Unit-3

Resource Sharing in Distributed System

Resource sharing in distributed systems is very important for optimizing performance, reducing redundancy, and enhancing collaboration across networked environments. By enabling multiple users and applications to access and utilize shared resources such as data, storage, and computing power, distributed systems improve efficiency and scalability.

Importance of Resource Sharing in Distributed Systems

Resource sharing in distributed systems is of paramount importance for several reasons:

- **Efficiency and Cost Savings**: By sharing resources like storage, computing power, and data, distributed systems maximize utilization and minimize waste, leading to significant cost reductions.
- **Scalability**: Distributed systems can easily scale by adding more nodes, which share the workload and resources, ensuring the system can handle increased demand without a loss in performance.
- **Reliability and Redundancy**: Resource sharing enhances system reliability and fault tolerance. If one node fails, other nodes can take over, ensuring continuous operation.
- **Collaboration and Innovation**: Resource sharing facilitates collaboration among geographically dispersed teams, fostering innovation by providing access to shared tools, data, and computational resources.
- **Load Balancing**: Efficient distribution of workloads across multiple nodes prevents any single node from becoming a bottleneck, ensuring balanced performance and preventing overloads.

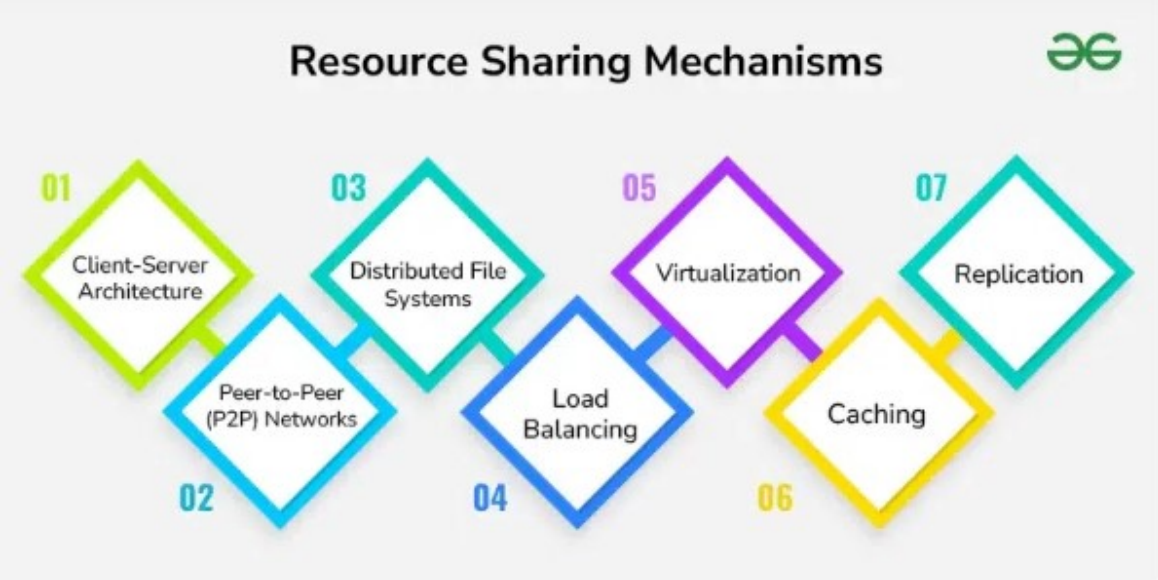
Types of Resources in Distributed Systems

In distributed systems, resources are diverse and can be broadly categorized into several types:

- **Computational Resources**: These include CPU cycles and processing power, which are shared among multiple users and applications to perform various computations and processing tasks.
- **Storage Resources**: Distributed storage systems allow data to be stored across multiple nodes, ensuring data availability, redundancy, and efficient access.
- **Memory Resources**: Memory can be distributed and shared across nodes, allowing applications to utilize a larger pool of memory than what is available on a single machine.
- **Network Resources**: These include bandwidth and network interfaces, which facilitate communication and data transfer between nodes in a distributed system.
- **Data Resources**: Shared databases, files, and data streams that are accessible by multiple users and applications for reading and writing operations.
- **Peripheral Devices**: Devices such as printers, scanners, and specialized hardware that can be accessed remotely within the distributed network.

Resource Sharing Mechanisms

Resource sharing in distributed systems is facilitated through various mechanisms designed to optimize utilization, enhance collaboration, and ensure efficiency. Some common mechanisms include:



Resource Sharing Mechanisms

1. **Client-Server Architecture**: A classic model where clients request services or resources from centralized servers. This architecture centralizes resources and services, providing efficient access but potentially leading to scalability and reliability challenges.
2. **Peer-to-Peer (P2P) Networks**: Distributed networks where each node can act as both a client and a server. P2P networks facilitate direct resource sharing between nodes without reliance on centralized servers, promoting decentralized and scalable resource access.
3. **Distributed File Systems**: Storage systems that distribute files across multiple nodes, ensuring redundancy and fault tolerance while allowing efficient access to shared data.
4. **Load Balancing**: Mechanisms that distribute workload across multiple nodes to optimize resource usage and prevent overload on individual nodes, thereby improving performance and scalability.
5. **Virtualization**: Techniques such as virtual machines (VMs) and containers that abstract physical resources, enabling efficient resource allocation and utilization across distributed environments.
6. **Caching**: Storing frequently accessed data closer to users or applications to reduce latency and improve responsiveness, enhancing overall system performance.
7. **Replication**: Creating copies of data or resources across multiple nodes to ensure data availability, fault tolerance, and improved access speed.

Best Architectures for Resource Sharing in Distributed System

The best architectures for resource sharing in distributed systems depend on the specific requirements and characteristics of the system. Here are some commonly adopted architectures that facilitate efficient resource sharing:

1. **Client-Server Architecture**:
   - **Advantages**: Centralized management simplifies resource allocation and access control. It is suitable for applications where clients primarily consume services or resources from centralized servers.
   - **Use Cases**: Web applications, databases, and enterprise systems where centralized control and management are critical.
2. **Peer-to-Peer (P2P) Architecture**:
   - **Advantages**: Decentralized nature facilitates direct resource sharing between peers without dependency on centralized servers, enhancing scalability and fault tolerance.
   - **Use Cases**: File sharing, content distribution networks (CDNs), and collaborative computing environments.
3. **Service-Oriented Architecture (SOA)**:
   - **Advantages**: Organizes services as reusable components that can be accessed and shared across distributed systems, promoting interoperability and flexibility.
   - **Use Cases**: Enterprise applications, where modular services such as authentication, messaging, and data access are shared across different departments or systems.
4. **Microservices Architecture**:
   - **Advantages**: Decomposes applications into small, independent services that can be developed, deployed, and scaled independently. Each microservice can share resources selectively, optimizing resource usage.
   - **Use Cases**: Cloud-native applications, where scalability, agility, and resilience are paramount.
5. **Distributed File System Architecture**:
   - **Advantages**: Distributes file storage across multiple nodes, providing redundancy, fault tolerance, and efficient access to shared data.
   - **Use Cases**: Large-scale data storage and retrieval systems, such as Hadoop Distributed File System (HDFS) for big data processing.
6. **Container Orchestration Architectures (e.g., Kubernetes)**:
   - **Advantages**: Orchestrates containers across a cluster of nodes, facilitating efficient resource utilization and management of applications in distributed environments.
   - **Use Cases**: Cloud-native applications, where scalability, portability, and resource efficiency are critical.

Choosing the best architecture involves considering factors such as scalability requirements, fault tolerance, performance goals, and the nature of applications and services being deployed.

Resource Allocation Strategies in Distributed System

Resource allocation strategies in distributed systems are crucial for optimizing performance, ensuring fairness, and maximizing resource utilization. Here are some common strategies:

1. Static Allocation:

- **Description**: Resources are allocated based on fixed, predetermined criteria without considering dynamic workload changes.

- **Advantages**: Simple to implement and manage, suitable for predictable workloads.
- **Challenges**: Inefficient when workload varies or when resources are underutilized during low-demand periods.

2. Dynamic Allocation:

- **Description**: Resources are allocated based on real-time demand and workload conditions.
- **Advantages**: Maximizes resource utilization by adjusting allocations dynamically, responding to varying workload patterns.
- **Challenges**: Requires sophisticated monitoring and management mechanisms to handle dynamic changes effectively.

3. Load Balancing:

- **Description**: Distributes workload evenly across multiple nodes or resources to optimize performance and prevent overload.
- **Strategies**: Round-robin scheduling, least connection method, and weighted distribution based on resource capacities.
- **Advantages**: Improves system responsiveness and scalability by preventing bottlenecks.
- **Challenges**: Overhead of monitoring and adjusting workload distribution.

4. Reservation-Based Allocation:

- **Description**: Resources are reserved in advance based on anticipated future demand or specific application requirements.
- **Advantages**: Guarantees resource availability when needed, ensuring predictable performance.
- **Challenges**: Potential resource underutilization if reservations are not fully utilized.

5. Priority-Based Allocation:

- **Description**: Assigns priorities to different users or applications, allowing higher-priority tasks to access resources before lower-priority tasks.
- **Advantages**: Ensures critical tasks are completed promptly, maintaining service-level agreements (SLAs).
- **Challenges**: Requires fair prioritization policies to avoid starvation of lower-priority tasks.

Challenges in Resource Sharing in Distributed System

Resource sharing in distributed systems presents several challenges that need to be addressed to ensure efficient operation and optimal performance:

- **Consistency and Coherency**: Ensuring that shared resources such as data or files remain consistent across distributed nodes despite concurrent accesses and updates.
- **Concurrency Control**: Managing simultaneous access and updates to shared resources to prevent conflicts and maintain data integrity.
- **Fault Tolerance**: Ensuring resource availability and continuity of service in the event of node failures or network partitions.
- **Scalability**: Efficiently managing and scaling resources to accommodate increasing demands without compromising performance.
- **Load Balancing**: Distributing workload and resource usage evenly across distributed nodes to prevent bottlenecks and optimize resource utilization.
- **Security and Privacy**: Safeguarding shared resources against unauthorized access, data breaches, and ensuring privacy compliance.
- **Communication Overhead**: Minimizing overhead and latency associated with communication between distributed nodes accessing shared resources.

- **Synchronization**: Coordinating activities and maintaining synchronization between distributed nodes to ensure consistent and coherent resource access.

DFS (Distributed File System)

A **Distributed File System (DFS)** is a file system that is distributed on multiple file servers or multiple locations. It allows programs to access or store isolated files as they do with the local ones, allowing programmers to access files from any network or computer. In this article, we will discuss everything about Distributed File System.

DFS (Distributed File System) is a technology that allows you to group shared folders located on different servers into one or more logically structured namespaces. The main purpose of the Distributed File System (DFS) is to allows users of physically distributed systems to share their data and resources by using a Common File System.

Components of DFS

- **Location Transparency:** Location Transparency achieves through the namespace component.
- **Redundancy:** Redundancy is done through a file replication component.

In the case of failure and heavy load, these components together improve data availability by allowing the sharing of data in different locations to be logically grouped under one folder, which is known as the "DFS root". It is not necessary to use both the two components of DFS together, it is possible to use the namespace component without using the file replication component and it is perfectly possible to use the file replication component without using the namespace component between servers.

Features of DFS

- **Transparency**
    - **Structure transparency:** There is no need for the client to know about the number or locations of file servers and the storage devices. Multiple file servers should be provided for performance, adaptability, and dependability.
    - **Access transparency:** Both local and remote files should be accessible in the same manner. The file system should be automatically located on the accessed file and send it to the client's side.
    - **Naming transparency:** There should not be any hint in the name of the file to the location of the file. Once a name is given to the file, it should not be changed during transferring from one node to another.
    - **Replication transparency:** If a file is copied on multiple nodes, both the copies of the file and their locations should be hidden from one node to another.
- **User mobility:** It will automatically bring the user's home directory to the node where the user logs in.

- **Performance:** Performance is based on the average amount of time needed to convince the client requests. This time covers the CPU time + time taken to access secondary storage + network access time. It is advisable that the performance of the Distributed File System be similar to that of a centralized file system.
- **Simplicity and ease of use:** The user interface of a file system should be simple and the number of commands in the file should be small.
- **High availability:** A Distributed File System should be able to continue in case of any partial failures like a link failure, a node failure, or a storage drive crash. A high authentic and adaptable distributed file system should have different and independent file servers for controlling different and independent storage devices.
- **Scalability:** Since growing the network by adding new machines or joining two networks together is routine, the distributed system will inevitably grow over time. As a result, a good distributed file system should be built to scale quickly as the number of nodes and users in the system grows. Service should not be substantially disrupted as the number of nodes and users grows.
- **Data integrity:** Multiple users frequently share a file system. The integrity of data saved in a shared file must be guaranteed by the file system. That is, concurrent access requests from many users who are competing for access to the same file must be correctly synchronized using a concurrency control method. Atomic transactions are a high-level concurrency management mechanism for data integrity that is frequently offered to users by a file system.
- **Security:** A distributed file system should be secure so that its users may trust that their data will be kept private. To safeguard the information contained in the file system from unwanted & unauthorized access, security mechanisms must be implemented.

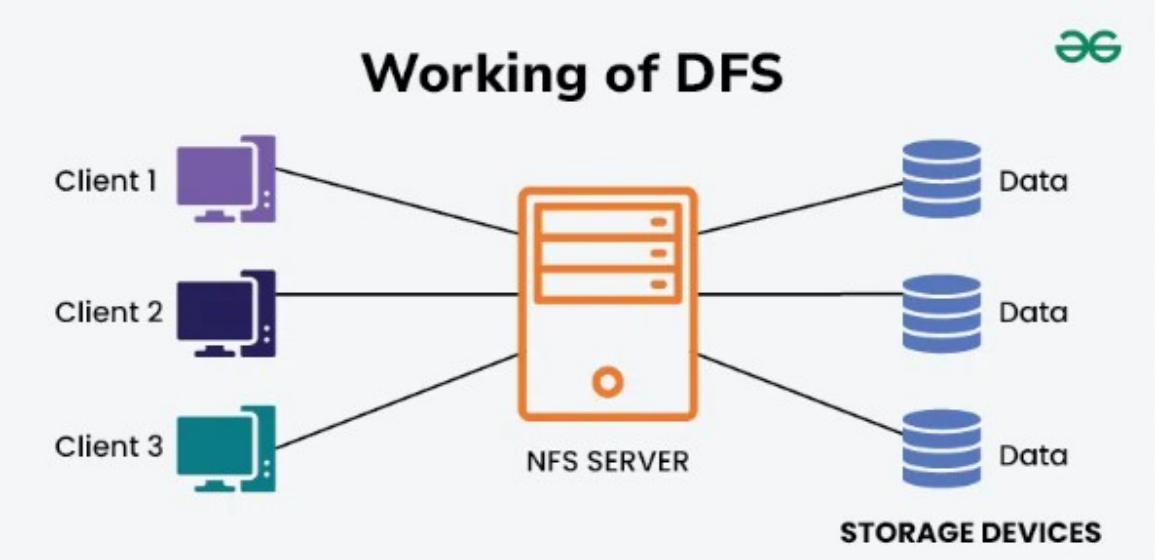
Applications of DFS

- **NFS:** NFS stands for Network File System. It is a client-server architecture that allows a computer user to view, store, and update files remotely. The protocol of NFS is one of the several distributed file system standards for Network-Attached Storage (NAS).
- **CIFS:** CIFS stands for Common Internet File System. CIFS is an accent of SMB. That is, CIFS is an application of SIMB protocol, designed by Microsoft.
- **SMB:** SMB stands for Server Message Block. It is a protocol for sharing a file and was invented by IBM. The SMB protocol was created to allow computers to perform read and write operations on files to a remote host over a Local Area Network (LAN). The directories present in the remote host can be accessed via SMB and are called as "shares".
- **Hadoop:** Hadoop is a group of open-source software services. It gives a software framework for distributed storage and operating of big data using the MapReduce programming model. The core of Hadoop contains a storage part, known as Hadoop Distributed File System (HDFS), and an operating part which is a MapReduce programming model.
- **NetWare:** NetWare is an abandon computer network operating system developed by Novell, Inc. It primarily used combined multitasking to run different services on a personal computer, using the IPX network protocol.

Working of DFS

There are two ways in which DFS can be implemented:

- **Standalone DFS namespace:** It allows only for those DFS roots that exist on the local computer and are not using Active Directory. A Standalone DFS can only be acquired on those computers on which it is created. It does not provide any fault liberation and cannot be linked to any other DFS. Standalone DFS roots are rarely come across because of their limited advantage.
- **Domain-based DFS namespace:** It stores the configuration of DFS in Active Directory, creating the DFS namespace root accessible at **<domainname>****<dfsroot>** or **<FQDN>****<dfsroot>**



Advantages of Distributed File System(DFS)

- DFS allows multiple user to access or store the data.
- It allows the data to be share remotely.
- It improved the availability of file, access time, and network efficiency.
- Improved the capacity to change the size of the data and also improves the ability to exchange the data.
- Distributed File System provides transparency of data even if server or disk fails.

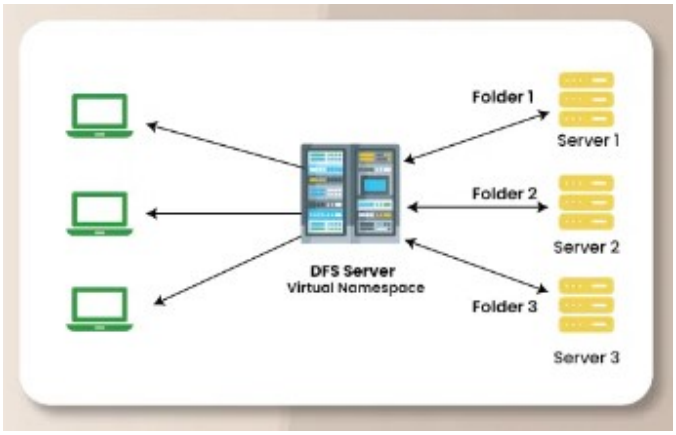
Disadvantages of Distributed File System(DFS)

- In Distributed File System nodes and connections needs to be secured therefore we can say that security is at stake.
- There is a possibility of lose of messages and data in the network while movement from one node to another.
- Database connection in case of Distributed File System is complicated.

- Also handling of the database is not easy in Distributed File System as compared to a single user system.
- There are chances that overloading will take place if all nodes tries to send data at once.

Mechanism for Building Distributed File System

Building a Distributed File System (DFS) involves intricate mechanisms to manage data across multiple networked nodes. This article explores key strategies for designing scalable, fault-tolerant systems that optimize performance and ensure data integrity in distributed computing environments.



Key Components of Distributed File System (DFS)

Below are the key components of Distributed File System:

- **Metadata Management**:
  - **Purpose**: Metadata includes information about files, such as file names, locations, permissions, and other attributes. Managing metadata efficiently is crucial for locating and accessing files across distributed nodes.
  - **Components**: Metadata servers or services maintain metadata consistency and provide lookup services to map file names to their physical locations.
- **Data Distribution and Replication**:
  - **Purpose**: Distributing data across multiple nodes improves performance and fault tolerance. Replication ensures data availability even if some nodes fail.
  - **Components**: Replication protocols define how data copies are synchronized and maintained across distributed nodes. Strategies include primary-backup replication, quorum-based replication, or consistency models like eventual consistency or strong consistency.
- **Consistency Models**:
  - **Purpose**: Ensuring data consistency across distributed nodes is challenging due to network delays and potential conflicts. Consistency models define how updates to data are propagated and synchronized to maintain a coherent view.

- **Components**: Models range from strong consistency (where all nodes see the same data simultaneously) to eventual consistency (where updates eventually propagate to all nodes).

Mechanism to build Distributed File Systems

Building a Distributed File System (DFS) involves implementing various mechanisms and architectures to ensure scalability, fault tolerance, performance optimization, and security. Let's see the mechanisms typically used in constructing a DFS:

1. Centralized File System Architecture

In a centralized DFS architecture, all file operations and management are handled by a single centralized server. Clients access files through this central server, which manages metadata and data storage.

1.1. Key Components and Characteristics:

- **Central Server**: Manages file metadata, including file names, permissions, and locations.
- **Client-Server Communication**: Clients communicate with the central server to perform read, write, and metadata operations.
- **Simplicity**: Easier to implement and manage compared to distributed architectures.
- **Scalability Limitations**: Limited scalability due to the centralized nature, as the server can become a bottleneck for file operations.
- **Examples**: Network File System (NFS) is a classic example of a centralized DFS where a central NFS server manages file access for multiple clients in a network.

1.2. Advantages:

- **Centralized Control**: Simplifies management and administration of file operations.
- **Consistency**: Ensures strong consistency since all operations go through the central server.
- **Security**: Centralized security mechanisms can be implemented effectively.

1.3. Disadvantages:

- **Scalability**: Limited scalability as all operations are bottlenecked by the central server's capacity.
- **Single Point of Failure**: The central server becomes a single point of failure, impacting system reliability.
- **Performance**: Potential performance bottleneck due to increased network traffic and server load.

2. Distributed File System Architecture

Distributed File Systems distribute files and their metadata across multiple nodes in a network. This approach allows for parallel access and improves scalability and fault tolerance.

2.1. Key Components and Characteristics:

- **Distributed Storage**: Files are divided into smaller units (blocks) and distributed across multiple nodes or servers in the network.
- **Metadata Handling**: Metadata is distributed or managed by dedicated metadata servers or distributed across storage nodes.
- **Data Replication**: Copies of data blocks may be replicated across nodes to ensure fault tolerance and data availability.
- **Load Balancing**: Techniques are employed to distribute read and write operations evenly across distributed nodes.
- **Examples**: Google File System (GFS), Hadoop Distributed File System (HDFS), and Amazon S3 (Simple Storage Service) are prominent examples of distributed file systems used in cloud computing and big data applications.

2.2. Advantages:

- **Scalability**: Easily scales by adding more storage nodes without centralized bottlenecks.
- **Fault Tolerance**: Redundancy and data replication ensure high availability and reliability.
- **Performance**: Parallel access to distributed data blocks improves read/write performance.
- **Load Balancing**: Efficient load distribution across nodes enhances overall system performance.

2.3. Disadvantages:

- **Complexity**: Increased complexity in managing distributed data and ensuring consistency across nodes.
- **Consistency Challenges**: Consistency models must be carefully designed to manage concurrent updates and maintain data integrity.
- **Security**: Ensuring data security and access control across distributed nodes can be more challenging than in centralized systems.

3. Hybrid File System Architectures

Hybrid architectures combine elements of both centralized and distributed approaches, offering flexibility and optimization based on specific use cases.

3.1. Key Components and Characteristics:

- **Combination of Models**: Utilizes centralized servers for managing metadata and control, while data storage and access may be distributed across nodes.
- **Client-Side Caching**: Clients may cache frequently accessed data locally to reduce latency and improve performance.
- **Examples**: Andrew File System (AFS) combines centralized metadata management with distributed data storage for efficient file access and management.

3.2. Advantages:

- **Flexibility**: Adaptable to different workload demands and scalability requirements.
- **Performance Optimization**: Combines centralized control with distributed data access for optimized performance.
- **Fault Tolerance**: Can leverage data replication strategies while maintaining centralized metadata management.

3.3. Disadvantages:

- **Complexity**: Hybrid architectures can introduce additional complexity in system design and management.
- **Consistency**: Ensuring consistency between centralized metadata and distributed data can be challenging.
- **Integration Challenges**: Integration of centralized and distributed components requires careful coordination and synchronization.

4. Implementation Considerations

- **Scalability**: Choose architecture based on scalability requirements, considering potential growth in data volume and user access.
- **Fault Tolerance**: Implement redundancy and data replication strategies to ensure high availability and reliability.
- **Performance**: Optimize data access patterns, employ caching mechanisms, and utilize load balancing techniques to enhance performance.
- **Security**: Implement robust security measures, including encryption, access controls, and authentication mechanisms, to protect data across distributed nodes.

Challenges and Solutions in building Distributed File Systems

Below are the challenges and solution in building distributed file systems:

- **Data Consistency and Integrity**
  - **Challenge**: Ensuring consistent data across distributed nodes due to network delays and concurrent updates.
  - **Solution**: Implement appropriate consistency models (e.g., strong or eventual consistency) and concurrency control mechanisms (e.g., distributed locking, transactions).
- **Fault Tolerance and Reliability**
  - **Challenge**: Dealing with node failures, network partitions, and hardware/software issues.
  - **Solution**: Use data replication strategies (e.g., primary-backup, quorum-based), automated failure detection, and recovery mechanisms to ensure data availability and system reliability.
- **Scalability and Performance**
  - **Challenge**: Scaling to handle large data volumes and increasing user access without performance degradation.
  - **Solution**: Scale horizontally by adding more nodes, employ load balancing techniques, and optimize data partitioning and distribution to enhance system performance.
- **Security and Access Control**
  - **Challenge**: Protecting data integrity, confidentiality, and enforcing access control across distributed nodes.
  - **Solution**: Utilize encryption for data at rest and in transit, implement robust authentication and authorization mechanisms, and monitor access logs for security compliance.

Sun's Network File System (NFS)

One of the first uses of distributed client/server computing was in the realm of distributed file systems. In such an environment, there are a number of client machines and one server (or a few); the server stores the data on its disks, and clients request data through well-formed protocol messages. Figure 49.1 depicts the basic setup.

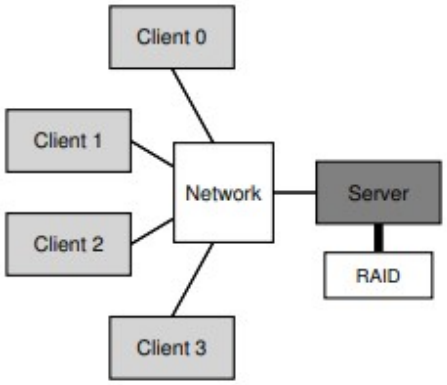
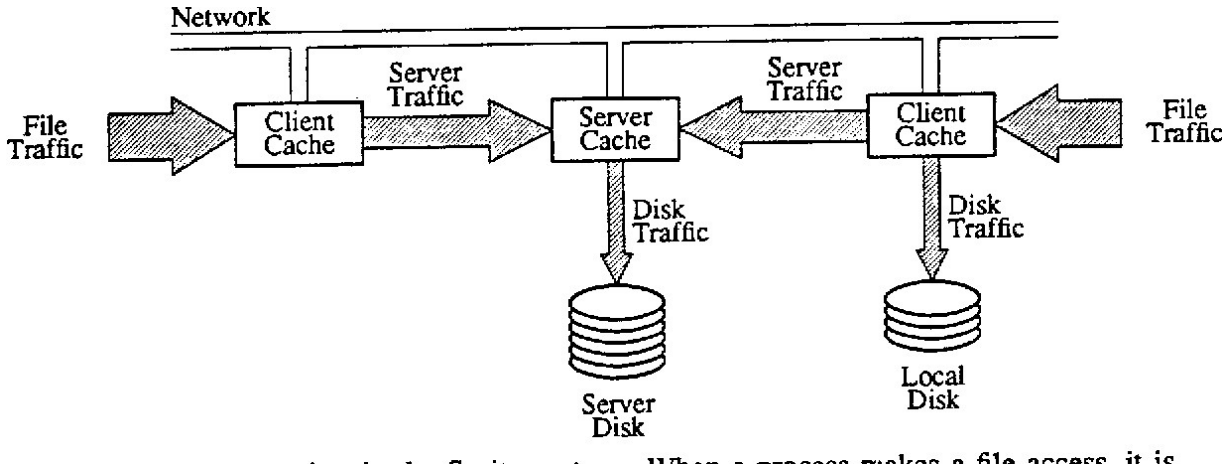


Figure 49.1: **A Generic Client/Server System**

As you can see from the picture, the server has the disks, and clients send messages across a network to access their directories and files on those disks. Why do we bother with this arrangement? (i.e., why don't we just let clients use their local disks?) Well, primarily this setup allows for easy sharing of data across clients. Thus, if you access a file on one machine (Client 0) and then later use another (Client 2), you will have the same view of the file system. Your data is naturally shared across these different machines. A secondary benefit is centralized administration; for example, backing up files can be done from the few server machines instead of from the multitude of clients. Another advantage could be security; having all servers in a locked machine room prevents certain types of problems from arising.

Sprite file system

A Sprite File System typically refers to a way of organizing and managing images or graphics that are used in a 2D game or application. These images, or "sprites," represent characters, objects, or other visual elements that appear on screen. The system allows for efficient storage, retrieval, and manipulation of these sprites.



There are generally two main components:

1. Sprite Sheets

- A sprite sheet is a large image that contains multiple smaller images (sprites) laid out in a grid. By grouping them in this way, it reduces the number of image files, which in turn optimizes the process of loading and rendering graphics, as only one file needs to be loaded instead of many individual images.
- Sprite sheets are especially useful in animations, where consecutive frames of an animation are stored in a single image file.

2. Sprite File System (Sprite Management)

- This refers to the software or code used to manage the individual sprites within a sprite sheet or group of sprite sheets. The system can be responsible for loading sprite sheets, extracting specific sprites, and organizing the sprites for rendering.
- This system can also handle operations like flipping, rotating, and scaling sprites, or handling collision detection and alignment.

Benefits of Using a Sprite File System

- **Performance Optimization:** Reduces the number of file I/O operations by consolidating multiple images into a single file.
- **Memory Efficiency:** Manages how sprites are loaded and unloaded, which can minimize memory usage.
- **Simplified Rendering:** Allows for easier batching of draw calls since multiple sprites can be rendered with fewer resources.
- **Easy Animation Handling:** For frame-by-frame animations, sprite sheets make it easier to quickly switch between frames.

Log-Structured File System (LFS)

Log-Structured File Systems were introduced by Rosenblum and Ousterhout in the early 90's to address the following issues.

- **Growing system memories:** With growing disk sizes, the amount of data that can be cached also increases. Since reads are serviced by the cache, the file system performance begins to depend solely on its write performance.
- **Sequential I/O performance trumps over random I/O performance:** Over the years, the bandwidth of accessing bits off the hard drive has increased because more bits can be

accommodated over the same area. However, it is physically difficult for the small rotors to move the disk more quickly. Therefore, sequential access can improve disk performance significantly.

- **Inefficiency of existing file systems:** Existing file systems perform a large number of writes for as much as creating a new file, including inode, bitmap and data block writes and subsequent updates. The short seeks and rotational delays incurred reduces bandwidth.
- **File systems are not RAID-aware:** Further, file systems do not have any mechanism to counter the small-write problem in RAID-4 and RAID-5.

Even though processor speeds and main memory sizes have increased at an exponential rate, disk access costs have evolved much more slowly. This calls for a file system which focusses on write performance, makes use of the sequential bandwidth, and works efficiently on both disk writes as well as metadata updates. This is where the motivation is Log-Structured File System (LFS) is rooted. While all reads are impossible to be carried out sequentially (since any file may be accessed at any point of time), we can exploit the efficiency of sequential writes. LFS keeps a small buffer of all writes in a memory **segment**. A log is simply a data structure which is written only at the head (one could think of the entire disk as a log). Once the log is full, it is written into an unused part of the disk in a sequential manner. New data and metadata (inodes, directories) are accumulated into the buffer cache and written all at once in large blocks (such as segments of 0.5M or 1M). The following are the data structures used in the LFS implementation.

- **Inodes:** As in Unix, inodes contain physical block pointers to files.
- **Inode Map:** This table indicates the location of each inode on the disk. The inode map is written in the segment itself.
- **Segment Summary:** This maintains information about each block in the segment.
- **Segment Usage Table:** This tells us the amount of data on a block.

Disk Management

Disk management is one of the critical operations carried out by the operating system. It deals with organizing the data stored on the secondary storage devices which includes the hard disk drives and the solid-state drives. It also carries out the function of optimizing the data and making sure that the data is safe by implementing various disk management techniques. We will learn more about disk management and its related techniques found in operating system.

The four main operating system management functions (each of which are dealt with in more detail in different places) are:

- Process Management
- Memory Management
- File and Disk Management
- I/O System Management

Most computer systems employ secondary storage devices (magnetic disks). It provides low-cost, non-volatile storage for programs and data (tape, optical media, flash drives, etc.). Programs and the user data they use are kept on separate storage devices called files. The operating system is responsible for allocating space for files on secondary storage media as needed.

Disk Management of the Operating System Includes:

- Disk Format
- Booting from disk
- Bad block recovery

Some common disk management techniques used in operating systems include:

1. Partitioning: This involves dividing a single physical disk into multiple logical partitions. Each partition can be treated as a separate storage device, allowing for better organization and management of data.
2. Formatting: This involves preparing a disk for use by creating a file system on it. This process typically erases all existing data on the disk.
3. File system management: This involves managing the file systems used by the operating system to store and access data on the disk. Different file systems have different features and performance characteristics.
4. Disk space allocation: This involves allocating space on the disk for storing files and directories. Some common methods of allocation include contiguous allocation, linked allocation, and indexed allocation.
5. Disk defragmentation: Over time, as files are created and deleted, the data on a disk can become fragmented, meaning that it is scattered across the disk. Disk defragmentation involves rearranging the data on the disk to improve performance.

Advantages of disk management include:

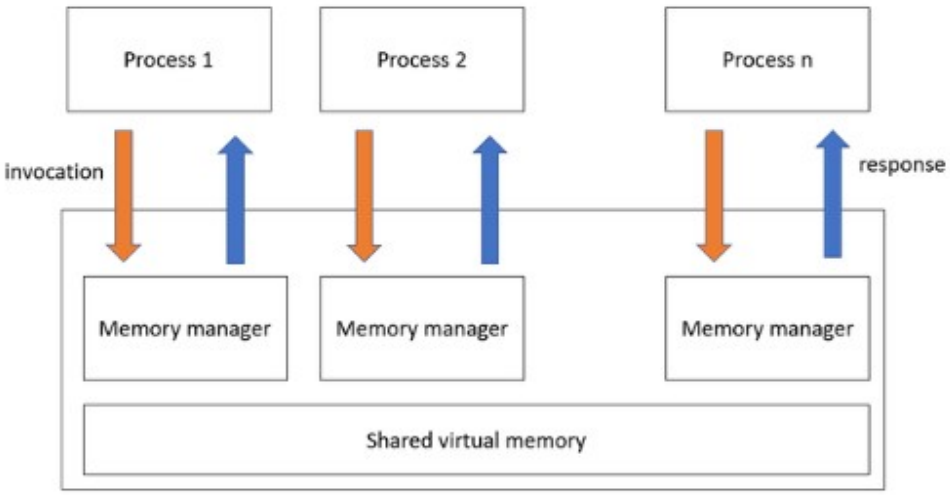
1. Improved organization and management of data.
2. Efficient use of available storage space.
3. Improved data integrity and security.
4. Improved performance through techniques such as defragmentation.

Disadvantages of disk management include:

1. Increased system overhead due to disk management tasks.
2. Increased complexity in managing multiple partitions and file systems.
3. Increased risk of data loss due to errors during disk management tasks.
4. Overall, disk management is an essential aspect of operating system management and can greatly improve system performance and data integrity when implemented properly.

Distributed Shared Memory

It is a mechanism that manages memory across multiple nodes and makes inter-process communications transparent to end-users. The applications will think that they are running on shared memory. DSM is a mechanism of allowing user processes to access shared data without using inter-process communications. In DSM every node has its own memory and provides memory read and write services and it provides consistency protocols. The distributed shared memory (DSM) implements the shared memory model in distributed systems but it doesn't have physical shared memory. All the nodes share the virtual address space provided by the shared memory model. The Data moves between the main memories of different nodes.



Types of Distributed Shared Memory

1. On-Chip Memory

- The data is present in the CPU portion of the chip.
- Memory is directly connected to address lines.
- On-Chip Memory DSM is expensive and complex.

2. Bus-Based Multiprocessors

- A set of parallel wires called a bus acts as a connection between CPU and memory.
- Accessing of same memory simultaneously by multiple CPUs is prevented by using some algorithms.
- Cache memory is used to reduce network traffic.

3. Ring-Based Multiprocessors

- There is no global centralized memory present in Ring-based DSM.
- All nodes are connected via a token passing ring.
- In ring-bases DSM a single address line is divided into the shared area.

Advantages of Distributed Shared Memory

- **Simpler Abstraction:** Programmer need not concern about data movement, as the address space is the same it is easier to implement than RPC.
- **Easier Portability:** The access protocols used in DSM allow for a natural transition from sequential to distributed systems. DSM programs are portable as they use a common programming interface.
- **Locality of Data:** Data moved in large blocks i.e. data near to the current memory location that is being fetched, may be needed future so it will be also fetched.
- **On-Demand Data Movement:** It provided by DSM will eliminate the data exchange phase.
- **Larger Memory Space:** It provides large virtual memory space, the total memory size is the sum of the memory size of all the nodes, paging activities are reduced.
- **Better Performance:** DSM improve performance and efficiency by speeding up access to data.
- **Flexible Communication Environment:** They can join and leave DSM system without affecting the others as there is no need for sender and receiver to existing,

- **Process Migration Simplified:** They all share the address space so one process can easily be moved to a different machine.

Disadvantages of Distributed Shared Memory

- **Accessibility:** The data access is slow in DSM as compare to non-distributed.
- **Consistency**: When programming is done in DSM systems, programmers need to maintain consistency.
- **Message Passing:** DSM use asynchronous message passing and is not efficient as per other message passing implementation.
- **Data Redundancy:** DSM allows simultaneous access to data, consistency and data redundancy is common disadvantage.
- **Lower Performance:** CPU gets slowed down, even cache memory does not aid the situation.

Algorithm for implementing Distributed Shared Memory

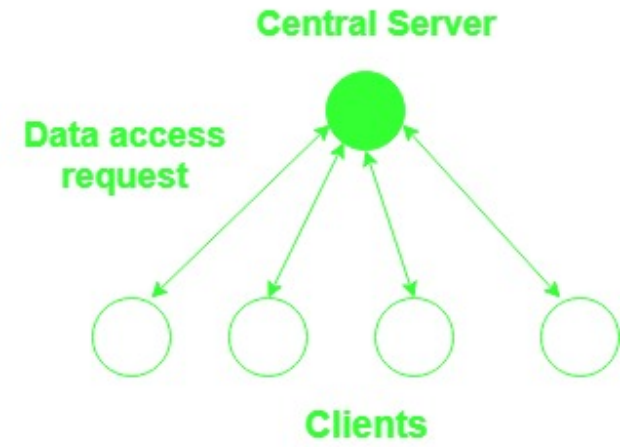
Distributed shared memory(DSM) system is a resource management component of distributed operating system that implements shared memory model in distributed system which have no physically shared memory. The shared memory model provides a virtual address space which is shared by all nodes in a distributed system.
The central issues in implementing DSM are:

- how to keep track of location of remote data.
- how to overcome communication overheads and delays involved in execution of communication protocols in system for accessing remote data.
- how to make shared data concurrently accessible at several nodes to improve performance.

Algorithms to implement DSM

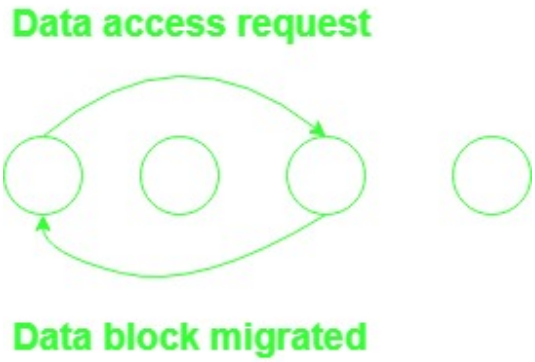
1. Central Server Algorithm:

- In this, a central server maintains all shared data. It services read requests from other nodes by returning the data items to them and write requests by updating the data and returning acknowledgement messages.
- Time-out can be used in case of failed acknowledgement while sequence number can be used to avoid duplicate write requests.
- It is simpler to implement but the central server can become bottleneck and to overcome this shared data can be distributed among several servers. This distribution can be by address or by using a mapping function to locate the appropriate server.



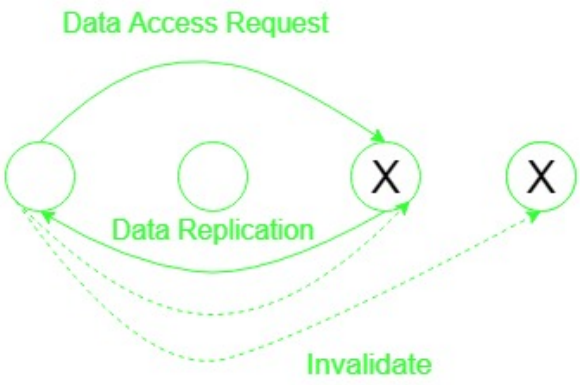
2. Migration Algorithm:

- In contrast to central server algo where every data access request is forwarded to location of data while in this data is shipped to location of data access request which allows subsequent access to be performed locally.
- It allows only one node to access a shared data at a time and the whole block containing data item migrates instead of individual item requested.
- It is susceptible to thrashing where pages frequently migrate between nodes while servicing only a few requests.
- This algo provides an opportunity to integrate DSM with virtual memory provided by operating system at individual nodes.



3. Read Replication Algorithm:

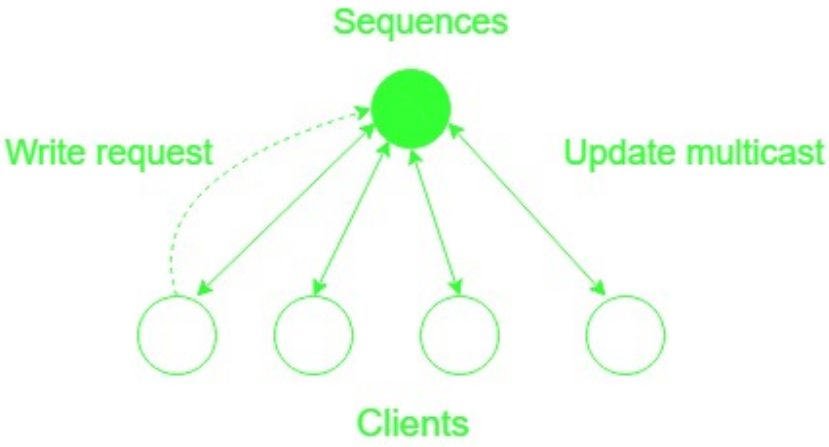
- This extends the migration algorithm by replicating data blocks and allowing multiple nodes to have read access or one node to have both read write access.
- It improves system performance by allowing multiple nodes to access data concurrently.
- The write operation in this is expensive as all copies of a shared block at various nodes will either have to invalidated or updated with the current value to maintain consistency of shared data block.
- DSM must keep track of location of all copies of data blocks in this.



Write operation in Read Replication algorithm

4. Full Replication Algorithm:

- It is an extension of read replication algorithm which allows multiple nodes to have both read and write access to shared data blocks.
- Since many nodes can write shared data concurrently, the access to shared data must be controlled to maintain it's consistency.
- To maintain consistency, it can use a gap free sequences in which all nodes wishing to modify shared data will send the modification to sequencer which will then assign a sequence number and multicast the modification with sequence number to all nodes that have a copy of shared data item.



Write operation in Full Replication algorithm

Memory Coherence

In distributed systems, memory coherence refers to the consistency of shared data when multiple processors or nodes have access to a shared memory space. It ensures that when one processor updates a value in shared memory, other processors that access the same memory will see the most recent update. This is critical for preventing inconsistencies and ensuring that the system functions correctly as a whole.

Memory coherence is closely related to the concept of cache coherence because, in distributed systems, each processor typically has its own cache to speed up data access. Without proper coherence mechanisms, processors could end up working with stale or outdated data, which would lead to incorrect computations or system states.

There are several key concepts related to memory coherence in distributed systems:

1**. Coherence Protocols**
Coherence protocols define rules for how changes to shared data are propagated throughout the system to maintain consistency. Common protocols include:

- **Write-invalidate protocol:** When a processor writes to a shared memory location, all other processors with cached copies of that memory are notified, and their copies are invalidated.
- **Write-update protocol:** When a processor writes to shared memory, it propagates the update to other caches so that every cache has the most recent version.

2. Sequential Consistency
Sequential consistency ensures that the results of execution in a distributed system appear as if the operations were executed in some sequential order. This means all nodes in the system see the same sequence of memory updates, though this can come at a performance cost.

3. Memory Consistency Models
Distributed systems may use different memory consistency models that define the rules for when updates made by one processor become visible to others. These include:

- **Strict consistency:** The most rigid model where any read of a memory location returns the most recent write, but it is difficult to implement efficiently in distributed systems.
- **Relaxed consistency models:** These models (such as eventual consistency, weak consistency, or causal consistency) allow for more flexibility and higher performance at the cost of temporarily inconsistent views of memory.

4. Distributed Shared Memory (DSM)
In some systems, distributed shared memory is used to provide an abstraction where physically separate memories (on different machines) appear as a single global memory

space. Memory coherence protocols ensure that the view of this shared memory remains consistent across all nodes.

5. **Challenges and Trade-offs**
Maintaining memory coherence in distributed systems involves trade-offs between performance and consistency. Stricter coherence protocols can slow down performance due to the overhead of maintaining a consistent state across all nodes, while relaxed protocols improve performance but may lead to temporary inconsistencies.

Example: Cache Coherence in Multiprocessors
In a multi-core or multi-processor system, each core may cache copies of memory blocks for faster access. Without coherence mechanisms, if one core updates a value in its cache, another core might still see the old value in its cache. Coherence protocols like **MESI (Modified, Exclusive, Shared, Invalid) are used to prevent this by ensuring that all caches reflect the most recent updates to shared data.

Difference between Cache Coherence and Memory Consistency

Sr. No.	Cache coherence	Memory consistency
1.	Cache Coherence describes the behavior of reads and writes to the same memory location.	Memory consistency describes the behavior of reads and writes in relation to other locations.
2.	Cache coherence required for cache-equipped systems.	Memory consistency required in systems that have or do not have caches.
3.	Coherence is the guarantee that caches will never affect the observable functionality of a program	Consistency is the specification of correctness for memory accesses, Defining guarantees on when loads and stores will happen and when they will be seen by the different cores
4.	It is concerned with the ordering of writes to a single memory location.	It handles the ordering of reads and writes to all memory locations
5.	A memory system is coherent if and only if – Can serialize all operations to that location – Read returns the value written to that location by the last store.	Consistency is a feature of a memory system if – It adheres to the rules of its Memory Model. – Memory operations are performed in a specific order.

Scheduling in Distributed Systems:

The techniques that are used for scheduling the processes in distributed systems are as follows:

1. Task Assignment Approach: In the Task Assignment Approach, the user-submitted process is composed of multiple related tasks which are scheduled to appropriate nodes in a system to improve the performance of a system as a whole.
2. Load Balancing Approach: In the Load Balancing Approach, as the name implies, the workload is balanced among the nodes of the system.
3. Load Sharing Approach: In the Load Sharing Approach, it is assured that no node would be idle while processes are waiting for their processing.

Characteristics of a Good Scheduling Algorithm:

The following are the required characteristics of a Good Scheduling Algorithm:

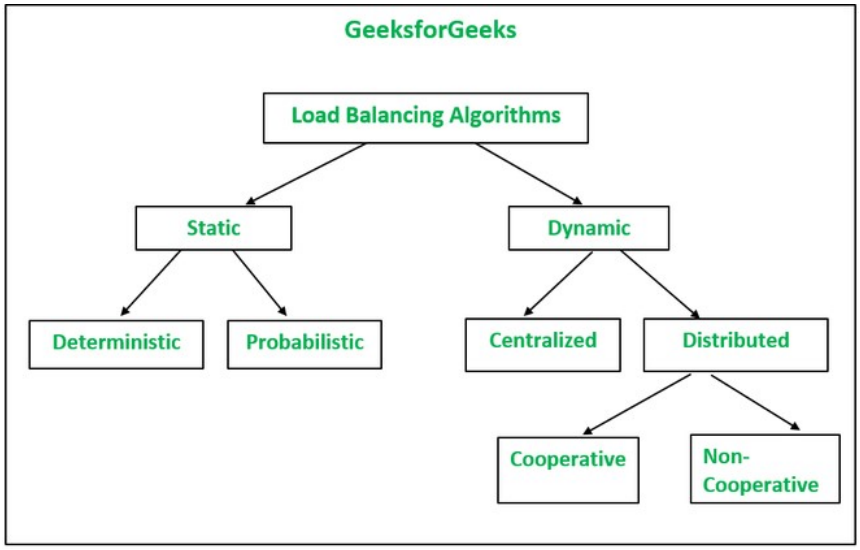
- The scheduling algorithms that require prior knowledge about the properties and resource requirements of a process submitted by a user put a burden on the user. Hence, a good scheduling algorithm does not require prior specification regarding the user-submitted process.
- A good scheduling algorithm must exhibit the dynamic scheduling of processes as the initial allocation of the process to a system might need to be changed with time to balance the load of the system.
- The algorithm must be flexible enough to process migration decisions when there is a change in the system load.
- The algorithm must possess stability so that processors can be utilized optimally. It is possible only when thrashing overhead gets minimized and there should no wastage of time in process migration.
- An algorithm with quick decision making is preferable such as heuristic methods that take less time due to less computational work give near-optimal results in comparison to an exhaustive search that provides an optimal solution but takes more time.
- A good scheduling algorithm gives balanced system performance by maintaining minimum global state information as global state information (CPU load) is directly proportional to overhead. So, with the increase in global state information overhead also increases.
- The algorithm should not be affected by the failure of one or more nodes of the system. Furthermore, even if the link fails and nodes of a group get separated into two or more groups then also it should not break down. So, the algorithm must possess decentralized decision-making capability in which consideration is given only to the available nodes for taking a decision and thus, providing fault tolerance.
- A good scheduling algorithm has the property of being scalable. It is flexible for scaling when the number of nodes increases in a system. If an algorithm opts for a strategy in which it inquires about the workload of all nodes and then selects the one with the least load then it is not considered a good approach because it leads to poor scalability as it will not work well for a system having many nodes. The reason is that the inquirer receives a lot many replies almost simultaneously and the processing time spent for reply messages is too long

for a node selection with the increase in several nodes (N). A straightforward way is to examine only m of N nodes.

- A good scheduling algorithm must be having fairness of service because in an attempt to balance the workload on all nodes of the system there might be a possibility that nodes with more load get more benefit as compared to nodes with less load because they suffer from poor response time than stand-alone systems. Hence, the solution lies in the concept of load sharing in which a node can share some of its resources until the user is not affected.

Load Balancing in Distributed Systems:

The Load Balancing approach refers to the division of load among the processing elements of a distributed system. The excess load of one processing element is distributed to other processing elements that have less load according to the defined limits. In other words, the load is maintained at each processing element in such a manner that neither it gets overloaded nor idle during the execution of a program to maximize the system throughput which is the ultimate goal of distributed systems. This approach makes all processing elements equally busy thus speeding up the entire task leads to the completion of the task by all processors approximately at the same time.



Types of Load Balancing Algorithms:

- **Static Load Balancing Algorithm:** In the Static Load Balancing Algorithm, while distributing load the current state of the system is not taken into account. These algorithms are simpler in comparison to dynamic load balancing algorithms. Types of Static Load Balancing Algorithms are as follows:
    - Deterministic: In Deterministic Algorithms, the properties of nodes and processes are taken into account for the allocation of processes to nodes. Because of the deterministic characteristic of the algorithm, it is difficult to optimize to give better results and also costs more to implement.

    - Probabilistic: n Probabilistic Algorithms, Statistical attributes of the system are taken into account such as several nodes, topology, etc. to make process placement rules. It does not give better performance.
- **Dynamic Load Balancing Algorithm:** Dynamic Load Balancing Algorithm takes into account the current load of each node or computing unit in the system, allowing for faster processing by dynamically redistributing workloads away from overloaded nodes and toward underloaded nodes. Dynamic algorithms are significantly more difficult to design, but they can give superior results, especially when execution durations for distinct jobs vary greatly. Furthermore, because dedicated nodes for task distribution are not required, a dynamic load balancing architecture is frequently more modular. Types of Dynamic Load Balancing Algorithms are as follows:
    - Centralized: In Centralized Load Balancing Algorithms, the task of handling requests for process scheduling is carried out by a centralized server node. The benefit of this approach is efficiency as all the information is held at a single node but it suffers from the reliability problem because of the lower fault tolerance. Moreover, there is another problem with the increasing number of requests.
    - Distributed: In Distributed Load Balancing Algorithms, the decision task of assigning processes is distributed physically to the individual nodes of the system. Unlike Centralized Load Balancing Algorithms, there is no need to hold state information. Hence, speed is fast.

Types of Distributed Load Balancing Algorithms:

- Cooperative In Cooperative Load Balancing Algorithms, as the name implies, scheduling decisions are taken with the cooperation of entities in the system. The benefit lies in the stability of this approach. The drawback is the complexity involved which leads to more overhead than Non-cooperative algorithms.
- Non-cooperative: In Non-cooperative Load Balancing Algorithms, scheduling decisions are taken by the individual entities of the system as they act as autonomous entities. The benefit is that minor overheads are involved due to the basic nature of non-cooperation. The drawback is that these algorithms might be less stable than Cooperative algorithms.

Issues in Designing Load-balancing Algorithms:

Many issues need to be taken into account while designing Load-balancing Algorithms:

- **Load Estimation Policies:** Determination of a load of a node in a distributed system.
- **Process Transfer Policies:** Decides for the execution of process: local or remote.
- **State Information Exchange:** Determination of strategy for exchanging system load information among the nodes in a distributed system.
- **Location Policy:** Determining the selection of destination nodes for the migration of the process.
- **Priority Assignment:** Determines whether the priority is given to a local or a remote process on a node for execution.
- **Migration limit policy:** Determines the limit value for the migration of processes.