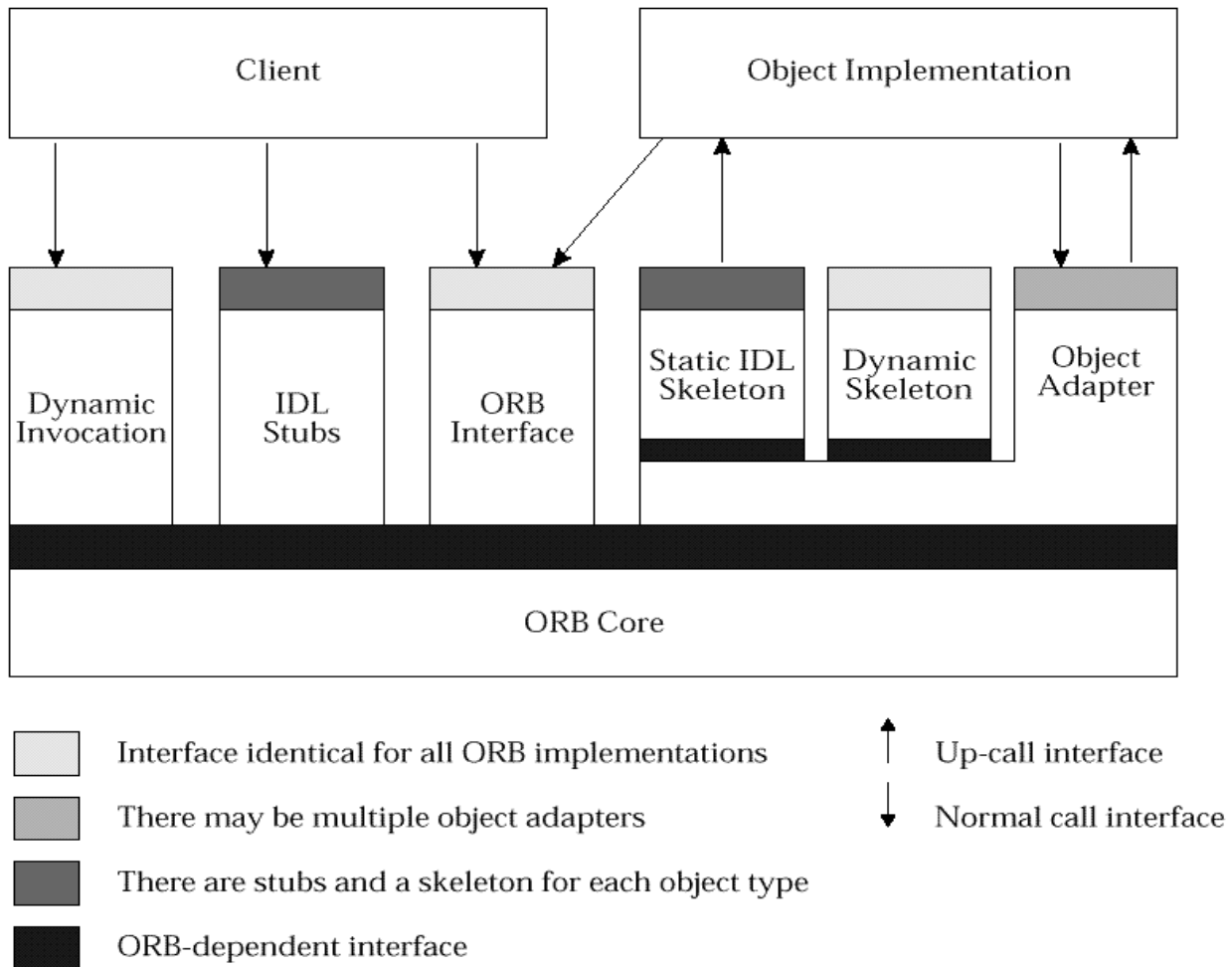
- Efficiency in object-oriented approach: RMI's object-oriented nature aligns with modern programming paradigms, allowing developers to work directly with objects. This reduces the need for data conversions and streamlines the communication process by sending and receiving whole objects. In RPC, working with objects requires manual serialization and deserialization, increasing complexity and potential errors. RMI can pass complex objects as parameters, reducing the need to serialize and deserialize data manually, as often required in RPC.
- Passing complex objects: RMI's capability to transmit complex objects as parameters simplifies the communication process. When passing intricate data structures or entities, RMI avoids the need to break down data into smaller units for transmission, reducing the overhead and improving efficiency. RPC often struggles with complex data types, leading to additional effort in handling data fragmentation and reconstruction.
- Pass by reference and by value: RMI's support for both pass by value and reference is a significant efficiency gain. By passing parameters by reference, RMI enables remote objects to share data without the necessity of full data transfer, enhancing communication speed and reducing network load. RPC, which primarily employs pass by value, often incurs redundant data transmission and processing.
- Reduced overheads: RMI's design is tailored to modern programming practices, resulting in a more streamlined communication protocol. This reduced overhead translates to faster data transmission, quicker method invocations, and minimized computational burden on both the client and server sides. RPC's older design may generate more protocol overhead, leading to slower communication.

Q. n. 4. Answer:

The role of middleware in DS are:

- Middleware acts as an abstraction layer between applications and underlying hardware and network complexities. It shields applications from low-level details, making development and maintenance easier.
- Middleware enables different software components developed in various languages or running on different platforms to work together. It ensures **compatibility** and data exchange.

- Middleware provides security mechanisms like encryption and authentication, safeguarding data and interactions in distributed environments. Examples include SSL/TLS for secure communication.
- Middleware incorporates error detection, handling, and recovery mechanisms. It ensures systems continue to operate even in the presence of failures.
- Examples of middleware include message-oriented middleware (MOM) like Apache Kafka, remote procedure call (RPC) frameworks like gRPC, and web service middleware like Java EE.




CORBA stands for Common Object Request Broker Architecture. It is a framework that enables software components (objects) to communicate and interact across networks, even if they are developed in different languages or run on different platforms. It acts as middleware, providing a standardized way for these distributed objects to locate each other, invoke methods, and exchange data transparently.

The following points further describes the CORBA:


- CORBA allows objects developed in various programming languages (like Java, C++, Python) to seamlessly communicate, thanks to its Interface Definition Language (IDL). IDL abstracts the communication details.
- The heart of CORBA is the Object Request Broker (ORB), which manages object communication. It handles locating objects, method invocations, parameter passing, and data serialization.
- CORBA fosters interoperability by enabling objects running on different hardware and software platforms to collaborate, as long as they adhere to the CORBA specifications.
- Developers define object interfaces using IDL, a neutral language. The IDL compiler then generates language-specific stubs and skeletons, facilitating communication.
- A client invokes a method on a remote object as if it were a local call. The request goes through the ORB, which manages data transformation and ensures seamless communication.
- CORBA provides location transparency, meaning clients don't need to know where remote objects reside. The ORB takes care of locating them, regardless of their physical location.
- Objects can invoke methods on other objects dynamically, without needing to know their specific interfaces in advance. This fosters adaptability and flexibility.
- CORBA offers security features and supports additional services like transactions, naming, and event notifications, enhancing the reliability and capabilities of distributed systems.

Various vendors provide CORBA implementations, like OMG's TAO, Java's JavaIDL, and ORBacus. These frameworks make it easier to develop and deploy CORBA-based applications. Despite its benefits, CORBA may involve complexity due to its initial learning curve and integration challenges between different programming languages and platforms. CORBA revolutionized distributed computing by providing a standardized, language-independent way for objects to interact across networks.

There are various CORBA services. They are given below:

 Naming Service:

- ✓ Role: Provides a directory service, allowing objects to be registered and located by name. Enables clients to find objects without knowing their physical locations.
- ✓ Object's reference is registered with a name, and clients use the name to locate the object. Example: CORBA Naming Service.

 Event Service:

- ✓ Role: Facilitates event-driven communication between objects. Enables asynchronous communication, where an object notifies others about changes or events.
- ✓ Example: A weather monitoring system sending notifications to registered clients when temperature exceeds a threshold.

✚ Life Cycle Service:

- ✓ Role: Manages object creation, deletion, and activation. Ensures objects are properly managed during their lifecycle.
- ✓ Example: An object requesting activation when needed and deactivation when idle to optimize resource utilization.

✚ Trading Service:

- ✓ Role: Allows clients to discover services available in the network dynamically. Enables service providers to register and clients to find suitable services.
- ✓ Example: A client searching for available printers in a network and choosing the most suitable one.

✚ Query Service:

- ✓ Role: Enables clients to query objects for their capabilities and features before invoking methods. Facilitates dynamic adaptation based on object capabilities.
- ✓ Example: A client queries a printer object for supported print formats before sending a print job.

✚ Concurrency Control Service:

- ✓ Role: Manages concurrent access to shared resources in a distributed environment, ensuring data consistency and preventing conflicts.
- ✓ Example: Ensuring that multiple clients can access and modify a shared document without conflicting changes.

● Externalization Service:

- ✓ Role: Converts object data into a transportable format for transmission across networks and vice versa. Handles data serialization and deserialization.
- ✓ Example: Converting a complex object into a byte stream for network transmission and reconstructing it on the receiving side.

● Transaction Service:

- ✓ Role: Ensures data integrity by supporting distributed transactions with ACID properties (Atomicity, Consistency, Isolation, Durability).
- ✓ Example: Ensuring that funds are transferred accurately between accounts across distributed banking systems.
- Notification Service:
 - ✓ Role: Enables objects to send notifications about specific events to registered listeners, enhancing event-driven communication.
 - ✓ Example: Notifying subscribers when a stock price reaches a predefined level in a distributed stock trading system.
- Security Service:
 - ✓ Role: Provides mechanisms for securing communication and data exchange among distributed objects, ensuring confidentiality and integrity.
 - ✓ Example: Encrypting data transmitted between a client and a server to prevent unauthorized access.
- Concurrency Control Service:
 - ✓ Role: Manages concurrent access to shared resources in a distributed environment, ensuring data consistency and preventing conflicts.
 - ✓ Example: Ensuring that multiple clients can access and modify a shared document without conflicting changes.

Q. n. 5. Answer:

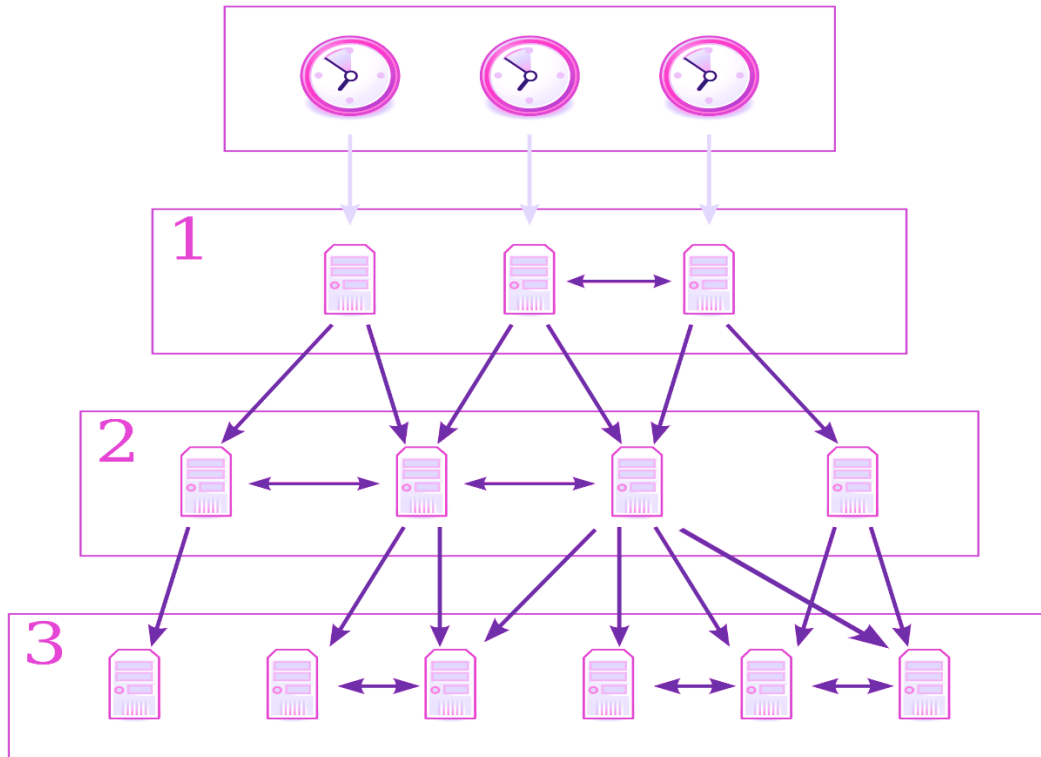
S.N.	Physical Clock	Logical Clock
1.	Measures the progression of time. For example, Wall clock, wristwatch, system clock in a computer.	It's a component for catching sequential and causal connections in a dispersed framework. It is a software counter. For example, Lamport's logical clock, vector clock.
2.	Since, different computers have different crystals that run at different rates, the	Achieves ordering of events but not tied to real-time. It assign time stamps to the events.

	physical clock gradually get out of synchronization.	
3.	Each computer timestamps messages using its local clock. B receives A's message, but due to network delay, its timestamp may not accurately reflect the event's actual order.	Each event (message transmission) gets a logical timestamp. B's message to C is assigned a higher logical timestamp than the message from A, reflecting the causal relationship accurately despite network delays.
4.	The methods of synchronization are: 1. Cristian's method 2. Berkeley's method 3. Network time protocol	The two widely used algorithms are: 1. Lamport's Clock 2. Vector's Clock

The physical clocks are difficult to synchronize due to the following reasons:

- **Hardware Variations:** Hardware components like quartz crystals have inherent variations affecting clock accuracy.
- **External Factors:** Temperature changes, electromagnetic interference, and other environmental factors impact clock speed.
- **Network Latency:** Synchronizing over a network introduces delays, leading to inconsistent clock readings.
- **Clock Drift:** Even if initially synchronized, clocks may drift apart due to minute differences in their internal mechanisms.

Network Time Protocol (NTP) is a widely used method for synchronizing physical clocks in distributed systems. It defines an architecture to enable clients, across the Internet to be synchronized accurately to UTC. NTP employs a client-server model. Clients request time information from one or more NTP servers, which are known to have accurate time references.



Server synchronization can be done in following ways:

- Multicast mode:
 - Servers periodically multicasts time to other servers in the network.
 - Receivers set their clock assuming small delay.
- Procedure Call mode:
 - One server accepts requests from other computers.
 - Server replies with its timestamp.
- Symmetric mode:
 - A pair of servers on higher subnet layers exchange messages to improve accuracy of synchronization over time.

The following points further describes this method:

- Timestamp Exchange: Clients periodically send requests to NTP servers for timestamp information. Servers respond with their current time, including any time offset.
- Clock Adjustment: The client calculates the time offset between its clock and the server's clock based on round-trip time. It adjusts its clock to minimize the time difference.
- Multiple Servers: To enhance accuracy and reliability, clients can query multiple NTP servers. They calculate an average offset from these responses, reducing the impact of network delays.

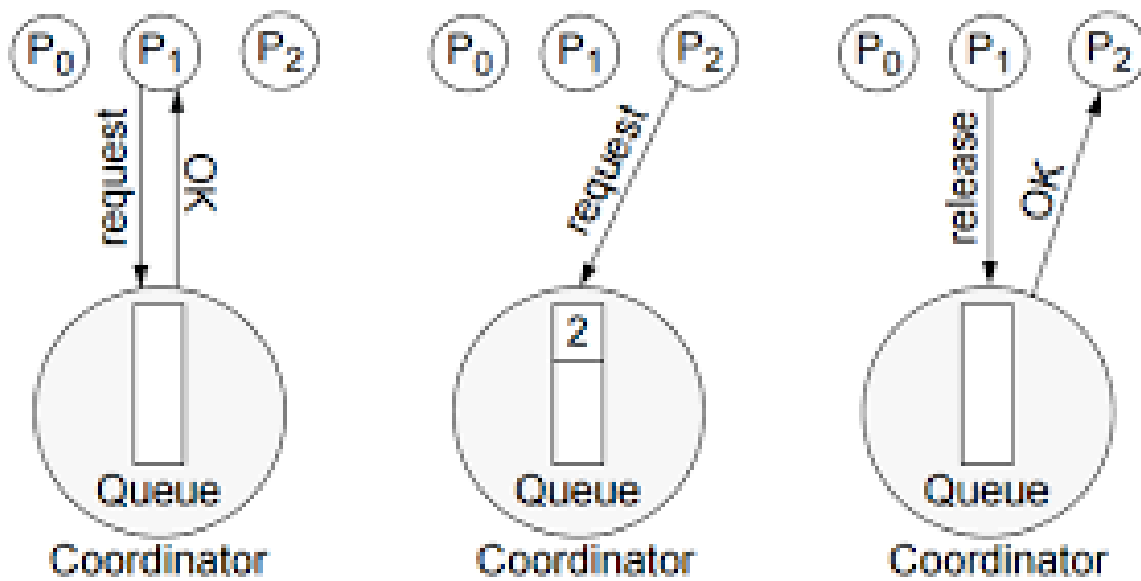
- Stratum Hierarchy: NTP servers are organized in a hierarchical structure with different strata (levels). Stratum 1 servers have direct access to highly accurate time sources like atomic clocks. Lower stratum numbers represent higher accuracy.
- Algorithm Complexity: NTP employs complex algorithms, including filtering, smoothing, and weighted averaging to ensure accurate time synchronization while accounting for potential anomalies.
- Security Measures: NTP includes security measures like authentication to prevent unauthorized time sources and malicious clock manipulations.
- Continuous Adjustment: NTP continually adjusts the local clock's rate to minimize the error and maintain synchronization over time.
- Widely Used: NTP is used across the internet, local networks, and critical systems to maintain accurate time references for various applications.
- Challenges: Despite its effectiveness, NTP can face challenges due to network delays, server reliability, and the complexity of algorithms. Stratum 1 servers are essential for maintaining high accuracy.

Q.n. 6. Answer:

The requirements are:

- Exclusion: Only one process should have access to a shared resource at a given time
- No Process Starvation: Every process requesting the resource eventually gets access.
- Fairness: The order of process access should not be biased; processes should have an equal chance to access the resource.
- No Assumptions: The algorithm should not assume particular properties like process speed, message delay, or system size.
- Progress: If no process is in its critical section and some processes want to enter, the selection of the process that can enter next should not be postponed indefinitely.
- Deadlock Freedom: The system should not deadlock, where processes are stuck indefinitely waiting for resources.

Non-token based algorithm: Center coordinator algorithm



Steps:

- To enter a CS a process sends a requests message to coordinator work while waiting for a reply. During this waiting period the process can continue with other work.
- The reply from coordinator gives right to access critical section based on request queue.
- After finishing critical section operation, the process notifies coordinator with a release messages.

Advantages:

- Easy to implement
- Require only 3 message per access to critical section.

Disadvantages:

- The coordinator can become a performance bottleneck.
- The coordinator is a critical point of failure. If it crashed a new one must be created. An election algorithm can be run to choose one.

S.N.	Non-token based Algorithm	Token-Based Algorithm
1.	No token circulation; processes request access directly. Generally simpler implementation without token management.	Token circulates among processes to grant access. Often involves more complex token management logic.

2.	Higher message overhead due to broadcasting requests.	Lower message overhead due to linear token path.
3.	Less predictable access pattern, as it depends on request arrival and acknowledgments.	More predictable access pattern as token enforces order.
4.	Non-token algorithms are useful for dynamic resource access scenarios.	Token-based algorithms are suitable for systems with fixed resource access sequences.
5.	Ricart-Agrawala Algorithm, Eisenberg-McGuire Algorithm, etc.	Distributed Token Ring Algorithm, Raymond's Tree-Based Algorithm, etc.

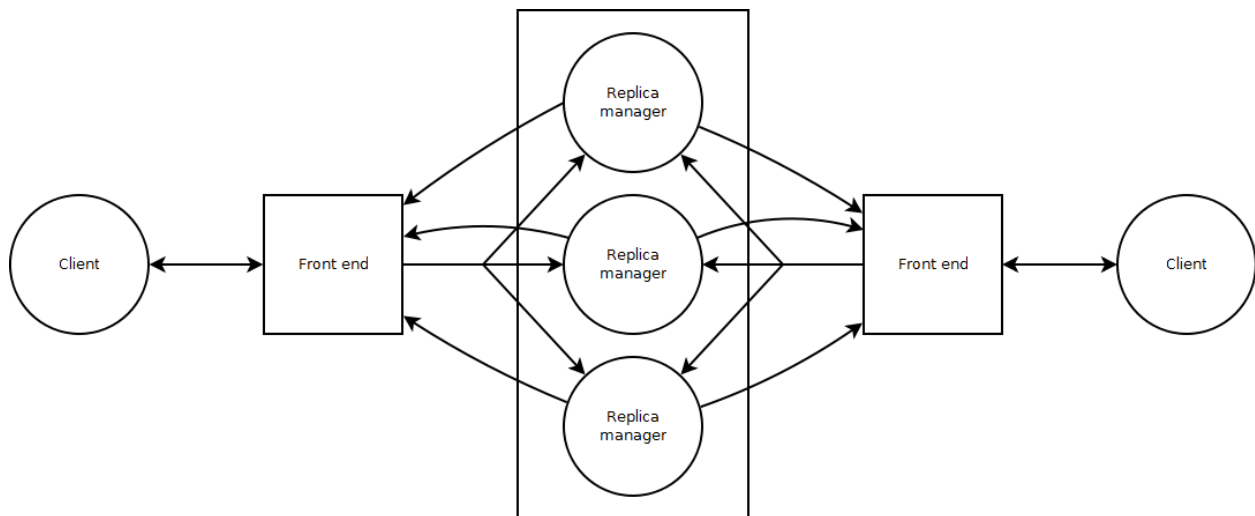
Q. n. 7. Answer:

The needs of replication are:

- Performance Enhancement:
 - ✓ Reduced Load on Servers: Replicating data across multiple servers distributes client requests, reducing the load on individual servers and improving response times.
 - ✓ Localized Data Access: Replicas can be placed closer to clients, minimizing network latency and accelerating data access.
 - ✓ Parallel Processing: Replication allows multiple servers to work in parallel, enhancing throughput for data-intensive operations.
 - ✓ Load Balancing: Requests are evenly distributed among replicas, achieving load balancing and preventing server bottlenecks.
 - ✓ Caching Benefits: Frequently accessed data can be cached in replicas, decreasing the need to fetch data from remote servers.
- Increased Availability:
 - ✓ Fault Detection and Recovery: If a replica fails, clients can switch to another replica, ensuring uninterrupted access to data.
 - ✓ Improved Redundancy: Replicas act as backups; if one replica is unavailable, clients can access others.
 - ✓ Isolation from Network Failures: Replicas can be placed on different network segments, reducing the impact of network failures.

- ✓ Enhanced Scalability: Additional replicas can be added to accommodate growing user demands without affecting availability.
- ✓ Geographic Distribution: Replicas located in different geographical regions offer continuous access, even in the presence of regional outages.
- Fault Tolerance:
 - ✓ Data Recovery: If a replica fails, data can be retrieved from other replicas, minimizing the risk of data loss.
 - ✓ Automatic Failover: In case of a primary server failure, clients can automatically switch to secondary replicas, ensuring uninterrupted service.
 - ✓ Rollback and Consistency: Replication combined with logging allows for recovery to a consistent state after a failure.
 - ✓ Distributed Error Detection: Discrepancies between replicas can be detected and resolved to maintain data consistency.
 - ✓ Data Durability: Replicas provide redundancy, safeguarding data against server crashes and ensuring data persistence.

In active replication, the replica managers are state machines. Active replication is a replication technique in distributed systems where multiple copies (replicas) of data or services are maintained, and all replicas actively process incoming requests from clients. This approach ensures that clients' requests are executed by all replicas in parallel, and the results are compared for consistency before being returned to the client. Active replication enhances fault tolerance and availability by allowing the system to continue functioning even if some replicas fail.



In active replication, the sequence of events involves parallel processing of client requests across multiple replicas while ensuring consistency. Here's the sequence:

- **Client Request:** A client sends a request to a primary replica, initiating the operation.
- **Request Forwarding:** The primary replica forwards the request to all other replicas in the system.
- **Parallel Execution:** Each replica independently executes the request concurrently with other replicas.
- **Result Comparison:** The results produced by each replica are compared to identify any discrepancies.
- **Consensus Decision:** A consensus algorithm (e.g., majority voting) determines the correct result based on replica responses.
- **Result Validation:** The primary replica validates the consensus result against its own execution.
- **Response to Client:** The primary replica responds to the client with the validated result.
- **Data Synchronization:** Replicas synchronize their state periodically to ensure consistency and fault tolerance.

The advantages of active replication are:

- **High Fault Tolerance:** Active replication provides enhanced fault tolerance, as the system can still function even if a subset of replicas fails.
- **Increased Availability:** Clients can direct their requests to any replica, ensuring that the service remains available even if certain replicas are inaccessible.
- **Low Latency:** With requests processed in parallel by multiple replicas, clients can experience lower latency as the system responds faster.
- **High Reliability:** The redundancy of replicas ensures that even if one or more replicas produce incorrect results, the correct result can be determined based on a majority vote.

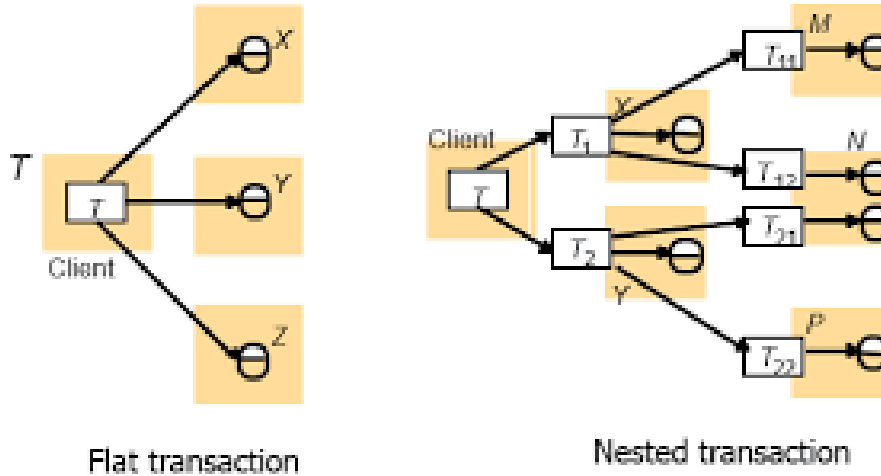
The disadvantages of active replication are:

- **Increased Communication Overhead:** Coordinating and comparing results from multiple replicas introduces additional communication overhead, potentially affecting system performance.
- **Complexity:** Active replication systems require sophisticated mechanisms for synchronizing and comparing results, leading to increased system complexity.
- **Consistency Challenges:** Ensuring consistency among replicas can be challenging, especially in the presence of network delays, failures, and varying execution speeds.

- Limited Scalability: As the number of replicas increases, the coordination and communication overhead can limit the scalability of active replication systems.

Q. n. 8. Answer:

A nested transaction is a transaction that is initiated within the scope of another transaction. It forms a hierarchy where the outer transaction is called the parent transaction, and the inner transaction is called the child transaction. Each nested transaction is treated as a unit of work, and it can either commit or abort independently of its parent transaction.



Advantages of Nested Transaction:

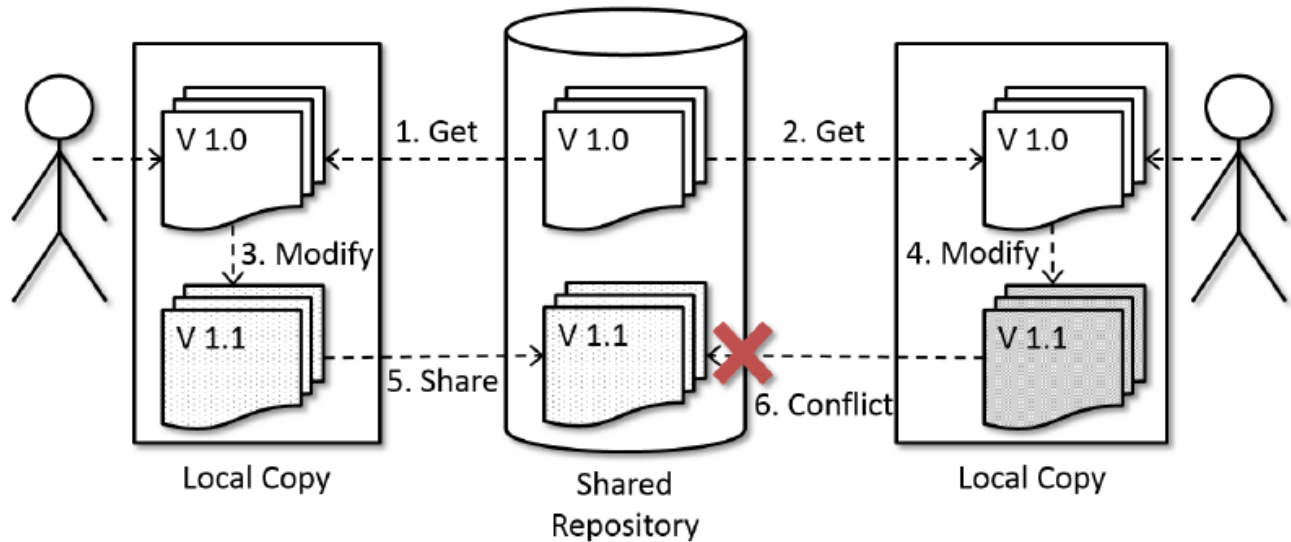
- ✓ Modularization and Isolation: Nested transactions allow breaking down complex operations into smaller, manageable units, enhancing code modularity and reducing complexity. Each nested transaction operates in isolation, providing a clear separation of concerns.
- ✓ Atomicity and Consistency: Nested transactions inherit the properties of atomicity and consistency from their parent transaction. If an inner transaction fails, the parent transaction can be rolled back to a consistent state, ensuring data integrity.

Optimistic concurrency control is a technique used in distributed systems to manage concurrent access to shared data by allowing transactions to proceed without immediate locking. Instead of preventing access, it permits transactions to proceed with the assumption that conflicts will be minimal. It performs conflict detection and resolution during the transaction's commit phase.

The process flow is:

- Read Phase: Transactions read the data they need without acquiring locks. They keep a copy of the read data and record a version number (timestamp or sequence number)

- Execution Phase: Transactions perform their operations without any locking, assuming no conflicts will arise.
- Validation Phase: During commit, transactions verify if the data they've read and modified is still consistent with the current state of the system. This involves checking for conflicts with other concurrent transactions.
- Commit Phase: If no conflicts are detected during validation, the transaction commits its changes. If conflicts are detected, the transaction is aborted, and the process is restarted.



Example:

Consider two transactions, T1 and T2, operating concurrently:

- ✚ T1 reads data (A=100) and plans to increment it by 20.
- ✚ T2 reads the same data (A=100) and plans to increment it by 30.

Both transactions operate optimistically, assuming there won't be any conflicts. During validation, the system detects that both T1 and T2 have modified the same data. Since the modifications conflict, one of the transactions (say, T2) is chosen to be aborted. T2 is restarted with the latest data, ensuring consistency.

Advantages:

- Reduced Lock Contention: Optimistic concurrency control minimizes lock contention, allowing transactions to proceed without waiting for locks.
- Higher Throughput: As transactions don't wait for locks, the system can achieve higher throughput and better utilization.

Q.n. 9. Answer:

Distributed OS (short notes):

A Distributed Operating System (DOS) is designed to manage and coordinate the resources and activities of a distributed computer system, where multiple interconnected computers work together as a unified computing environment.

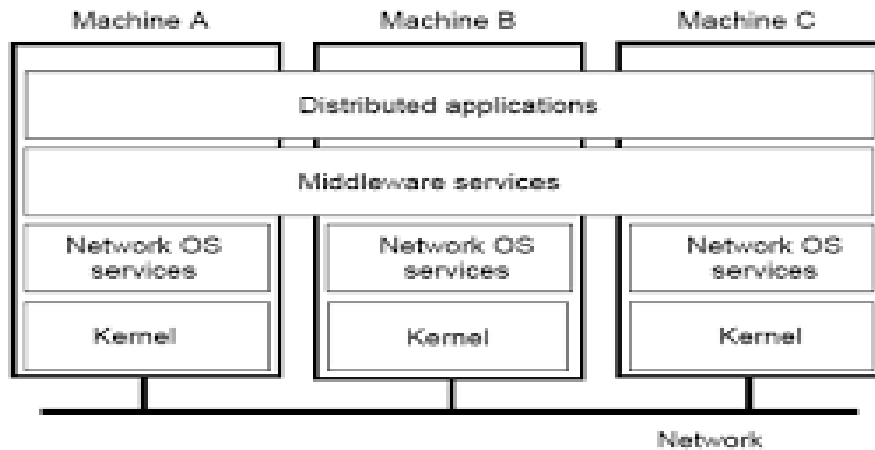


Fig. 1-22. General structure of a distributed system as middleware.

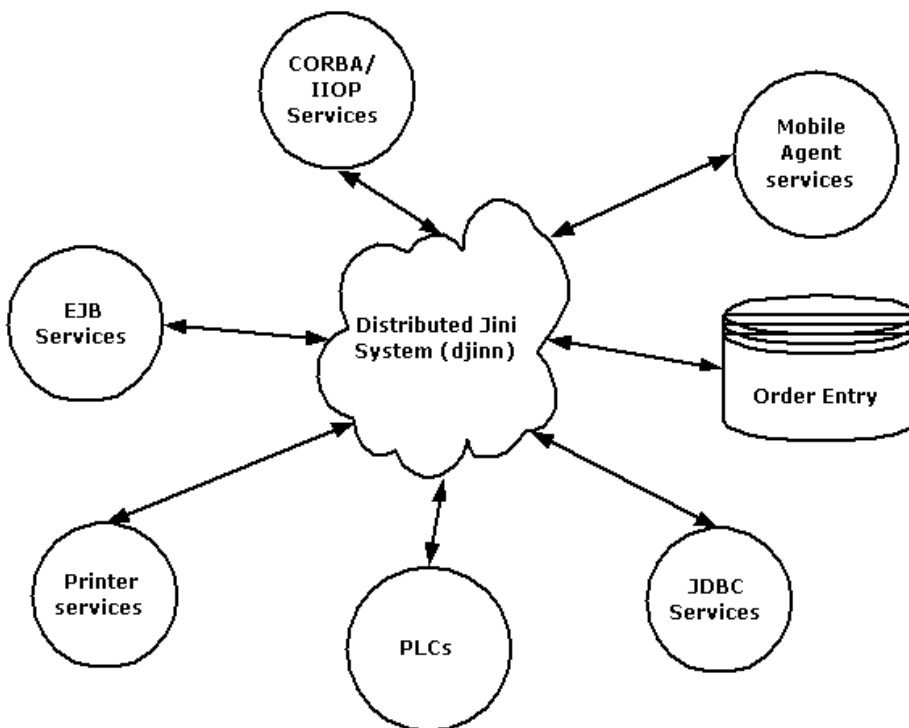
The following points capture the necessary notes or information on Distributed OS (DOS):

- **Transparency:** DOS aims to provide transparency to users and applications, hiding the complexities of the underlying distributed infrastructure. Transparency types include access, location, migration, relocation, replication, concurrency, failure, and scaling transparency.
- **Communication:** Communication mechanisms are crucial in DOS to enable seamless interaction among distributed components. Remote Procedure Call (RPC), message passing, and sockets are commonly used communication models.
- **Naming and directory service:** DOS provides naming services that allow users and applications to access remote resources using human-readable names rather than numerical addresses. Distributed directory services help locate resources in a network using a directory structure.
- **Security:** Security mechanisms in DOS ensure data confidentiality, integrity, authentication, and authorization across a distributed environment. Techniques like encryption, access control, and firewalls are used to enhance security.
- **Process management:** DOS manages processes across multiple machines, providing features like process creation, migration, synchronization, and communication. Distributed scheduling algorithms balance load and optimize resource utilization.

- **Memory management:** Distributed memory management involves managing memory across different machines to efficiently store and access data. Techniques like remote memory access and caching enhance memory management.
- **File System:** DOS supports distributed file systems that provide a unified view of files distributed across the network. Features include file sharing, replication, caching, and consistency maintenance.
- **Fault tolerance:** DOS employs fault tolerance techniques to ensure system reliability despite component failures. Redundancy, error detection, recovery, and replication are used to manage failures.
- **Real-time scalable system:** DOS may support real-time systems, which require timely responses to events with strict deadlines. Real-time scheduling and communication mechanisms are crucial in such scenarios. DOS is designed to scale horizontally (adding more machines) and vertically (adding more resources to a machine) to accommodate growing workloads.

JINI (Short notes):

Jini, also called Apache River, is a distributed computing technology that aims to simplify the development and deployment of networked services and devices. It allows devices and services to dynamically join and leave a network, providing a flexible and adaptable distributed environment.



The following points further describe JINI:

- **Dyanmic Federation:** Jini enables devices and services to join or leave a network dynamically without complex configuration or administration. This supports the creation of ad-hoc networks and simplifies resource discovery.
- **Service Oriented Architecture:** Jini is based on a service-oriented architecture where devices and services expose their capabilities as services. Clients can discover and use these services through a standardized interface.
- **Lookup services:** Jini uses lookup services to facilitate service discovery. Devices and services register themselves with a lookup service, and clients can find and use these services through the lookup service.
- **Service Interfaces:** Jini services expose their capabilities through standardized interfaces, known as service interfaces. Clients use these interfaces to interact with services. Service discovery involves finding available services in the network.
- **Leasing and Renewal:** Jini introduces the concept of leasing, where clients lease resources for a specific duration. Clients need to renew their leases periodically to continue using the resources. If a client fails to renew, the resource is released.
- **Transaction:** Jini supports distributed transactions and provides mechanisms for coordinating transactions across multiple services. It also includes persistence support, allowing stateful services to store their state reliably.
- **Network and Code Mobility:** Jini's dynamic federation and service discovery capabilities enable network mobility, where devices and services can move across different networks seamlessly. Jini supports code mobility, where code can be dynamically downloaded to clients or services when needed.
- **Collaboration and Flexibility:** Jini encourages collaboration between devices and services, promoting the creation of flexible and adaptable distributed systems.

33 TRIBHUVAN UNIVERSITY
 INSTITUTE OF ENGINEERING
Examination Control Division
 2072 Kartik

Exam.	New Back (2066 & Later Batch)		
Level	BE	Full Marks	80
Programme	BCT	Pass Marks	32
Year / Part	IV / 1	Time	3 hrs.

Subject: - Distributed System (CT703)

- ✓ Candidates are required to give their answers in their own words as far as practicable.
- ✓ Attempt All questions.
- ✓ The figures in the margin indicate Full Marks.
- ✓ Assume suitable data if necessary.

1. What is Distributed System? Discuss the challenges of Distribution System with example. [2+6]
2. Mention the role of IDL and middleware in Distributed System. Explain RMI approach in the distributed object based system. [4+6]
3. Define DFS. How does DFS encourage sharing a storage device? Explain with the help of suitable architecture. [8]
4. How threads differ from process? How does checkpoint help in recovery? What does distributed commit refer to? [4+2+2]
5. Define flat and nested transaction. Discuss the approach of optimistic concurrency control in distributed transactions. [4+6]
6. Why it is difficult to synchronize physical clock? Explain how clock synchronization can be solved using logical clock. [2+6]
7. What are the reasons for replicating the service provide? Discuss about fault tolerant services. [4+4]
8. How cascading aborts occurs and can be solved? Explain the needs and roles of atomic commit protocol in distributed system. [8]
9. Write short notes on: [4×3]
 - a) Christian's Algorithm
 - b) Recovery approach in Distributed System
 - c) CORBA services
 - d) Monolithic and Microkernel

Q. No. 1. Answer:

A distributed system refers to a collection of independent computers or nodes that work together as a single coherent system to achieve a common goal. In a distributed system, components are located on different machines and communicate and coordinate their actions through a network. This architecture is commonly used to improve performance, scalability, fault tolerance, and resource utilization. Distributed systems are prevalent in various applications, including cloud computing, large-scale data processing, content delivery networks, and more.

Challenges of Distributed Systems:

- i. **Network Communication:** Since components in a distributed system communicate over a network, network latency, bandwidth limitations, and potential network failures can significantly impact system performance and responsiveness. For example, in a distributed database system, slow network communication might result in delayed data retrieval and updates.
- ii. **Concurrency and Synchronization:** Multiple components may attempt to access shared resources concurrently. Ensuring data consistency and avoiding race conditions require sophisticated synchronization mechanisms. In distributed systems, managing synchronization becomes complex due to the lack of a shared memory space. For instance, in a distributed file system, multiple users might attempt to modify the same file simultaneously.
- iii. **Fault Tolerance:** Distributed systems must be designed to handle node failures gracefully without compromising the overall system's stability. This involves mechanisms like replication, fault detection, and recovery. For example, in a distributed web application, if one server fails, requests should automatically be redirected to other healthy servers.
- iv. **Consistency and Replication:** Maintaining data consistency across distributed nodes is a challenge, especially when data is replicated for improved availability. Striking a balance between consistency and availability, as exemplified in distributed databases, requires careful design and trade-offs.
- v. **Load Balancing:** Distributing the workload evenly across nodes to ensure efficient resource utilization is a key challenge. Load imbalances can lead to performance degradation and uneven resource utilization. In a distributed web service, if some servers are overloaded while others are underutilized, users may experience slow response times.
- vi. **Security and Privacy:** Securing communications, authenticating users, and protecting sensitive data in a distributed environment is complex. Ensuring that data is encrypted and unauthorized access is prevented across distributed components is a constant challenge. In a distributed healthcare system, patient records must be securely accessed only by authorized medical personnel.
- vii. **Coordination and Consensus:** Achieving consensus among distributed nodes is challenging due to network delays and potential failures. Distributed algorithms like the Paxos or Raft consensus algorithms are used to ensure that all nodes agree on a particular value or decision.
- viii. **Scalability:** Distributed systems should be able to handle increasing workloads by adding more nodes. However, adding nodes may introduce additional complexities in terms of communication

overhead and coordination. Scalability challenges are evident in large-scale online gaming systems that need to handle a growing number of players.

- ix. **Debugging and Monitoring:** Identifying and diagnosing issues in a distributed system can be complex due to the interactions between multiple components. Monitoring the system's health and performance across various nodes and ensuring quick identification of problems are critical. Debugging a distributed application like a microservices-based e-commerce platform can be daunting.
- x. **Legacy Integration:** Integrating new distributed components with existing legacy systems can be difficult due to differences in communication protocols, data formats, and technologies. Migrating from a monolithic architecture to a distributed one, as seen in enterprise software, requires careful planning and execution.

In conclusion, while distributed systems offer numerous advantages, they also pose significant challenges that require careful consideration during design, implementation, and maintenance. Overcoming these challenges involves a combination of architectural decisions, algorithmic design, and ongoing management to ensure the reliability, scalability, and performance of the distributed system.

Q. No. 2 Answer

In a distributed system, Interface Definition Language (IDL) and middleware play crucial roles in facilitating communication and interaction between distributed components. They help abstract the complexities of network communication and enable seamless interaction between distributed objects and services.

i. Interface Definition Language (IDL): IDL is a specification language used to define the interfaces of distributed objects or services in a language-neutral and platform-independent manner. It serves as a contract between different components, ensuring that they understand how to communicate with each other regardless of the programming languages or platforms they are implemented in. IDL typically includes information about data types, methods, and parameters that can be used to invoke remote services.

IDL plays a role in:

- **Enabling Interoperability:** By providing a common language for defining interfaces, IDL enables components written in different programming languages to communicate and collaborate seamlessly.
- **Generating Stubs and Skeletons:** IDL compilers generate code stubs on the client side and skeletons on the server side. Stubs help the client-side code communicate with remote services, while skeletons handle incoming requests on the server side.
- **Type Marshalling:** Distributed systems often involve passing data between different machines with varying data representations. IDL ensures proper data serialization and deserialization, ensuring that data can be transmitted and understood correctly across the network.

- **Decoupling Implementation:** IDL allows the interface to be defined separately from the implementation, promoting loose coupling between components. This enables changes to the implementation without affecting clients using the service.

ii. Middleware: Middleware is a layer of software that sits between the operating system and the application in a distributed system. It provides a set of services and abstractions that simplify the development, deployment, and management of distributed applications. Middleware handles communication, security, and various aspects of distributed computing, abstracting away the low-level network details.

Roles of middleware in distributed systems:

- **Communication Abstraction:** Middleware abstracts the complexities of network communication, allowing developers to interact with remote components using familiar programming constructs. It provides APIs and protocols for remote procedure calls (RPC), message passing, and other communication patterns.
- **Location Transparency:** Middleware provides location-independent addressing, allowing components to interact with remote services using logical addresses rather than dealing with physical network locations.
- **Scalability and Load Balancing:** Middleware often includes mechanisms for load balancing and managing the distribution of workloads across distributed nodes. This helps in improving system performance and resource utilization.
- **Security and Authentication:** Middleware offers security mechanisms to ensure secure communication and data protection between distributed components. It provides authentication, authorization, and encryption services.
- **Transaction Management:** Middleware can handle distributed transactions, ensuring that operations across multiple components maintain data consistency and integrity.
- **Error Handling and Fault Tolerance:** Middleware provides error handling and recovery mechanisms to manage failures and faults in distributed systems. It can handle issues like network failures, node crashes, and retries.
- **Concurrency and Synchronization:** Middleware offers synchronization and coordination mechanisms to manage concurrent access to shared resources across distributed nodes.

Remote Method Invocation (RMI) in Distributed Object-Based Systems:

RMI is an approach used in distributed object-based systems to enable communication and interaction between objects residing on different machines. It allows a client object to invoke methods on a remote object as if it were a local method call.

Working of RMI

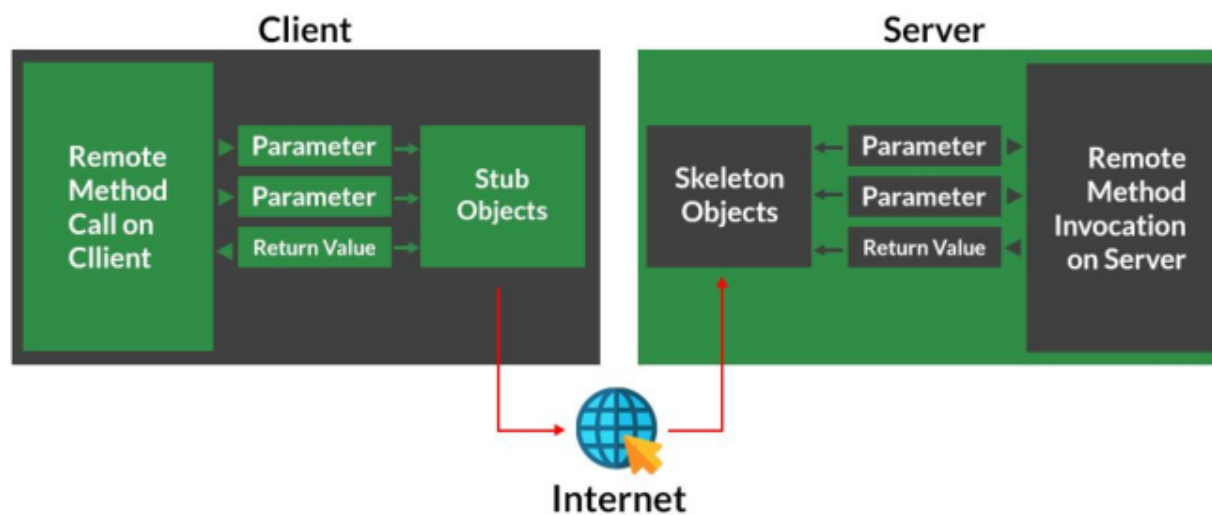


Figure 1 Demonstration of Working of RMI

Here's how RMI works:

- **Interface Definition:** The interface of the remote object is defined using an IDL, specifying the methods that can be invoked remotely.
- **Stubs and Skeletons:** The IDL compiler generates stubs on the client side and skeletons on the server side. Stubs act as proxies for the remote objects, intercepting method calls and transmitting them over the network. Skeletons receive incoming method calls, decode them, and invoke the actual methods on the server object.
- **Registry:** The RMI registry acts as a central repository for remote object references. Clients use the registry to look up the references of remote objects they want to interact with.
- **Marshalling and Unmarshalling:** Parameters and return values passed between client and server are marshalled (serialized) on the client side and unmarshalled (deserialized) on the server side. This ensures that data can be transmitted over the network in a platform-independent format.
- **Invocation:** The client-side stub marshals the method call parameters, sends them over the network to the server-side skeleton, which unmarshals the parameters, invokes the actual method on the server object, marshals the return value, and sends it back to the client.
- **Synchronization and Concurrency:** Middleware might provide mechanisms for synchronization and concurrency control to manage multiple clients interacting with remote objects concurrently.

RMI abstracts away the complexities of network communication and allows developers to work with remote objects in a manner similar to working with local objects, promoting a natural and intuitive programming experience in distributed systems.

Q. No. 3 Answer:

A Distributed File System (DFS) is a networked file system that allows multiple users and applications to access and share files and directories across a network of computers. It provides a unified and transparent view of storage resources distributed across different machines, enabling efficient data management, access, and collaboration in a distributed environment.

Encouraging Sharing with DFS: DFS encourages sharing a storage device by providing a consistent and shared file storage environment across multiple machines. This allows users and applications to access and manipulate files and directories as if they were stored locally, regardless of the physical location of the storage devices. The sharing is facilitated through a well-defined architecture that abstracts the complexities of network communication and distributed storage management.

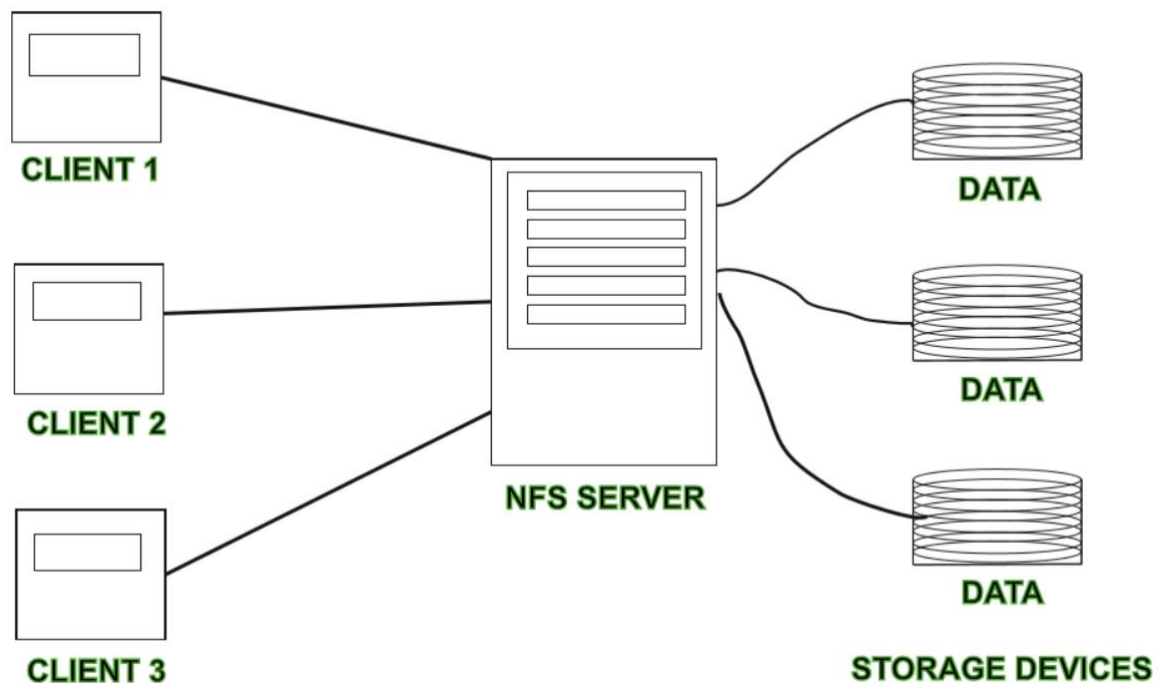


Figure 2 Architecture of Distributed File System

Architecture of DFS:

- i. **Client Machines:** These are the computers where users or applications initiate file access and manipulation requests. Clients interact with the DFS to perform operations on files, such as reading, writing, and deleting.
- ii. **DFS Servers:** These are the machines that host the actual storage devices and manage the distribution and replication of files. DFS servers provide access to files and handle file-related operations on behalf of clients.

- iii. **Namespace:** The namespace is a logical abstraction that defines the structure and hierarchy of files and directories in the DFS. It provides a consistent naming convention, allowing users and applications to refer to files using familiar paths.
- iv. **Metadata Servers:** Metadata servers store the metadata associated with files and directories in the DFS. Metadata includes information like file names, permissions, ownership, timestamps, and file structures. Metadata servers maintain the directory structure and keep track of where data blocks are stored.
- v. **Data Servers:** Data servers are responsible for storing the actual file data. Files are typically divided into smaller data blocks that are distributed across different data servers for better performance and fault tolerance.

DFS encourages sharing in the following ways:

- i. **Transparency:** DFS abstracts the underlying complexities of distributed storage and network communication. Clients interact with files using familiar file paths, regardless of the physical location of the data.
- ii. **Location Independence:** Clients do not need to know the exact location of files on the storage devices. The DFS handles the mapping between logical file paths and physical storage locations.
- iii. **Scalability:** DFS allows the addition of more storage devices and servers as needed, enabling seamless expansion of storage capacity without disrupting user access.
- iv. **Data Replication:** To enhance fault tolerance and availability, DFS can replicate data across multiple data servers. This ensures that even if one server fails, the data is still accessible from other servers.
- v. **Caching:** DFS may implement caching mechanisms to store frequently accessed data in client-side caches. This improves access speed and reduces network traffic.
- vi. **Concurrency Control:** DFS provides mechanisms for concurrent access to files by multiple clients. It manages locks and synchronization to ensure data consistency.
- vii. **Security:** DFS enforces access control policies, allowing administrators to set permissions and restrict user access to files and directories.
- viii. **Backup and Recovery:** DFS architecture often includes mechanisms for regular backups and efficient data recovery in case of failures.

In summary, a Distributed File System (DFS) encourages sharing a storage device by abstracting the complexities of distributed storage, providing a consistent naming convention, and managing metadata and data distribution across multiple servers. This architecture allows users and applications to access and manipulate files in a seamless and transparent manner, promoting efficient collaboration and resource utilization in a distributed environment.

Q. No. 4 Answer:

The differences between thread and processes are as follows:

Threads	Processes
Smallest unit of a process	Independent and complete execution unit
Less overhead as they share resources	More overhead due to separate resources
Easier communication due to shared memory	Requires inter-process communication mechanisms
Faster context switching	Slower context switching
Share memory and resources within the process	Have separate memory and resources
Less isolation, as threads share resources	High isolation, processes run independently
Typically, easier to scale due to shared state	Scaling can be complex due to separate states
Faster to create threads compared to processes	Processes take longer to create
One thread can impact the entire process	One process failure usually doesn't affect others
Harder to recover due to shared resources	Easier to recover due to isolation
Multiple threads in a web server	Separate processes for different applications

Checkpoint and Recovery: A checkpoint is a mechanism in a computing system that involves periodically saving the current state of a process or application. This state includes variables, data structures, and other relevant information. The purpose of checkpoints is to provide a point of recovery in case of system failures. If a failure occurs, the system can restore the application to a previously saved checkpoint state, reducing the amount of work lost due to the failure.

Distributed Commit: Distributed commit refers to the process of coordinating and ensuring that all distributed components involved in a distributed transaction either successfully complete their part of the transaction or none of them commit at all. In a distributed system, a transaction may involve multiple nodes or processes, and ensuring that all nodes agree to commit or abort a transaction is critical for maintaining data consistency and integrity.

Distributed commit involves several steps:

1. **Transaction Execution:** Each node performs its part of the transaction, which could involve updates to local and remote resources.
2. **Voting Phase:** Each participating node votes whether it is ready to commit the transaction or not. If any node encounters an issue or cannot commit, it votes to abort.

3. **Coordinator's Decision:** A central coordinator or a consensus algorithm evaluates the votes. If all votes are to commit, the coordinator decides to commit. If any vote is to abort, the coordinator decides to abort.
4. **Commit/Abort Phase:** Based on the coordinator's decision, each node performs the actual commit or abort action.

Distributed commit is essential to ensure that even in the face of failures or network issues, the system maintains data integrity and consistency. If any node fails during the commit process, recovery mechanisms are employed to bring the system back to a consistent state.

Q. No. 5 Answer:

Flat transactions, also referred to as simple transactions, are self-contained units of work in transaction processing where no sub-transactions are included. They represent discrete operations that are executed as a single entity and are either fully committed or entirely rolled back. This straightforward structure facilitates the management of individual transactions.

In contrast, nested transactions introduce a hierarchical structure by allowing transactions to encompass sub-transactions. Each sub-transaction is treated as an independent unit of work within the scope of the parent transaction. The outcome of a nested transaction can influence the overall outcome of the parent transaction, providing a more granular and organized approach to complex operations.

Optimistic concurrency control is an advanced strategy employed in distributed transaction systems to manage concurrent access to shared resources with minimized contention and locking. The approach is grounded in the premise that conflicts between transactions are infrequent and that most transactions can proceed independently.

The workflow of optimistic concurrency control follows several key stages:

- i. **Transaction Execution:** A transaction initiates by reading required data without immediate acquisition of locks. It then performs its operations, assuming independence from other concurrent transactions.
- ii. **Validation Phase:** Upon completion of operations, the transaction seeks to commit. Before finalizing the commit, it undergoes a validation step where it verifies whether any conflicting transactions have altered the data it accessed.
- iii. **Validation Check:** The validation procedure involves a comparison of the data read during the transaction's execution against the current state of the system. If conflicting changes are detected, the validation fails.
- iv. **Commit or Abort:** Successful validation allows the transaction to proceed with committing its changes. If validation fails due to conflicts, the transaction is aborted.
- v. **Retrying or Rolling Back:** In case of validation failure, the transaction can be retried by revisiting its data and resolving conflicts. Alternatively, the transaction can be rolled back to maintain data consistency.

Advantages and Challenges of Optimistic Concurrency Control:

Advantages:

- **Reduced Locking:** Optimistic control diminishes lock usage, curbing contention and ameliorating system performance.
- **Elevated Throughput:** Independent transaction execution escalates throughput, surpassing conventional locking-based methods.
- **Scalability Enhancement:** Optimistic control aligns with distributed systems, alleviating cross-node lock management complexities.
- **Deadlock Mitigation:** By not holding locks throughout execution, the risk of deadlocks is mitigated.

Challenges:

- **Validation Overhead:** Validation phase can introduce additional overhead, particularly when multiple transactions access the same data.
- **Rollback Management:** Handling aborted transactions and their ramifications necessitates meticulous management.
- **Limited Applicability:** Optimistic control may not be universally suitable, particularly for transactions exhibiting high contention.

In summation, optimistic concurrency control provides an effective mechanism for orchestrating concurrent access within distributed transactions. It combines minimized locking with efficient throughput, though challenges in validation and rollback management must be addressed for optimal implementation.

Q. No. 6 Answer:

Synchronizing physical clocks across distributed systems presents challenges due to factors like network delays, clock drift, and varying clock speeds among different machines. Achieving precise and consistent clock synchronization in such an environment is intricate. However, logical clocks provide an effective solution to address these challenges and achieve a notion of synchronized time within distributed systems.

Difficulty in Synchronizing Physical Clocks:

Synchronizing physical clocks across distributed systems is complex due to several reasons:

- i. **Network Delays:** Network latencies and varying communication delays introduce discrepancies in clock readings between different machines.
- ii. **Clock Drift:** Inherent inaccuracies in physical clocks cause them to drift over time, leading to inconsistencies in timekeeping.
- iii. **Clock Skew:** Different machines have distinct clock speeds, causing the clocks to drift apart at varying rates.

- iv. **Asymmetric Communication:** Unequal communication times for messages lead to asymmetrical clock adjustments.

Clock Synchronization using Logical Clocks:

Logical clocks provide a solution to clock synchronization challenges by focusing on the order of events rather than achieving globally synchronized physical time. Logical clocks assign logical timestamps to events based on causality relationships, allowing consistent ordering of events across distributed systems. Two well-known algorithms for logical clocks are Lamport's Logical Clocks and Vector Clocks.

1. Lamport's Logical Clocks:

Lamport's Logical Clocks algorithm is a fundamental approach to capturing causality relationships and achieving event ordering in distributed systems. This algorithm addresses the challenge of synchronizing physical clocks by prioritizing the sequence of events rather than striving for perfect time synchronization.

Algorithm Description:

- i. Each process maintains its own logical clock, initially set to zero.
- ii. When an event occurs within a process, the process increments its logical clock by one, generating a timestamp for the event.
- iii. When a message is sent from one process to another, the sender includes its current logical clock value in the message.
- iv. Upon receiving a message, the receiver updates its logical clock to the maximum of its current value and the received timestamp. This adjustment ensures that causally related events are correctly ordered.
- v. The logical clock values for each process capture the progression of events in a causally consistent manner, allowing events to be arranged based on their logical timestamps.

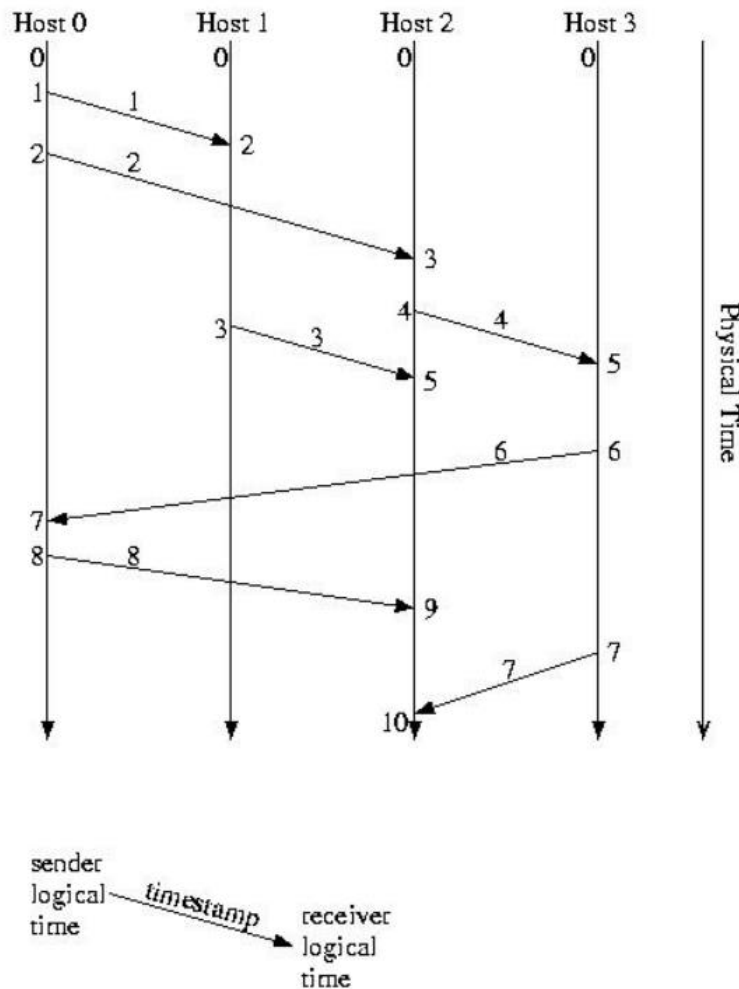


Figure 3 Demonstration of Lamport's Logical Clock Algorithm

Advantages and Limitations:

- **Advantages:** Lamport's Logical Clocks algorithm provides a simple and intuitive approach to ordering events based on causality. It ensures that events that are causally related are always correctly ordered.
- **Limitations:** The algorithm assumes that the events are causally related if they are connected by message sends and receives. However, events that are not directly related by messages may still have an ordering ambiguity.

2. Vector Clocks:

Vector Clocks extend the concept of logical clocks by considering causal relationships among events involving multiple processes. This algorithm addresses the limitations of Lamport's algorithm by maintaining additional context about the causal history of events across processes.

Algorithm Description:

- i. Each process maintains a vector clock, initially set to zero for all processes. The size of the vector corresponds to the number of processes in the system.
- ii. When an event occurs within a process, the process increments its own vector clock element by one.
- iii. When a message is sent from one process to another, the sender includes its entire vector clock in the message.
- iv. Upon receiving a message, the receiver updates its vector clock elements to the maximum of its own elements and the corresponding elements in the received vector clock. This adjustment captures the causal history of events involving different processes.
- v. The vector clock values for each process provide a comprehensive view of the causal relationships across processes, enabling accurate event ordering.

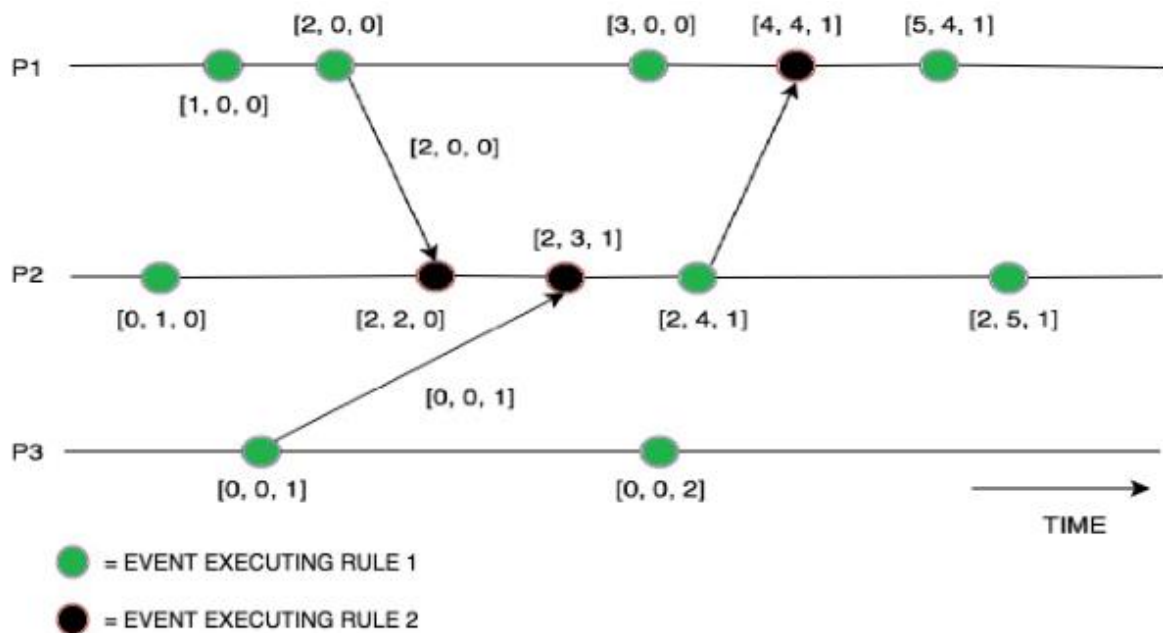


Figure 4 Demonstration of Vector Clock Algorithm

Advantages and Limitations:

- **Advantages:** Vector Clocks address the limitations of Lamport's algorithm by providing a more nuanced understanding of causal relationships across processes. They allow for accurate event ordering in scenarios where events are distributed across multiple processes.
- **Limitations:** Vector Clocks still rely on the assumption that message sends and receives establish causal relationships. They require knowledge of the total number of processes in the system, which can be challenging to maintain dynamically.

Advantages of Logical Clocks:

- **Causality Preservation:** Logical clocks ensure that causally related events are correctly ordered, irrespective of clock inaccuracies.
- **Event Ordering:** Logical clocks provide a consistent way to order events, facilitating accurate understanding of the sequence of events in distributed systems.
- **Decentralized:** Logical clocks do not require global synchronization and are based on local observations of events, making them suitable for distributed environments.

In conclusion, synchronizing physical clocks in distributed systems is challenging due to network delays, clock drift, and varying speeds. Logical clocks offer a solution by focusing on event ordering rather than achieving precise physical time synchronization. Algorithms like Lamport's Logical Clocks and Vector Clocks ensure consistent event sequencing while addressing the complexities inherent in physical clock synchronization.

Q. No. 7 Answer:

Replicating services is a fundamental technique in distributed systems that involves creating duplicate instances of a service or application across multiple nodes. This approach addresses the challenges posed by potential failures and contributes to improved availability, reliability, and performance of services.

Reasons for Replicating Services:

- Fault Tolerance:** One of the primary motivations for replicating services is to enhance fault tolerance. By maintaining multiple copies of a service, the system can continue to function even if one or more replicas fail due to hardware failures, software bugs, or other issues. If one replica becomes unavailable, clients can be seamlessly directed to a healthy replica, minimizing service disruptions.
- Load Balancing:** Replicating services aids in distributing incoming requests evenly among available replicas. This load balancing strategy prevents a single instance from being overwhelmed by high traffic and ensures that resources are efficiently utilized. Load balancing improves response times and prevents performance degradation during periods of high demand.
- Geographical Distribution:** Replicating services across different geographical locations allows users to access services from a nearby replica, reducing network latency and enhancing the user experience. This approach is particularly crucial for applications that require low-latency interactions, such as real-time communication or content delivery.
- Scalability:** Replication supports horizontal scalability, allowing the system to accommodate growing workloads by adding more replicas. This dynamic scaling ensures that the system can handle increased demand without compromising performance.
- High Availability:** Replication contributes to high availability by ensuring that there are multiple points of access to the service. This availability is especially important for critical applications that require constant access, such as online banking or e-commerce platforms.

- vi. **Redundancy:** Replicating services creates redundancy, which acts as a safety net in case of unexpected failures. Redundancy safeguards against single points of failure and reduces the likelihood of service outages.
- vii. **Resilience to Updates:** Replication facilitates seamless updates and maintenance of services. During updates, one replica can be taken offline while the others continue to provide service, ensuring continuous availability.
- viii. **Data Locality:** In scenarios where data needs to be stored and accessed locally, replicating services with localized data ensures faster access times and minimizes data transfer over networks.
- ix. **Parallel Processing:** Replicated services can process requests in parallel across multiple instances, enabling faster data processing and computation.

Challenges and Considerations:

- **Data Consistency:** Maintaining consistent data across replicas can be complex, especially in scenarios where updates need to be synchronized.
- **Conflict Resolution:** In distributed systems, conflicts may arise when different replicas update the same data simultaneously. Effective conflict resolution mechanisms are necessary to ensure data integrity.
- **Replication Overhead:** Replicating services introduces overhead in terms of network communication, data synchronization, and resource utilization.
- **Synchronization:** Ensuring synchronization between replicas requires careful design to avoid issues like split-brain scenarios.
- **Cost:** Running multiple replicas incurs costs related to hardware, maintenance, and operational management.

Thus, replicating services is a strategic approach that addresses challenges related to fault tolerance, load balancing, geographical distribution, scalability, and high availability in distributed systems. By maintaining duplicate instances of services, organizations can deliver reliable, responsive, and high-performance applications to users, even in the face of failures and fluctuations in demand. While replication introduces complexities, the benefits in terms of improved service quality and user experience make it an indispensable technique in modern distributed computing environments.

Fault tolerance refers to a system's ability to continue functioning even in the presence of failures or faults. Replicating services is a common strategy to achieve fault tolerance in distributed systems. Fault-tolerant services aim to provide consistent and reliable functionality despite hardware failures, software bugs, or other unforeseen issues.

Approaches to Achieving Fault Tolerant Services:

- i. **Active Replication:** In active replication, multiple copies of the service run concurrently, and all incoming requests are executed by each replica. The results are compared to detect faults, and

only correct responses are returned to clients. This approach ensures high reliability but may introduce overhead due to multiple executions.

- ii. **Passive Replication:** In passive replication, only one replica, called the primary, processes incoming requests, while the others, called backups or standbys, remain idle. If the primary fails, a standby is promoted to become the new primary. Passive replication minimizes overhead but may introduce a delay in failover.
- iii. **Stateful vs. Stateless Replicas:** Stateful replicas maintain their own local states, which are synchronized with the primary. Stateful replicas are used when the service has an internal state that needs to be preserved. Stateless replicas do not maintain internal states and can quickly take over the primary's responsibilities.
- iv. **Quorum-based Approaches:** Quorum-based protocols ensure consistency and fault tolerance by requiring a certain number of replicas to agree on a decision. This approach prevents split-brain situations and ensures that only a correct replica is chosen to take over.

Benefits of Fault Tolerant Services:

- **Continuous Availability:** Fault-tolerant services maintain their functionality even during failures, preventing service downtime.
- **Data Integrity:** Replicating data ensures that data remains consistent and accurate across multiple copies.
- **Disaster Recovery:** In case of data center failures or natural disasters, backup replicas can be brought online to ensure service continuity.
- **User Experience:** Fault tolerance leads to a smoother user experience, as users encounter minimal disruptions or delays due to failures.

Challenges:

- **Consistency:** Ensuring consistent data across replicas and handling updates or changes can be complex.
- **Overhead:** Replicating and synchronizing data introduces network and computational overhead.
- **Conflict Resolution:** Handling conflicts when replicas have divergent states requires careful design and coordination.
- **Complexity:** Implementing and managing fault-tolerant services can be intricate and demand additional resources.

In conclusion, replicating services is a crucial strategy to achieve fault tolerance, enhance availability, and improve performance in distributed systems. By maintaining multiple copies of services and employing various replication strategies, systems can mitigate the impact of failures and provide reliable services to users.

Q. No. 8 Answer:

Cascading aborts, also known as the "domino effect," occur in distributed systems when the failure of one transaction triggers a chain reaction of aborts in dependent or subsequent transactions. This can lead to a widespread impact on the system's stability, performance, and data integrity.

Causes of Cascading Aborts:

- i. **Dependency Chains:** In a distributed environment, transactions often have dependencies on one another. When a transaction fails and is aborted, the changes made by that transaction may have already been propagated to other transactions that depended on it. These dependent transactions might no longer be valid due to the failure of the initial transaction, leading to their own aborts.
- ii. **Inconsistent State:** If a failed transaction modifies shared data before it is aborted, the data might be left in an inconsistent or incorrect state. Subsequent transactions that depend on this modified data can encounter errors or conflicts, resulting in their own aborts.

In-depth Solutions to Cascading Aborts:

- i. **Isolation and Consistency:** Ensuring strong isolation between transactions using techniques like multi-version concurrency control (MVCC) can prevent cascading aborts. MVCC allows each transaction to work on a separate version of the data, eliminating interference from other transactions' changes.
- ii. **Transaction Prioritization:** Prioritize the execution of dependent transactions over independent ones. This way, if a transaction is likely to fail, its dependent transactions are not executed, preventing cascading effects.
- iii. **Rollback Segmentation:** Instead of completely rolling back a transaction, consider segmenting the rollback. Only the parts of the transaction affected by the failure are rolled back, minimizing the impact on other transactions.
- iv. **Deadlock Detection and Resolution:** A well-designed deadlock detection and resolution mechanism can prevent transactions from waiting indefinitely due to cascading aborts caused by deadlock scenarios.
- v. **Retry and Compensation:** If a transaction fails due to a temporary issue, it can be retried after the issue is resolved. Compensation mechanisms can be employed to reverse any partial effects of the failed transaction.
- vi. **Consistent Recovery Mechanisms:** In scenarios where a failure occurs during a transaction's execution, having mechanisms in place to recover to a consistent state can prevent cascading effects on dependent transactions.
- vii. **Advanced Locking and Synchronization:** Use advanced locking mechanisms that allow finer control over data access and prevent multiple transactions from accessing the same data simultaneously.

- viii. **Snapshot Isolation:** Use snapshot isolation techniques that allow each transaction to work with a consistent snapshot of the database, ensuring that changes made by a failed transaction are not propagated to other transactions.
- ix. **Deterministic Execution:** Ensure that transactions are executed deterministically, regardless of the order of execution. This prevents variations in execution that could lead to different outcomes and cascading effects.

Benefits of Solving Cascading Aborts:

- **Data Integrity:** Preventing cascading aborts maintains data integrity and consistency across the system.
- **Stability:** By avoiding the domino effect, the stability of the distributed system is enhanced, leading to more reliable and predictable behavior.
- **Efficiency:** Preventing cascading aborts reduces unnecessary work, resource usage, and system downtime caused by aborting valid transactions.

Challenges:

- Implementing effective solutions requires careful design and coordination, as the interdependencies between transactions can be complex.
- Some solutions might introduce overhead in terms of performance or complexity, especially in scenarios with high transaction rates.

In conclusion, cascading aborts pose a significant challenge in distributed systems, affecting data consistency and system stability. Addressing cascading aborts demands a combination of proper isolation techniques, prioritization, rollback segmentation, and consistent recovery mechanisms. By understanding the causes and applying in-depth solutions, distributed systems can effectively prevent the domino effect of transaction failures.

The atomic commit protocol is a fundamental mechanism used in distributed systems to ensure that a group of transactions either all commit or all abort, maintaining data consistency and integrity across multiple nodes. In scenarios where multiple transactions need to be coordinated and committed together, the atomic commit protocol plays a crucial role.

Needs of Atomic Commit Protocol:

- i. **Data Consistency:** To maintain data consistency across distributed systems, it's essential that a group of related transactions commit together or none of them commit at all.
- ii. **Fault Tolerance:** The atomic commit protocol ensures that even if a node or participant fails during the commit process, the system can recover without causing inconsistencies.
- iii. **Coordination:** When transactions span multiple nodes, ensuring that they all reach a consensus on whether to commit or abort is vital for proper operation.

Roles of Atomic Commit Protocol:

- i. **Coordinator:** The coordinator initiates the atomic commit process and sends commit or abort requests to all participants.
- ii. **Participants:** Participants are nodes or processes involved in the distributed transaction. They respond to the coordinator's request with their readiness to commit or abort.
- iii. **Voting Phase:** During the voting phase, participants respond to the coordinator's request with a vote to commit or abort. This phase ensures that all participants are ready to proceed.
- iv. **Decision Phase:** Based on participants' votes, the coordinator makes a decision to commit or abort the transaction. If any participant votes to abort, the coordinator decides to abort.
- v. **Commit Phase:** If the coordinator decides to commit, it sends a commit request to all participants. Participants then finalize their transactions and release any locks.
- vi. **Abort Phase:** If the coordinator decides to abort, it sends an abort request to all participants. Participants undo their changes and release any locks acquired during the transaction.

Benefits of Atomic Commit Protocol:

- **Data Integrity:** The atomic commit protocol ensures that transactions are committed consistently, preventing data inconsistencies.
- **Coordination:** It provides a mechanism for coordinating multiple transactions across distributed nodes.
- **Fault Tolerance:** The protocol handles failures in participants or the coordinator, allowing recovery without leaving the system in an inconsistent state.

Challenges:

- **Two-Phase Commit Blocking:** The protocol might block indefinitely if the coordinator crashes after the voting phase but before the decision phase. This issue is addressed by protocols like Three-Phase Commit.
- **Performance Overhead:** The protocol introduces communication overhead and can affect performance, especially in scenarios with high contention or network latency.

In conclusion, cascading aborts can be prevented through proper isolation, rollback segmentation, timeouts, and retry mechanisms. The atomic commit protocol ensures that transactions are consistently committed or aborted in distributed systems, addressing the needs for data consistency, fault tolerance, and coordination across multiple participants.

Q. No. 9. Answer

a. Christian's Algorithm

Christian's algorithm is a time synchronization algorithm designed for distributed systems. It was proposed by Gerard J. Holzmann in 1989. The algorithm is used to synchronize the clocks of multiple computers or

nodes in a network, ensuring that they have a consistent notion of time. It's particularly useful in scenarios where precise time synchronization is necessary for coordinated actions or accurate event ordering.

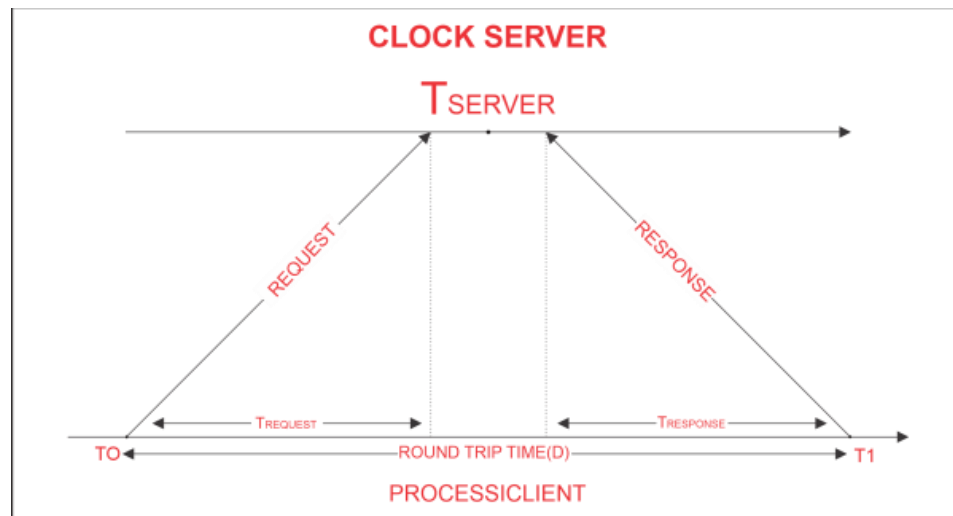


Figure 5 Cristian's Algorithm

Algorithm Overview:

Christian's algorithm is relatively simple and relies on the concept of estimating network delays to achieve clock synchronization. The algorithm involves two main entities: the time requester (client) and the time server.

- i. The time requester sends a request for the current time to the time server.
- ii. Upon receiving the request, the time server records its current time and sends the response back to the time requester.
- iii. When the time requester receives the response, it notes the time it received the response.
- iv. The time requester calculates the round-trip time by subtracting the time it sent the request from the time it received the response. Since the response contains the time according to the time server, the halfway point between sending the request and receiving the response represents the most accurate time at which the request was initiated.
- v. The time requester then estimates the clock offset by taking half of the round-trip time and subtracting it from the time the response was received.

Advantages:

- i. **Simple:** Christian's algorithm is straightforward to implement and doesn't require specialized hardware.
- ii. **Decentralized:** The algorithm doesn't rely on a central time server, which can be useful in scenarios where such a server might be a single point of failure.
- iii. **Reasonably Accurate:** The algorithm can achieve reasonable accuracy in terms of time synchronization, especially when network delays are consistent.

Limitations:

- i. **Network Variability:** The accuracy of the algorithm heavily depends on the assumption that network delays are consistent and symmetric. In real-world networks, delays can vary due to congestion, routing changes, and other factors.
- ii. **Clock Drift:** The algorithm doesn't account for clock drift, which can lead to time discrepancies over extended periods.
- iii. **Latency Impact:** The algorithm requires the round-trip time, which can be affected by network latency and other factors, impacting the accuracy of the synchronization.

While Cristian's algorithm provides a basic approach to clock synchronization, modern distributed systems often use more sophisticated algorithms, such as the Network Time Protocol (NTP) or the Precision Time Protocol (PTP), which account for factors like network variability, clock drift, and scalability to achieve higher levels of time synchronization accuracy.

b. Recovery Approach in Distributed System

In distributed systems, recovery approaches are strategies and techniques used to restore the system to a consistent and operational state after failures or faults have occurred. Recovering from failures is crucial to ensure data integrity, system availability, and continued functionality. There are several recovery approaches in distributed systems, each tailored to address different types of failures and scenarios. Here are some common recovery approaches:

- i. **Checkpoint and Rollback:**
 - Checkpointing involves periodically saving the system's state, including process states and data, to stable storage.
 - If a failure occurs, the system can be restored to a consistent state by rolling back to the most recent checkpoint and replaying the operations that occurred since that checkpoint.
 - This approach is effective for transient failures but can be resource-intensive due to the need for frequent checkpoints.
- ii. **Process Migration:**
 - In the event of a failure, a process can be migrated from a failed node to a healthy one.
 - This approach requires maintaining consistent process states and communication channels to facilitate seamless migration.
 - Process migration can be useful for load balancing and maintaining service availability.
- iii. **Replication and Redundancy:**
 - Replicating data and services across multiple nodes ensures that if one node fails, another can take over.

- Replication introduces redundancy and can be paired with mechanisms like quorum-based decision-making to ensure consistent data even in the presence of failures.

iv. **Message Logging and Replay:**

- Log messages and operations as they occur to a durable storage.
- In case of a failure, replay the logged messages and operations to restore the system's state to a consistent point.
- This approach helps ensure that no operation is lost even if a node fails.

v. **Quorum-Based Techniques:**

- Quorum-based protocols ensure that a certain number of nodes must agree for a decision (commit or abort) to be made.
- Quorum techniques help prevent split-brain situations and maintain data consistency during failures.

vi. **Shadow Copies:**

- Maintain a shadow copy of a resource or system that is updated concurrently.
- In case of failure, the system can switch to the shadow copy, ensuring minimal downtime.

vii. **Reconfiguration:**

- Dynamically reconfigure the system by adjusting the roles of nodes or redistributing tasks to healthy nodes.
- Reconfiguration can help maintain system functionality while isolating and recovering from failures.

viii. **Replication with Lazy Replication or Eager Replication:**

- In a replicated environment, lazy replication defers updates until they are absolutely necessary, while eager replication updates replicas as soon as the original copy is updated.
- These approaches balance between data consistency and performance.

ix. **Atomic Commit Protocols:**

- Ensure that a group of transactions either all commit or all abort, maintaining data integrity across multiple nodes.
- Atomic commit protocols help recover from scenarios where distributed transactions need coordinated actions.

x. **Recovery Blocks:**

- Divide the distributed system into recovery blocks that are independently recoverable.
- This approach simplifies recovery by minimizing the scope of impact during a failure.

Each recovery approach comes with its own advantages and challenges. The choice of approach depends on factors such as the nature of the system, the types of failures expected, the required level of consistency, and the desired trade-off between resource utilization and recovery time. An effective recovery strategy should balance these considerations to ensure the system's robustness and ability to handle failures while minimizing disruptions.

c. CORBA Services

CORBA (Common Object Request Broker Architecture) is a middleware technology that facilitates communication and interaction between distributed objects in a networked environment. CORBA services refer to a set of standardized services provided by the CORBA framework to handle various aspects of distributed computing, such as communication, naming, time, security, and more. These services help developers build distributed applications by abstracting and simplifying complex tasks.

Here are some of the CORBA services:

i. **Naming Service:**

- Provides a way to locate distributed objects in a network using human-readable names.
- Allows clients to find objects by their names without knowing their network addresses.
- Enables object location transparency.

ii. **Trading Service:**

- Enables objects to register their availability and capabilities in a directory service.
- Clients can query the directory to find objects that offer specific services.
- Facilitates dynamic discovery of services in a distributed environment.

iii. **Event Service:**

- Provides a mechanism for asynchronous communication between objects.
- Objects can publish events, and other objects can subscribe to those events.
- Enables efficient and decoupled communication in distributed systems.

iv. **Notification Service:**

- Similar to the Event Service, but focuses on notifications and alarms.
- Allows objects to send notifications to interested parties about certain events or conditions.

v. **Concurrency Control Service:**

- Provides mechanisms for managing concurrent access to shared resources in a distributed environment.

- Helps prevent data corruption and inconsistencies due to concurrent access.
- vi. **Externalization Service:**
- Enables objects to be converted to a platform-independent format for communication and storage.
 - Ensures that objects can be exchanged between different programming languages and platforms.
- vii. **Persistence Service:**
- Allows objects and their states to be saved and retrieved from a persistent storage system.
 - Ensures that objects maintain their state across system restarts or crashes.
- viii. **Security Service:**
- Provides authentication and authorization mechanisms to ensure secure communication between distributed objects.
 - Supports secure authentication, access control, and data encryption.
- ix. **Transaction Service:**
- Provides a way to manage distributed transactions involving multiple objects.
 - Ensures that a group of operations either all commit or all abort to maintain data consistency.
- x. **Time Service:**
- Provides a consistent notion of time across distributed systems.
 - Ensures that timestamps and time-related operations are synchronized among objects.

CORBA services are defined in the CORBA specification and are accessible through interfaces and APIs. They abstract complex distributed computing concerns, allowing developers to focus on building applications without having to implement low-level communication, security, and coordination mechanisms. The standardized nature of CORBA services promotes interoperability among different CORBA-compliant systems and simplifies the development of distributed applications.

d. Monolithic and Microkernel

A monolithic kernel is an operating system design where the entire operating system, including essential functions like memory management, file system management, device drivers, and system calls, is integrated into a single large and tightly interconnected executable binary. In a monolithic kernel architecture, all services and functionalities run in a single address space, which makes communication between different components efficient but also tightly coupled.

Advantages of a Monolithic Kernel:

- i. **Efficiency:** Since all components reside in the same address space, function calls and data sharing between components are relatively fast.
- ii. **Simplicity:** Developing a monolithic kernel is often simpler because the entire system is contained within a single codebase.
- iii. **Performance:** Direct communication and interaction between components result in lower overhead compared to inter-process communication in a microkernel.
- iv. **Resource Sharing:** Components can easily share data structures and resources, leading to efficient resource management.

Disadvantages of a Monolithic Kernel:

- i. **Limited Modularity:** Due to the tightly integrated nature of components, modifying or adding functionality can be complex and risky.
- ii. **Reliability:** A bug or failure in any component can potentially crash the entire system, reducing overall reliability.
- iii. **Scalability:** As the system grows, managing a monolithic kernel can become more challenging, leading to potential performance bottlenecks.
- iv. **Security:** Security vulnerabilities in one component can potentially compromise the entire system's security.

Microkernel:

A microkernel is an alternative operating system architecture where the core functionality of the operating system is kept minimal, with only essential services like process scheduling, memory management, and inter-process communication (IPC) handled in the kernel. Other non-essential services, such as file systems, device drivers, and networking protocols, are implemented as separate user-space processes. This design aims to achieve modularity and separation of concerns, allowing for easier development, maintenance, and extensibility.

Advantages of a Microkernel:

- i. **Modularity:** Microkernels promote modular design by separating non-essential services from the kernel, making the system easier to modify and extend.
- ii. **Reliability:** With a smaller kernel, the potential impact of a kernel failure is minimized, improving overall system reliability.
- iii. **Flexibility:** Adding or updating individual components is easier in a microkernel architecture, reducing the risk of affecting the entire system.
- iv. **Security:** A smaller trusted computing base (the microkernel) can improve system security by limiting the attack surface.
- v. **Portability:** Microkernels are often more portable since much of the system's functionality is implemented in user-space components.

Disadvantages of a Microkernel:

- i. **Performance Overhead:** The need for inter-process communication between user-space components can introduce performance overhead compared to direct function calls in a monolithic kernel.
- ii. **Complexity of IPC:** Implementing efficient inter-process communication mechanisms can be challenging and can introduce overhead.
- iii. **Limited Efficiency:** Critical system services, such as device drivers, may have reduced performance due to the context switches required for user-space communication.
- iv. **Resource Management:** Some essential services, such as memory management, may need to be implemented in both the kernel and user space, potentially increasing complexity.

In summary, the choice between a monolithic kernel and a microkernel depends on factors such as performance requirements, modularity, reliability, and security. Monolithic kernels offer efficiency and simplicity but can be less modular and secure. Microkernels emphasize modularity and security but can introduce overhead due to inter-process communication. Modern variations, such as hybrid kernels, aim to combine the advantages of both architectures to find a suitable balance for specific use cases.