1. **"Distributed system acts as a single coherent system to its end user" justify the statement with its features and challenges.**

   A distributed system is designed to provide the illusion of a single coherent system to its end users, despite being composed of multiple interconnected and independent components. This illusion is maintained through various features and mechanisms that ensure consistent behavior and seamless interaction. However, achieving this goal comes with its own set of challenges.

   Features:

   - **Transparency**: A distributed system aims to hide the complexities of its underlying architecture from end users. Different types of transparency, such as location transparency (users don't need to know where data is located), access transparency (users access data in a consistent manner regardless of its location), and failure transparency (users are shielded from system failures), contribute to the coherent user experience.
   - **Interconnectedness**: Distributed systems connect multiple devices, nodes, or computers over a network. Communication mechanisms, such as remote procedure calls (RPCs) or message passing, allow users to interact with various parts of the system as if they were all part of a single entity.
   - **Scalability**: Distributed systems can scale horizontally by adding more nodes to accommodate increasing demands. This scalability allows the system to handle larger user loads and data volumes while maintaining the illusion of a single system.
   - **Fault Tolerance**: To ensure reliability, distributed systems often implement fault tolerance mechanisms. Replication, where data is stored on multiple nodes, and redundancy, where backup components are available to take over in case of failure, help maintain the coherent experience by minimizing disruptions.

   Challenges:

   - **Network Communication**: Ensuring consistent and reliable communication between distributed components can be challenging due to factors like latency,

bandwidth limitations, and potential packet loss. This can lead to delays or inconsistencies in the user experience.

- **Data Consistency**: Maintaining data consistency across multiple nodes is complex. Synchronization mechanisms, like distributed transactions or consensus protocols, are necessary to ensure that data remains accurate and coherent even in the face of failures.
- **Concurrency Control**: When multiple users or components access shared resources simultaneously, concurrency control mechanisms are required to prevent conflicts and maintain data integrity. These mechanisms can introduce complexity and potential performance bottlenecks.
- **Failure Management**: Handling node failures and maintaining system availability is crucial. Implementing techniques like replication and failover introduces complexity to ensure that the system remains coherent even when parts of it are temporarily unavailable.
- **Scalability Complexity**: While distributed systems offer scalability, managing a growing number of nodes and resources requires careful design and management. Ensuring that the system scales without sacrificing performance or introducing new points of failure can be challenging.
- **Security and Privacy**: Distributed systems often involve transmitting data across networks, which raises concerns about security and privacy. Ensuring secure communication, access control, and data encryption are essential to maintain user trust and the illusion of a single coherent system.

In conclusion, the statement that a distributed system acts as a single coherent system to its end users is justified by the features and challenges inherent to such systems. The features enable the system to provide a seamless user experience, while the challenges highlight the complexities involved in achieving this goal, from communication and data consistency to fault tolerance and security.

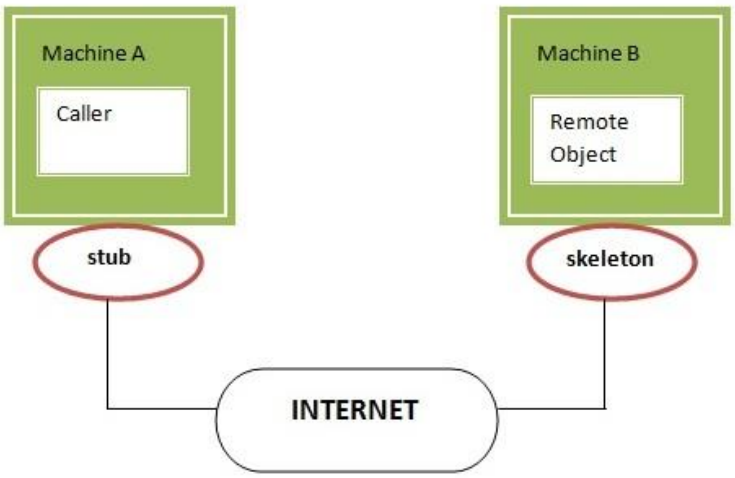
Fundamental Model: A fundamental model is an abstract representation or theoretical framework that defines the basic principles, concepts, and interactions that guide the design, analysis, and understanding of distributed computing environments. This model

provides a structured approach to conceptualize the behavior, communication patterns, and relationships among different components within a distributed system.

2. **What is DFS? How RMI perform communication between distributed objects? Explain.**

Distributed File System (DFS): A Distributed File System is a networked file system that allows multiple users or applications across different machines to access and share files as if they were stored on a local file system. DFS abstracts the physical locations of files and provides a unified view of the distributed storage space.

Remote Method Invocation (RMI) is a Java technology that allows objects to invoke methods on other objects that are located in different Java Virtual Machines (JVMs). This is done by using a stub and skeleton mechanism. The stub is an object that is created on the client side. It acts as a proxy for the remote object. When the client invokes a method



on the stub, the stub packages the method call and sends it to the server. The skeleton is an object that is created on the server side. It receives the method call from the stub and executes the method on the remote object. The skeleton then returns the result of the method call to the stub, which in turn returns it to the client.

The RMI communication between distributed objects works as follows:

1. The client creates a stub object for the remote object.
2. The client invokes a method on the stub object.

3. The stub packages the method calls and sends it to the server.
4. The skeleton on the server receives the method call and executes it on the remote object.
5. The skeleton returns the result of the method call to the stub.
6. The stub returns the result of the method call to the client.

This process is transparent to the client, which makes it appear as if the remote object is local.

Here are some of the advantages of using RMI for communication between distributed objects:

- It is easy to use. The RMI API is very similar to the Java standard library API, so it is easy for developers to learn and use.
- It is efficient. RMI uses a variety of techniques to optimize the performance of remote method invocations, such as caching and compression.
- It is reliable. RMI provides a number of features to ensure the reliability of remote method invocations, such as retries and timeouts.
- It is secure. RMI supports a variety of security features, such as authentication and authorization.

Overall, RMI is a powerful and versatile technology that can be used to facilitate communication between distributed objects in a variety of applications.

3. Verify with proper explanations that DNS is a distributed hierarchical database system.

The Domain Name System (DNS) is a distributed hierarchical database system that maps domain names to IP addresses. It is used to translate human-readable domain names into the numerical IP addresses that computers use to communicate with each other.

DNS is a distributed system because it is composed of a network of name servers that are located all over the world. Each name server is responsible for a certain portion of the DNS namespace. When a client needs to resolve a domain name, it sends a query to the nearest

name server. The name server then queries other name servers as needed until it finds the answer.

DNS is also a hierarchical system. The DNS namespace is divided into a hierarchy of domains, with each domain having a parent domain. The top of the hierarchy is the root domain, which is represented by the single dot (.). The root domain has two child domains: .com and .net. These domains, in turn, have child domains, and so on.

The hierarchical structure of DNS makes it efficient to store and manage a large number of domain names. Each name server only needs to store information about the domains that it is responsible for. This reduces the amount of traffic on the network and makes it faster to resolve domain names. The distributed and hierarchical nature of DNS makes it a reliable and scalable system. If one name server fails, other name servers can still resolve domain names. This makes DNS a critical part of the Internet infrastructure.

Some additional points that support the claim that DNS is a distributed hierarchical database system:

- DNS is a distributed system because it is composed of a network of name servers that are located all over the world. This means that no single point of failure can bring down the entire DNS system.
- DNS is a hierarchical system because the DNS namespace is divided into a hierarchy of domains. This makes it easy to organize and manage a large number of domain names.
- DNS is a database because it stores information about domain names and their corresponding IP addresses. This information is used to translate domain names into IP addresses, which is necessary for computers to communicate with each other on the Internet.

4. **Write the importance of election algorithm. Explain BULLY algorithm with suitable example. Compare it with Ring based algorithm**

Election algorithms in distributed systems play a crucial role in selecting a leader or coordinator among a group of nodes. This leader node is responsible for making decisions, coordinating actions, and maintaining system stability. There are several advantages to using election algorithms in distributed systems:

Fault Tolerance: In a distributed system, nodes can fail due to various reasons such as hardware failures, network issues, or software errors. An election algorithm ensures that if the current leader node fails, a new leader can be elected quickly. This promotes fault tolerance and helps maintain the system's functionality even in the presence of failures.

Load Balancing: Election algorithms can be designed to distribute the workload evenly among nodes. By electing a new leader based on factors such as the current load of each node, the system can balance the processing and communication load, preventing any single node from becoming overwhelmed.

Coordination: A leader or coordinator node helps manage and coordinate various tasks within the distributed system. By electing a leader, the system can avoid conflicts and ensure that tasks are performed in a synchronized and organized manner.

Reduced Communication: Election algorithms typically involve communication between nodes to determine the new leader. However, once a leader is elected, it can reduce the need for constant communication between all nodes for decision-making. Instead, nodes can communicate with the leader, simplifying the overall communication structure.

Efficient Decision Making: Having a single leader can expedite decision-making processes in the system. The leader can make critical choices without requiring consensus from all nodes, which can help streamline operations and reduce delays.

Consistency and Replication: In systems that use replication for data consistency and fault tolerance, a centralized leader can help maintain consistency by coordinating data updates across replicas. This ensures that all nodes have a consistent view of the data.

BULLY ALGORITHM

This algorithm was proposed by Garcia-Molina.

When the process notices that the coordinator is no longer responding to requests, it initiates an election. A process, P, holds an election as follows:

(I) P sends an ELECTION message to all processes with higher numbers.

(II) If no one responds, P wins the election and becomes the coordinator.

(III) If one of the higher-ups' answers, it takes over. P's job is done.

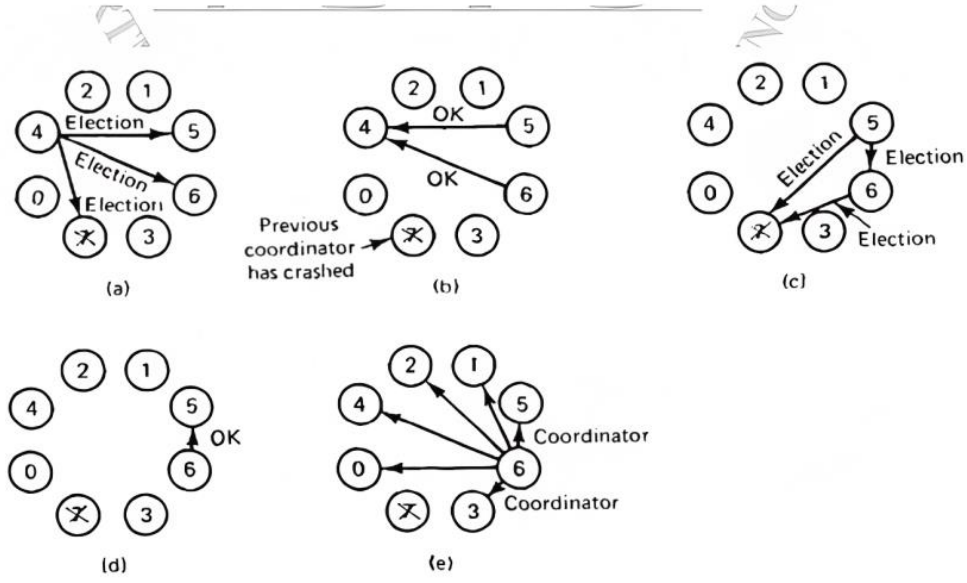
a) A process can get an ELECTION message at any time from one of its lower numbered colleagues.

b) When such a message arrives, the receiver sends an OK message back to the sender to indicate that he is alive and will take over. The receiver then holds an election, unless it is already holding one.

c) All processes give up except one that is the new coordinator. It announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator.

d) If a process that was previously down comes back up, it holds an election. If it happens to the highest numbered process currently running, it will win the election and take over the coordinator's job. Thus, the biggest guy in town always wins, hence the name "bully algorithm".

e) Example:



Comparison of Bully algorithm and Ring based Algorithm

Feature	Bully Algorithm	Ring based Algorithm
Assumptions	All nodes have unique priority numbers.	Nodes are arranged in a logical ring.
Election process	When a node detects that the coordinator has failed, it sends an ELECTION message to all nodes with higher priority numbers. The node with the highest priority number that responds to the ELECTION message becomes the new coordinator.	When a node detects that the coordinator has failed, it sends an ELECTION message to its successor in the ring. If the successor is down, the sender skips over the successor and sends the ELECTION message to the next node in the ring. This process continues until a node is found that is up and running. The node that receives the ELECTION message first becomes the new coordinator.
Efficiency	More efficient than the Ring based algorithm.	Less efficient than the Bully algorithm.
Fault-tolerance	Less fault-tolerant than the Ring based algorithm.	More fault-tolerant than the Bully algorithm.

5. **List the goals of JINI. What are CORBA services? How does operating system support for distributed system?**

- JINI is a Java-based technology developed by Sun Microsystems (now Oracle) to create distributed systems in a network environment. It allows devices and services to dynamically join and leave a network, forming a flexible and adaptive distributed environment.

- Ease of Use: Jini aims to make the creation and interaction of distributed services as easy as using local services. It provides a simple programming model that abstracts the complexities of distributed computing, allowing developers to focus on the business logic rather than the intricacies of network communication.
- Dynamic Discovery: Jini promotes the concept of dynamic service discovery. Services register themselves with the network, and clients can dynamically discover and use these services without prior knowledge of their locations.
- Network Mobility: Jini supports mobility in a network by allowing services to be moved from one location to another without causing disruptions to clients. Clients can access services irrespective of their physical location, promoting flexibility and adaptability.

- Interoperability: Jini services can be written in any language as long as they adhere to the Jini programming model. This enables heterogeneous systems to work together seamlessly.
- Security: Jini provides mechanisms for secure communication and authentication between services and clients. This helps protect sensitive data and prevent unauthorized access.
- Scalability: Jini's architecture is designed to handle large-scale distributed systems with ease. The dynamic discovery and mobility features allow the system to scale up or down as needed.
- Fault Tolerance: Jini supports fault tolerance by enabling services to reappear in the network after recovering from failures. This ensures that critical services are always available even in the face of hardware or software failures.

CORBA (Common Object Request Broker Architecture) is a middleware platform that allows distributed objects to communicate with each other. CORBA services are a set of standard services that are provided by CORBA implementations.

CORBA services are:

a) Naming service: Provides a directory of objects in a distributed system.
b) Event service: Provides a way for objects to subscribe to and receive events from other objects.
c) Transaction's service: Provides a way for objects to ensure that a set of operations are performed atomically.
d) Security service: Provides a way for objects to authenticate each other and to encrypt their communications.
e) Concurrency Control Service: Helps manage concurrency and synchronization issues in a distributed environment.

Operating system support for distributed systems:

a) Operating systems provide a variety of features that support distributed systems, including:
b) Networking support: Provides a way for computers to communicate with each other over a network.
c) Process management: Provides a way for the operating system to manage the execution of multiple processes on a single computer.
d) Memory management: Provides a way for the operating system to manage the memory of multiple processes.
e) File system: Provides a way for multiple processes to access files on a shared disk.
f) Security: Provides a way to protect data and resources from unauthorized access.

6. Explain Byzantine general problem to handle faulty process with example. Describe any one failure recovery technique.

The Byzantine Generals Problem (BGP) is a theoretical problem in computer science that asks how a group of unreliable computers can reach a consensus on a decision, even if some of the computers are Byzantine, meaning that they may behave arbitrarily and maliciously. The problem is named after a thought experiment in which a group of Byzantine generals are camped outside an enemy city. The generals need to decide whether to attack the city, but they cannot communicate directly with each other. Instead, they must communicate through messengers.

If all of the generals are loyal, then they can easily reach a consensus on whether to attack. However, if even one of the generals is Byzantine, then the problem becomes much more difficult. The Byzantine general could send false messages to the other generals, in an attempt to sow discord and prevent them from reaching a consensus.

One way to solve the Byzantine Generals Problem is to use a voting algorithm. In a voting algorithm, each general vote on whether to attack or retreat. If a majority of the generals vote to attack, then the attack proceeds. However, if a majority of the generals vote to retreat, then the attack is canceled. The voting algorithm can be made more robust to Byzantine failures by using a quorum system. A quorum is a set of generals that is large enough to make a decision, even if some of the generals are Byzantine. For example, a quorum of three generals would be sufficient, even if one of the generals is Byzantine.

Another way to solve the Byzantine Generals Problem is to use a consensus algorithm. A consensus algorithm is a more sophisticated algorithm that takes into account the possibility of Byzantine failures. Consensus algorithms typically require more communication between the generals, but they can be more robust to Byzantine failures.

Failure recovery is the process of detecting and recovering from failures in a distributed system. Failures can occur in any part of a distributed system, including the hardware, software, and network. Failure recovery techniques are used to ensure that the system continues to function even in the event of a failure.

There are a variety of failure recovery techniques, including:

a) Checkpointing: Checkpointing is a technique for periodically saving the state of a system. If a failure occurs, the system can be restored to its last checkpointed state.

b) Replication: Replication is a technique for duplicating data and processes across multiple nodes in a system. If a node fails, the data and processes can be restored from the replicas.

c) Reconfiguration: Reconfiguration is a technique for dynamically changing the configuration of a system in response to a failure. For example, if a node fails, the system can be reconfigured to distribute the load of the failed node to the remaining nodes.

d) Fault tolerance: Fault tolerance is a technique for designing a system to be able to withstand failures. Fault-tolerant systems typically use a combination of techniques, such as checkpointing, replication, and reconfiguration.

7. Explain with algorithmic steps, how token ring algorithm works for mutual exclusion in distributed system.

The Token Ring algorithm is a distributed mutual exclusion algorithm that allows processes in a distributed system to access a shared resource in a coordinated and synchronized manner. The algorithm employs a "token," which is a special message that circulates among the processes. A process holding the token has the right to access the shared resource, ensuring that only one process can access the resource at a time. Here's how the Token Ring algorithm works with algorithmic steps:

Initialization:

- Each process is assigned a unique identifier.
- The processes are organized in a logical ring topology, where each process is connected to its neighbors.

Token Creation and Circulation:

- A token is initially created and assigned to a designated process (usually the process with the lowest identifier).
- The token circulates among the processes in a predetermined order, usually in the direction of increasing process identifiers.

Process Request for Mutual Exclusion:

- When a process wants to access the shared resource, it waits until it receives the token.

Token Handling by the Requesting Process:

- When a process holding the token receives a request for mutual exclusion, it releases the token and forwards it to the next process in the ring.

Token Reception and Resource Access:

- The requesting process receives the token and gains the right to access the shared resource.
- It performs its critical section (accesses the shared resource) while holding the token.

Exiting Critical Section and Token Forwarding:

- After the critical section, the process releases the token and forwards it to the next process in the ring.

Token Circulation Continues:

- The token continues to circulate among the processes, and other processes waiting for mutual exclusion will eventually receive the token and access the resource.

Termination:

- The processes can continue to request mutual exclusion as needed, and the token will circulate as long as there are processes requesting access.

The Token Ring algorithm ensures that only one process at a time can hold the token and access the shared resource. This guarantees mutual exclusion and prevents concurrent access to the critical section. The Algorithm also ensures fairness by giving each process an equal opportunity to access the shared resource.

8. **Define lock in concurrency control. How can concurrency be controlled in distributed transaction? What situation does lead to distributed deadlock?**

A lock is a synchronization mechanism used to coordinate access to shared resources in a multi-threaded or multi-process environment. Locks help prevent multiple threads or processes from simultaneously accessing the same resource, which could lead to data corruption or inconsistent results. Locks are used to enforce mutual exclusion, ensuring that only one thread or process can access a critical section of code or a shared resource at a time.

There are two types of locks used in concurrency control:

Shared Lock (Read Lock): This type of lock allows multiple threads or processes to access a resource simultaneously for read-only operations. It's used when multiple entities can safely read the same data concurrently without causing conflicts.

Exclusive Lock (Write Lock): This type of lock ensures exclusive access to a resource, preventing any other thread or process from accessing it, even for read operations. It's used when a thread or process intends to modify the data, preventing other threads or processes from accessing the data simultaneously and causing inconsistencies.

In distributed transactions, concurrency control becomes even more complex due to the presence of multiple nodes, potentially distributed across different machines or locations. Controlling concurrency in a distributed transaction involves maintaining the ACID properties (Atomicity, Consistency, Isolation, Durability) while allowing multiple transactions to proceed concurrently. Concurrency can be controlled in the following manner:

Two-Phase Locking (2PL): This technique ensures that transactions acquire locks in two phases - an expanding phase (locks are acquired) and a shrinking phase (locks are released). It guarantees serializability and helps prevent conflicts between transactions.

Timestamp Ordering: Transactions are assigned timestamps based on their start times. A transaction can proceed only if its timestamp is the smallest among conflicting transactions' timestamps. This approach ensures that transactions are executed in a serializable order.

Concurrency Control Protocols: Distributed databases implement protocols like Serializable Snapshot Isolation (SSI) or Serializable Snapshot Database (SSD) to ensure serializability of transactions while allowing for concurrency.

Distributed deadlocks occur when two or more transactions, each holding locks on resources that the other transaction needs, are unable to proceed. In a distributed environment, a deadlock can arise due to:

Circular Wait: Each transaction is waiting for a resource held by another transaction in a circular manner, creating a deadlock.

Communication Delays: In a distributed system, network delays or communication failures can lead to situations where transactions appear to be deadlocked even though they are not.

Inconsistent Locking: Inconsistent locking protocols or a lack of coordination between different nodes can result in distributed deadlocks.

9. Write short notes on Heterogeneity in distributed system.

Heterogeneity in a distributed system refers to the coexistence of a diverse range of components, encompassing varying hardware configurations, operating systems, programming languages, and communication protocols. This diversity creates challenges while also presenting opportunities for building robust and adaptable systems. Such systems consist of a mix of devices and technologies due to reasons such as legacy systems, specific functional needs, or preferences for certain hardware platforms.

Interoperability is major concern in heterogeneous environments as seamless communication and collaboration among disparate components require addressing differing communication protocols, data formats, and encoding schemes.

Middleware solutions play a pivotal role in managing this complexity by offering a layer of abstraction that conceals the underlying differences and provides standardized interfaces for interaction. Adaptation and translation mechanisms come into play to facilitate

communication between components that operate on distinct protocols or handle data in dissimilar formats.

In addition to technical challenges, managing performance and scalability becomes critical as components may exhibit varying processing capabilities. Resource management and load balancing strategies are employed to optimize the overall performance of heterogeneous systems. Despite the complexities, heterogeneity also brings flexibility and innovation. Organizations can select components that best suit their requirements, and the integration of new technologies is possible without necessitating an overhaul of the entire system. While heterogeneity enhances system capabilities, it also amplifies design, development, and maintenance intricacies, demanding careful debugging, troubleshooting, and security measures.

10. Write short notes on Rendezvous concept and implementation.

The rendezvous concept is a fundamental synchronization mechanism in concurrent computing that ensures coordinated interactions between multiple processes or threads. It involves orchestrating processes to meet at a predetermined point in their execution before proceeding further. This coordination is often achieved using synchronization primitives like semaphores. For instance, consider two processes, A and B, needing to synchronize their actions. Process A performs some initial computations, signals its readiness by releasing a semaphore, and then waits on another semaphore for Process B. Similarly, Process B follows a similar sequence: it performs its tasks, signals its readiness, and then awaits the signal from Process A. Only when both processes have signaled and are waiting for each other, the semaphores allow them to proceed concurrently. This mechanism is crucial in scenarios where specific order and synchronized interaction among concurrent entities are necessary, such as in parallel algorithms, producer-consumer problems, or distributed communication patterns. Rendezvous helps prevent race conditions, ensure orderly execution, and maintain the desired sequence of operations in a concurrent environment.

11. Write short notes on Flat versus nested locks.

Flat Locks:Flat locks, often referred to as simple locks, provide a direct and flexible approach to concurrency management. In this model, threads or processes can individually acquire locks, granting control over fine-grained synchronization. While their simplicity is advantageous, they demand meticulous consideration of the sequence in which locks are acquired and released to avoid potential deadlocks. Flat locks are best suited for scenarios where granularity is paramount, and the risk of deadlocks can be mitigated through well-defined coding practices and coordination efforts.

Nested Locks:Nested locks introduce a structured hierarchy into the locking mechanism, ensuring a predefined order for acquiring locks. When a higher-level lock is obtained, it inherently encapsulates all subordinate locks within its hierarchy. This design inherently thwarts deadlock scenarios by virtue of its regulated lock acquisition sequence. However, the application of nested locks entails the navigation of a more complex management process for the hierarchy, and their usage may introduce performance overhead due to the requirement of acquiring locks hierarchically. Nested locks find their stride in situations necessitating meticulous lock acquisition sequencing to prevent deadlocks, although their implementation and upkeep may involve a greater level of complexity.

12. Write short notes on Process Resilience.

Process resilience refers to the ability of a system's individual processes to withstand and recover from various failures, errors, or disruptions without compromising the overall functionality of the system. In a distributed computing environment, where processes can be distributed across multiple nodes or machines, process resilience is crucial for maintaining system availability and reliability. Resilient processes are designed to gracefully handle scenarios such as hardware failures, software bugs, communication errors, and resource constraints. This involves implementing fault detection mechanisms, error recovery strategies, and strategies for handling resource allocation and contention issues. Process resilience also encompasses strategies like checkpointing, where process states are periodically saved to enable recovery in case of failure, and replication, where processes are duplicated across different nodes to enhance availability. By ensuring process

resilience, distributed systems can continue to operate effectively despite the presence of failures or unexpected events, contributing to the overall stability and robustness of the system.

Process resilience is a fundamental aspect of building reliable and fault-tolerant distributed systems. It involves not only the design and implementation of mechanisms to handle failures but also the proactive identification of potential vulnerabilities and weaknesses in the system. Achieving process resilience often requires a combination of architectural choices, coding practices, and deployment strategies. This might include designing processes that can recover their state after a failure, implementing error-handling mechanisms that gracefully degrade system functionality, and employing redundancy through replication or load balancing to ensure uninterrupted service. Ultimately, process resilience contributes to the system's ability to provide consistent and dependable services even when confronted with unexpected challenges, bolstering user confidence and satisfaction.