**Basic Concepts – Centralized Parallel and Distributed Algorithms, Examples of distributed systems, Correctness proof of distributed algorithms; Design issues in a distributed system; Distributed Computing Models-Distributed Operating System, Network Operating System, Middleware based Distributed System; Client Server model – 2 tier and n-tier CS model.**

# Basic Concepts of Distributed Systems and Cloud Computing

## 1. Centralized, Parallel, and Distributed Systems

### 1.1 Centralized Systems

- A single computing unit (server) handles all tasks.
- Users access it through terminals.
- Examples: Mainframe computing, early Unix systems.

### 1.2 Parallel Systems

- Multiple processors work together in a tightly coupled manner to perform a task.
- Shared memory or high-speed interconnects allow communication.
- Used in supercomputers and high-performance computing.

### 1.3 Distributed Systems

- Independent computers communicate over a network to achieve a common goal.
- No global clock; processes run asynchronously.
- Examples: Google Cloud, Hadoop, Blockchain networks.

---

## 2. Examples of Distributed Systems

1. **Google Search Engine** – Uses distributed data centers worldwide to provide search results quickly.
2. **Bitcoin and Blockchain** – A decentralized, peer-to-peer ledger system.
3. **Distributed File Systems** – NFS (Network File System), Google File System (GFS).
4. **Cloud Computing Platforms** – Amazon Web Services (AWS), Microsoft Azure.

---

# 3. Correctness Proof of Distributed Algorithms

Distributed algorithms must satisfy:

- **Safety** – The system should not reach an incorrect state.
- **Liveness** – The algorithm should eventually complete its task.
- **Termination** – The algorithm should reach a final state.

### Example of Correctness Proof

For **Mutual Exclusion in Distributed Systems**:

1. **Safety**: Ensure that only one process enters the critical section at a time.
2. **Liveness**: Every requesting process gets a chance to enter the critical section.
3. **Fairness**: No process is indefinitely delayed (no starvation).

---

# 4. Design Issues in a Distributed System

1. **Transparency**
   - **Access Transparency** – Users should not know if data is remote or local.
   - **Location Transparency** – The physical location of resources is hidden.
   - **Replication Transparency** – Users should not be aware of multiple copies of data.
2. **Fault Tolerance**
   - The system should handle failures without affecting overall functionality.
   - Techniques: Replication, checkpointing, message logging.
3. **Scalability**
   - The system should efficiently handle an increasing number of nodes.
4. **Concurrency**
   - Multiple users should access the system simultaneously without conflicts.
5. **Security**
   - Authentication, Authorization, Encryption, and Secure Communication.

---

# 5. Distributed Computing Models

## 5.1 Distributed Operating System (DOS)

- A single OS manages multiple nodes as one unit.
- Example: Amoeba, Plan 9, Google's Borg OS.

## 5.2 Network Operating System (NOS)

- Computers operate independently but communicate via a network.
- Example: Microsoft Windows Server, Novell NetWare.

### 5.3 Middleware-Based Distributed System

- Middleware sits between OS and applications, facilitating communication.
- Example: CORBA, Java RMI, Apache Kafka.

---

## 6. Client-Server Model

The client-server model is the backbone of distributed systems.

### 6.1 2-Tier Model

- Client directly communicates with the server.
- Example: Database applications, where the frontend (client) directly talks to a backend (MySQL, PostgreSQL).

### 6.2 N-Tier Model

- Involves multiple layers like Presentation, Business Logic, and Data Storage.
- Example: Web Applications using Angular (frontend), Spring Boot (backend), and MySQL (database).

---

**Shared memory and message passing systems,**

**synchronous and asynchronous systems, transient and**

**persistent system, point-to-point and publish-subscribe**

**systems; Distributed shared memory and memory**

**consistency models; Group communication - multicast**

**and anycast communication, message ordering and**

**message delivery; Remote Procedure Call (RPC).**

**Case studies – Socket, MPI, JMS and RMI**

# Distributed System and Cloud Computing – Advanced Concepts

## 1. Shared Memory and Message Passing Systems

### 1.1 Shared Memory Systems

- Multiple processes share a common memory space.
- Processes communicate by reading/writing to shared memory.
- Requires synchronization mechanisms like semaphores and locks.
- **Example**: OpenMP (for parallel programming).

### 1.2 Message Passing Systems

- Processes communicate by sending and receiving messages.
- No shared memory; suitable for distributed environments.
- **Example**: MPI (Message Passing Interface), TCP/IP sockets.

**Comparison:**

| Feature | Shared Memory | Message Passing |
|---|---|---|
| Communication Overhead | Low (direct memory access) | High (data transmission over network) |
| Synchronization | Complex (requires locks) | Easier (uses message queues) |
| Scalability | Limited (memory contention) | High (independent processes) |

---

## 2. Synchronous and Asynchronous Systems

### 2.1 Synchronous Systems

- Processes execute in lockstep with a global clock.
- **Advantages**: Predictable execution, easier debugging.
- **Disadvantages**: Blocking delays if one process is slow.
- **Example**: Real-time systems, Distributed databases with two-phase commit.

### 2.2 Asynchronous Systems

- Processes execute independently without a global clock.
- **Advantages**: More flexible, non-blocking operations.
- **Disadvantages**: Harder to debug due to race conditions.
- **Example**: Internet-based systems, email servers.

---

## 3. Transient and Persistent Systems

### 3.1 Transient Systems

- Messages exist only while processes are running.
- If a message is not received immediately, it is lost.

- **Example**: UDP-based communication (video streaming).

### 3.2 Persistent Systems

- Messages are stored until the receiver retrieves them.
- **Example**: Message queues (Kafka, RabbitMQ), Email systems.

---

## 4. Point-to-Point and Publish-Subscribe Systems

### 4.1 Point-to-Point Communication

- Direct communication between sender and receiver.
- Example: TCP sockets, direct RPC calls.

### 4.2 Publish-Subscribe Systems

- A publisher sends messages without knowing specific receivers (subscribers).
- Subscribers register for specific message topics.
- Example: Kafka, JMS (Java Message Service), MQTT.

**Comparison:**

| Feature | Point-to-Point | Publish-Subscribe |
|---|---|---|
| Communication | Direct | Indirect |
| Scalability | Limited | High |
| Use Case | Request-Response | Event-driven systems |

---

## 5. Distributed Shared Memory (DSM) and Memory Consistency Models

### 5.1 Distributed Shared Memory (DSM)

- Simulates a shared memory system across distributed nodes.
- Data consistency must be maintained.
- Example: NUMA (Non-Uniform Memory Access), TreadMarks (a DSM system).

### 5.2 Memory Consistency Models

1. **Strict Consistency** – Read always returns the latest write (ideal but impractical).
2. **Sequential Consistency** – All operations appear in the same order to all processes.
3. **Causal Consistency** – Related writes appear in order, but independent operations may be unordered.
4. **Eventual Consistency** – Updates propagate gradually; used in NoSQL databases (Cassandra, DynamoDB).

---

# 6. Group Communication

## 6.1 Multicast Communication

- One-to-many communication.
- Used for group messaging (e.g., video conferencing).
- Example: IP multicast, PGM (Pragmatic General Multicast).

## 6.2 Anycast Communication

- Sender sends data to the nearest available receiver.
- Example: CDN (Content Delivery Networks) where users get content from the nearest server.

---

# 7. Message Ordering and Message Delivery

## 7.1 Message Ordering

1. **FIFO Ordering** – Messages from a sender are received in the order sent.
2. **Causal Ordering** – Messages related to each other follow a specific order.
3. **Total Ordering** – All messages appear in the same order to all recipients.

## 7.2 Message Delivery

- **Reliable Delivery**: Ensures messages are not lost.
- **Atomic Delivery**: Messages are either delivered to all or none.

---

# 8. Remote Procedure Call (RPC)

- Allows a process to execute code on a remote server as if it were a local function.
- **Steps in RPC:**
    1. Client calls a local stub function.
    2. Stub function marshals (packs) data and sends a request to the server.
    3. Server receives the request, executes the function, and sends back the result.
    4. Client stub receives the response and returns the result.
- **Example Implementations:** gRPC, Java RMI, CORBA.

---

# 9. Case Studies

### 9.1 Socket Programming

- Provides low-level network communication using TCP/UDP.
- Used for building custom communication protocols.
- Example: Python `socket` module, Java `Socket` class.

### 9.2 MPI (Message Passing Interface)

- Used for high-performance parallel computing.
- Provides primitives for sending/receiving messages between processes.
- Example: Used in scientific computing applications.

### 9.3 JMS (Java Message Service)

- A publish-subscribe model for Java applications.
- Supports durable message storage.
- Example: Apache ActiveMQ, RabbitMQ.

### 9.4 RMI (Remote Method Invocation)

- Java's built-in RPC framework for distributed objects.
- Supports method calls over a network.
- Example: A Java server exposes a method, and a remote client calls it.

---

**Clock, events and process state in distributed systems;**

**Physical clock synchronization, Logical clocks and**

**Vector clocks; Distributed Snapshots – Global States;**

**Distributed mutual exclusions; Leader election**

**algorithms; Distributed transactions; Deadlocks in**

**distributed systems;**

# Distributed Systems: Clocks, Synchronization, and Coordination

## 1. Clock, Events, and Process State in Distributed Systems

### 1.1 Clocks in Distributed Systems

- Each node in a distributed system has its own clock.

- Clocks may drift due to different clock speeds, leading to time inconsistencies.

### 1.2 Events in Distributed Systems

- **Local Events**: Actions occurring within a single process (e.g., variable update).
- **Message Passing Events**: Sending and receiving messages between processes.
- **Global Events**: Events affecting multiple processes (e.g., system failure).

### 1.3 Process State

- Each process has its own execution state.
- The state consists of local variables, pending messages, and the logical clock value.

---

## 2. Physical Clock Synchronization

Clock synchronization ensures consistency across nodes.

### 2.1 Christian's Algorithm (Centralized Approach)

- A time server provides clock values to all nodes.
- Each node adjusts its time using:

$$T = T_{\text{server}} + \frac{(T_{\text{request sent}} - T_{\text{response received}})}{2}$$

- Drawback: Single point of failure (if the server crashes).

### 2.2 Berkeley's Algorithm (Average Consensus Approach)

- The master node asks all nodes for their time.
- It computes the average and tells all nodes to adjust accordingly.
- Works well for LANs but not in large-scale distributed systems.

### 2.3 Network Time Protocol (NTP) (Hierarchy-Based Approach)

- A hierarchical system where stratum-1 servers sync with atomic clocks.
- Uses offset and round-trip delay calculations for precision.
- Widely used for internet-scale clock synchronization.

---

## 3. Logical Clocks and Vector Clocks

### 3.1 Logical Clocks (Lamport's Timestamps)

- Instead of real-time, processes use logical timestamps.
- **Rules:**
  1. Each process maintains a counter (`LC`).
  2. Before an event, increment `LC`.
  3. On sending a message, send `LC`.
  4. On receiving a message, update `LC = max(LC, received_LC) + 1`.
- Ensures **causal ordering** but does not detect concurrency.

### 3.2 Vector Clocks

- Each process maintains a vector `[P1, P2, ..., Pn]`.
- When a process sends a message, it updates its own entry.
- On receiving, it updates by taking the **maximum of corresponding values**.
- **Advantage**: Detects concurrency (unlike Lamport's Clocks).
- **Drawback**: Space complexity **O(n)**.

---

## 4. Distributed Snapshots – Global State

- Captures a consistent global state of a distributed system.
- **Chandy-Lamport Algorithm**:
  1. A process initiates the snapshot and records its state.
  2. It sends a marker to all other processes.
  3. When a process receives a marker for the first time, it records its state and forwards the marker.
  4. The process records incoming messages until all markers are received.
- **Use Case**: Checkpointing in distributed databases.

---

## 5. Distributed Mutual Exclusion

Mutual exclusion ensures only one process accesses a critical section (CS) at a time.

### 5.1 Ricart-Agrawala Algorithm (Timestamp-Based Approach)

1. A process sends a request message with a timestamp.
2. Other processes reply with **OK** if they have a later timestamp or are not in CS.
3. Process enters CS after receiving **OK** from all.
4. On exit, it sends a **release** message.

- **Advantage**: Message complexity **O(N)**.
- **Drawback**: High overhead in large systems.

### 5.2 Token-Based Algorithms

- A **unique token** grants access to the CS.

- Token circulates among processes.
- Example: **Token Ring Algorithm** (a logical ring structure).

---

# 6. Leader Election Algorithms

Leader election is essential for coordination in distributed systems.

### 6.1 Bully Algorithm

1. A process detects a leader failure and initiates an election.
2. It sends an **Election** message to higher-ID processes.
3. If no response, it declares itself the leader.
4. If a higher-ID process responds, it takes over.

- **Drawback**: High message overhead.

### 6.2 Ring Algorithm

1. Processes are arranged in a logical ring.
2. A process initiates an election by passing an election message.
3. The highest-ID process is elected as the leader.

- **Advantage**: Less message complexity than Bully Algorithm.

---

# 7. Distributed Transactions

A transaction ensures atomicity in a distributed system.

### 7.1 ACID Properties in Distributed Transactions

- **Atomicity**: Either all operations commit, or none do.
- **Consistency**: The system moves from one valid state to another.
- **Isolation**: Transactions do not interfere.
- **Durability**: Once committed, changes persist.

### 7.2 Two-Phase Commit (2PC)

1. **Prepare Phase**: Coordinator asks all nodes if they can commit.
2. **Commit Phase**: If all nodes agree, commit the transaction; otherwise, abort.

- **Issue**: If the coordinator crashes, the system may hang.

### 7.3 Three-Phase Commit (3PC)

- Adds a **pre-commit phase** before commit.
- Ensures that if a crash occurs, nodes can safely recover.

---

## 8. Deadlocks in Distributed Systems

A **deadlock** occurs when processes wait indefinitely for resources held by others.

### 8.1 Conditions for Deadlock (Coffman's Conditions)

1. **Mutual Exclusion** – Resources are non-shareable.
2. **Hold and Wait** – Processes hold some resources while waiting for others.
3. **No Preemption** – Resources cannot be forcibly taken.
4. **Circular Wait** – A circular chain of waiting exists.

### 8.2 Deadlock Detection Algorithm

- Construct a **Wait-for Graph (WFG)** where nodes represent processes and edges represent dependencies.
- A cycle in the WFG means a deadlock exists.

### 8.3 Deadlock Prevention Techniques

- **Timeout-Based**: Abort transactions if they exceed a threshold.
- **Request All at Once**: Processes request all required resources upfront.
- **Resource Ordering**: Impose an order to prevent circular waits.

---

**Fault Tolerance**

**Basic concepts, fault models, consensus problems and**

**its applications, commit protocols, voting protocols,**

**Checkpointing and recovery, reliable communication**

# Fault Tolerance in Distributed Systems

## 1. Basic Concepts of Fault Tolerance

Fault tolerance is the ability of a system to continue operating correctly despite failures. It ensures system reliability, availability, and maintainability.

### 1.1 Types of Faults

1. **Transient Faults**: Temporary errors that disappear after some time (e.g., network glitches).
2. **Intermittent Faults**: Occur unpredictably due to unstable components (e.g., loose connections).
3. **Permanent Faults**: Hardware or software failures requiring repair or replacement (e.g., disk failure).

### 1.2 Fault Tolerance Techniques

- **Redundancy**: Extra components to take over in case of failure.
- **Replication**: Multiple copies of data or processes running concurrently.
- **Checkpointing**: Saving system state periodically for recovery.
- **Recovery Mechanisms**: Rollback to a consistent state after a failure.

---

## 2. Fault Models

Fault models classify failures based on their behavior and impact.

### 2.1 Crash Fault Model

- A process **fails by stopping** and does not recover.
- Other processes must detect and handle the failure.
- Example: Node failures in cloud computing.

### 2.2 Omission Fault Model

- A process fails to send or receive messages.
- Example: Network packet loss in unreliable channels.

### 2.3 Byzantine Fault Model

- A process behaves arbitrarily, possibly sending conflicting messages.
- **Example**: Malicious attacks, software bugs causing incorrect computations.
- **Solution**: Byzantine Fault Tolerant (BFT) algorithms like PBFT (Practical Byzantine Fault Tolerance).

### 2.4 Timing Fault Model

- A process does not meet timing constraints.
- Example: Real-time systems where delayed responses lead to failures.

---

## 3. Consensus Problems and Applications

Consensus ensures that all non-faulty processes agree on a single decision, even in the presence of failures.

### 3.1 Requirements for Consensus

1. **Termination** – Every correct process eventually decides on a value.
2. **Agreement** – All correct processes agree on the same value.
3. **Integrity** – The agreed value must have been proposed by a process.
4. **Validity** – The decision must be based on actual inputs.

### 3.2 FLP Impossibility Theorem

- In an **asynchronous** system, achieving consensus is impossible if even **one process can fail**.
- **Workaround**: Use probabilistic or partially synchronous models (e.g., Paxos, Raft).

### 3.3 Consensus Algorithms

- **Paxos**: A widely used consensus algorithm in distributed systems.
- **Raft**: Simplifies Paxos by using a leader-based approach.
- **Byzantine Agreement**: Extends consensus to Byzantine faults (e.g., PBFT).

### 3.4 Applications of Consensus

- Distributed databases (e.g., Google Spanner).
- Blockchain (e.g., Bitcoin uses Proof-of-Work for consensus).
- Leader election in distributed systems.

---

## 4. Commit Protocols

Commit protocols ensure atomic transactions in distributed databases.

### 4.1 Two-Phase Commit (2PC)

- Ensures that either all nodes commit or none do.

**Steps in 2PC:**

1. **Prepare Phase**: Coordinator asks participants if they can commit.
2. **Commit Phase**: If all agree, coordinator sends a commit; otherwise, it aborts.

**Drawbacks of 2PC:**

- If the coordinator crashes, the system may hang.
- Requires strict synchronization, leading to high latency.

### 4.2 Three-Phase Commit (3PC)

- Adds a **pre-commit phase** to reduce blocking issues.

**Steps in 3PC:**

1. **Prepare Phase** – Coordinator asks nodes if they are ready.
2. **Pre-Commit Phase** – If all agree, coordinator sends a pre-commit message.
3. **Commit Phase** – If no failures occur, the final commit is sent.

**Advantage of 3PC:**

- **Non-blocking**: Nodes can safely recover from coordinator failure.

---

# 5. Voting Protocols

Voting protocols ensure fault tolerance by requiring agreement from a majority of processes.

## 5.1 Majority Voting Protocol

- A transaction is committed if more than 50% of nodes approve.
- Used in **quorum-based systems** (e.g., ZooKeeper).

## 5.2 Weighted Voting Protocol

- Different nodes have different weights in the voting process.
- Useful in **heterogeneous distributed systems**.

---

# 6. Checkpointing and Recovery

Checkpointing saves the system state periodically to enable recovery from failures.

## 6.1 Types of Checkpointing

1. **Coordinated Checkpointing** – All processes take a checkpoint simultaneously.
    - Avoids inconsistency but requires synchronization.
2. **Uncoordinated Checkpointing** – Processes checkpoint independently.
    - Can lead to **domino effect**, where old checkpoints must be rolled back.
3. **Incremental Checkpointing** – Saves only changes since the last checkpoint.
    - Reduces storage overhead.

## 6.2 Recovery Techniques

- **Rollback-Recovery**: Restores a process to its last checkpoint.
- **Message Logging**: Logs messages for replaying after a failure.

# 7. Reliable Communication in Distributed Systems

Reliable communication ensures that messages are delivered correctly and in order.

## 7.1 Challenges in Reliable Communication

- Network failures (packet loss, duplication, or corruption).
- Process failures (crashes, Byzantine behavior).

## 7.2 Techniques for Reliable Communication

1. **Acknowledgment-based Protocols**
   - **Stop-and-Wait**: Sender waits for an acknowledgment (ACK) before sending the next message.
   - **Sliding Window Protocol**: Allows multiple messages in transit (e.g., TCP).
2. **Retransmission Mechanisms**
   - **Automatic Repeat reQuest (ARQ)**: Retransmits lost packets.
   - **Forward Error Correction (FEC)**: Adds redundancy for error detection.
3. **Ordered Delivery**
   - **FIFO Ordering**: Messages from a sender are received in order.
   - **Causal Ordering**: Messages maintain causality constraints.
   - **Total Ordering**: All recipients receive messages in the same order.

# Summary Table

| Concept | Key Points | Example Algorithms |
|---|---|---|
| **Fault Models** | Crash, Omission, Byzantine, Timing | PBFT for Byzantine faults |
| **Consensus** | Agreement, Integrity, FLP Impossibility | Paxos, Raft, Byzantine Agreement |
| **Commit Protocols** | Ensure atomic transactions | 2PC, 3PC |
| **Voting Protocols** | Majority-based decision making | Quorum-based voting |
| **Checkpointing** | Saves system state | Coordinated, Uncoordinated, Incremental |
| **Reliable Communication** | Ensures correct and ordered delivery | TCP, ARQ, Sliding Window |

**Big Data; Relation between Big Data and Distributed**

**File System; Map Reduce Programming Paradigm.**

**Case Studies: HDFS and Hadoop Map Reduce**

# Distributed File System (DFS) and Big Data

## 1. Distributed File System (DFS)

A **Distributed File System (DFS)** is a system that allows files to be stored across multiple machines while appearing as a single logical file system to users. DFS provides:

- **Scalability**: Handles large amounts of data.
- **Fault tolerance**: Ensures availability despite node failures.
- **Concurrency**: Supports multiple users accessing files simultaneously.

### 1.1 Characteristics of DFS

1. **Transparency**
    - **Access Transparency**: Users access remote files like local files.
    - **Location Transparency**: File location is hidden from users.
    - **Replication Transparency**: Multiple copies of a file exist without user intervention.
    - **Failure Transparency**: System recovers automatically from failures.
2. **Scalability**
    - Handles a growing amount of data and users.
3. **Fault Tolerance**
    - Uses replication, checkpoints, and recovery techniques to handle failures.
4. **Concurrency Control**
    - Ensures consistency when multiple users access or modify files.

---

## 2. Big Data and Its Relation to DFS

### 2.1 What is Big Data?

Big Data refers to massive datasets that are difficult to store, process, and analyze using traditional methods. It is characterized by:

- **Volume**: Large-scale data storage.
- **Velocity**: High-speed data processing.
- **Variety**: Different formats (structured, semi-structured, unstructured).
- **Veracity**: Data accuracy and reliability.
- **Value**: Extracting useful insights from data.

### 2.2 Why DFS for Big Data?

Big Data requires a **scalable and distributed** storage system. DFS efficiently handles:

- **Storage of massive datasets**.
- **Efficient data retrieval** across multiple nodes.
- **Fault tolerance** through replication.
- **Parallel processing support** (used in **MapReduce**).

**Example:** Hadoop Distributed File System (HDFS) is widely used for handling Big Data.

---

# 3. MapReduce Programming Paradigm

**MapReduce** is a programming model for processing large datasets in parallel across a distributed system.

## 3.1 Key Concepts

- **Map Phase**: Divides input data into smaller chunks and processes them in parallel.
- **Shuffle Phase**: Groups and sorts intermediate results.
- **Reduce Phase**: Aggregates and processes the intermediate data.

## 3.2 Example Workflow

**Problem**: Count word occurrences in a large dataset.

1. **Map Phase**:
    - Input: `["Hello world", "Hello Hadoop"]`
    - Output: `(Hello, 1), (world, 1), (Hello, 1), (Hadoop, 1)`
2. **Shuffle Phase**:
    - Groups words together:
      `(Hello, [1,1]), (world, [1]), (Hadoop, [1])`
3. **Reduce Phase**:
    - Aggregates counts:
      `(Hello, 2), (world, 1), (Hadoop, 1)`

---

# 4. Case Studies: HDFS and Hadoop MapReduce

## 4.1 Hadoop Distributed File System (HDFS)

HDFS is a DFS designed for handling Big Data.

### Key Features

- **Master-Slave Architecture**:
    - **NameNode** (Master) manages metadata and file structure.
    - **DataNodes** (Slaves) store actual file blocks.

- **Block Storage**: Files are divided into **blocks (128 MB default)** and stored across multiple nodes.
- **Replication**: Default **3x replication** for fault tolerance.
- **Write-Once-Read-Many (WORM) Model**: Optimized for large sequential reads.

#### HDFS Architecture

```
Client → NameNode → DataNodes (Stores Blocks)
```

- Clients interact with **NameNode** to locate files.
- Data is stored and replicated across **DataNodes**.

### 4.2 Hadoop MapReduce

Hadoop MapReduce uses a DFS (like HDFS) for parallel processing of data.

#### Advantages of Hadoop MapReduce

- **Parallel Execution**: Tasks are executed concurrently on multiple nodes.
- **Fault Tolerant**: Failed tasks are automatically restarted.
- **Scalable**: Works on clusters of thousands of machines.

---

## 5. Summary Table

| Concept | Key Points | Example |
|---|---|---|
| **DFS** | Stores files across multiple nodes | HDFS |
| **Big Data** | Large-scale, high-speed data processing | Apache Spark, Hadoop |
| **MapReduce** | Distributed processing model | Word count, Log analysis |
| **HDFS** | Distributed file storage, Replication | Used in Hadoop |
| **Hadoop MapReduce** | Processes data in parallel | Log analysis, Data mining |

---

**Cloud Computing**

**Definition of Cloud Computing; Cloud Service Models;**

**Cloud Deployment Models; Virtualization Technologies.**

**Case Studies: AWS, GCP and Windows Azure**

# Cloud Computing

## 1. Definition of Cloud Computing

Cloud Computing is the **on-demand** delivery of computing resources such as servers, storage, databases, networking, software, and analytics over the **internet**. It allows users to access and manage resources **without maintaining physical infrastructure**.

### 1.1 Characteristics of Cloud Computing

1. **On-Demand Self-Service** – Users can provision resources as needed.
2. **Broad Network Access** – Services are accessible from anywhere.
3. **Resource Pooling** – Multiple users share computing resources.
4. **Rapid Elasticity** – Resources can scale up/down automatically.
5. **Measured Service** – Users pay only for what they use (pay-as-you-go).

---

## 2. Cloud Service Models (SPI Model)

Cloud services are categorized into **three main models**, also called the **SPI model**:

| Service Model | Description | Examples |
|---|---|---|
| **Software as a Service (SaaS)** | Provides software applications over the internet. Users do not manage the infrastructure. | Gmail, Google Drive, Dropbox |
| **Platform as a Service (PaaS)** | Provides a platform for developers to build applications without managing underlying hardware. | Google App Engine, AWS Elastic Beanstalk |
| **Infrastructure as a Service (IaaS)** | Provides virtualized computing resources like servers, storage, and networking. | AWS EC2, Google Compute Engine |

### 2.1 Comparison of Service Models

- **SaaS**: End-users consume applications via the cloud.
- **PaaS**: Developers deploy and manage applications.
- **IaaS**: IT administrators manage servers, storage, and networking.

---

## 3. Cloud Deployment Models

| Deployment Model | Description | Use Case |
|---|---|---|
| **Public Cloud** | Resources are owned and managed by a third-party provider. Shared among multiple users. | Startups, Web applications |
| **Private Cloud** | Dedicated infrastructure for a single organization. More control and security. | Banking, Healthcare |
| **Hybrid Cloud** | Combination of public and private clouds. Used for sensitive and scalable workloads. | Enterprises, Large-scale businesses |

| Deployment Model | Description | Use Case |
| --- | --- | --- |
| **Community Cloud** | Shared cloud infrastructure for a specific group of organizations. | Government, Research institutions |

### 3.1 Public vs Private Cloud

| Feature | Public Cloud | Private Cloud |
| --- | --- | --- |
| **Security** | Lower | Higher |
| **Cost** | Pay-per-use (cheaper) | Expensive |
| **Scalability** | High | Limited |
| **Control** | Less control | Full control |

---

## 4. Virtualization Technologies

Virtualization is a key technology that enables cloud computing by creating multiple **virtual instances** of hardware or software.

### 4.1 Types of Virtualization

1. **Hardware Virtualization**
   - Uses a **hypervisor** to create Virtual Machines (VMs).
   - **Examples**: VMware, VirtualBox, KVM.
2. **Server Virtualization**
   - Multiple servers run as independent VMs on one physical server.
   - **Example**: AWS EC2 instances.
3. **Storage Virtualization**
   - Abstracts multiple storage devices into a single logical unit.
   - **Example**: Amazon S3, Google Cloud Storage.
4. **Network Virtualization**
   - Virtualizes network resources for better efficiency.
   - **Example**: Software-Defined Networking (SDN).
5. **Desktop Virtualization**
   - Allows remote access to a desktop environment.
   - **Example**: Windows Virtual Desktop.

### 4.2 Hypervisors

A **hypervisor** is software that enables virtualization by managing VMs.

| Type | Description | Example |
| --- | --- | --- |
| **Type-1 (Bare Metal)** | Runs directly on hardware | VMware ESXi, Microsoft Hyper-V |
| **Type-2 (Hosted)** | Runs on top of an OS | VirtualBox, VMware Workstation |

---

# 5. Case Studies: AWS, GCP, and Microsoft Azure

## 5.1 Amazon Web Services (AWS)

AWS is the **largest cloud provider**, offering **IaaS, PaaS, and SaaS** solutions.

- **Compute Services**: EC2 (VMs), Lambda (serverless computing).
- **Storage**: S3 (object storage), EBS (block storage).
- **Database**: RDS (relational DB), DynamoDB (NoSQL).
- **Networking**: VPC, CloudFront (CDN).

**Use Case**: Netflix uses AWS for streaming content globally.

---

## 5.2 Google Cloud Platform (GCP)

GCP provides cloud infrastructure, AI, and analytics tools.

- **Compute**: Compute Engine (VMs), Kubernetes Engine.
- **Storage**: Cloud Storage, Persistent Disk.
- **AI & ML**: TensorFlow, Vertex AI.
- **Big Data**: BigQuery, Dataflow.

**Use Case**: Spotify uses GCP for machine learning and recommendations.

---

## 5.3 Microsoft Azure

Azure is widely used for enterprise solutions and integrates with Microsoft products.

- **Compute**: Azure Virtual Machines, Functions (serverless).
- **Storage**: Blob Storage, Azure Files.
- **AI & Analytics**: Azure Machine Learning, Power BI.
- **Hybrid Cloud**: Azure Stack, Hybrid Networking.

**Use Case**: BMW uses Azure for IoT and AI-based car analytics.

---

# 6. Summary Table

| Concept | Key Points | Examples |
|---|---|---|
| **Cloud Computing** | Internet-based computing, scalable & cost-effective | AWS, GCP, Azure |
| **Cloud Service Models** | SaaS, PaaS, IaaS | Google Drive (SaaS), AWS EC2 (IaaS) |

| Concept | Key Points | Examples |
|---|---|---|
| **Deployment Models** | Public, Private, Hybrid, Community | AWS (Public), Azure Stack (Hybrid) |
| **Virtualization** | Creates virtual resources | Hypervisors: VMware, KVM |
| **AWS** | Largest cloud provider | EC2, S3, Lambda |
| **GCP** | Strong AI & ML services | Compute Engine, BigQuery |
| **Azure** | Best for enterprise solutions | Azure VM, Blob Storage |