Question Bank

Q1 Explain desirable features of Global scheduling Algorithm

Q 2 Explain Load balancing Approach with its taxonomy

Q3 Explain process migration in distributed computing ?

Q4 Explain code migration in distributed systems ?

Q5 Explain data centric consistency with example

Q 6 Explain client centric consistency ?

Q 7 Explain Replication management in distributed systems ?

Q 8Explain Fault tolerance in context of distributed systems

Q9 What is DFS (Distributed file system and what are the features of DFS)?

Q 10 Discuss different file access models and Diffrentiate between Remote service and file caching ?

Q11 Explain Different File Caching Schemes and Modification Propagation in Distributed File Systems.

Q12 Explain AFS,HDFS,NFS and also differentiate between all the three

Q.1 What are the desirable features of a global scheduling algorithm?

Ans1: The feature of a global scheduling algorithm are as follows:—

1] No prior knowledge about processes required: The working of scheduling algorithm is based on information about characteristics and resource requirements of processes.
→ However these pose an extra burden on the users to provide this information while submitting their processes for execution. No such information is required for global scheduling algorithm.

2] Dynamic in nature: The decision regarding the assignment of a process should be dynamic which means it should be based on current load of the system and not on some static policy. The flexibility to migrate the process more than once should be the property of the system.

3] Decision making capability: If we think about the Heuristic methods, they require less computational efforts that results in less time requirement for the output. This will provide ~~this~~ near optimal result and has decision making ability.

4) Balancing system performance and scheduling overhead: Here we require algorithms that will provide us with near optimal system performance. It is desirable to collect minimum of global state information such as CPU load. Such information is crucial because a amount of global state information collected increases, the overhead increases. This will have an impact on the result of the cost of g and processing the extra information. So the performance of the syst can be improved by minimizing scheduling overhead.

Stability: Processor thrashing (due to fruitless migration) must be prevented. For example if nodes n1 and n2 are loaded with processes and it is observed that n3 is idle, they offload a portion of work to n3, without being aware of the offloading decision made by some of the other nodes. In case n3 becomes overloaded due to this, it may again start transfering its processes to other nodes.

Scalability: The scheduling algorithm should be scale as the number of nodes increases. An algorithm can be turned as poor scalability if it makes scheduling decisions by first inquiring the workload from all nodes and then selecting the most lightly loaded node. This concept will work fine only when we have few nodes in the system. This will happen because the inquirer receives a flood of replies simultaneously and time required to process the reply message for making a node selection is too long. As the number of nodes increases, the network traffic consumes network bandwidth quickly.

Fault Tolerance: In case one or more nodes of the system crash, a good scheduling algorithm should not be disabled by this and mechanism to handle this should be available. The algorithm should be capable of functioning properly when the nodes of it is observed that the nodes are partitioned into two or more groups due to link failures. In order to have a better tolerance capability, algorithms should decentralise the decision making capability, and must consider only available nodes in this decision making and better fault tolerance capability.

Fairness of service: If the global scheduling policy blindly attempts to balance the load of all nodes of the system they are not good in terms of fairness, as all the heavily loaded nodes will obtain the benefits while ~~poor~~ lightly loaded nodes will have a poorer response time. It is necessary to have a fair strategy that will improve the response time without affecting load distribution. Thus load balancing must be replaced with load sharing, the node will share some of its resources as long as its users are not affected.

Q 2 Explain Load balancing Approach with its taxonomy

Ans 2 : **Load Balancing in Distributed Systems**

Definition:
Load balancing is the technique of distributing workloads (such as processes or tasks) across multiple processors or nodes in a distributed system. The main goal is to ensure that no single node is overwhelmed while others are underutilized. By doing so, all processors can remain equally busy and complete tasks approximately at the same time.

Key Purpose:

- To **improve system performance** by minimizing task completion time.
- To **increase fault tolerance** and **system reliability**.
- To **enhance security** by filtering out malicious traffic (e.g., DDoS).
- To ensure **fair resource utilization** and **scalability**.

📊 Taxonomy of Load Balancing

The taxonomy classifies different types of load balancing strategies based on several dimensions:

---

1. Static vs. Dynamic

- **Static Load Balancing**:
    - Decisions are made using pre-known, average system behavior.
    - Does **not** consider the current state of nodes.
    - Simpler to implement but **less adaptive**.
- **Dynamic Load Balancing**:
    - Uses real-time data to make decisions.

    - Continuously monitors node status and reacts to changes.
    - More accurate but requires more **overhead**.

---

2. Deterministic vs. Probabilistic

- **Deterministic**:
    - Uses fixed logic based on system and process characteristics.
    - Can be more **precise**, but harder to optimize in complex environments.
- **Probabilistic**:
    - Uses randomness or statistical information.
    - Easier to implement, but **performance may vary**.

---

3. Centralized vs. Distributed

- **Centralized Load Balancing**:
    - One central node collects all information and makes decisions.
    - Can be efficient but suffers from **bottlenecks** and is a **single point of failure**.
- **Distributed Load Balancing**:
    - Each node makes its own decisions based on local or shared data.
    - Offers **higher fault tolerance** and **scalability**, but more complex.

---

4. Cooperative vs. Non-Cooperative

- **Cooperative**:

    - Nodes **share information** and work together to balance load.
    - More **accurate and stable**, but involves more communication overhead.
- **Non-Cooperative**:
    - Nodes act **independently** without coordinating with others.
    - Less communication, but may result in **inefficient decisions**.

Q2 Explain key design policies for load balancing

Key Design Policies for Load Balancing

To effectively balance load across multiple nodes in a distributed system, several **design policies** are considered. These policies determine **how, when, and where** tasks should be transferred for execution.

---

1. Load Estimation Policy

This policy defines **how the system measures the workload** of a node.

Common estimation methods:

- Total number of active processes.
- CPU utilization (e.g., % of busy time).
- Process service time (estimated via memoryless or statistical methods).
- Architecture and processing speed of the node.

Goal: Identify overloaded or underloaded nodes based on accurate workload metrics.

---

2. Process Transfer Policy

Decides **whether a process should be executed locally or migrated** to another node.

Types:

- **Single Threshold**: A fixed load level is used to decide migration.
- **Double Threshold** (High/Low policy):
    - Overloaded → sends out tasks
    - Underloaded → accepts tasks
    - Normal → neither sends nor accepts

Goal: Prevent instability and unnecessary migration by using thresholds.

---

3. Location Policy

Determines **which node should receive a migrated process**.

Methods:

- **Threshold Method**: Randomly probes nodes until a suitable one is found.
- **Shortest Method**: Polls a few nodes and chooses the least loaded.
- **Bidding Method**: Nodes send bids and the best offer is selected.
- **Pairing**: Randomly forms pairs between high-load and low-load nodes for direct migration.

Goal: Select optimal destination for load transfer.

---

4. State Information Exchange Policy

Controls **how and when nodes share their current load status**.

Types:

- **Periodic Broadcast**: Sends updates at regular intervals.
- **On State Change**: Broadcasts only when load region changes (e.g., normal → overloaded).

- **On Demand**: Node requests info only when it needs to offload or receive.
- **Polling**: Actively queries other nodes one by one.

Goal: Balance the trade-off between **decision accuracy** and **message overhead**.

---

5. Priority Assignment Policy

Determines **which processes get execution priority**—local or remote.

Types:

- **Selfish**: Local > Remote (less efficient).
- **Altruistic**: Remote > Local (more efficient).
- **Intermediate**: Priority depends on number of local vs. remote tasks.

Goal: Ensure fairness and reduce execution delays.

---

6. Migration Limiting Policy

Limits the **number of times a process can be migrated** to avoid excessive overhead.

Types:

- **Uncontrolled**: No limit on migrations.
- **Controlled**: Uses a migration counter to restrict how often a task is transferred.
- **Irrevocable Migration**: Only one migration allowed per task.

Goal: Reduce system instability caused by frequent migrations.

Q3 Explain process migration in distributed computing ?

Process Migration in Distributed Systems

Process migration is a key component of process management in distributed systems. It refers to the **relocation of a process from its current node to another node** with the aim of improving resource utilization and system performance.

The main goal of process management is to make the best possible use of existing resources by implementing mechanisms and policies that enable efficient sharing of processes among various processors. This is achieved through:

- **Process Allocation**: Assigning processes to appropriate nodes at the start.
- **Process Migration**: Moving processes to other nodes during or before execution.
- **Thread Facilities**: Enabling parallelism to maximize CPU usage.

Types of Process Migration

1. **Non-Preemptive Migration**:
    - The process is migrated **before it begins execution**.
    - This method is simpler and less resource-intensive.
2. **Preemptive Migration**:
    - The process is migrated **during execution**.
    - It is more complex and expensive since the execution environment (registers, memory, open files, etc.) must be transferred.

Desirable Features of Good Process Migration

- **Efficiency**: Low migration time, efficient object location, and minimal overhead for remote execution.
- **Robustness**: Migration should not be disrupted by failures in other nodes.
- **Communication Transparency**: Processes should be able to communicate regardless of their physical location.

Steps in Process Migration

1. Selection of the process to be migrated.
2. Identification of a suitable destination node.
3. Transfer of the process to the selected destination.

The **migration policy** is responsible for selecting the process and the destination, while the **migration mechanism** carries out the actual transfer.

Mechanisms for Address Space Transfer

1. **Total Freezing**:
    - The process is stopped entirely while its address space is copied.
    - Simple but can cause long delays.
2. **Pre-Transferring**:
    - The address space is transferred while the process continues running.
    - Reduces freezing time but may lead to redundant transfers.
3. **Transfer on Reference**:
    - The process starts executing on the new node while memory pages are fetched from the source node on demand.
    - Lightweight initially but risky if the source node fails.

Message Forwarding Mechanisms

1. **Resending Messages**:

- Messages not delivered are sent again by the sender.
- Simple but non-transparent.

2. **Ask Origin Site**:
   - The origin node forwards messages to the new location.
   - Fails if the origin site goes down.
3. **Link Traversal**:
   - A forwarding link chain is maintained.
   - Inefficient and prone to failure if any link breaks.
4. **Link Update**:
   - The system updates the new address across all relevant nodes.
   - Efficient and synchronized.
     (for diagrams refer ppt)

Q4 Explain code migration in distributed systems ?

Ans **Code Migration in Distributed Systems**

Code migration refers to the act of **moving code or entire program components** from one machine to another within a distributed system, typically for the purposes of **performance improvement**, **flexibility**, and **efficient resource utilization**.

Traditionally, code migration was done through **process migration**, where a running process was moved to another node. However, this is expensive and complex, and thus newer approaches focus on moving only code components or lightweight mobile programs.

🔍 Why Code Migration?

1. **Performance Optimization**:
   Moving code closer to data (e.g., migrating a data-heavy client task to the server) reduces network load and improves execution speed.
2. **Flexibility**:
   Allows dynamic reconfiguration of distributed applications. For example, clients can

download server-side components only when needed, enabling on-demand service composition.

3. **Parallelism Without Complexity**:
   Mobile agents can be used to perform tasks like web searching in parallel by cloning and dispatching copies across sites.

🧱 Models of Code Migration

Fuggetta et al. describe a process as consisting of three segments:

1. **Code Segment**: The program instructions.
2. **Resource Segment**: References to files, devices, or network connections.
3. **Execution Segment**: The current state (stack, data, program counter).

Depending on what is moved, we distinguish between:

- **Weak Mobility**: Only the **code** and possibly some data is moved. Execution restarts from the beginning (e.g., Java applets).
- **Strong Mobility**: Both code and **execution state** are moved. Execution resumes from the exact point it was paused (e.g., process migration, remote cloning).

🔁 Initiation of Code Migration

- **Sender-Initiated Migration**: The source machine decides to send code (e.g., uploading a search agent to a web server).
- **Receiver-Initiated Migration**: The destination machine requests and loads code (e.g., Java applets loaded by browsers).

Receiver-initiated is simpler and more common, especially in client-server systems where the client dynamically loads code.

💡 Execution Strategies

- Code can be executed within an existing process (e.g., applets in browser) or in a **separate process**, which offers better security and isolation.
- **Remote cloning** is another form, where the original continues to run while a copy is executed remotely.

📎 Process-to-Resource Bindings

When code migrates, its references to resources must be handled carefully. Fuggetta et al. define three types of bindings:

1. **By Identifier**: Must access a *specific* resource (e.g., a port or IP).
2. **By Value**: Needs the *content*, not the specific instance (e.g., standard libraries).
3. **By Type**: Requires any resource of a given type (e.g., any printer).

And three types of **resource-to-machine** bindings:

1. **Unattached**: Can easily be moved (e.g., data files).
2. **Fastened**: Movable but expensive (e.g., databases).
3. **Fixed**: Bound to a physical location (e.g., hardware devices).

Different combinations of these bindings determine how migration and resource handling are done.

🌐 Code Migration in Heterogeneous Systems

In heterogeneous environments (different OS or architectures), migration becomes more complex. Solutions include:

- **Portable intermediate code** (e.g., Java bytecode, scripting languages)

- **Process virtual machines** (e.g., JVM), which allow the same code to run on different platforms.

🖥️ Virtual Machine Migration

An advanced form of code migration is **virtual machine (VM) migration**, where an entire OS environment is moved. Key techniques include:

1. **Push Strategy**: Send memory pages first, then update modified ones.
2. **Stop-and-Copy**: Pause the VM, transfer everything, then resume.
3. **Pull-on-Demand**: Let the new VM load memory pages as needed.

Each has trade-offs between **downtime** and **performance**.

✅ Conclusion

Code migration enhances distributed systems by providing **dynamic execution**, **load balancing**, **flexibility**, and **performance gains**. However, it also introduces challenges related to **execution context**, **resource binding**, **security**, and **platform compatibility**. Careful design and use of virtual machines, portable code, and migration models are essential to implement it effectively.

Q5 Explain data centric consistency with example

Ans Data-Centric Consistency

🧠 What It Means:

Data-centric consistency refers to the rules or guarantees about how **read and write operations** on shared or replicated data behave **across multiple servers or locations** in a distributed system.

In simple terms, it answers the question:

“If data is stored in multiple places, how consistent are those copies when users read or update them?”

🔹 A. Strict Consistency

- **Definition:** The most rigid model.
- A read operation **always returns the latest written value**, regardless of which replica is accessed.
- **Real-world analogy:** Like everyone seeing the exact same Google Doc update at the same instant.
- **Why it’s rare:** It needs perfect clock synchronization and instant updates — very hard in real systems.

🔹 B. Sequential Consistency

- **Definition:** All users see operations (reads/writes) in the **same order**, but not necessarily the **real-time order**.
- **Analogy:** Think of a queue at a coffee shop — everyone is served one after the other, but some may have placed their order earlier.
- **Use Case:** Systems where **operation order matters**, like financial apps.

🔹 C. Causal Consistency

- **Definition:** Only operations that are **logically related** need to be seen in order.
- **Example:**
    - If user A posts a photo, and user B comments on it, then all users must see the photo **before** the comment.

    - But if someone else likes a different post at the same time, that order doesn't matter.

- **Use Case:** Social media timelines, collaborative editing tools.

🔹 D. Eventual Consistency

- **Definition:** All replicas will **become consistent over time**, but might **differ temporarily**.
- **Analogy:** Like telling friends a story — at first only a few know, but eventually everyone knows the same version.
- **Real-world Examples:**
    - Amazon shopping cart updates.
    - DNS records: changes take time to reach all servers.

🔹 E. Entry Consistency

- **Definition:** Data becomes consistent **only when you use a certain lock or access mechanism**.
- **Analogy:** Like checking out a library book — only one person can modify the content at a time, ensuring consistency when it's returned.
- **Use Case:** Shared memory systems or collaborative editing with locking.

📌 Summary Table

Model	Main Feature	Where Used

Strict Consistency	Always get the latest data instantly	Rare (theoretical)
Sequential Consistency	Everyone sees the same order of operations	Databases, banking systems
Causal Consistency	Related actions are seen in order	Social media, collaborative tools
Eventual Consistency	All replicas sync eventually	DNS, Amazon Dynamo, cloud storage
Entry Consistency	Consistency via locks or permissions	Shared memory apps, version control

Q 7 Explain client centric consistency ?

Ans **Client-Centric Consistency**

🟢 Introduction:

In modern distributed systems, users may access services from **different devices**, **locations**, or **replica servers**. When these systems use **replication** for high availability and performance, it's not guaranteed that all replicas are instantly updated.

This leads to an important question:

> *"How can we ensure that **each individual user** experiences a consistent view of data, even if the system is eventually consistent overall?"*

That's where **Client-Centric Consistency** comes in.

Rather than focusing on how the entire system synchronizes data (which is the domain of data-centric consistency), **client-centric models focus on what each user sees**, based on their previous interactions. The goal is to provide a smooth, predictable user experience, even if updates are still propagating behind the scenes.

🧩 Client-Centric Consistency Models

There are four main types:

🔹 1. Monotonic Reads

- **Definition:** Once a user sees a particular value, they will **never see an older (stale) version** of that data in future reads.
- **Example:**
  You check your flight status in a travel app — it shows “Boarding.” Later, when you refresh, it still shows “Boarding” or a newer status like “Departed,” but **never reverts to an older status like “Scheduled.”**
- **Why It Matters:** Prevents confusion and increases user trust in the system.

🔹 2. Monotonic Writes

- **Definition:** If a user sends multiple updates, the system ensures those **writes are applied in the order they were issued**.
- **Example:**
  A person updates their delivery address and then immediately updates their phone number. The system will process and reflect both updates in the right sequence.
- **Use Case:** Important in **profile management, form submissions**, and systems that rely on **user workflows.**

🔹 3. Read Your Writes

- **Definition:** After a user performs a write operation (like an update), any **future reads should reflect that change** immediately for the same user.
- **Example:**
  You post a tweet — it instantly appears on your timeline, even if other users might not see it yet due to propagation delays.

- **Why It's Crucial:** Gives users **feedback** that their actions were successful. Common in **social media, e-commerce, and online banking.**

🔹 4. Writes Follow Reads

- **Definition:** If a user **reads** some data and then makes a **write decision based on it**, the system guarantees that the **write is based on the same or newer version** of the data.

- **Example:**
  A warehouse manager checks current stock of a product (10 units) and decides to ship 2 units. The system ensures that his update doesn't overwrite any newer inventory changes.

- **Why It Helps:** Prevents **write conflicts** and **data loss**, especially in **collaborative or transaction-based systems**.

✅ Quick Summary Table

Consistency Model	Guarantee	Real-World Analogy
Monotonic Reads	Once you see new data, you never see older data again	Flight status check
Monotonic Writes	Your updates are applied in the order you sent them	Form submission steps
Read Your Writes	You always see your own updates immediately	Profile or post update

Writes Follow Reads	Writes are based on the latest info you've seen	Inventory control, ticket booking

Q Explain Replication management in distributed systems ?

Ans

Definition:

Replica Management is a critical process in distributed systems that involves creating, maintaining, and controlling multiple copies (replicas) of data across various nodes. This ensures that data remains available, reliable, and accessible, even in the presence of failures or increased load.

Objectives of Replica Management:

1. **High Availability:**
   Ensures data access even when some nodes fail or become unreachable.

2. **Fault Tolerance:**
   Prevents data loss by maintaining backup copies on different nodes.

3. **Load Balancing:**
   Distributes requests across replicas to avoid overloading any single server.

4. **Latency Reduction:**
   Improves response time by accessing the nearest available replica.

5. **Disaster Recovery:**
   Provides resilience against large-scale failures like data center outages.

Key Components of Replica Management:

1. **Replica Placement:**
   Strategic decision on where to place replicas, based on access patterns, geographical location, network conditions, and node reliability.

2. **Replica Consistency:**
   Ensures all replicas reflect accurate and updated data. Common models include:
   - *Strong Consistency:* All replicas are updated before confirming a transaction.
   - *Eventual Consistency:* Replicas will converge to the same state over time.
   - *Causal Consistency:* Maintains the order of related operations.
3. **Replica Synchronization:**
   Propagation of updates across replicas can be:
   - *Synchronous:* Updates all replicas before acknowledgment.
   - *Asynchronous:* Updates primary replica first, then propagates changes.
   - *Quorum-based:* Uses a voting system to ensure consistency.
4. **Replica Selection:**
   Chooses the best replica to serve a request based on latency, server load, or proximity.

Replication Models:

Model	Description	Use Case
Full Replication	All nodes have a full copy of data	Best for read-heavy systems
Partial Replication	Some nodes store subsets of data	Useful for large datasets
Dynamic Replication	Replicas are created/removed based on demand	Adaptive systems

---

Types of Replication:

1. **Primary-Backup (Master-Slave):**
   One primary replica handles all write operations; others act as backups.
   Example: Traditional relational databases.
2. **Multi-Master (Peer-to-Peer):**
   All replicas can handle reads and writes. Conflict resolution is essential.

Example: CouchDB, Cassandra.

Challenges in Replica Management:

1. **CAP Theorem Limitations:**
   A system can only guarantee two of the three: *Consistency, Availability, Partition Tolerance.*

2. **Consistency Overhead:**
   Synchronizing replicas can increase latency and complexity.

3. **Data Staleness:**
   In asynchronous replication, replicas may serve outdated data.

4. **High Storage and Bandwidth Use:**
   More replicas mean more storage and network resources.

5. **Conflict Resolution:**
   Multi-master systems must handle conflicting updates using mechanisms like timestamps or vector clocks.

Real-World Examples:

System	Replica Management Usage
Google File System (GFS)	Stores multiple copies of each chunk for fault tolerance
Amazon S3 / DynamoDB	Uses eventual consistency and global replication
Content Delivery Networks (CDNs)	Replicate content across locations for low-latency access
Hadoop HDFS	Maintains 3 replicas per block for high availability
Apache Cassandra	Offers configurable replication factor and consistency levels

Q Explain Fault tolerance in context of distributed systems

Fault Tolerance in Distributed Systems

Introduction: Fault tolerance refers to a system's ability to continue functioning correctly even in the presence of faults or failures. In distributed systems, where components are spread across different physical machines and connected by networks, faults are more frequent due to the increased complexity and numerous points of failure. Ensuring that a system can continue delivering its services despite these issues is critical.

Core Concepts of Fault Tolerance

A fault-tolerant distributed system is essentially a *dependable system*. Dependability is achieved through four interrelated attributes:

1. **Availability** – The probability that the system is operational at any given moment.
2. **Reliability** – The ability of the system to function correctly over a time interval without interruptions.
3. **Safety** – Even in the event of failures, the system should not cause any catastrophic results.
4. **Maintainability** – The ease with which the system can be repaired and returned to service.

> **Example:** A cloud storage service should always be available (high availability), should not lose data (safety and reliability), and must allow quick recovery in case of hardware failures (maintainability).

Understanding Faults, Errors, and Failures

- A **fault** is the root cause—like a hardware malfunction or software bug.
- An **error** is a corrupted state that might lead to failure.
- A **failure** is when the system does not perform its expected function.

> **Example:** If a hard disk crashes (fault), the system might lose access to some files (error), leading to the failure of the file-sharing service.

Types of Faults

1. **Transient Faults** – Temporary and disappear on their own (e.g., brief network interference).
2. **Intermittent Faults** – Come and go repeatedly (e.g., a loose cable causing occasional disconnections).
3. **Permanent Faults** – Persist until the component is fixed or replaced (e.g., a burnt-out processor).

Failure Models in Distributed Systems

1. **Crash Failures** – The server stops and does not respond.
2. **Omission Failures** – Server fails to receive or send messages.
3. **Timing Failures** – Responses are sent too early or too late.
4. **Response Failures** – Incorrect results are sent back.
    - *Value failure:* Wrong output.
    - *State transition failure:* Unexpected behavior.
5. **Arbitrary/Byzantine Failures** – The most severe; system behaves erratically or maliciously.

Example: In a distributed banking application:

- If a node crashes and stops processing transactions, it's a **crash failure**.
- If it delays or sends wrong balances, it could be a **timing** or **value failure**.
- If it acts maliciously (e.g., giving different outputs to different clients), that's a **Byzantine failure**.

Techniques to Achieve Fault Tolerance

1. **Redundancy** – The core approach to fault tolerance. It comes in several forms:
    - **Information Redundancy:** Add extra bits for error detection/correction.

      Example: Hamming codes for error correction in data transmission.

    - **Time Redundancy:** Retry operations or repeat processes.

      Example: Re-executing a failed transaction.

    - **Physical Redundancy:** Duplicate hardware or software components.

      Example: Having multiple database replicas or backup servers.

2. **Triple Modular Redundancy (TMR):**
    - Each component (e.g., processing unit) is replicated three times.
    - A voting system decides the correct output based on majority.
3. *Example: Aircraft control systems using TMR to continue functioning even if one unit fails.*

Practical Applications of Fault Tolerance

- **Cloud Platforms:** Use replication and failover strategies to ensure services remain available.
- **Distributed Databases:** Maintain consistency and availability via consensus protocols (e.g., Paxos, Raft).

- **Real-Time Systems:** Design with strict timing constraints and fail-safe mechanisms (e.g., automotive systems).

Conclusion

Fault tolerance is not about eliminating all faults—it's about building systems that *gracefully handle them*. By combining intelligent fault detection, redundancy, and recovery strategies, distributed systems can continue delivering reliable service despite the inevitability of hardware malfunctions, software bugs, or unpredictable network conditions.

Q What is DFS (Distributed file system and what are the features of DFS)?

Ans A **Distributed File System (DFS)** is a file system that allows files to be stored across multiple physical machines but accessed and managed as if they reside on a single system. In a DFS, users and applications can access and manipulate files stored on remote servers using standard file operations, without needing to know the files' physical locations.

It follows a **client-server architecture**, where servers manage the actual storage and clients request file services. The primary goal is to enable **efficient sharing of data and resources** across a network in a **transparent and reliable** manner.

A well-designed DFS ensures that file access, naming, and management remain seamless to the end-user, even though the files might be spread across various locations in a distributed environment.

Features of Distributed File System (DFS):

1. Transparency

DFS provides several types of transparency to ensure a seamless experience:

- **Access Transparency:** Files are accessed using the same operations regardless of where they are located in the system.

- **Location Transparency:** The file's physical storage location is hidden from the user; a consistent naming convention is maintained.

- **Concurrency Transparency:** Multiple users can access or update the same file concurrently without interference or data loss.
- **Replication Transparency:** Users are unaware of whether a file is replicated or how many copies exist.
- **Failure Transparency:** The system continues functioning even if one or more components fail.

---

2. Scalability

DFS can scale both in terms of storage and number of users. As the need for more storage or users increases, new servers and storage devices can be added easily without disrupting the existing system.

---

3. Fault Tolerance and Reliability

By replicating data and distributing it across multiple servers, DFS ensures that the failure of a single node or disk does not result in data loss. If one server goes down, a replicated copy from another server can serve the data.

---

4. Performance

DFS improves performance through:

- **Caching:** Frequently accessed data is temporarily stored closer to the user.
- **Load Balancing:** Requests are distributed across multiple servers to avoid overloading any single node.
- **Parallel Access:** Multiple files can be accessed in parallel, improving throughput.

---

5. Security

DFS includes mechanisms for:

- **Authentication:** Verifying the identity of users accessing the system.
- **Authorization:** Controlling user permissions and access levels.
- **Encryption:** Protecting data during transmission across networks.

---

6. Heterogeneity

DFS can operate across different types of hardware, operating systems, and network technologies. This makes it flexible and adaptable to diverse computing environments.

---

7. Consistency

Despite data being replicated across various nodes, DFS ensures that users see a consistent view of the data. If a file is updated on one node, the changes are reflected across all replicas.

---

8. Replication and Caching

- **Replication** improves availability and fault tolerance by storing copies of files on multiple servers.
- **Caching** enhances speed by storing frequently accessed files temporarily on client machines or nearby nodes.

Q2 Discuss different file access models and Diffrentiate between Remote service and file caching ?

Ans In Distributed File Systems (DFS), file access models define how clients access files stored on remote servers. Two commonly used models are the **Remote Service Model** and the **Caching Model**. Both provide access to distributed data but differ significantly in how data is handled, stored, and synchronized.

---

1. Remote Service Model

In the **Remote Service Model**, every file operation performed by the client (such as read, write, open, close) is sent as a request to the server. The server processes the request and sends the result back to the client.

Key Features:

- The file remains stored and managed entirely on the server.
- No data is permanently stored or cached at the client side.
- Each operation involves **network communication**.

Advantages:

- Simple design and easy to implement.
- Centralized control makes it easier to maintain **consistency and security**.
- Fault recovery is easier since the server retains all data and state.

Disadvantages:

- **High latency** due to frequent network requests.
- Can create **bottlenecks** at the server.
- Poor performance when multiple small operations are performed on a file.

---

2. Caching Model

In the **Caching Model**, when a client accesses a file, a **copy (or part of the file)** is cached locally. Subsequent operations (read/write) are performed **locally** on the cached copy, and updates are sent to the server **later**, either on file close or at scheduled intervals.

Key Features:

- Supports **file-level or block-level caching**.
- Reduces network traffic by avoiding repeated server communication.

- Requires mechanisms to handle **cache coherence and consistency**.

Advantages:

- **High performance** due to local processing.
- Reduces server load and network usage.
- Suitable for read-heavy or less frequently updated files.

Disadvantages:

- **Consistency issues** may arise if multiple clients access the same file simultaneously.
- Complex synchronization and cache invalidation logic needed.
- Potential risk of **stale data** if cache is not updated promptly.

Comparison: Remote Service Model vs. Caching Model

Aspect	Remote Service Model	Caching Model
Data Location	Data resides on the server; accessed remotely	Data is copied and accessed locally at the client
Network Usage	High, for every operation	Low, as repeated operations are performed locally
Performance	Slower due to network delays	Faster due to local processing
Consistency	Easier to maintain, centralized control	More difficult; needs coherence mechanisms

Fault Tolerance	Easier recovery (server has full state)	Complex recovery (client has partial or cached data)
Implementation	Simple	More complex
Example System	Early versions of NFS	AFS (Andrew File System)

Q3 Q. Explain Different File Caching Schemes and Modification Propagation in Distributed File Systems.

Introduction:

In a **Distributed File System (DFS)**, file access is slower compared to local access due to network delays. To improve performance and reduce latency, **file caching** is employed — i.e., storing file data closer to the user after it is fetched from the server. Proper caching also reduces server load and network traffic.

However, to maintain correctness, **modifications made to cached data** must be properly propagated back to the server.

Where Can the Cache Be Present? *(As per notes)*

Cache can be maintained at various places in the system:

1. **Client's Main Memory**
    - Fastest access, but volatile (lost on reboot or crash).
2. **Client's Disk**
    - Slower than main memory, but persistent across sessions.
3. **Server's Main Memory or Disk**
    - Helps serve repeated requests from multiple clients more efficiently.

- Note : Refer to diagrams on ppt .Diagrams carry marks

File Caching Schemes:

1. Client Caching

- File is cached at the client side (in memory or disk) after being fetched.
- All read and write operations are done on this local copy.
- The updated file is later propagated back to the server.

Pros:

- Reduces repeated server access.
- Minimizes network traffic.

Cons:

- Risk of **stale data** if multiple clients cache the same file.

---

2. Server Caching

- Server maintains a cache (usually in main memory) of frequently accessed files or blocks.
- Repeated client requests are served from this cache.

Pros:

- Reduces disk access on the server.
- Better for **read-heavy workloads**.

Cons:

- Does not reduce client-server communication.

---

3. Proxy or Intermediate Caching

- Implemented at a **proxy server** between clients and the file server.
- Useful in networks where **many clients access the same files**.

Pros:

- Local clients benefit from faster access.
- Reduces load on central file server.

Cons:

- Cache consistency across proxies must be maintained.

Modification Propagation:

After a file is modified in a cache, changes must be propagated back to the server. Your notes mention two main strategies:

---

1. Write-Through:

- **Every write operation is immediately sent** to the server.
- Keeps the server copy consistent at all times.

Drawback: High communication overhead and slower performance.

---

2. Delayed Write (Write-Back):

- Writes are done **locally**, and changes are sent to the server **later** (e.g., on file close).
- Reduces network load and improves performance.

Drawback: May result in **data inconsistency** if the client crashes before writing back.

Summary Table:

Aspect	Client Caching	Server Caching	Proxy Caching
Cache Location	Client memory/disk	Server memory/disk	Proxy/gateway server
Access Speed	Fast (local)	Fast for repeated requests	Fast for local group clients
Consistency Risk	High if not propagated	Low	Medium
Best For	Read-write by single client	Frequent shared access	Group of clients sharing files

Q4 Explain AFS,HDFS,NFS and also differentiate between all the three

Network File System (NFS):

- NFS is a **distributed file system protocol** that allows clients to access files over a network as if they were on the local disk.
- It follows an **open standard**, making it widely implementable.

- A **centralized server** stores all data; clients mount the shared file system using a mount command.

- **Limitation**: If the main system fails, data access is lost, and there is no built-in redundancy.

Example: In a university lab, all students access their files from a centralized NFS server. If that server crashes, no student can access their data.

---

2. Andrew File System (AFS):

- AFS was developed at **Carnegie Mellon University** to improve **scalability** in distributed systems.

- It introduced **client-side caching** — entire files are cached on the client's local disk to reduce repeated access to the server.

- AFS uses a **complex directory traversal** mechanism, which increases server overhead.

- It also suffers from **high traffic** due to constant validation messages checking file freshness.

Example: A software development team using AFS might cache source code files on each developer’s machine, reducing repeated access to a shared server.

---

3. Hadoop Distributed File System (HDFS):

- HDFS is designed to **store and process massive datasets** on **commodity hardware**.

- Data is distributed across many nodes with **default replication (usually 3)**, making it **fault-tolerant**.

- It’s a core part of the Apache Hadoop ecosystem.

- Ideal for **Big Data applications** and supports high-throughput data access.

Example: In a telecom company, call logs for millions of users are stored and processed using HDFS clusters for analytics and reporting.

Feature	HDFS (Hadoop Distributed File System)	NFS (Network File System)
Purpose	Designed for big data storage and processing in a distributed environment	Enables remote access to files over a network
Architecture	Distributed, fault-tolerant file system used in Hadoop clusters	Centralized or distributed file system used for file sharing across networked systems
Data Processing	Optimized for batch processing of large files	Suitable for general-purpose file storage and access
Scalability	Highly scalable, designed to handle petabytes of data	Scalable but not optimized for handling large-scale data analytics
Data Consistency	Writes are append-only, ensuring high throughput but not real-time updates	Supports real-time read/write operations with strong consistency
Fault Tolerance	Provides built-in replication (default is 3 copies of each file) for fault tolerance	Relies on RAID or other backup solutions for fault tolerance
Performance	Optimized for sequential read/write access	Optimized for low-latency, random access to files
Use Case	Big data analytics, batch processing (e.g., Hadoop, Spark)	File sharing in enterprises, remote access to files

Feature	AFS (Andrew File System)	NFS (Network File System)	HDFS (Hadoop Distributed File System)
Developed By	Carnegie Mellon University, later IBM	Sun Microsystems	Apache Software Foundation
Use Case	Enterprise-wide file sharing	General file sharing in UNIX/Linux	Big data storage and processing
Architecture	Client-server with local caching	Client-server with network mounts	Master-slave (NameNode, DataNodes)
Scalability	High (caching reduces load)	Limited	Very high
Performance	Good due to caching	Lower (depends on network speed)	High for batch processing, not real-time
Fault Tolerance	Limited (depends on volume replication)	Low (no built-in replication)	High (automatic replication)
Security	Kerberos authentication, ACLs	Basic (Kerberos optional)	Limited security, focus on replication
Data Access	Local caching improves access speed	Remote access over network mounts	Optimized for sequential read/write
Best For	Universities, enterprises	UNIX/Linux-based network file sharing	Big data analytics and storage