Q. Explain the features of Distributed File Systems (DFS) and draw and explain the model file service architecture

Here's a simplified explanation of the **desirable features** of a good **Distributed File System (DFS)**:

1. **Transparency**:
   - **Structure Transparency**: Users shouldn't know how many file servers or storage devices are being used. It should appear as if there's just one system.
   - **Access Transparency**: Users should access files the same way whether they are stored locally or on another computer.
   - **Naming Transparency**: The name of the file shouldn't reveal where it is stored. You should be able to move files without changing their name.
   - **Replication Transparency**: If a file is stored in multiple locations, users shouldn't be aware of it. The system handles it in the background.
2. **User Mobility**:
   - Users should be able to work on different computers or nodes without being tied to a specific one. They should have flexibility to switch machines easily.
3. **Performance**:
   - The system should be fast and responsive, similar to using a local file system, even when accessing remote files. It should minimize delays caused by network communication.
4. **Simplicity and Ease of Use**:
   - The system should be easy to understand and use. The file system interface should be simple, with few commands for the users to remember.
5. **Scalability**:
   - The system should be able to grow as more users and files are added. It should handle more machines and more users easily without a loss in performance.
6. **High Availability**:
   - The system should continue working even if some parts fail. It should provide access to files even if some servers or networks go down, and only cause temporary loss of service.
7. **High Reliability**:
   - The chance of losing data should be very low. The system should be reliable, ensuring users' data is safe.

8. **Data Integrity**:
    - If multiple users access the same file at once, the system should properly manage access to prevent data corruption. This is done through **concurrency control mechanisms**.
9. **Security**:
    - The system must be secure, ensuring that users' data is private and protected. Security measures should be in place to prevent unauthorized access.
10. **Heterogeneity**:

- The system should support different types of computer platforms. Users with different devices or operating systems should be able to access files without compatibility issues.

File Service Architecture Model:

Components:

1. **Client Machines (User Side)**:
    - Clients request files from the DFS. These clients interact with the **File System Interface (FSI)**.
2. **File Server**:
    - The server stores and manages files. It handles the actual reading and writing of files to disk.
3. **File System Interface (FSI)**:
    - This acts as a bridge between the client and the file server. It makes it possible for users to access files from remote servers as if they were local.
4. **File Data Manager**:
    - Responsible for the actual reading, writing, and caching of files. It handles the data that is being transferred between servers and clients.
5. **Meta Data Server**:
    - Stores information about the files, such as file names, locations, permissions, and access controls. It helps clients find the files they need.

How it works:

1. A **client** sends a request to access a file.
2. The **FSI** checks if the file is local or needs to be fetched from another server.
3. If the file is remote, the **Meta Data Server** helps the system find the file's location.

4. The **File Data Manager** retrieves the data and sends it to the client, either by copying it from the local server or accessing it from another remote server.
5. If necessary, data is **cached** on the client side for faster access next time.

This architecture allows **file access transparency** and efficient management of files in a distributed environment.

Q. Explain file sharing semantic of it.

File sharing semantics define how changes to a file are visible to multiple users or processes that access the file concurrently, especially in a distributed system.

There are four main types:

I) UNIX Semantics

- **Definition:** Follows strict time-based ordering of operations.
- **How it works:**
  - Every read gets the most recent write (if a write happened before it).
  - Assumes a single, consistent view of the file.
- **Example:** If Process A writes something to a file, and Process B reads it immediately after, B will see A's changes.
- **Use case:** Best when consistency is important.

II) Session Semantics

- **Definition:** Each process or client works in its own "session."
- **How it works:**
  1. Client **opens** a file.
  2. Performs **read/write**.
  3. **Closes** the file.
  - Changes made are only visible to that client until the session ends (file is closed).
- **Example:** Two users edit the same file at the same time, but don't see each other's changes until after saving.
- **Use case:** Common in network file systems like NFS.

III) Immutable Shared Files Semantics

- **Definition:** Files, once created and shared, **cannot be changed**.
- **How it works:**
    - A file is made *immutable* and shared.
    - Any "change" results in creating a new version.
- **Example:** Software version releases, where version 1.0 never changes.
- **Use case:** For content that must not change (e.g., scientific data, logs).

IV) Transaction-like Semantics

- **Definition:** Uses **transactions** to manage concurrent access.
- **How it works:**
    - A transaction starts (begin-transaction), operations are done, then it ends (end-transaction).
    - Ensures **atomicity**, **consistency**, **isolation**, and **durability (ACID)**.
    - Either all changes are committed or none.
- **Example:** Bank applications where concurrent transactions shouldn't interfere with each other.
- **Use case:** When consistency under concurrent operations is crucial.

Summary Table:

Semantics Type	**Visibility of Changes**	**Can Modify File?**	**Use Case**
UNIX Semantics	Immediately visible to all	Yes	Strict consistency
Session Semantics	Only visible after session	Yes	Loose consistency (NFS-like)
Immutable Files	Never changes after creation	No	Read-only shared content
Transaction Semantics	After successful commit	Yes	Banking, databases, critical apps

Q. Discuss File caching for Distributed Algorithm

File caching is a technique used to speed up file access in distributed systems. Instead of constantly reading data from a disk, the system stores frequently used files (or parts of files) in memory so they can be accessed more quickly.

Why Use File Caching?

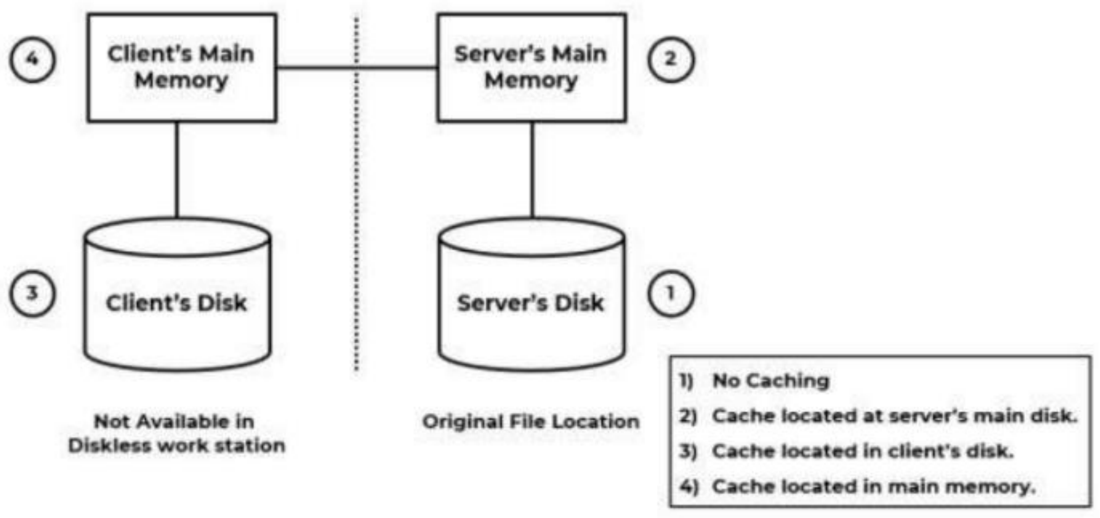
- **Goal:** The main goal of file caching is to store recently used data in memory. This way, when the same data is needed again, it can be retrieved much faster without having to go to the disk.

Key Design Issues in File Caching

When setting up file caching in a distributed system, there are a few important decisions to make:

1. Cache Location

Where should the cached data be stored? There are three options:


4
Client's Main Memory
Server's Main Memory
2
3
Client's Disk
Server's Disk
1
Not Available in Diskless work station
Original File Location
1) No Caching
2) Cache located at server's main disk.
3) Cache located in client's disk.
4) Cache located in main memory.

- **Server's Main Memory:**
  - Files are stored in the server's memory.
  - **Pro:** Fast access because no disk is involved.
  - **Con:** The file still needs to be sent over the network to the client.
- **Client's Disk:**
  - Files are cached on the client's disk.
  - **Pro:** No network access needed.
  - **Con:** Disk access is still slow.
- **Client's Main Memory:**
  - Files are cached in the client's RAM.

        - **Pro:** Fastest access, as it avoids both network and disk access.
        - **Con:** Uses client's memory, which might be limited.

2. Modification Propagation

When one client changes the cached file, how do we make sure that other clients have the latest version?

- **Write-through Scheme:**
    - Any change made to the file is immediately sent to the server to update the master copy.
    - **Pro:** The data is always up-to-date.
    - **Con:** Can slow down the system because it involves frequent network communication.
- **Delayed-write Scheme:**
    - Changes are first made to the cache. Later, the updated data is sent to the server.
    - Three approaches:
        - **Write on Ejection:** The file is sent to the server when the cache decides to remove it.
        - **Periodic Write:** Data is sent to the server after certain time intervals.
        - **Write on Close:** Data is sent when the client finishes using the file (i.e., when the file is closed).

3. Cache Validation

To keep cached data accurate, we need to make sure that the cached copy is still up-to-date with the master copy on the server.

- **Client-Initiated Validation:**
    - The client checks with the server to see if the cached file is still valid.
    - Different approaches to validation:
        - **Check before Every Access:** The client checks with the server every time it needs to access the file.
            - **Con:** This defeats the purpose of caching because the server is contacted all the time.
        - **Periodic Check:** The client checks after a certain amount of time.
        - **Check on File Open:** The client checks when it opens the file.

- **Server-Initiated Validation:**
    - The client tells the server when it opens a file, and the server keeps track of who is using the file and in what way (reading or writing).
    - **Pro:** The server manages which clients can read/write to the file, avoiding conflicts.
    - **Con:** It adds complexity for the server to track file usage.

Key Challenges:

- **Consistency:** We need to make sure all clients have the same version of the file.
- **Efficiency:** File caching should make things faster, not slower, so balancing when and how often updates and checks happen is crucial.

Q. Explain File Accessing Models

In a **distributed file system**, when a client wants to access a file, it uses a **file accessing model**. This model decides how the file is transferred and accessed. There are two key parts:

1. **Unit of Data Access**: How much data is transferred in a single operation.
2. **How Remote Files are Accessed**: How the client interacts with the file server to get the file.

I) Unit of Data Access

This refers to how much file data is transferred to or from the client in one read or write action. There are four ways to handle this:

a) File Level Transfer Model

- **How it works**: The entire file is sent to the client.
- **Advantages**:
    - Efficient because the network setup happens only once.
    - Reduces load on the server and network traffic.
- **Disadvantage**:
    - Needs a lot of storage space to store the entire file.
- **Example systems**: Amoeba, CFS, Andrew File System.

b) Block Level Transfer Model

- **How it works**: The file is divided into blocks (smaller parts), and only the needed blocks are transferred.
- **Advantages**:
  - Doesn't need as much storage because only blocks are transferred.
- **Disadvantages**:
  - Increases network traffic because more blocks need to be sent.
  - Higher overhead (extra work to manage the network).
  - Slower performance.
- **Example systems**: Sun Microsystems NFS, Apollo Domain File System.

c) Byte Level Transfer Model

- **How it works**: Data is transferred one byte at a time.
- **Advantages**:
  - Very flexible; you can send just small parts of a file.
- **Disadvantage**:
  - Harder to manage cached data (keeping copies locally).

d) Record Level Transfer Model

- **How it works**: The file is split into records (like rows in a database), and only the records that are needed are transferred.
- **Best for**: Files with structured data (like databases).
- **Example system**: RSS (Research Storage System).

II) Accessing Remote Files

When a client wants to access a file on a different machine (server), there are two main models:

a) Remote Service Model

- **How it works**:
  - The client sends a request to the server.
  - The server processes the request and sends the result back to the client.
- **Disadvantage**:
  - Every time a client needs a file, it has to send a request, causing network traffic.

b) Data Caching Model

- **How it works**:
  - The client keeps a **local copy** of data that is used frequently (in a cache).
  - If the client needs data that isn't in the cache, it fetches it from the server and stores it locally.
  - **LRU (Least Recently Used)** helps manage the cache size by removing old data.
- **Advantages**:
  - Reduces network traffic by avoiding repeated requests for the same data.
- **Disadvantages**:
  - There's a **cache consistency problem** – if the data changes on the server, the local copy might be outdated.

Q. Explain Andrew File System (AFS).

The **Andrew File System (AFS)** is a **distributed file system** that was created to help share files across many computers in a network. It was developed by Carnegie Mellon University as part of the **Andrew Project**. Here's a simple explanation of AFS:

Key Points about AFS:

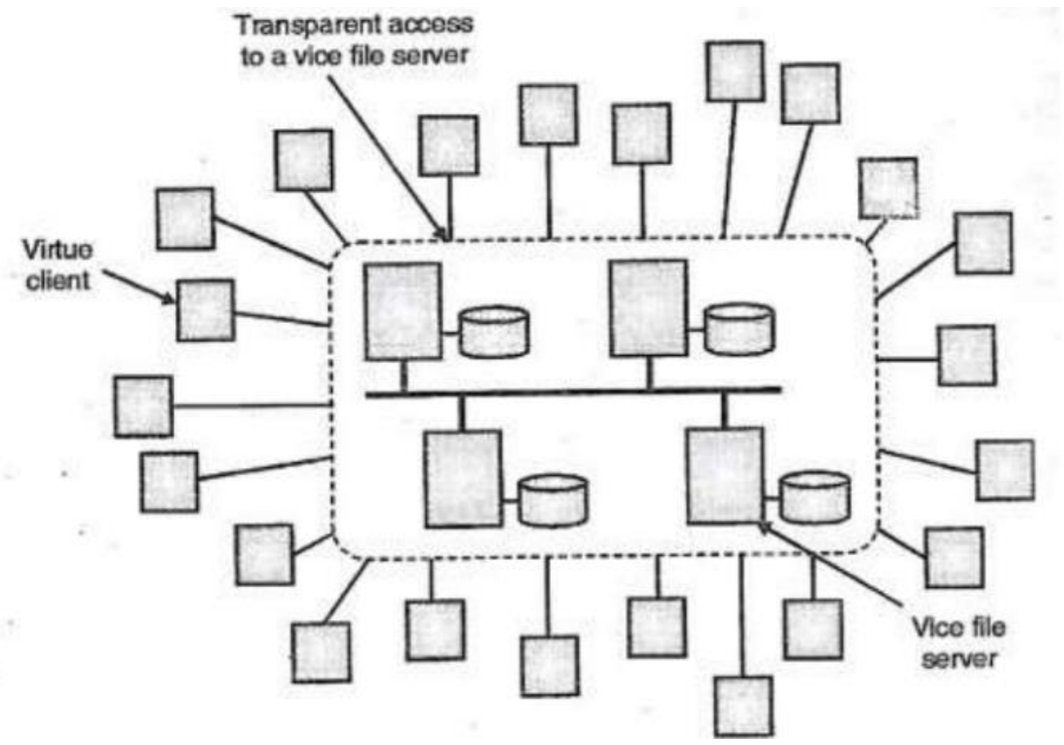
1. **Distributed File System**: AFS allows files to be stored and accessed across multiple computers in a network, making it easier to share files.
2. **Origin**: It was originally called "Vice" and is now named after **Andrew Carnegie** and **Andrew Mellon**.
3. **Purpose**: AFS is mainly used in **distributed computing**, where data is shared and accessed across many machines.
4. **Session Semantics**: AFS operates with session semantics, meaning it ensures that users can access files as though they are on their local machines, even if they are stored on different servers.
5. **Large Scale Sharing**: It supports sharing of information among thousands of users and works well for large-scale environments.

Features of AFS:

- **Scalability**: AFS is designed to handle large amounts of data (terabytes) and thousands of users.

- **Wide Area Network (WAN) Support**: It is designed to work in **WAN environments**, meaning it can efficiently operate across large geographical distances.

AFS Architecture:



AFS has two main types of nodes:

1. **Vice Node**: This is a **file server** that stores and manages the actual files.
2. **Virtue Node**: This is a **client machine** that accesses the files stored on the Vice Node.

When a file is opened:

- The entire file is copied from the Vice Node (server) to the Virtue Node (client machine).
- After the file is modified, the changes are sent back to the server when the file is closed.

Limitations of AFS:

1. **Service Unavailability**: If the servers or network components fail, the system becomes unavailable.
2. **Scaling Problems**: AFS can face challenges as it grows in size or is used by more people.

Q. Explain Naming and explain system-oriented names in details

NAMING:

In a **distributed system**, there are many different objects like:

- Processes
- Files
- I/O devices
- Mailboxes
- Nodes (computers or machines)

To manage these objects, the system provides a **naming facility**. This allows users and programs to give names (like labels or identifiers) to these objects. The naming system helps achieve **location transparency**, which means that users don't need to know where an object is physically located in the system.

SYSTEM ORIENTED NAMES (Unique Identifiers)

These names are also called **low-level names** or **unique identifiers**. They are:

- **Bit patterns or large numbers** that computers can easily understand and manage.
- Used to **uniquely identify objects** in the system.
- These names help the system operate efficiently by tracking and organizing objects.

Types of System-Oriented Names:

1. **Structured Names**:
   - They have **multiple parts** and may contain information about the object's ID.
   - Example: A name could be something like user123.filesystem.device, where each part of the name gives more information about the object.
2. **Unstructured Names**:
   - They are **simple** and usually consist of a large number or bit string.
   - They don't provide any extra information about the object.
   - Example: A name could be just a large number like 1234567890.

Characteristics of System-Oriented Names:

1. They are **large numbers or bit strings** (a string of 0s and 1s).
2. **Unique**: Each name is unique to an object.
3. All system-oriented names are of the **same size**.
4. They are **shorter** than names meant for humans (like filename.txt).

5. They are **hard to guess**, so they are also used for **security** (e.g., passwords).
6. These names are **automatically generated** by the system.

Two Approaches to Generating System-Oriented Names:

I) Centralized Approach:

- In this approach, a central system or server generates names for all the objects in the system.
- **Unstructured names** are mostly used here.
- **Global unique identifiers** (a special name) are given to objects in the system.
- Each **node (machine)** in the system either:
    - Uses the global identifier directly or
    - Maps it to a local identifier if it's a local object.

Advantages:

- Simple and easy to set up.
- It's the only method for generating global **unstructured identifiers**.

Disadvantages:

- **Low reliability**: If the central system goes down, the whole process might fail.
- **Low efficiency**: The central system can get overloaded.

II) Distributed Approach:

- This approach is used to overcome the issues of the centralized system (like reliability and efficiency).
- It uses **structured names** for objects.
- The **global unique identifier** is created by combining the identifier of the **domain** (group of objects) with the identifier of the object within that domain.

How it works:

1. A unique identifier is created for each domain.
2. A global identifier is then created by combining the domain's identifier with the object's local identifier.

Issues:

- If the system **crashes**, it may lose the state information needed to generate unique identifiers.
- Even after recovery, the identifier generator may not work correctly.

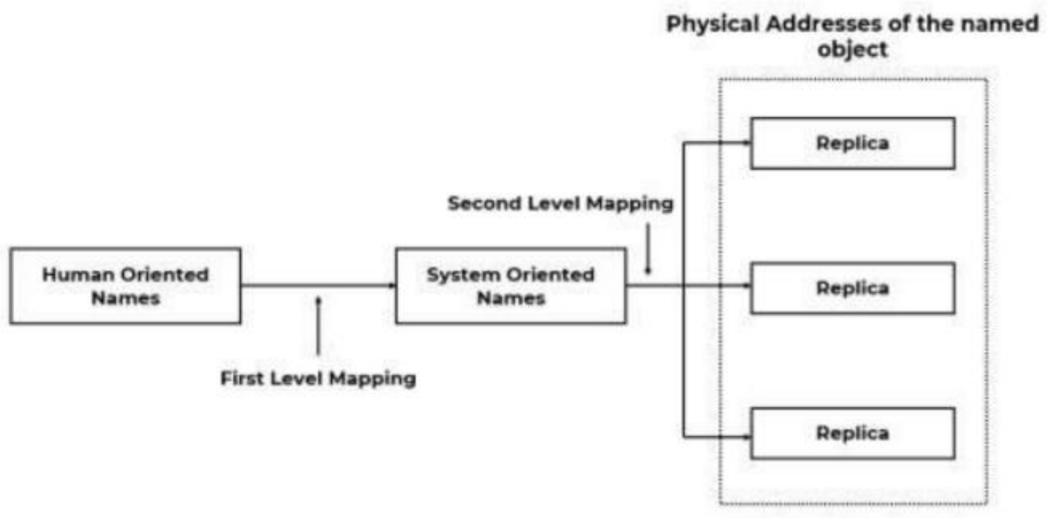
Solutions:

1. **Clock-based Approach**: Use a **clock** that continues to work even if there is a failure, helping to avoid generating duplicate IDs.
2. **Multiple Levels of Storage**: Organize unique identifiers in a **hierarchical way** (using multiple fields for each level) to prevent conflicts.

Q. Explain human oriented names in details

HUMAN ORIENTED NAMES:

1. **Human-oriented names** are **names that make sense to people** (like file names or object labels).
2. These names are **created and used by people** (not by the system).
3. **Different people** can give **different names** to the same object. This helps people use names that make sense to them.
4. **Aliasing** is supported, which means two people can use the **same name** to refer to different objects.
5. This system makes managing names easy because users can use **names they understand**.



Characteristics of Human-Oriented Names:

1. They are **meaningful** to people (like naming a file “resume.doc” or “home_folder”).
2. **Users define and use** these names.
3. **Multiple names** can refer to the **same object** for different users.

4. These names can be **short or long** (so they are of **variable length**).
5. They are **not unique**—different people can give the same name to **different things**.

Issues with Human-Oriented Names:

There are a few challenges:

1. **Choosing a naming system** that works for objects everyone can access.
2. Deciding how to **organize names** into **groups or categories** (like organizing files in folders).
3. Making sure names are **correctly connected** to what they refer to (e.g., ensuring a file named "meeting_notes" actually refers to the notes).
4. Figuring out a way to **resolve names**—that is, finding out what the name is referring to when someone uses it.

Q. Explain name resolution in DNS

What is DNS?

- **DNS** stands for **Domain Name System**.
- It helps you find **host addresses** (like the IP address of a website) and **mail servers**.
- It converts easy-to-remember names (like **google.com**) into IP addresses (like **172.217.6.68**) that computers use to communicate.

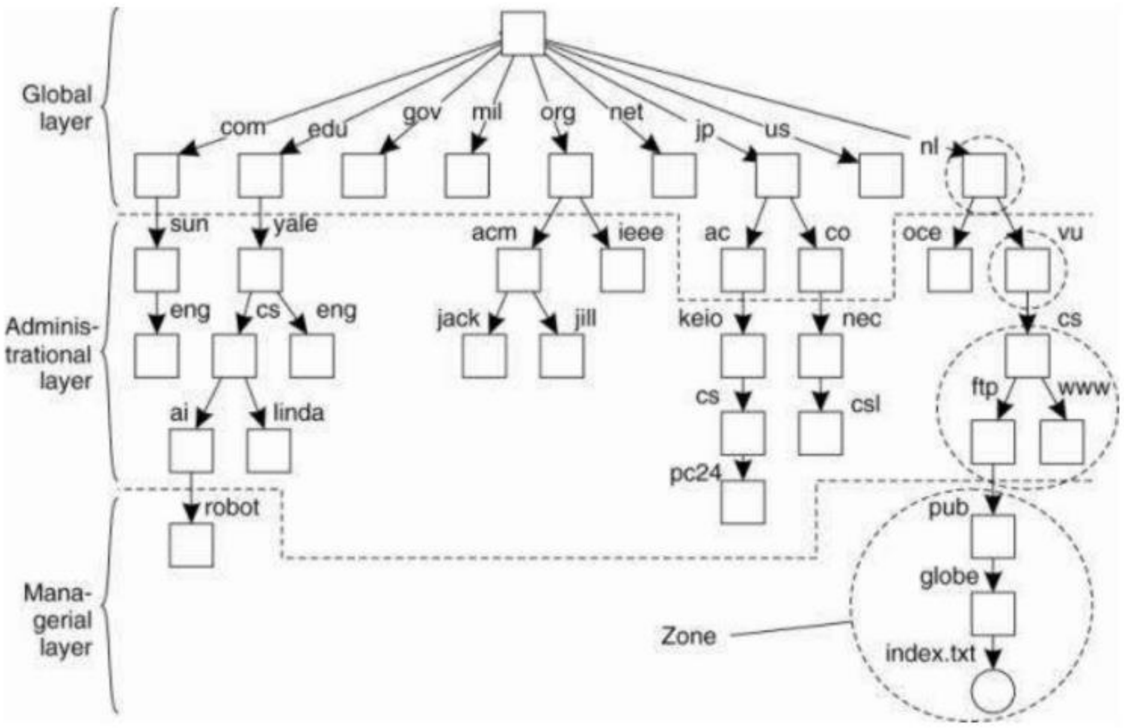
How DNS Names Are Structured:

- DNS names are written as a list of labels separated by **dots**.
    - Example: **aa.bb.cc.dd** — where the rightmost part (dd) is the **root**.
- A DNS **name** can be:
    - **Absolute**: The full name with the root.
    - **Relative**: A shorter version without the full path.

DNS Name Resolution:

1. **Name Resolution** is the process of converting a **name** (like "google.com") into useful information, like its **IP address**.
2. It connects a **name** to the **authoritative name servers** (servers that have the correct information about the domain).

3. In a **distributed system**, each **name agent** (like your computer) knows at least one **DNS server**.
4. The DNS tries to turn human-friendly names into machine-readable addresses.
5. A **label** (a part of the DNS name) is made up of **letters** and **numbers**, and can be **up to 63 characters** long.
6. The **root** is represented by a **dot** (.), and the entire DNS system is organized like a **tree**.



DNS Name Resolution Process:

1. **Request**: When you type a domain name (like "google.com"), your computer asks a **DNS server** to find the corresponding IP address.
2. **Check Cache**: The DNS server first checks its **local cache** to see if it already has the IP address saved.
3. **Host File**: If it's not in the cache, the DNS server checks its **host file** (a list of known names and addresses).
4. **Forwarding**: If the server still doesn't know, it forwards the request to a **higher-level DNS server**.
5. **Root DNS Server**: If needed, the request is sent to the **root DNS server**, which directs it to a **Top-Level Domain (TLD)** server (like .com, .net).
6. **Final Response**: The process continues until the correct **IP address** is found and returned.

DNS Hierarchy:

1. **Top-Level Domains (TLDs)**: These are the last parts of a domain name, like .com, .org, .edu.
2. **Domains**: These are names under TLDs, like **google** in **google.com**.
3. **Hosts**: These are individual machines or services within a domain, like the web server for **google.com**.

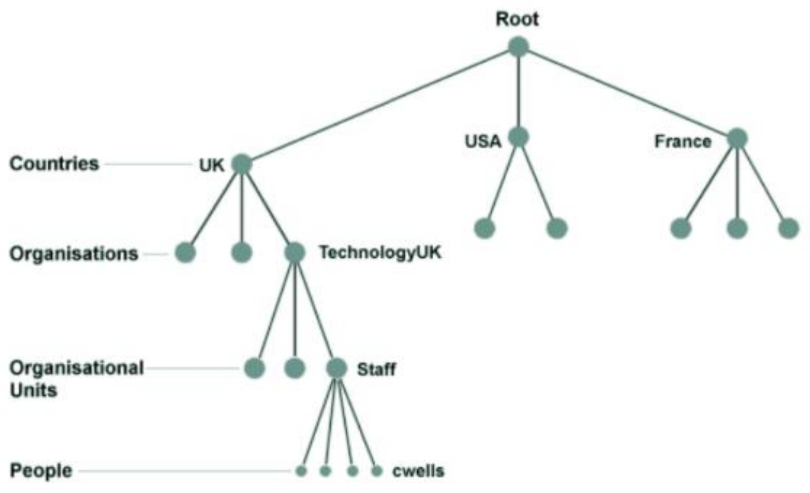
Q. Explain x.500 directory service in detail

What is X.500?

- **X.500** is a set of standards for **directory services** used in computer networks.
- It was developed by the **International Telecommunication Union (ITU-T)**.
- It defines a global, hierarchical directory service to store and retrieve information about objects like people, devices, or resources in a network.

Key Features of X.500:

1. **Directory Service**: It is a standard-based directory system for applications that need to store and access directory information.
2. **Global Directory**: X.500 provides a **global directory** with a hierarchical structure, meaning it is organized in levels (like folders inside folders).
3. **Data Management**: It allows adding, viewing, changing, and deleting information in the directory.
4. **Search Capabilities**: You can search the directory based on specific data queries to find exactly what you need.



How Does X.500 Work?

1. **Global Directory Service**: X.500 is a global directory system that is made up of multiple components.
2. **Directory Management Domains (DMDs)**: These are the main units that manage the X.500 service. Each DMD includes a **Directory System Agent (DSA)** and is managed by a **Domain Management Organization (DMO)**.
3. **Two Types of DMDs**:
    - **ADDMDs (Administrative DMDs)**: These provide public directory services, like services that help find contact information over the internet (e.g., **Four11** and **Bigfoot**).
    - **PRDMDs (Private DMDs)**: These provide private directory access, like a company's directory service (e.g., **Microsoft Active Directory**).

Components of X.500:

1. **Directory Information Base (DIB)**:
    - It is a **hierarchical database** that stores all the directory information.
    - The data is spread across different servers in different locations.
2. **Directory System Agent (DSA)**:
    - A **server** that holds a part of the DIB and allows users to connect and access the directory.
3. **Directory User Agents (DUAs)**:
    - **Client software** that allows users to search and access the directory information through the DSA.

Q. Explain NFS in Distributed Systems

What is NFS?

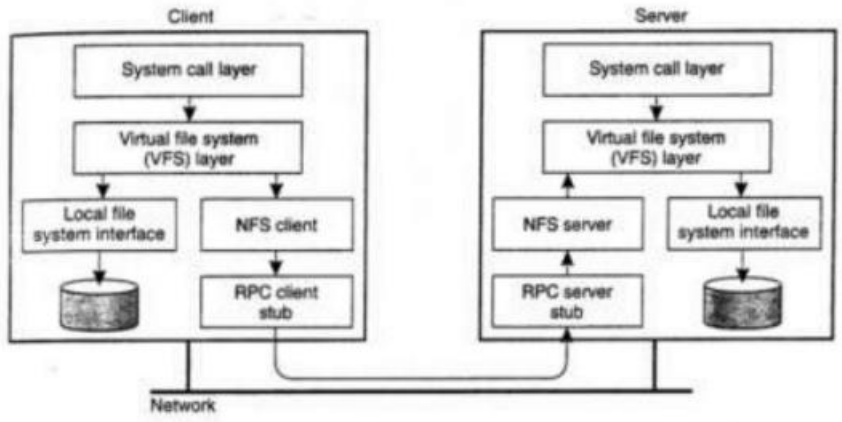
- **NFS** stands for **Network File System**.
- It was created by **Sun Microsystems** in **1984** to allow file sharing across different computers.
- NFS allows one computer (called the **server**) to share files with other computers (called **clients**) over a network.

How NFS Works:

1. **Client-Server Model**:
    - NFS follows a **client-server** model where the server shares its files, and clients can access them.
    - Clients can access files on the server **as if they were local files** on their own computer.
2. **Virtual File System (VFS)**:
    - NFS works using the **RPC protocol** (Remote Procedure Call) and **Virtual File System (VFS)** to access files over the network.
    - VFS makes it easy for the system to work on top of the network and share files as if they were local.

NFS Servers (Computers that Share Files):

- **NFS Servers** are computers that share their files.
- In the past, powerful workstations with lots of storage would act as servers, and **diskless workstations** (computers without local storage) would access the files from these servers.


Client
Server
System call layer
Virtual file system (VFS) layer
Local file system interface
NFS client
RPC client stub
System call layer
Virtual file system (VFS) layer
NFS server
Local file system interface
RPC server stub
Network

Benefits of NFS:

1. **Simplifies File Management**:
    - Instead of having the same files on every system, NFS allows one copy of files (like **/usr/local**) to be shared by all computers on the network.
2. **Simplifies Backups**:
    - With NFS, you only need to back up the files on the server, rather than backing up files on every client computer.

NFS Clients (Computers that Access Shared Files):

- **NFS Clients** are computers that access files shared by the NFS server.

- Clients use a combination of **kernel support** and **user-space software** to access remote files.
- Multiple clients can access the same shared files at once, and the system can mount files automatically when the computer starts up.

Goals of NFS Design:

1. **Compatibility**:
   - NFS should work just like a local file system, meaning programs shouldn't have to know if a file is local or remote.
2. **Easy Deployment**:
   - NFS should be easy to set up without requiring changes to existing systems.
3. **Machine and OS Independence**:
   - NFS should work across different machines and operating systems, not just Unix-based ones.
4. **Efficiency**:
   - NFS is designed to be good enough for users, even if it's not as fast as a local file system.
   - It should handle machine crashes and network problems smoothly.

NFS Protocol:

- NFS uses **Remote Procedure Calls (RPC)** to communicate between the client and the server.
- **Synchronous RPC**: The client waits for the server to respond before continuing.
- **Stateless Protocol**: The server doesn't remember past requests. This makes it easier to recover if something goes wrong. If the server crashes, the client can simply resend the request.

Q. Analyse the architecture and performance of the Andrew File System (AFS) compared to the Network File System (NFS). Discuss the advantages and limitations of each.

Andrew File System (AFS)

- **Overview**:

  - AFS is a distributed file system developed by Carnegie Mellon University for large-scale distributed computing environments.
  - It is mainly used for file sharing across large networks, supporting information sharing among many users and large data sets.
- **Architecture**:
  - AFS uses two types of nodes:
    1. **Vice Nodes**: These are the file servers where data is stored.
    2. **Virtue Nodes**: These are the client machines that access files.
  - When a file is opened, it is copied from the server (Vice Node) to the client machine (Virtue Node). Once the file is changed, it is sent back to the server when closed.
- **Performance**:
  - **File Caching**: AFS caches entire files locally on the client side, which improves performance for frequently accessed files.
  - **Scalability**: AFS is good for handling large volumes of data and many users but faces challenges in scaling, especially when servers or network components fail.
- **Limitations**:
  - **Server/Network Failures**: If the server or network fails, access to files can be temporarily unavailable.
  - **Scaling Problems**: As the number of users and files increases, performance may degrade.

Network File System (NFS)

- **Overview**:
  - NFS is a network-based file system created by Sun Microsystems that allows file sharing across different machines, regardless of their operating systems.
  - It provides a way for applications to access remote files just as if they were local files.
- **Architecture**:
  - NFS works on a **client-server** model, where the server shares files, and the client accesses them over the network.
  - It uses **Remote Procedure Calls (RPC)** to communicate between clients and servers.

- **Performance**:
    - **Stateless Protocol**: NFS is stateless, meaning the server doesn't store information about past requests. This helps in crash recovery but can result in inefficiencies, especially for frequent file updates.
    - **Network Dependency**: NFS relies on the network for file access, so performance can degrade if the network is slow or unreliable.
- **Limitations**:
    - **Network Issues**: Since it depends heavily on the network, NFS can face performance problems if the network is slow or there are interruptions.
    - **No Local Caching**: NFS doesn't cache files locally on the client, which means more network traffic and potentially slower access for frequently used files.

Comparison: Advantages and Limitations

Feature	**AFS (Andrew File System)**	**NFS (Network File System)**
File Access	Copies entire files to the client (local caching).	Does not cache files locally, always gets them from the server.
Performance	Faster for files that are used often (due to caching).	Can be slower, as it always fetches files over the network.
Scalability	Works well with large amounts of data and users, but can struggle when scaling too much.	Works fine in small to medium setups, but can slow down with many clients.
Network Dependency	Works well in WAN (wide-area networks), but still needs the server.	Very dependent on the network for file access. A slow network can slow it down.
Fault Tolerance	Not very fault-tolerant; if the server or network fails, file access is affected.	Easy recovery if server fails, as it doesn't store request information (stateless).
Cross-Platform Support	Works in specific environments (often used in academic or enterprise settings).	Works across many platforms like Unix, Linux, and Windows.
File Locking	Supports file locking, ensuring no conflicts when multiple users access a file.	Limited file locking, which can lead to conflicts if many users access the same file.

Feature	AFS (Andrew File System)	NFS (Network File System)
Security	Strong security (uses Kerberos for authentication).	Basic security (but can be enhanced with NFSv4 and Kerberos).
Replication	Supports data replication for better availability.	Does not have built-in replication.
Setup & Management	More complex to set up and manage.	Easier to set up and manage, widely used.
Backup	Centralized backups since files are managed centrally.	Only the server's files need to be backed up.

Q. Explain Hadoop Distributed File System (HDFS).

HDFS is a distributed file system designed to store large amounts of data across many computers in a network. It's part of the **Hadoop** ecosystem, which is used for big data processing. Here's a simple breakdown of how it works:

Key Features of HDFS:

1. **Designed for Large Files**:
   - HDFS is optimized to store and process very large files (e.g., videos, images, logs) across multiple computers.
2. **Fault Tolerance**:
   - It stores multiple copies (replications) of each file in case something goes wrong (like a computer or disk failure). By default, it keeps 3 copies of each file to ensure reliability.
3. **Data is Stored in Blocks**:
   - Instead of storing the whole file as one unit, HDFS breaks the file into smaller pieces called **blocks** (usually 128MB or 256MB).
   - Each block is distributed across different machines in the system.
4. **Master-Slave Architecture**:

    - **NameNode**: The "master" that manages the file system. It keeps track of where each file is stored and manages metadata (like file names, permissions, etc.).
    - **DataNode**: The "slaves" that store the actual data blocks. There are usually many DataNodes in a Hadoop cluster.
5. **High Throughput**:
    - HDFS is optimized for reading large amounts of data at high speed. It's not designed for fast random access like a traditional file system.

How HDFS Works:

1. **Writing Data**:
    - When a file is uploaded to HDFS, it's split into blocks and stored across multiple DataNodes.
    - The **NameNode** keeps a record of where these blocks are stored, but doesn't actually store the data itself.
2. **Reading Data**:
    - To read a file, the client requests the file from the **NameNode**, which tells it where the blocks are stored on the DataNodes.
    - The client then retrieves the data directly from the DataNodes.
3. **Data Replication**:
    - Each block of data is replicated (by default 3 times) on different DataNodes for fault tolerance.
    - If one DataNode fails, another copy of the block is still available on a different node.

Advantages of HDFS:

1. **Scalability**:
    - HDFS can scale out easily by adding more DataNodes, allowing it to store petabytes of data.
2. **Fault Tolerance**:
    - The system ensures that even if a DataNode fails, data is still safe because there are multiple copies (replicas) of each block.
3. **High Throughput**:
    - It's designed for fast data processing, especially for large files, which is why it works well with **big data** tools like **Hadoop MapReduce**.

4. **Cost-Effective**:
    - Since HDFS can run on commodity hardware (cheap, regular servers), it's very cost-effective compared to traditional storage systems.

Disadvantages of HDFS:

1. **Not for Small Files**:
    - HDFS is optimized for storing large files, so it's not suitable for applications that need to handle small files (like databases).
2. **Not Ideal for Low-Latency Access**:
    - HDFS isn't designed for real-time access or low-latency applications. It works better for batch processing.

Q. compare replication and caching

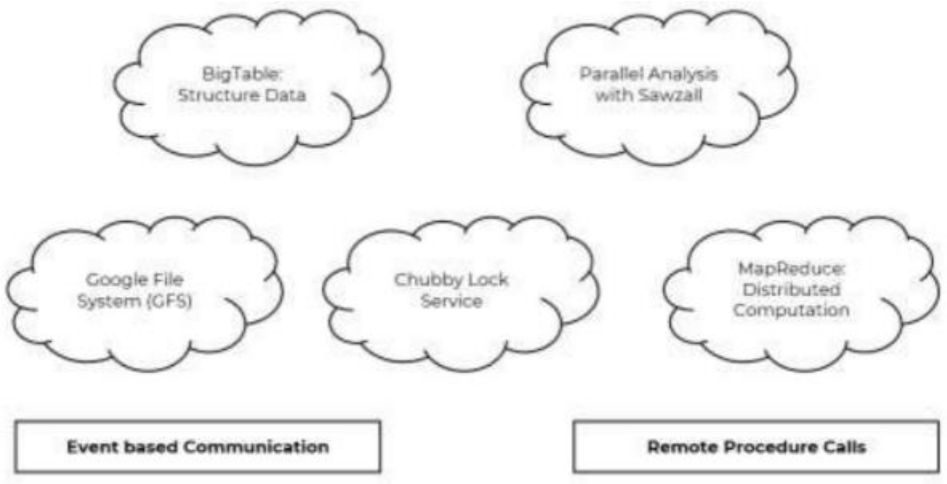
Aspect	Replication	Caching
Definition	Making copies of data at different locations.	Storing data temporarily for faster access.
Purpose	To make sure data is always available.	To speed up access to data.
Storage	Data is stored in multiple places (servers).	Data is stored temporarily (e.g., in memory).
Consistency	Needs to keep all copies of data the same.	Data may not always be up-to-date.
Updates	Changes are copied to all locations.	Data may not update until refreshed.
Speed	Can be slower due to syncing between copies.	Much faster as data is stored locally.
Fault Tolerance	Works well if some copies go down.	Doesn't provide backup if the original data fails.
Examples	Distributed databases, cloud storage.	Web browser history, CDN (Content Delivery Network).

Aspect	Replication	Caching
Data Duration	Long-term storage.	Short-term storage for quick access.
Cost	More expensive because of extra storage.	Less expensive, uses temporary storage.

Q. Explain components of Google infrastructure

Google Infrastructure Components:

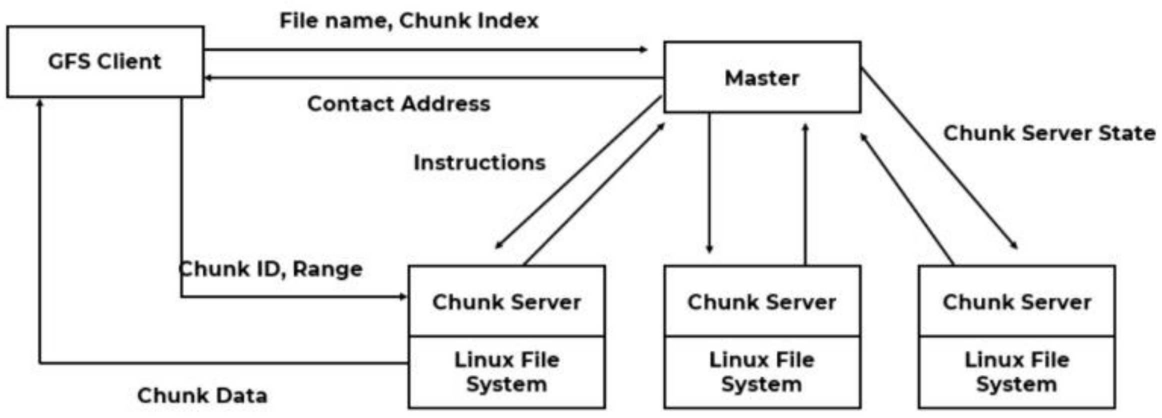
1. **Products**: These are the services that Google offers to users. Examples include:
    - **Search**
    - **Advertising**
    - **Email**
    - **Maps**
    - **Video (YouTube)**
    - **Chat**
    - **Blogger**
2. **Distributed Systems Infrastructure**: These are the tools and technologies that Google uses to manage its data and services at scale. Key components are:
    - **Google File System (GFS)**: A system that stores large files across many servers.
    - **MapReduce**: A system for processing large datasets in parallel.
    - **BigTable**: A distributed database system for managing large amounts of structured data.
3. **Computing Platforms**: This is the hardware that Google uses to run its services, consisting of:
    - A **massive number of servers** in different data centers located all around the world.


BigTable: Structure Data
Parallel Analysis with Sawzall
Google File System (GFS)
Chubby Lock Service
MapReduce: Distributed Computation
Event based Communication
Remote Procedure Calls

Key Components Explained:

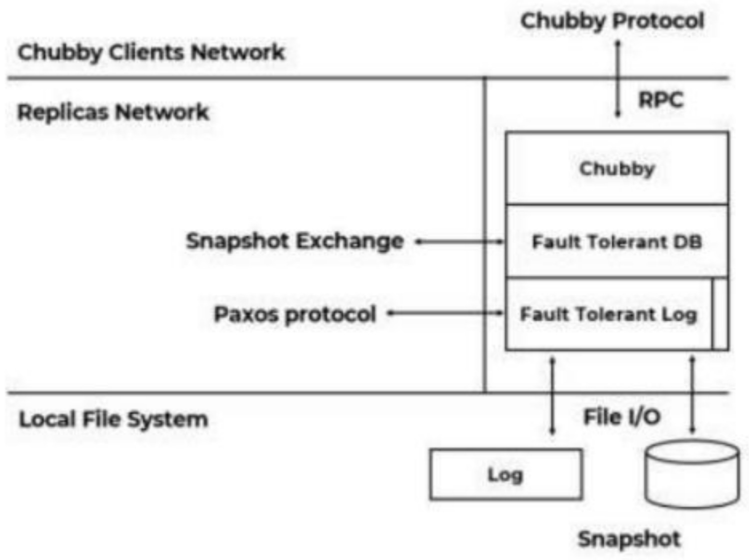
I) Google File System (GFS):

- Google uses **GFS** to store **large files** (sometimes gigabytes or more).
- Instead of changing parts of a file, GFS **appends** data.
- GFS is designed to handle **server failures** by having **multiple copies of data**.
- It works by breaking files into **64MB chunks**, which are stored across multiple **chunkservers**.



II) Chubby Lock Service:

- **Chubby** is a **lock service** used to manage access to shared resources in Google's distributed systems.
- It helps with:
  - **Coarse-grained locking** (avoiding conflicts in distributed systems).
  - Acting as a **small file system** (for metadata storage).
  - **Achieving consensus** in distributed systems using algorithms like **Paxos**.



III) BigTable:

- **BigTable** is a **distributed table** that stores data in a highly scalable way.
- It's like a **huge database** organized by rows, columns, and timestamps.
- **BigTable** is used by many Google services like **Google Earth**, and it's built on top of **GFS** and **Chubby**.

IV) MapReduce:

- **MapReduce** is a framework for processing large datasets across many computers.
- It makes it easier to perform **parallel computations** (like searching through huge amounts of data).
- It automatically handles things like:
    - **Parallelization** (dividing tasks across machines).
    - **Load balancing** (distributing work evenly).
    - **Error recovery** (handling server crashes).

Q.