Distributed-Computing-Module-5-Important-Topics-PYQs

    - 6. Concurrent File Updates
    - 7. File Replication
    - 8. Hardware and Operating System Heterogeneity
    - 9. Fault Tolerance
    - 10. Consistency
    - 11. Security
    - 12. Efficiency
- 5. Differentiate between whole file serving and whole file caching in Andrew file System.
- 6. Define flat file service and directory service components.
    - Flat File Service
    - Directory Service
- 7. What are the advantages of Google File System.
- 8. Explain consensus algorithm for crash failures under synchronous systems.
    - What is a Consensus Algorithm?
    - Understanding Synchronous Systems
    - Consensus in the Presence of Crash Failures
        - Steps to Reach Consensus in Crash Failures:
- 9. Explain Andrew File System in detail
    - Key Features of AFS
        - Whole-File Caching for Faster Access
        - Whole-File Serving
    - How AFS Works (Step-by-Step)
    - AFS Components
        - Vice (Server-Side)
        - Venus (Client-Side)
    - Cache Consistency in AFS
- 10. Explain in detail Network File System Architecture
    - Key Components of NFS Architecture
    - 1. Client System
    - 2. Server System
    - 3. NFS Client Module
    - 4. NFS Server Module

    - Important Concepts in NFS
        - 1. Virtual File System (VFS)
        - 2. File Handle
        - 3. Access Transparency
        - 4. Mounting
- 11. Which are the assumptions made in Consensus and Agreement Algorithm
    - 1. Failure Model
    - 2. System Type: Synchronous vs Asynchronous
    - 3. Network Connectivity
    - 4. Sender Identification
    - 5. Reliable Channels
    - 6. Message Authentication
    - 7. Agreement Variable Type
- 12. Explain about the file service architecture
    - Three Main Parts of File Service Architecture
        - 1. Flat File Service (Manages File Contents)
        - 2. Directory Service (Manages File Names)
        - 3. Client Module (Acts as a Bridge)
    - Operations of File Service (What It Can Do)
        - Basic File Operations
        - Creating and Deleting Files
        - File Attributes (Metadata)
    - How It Works (Step-by-Step)
- 13. Write in detail about distributed file system characteristics.
    - Characteristics of Distributed File Systems
        - 1. Organization & Storage
        - 2. File Location & Access
        - 3. Data and Attributes
        - 4. Directories and Naming
        - 5. Metadata
- 14. Differentiate between AFS and NFS

1. Define Byzantine agreement problem.

he **Byzantine Agreement Problem** is a fundamental issue in **distributed systems**. It deals with how a group of processes (or computers) can **reach a common decision**, even if some of them behave incorrectly or maliciously (called **Byzantine faults**).

One process is called the **source process**, and it starts with an initial value. The goal is for all **non-faulty (honest)** processes to **agree** on a value, following three important conditions:

1. **Agreement**
    1. All non-faulty processes must decide on the **same value**, no matter what faulty processes do.
2. **Validity**
    1. If the source process is **not faulty**, then all non-faulty processes must decide on the **same value as the source's initial value**.
3. **Termination**
    1. Every non-faulty process must eventually **decide** on a value. No process should keep waiting forever.

2. Write the features of SUN Network File System.

Short story to learn

We want to **distribute (Distributed File Access)** some items to a remote location. First, we check the **architecture (Architecture Support)** of the destination to understand how many items need to be delivered. Then, we **virtually(Virtual File System)** place the order.

All the transaction details are carefully recorded in **files and handled (File Handles and Inodes)** by an accountant. Once the items reach their destination, we must **authenticate(Access Control and Authentication)** before handing them over to the **client**.

Finally, we **mount(Naming and Mounting)** each item into its designated **cache (Client and Server Caching)**, making them ready for both the **client** and **server** to access and use efficiently.

1. Distributed File Access

- NFS allows users to access **remote files** as if they were **local files**.
- It follows a **client-server model** using **Remote Procedure Call (RPC)** for communication.

2. Architecture Support

- NFS works on top of the **UNIX kernel** and supports the **Virtual File System (VFS)** layer.
- VFS helps to manage both **local** and **remote** files without any difference from the user's point of view.

3. Virtual File System (VFS)

- Provides **access transparency**: users don't need to know if a file is local or remote.
- Keeps track of all active file systems and routes requests accordingly.
- Uses **file handles** to uniquely identify files.

4. File Handles and Inodes

- A **file handle** includes:
  - File system ID
  - i-node number (identifies a file in the system)
  - i-node generation number (used when inode numbers are reused)

5. Client Integration

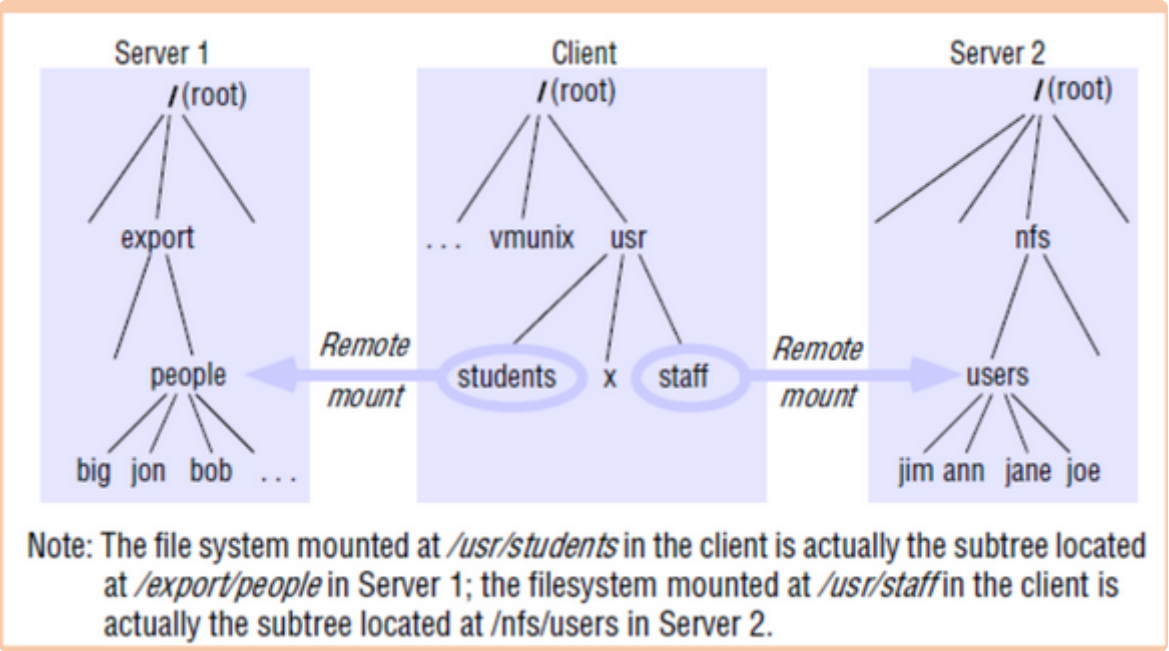
- The NFS client works with VFS to:
  - Access local files via the local file system.
  - Send requests to the NFS server for remote files.
- Caches file data locally to improve performance.

6. Access Control and Authentication

- NFS is **stateless**: the server does not keep track of open files.
- Each access is **individually checked** for user permissions.
- Supports **Kerberos** for strong authentication and security.

7. Naming and Mounting

- Files are accessed using **pathnames** (e.g., `host:/folder/file.txt`).

- **Mount service** lets you link remote directories to your local system.
  - 

    Note: The file system mounted at */usr/students* in the client is actually the subtree located at */export/people* in Server 1; the filesystem mounted at */usr/staff* in the client is actually the subtree located at /nfs/users in Server 2.
    - Types of mount:
      - **Soft mount**: retries for a limited time.
      - **Hard mount**: retries indefinitely until success.
      - **Auto mount**: mounted when accessed.

8. Client Caching

- Client saves file data to **reduce network traffic**.
- Can lead to **inconsistent versions** across clients.
- Clients use **timestamps** to check if their cached data is still valid:
  - `Tc` : time cache was last validated
  - `Tm` : time file was last modified on the server

9. Server Caching

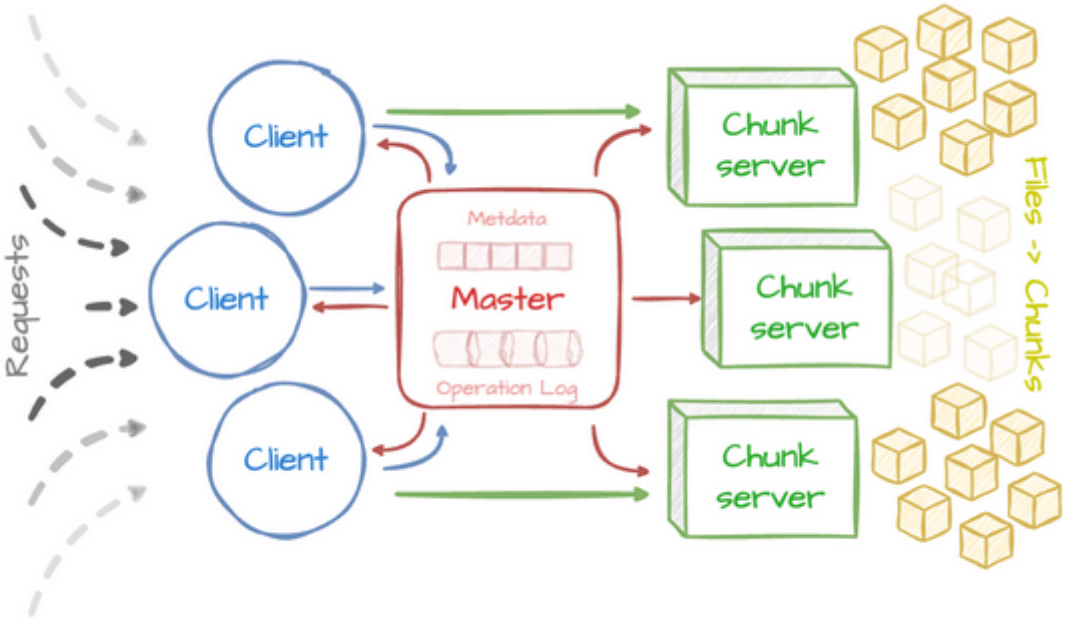
- Server caches recently accessed data to reduce disk usage.
- Write operations need **extra safety checks** to ensure data is **not lost** if the server crashes.

---

3. Explain the components of Google File System

The **Google File System (GFS)** is a **distributed file system** developed by Google to handle **large-scale data processing** across thousands of machines. It is designed to store **huge files** efficiently and support applications like **Google Search, YouTube, and Google Drive**.

GFS follows a **Master-Slave Architecture** with **three main components**:



Master Server

- Acts as the **brain** of GFS.
- Manages **metadata** (file locations, chunk details).
- Keeps track of which **Chunk Server** has which file pieces.
- Does **not store actual file data**.

Chunk Servers

- Store **large file chunks (typically 64MB each)**.
- Multiple copies (replicas) of each chunk are stored for **fault tolerance**.
- Handles **read and write requests** from clients.

Clients

- Users or applications that **request files** from GFS.
- First, they ask the **Master Server** for chunk locations.
- Then, they directly retrieve the file chunks from **Chunk Servers**.

How GFS Reads a File

- **Example:** A user wants to **read a file** stored in GFS.

1. **Client asks the Master Server** for file chunk locations.
2. **Master Server replies** with the list of Chunk Servers that store the chunks.
3. **Client directly fetches the file** from the Chunk Servers.

How GFS Writes a File

- **Example:** A user wants to **save a file** in GFS.

1. **Client sends a write request** to the Master Server.
2. The Master tells the client **which Chunk Server to write to**.
3. The file is **broken into chunks (64MB each)** and saved on multiple Chunk Servers.
4. **Replication happens automatically** to ensure data is not lost.

4. List distributed file system requirements

A **distributed file system (DFS)** allows users to access files from multiple computers as if they were stored locally. To ensure efficiency and usability, the system must meet certain **requirements**.

Short story to remember

We open a system and **access** files at a specific **location**. Then, we pick up our **mobile** to check its **performance**. Next, we **scale** this setup and test multiple mobiles **concurrently**. After that, we return to the system to **replicate** some files, ensuring they work smoothly across different **hardware** and **operating systems**. Everything continues to function **consistently**, with strong **fault tolerance**. The system also maintains high **security** and excellent **efficiency**.

1. Access Transparency

- Users and programs should not need to know whether a file is stored locally or remotely.
- The same file operations should work for both.

- **Example:** Accessing a file on Google Drive should feel the same as opening a file stored on a personal computer.

2. Location Transparency

- Files can be moved between servers, but their path remains unchanged.
- Users do not need to know where the file is physically stored.
- **Example:** A video on a streaming platform may move to different data centers, but users can still access it using the same link.

3. Mobility Transparency

- Files can be moved without requiring changes in client applications or system settings.
- Ensures that files remain accessible even when they are relocated.
- **Example:** A company might move employee files from one server to another without employees noticing any change.

4. Performance Transparency

- The system should maintain stable performance even when the load on servers varies.
- **Example:** A cloud storage service should provide smooth access to files even when many users are active.

5. Scaling Transparency

- The system should be able to expand and handle an increasing number of users and data without major changes.
- **Example:** A cloud-based file service should function efficiently whether it serves ten users or a million users

6. Concurrent File Updates

- Multiple users should be able to update files simultaneously without conflicts.
- **Example:** In a shared document, two users editing at the same time should not overwrite each other’s changes.

7. File Replication

- The system should maintain multiple copies of files across different locations.
- Helps in load balancing and fault tolerance.
- **Example:** A distributed file system storing data on multiple servers ensures availability even if one server fails.

8. Hardware and Operating System Heterogeneity

- The system should work across different operating systems and hardware.
- **Example:** A file service should be accessible from Windows, Linux, and macOS without compatibility issues.

9. Fault Tolerance

- The system should continue functioning even if some servers or clients fail.
- **Example:** If one storage server crashes, another should take over automatically to prevent data loss.

10. Consistency

- When files are updated, all copies should reflect the latest changes.
- There might be delays in propagating updates across different sites.
- **Example:** When an email is deleted from one device, it should also disappear from all other devices.

11. Security

- The system should protect data using authentication, access control, and encryption.
- **Example:** Only authorized users should be able to access confidential company files, and data should be encrypted to prevent unauthorized access.

12. Efficiency

- The system should provide high performance comparable to traditional file systems.
- **Example:** Opening and saving files in a distributed system should be as fast as working with local files.

5. Differentiate between whole file serving and whole file caching in Andrew file System.

Feature	Whole File Serving	Whole File Caching
What it means	The **entire file** is sent to the client in one go.	The **entire file** is **stored locally** after being received.
Purpose	To **reduce network traffic** by avoiding multiple small transfers.	To **improve speed and offline access** by keeping a local copy.
When it happens	**When a user requests a file** from the server.	**After the file is received**, it is cached for future use.
Effect on Network	Reduces load during file transfers.	Reduces need for repeated server requests.
Offline Use	Not supported directly.	**Cached file can be used even offline**.
Persistence	File must be re-requested each time.	Cached file remains even after restart.

6. Define flat file service and directory service components.

Flat File Service

- Responsible for handling the **actual contents of files**.
- It performs operations like **create, read, write, and delete**.
- Files are identified using **Unique File Identifiers (UFIDs)** instead of file names.
- Think of it as the part that works with the **real data inside the files**.

Directory Service

- Manages the **file names and directory structure**.
- It maps each **file name to its UFID**, helping the system **locate the actual file**.
- This allows users to **search and access files by name**, even though files are stored using IDs internally.

7. What are the advantages of Google File System.

Trick to learn

“GFS is SAFE and BIG”

- **S**calable
- **A**vailable (fault tolerant)
- **F**ast (direct access)
- **E**fficient (metadata split from data)
- and handles **BIG files**

1. **Designed for Huge Files**
   - GFS can handle files that are **gigabytes or even terabytes** in size efficiently.
2. **High Fault Tolerance**
   - Each file chunk is stored in **multiple replicas** (usually 3) across different servers.
   - If one server fails, data is still safe on others.
3. **Efficient Data Access**
   - Clients **communicate directly** with chunk servers (after getting metadata from the master), reducing delays.
4. **Scalability**
   - GFS works smoothly even across **thousands of machines**.
   - New machines can be added without affecting the system.
5. **Master-Slave Architecture**
   - The **Master Server handles only metadata**, not file content.
   - This keeps the master **lightweight and efficient**.
6. **Automatic Recovery**
   - If a chunk server crashes, the system **automatically creates new replicas** on other servers.
7. **Optimized for Large Data Processing**
   - Supports high-throughput operations, making it ideal for **big data applications** like Google Search, YouTube, etc.
8. **Efficient Write and Append Operations**

- GFS is optimized for **frequent appends** and writes, which is common in logs and data processing systems.

9. **Load Balancing**
    - Distributes files and chunks smartly to avoid overloading any single server.

8. Explain consensus algorithm for crash failures under synchronous systems.

What is a Consensus Algorithm?

A **consensus algorithm** helps multiple processes (computers) **agree on a single value** in a distributed system. This is important when some processes **fail or crash** but the system must continue working.

Understanding Synchronous Systems

A **synchronous system** means:
✔ **Time is predictable** – All messages are delivered within a known time.
✔ **Processing is predictable** – Each process takes a known time to compute.
✔ **Failures can be detected** – If a process doesn't respond in time, we assume it has **crashed**.

🔹 **Example:** Imagine a group of friends voting on where to eat. Everyone must answer **within 10 seconds**, or they are considered unavailable.

Consensus in the Presence of Crash Failures

In a **synchronous system**, crash failures happen when **some processes stop working** but don't send incorrect data. The goal of the consensus algorithm is to let the **remaining processes agree** on a decision, even if some fail.

Steps to Reach Consensus in Crash Failures:

1. **Each process proposes a value** (e.g., "Let's eat pizza" or "Let's eat burgers").
2. **Processes exchange values with each other** within a fixed time.
3. **If a process crashes, it stops responding**, but the remaining processes continue.
4. **Majority rule:** If more than half of the processes agree on a value, that value is chosen.
5. **Final Decision:** All non-crashed processes adopt the agreed-upon value.

9. Explain Andrew File System in detail

AFS is a **distributed file system** that allows users to access files from multiple computers **as if they were stored locally**. It is designed to **handle large numbers of users** efficiently by using **caching** to speed up file access.

Key Features of AFS

Whole-File Caching for Faster Access

- When a user **opens a file**, the **entire file** is downloaded to their computer.
- This file is then **cached (stored temporarily)** so future access is **faster**.
- Even if the computer restarts, cached files **remain available**.

Whole-File Serving

- AFS sends **the full file** instead of small parts at a time.
- This reduces **network traffic** and makes file access **more efficient**.

How AFS Works (Step-by-Step)

🔹 **Scenario: A User Opens a File**

1. The **user requests a file** (e.g., "notes.txt").
2. The **AFS server (Vice)** finds the file and **sends a copy** to the user's computer.
3. The **file is stored (cached) locally** on the user's computer.
4. The user can **read, write, or edit the file** from the local copy.
5. When the user **closes the file**, any changes are **sent back to the server**.
6. The **AFS server updates the main file** so others can see the changes.

AFS Components

Vice (Server-Side)

- Runs on **AFS servers** and **stores all shared files**.
- Sends files to users when requested.
- Ensures **file consistency** across multiple users.

Venus (Client-Side)

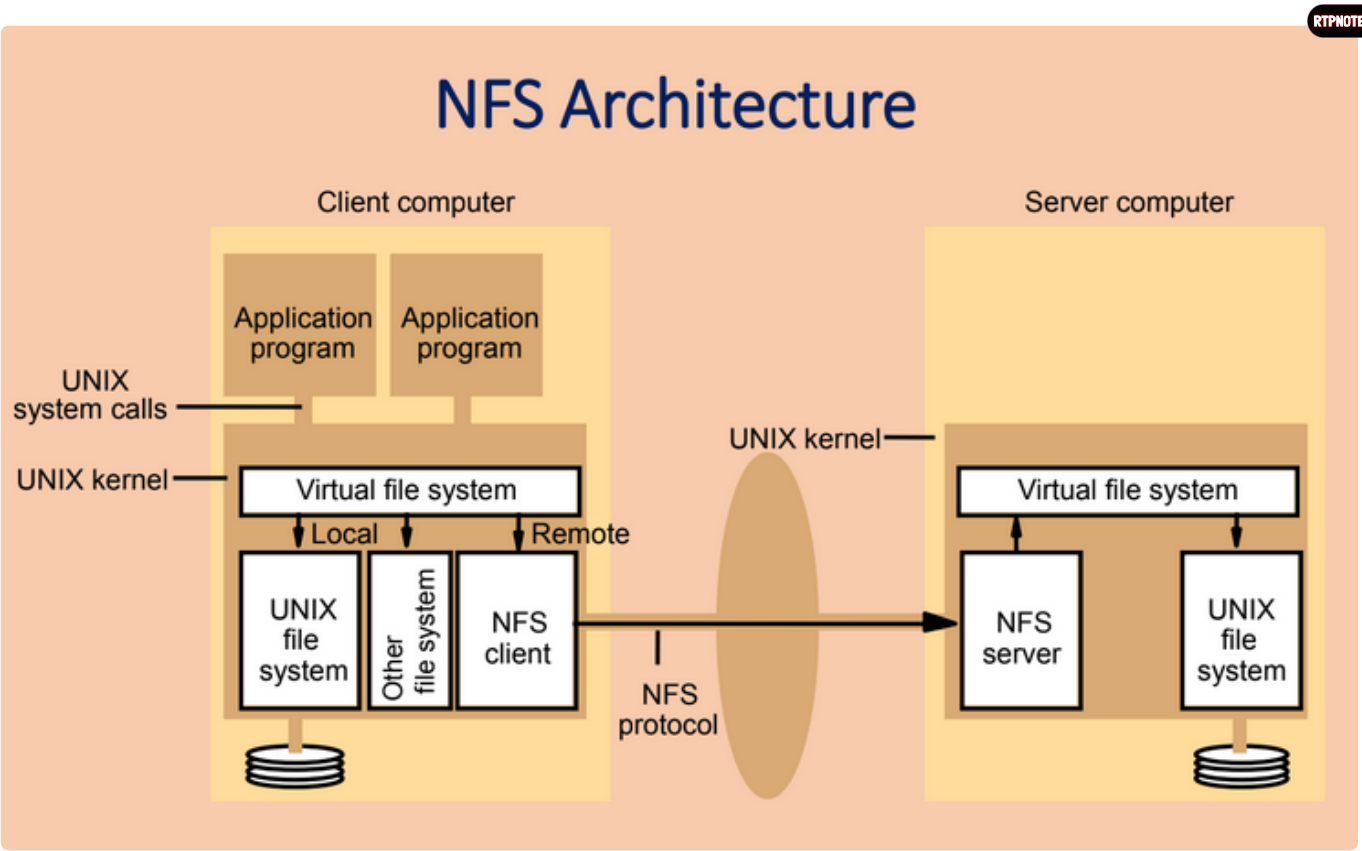
- Runs on **each user's computer**.
- Manages **caching of files** so they can be accessed quickly.
- Communicates with Vice to **fetch or update files**.

Cache Consistency in AFS

- AFS ensures that **all users see the latest version of a file** using a system called **Callback Promises**:
- ✔ **When a file is cached**, the AFS server **promises** to notify the user if the file is modified by someone else.
- ✔ If another user **modifies the file**, the server sends a **callback message** to all users with the old version.
- ✔ If a file’s callback is **canceled**, Venus downloads a fresh copy.

10. Explain in detail Network File System Architecture

The **Network File System (NFS)** is a **distributed file system** protocol developed by **Sun Microsystems**. It allows users to access files **stored on remote servers** just like they access files on their local machine.



Key Components of NFS Architecture

The NFS architecture mainly consists of the following components:

1. Client System

This is the user's computer where the file is accessed.

2. Server System

This is the remote machine that actually stores the files.

3. NFS Client Module

This module runs on the client machine. It:

- Sends requests to the server for file operations (read, write, open, etc.).
- Receives the responses and acts like the file is local.
- Caches the file data in local memory for faster access.

4. NFS Server Module

This module runs on the server machine. It:

- Responds to client requests.
- Performs actual file operations on the server’s local file system.
- Uses **RPC (Remote Procedure Call)** to communicate with clients.

Important Concepts in NFS

1. Virtual File System (VFS)

- Acts as a **middle layer** between user applications and the file system.
- It checks whether a file is **local or remote**.
- Sends the request to either the **local file system** or the **NFS client module**.

2. File Handle

- A **unique identifier** used in NFS instead of a filename.
- It includes:
  - File system ID
  - i-node number (file's metadata and location)
  - i-node generation number

3. Access Transparency

- The user doesn't need to know whether the file is remote or local.
- All file operations look the same.

4. Mounting

- Mounting is the process of **attaching remote directories** to the local file system.
- Example: `mount(remote_host:/data, /home/user/data)`
- Types of mounting:
  - **Soft mount**: times out if server doesn't respond
  - **Hard mount**: keeps retrying until it succeeds
  - **Auto mount**: mounted automatically when accessed

11. Which are the assumptions made in Consensus and Agreement Algorithm

In distributed systems, getting processes to **agree** is hard, especially when some might **fail** or **misbehave**. To study and design consensus algorithms, we make a few important assumptions:

> **Short story to remember**
>
> When we handle processes, some of them might **fail (Failure Model)**. So, our first step is to check the expected delivery time to determine whether the system is **synchronous or asynchronous**.
>
> Once that's clear, we verify if the issue is due to **network connectivity**. If the connection still doesn't work, we send a complaint, making sure to include the **sender's identity**, and transmit it through **reliable channels**.
>
> This complaint is sent as an **authenticated message**, so it can't be forged or tampered with. Once it reaches the destination, an **agreement** can finally be reached.

1. Failure Model

- At most **f out of n processes** can fail.
- The behavior of faulty processes depends on the **failure model** (e.g., crash, Byzantine).
- This assumption affects **whether consensus is possible** and how complex the algorithm will be.

2. System Type: Synchronous vs Asynchronous

- In **asynchronous systems**, it's impossible to tell if a message is lost or just delayed.
- In **synchronous systems**, messages have a **known delivery time**, so missing messages can be detected and handled with defaults.
- The timing model greatly affects how consensus can be achieved.

3. Network Connectivity

- The network is **fully connected**: every process can send messages to every other process.

4. Sender Identification

- Every message includes the **true identity** of the sender.

- Even if a message is tampered with, **the receiver knows who sent it**.
- This helps limit the damage that faulty processes can do.

5. Reliable Channels

- Communication channels are assumed to be **reliable**.
- Only processes can fail, **not the network** (this simplifies the problem, though reality may differ).

6. Message Authentication

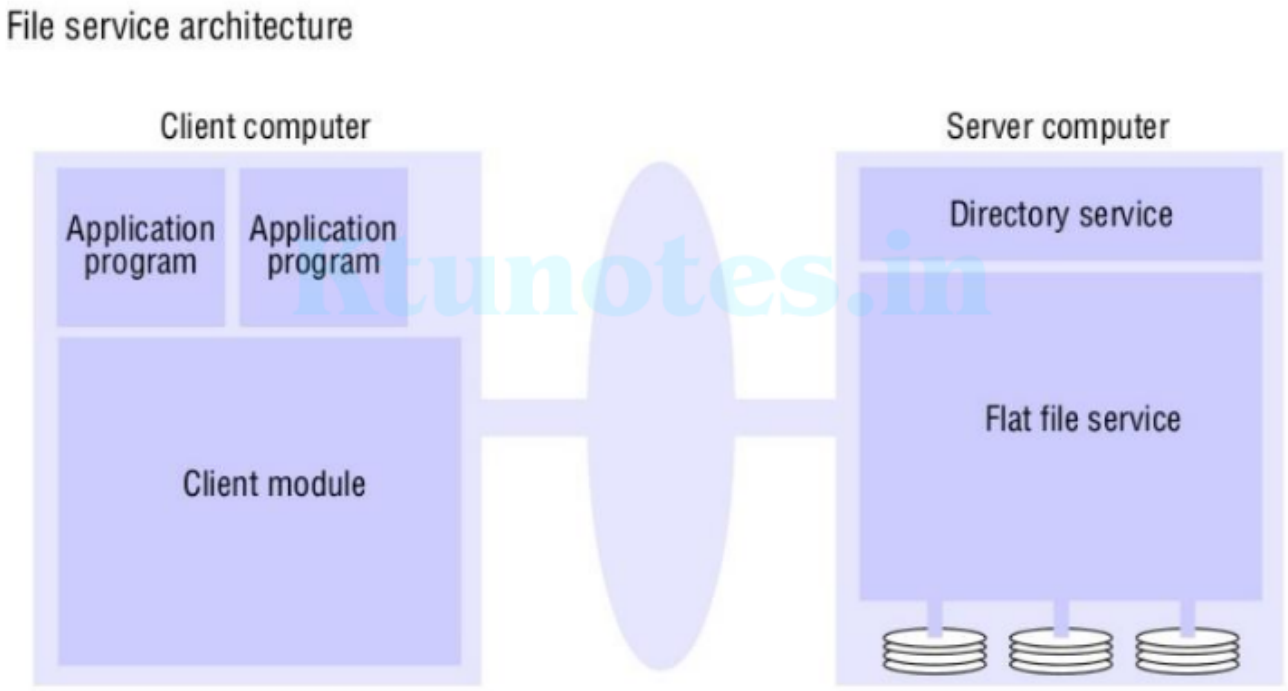
- Two cases:
  - **Unauthenticated messages**: Faulty processes can **forge or tamper** with messages (like spreading rumors).
  - **Authenticated messages**: With **digital signatures**, forgeries and tampering can be **detected**, making consensus easier.
- Authenticated systems are **more robust** against Byzantine faults.

7. Agreement Variable Type

- The variable being agreed upon can be **boolean or multivalued**.
- Often, a **boolean variable** is used to simplify examples and explanations, but it doesn't limit the generality of algorithms.

12. Explain about the file service architecture

Imagine you are using **Google Drive** to store, open, and edit files. You don’t worry about **where** the files are stored or how they are managed. **File Service Architecture** works in a similar way in a distributed system. It allows users to **store, retrieve, and manage files** over a network as if they were stored locally.



Three Main Parts of File Service Architecture

The architecture has **three key components**:

1. Flat File Service (Manages File Contents)

- Handles actual **reading, writing, creating, and deleting of files**.
- Uses **Unique File Identifiers (UFIDs)** instead of names to identify files.

2. Directory Service (Manages File Names)

- Keeps track of **file names** and maps them to **UFIDs**.
- Helps users **locate** files easily.

3. Client Module (Acts as a Bridge)

- Receives **user requests** (e.g., "open a file").
- Communicates with the **Directory Service** and **Flat File Service** to get the required file.
- Caches frequently used files for **faster access**.

Operations of File Service (What It Can Do)

The **Flat File Service** performs the following operations:

Basic File Operations

- **Read(i, n)** → Reads **n data items** starting from position **i** in the file.
- **Write(i, Data)** → Writes **Data** starting from position **i**, replacing old content if needed.

Creating and Deleting Files

- **Create()** → Makes a **new empty file** and assigns a **UFID**.
- **Delete(UFID)** → **Removes** the file from the system.

File Attributes (Metadata)

- **GetAttributes(UFID)** → Retrieves file details (size, owner, type).
- **SetAttributes(UFID, NewData)** → Updates file details (only allowed for authorized users).

How It Works (Step-by-Step)

1. **User wants to open a file (e.g., "notes.txt")**.
2. The **Client Module** asks the **Directory Service** for the file's **UFID**.
3. The **Directory Service** finds the **UFID** and sends it back.
4. The **Client Module** sends the **UFID** to the **Flat File Service**, which retrieves the file.
5. The **Client Module** sends the file to the user.

13. Write in detail about distributed file system characteristics.

A **Distributed File System** allows users to access and manage files stored across multiple servers as if they were located on their local system. Unlike traditional file systems that operate on a single machine, DFS provides seamless file access, sharing, and storage across a network of computers.

Characteristics of Distributed File Systems

Short story to learn

When we need to store something, we start by deciding how to **organize** the files and where to place them. Once that's set, we define the **file location** and store both the **data**

and its attributes. To keep things tidy, we place the files in separate **directories** and give them meaningful **names**. Finally, we save the **metadata** to help track and manage the files in the future.

1. Organization & Storage

- DFS handles the **organization, naming, retrieval, sharing, and protection** of files spread across various locations.
- It provides a consistent **programming interface** that abstracts the low-level details of storage allocation and file layout.

2. File Location & Access

- Files are stored on **non-volatile media** like hard disks or SSDs across different servers.
- From a user's perspective, accessing a file over DFS feels the same as accessing it locally.

3. Data and Attributes

- Each file consists of two parts:
  - **Data**: The actual content (e.g., documents, images, code).
  - **Attributes (Metadata)**: Information *about* the file.

4. Directories and Naming

- DFS supports **hierarchical file organization** using **directories**.
- A **directory** is a special file that maps **human-readable file names** to **internal file identifiers (IDs)**.
- This naming system allows users to locate and manage files easily, even in a large-scale distributed setup.

5. Metadata

- **Metadata** refers to all extra information that the file system keeps for managing files (including attributes, directory structure, and access permissions).
- It plays a key role in **tracking, securing, and organizing** files within the system.

14. Differentiate between AFS and NFS

Feature	AFS (Andrew File System)	NFS (Network File System)
Architecture	Uses a **client-server** model with **Vice** (server) and **Venus** (client)	Uses a **client-server** model with standard UNIX file system integration
Caching Strategy	**Whole-file caching**: Downloads entire file to client	**Block-level caching**: Transfers files in smaller chunks
Consistency Management	Uses **callback promises** for consistency	Uses **timestamp checking** for cache validation
Fault Tolerance	High fault tolerance with local cached copies	Moderate fault tolerance, relies more on the server
Performance	Better for **read-heavy workloads** due to local caching	May face **performance issues** under high load
Security	Supports **stronger authentication** methods (e.g., Kerberos)	Basic UNIX-style permission checks
Scalability	Highly scalable, designed for large numbers of users	Less scalable; performance degrades with more clients
File Sharing Granularity	Coarse-grained (entire files)	Fine-grained (blocks or parts of files)
Use Case	Designed for **academic and enterprise environments**	Widely used in **UNIX/Linux systems** for local networks
Statefulness	**Stateful** (remembers which files are cached)	**Stateless** (treats each request independently)