Distributed System Solutions

1) **A. Define Distributed Systems. List and explain the significant consequences of Distributed Systems.**

Ans. A **distributed system** is defined as a system in which hardware or software components located at networked computers communicate and coordinate their actions solely by passing messages. This definition encompasses a wide range of systems where networked computers can be deployed, regardless of their physical separation, which can range from being in the same room to being on different continents

Significant Consequences of Distributed Systems:

- **Geographical Distribution**: Components of a distributed system can be located anywhere in the world, allowing for flexibility in resource allocation and usage. This can enhance performance and availability but may also introduce latency due to distance
- **Resource Sharing**: Distributed systems enable the sharing of resources such as hardware (e.g., printers, disks) and software (e.g., databases, files). This facilitates collaboration and efficiency, as resources can be utilized by multiple users or applications
- **Scalability**: Distributed systems can be designed to scale easily. As the number of users or resources increases, the system can be expanded without significant redesign, allowing for growth in demand
- **Heterogeneity**: A distributed system can consist of various types of hardware, operating systems, and programming languages. This diversity can enhance functionality but also poses challenges in ensuring compatibility and communication among different components
- **Failure Handling**: Distributed systems must be designed to handle failures gracefully. Since components can fail independently, the system should be resilient and able to recover from such failures without significant disruption to overall service
- **Concurrency**: Multiple processes can operate simultaneously in a distributed system, which can improve performance but also requires careful management to avoid conflicts and ensure data consistency
- **Transparency**: Ideally, a distributed system should hide the complexities of its distributed nature from users. This includes transparency in access, location, migration, replication, and concurrency, making the system easier to use
- **Quality of Service**: Ensuring a consistent quality of service is crucial in distributed systems, as variations in performance can affect user experience. This includes managing latency, bandwidth, and reliability.

B. Discuss the key challenges in the Distributed Systems.

Ans. Key challenges in distributed systems include:

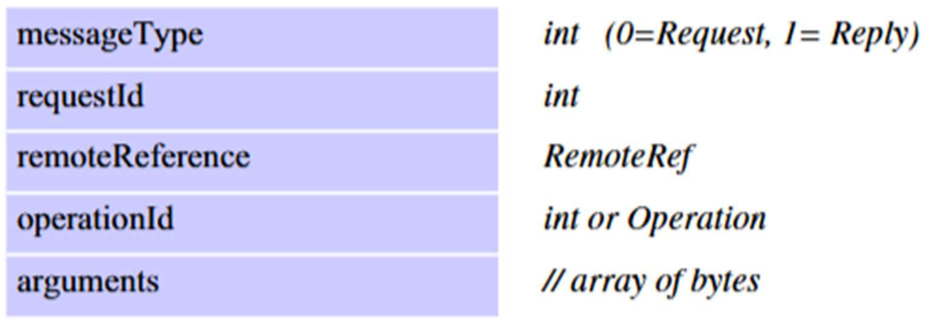
- **Network Reliability**: Communication between distributed components relies on the network, which can be unreliable. Network failures can disrupt communication, leading to inconsistencies and requiring robust error handling mechanisms.
- **Data Consistency**: Ensuring that all nodes in a distributed system have a consistent view of the data is challenging, especially when multiple nodes can read and write data simultaneously. Techniques like distributed transactions and consensus algorithms are often required to maintain consistency.
- **Scalability**: As the number of nodes and users increases, the system must scale efficiently. This involves not only adding more resources but also managing the complexity that comes with a larger system.
- **Security**: Protecting data and resources in a distributed environment is complex due to the potential for unauthorized access and attacks. Implementing security measures like encryption, authentication, and access control is essential.
- **Latency**: The physical distance between nodes can introduce latency, affecting the performance of the system. Minimizing latency while maintaining reliability and consistency is a significant challenge.
- **Fault Tolerance**: Distributed systems must be designed to continue functioning even when some components fail. This requires redundancy, replication, and effective recovery strategies.
- **Heterogeneity**: The diversity of hardware, operating systems, and network protocols can complicate the design and implementation of distributed systems. Ensuring interoperability among different components is a key challenge.
- **Concurrency Control**: Managing concurrent access to shared resources is crucial to prevent conflicts and ensure data integrity. Techniques such as locking, versioning, and timestamps are often employed.

C. Explain the Reply-Request Protocol used in the Distributed Systems for process communication.

Ans. The **Request-Reply Protocol** in distributed systems is a fundamental communication mechanism designed to facilitate interactions between clients and servers. It typically involves a synchronous communication model where a client sends a request to a server and waits for a reply. Here's how it works:

Key Components of the Request-Reply Protocol:

- **Communication Primitives**:
  - **doOperation**: This method is invoked by the client to send a request to the server. It includes parameters such as the server's reference, the operation to be performed, and any necessary arguments. The client is blocked until a reply is received.
  - **getRequest**: This method is used by the server to receive incoming requests from clients. It waits for a request message to arrive.
  - **sendReply**: After processing the request, the server uses this method to send a response back to the client, including the results of the operation.
- **Message Structure**: Each message exchanged in the protocol typically contains:



messageType	*int (0=Request, 1= Reply)*
requestId	*int*
remoteReference	*RemoteRef*
operationId	*int or Operation*
arguments	*// array of bytes*

  - **messageType**: Indicates whether the message is a request or a reply.
  - **requestId**: A unique identifier for the request, allowing the client to match replies with requests.
  - **remoteReference**: A reference to the remote server.
  - **operationId**: Identifies the specific operation to be executed.
  - **arguments**: An array of bytes containing the necessary data for the operation.
- **Synchronous Communication**: In the standard operation of the protocol, the client sends a request and waits (blocks) for the server's reply. This synchronous nature ensures that the client has the necessary information to proceed only after receiving a response.
- **Error Handling and Timeouts**: The protocol includes mechanisms to handle communication failures. If the client does not receive a reply within a specified timeout period, it may resend the request. This is crucial for ensuring reliability in the face of potential message loss.

- **Duplicate Requests**: To manage situations where a request may be sent multiple times (e.g., due to timeouts), the server is designed to recognize duplicate requests using the requestId. This prevents the server from executing the same operation multiple times unnecessarily.
- **Idempotent Operations**: The protocol can benefit from idempotent operations, where repeated executions yield the same result. This characteristic simplifies handling duplicate requests and enhances reliability.

2. A. Discuss the design issues for Remote Procedure Call (RPC).

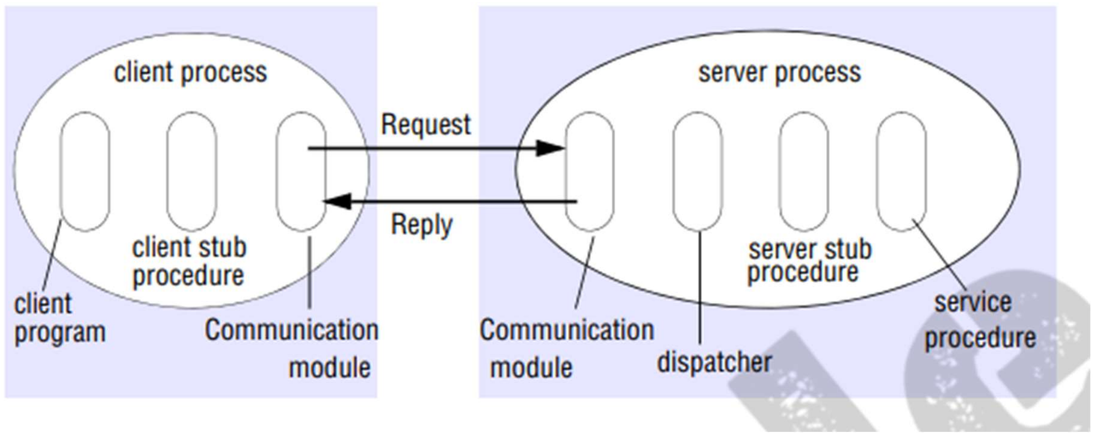
Ans. Designing a Remote Procedure Call (RPC) system involves addressing several key issues to ensure effective communication between distributed components. Here are the main design issues for RPC:

- **Transparency**: RPC aims to provide a level of transparency so that remote calls appear similar to local calls. This includes:
    - **Access Transparency**: Users should not need to know whether a procedure is local or remote.
    - **Location Transparency**: Users should not need to know the physical location of the server.
- **Call Semantics**: The semantics of the call must be defined clearly. This includes:
    - **At-least-once**: The procedure may be executed more than once, ensuring that the request is processed.
    - **At-most-once**: The procedure is executed at most once, avoiding duplicate processing.
- **Parameter Passing**: The method of passing parameters between the client and server needs to be designed carefully. This includes:
    - **Serialization**: Converting parameters to a format suitable for transmission over the network.
    - **Data Types**: Ensuring that data types are compatible between different systems, which may involve dealing with issues like byte order and data representation.
- **Error Handling**: Robust error handling mechanisms are essential to manage communication failures, server crashes, and other issues. This includes:

  - **Timeouts**: Implementing timeouts for requests to handle cases where the server does not respond.
  - **Retries**: Allowing clients to retry requests in case of failures, while ensuring that duplicate requests do not lead to unintended consequences.
- **Security**: Ensuring secure communication between the client and server is crucial. This involves:
  - **Authentication**: Verifying the identity of the client and server.
  - **Encryption**: Protecting data during transmission to prevent unauthorized access.
- **Performance**: The design must consider performance factors such as:
  - **Latency**: Minimizing the time taken for requests and responses to travel across the network.
  - **Throughput**: Ensuring that the system can handle a high volume of requests efficiently.
- **Scalability**: The RPC system should be able to scale as the number of clients and servers increases. This may involve:
  - **Load Balancing**: Distributing requests evenly across multiple servers to avoid bottlenecks.
  - **Replication**: Using multiple server instances to handle increased load and improve availability.
- **Versioning**: As systems evolve, maintaining compatibility between different versions of client and server interfaces is important. This requires careful management of changes to procedures and data structures.

B. Illustrate the implementation of Remote Procedure Call (RPC) in a Distributed Systems environment.

Ans.



The implementation of Remote Procedure Call (RPC) in a distributed systems environment involves several key components and steps. Here's an illustration of how RPC works, including the architecture and the process flow:

Key Components of RPC

- **Client**: The application that invokes the remote procedure.
- **Server**: The application that provides the remote procedure.
- **Stub**: A piece of code that acts as a proxy for the remote procedure. There are two types:
    - **Client Stub**: Marshals (serializes) the parameters and sends the request to the server.
    - **Server Stub**: Unmarshals (deserializes) the parameters and invokes the actual procedure on the server.
- **Transport Protocol**: The underlying protocol (e.g., TCP or UDP) used for communication between the client and server.

Implementation Steps

- **Define the Interface**:
    - Specify the procedures that can be called remotely, including their parameters and return types. This is often done using an Interface Definition Language (IDL).
- **Generate Stubs**:
    - Use a tool to automatically generate the client and server stubs from the defined interface. These stubs handle the serialization and deserialization of parameters and results.
- **Client-Side Execution**:
    - When the client wants to invoke a remote procedure, it calls the client stub instead of the actual procedure. The client stub:
        - Marshals the parameters into a message format suitable for transmission.

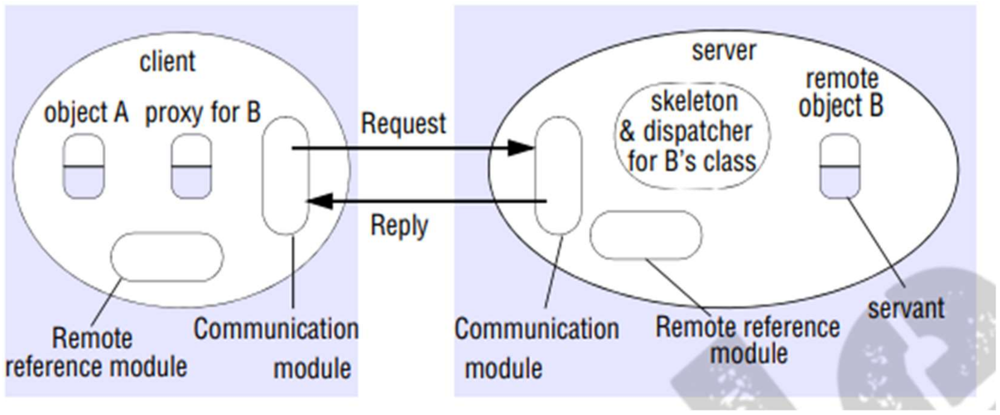
        - Sends the message to the server using the transport protocol.
- **Server-Side Execution**:
    - Upon receiving the request, the server stub:
        - Unmarshals the parameters from the received message.
        - Calls the actual procedure on the server with the unmarshaled parameters.
    - The procedure executes and returns a result to the server stub.
- **Return the Result**:
    - The server stub marshals the result into a message and sends it back to the client.
    - The client stub receives the response, unmarshals the result, and returns it to the client application.

Example Flow

- **Client Application** calls clientStub.procedureName(param1, param2).
- **Client Stub** marshals param1 and param2 and sends a request to the server.
- **Server** receives the request and passes it to the **Server Stub**.
- **Server Stub** unmarshals the parameters and calls actualProcedure(param1, param2).
- **Actual Procedure** executes and returns a result to the **Server Stub**.
- **Server Stub** marshals the result and sends it back to the **Client**.
- **Client Stub** receives the result, unmarshals it, and returns it to the **Client Application**.

C. Discuss the implantation of Remote method invocation (RMI).

Ans.



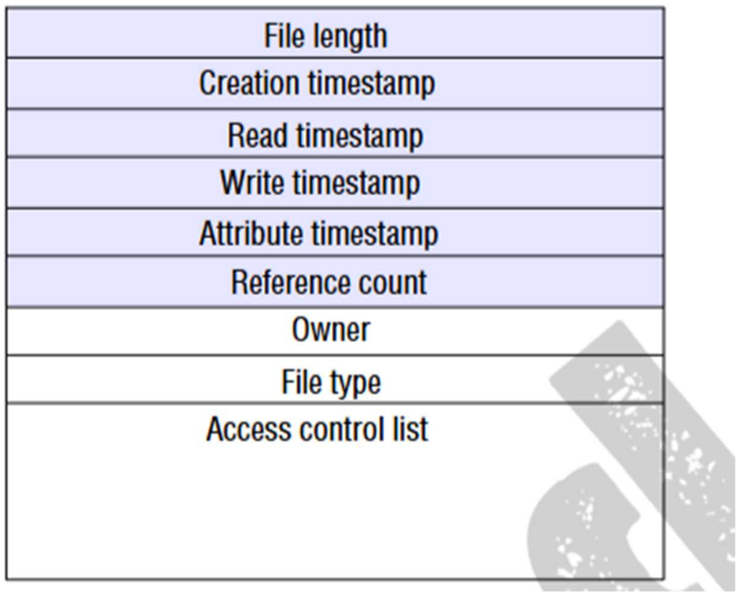
The implementation of Remote Method Invocation (RMI) involves several components and modules that work together to facilitate remote method calls between objects in a distributed system. Here are the key aspects:

- **Communication Module**: This module handles the request-reply protocol, transmitting messages between the client and server. It ensures that the messages contain the necessary information, such as the message type and the remote reference of the object to be invoked.
- **Remote Reference Module**: This module translates between local and remote object references and creates remote object references. It maintains a remote object table that records the correspondence between local and remote references.
- **Servants**: A servant is an instance of a class that provides the implementation of a remote object. It processes the remote requests passed on by the corresponding skeleton and exists within a server process.
- **RMI Software**: This layer acts as middleware between application-level objects and the communication and remote reference modules. It includes:
    - **Proxy**: This component makes remote method invocation transparent to clients by acting like a local object. It forwards invocations to the remote object and handles marshalling and unmarshalling of arguments and results.
    - **Dispatcher**: Each server has a dispatcher that receives request messages and selects the appropriate method to invoke based on the operation ID.
    - **Skeleton**: The skeleton implements the methods in the remote interface and is responsible for unmarshalling arguments from the request message and invoking the corresponding method in the servant.
- **Dynamic Invocation**: In cases where the remote interface is not known at compile time, dynamic invocation allows clients to invoke methods using a generic representation, enabling flexibility in handling remote objects.
- **Binding and Naming**: A binder service maintains a mapping from textual names to remote object references, allowing clients to look up remote objects by name.
- **Garbage Collection**: RMI includes a distributed garbage collection mechanism that ensures objects are only collected when no references to them exist. This is managed through reference counting and lease mechanisms to handle communication failures.

3. A. Review the characteristics of file systems.

Ans.File systems are responsible for the organization, storage, retrieval, naming, sharing, and protection of files. They provide a programming interface that abstracts the details of storage allocation and layout. Key characteristics include:

- **Data and Attributes**: Files contain data (typically as a sequence of 8-bit bytes) and attributes, which include information like file length, timestamps, file type, owner's identity, and access control lists.
- **File Management**: File systems manage large numbers of files and provide facilities for creating, naming, and deleting them. They use directories to map text names to internal file identifiers, supporting a hierarchical naming scheme.
- **Access Control**: File systems control access to files based on user authorizations and the type of access requested (e.g., reading, updating).
- **Metadata**: This term refers to the extra information stored by a file system for managing files, including file attributes and directories.



File length
Creation timestamp
Read timestamp
Write timestamp
Attribute timestamp
Reference count
Owner
File type
Access control list

B. Discuss the key requirements for Distributed File System.

Ans. Key requirements for a Distributed File System (DFS) include:

- **Transparency**:
    - **Access Transparency**: Users should be unaware of whether files are local or remote, allowing seamless access to files without modification to programs.
    - **Location Transparency**: The file system should present a uniform namespace, enabling files to be relocated without changing their pathnames.

    - **Mobility Transparency**: Files can be moved without requiring changes in client programs or administration tables.
    - **Performance Transparency**: Client performance should remain satisfactory despite variations in service load.
    - **Scaling Transparency**: The system should be able to expand incrementally to accommodate different loads and network sizes.

- **Concurrency Control**: Changes made by one client should not interfere with other clients accessing or modifying the same file, necessitating effective concurrency control mechanisms.
- **File Replication**: The ability to maintain multiple copies of files across different locations enhances scalability and fault tolerance, allowing clients to access files even if one server fails.
- **Fault Tolerance**: The system must continue to operate despite client or server failures, often through techniques like file replication and stateless server designs.
- **Consistency**: The system should ensure that all clients see a consistent view of file contents, particularly when files are replicated across different sites.
- **Security**: Access control mechanisms must be in place to authenticate client requests and protect data through encryption and digital signatures.
- **Efficiency**: The DFS should provide comparable performance and functionality to local file systems, ensuring it is convenient for administration.

C. Explain the Distributed File Service architecture.

Ans. The architecture of a Distributed File Service (DFS) typically consists of several key components that work together to provide file access and management across a network. The architecture can be divided into three main modules:

- **Client Module**:
    - This module emulates a conventional file system interface for application programs. It allows client applications to access remote files as if they were local, providing a familiar interface for users and developers.
- **Server Modules**:

    - These modules perform operations for clients on directories and files. They handle requests from clients, manage file storage, and ensure that file operations are executed correctly. The server modules can be designed to be stateless, which simplifies recovery and fault tolerance.

- **Directory Module**:

    - This component relates file names to file identifiers (IDs), enabling the organization and retrieval of files based on their names. It maintains the file namespace and handles operations related to file naming and directory management.

The architecture is designed to enable efficient communication between clients and servers while ensuring that the system can scale and adapt to varying loads. The separation of responsibilities among these modules allows for modular design, making it easier to implement and maintain the system.

4. A. Explain the followings w.r.t Name Services: (a) Uniform Resource Identifiers (URIs) and (b) Uniform Resource Locators (URL).

Ans. **(a) Uniform Resource Identifiers (URIs)**

Uniform Resource Identifiers (URIs) are a broad category of identifiers used to uniquely identify resources on the internet. A URI can be classified into two types: URLs and URNs (Uniform Resource Names). URIs provide a simple and extensible way to reference resources, allowing for both identification and location. The structure of a URI typically includes the following components:

- **Scheme**: Indicates the protocol used (e.g., http, ftp, mailto).
- **Path**: Specifies the specific resource within the scheme (e.g., /path/to/resource).
- **Query**: Optional parameters that can be passed to the resource (e.g., ?key=value).

URIs are essential for web services and applications, as they provide a standard way to access and manipulate resources across different systems.

(b) Uniform Resource Locators (URLs)

Uniform Resource Locators (URLs) are a specific type of URI that not only identifies a resource but also provides a means to locate it by describing its primary access mechanism. A URL typically includes the following components:

- **Scheme**: The protocol used to access the resource (e.g., http, https, ftp).

- **Host**: The domain name or IP address of the server hosting the resource (e.g., www.example.com).
- **Port**: An optional port number (e.g., :8080 for HTTP).
- **Path**: The specific location of the resource on the server (e.g., /path/to/resource).
- **Query**: Optional parameters for the resource (e.g., key=value).

For example, the URL https://www.example.com:80/path/to/resource?key=value specifies that the resource can be accessed over HTTPS at the given host and path, with an optional query string.

B. What is navigation w.r.t Name Servers? Explain the following navigations wr.t Name Servers: (a) iterative (b) multicast (c) nonrecursive server-controlled and (d) recursive server-controlled.

Ans.Navigation in the context of name servers refers to the methods used to resolve domain names into their corresponding IP addresses or other resource attributes. Here’s a brief explanation of the different types of navigation with respect to name servers:

(a) Iterative Navigation

In iterative navigation, the client sends a query to a name server, which responds with the best answer it has. If the name server does not have the answer, it will provide the client with a referral to another name server. The client then queries the referred server directly. This process continues until the client receives a definitive answer or reaches a server that can provide the necessary information.

(b) Multicast Navigation

Multicast navigation involves sending a query to multiple name servers simultaneously. This method is useful in scenarios where the client wants to obtain responses from several servers at once, potentially speeding up the resolution process. The client can then select the best response from the replies received.

(c) Nonrecursive Server-Controlled Navigation

In nonrecursive server-controlled navigation, the client sends a query to a name server, which does not perform any further queries on behalf of the client. Instead, the server provides a referral to another server, and it is the client’s responsibility to follow the referral and continue the resolution process. This method places more control in the hands of the client.

(d) Recursive Server-Controlled Navigation

In recursive server-controlled navigation, the client sends a query to a name server, which takes full responsibility for resolving the query. If the server does not have the answer, it will query other name servers on behalf of the client until it finds the answer or determines that the name cannot be resolved. This method simplifies the process for the client, as it does not need to manage multiple queries.

These navigation methods are fundamental in the design of name services, facilitating efficient communication and resource sharing in distributed systems.

C. What is Domain Name System? Explain the Domain Name System with suitable example.

Ans. The Domain Name System (DNS) is a hierarchical and decentralized naming system used to translate human-readable domain names (like www.example.com) into machine-readable IP addresses (like 192.0.2.1). This system is essential for the functioning of the internet, as it allows users to access websites using easy-to-remember names instead of numerical addresses.

Structure of DNS

- **Domain Names**: These are structured hierarchically, with levels separated by dots. For example, in the domain name www.example.com:
    - com is the top-level domain (TLD).
    - example is the second-level domain.
    - www is a subdomain.
- **DNS Records**: DNS stores various types of records, including:
    - **A Record**: Maps a domain to an IPv4 address.
    - **AAAA Record**: Maps a domain to an IPv6 address.
    - **CNAME Record**: Alias of one domain to another.
    - **MX Record**: Mail exchange record, directing email to the correct server.

Example

When you type www.example.com into your web browser:

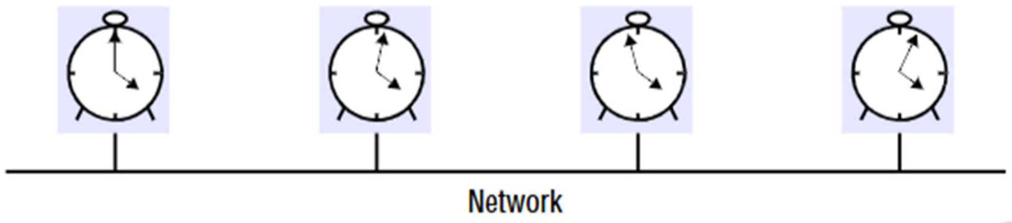
- The browser checks its cache for the IP address.
- If not found, it queries a DNS resolver (often provided by your ISP).
- The resolver checks its cache. If still not found, it queries the root DNS servers.

- The root server directs the resolver to the .com TLD server.
- The TLD server points to the authoritative DNS server for example.com.
- The authoritative server responds with the IP address for www.example.com.
- The resolver returns the IP address to your browser, which can then connect to the web server at that address.

5. A. Discuss the followings: (a) Clock Skew, (b) Clock Drift and (c) Coordinated Universal Time.

Ans.

(a) **Clock Skew** refers to the difference in time readings between two clocks at a given moment. It is often a result of variations in the transmission time of messages between systems. In a synchronous system, the optimum bound that can be achieved on clock skew when synchronizing (N) clocks is given by u·11+N*u*·1+*N*1, where (u) is the uncertainty in message transmission time.

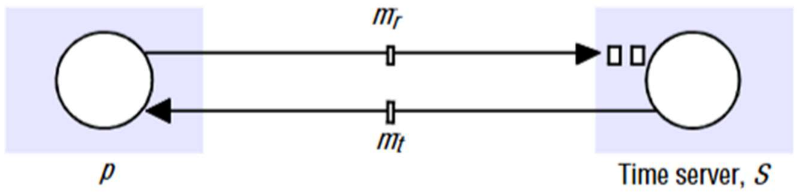


(b) **Clock Drift** is the gradual deviation of a clock from the correct time over a period. It occurs due to imperfections in the clock's mechanism, leading to a consistent gain or loss of time. For example, a clock might gain four seconds every hour, which can be adjusted in software or hardware to maintain accuracy.

(c) **Coordinated Universal Time (UTC)** is the time standard used globally, which is not affected by time zones or Daylight Saving Time. It is based on International Atomic Time (TAI) and is adjusted with leap seconds to account for the Earth's rotation variations. UTC serves as the basis for timekeeping in distributed systems, ensuring that all clocks can synchronize to a common reference.

B. Explain the Cristian's method for synchronizing clocks.

Ans.



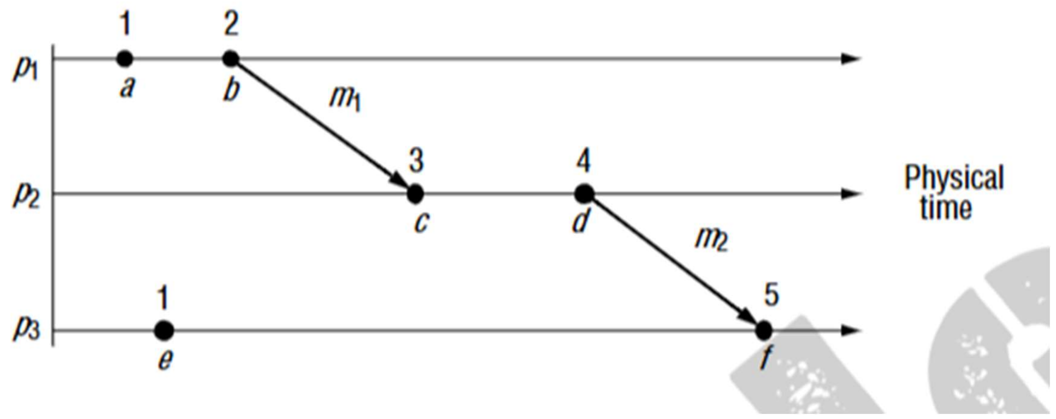
Cristian's method for synchronizing clocks involves using a time server that is connected to a device receiving signals from a source of Coordinated Universal Time (UTC). The process works as follows:

- **Request for Time**: A client process (p) sends a request message (m_r) to the time server (S) asking for the current time.
- **Response from Server**: The server (S) responds with a message (m_t) that contains the current time (t) from its clock, which is recorded just before sending the message.
- **Round-Trip Time Measurement**: The client records the total round-trip time (T_{round}) taken to send the request and receive the response. This time can be measured accurately if the clock drift rate is small.
- **Time Adjustment**: The client estimates the time to set its clock using the formula: t+Tround2*t*+2*Tround* This assumes that the time taken for the request and response is approximately equal, which is generally a reasonable assumption unless the messages travel over different networks.
- **Accuracy Consideration**: If the minimum transmission time (min) is known or can be estimated, the accuracy of the synchronization can be assessed. The time when the reply arrives is within the range of (t + min) and (t + T_{round} - min), leading to an accuracy of approximately: ±Tround2+min±2*Tround*+*min*
- **Handling Variability**: To improve accuracy, the client can make multiple requests to the server and take the minimum round-trip time observed, which helps mitigate the effects of transient network congestion.

This method is probabilistic, meaning it achieves synchronization only if the observed round-trip times are sufficiently short compared to the required accuracy.

C. What is a Logical Clock? Explain the Lamport's logical clock.

Ans. A **Logical Clock** is a mechanism used in distributed systems to order events without relying on synchronized physical clocks. It provides a way to capture the chronological order of events in a system where processes may not have a consistent view of time.



Lamport's Logical Clock, introduced by Leslie Lamport in 1978, is a specific implementation of logical clocks. It works as follows:

- **Monotonically Increasing Counter**: Each process (p_i) in the system maintains its own logical clock (L_i), which is a monotonically increasing counter. The value of this clock does not need to correspond to any physical time.
- **Timestamping Events**: When a process (p_i) performs an event (like sending or receiving a message), it updates its logical clock:
    - Before each event, the process increments its logical clock: Li:=Li+1*Li*:=*Li*+1
    - The timestamp of the event is then recorded as (L_i).
- **Sending Messages**: When a process (p_i) sends a message (m), it includes its current logical clock value (L_i) with the message.
- **Receiving Messages**: Upon receiving a message (m) with timestamp (t) from another process (p_j), the receiving process (p_k) updates its logical clock:
    - It sets its clock to the maximum of its current logical clock and the received timestamp: Lk:=max(Lk,t)*Lk*:=max(*Lk*,*t*)
    - It then increments its logical clock before timestamping the receive event: Lk:=Lk+1*Lk*:=*Lk*+1

- **Happened-Before Relation**: Lamport's logical clocks help establish a partial ordering of events in the system, known as the **happened-before relation** (denoted as (e_1 \rightarrow e_2)). If event (e_1) happened before event (e_2), then the timestamp of (e_1) will be less than that of (e_2) (i.e., (L(e_1) < L(e_2))).

6. A. Discuss the followings w.r.t Network Time Protocol: (a) Design aims and features (b) Modes of NTP server synchronization.

Ans. **(a) Design Aims and Features of NTP**

The Network Time Protocol (NTP) is designed with several key aims and features:

- **Accurate Synchronization**: NTP enables clients across the Internet to synchronize accurately to Coordinated Universal Time (UTC), despite the presence of large and variable message delays.
- **Reliability**: It provides a reliable service that can withstand lengthy losses of connectivity by utilizing redundant servers and paths. The system can reconfigure itself to continue providing time services even if some servers become unreachable.
- **Frequent Resynchronization**: NTP is designed to allow clients to resynchronize frequently enough to counteract the drift rates found in most computers, ensuring that time remains accurate.
- **Protection Against Interference**: The protocol incorporates authentication techniques to verify that timing data comes from trusted sources, protecting against both accidental and malicious interference.

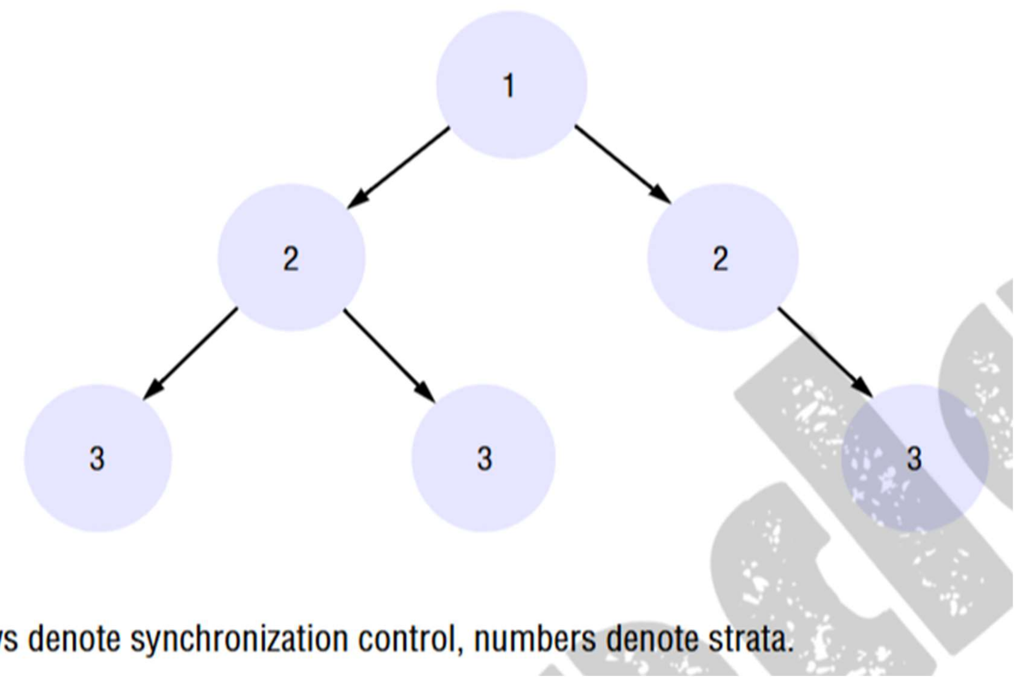
(b) Modes of NTP Server Synchronization

NTP servers can synchronize in three primary modes:

- **Multicast Mode**: This mode is intended for high-speed Local Area Networks (LANs). One or more servers periodically send out time information to other servers on the network, which set their clocks based on this information, assuming a small transmission delay. This mode typically achieves lower accuracy.
- **Procedure-Call Mode**: Similar to Cristian's algorithm, this mode involves one server accepting requests from other computers and replying with its timestamp. It is suitable for scenarios where higher accuracy is required than what multicast can provide.

- **Symmetric Mode**: This mode is used by servers that provide time information within LANs and by higher strata servers in the synchronization subnet. In symmetric mode, pairs of servers exchange timing information, maintaining an association that improves synchronization accuracy over time.

An example synchronization subnet in an NTP implementation



Arrows denote synchronization control, numbers denote strata.

B. Explain the Global states and consistent cuts with suitable example.

Ans. **Global States** refer to the overall state of a distributed system at a particular point in time, encompassing the states of all processes and communication channels. However, due to the lack of global time in distributed systems, determining a meaningful global state can be challenging.

A **Consistent Cut** is a specific type of global state that respects the causal relationships between events in the system. In other words, a cut is consistent if it includes all events that must have occurred before any event in the cut.

Example

Consider a distributed system with two processes, (P_1) and (P_2), that communicate via messages. The events occurring in these processes can be represented as follows:

- **Process (P_1)**:

    - (e_{1,0}): Send message (m_1) to (P_2)
    - (e_{1,1}): Internal event (e.g., update a variable)
- **Process (P_2)**:
    - (e_{2,0}): Receive message (m_1) from (P_1)
    - (e_{2,1}): Send message (m_2) to (P_1)
    - (e_{2,2}): Internal event

Global States

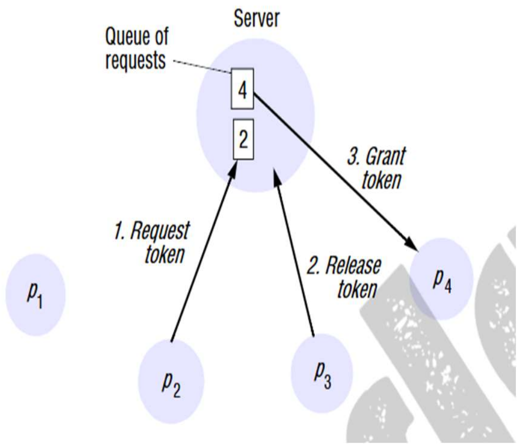
- **Global State 1**: After (e_{1,0}) and (e_{2,0}) have occurred.
- **Global State 2**: After (e_{1,1}) and (e_{2,1}) have occurred.

Consistent Cuts

- **Consistent Cut**: A cut that includes (e_{1,0}) (sending (m_1)) and (e_{2,0}) (receiving (m_1)) is consistent because it respects the causal relationship; (P_2) must receive (m_1) after (P_1) sends it.
- **Inconsistent Cut**: A cut that includes (e_{2,1}) (sending (m_2)) without including (e_{1,0}) is inconsistent because it shows (P_2) sending a message without having received (m_1) first, which violates the causal relationship.

7. A. Discuss the followings algorithms for mutual exclusion in Distributed Systems: (a) central server algorithm (b) ring-based algorithm (c) multicast and logical clocks.

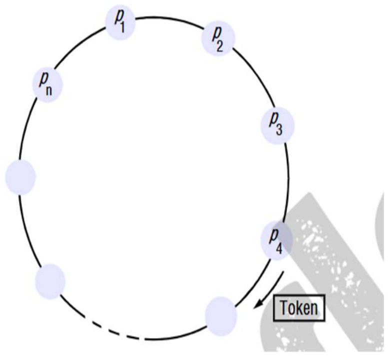
Ans. **(a) Central Server Algorithm**



The central server algorithm for mutual exclusion involves a server that manages access to the critical section. Each process sends a request message to the server to enter the critical section and waits for a reply, which acts as a token granting permission. If no other process holds the token, the server replies immediately; otherwise, it queues the request. When a process exits the critical section, it informs the server, which then grants the token to the next waiting process. This algorithm ensures safety and liveness, but it does not satisfy the ordering requirement (ME3), meaning it may not grant access in the order requests were made.

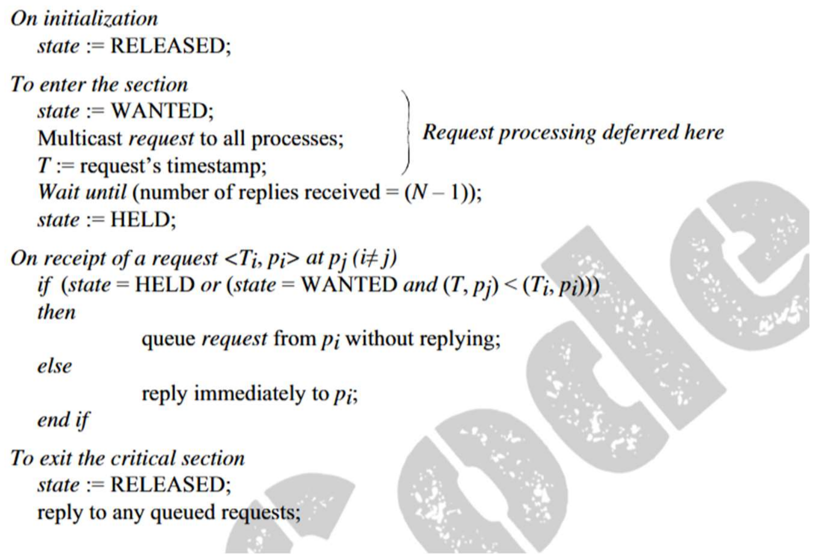
(b) Ring-Based Algorithm

In the ring-based algorithm, processes are arranged in a logical ring. Each process has a communication channel to its neighbor. A token circulates around the ring, and a process can enter the critical section only when it holds the token. If a process does not need to enter the critical section, it forwards the token to the next process. This algorithm meets the safety and liveness conditions, but it continuously consumes bandwidth as messages are sent around the ring, even when no process requires entry.

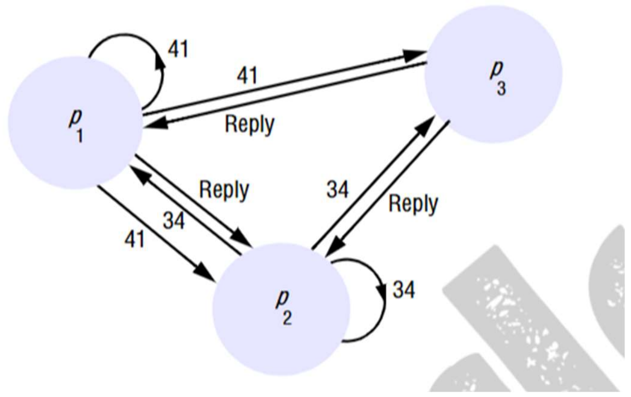


(c) Multicast and Logical Clocks (Ricart and Agrawala's Algorithm)

This algorithm uses multicast to manage mutual exclusion among peer processes. When a process wants to enter the critical section, it multicasts a request message to all other processes and waits for replies. A process replies to a request based on its state and the timestamps of the requests, ensuring that the process with the earliest timestamp gets priority. This algorithm satisfies all three mutual exclusion properties: safety (ME1), liveness (ME2), and ordering (ME3). The synchronization delay is minimal, making it efficient in terms of message transmission.



```
On initialization
    state := RELEASED;
To enter the section
    state := WANTED;                              ⎫
    Multicast request to all processes;           ⎬ Request processing deferred here
    T := request's timestamp;                     ⎭
    Wait until (number of replies received = (N – 1));
    state := HELD;
On receipt of a request <T_i, p_i> at p_j (i ≠ j)
    if (state = HELD or (state = WANTED and (T, p_j) < (T_i, p_i)))
    then
                queue request from p_i without replying;
    else
                reply immediately to p_i;
    end if
To exit the critical section
    state := RELEASED;
    reply to any queued requests;
```

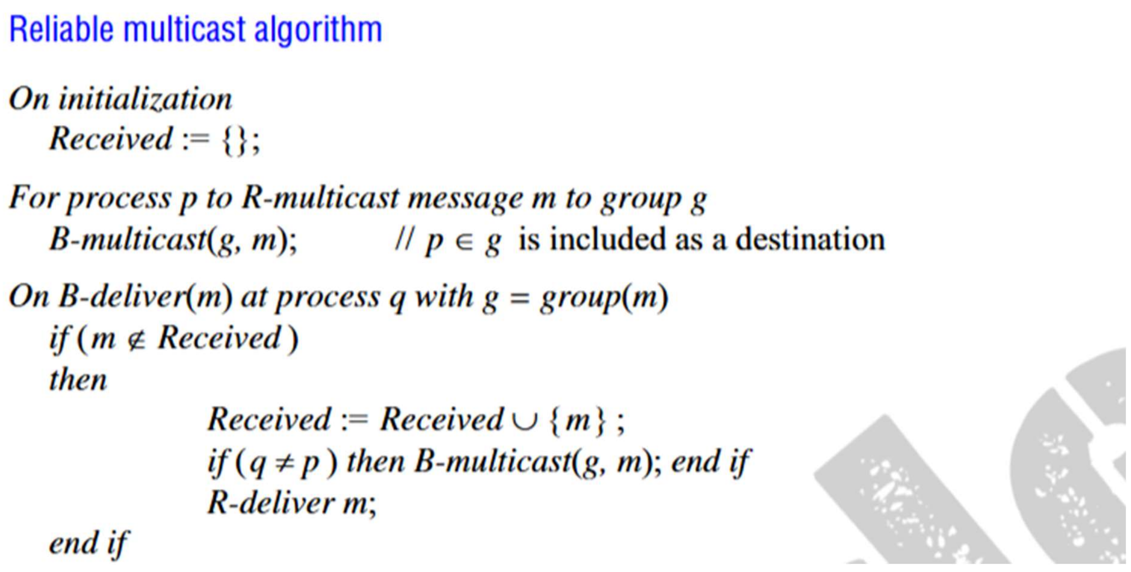


B. What are the properties of Reliable multicast? Explain the Reliable multicast algorithm.

Ans. Reliable multicast has three key properties:

- **Integrity**: A correct process delivers a message at most once. Additionally, the message must have been sent by a valid sender to the correct group.
- **Validity**: If a correct process multicasts a message, then it will eventually be delivered by all correct processes in the group.
- **Agreement**: If a correct process delivers a message, then all other correct processes in the group will eventually deliver that same message.

Reliable Multicast Algorithm



Reliable multicast algorithm

```
On initialization
    Received := {};

For process p to R-multicast message m to group g
    B-multicast(g, m);          // $p \in g$ is included as a destination

On B-deliver(m) at process q with g = group(m)
    if ($m \notin$ Received)
    then
            Received := Received $\cup$ {m};
            if ($q \neq p$) then B-multicast(g, m); end if
            R-deliver m;
    end if
```

The reliable multicast algorithm typically involves the following steps:

- **Initialization**: Each process maintains a set of received messages to track which messages have been delivered.
- **R-multicast Operation**: When a process wants to multicast a message m to a group g, it first performs a basic multicast operation (B-multicast) to send the message to all members of the group, including itself.
- **B-deliver Operation**: When a process receives a message via B-multicast, it checks if it has already delivered that message. If not, it marks the message as received and may multicast it again to ensure that other processes also receive it.
- **R-deliver Operation**: After a process B-delivers a message, it then R-delivers it, which means it acknowledges the delivery to the application layer.
- **Handling Duplicates**: The algorithm ensures that duplicates are detected and not delivered multiple times to the application layer.

8. A. Explain the Maekawa's voting algorithm for mutual exclusion in Distributed Systems.

Ans, Maekawa's voting algorithm is designed to achieve mutual exclusion in distributed systems by allowing processes to obtain permission to enter a critical section from subsets of their peers, rather than requiring permission from all processes. Here's how it works:

Key Features

- **Voting Sets**: Each process (p_i) is associated with a voting set (V_i), which includes a subset of processes. The voting sets are designed such that:
  - Each process is included in its own voting set.
  - Any two voting sets (V_i) and (V_j) share at least one common member, ensuring safety (ME1).
  - All voting sets have the same size (K).
  - Each process is contained in (M) voting sets.
- **Requesting Entry**: To enter the critical section, a process (pi) sends request messages to all members of its voting set (Vi). It cannot enter the critical section until it receives replies (votes) from all members of (Vi).
- **Replying to Requests**: When a process (pj) receives a request from (pi):

  - If (pj) is currently in the critical section (state HELD) or has already voted since its last release, it queues the request without replying.
  - Otherwise, it sends a reply (vote) to (pi) and marks that it has voted.
- **Exiting the Critical Section**: When a process (pi) exits the critical section, it sends release messages to all members of its voting set (Vi). Upon receiving a release message, a process (pj) will check its queue of requests and reply to the next request if there are any pending.

Properties

- **Safety (ME1)**: At most one process can enter the critical section at a time because of the overlapping voting sets.
- **Liveness (ME2)**: Every process that requests entry will eventually be granted access, assuming no process fails indefinitely.
- **Fairness**: The algorithm can be adapted to ensure fairness, where processes are granted access in a fair manner.

Limitations

- **Deadlock Potential**: The algorithm can lead to deadlock if processes hold onto their votes while waiting for votes from others. This can be mitigated by adapting the algorithm to ensure that processes queue requests in a consistent order.

Maekawa's algorithm is efficient in terms of message complexity and bandwidth usage, making it a practical solution for mutual exclusion in distributed systems.

B. What is an Election algorithm? What are its requirements? Explain the ring-based election algorithm.

Ans. An **election algorithm** is a method used in distributed systems to select a unique process to perform a specific role, such as a coordinator or leader among a group of processes. This is crucial for ensuring coordination and agreement in systems where multiple processes may need to communicate or share resources.

Requirements of Election Algorithms

- **Uniqueness**: There must be a unique elected process at any given time. No two processes should be elected simultaneously.
- **Agreement**: All processes in the system must agree on which process has been elected.
- **Termination**: The algorithm must eventually complete, meaning that the election process should finish in a finite amount of time.
- **Fault Tolerance**: The algorithm should be able to handle process failures gracefully, allowing for re-election if the elected process fails.

Ring-Based Election Algorithm

The ring-based election algorithm, proposed by Chang and Roberts, is designed for a collection of processes arranged in a logical ring. Here's how it works:

- **Initialization**: Each process (p_i) has a communication channel to its clockwise neighbor (p_{i+1}) (with wrapping around at the end of the ring).
- **Starting an Election**: Any process can initiate an election by marking itself as a participant and sending its identifier in an election message to its clockwise neighbor.
- **Message Handling**:
    - When a process (p_j) receives an election message, it compares the identifier in the message with its own:
        - If the received identifier is greater, (p_j) forwards the message to its neighbor.
        - If the received identifier is smaller and (p_j) is not a participant, it replaces the identifier in the message with its own and forwards it.
        - If the received identifier is equal to its own, (p_j) recognizes itself as the coordinator.
- **Election Completion**: Once a process becomes the coordinator, it sends an elected message to its neighbors, announcing its election. Each process that receives this message marks itself as a non-participant and updates its record of the elected process.
- **Handling Concurrent Elections**: If multiple processes start elections simultaneously, the algorithm ensures that only one process will emerge as the coordinator due to the unique identifier comparison.

Performance

The ring-based election algorithm is efficient in terms of message complexity, requiring at most (3N - 1) messages in the worst case, where (N) is the number of processes. The turnaround time is also (3N - 1), as messages are sent sequentially around the ring.

9. A. Explain the two-phase commit protocol w.r.t distributed transactions.

Ans. The **two-phase commit (2PC) protocol** is a consensus algorithm used in distributed systems to ensure that all participants in a distributed transaction either commit to the transaction or abort it, maintaining consistency across the system. It is particularly important in scenarios where a transaction spans multiple databases or nodes.

Phases of the Two-Phase Commit Protocol

- **Phase 1: Prepare Phase (Voting Phase)**
    - The coordinator process sends a **prepare** request to all participant processes involved in the transaction.
    - Each participant receives the request and performs the necessary checks to determine if it can commit the transaction (e.g., checking local conditions, ensuring resources are available).
    - Each participant then responds to the coordinator with either:
        - **Vote Commit**: If the participant is ready to commit.
        - **Vote Abort**: If the participant cannot commit (e.g., due to a failure or violation of constraints).
- **Phase 2: Commit Phase (Decision Phase)**
    - Once the coordinator receives votes from all participants:
        - If all participants vote to commit, the coordinator sends a **commit** message to all participants, instructing them to finalize the transaction.
        - If any participant votes to abort, the coordinator sends an **abort** message to all participants, instructing them to roll back the transaction.
    - Each participant then acknowledges the commit or abort decision back to the coordinator.

Requirements and Properties

- **Atomicity**: The two-phase commit protocol ensures that a distributed transaction is either fully committed or fully aborted, maintaining atomicity.
- **Consistency**: The protocol guarantees that all participants see the same outcome (either commit or abort).
- **Durability**: Once a commit decision is made, it is permanent, and participants must ensure that the changes are durable.
- **Fault Tolerance**: The protocol can handle certain types of failures, but it is vulnerable to coordinator failures during the commit phase, which may lead to uncertainty about the transaction's outcome.

Limitations

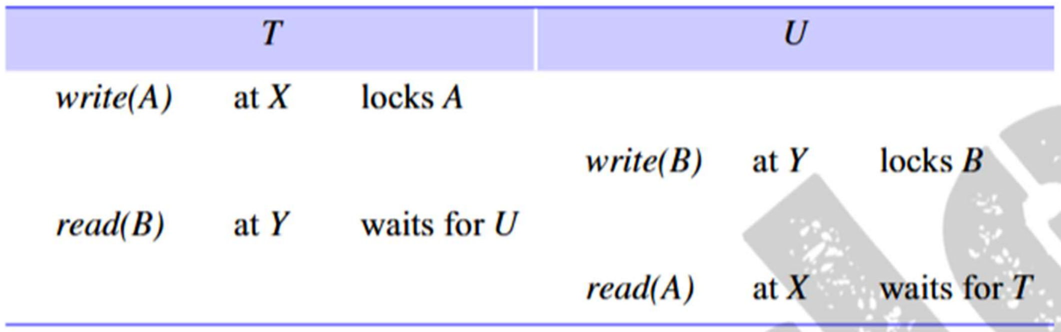
- **Blocking**: If the coordinator fails after sending a prepare request but before receiving all responses, participants may block indefinitely, waiting for a decision.
- **Performance**: The two-phase commit protocol can introduce latency due to the need for multiple message exchanges between the coordinator and participants.
- **Single Point of Failure**: The coordinator represents a single point of failure, which can impact the availability of the system.

B. Discuss the various methods of concurrency control in distributed transactions.

Ans. Concurrency control in distributed transactions is essential to ensure that transactions are executed in a consistent manner, especially when multiple transactions access shared resources simultaneously. The main methods of concurrency control include:

1. Locking

- **Local Locks**: Each server holds locks on objects locally. A local lock manager decides whether to grant a lock or make a transaction wait.
- **Deadlocks**: Different servers may impose different orderings on transactions, leading to potential deadlocks. If a deadlock is detected, one transaction is aborted to resolve the situation.



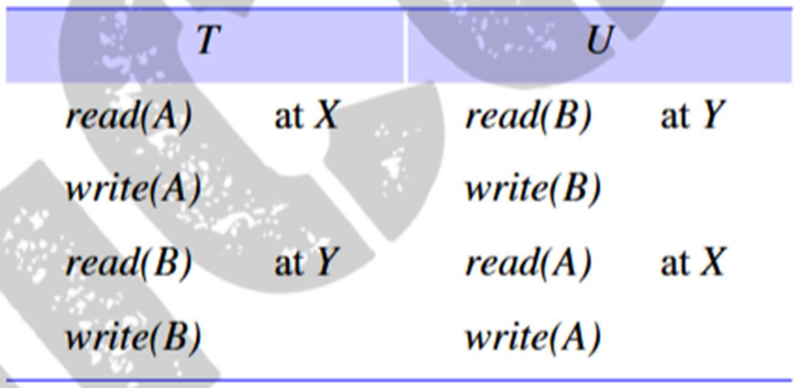
T			*U*		
write(A)	at *X*	locks *A*			
			write(B)	at *Y*	locks *B*
read(B)	at *Y*	waits for *U*			
			read(A)	at *X*	waits for *T*

2. Timestamp Ordering

- **Unique Timestamps**: Each transaction is assigned a globally unique timestamp by the coordinator. The order of transaction commits is based on these timestamps.
- **Serial Equivalence**: Transactions must commit in the order of their timestamps across all servers, ensuring a consistent view of the data.

3. Optimistic Concurrency Control

- **Validation Phase**: Transactions are validated before committing. Each server validates transactions that access its objects during the first phase of the two-phase commit protocol.
- **Commitment Deadlock**: If two transactions are validated at the same time but in different orders across servers, it can lead to commitment deadlocks. To mitigate this, parallel validation protocols can be implemented.



T		*U*	
read(A)	at *X*	*read(B)*	at *Y*
write(A)		*write(B)*	
read(B)	at *Y*	*read(A)*	at *X*
write(B)		*write(A)*	

4. Global Validation

- After local validation, a global validation checks that the combination of orderings at individual servers is serializable, ensuring no cycles exist among transactions.

5. Multi-Version Concurrency Control (MVCC)

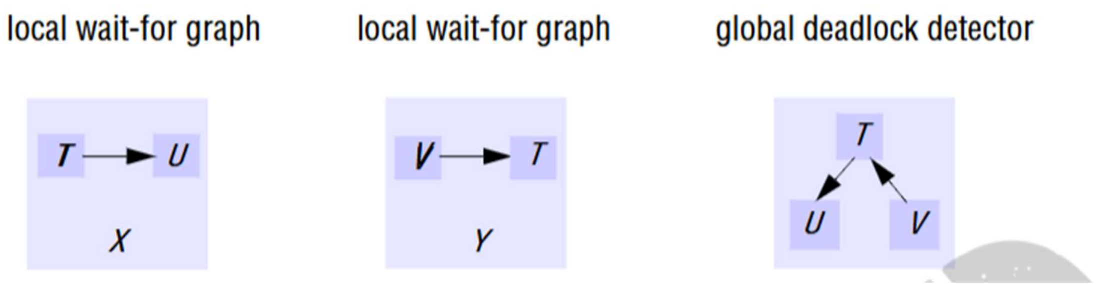
- **Multiple Versions**: This method allows multiple versions of an object to exist simultaneously, enabling read operations to access older versions while write operations create new versions.
- **Read-Only Transactions**: MVCC can favor read-only transactions, allowing them to proceed without waiting for write locks.

10. A. Discuss (a) Phantom deadlocks and (b) Edge chasing w.r.t deadlock in Distributed Systems.

Ans. **(a) Phantom Deadlocks**

Phantom deadlocks occur when a deadlock is detected based on outdated information about transaction states. This situation arises in distributed systems where transactions may release locks after the deadlock detection process has started.

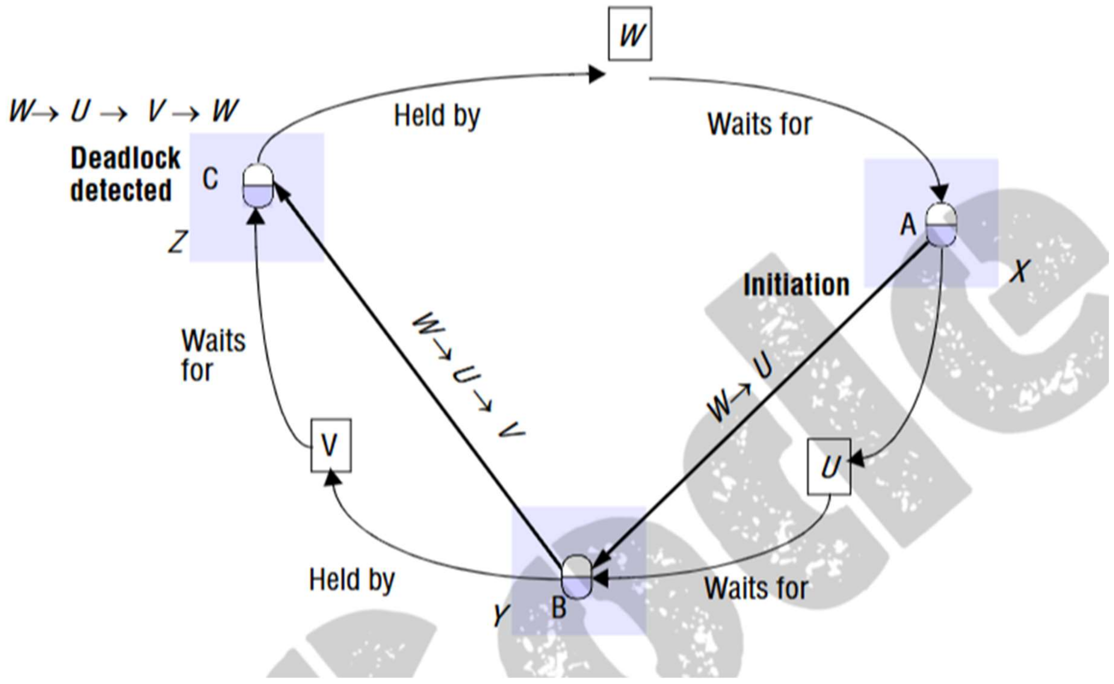
- **Detection Process**: A global deadlock detector collects local wait-for graphs from various servers. If a transaction releases a lock while the detector is still processing, it may create a situation where the detector identifies a cycle that no longer exists. For example, if transaction U releases a lock and then requests another held by transaction V, the detector might still see a cycle involving U and V, even though U is no longer waiting.
- **Consequences**: This can lead to unnecessary transaction aborts, as the system believes a deadlock exists when it does not. Phantom deadlocks can complicate the deadlock detection process and reduce overall system efficiency.



(b) Edge Chasing

Edge chasing, also known as path pushing, is a distributed approach to deadlock detection that does not require constructing a global wait-for graph. Instead, it uses a message-passing mechanism to identify cycles in the wait-for relationships among transactions.

- **Mechanism**: When a transaction T starts waiting for another transaction U, the server initiates detection by sending a probe message containing the edge <T → U> to the server managing U. If U is also waiting for another transaction V, the probe is updated to <T → U → V> and forwarded accordingly.
- **Cycle Detection**: As probes traverse the network, each server checks if the probe forms a cycle. If a cycle is detected, it indicates a deadlock, and one of the transactions in the cycle must be aborted to resolve the deadlock.
- **Advantages**: Edge chasing reduces the overhead of maintaining a global wait-for graph and allows for more dynamic detection of deadlocks as transactions proceed. However, it may require multiple messages to fully trace the paths, and care must be taken to avoid false positives in deadlock detection.



B. Explain the following approaches used in the file recovery in Distributed Systems: (a) Logging and (b) Shadow versions.

Ans, **(a) Logging**

Logging is a technique used in file recovery that involves maintaining a record of all changes made to files. This log helps in recovering files to a consistent state after a failure.

- **Write-Ahead Logging**: Before any changes are made to the actual file, the changes are first recorded in a log. This ensures that if a failure occurs during the write process, the system can refer to the log to either complete the operation or roll back to the last consistent state.
- **Types of Logs**:
    - **Transaction Logs**: These logs record all operations of a transaction, allowing for recovery by replaying or undoing specific transactions.
    - **Checkpointing**: Periodically, the system saves a snapshot of the current state of files. In case of a failure, the system can revert to the last checkpoint and apply the log entries to bring the files up to date.
- **Recovery Process**: Upon recovery, the system reads the log to determine which operations were completed and which were not, allowing it to restore the files to a consistent state.

(b) Shadow Versions

Shadow versions involve maintaining a copy of the file (the shadow) that represents its previous state before any changes are made. This approach allows for quick recovery without the need to roll back changes.

- **How It Works**: When a file is modified, the system creates a shadow copy of the file's current state. The new changes are then applied to the original file. If a failure occurs or if the changes need to be discarded, the system can simply revert to the shadow version without needing to analyze logs or undo operations.
- **Advantages**:
    - **Fast Recovery**: Since the previous state is readily available, recovery can be instantaneous.
    - **Minimal Overhead**: The system does not need to maintain extensive logs, reducing the overhead associated with logging.
- **Considerations**: While shadow versions provide quick recovery, they may require additional storage space for the copies, and managing multiple versions can become complex if not handled properly.