Short Notes DSCC

UNIT-I

- [ ] Introduction to Distributed Systems:
- [ ] Characteristics of Distributed Systems-Introduction,
- [ ] Examples of Distributed systems (Client server,
- [ ] peer to peer,
- [ ] grid and cloud computing),
- [ ] Advantages of distributed systems,
- [ ] System models - Introduction,
- [ ] Architectural and Fundamental models,
- [ ] Networking and Internetworking,
- [ ] Interposes Communication (message passing and shared memory),
- [ ] Distributed objects and Remote Method Invocation,
- [ ] RPC,
- [ ] Events and notifications,
- [ ] Case study-Java RMI.
- [ ] UNIT-II
- [ ] Synchronization:
- [ ] Time and Global States-Introduction,
- [ ] Logical Clocks,
- [ ] Synchronizing physical clocks events and Process states,
- [ ] Synchronizing physical clocks,
- [ ] logical time and logical clocks,
- [ ] global states,
- [ ] distributed debugging,
- [ ] Coordination and Agreement:
- [ ] Distributed mutual exclusion,
- [ ] Elections,
- [ ] Multicast communication,
- [ ] consensus and related problems.
- [ ] Here are comprehensive notes with **10 points** for each topic in the **Distributed Systems and Cloud Computing (DSCC)** syllabus.

UNIT I: Introduction to Distributed Systems

1. Characteristics of Distributed Systems:

1. **Concurrency**: Supports multiple simultaneous processes and user interactions.
2. **No Global Clock**: Lacks a unified time reference, requiring synchronization mechanisms.
3. **Independent Failures**: Individual node failures do not cause the entire system to fail.
4. **Scalability**: Can efficiently grow by adding more nodes to handle increased loads.
5. **Heterogeneity**: Integrates diverse hardware and software platforms seamlessly.
6. **Transparency**: Users experience the system as a single entity, unaware of the underlying complexity.
7. **Resource Sharing**: Enables the sharing of resources (e.g., files, processing power) among nodes.
8. **Openness**: The system is open to new components and services, allowing for integration and flexibility.
9. **Security**: Requires mechanisms for secure communication and data protection across nodes.
10. **Interactivity**: Allows real-time interaction between users and services, enhancing user experience.

2. Examples of Distributed Systems:

1. **Client-Server Model**: Servers provide resources or services to multiple clients (e.g., web services).
2. **Peer-to-Peer (P2P)**: All nodes share equal responsibility and resources (e.g., file-sharing networks like BitTorrent).
3. **Grid Computing**: Combines resources from various locations to perform large-scale computations (e.g., SETI@home).
4. **Cloud Computing**: Delivers on-demand resources and services over the internet (e.g., AWS, Google Cloud).
5. **Distributed Databases**: Multiple databases located at different sites but provide a unified view (e.g., NoSQL databases).
6. **Microservices Architecture**: Applications are built as a collection of loosely coupled services (e.g., Netflix architecture).
7. **Distributed File Systems**: File systems that allow files to be stored across multiple servers (e.g., Google File System).
8. **Sensor Networks**: Networks of distributed sensors for monitoring environmental data (e.g., smart cities).
9. **Blockchain**: A distributed ledger technology that allows multiple parties to maintain a shared database (e.g., Bitcoin).
10. **Content Delivery Networks (CDN)**: Distributes content across multiple servers to improve access speed (e.g., Akamai).

3. Advantages of Distributed Systems:

1. **Improved Performance**: Increased throughput and response times due to parallel processing.
2. **Fault Tolerance**: Redundancy and replication ensure system reliability in case of failures.
3. **Scalability**: Easy to add resources without downtime, allowing growth with demand.
4. **Resource Utilization**: Efficient use of resources by pooling together distributed nodes.
5. **Cost Efficiency**: Leverages commodity hardware to reduce infrastructure costs.
6. **Geographical Distribution**: Resources can be located near users to decrease latency.
7. **Flexibility**: Can accommodate different services and components dynamically.
8. **Ease of Maintenance**: Individual components can be updated or replaced without system-wide disruptions.

9. **Enhanced Collaboration**: Facilitates collaboration across organizations or departments by sharing resources.
10. **Modularity**: Systems can be designed with modular components, making it easier to develop and maintain.

4. System Models:

1. **Architectural Models**:
   - Client-Server
   - Peer-to-Peer
   - Multi-tier Architecture
2. **Fundamental Models**:
   - Interaction Models: Describes how processes communicate (synchronous/asynchronous).
   - Failure Models: Types of failures like crash, omission, and timing failures.
3. **Communication Models**: Defines how processes share information (message passing vs. shared memory).
4. **Consistency Models**: Determines how updates are propagated and viewed (strong vs. eventual consistency).
5. **Security Models**: Addresses authentication, authorization, and data integrity.
6. **Replication Models**: Strategies for replicating data across nodes for availability.
7. **Location Models**: Determines how nodes and resources are located and accessed.
8. **Resource Management Models**: How resources are allocated and managed among processes.
9. **Data Distribution Models**: Methods for distributing data across nodes (partitioning and replication).
10. **Service-Oriented Architecture (SOA)**: A design approach based on the use of services to support software.

5. Networking and Internetworking:

1. **Network Topologies**: Describes how nodes are connected (star, ring, mesh).
2. **Protocols**: Defines rules for communication (TCP/IP, HTTP, FTP).
3. **Network Types**: Distinguishes between LAN, WAN, MAN, and the internet.
4. **Routing and Switching**: Techniques for directing data across networks.
5. **Network Security**: Protects data in transit through encryption and secure protocols (SSL/TLS).
6. **Quality of Service (QoS)**: Ensures reliable service and performance metrics.
7. **Bandwidth Management**: Controls data flow to prevent congestion and ensure efficient use of resources.
8. **Network Addressing**: IP addressing schemes and the role of DNS.
9. **Network Management**: Tools and techniques for monitoring and managing network performance.
10. **Interoperability**: Ability of different systems to work together across various networks.

6. Interprocess Communication:

1. **Message Passing**: Processes communicate by sending and receiving messages, supporting both synchronous and asynchronous modes.
2. **Shared Memory**: Processes share a common memory space for communication, requiring synchronization mechanisms (e.g., semaphores).
3. **Remote Procedure Calls (RPC)**: Allows executing procedures on remote machines as if they were local calls.
4. **Sockets**: Provides an interface for network communication between processes over TCP/IP.
5. **Streams**: Continuous flow of data between processes, supporting byte-oriented communication.
6. **Pipes**: Unidirectional data channels for communication between processes, often used in UNIX systems.
7. **Events and Notifications**: Mechanisms to notify processes about changes or occurrences (e.g., publish/subscribe models).
8. **Signals**: Notifications sent to processes to indicate events (e.g., interrupt handling).
9. **Inter-Thread Communication**: Mechanisms like mutexes and condition variables for communication between threads within the same process.
10. **Data Serialization**: Converting data structures into a format suitable for transmission (e.g., JSON, XML).

7. Distributed Objects and Remote Method Invocation (RMI):

1. **Distributed Objects**: Objects that can be accessed from multiple nodes in a distributed system.
2. **Remote Method Invocation (RMI)**: Java-based technology that allows method calls on remote objects as if they were local.
3. **Stubs and Skeletons**: Client-side and server-side components in RMI that facilitate communication.
4. **Marshalling and Unmarshalling**: Process of packaging data for transmission and extracting it at the destination.
5. **Java Naming and Directory Interface (JNDI)**: Used to look up remote objects in RMI.
6. **Exception Handling**: Managing exceptions that occur during remote method calls.
7. **Security Policies**: Configuring security settings for RMI applications (e.g., access control).
8. **Serialization**: Converting objects into a byte stream for transmission over a network.
9. **RMI Registry**: A service that allows remote objects to be registered and looked up by clients.
10. **Performance Considerations**: Issues related to latency and throughput in RMI.

8. RPC (Remote Procedure Call):

1. **Definition**: A protocol that allows a program to execute a procedure on a remote server as if it were local.
2. **Client-Server Architecture**: RPC involves a client sending a request to a server to execute a function.
3. **Stub Mechanism**: Client-side stub prepares the request, while server-side stub processes it.
4. **Marshalling**: Packing the parameters into a message format for transmission.
5. **Unmarshalling**: Extracting the parameters on the server side.
6. **Transport Protocols**: Commonly uses TCP or UDP for communication.
7. **Error Handling**: Mechanisms to handle errors in remote calls (e.g., timeouts, retries).
8. **Synchronous vs. Asynchronous RPC**: Synchronous blocks the client until the response is received; asynchronous allows the client to continue processing.
9. **Security**: Authentication and encryption are vital for securing RPC communications.
10. **Performance Issues**: Network latency and overhead can affect the efficiency of RPC.

9. Events and Notifications:

1. **Event-Driven Architecture**: Components communicate by producing and responding to events.
2. **Event Producers and Consumers**: Producers generate events, and consumers act upon them.
3. **Publish-Subscribe Model**: Decouples event producers from consumers, allowing for flexible communication.
4. **Notification Services**: Systems that manage event notifications and delivery (e.g., JMS).
5. **Event Queues**: Buffering events for processing by consumers.
6. **Asynchronous Communication**: Events are processed independently of the producer's execution flow.
7. **Event Filtering**: Mechanisms to control which events are sent to which consumers.

8. **Event Processing**: Techniques for analyzing and responding to events in real-time (e.g., Complex Event Processing).
9. **Reliability and Durability**: Ensuring events are not lost during transmission or processing.
10. **Use Cases**: Common in systems like messaging apps, real-time monitoring, and

notification systems.

10. Case Study – Java RMI:

1. **Overview of Java RMI**: Allows Java applications to invoke methods on objects located remotely.
2. **Architecture**: Involves client stubs, server skeletons, and RMI registry for object discovery.
3. **Use Cases**: Ideal for building distributed applications that require remote object interaction.
4. **Serialization**: Automatic handling of object serialization for parameter passing.
5. **Exception Handling**: Provides mechanisms for handling remote exceptions.
6. **Security Features**: Supports SSL and access control for secure remote communications.
7. **JNDI Integration**: Allows looking up remote objects by name.
8. **Configuration**: Setup of RMI services, including port configuration and security policies.
9. **Performance**: Considerations for optimizing remote calls (e.g., caching).
10. **Limitations**: Constraints related to network latency and complexity of managing distributed objects.

UNIT II: Synchronization

1. Time and Global States:

1. **Global State**: Represents the state of a distributed system at a given time.
2. **Snapshot Algorithm**: Captures a consistent global state of the system without halting processes.
3. **Causality**: Understanding the cause-effect relationship between events in a distributed system.
4. **Logical Time**: Uses a counter to order events without relying on physical clocks (e.g., Lamport timestamps).
5. **Physical Time**: Actual wall-clock time used to coordinate events across distributed nodes.
6. **Global Clock Synchronization**: Techniques to synchronize physical clocks across distributed nodes (e.g., NTP).
7. **Logical Clocks**: Mechanisms to order events based on logical time (e.g., vector clocks).
8. **Distributed Debugging**: Techniques for debugging distributed systems, often using global states and event tracing.
9. **State Consistency**: Ensuring that global states reflect a consistent view across all nodes.
10. **Time-Stamping**: Assigning timestamps to events for ordering and consistency.

2. Logical Clocks:

1. **Definition**: A mechanism to order events in a distributed system without a global clock.
2. **Lamport Timestamps**: Assigns a numerical timestamp to each event to maintain a causal ordering.
3. **Vector Clocks**: An extension of Lamport timestamps to capture causality among distributed events.
4. **Clock Synchronization**: Techniques to synchronize logical clocks across nodes.
5. **Ordering Events**: Determines the causal relationship between events based on timestamps.
6. **Event Categories**: Classifies events into concurrent, causal, or unrelated based on timestamps.
7. **Algorithm Implementation**: Common algorithms for implementing logical clocks.
8. **Applications**: Used in distributed databases, version control systems, and concurrent programming.
9. **Limitations**: Challenges related to overhead and complexity in maintaining logical clocks.
10. **Comparison**: Differences between logical clocks and physical clocks in distributed systems.

3. Synchronizing Physical Clocks:

1. **Clock Drift**: The gradual deviation of a clock from the correct time due to inaccuracies.
2. **Network Time Protocol (NTP)**: A widely used protocol for synchronizing clocks over a network.
3. **Time Servers**: Special servers that provide time information to synchronize client clocks.
4. **Stratum Levels**: NTP hierarchy indicating the distance from an authoritative time source.
5. **Synchronization Algorithms**: Techniques for adjusting local clocks based on server time.
6. **Precision and Accuracy**: Measures of how closely synchronized clocks are to the actual time.
7. **Latency Considerations**: Network delays that affect synchronization accuracy.
8. **Clock Adjustment Techniques**: Methods to correct clock differences (e.g., stepping, slewing).
9. **Synchronization Challenges**: Issues such as network delays, variable latency, and failures.
10. **Applications**: Importance of synchronized clocks in financial systems, telecommunications, and distributed databases.

4. Logical Time and Logical Clocks:

1. **Logical Time**: A way to order events based on their occurrence rather than actual time.
2. **Event Ordering**: Establishes a sequence for events that is independent of physical time.
3. **Lamport Timestamps**: A simple method of implementing logical time using counters.
4. **Vector Clocks**: Extends Lamport timestamps to capture more complex causal relationships.
5. **Concurrency Detection**: Identifying concurrent events based on logical clock values.
6. **Causality Tracking**: Determining which events influence others using logical clocks.
7. **Implementation Challenges**: Complexity in maintaining logical clocks across distributed nodes.
8. **Impact on Consistency**: Logical clocks contribute to ensuring consistency in distributed systems.
9. **Applications**: Used in distributed databases, messaging systems, and collaborative applications.
10. **Comparison**: Logical clocks vs. physical clocks in maintaining order and causality.

5. Global States:

1. **Definition**: The complete state of a distributed system at a specific point in time.
2. **Consistency Models**: Rules for maintaining a consistent global state across distributed nodes.
3. **Snapshot Algorithms**: Techniques for capturing global states without stopping processes.
4. **Causal Relationships**: Understanding how events affect the global state.

5. **State Capture**: Methods for collecting state information from all nodes.
6. **Global Snapshot Properties**: Must reflect a consistent view of the system.
7. **Applications**: Used in debugging, fault tolerance, and state recovery.
8. **Challenges**: Difficulty in capturing a consistent global state due to concurrent processes.
9. **State Representation**: How global states are represented and stored.
10. **Impact on System Design**: Considerations for designing systems to maintain global state consistency.

6. Distributed Debugging:

1. **Definition**: Techniques for debugging programs running on distributed systems.
2. **Challenges**: Complexity due to concurrency, communication delays, and lack of global state visibility.
3. **Event Tracing**: Recording events to analyze system behavior and interactions.
4. **Global States in Debugging**: Using snapshots to understand system state at a specific time.
5. **Causal Analysis**: Understanding how events relate to one another in distributed debugging.
6. **Debugging Tools**: Software and frameworks that assist in debugging distributed applications.
7. **Monitoring and Logging**: Collecting data from distributed processes for analysis.
8. **Replay Mechanisms**: Techniques for replaying events to understand issues.
9. **Error Localization**: Identifying the source of errors in distributed systems.
10. **User Interfaces**: Tools for visualizing and interacting with distributed debugging data.

7. Coordination and Agreement:

1. **Definition**: Mechanisms that allow distributed processes to coordinate their actions and reach agreements.
2. **Distributed Mutual Exclusion**: Techniques to ensure only one process can access a resource at a time.
3. **Election Algorithms**: Processes for electing a coordinator in distributed systems (e.g., Bully algorithm).
4. **Consensus Protocols**: Algorithms for reaching agreement among distributed processes (e.g., Paxos, Raft).
5. **Multicast Communication**: Sending messages to multiple recipients simultaneously.
6. **Atomic Broadcast**: Guarantees that messages are delivered to all processes in the same order.
7. **Resource Allocation**: Strategies for allocating resources in a distributed environment.
8. **Failure Handling**: Mechanisms for managing process failures during coordination.
9. **Consistency Models**: Ensuring all processes agree on the same state or value.
10. **Use Cases**: Applications of coordination and agreement in distributed databases, cloud computing, and collaborative systems.

8. Distributed Mutual Exclusion:

1. **Definition**: A mechanism to ensure that multiple processes do not access a critical section simultaneously.
2. **Centralized Approach**: A central coordinator grants access to the critical section (e.g., centralized locking).
3. **Token Ring Algorithm**: A token circulates in a logical ring, granting access to the critical section.
4. **Ricart-Agrawala Algorithm**: A request-based approach using timestamps for granting access.
5. **Lamport's Bakery Algorithm**: A ticket-based mechanism for mutual exclusion.
6. **Performance Metrics**: Evaluating efficiency based on response time and resource utilization.
7. **Scalability**: Ability to maintain mutual exclusion with increasing numbers of processes.
8. **Fairness**: Ensuring all processes get a chance to access the critical section.
9. **Deadlock Prevention**: Mechanisms to avoid deadlocks in mutual exclusion protocols.
10. **Use Cases**: Important in databases, file systems, and concurrent programming.

9. Elections:

1. **Definition**: Algorithms to select a leader or coordinator in distributed systems.
2. **Bully Algorithm**: Processes with higher IDs claim leadership, forcing lower IDs to step down.
3. **Ring Algorithm**: A token circulates in a logical ring, and processes vote to select a leader.
4. **Performance Metrics**: Efficiency and speed of the election process.
5. **Failure Handling**: Detecting and handling failures of the elected leader.
6. **Fairness**: Ensuring all processes have an equal opportunity to become leaders.
7. **Scalability**: Ability to elect leaders in large distributed systems.
8. **Complexity**: Analyzing time and message complexity of election algorithms.
9. **Use Cases**: Leader election is critical in distributed databases, cluster management, and resource allocation.
10. **Variations**: Different election algorithms designed for specific system requirements (e.g., crash vs. Byzantine failures).

10. **

Consensus Protocols**:

1. **Definition**: Algorithms that enable a group of distributed processes to agree on a common value.
2. **Paxos Algorithm**: A well-known consensus protocol for achieving agreement in distributed systems.
3. **Raft Algorithm**: An alternative to Paxos, designed for understandability and practicality.
4. **Quorum Systems**: A method to ensure agreement by requiring a subset of processes to respond.
5. **Failure Tolerance**: Mechanisms to achieve consensus despite process failures.
6. **Message Complexity**: Analyzing the number of messages required to reach consensus.
7. **Performance Metrics**: Evaluating latency and throughput in consensus protocols.
8. **Use Cases**: Critical for distributed databases, cloud computing, and fault-tolerant systems.
9. **Scalability Challenges**: Issues arising in large distributed systems regarding consensus.
10. **Variations**: Adaptations of consensus algorithms for different applications (e.g., asynchronous systems).

Feel free to modify or expand upon these points as needed!