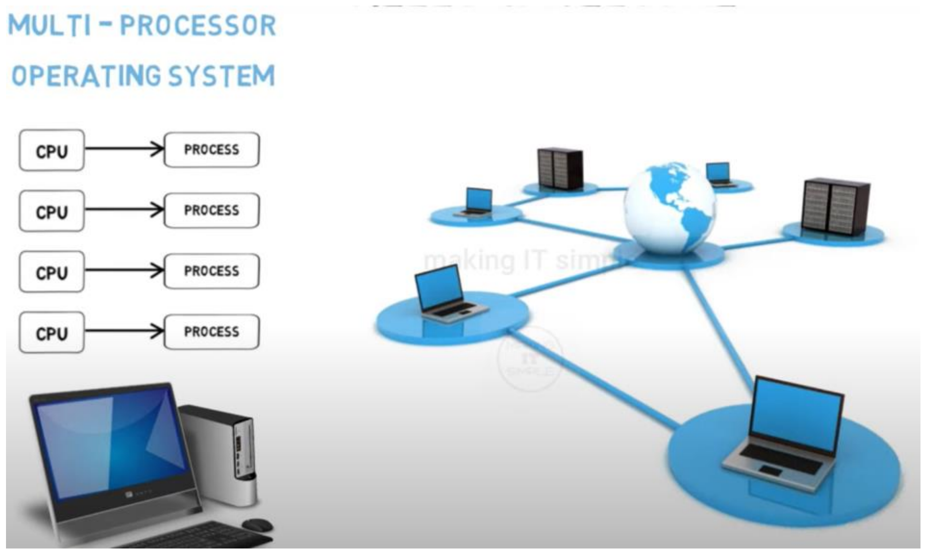
# L01
# Subject:
# Distributed Computing:

INTRODUCTION TO DISTRIBUTED SYSTEM

DEFINITION, GOALS, EXAMPLES OF DISTRIBUTED SYSTEM-INTERNET. SYSTEM ARCHITECTURES-CENTRALIZED ARCHITECTURE, DECENTRALIZED ARCHITECTURE, HYBRID ARCHITECTURE, CLIENT-SERVER MODEL, SERVERS-GENERAL DESIGN ISSUES, SERVER CLUSTERS, MANAGING SERVER CLUSTERS.

## What Is A Distributed System?



- A **Distributed System** is a collection of **independent** computers that appears to its users as a single coherent system.
- They help in sharing different **resources** and **capabilities** to provide users with a single and integrated coherent network.
- A *distributed system* is one in which **components** located at networked computers, communicate and **coordinate** their actions only by passing **messages**.
- Ideal: to present a single-system image:
  - The distributed system "looks like" a single computer rather than a collection of separate computers.

## What Is A Distributed System?

A collection of independent computers that appears to its users as a single coherent system.

Features:

- No shared memory – message-based communication
- Each runs its own local OS
- Heterogeneity

Ideal: to present a single-system image:

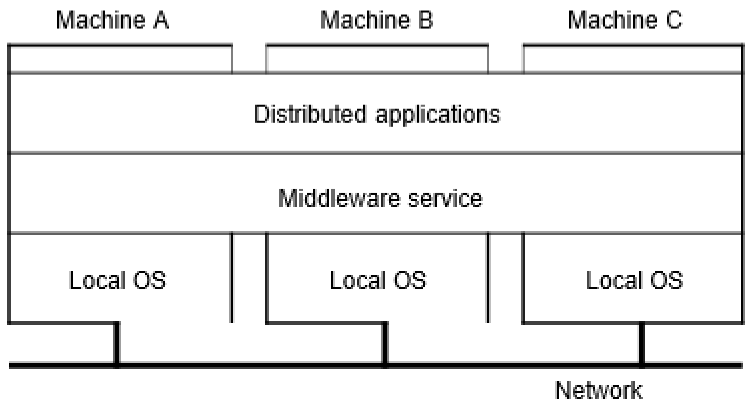
- The distributed system “looks like” a single computer rather than a collection of separate computers.

## Distributed System: Definition

A distributed system is a piece of software that ensures that:

*A collection of independent computers that appears to its users as a single coherent system*

Two aspects: (1) independent computers and (2) single system **middleware**.



## Examples of distributed systems...

- Intra-nets, Inter-net, WWW, email, …
- DNS (Domain Name System)
  - Hierarchical distributed database
- Distributed supercomputers, Grid/Cloud computing
- Electronic banking
- Airline reservation systems
- Peer-to-peer networks
- Sensor networks
- Mobile and Pervasive Computing

What is a real-time example of a distributed system?

Telephone and cellular networks are also examples of distributed networks. Telephone networks have been around for over a century and it started as an early example of a peer to peer network. Cellular networks are distributed networks with base stations physically distributed in areas called cells.

# Distributed System Characteristics

To present a single-system image:

- ❑Hide internal organization, communication details
- ❑Provide uniform interface

Easily expandable

- ❑Adding new computers is hidden from users

Continuous availability

- ❑Failures in one component can be covered by other components

Supported by middleware

## Distributed System Goals

- Resource Accessibility
- Distribution Transparency
- Openness
- Scalability

## Goals
## 1. Resource Availability

- Support user access to remote resources (printers, data files, web pages, CPU cycles) and the fair sharing of the resources
- Economics of sharing expensive resources
- Performance enhancement – due to multiple processors; also due to ease of collaboration and info exchange – access to remote services
  - Groupware: tools to support collaboration
- Resource sharing introduces security problems.

## Degree of Transparency

**Observation:** Aiming at full distribution transparency may be too much:

- Users may be located in different continents; distribution is apparent and not something you want to hide
- Completely hiding failures of networks and nodes is (theoretically and practically) impossible
  - You cannot distinguish a slow computer from a failing one
  - You can never be sure that a server actually performed an operation before a crash
- Full transparency will cost performance, exposing distribution of the system
  - Keeping Web caches *exactly* up-to-date with the master copy
  - Immediately flushing write operations to disk for fault tolerance

## Degree of Transparency

**Observation:** Aiming at full distribution transparency may be too much:

- Users may be located in different continents; distribution is apparent and not something you want to hide
- Completely hiding failures of networks and nodes is (theoretically and practically) impossible
  - You cannot distinguish a slow computer from a failing one
  - You can never be sure that a server actually performed an operation before a crash
- Full transparency will cost performance, exposing distribution of the system
  - Keeping Web caches *exactly* up-to-date with the master copy
  - Immediately flushing write operations to disk for fault tolerance

## Distribution Transparency

| Transparency | Description |
|---|---|
| Access | Hide differences in data representation & resource access (enables interoperability) |
| Location | Hide location of resource (can use resource without knowing its location) |
| Migration | Hide possibility that a system may change location of resource (no effect on access) |
| Replication | Hide the possibility that multiple copies of the resource exist (for reliability and/or availability) |
| Concurrency | Hide the possibility that the resource may be shared concurrently |
| Failure | Hide failure and recovery of the resource. How does one differentiate betw. slow and failed? |
| Relocation | Hide that resource may be moved <u>during use</u> |

## Openness of Distributed Systems

**Open distributed system:** Be able to interact with services from other open systems, irrespective of the underlying environment:

- Systems should conform to well-defined **interfaces**
- Systems should support **portability** of applications
- Systems should easily **interoperate**

**Achieving openness:** At least make the distributed system independent from **heterogeneity** of the underlying environment:

- Hardware
- Platforms
- Languages

## Policies versus Mechanisms

**Implementing openness:** Requires support for different **policies** specified by applications and users:

- What level of consistency do we require for client-cached data?
- Which operations do we allow downloaded code to perform?
- Which QoS requirements do we adjust in the face of varying bandwidth?
- What level of secrecy do we require for communication?

**Implementing openness:** Ideally, a distributed system provides only **mechanisms**:

- Allow (dynamic) setting of caching policies, preferably per cachable item
- Support different levels of trust for mobile code
- Provide adjustable QoS parameters per data stream
- Offer different encryption algorithms

## Scale in Distributed Systems

**Observation:** Many developers of modern distributed system easily use the adjective "scalable" without making clear *why* their system actually scales.

**Scalability:** At least three components:

- Number of users and/or processes **(size scalability)**
- Maximum distance between nodes **(geographical scalability)**
- Number of administrative domains **(administrative scalability)**

## Techniques for Scaling

**Distribution:** Partition data and computations across multiple machines:

- Move computations to clients (Java applets)
- Decentralized naming services (DNS)
- Decentralized information systems (WWW)

**Replication:** Make copies of data available at different machines:

- Replicated file servers (mainly for fault tolerance)
- Replicated databases
- Mirrored Web sites
- Large-scale distributed shared memory systems

**Caching:** Allow client processes to access local copies:

- Web caches (browser/Web proxy)
- File caching (at server and client)

## Scaling – The Problem

**Observation:** Applying scaling techniques is easy, except for one thing:

> *Having multiple copies (cached or replicated), leads to* ***inconsistencies****: modifying one copy makes that copy different from the rest.*
>
> *Always keeping copies consistent and in a general way requires* ***global synchronization*** *on each modification.*
>
> *Global synchronization precludes large-scale solutions.*

**Observation:** If we can tolerate inconsistencies, we may reduce the need for global synchronization.

**Observation:** Tolerating inconsistencies is application dependent.

https://www.youtube.com/watch?v=NYBKXzl5bWU

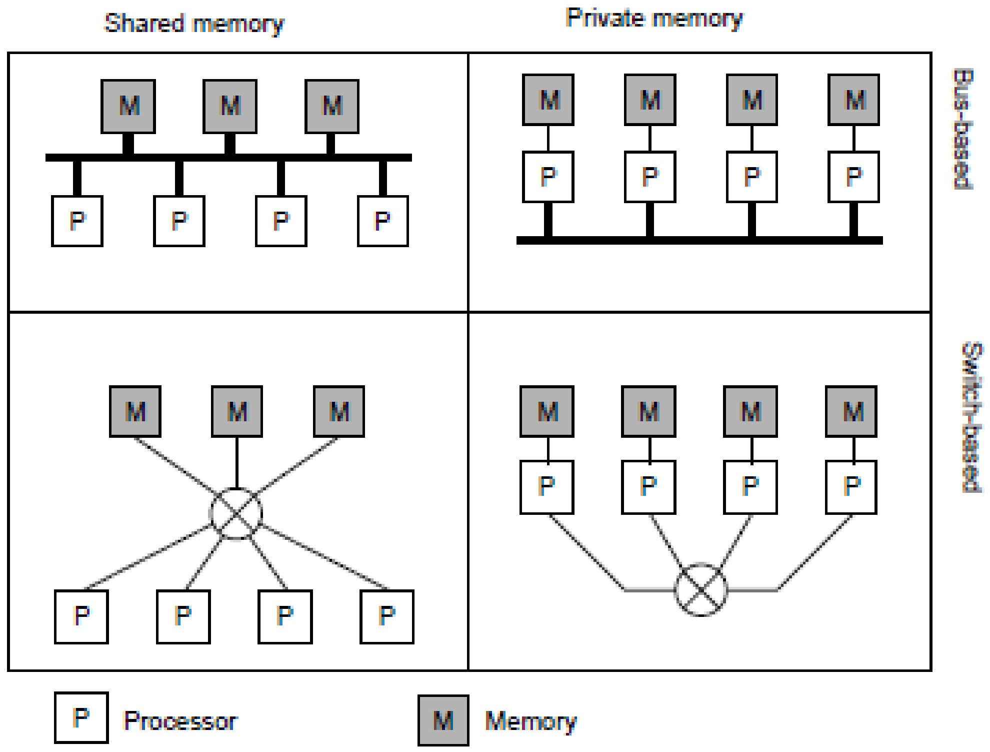
## Distributed Systems: Hardware Concepts

- Multiprocessors
- Multicomputers
- Networks of Computers

## Multiprocessors and Multicomputers

**Distinguishing features:**

- Private versus shared memory
- Bus versus switched interconnection



## Networks of Computers

**High degree of node heterogeneity:**

- High-performance parallel systems (multiprocessors as well as multicomputers)
- High-end PCs and workstations (servers)
- Simple network computers (offer users only network access)
- Mobile computers (palmtops, laptops)
- Multimedia workstations

**High degree of network heterogeneity:**

- Local-area gigabit networks
- Wireless connections
- Long-haul, high-latency POTS connections
- Wide-area switched megabit connections

**Observation:** Ideally, a distributed system hides these differences

## Distributed Systems: Software Concepts

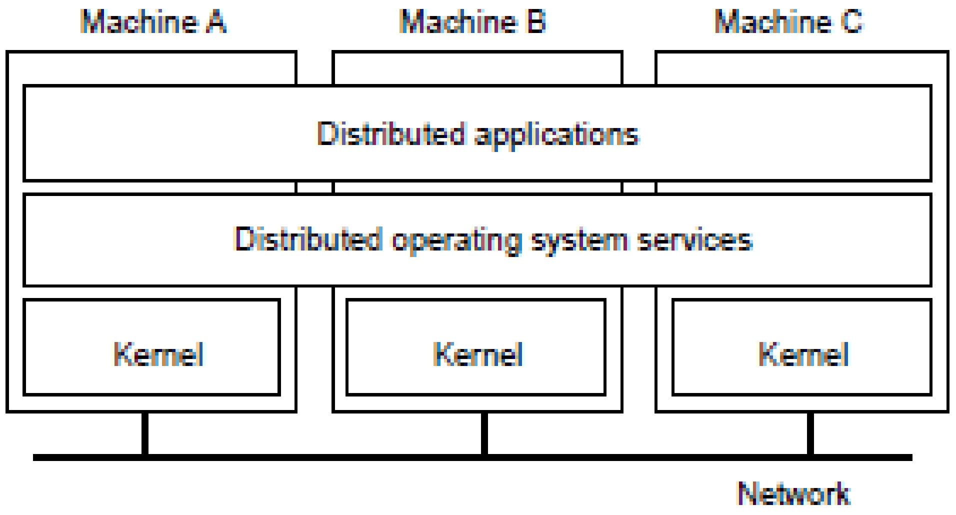
- Distributed operating system
- Network operating system
- Middleware

| System | Description | Main goal |
|---|---|---|
| DOS | Tightly-coupled OS for multiprocessors and homogeneous multicomputers | Hide and manage hardware resources |
| NOS | Loosely-coupled OS for heterogeneous multicomputers (LAN and WAN) | Offer local services to remote clients |
| Middle-ware | Additional layer atop of NOS implementing general-purpose services | Provide distribution transparency |

## Distributed Operating System

**Some characteristics:**

- OS on each computer knows about the other com- puters
- OS on different computers generally the same
- Services are generally (transparently) distributed across computers



## Multicomputer Operating System

**Harder than traditional (multiprocessor) OS:** Be- cause memory is not shared, emphasis shifts to pro- cessor communication by message passing:

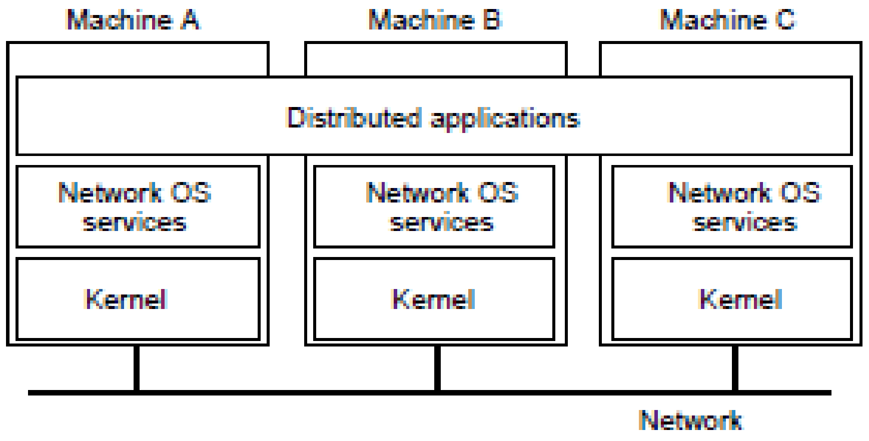
- Often no simple global communication:
  - Only bus-based multicomputers provide hard- ware broadcasting
  - Efficient broadcasting may require network in- terface programming techniques
- No simple systemwide synchronization mechanisms
- Virtual (distributed) shared memory requires OS to maintain global memory map in software
- Inherent distributed resource management: no cen- tral point where allocation decisions can be made

**Practice:** Only very few truly multicomputer operating systems exist (example: Amoeba)

## Network Operating System

**Some characteristics:**

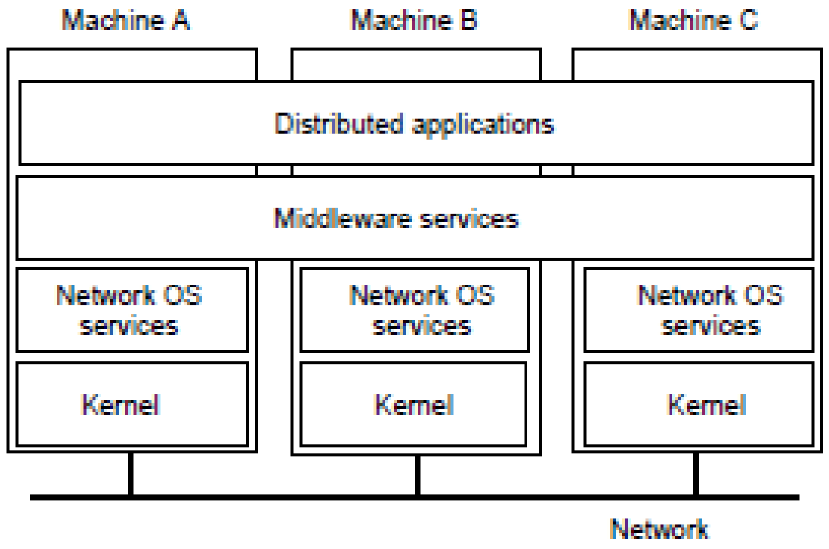
- Each computer has its own operating system with networking facilities
- Computers work independently (i.e., they may even have different operating systems)
- Services are tied to individual nodes (ftp, telnet, WWW)
- Highly file oriented (basically, processors share *only* files)



## Distributed System (Middleware)

**Some characteristics:**

- OS on each computer need not know about the other computers
- OS on different computers need not generally be the same
- Services are generally (transparently) distributed across computers



## Need for Middleware

**Motivation:** Too many networked applications were hard or difficult to integrate:

- Departments are running different NOSs
- Integration and interoperability only at level of primitive NOS services
- Need for federated information systems:
  - Combining different databases, but providing a single view to applications
  - Setting up enterprise-wide Internet services, making use of existing information systems
  - Allow transactions across different databases
  - Allow extensibility for future services (e.g., mobility, teleworking, collaborative applications)
- Constraint: use the existing operating systems, and treat them as the underlying environment (they provided the basic functionality anyway)

## Middleware Services (1/2)

**Communication services:** Abandon primitive socket-based message passing in favor of:

- Procedure calls across networks
- Remote-object method invocation
- Message-queuing systems
- Advanced communication streams
- Event notification service

**Information system services:** Services that help manage data in a distributed system:

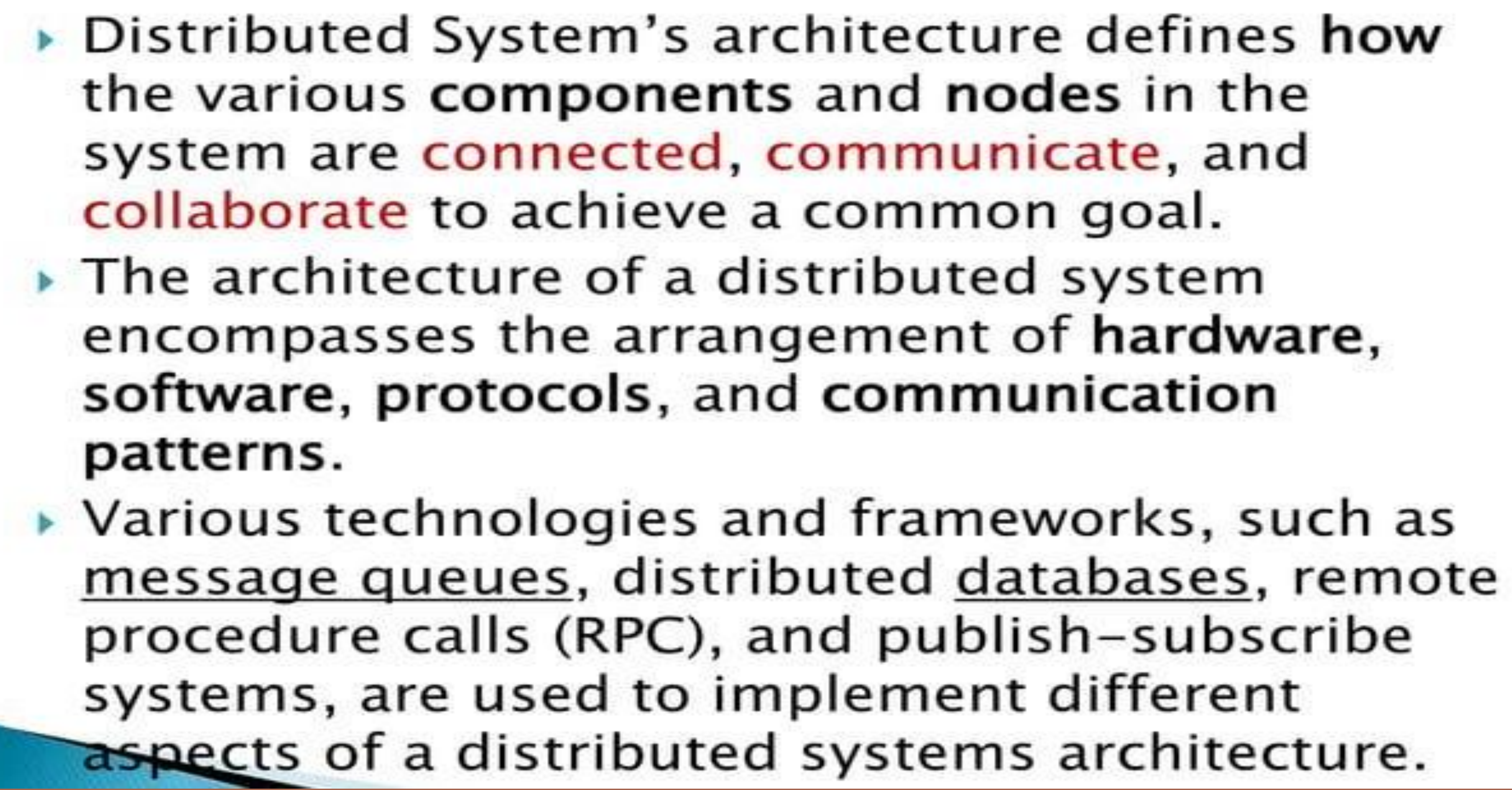
- Large-scale, systemwide naming services
- Advanced directory services (search engines)
- Location services for tracking mobile objects
- Persistent storage facilities
- Data caching and replication

## Comparison of DOS, NOS, and Middleware

1: Degree of transparency
2: Same operating system on each node?
3: Number of copies of the operating system
4: Basis for communication
5: How are resources managed?
6: Is the system easy to scale?
7: How open is the system?

| Item | Distributed OS | | Network OS | Middle-ware DS |
|---|---|---|---|---|
| | multiproc. | multicomp. | | |
| 1 | Very High | High | Low | High |
| 2 | Yes | Yes | No | No |
| 3 | 1 | N | N | N |
| 4 | Shared memory | Messages | Files | Model specific |
| 5 | Global, central | Global, distributed | Per node | Per node |
| 6 | No | Moderately | Yes | Varies |
| 7 | Closed | Closed | Open | Open |

## System Architecture



- Distributed System's architecture defines **how** the various **components** and **nodes** in the system are connected, communicate, and collaborate to achieve a common goal.
- The architecture of a distributed system encompasses the arrangement of **hardware**, **software**, **protocols**, and **communication patterns**.
- Various technologies and frameworks, such as message queues, distributed databases, remote procedure calls (RPC), and publish–subscribe systems, are used to implement different aspects of a distributed systems architecture.

- Common types of architecture are

1. **Centralized architectures ( traditional )** in which a single server implements most of the software components (and thus functionality), while remote clients can access that server using simple communication means.
2. **Decentralized architectures** in which machines more or less play **equal** roles, as well as hybrid organizations.

# Client-Server Architectures

**Single-tiered:** dumb terminal/mainframe configuration

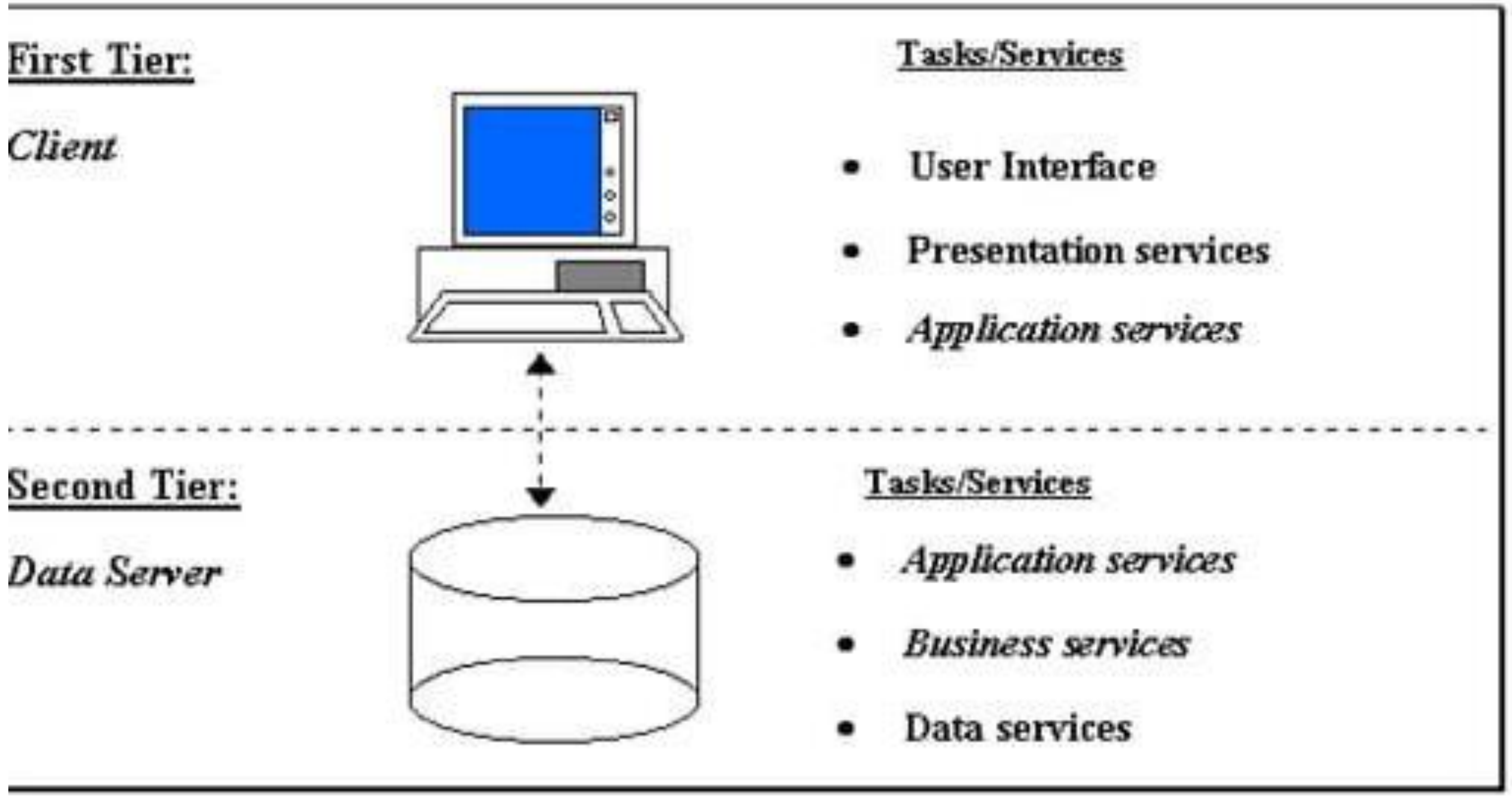
**Two-tiered:** client/single server configuration

**Three-tiered:** each layer on separate machine

## 2-Tier Architecture

- The Two-tier architecture is divided into two parts:
  - **Client Application (Client Tier)**
  - **Database (Data Tier)**
- **For e.g.:** On client application side the code is written for saving the data in the SQL server database. Client sends the request to server and it process the request & send back with data. The main problem of two tier architecture is the server cannot respond multiple request same time, as a result it cause a data integrity issue.
- **Advantages:**
  - Easy to maintain and modification is bit easy
  - Communication is faster
- **Disadvantages**:
  - In two tier architecture application performance will be degrade upon increasing the users.
  - Cost-ineffective

# 2-Tier Architecture



## What is Fat and Thin client?

- A fat client (also called heavy, rich or thick client) is a computer (client) in client server architecture or networks that typically provides rich functionality independent of the central server
  - Eg. SMTP,FTP,DNS Server
- A thin client (sometimes also called a lean, zero or slim client) is a computer or a program or an application that depends heavily on another computer (its *server*) to fulfill its computational roles.
  - Eg. Thunderbird, MS Outlook, Remote

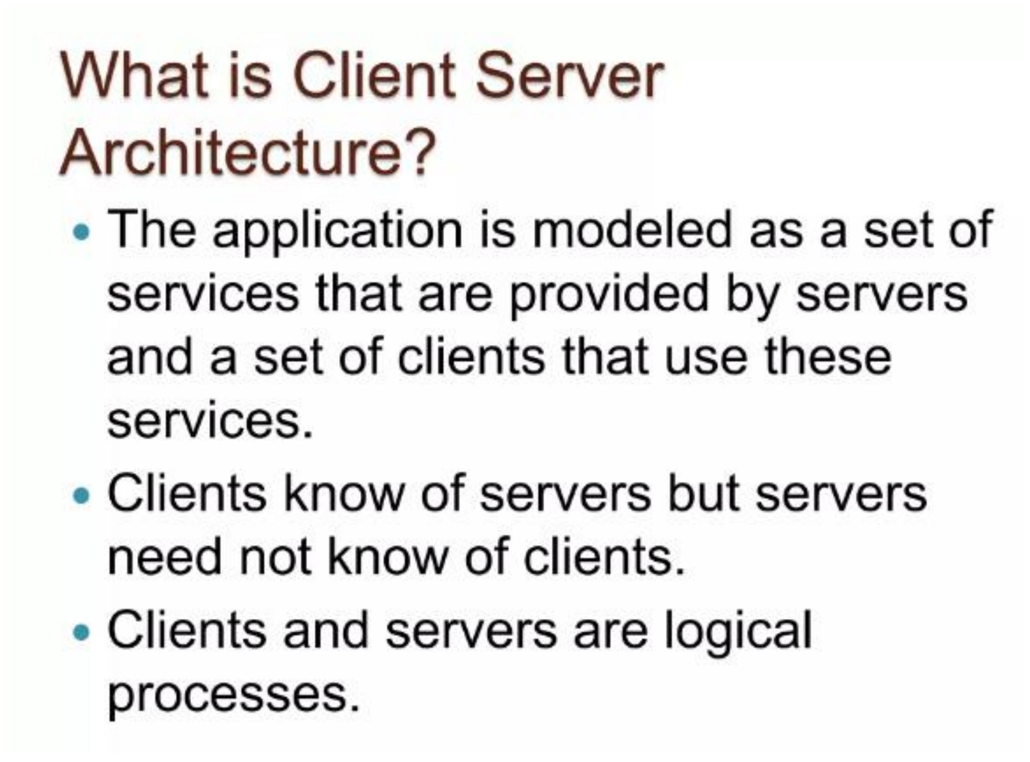
**Centralized Architecture in Distributed System**

The centralized architecture is defined as every node being connected to a central coordination system, and whatever information they desire to exchange will be shared by that system. A centralized architecture does not automatically require that all functions must be in a single place or circuit, but rather that most parts are grouped and none are repeated elsewhere as would be the case in a distributed architecture.

It consists following types of architecture:

•Client-server

•Application Layering



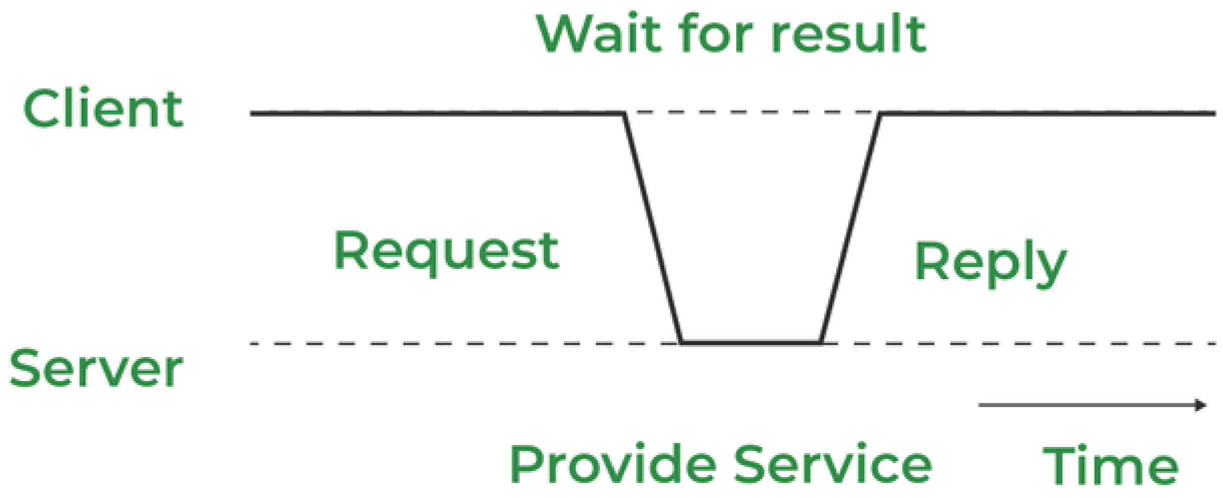
What is Client Server Architecture?

- The application is modeled as a set of services that are provided by servers and a set of clients that use these services.
- Clients know of servers but servers need not know of clients.
- Clients and servers are logical processes.

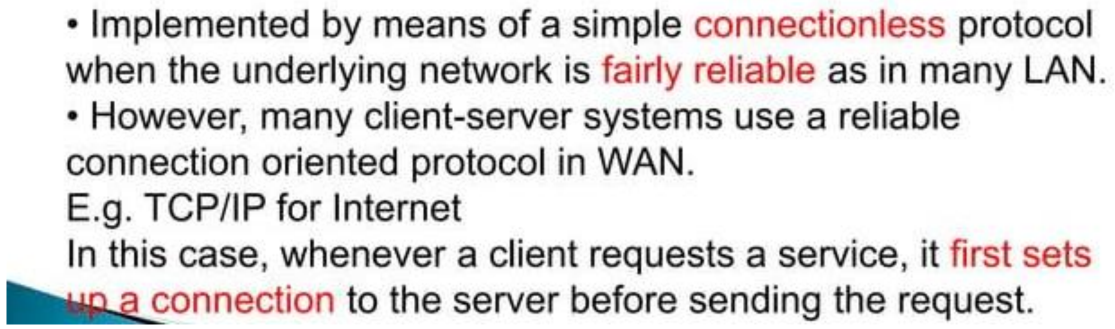
## Centralized Architecture in Distributed System

### Client Server

Processes in a distributed system are split into two (potentially overlapping) groups in the fundamental client-server architecture. A server is a program that provides a particular service, such as a database service or a file system service. A client is a process that sends a request to a server and then waits for the server to respond before requesting a service from it. This client-server interaction, also known as request-reply behavior is shown in the figure be



- When the underlying network is reasonably dependable, as it is in many local-area networks, communication between a client and a server can be implemented using a straightforward connection-less protocol. In these circumstances, a client simply bundles a message for the server specifying the service they want along with the relevant input data when they make a service request. After that, the server receives the message. The latter, on the other hand, will always await an incoming request, process it after that, and then package the outcomes in a reply message that is then provided to the client.
- Efficiency is a clear benefit of using a connectionless protocol. The request/reply protocol just sketched up works as long as communications do not get lost or damaged. It is unfortunately not easy to make the protocol robust against occasional transmission errors. When no reply message is received, our only option is to perhaps allow the client to resubmit the request. However, there is a problem with the client's ability to determine if the original request message was lost or if the transmission of the reply failed.
- A reliable connection-oriented protocol is used as an alternative by many client-server systems. Due to its relatively poor performance, this method is not totally suitable for local-area networks, but it is ideal for wide-area networks where communication is inherently unreliable.



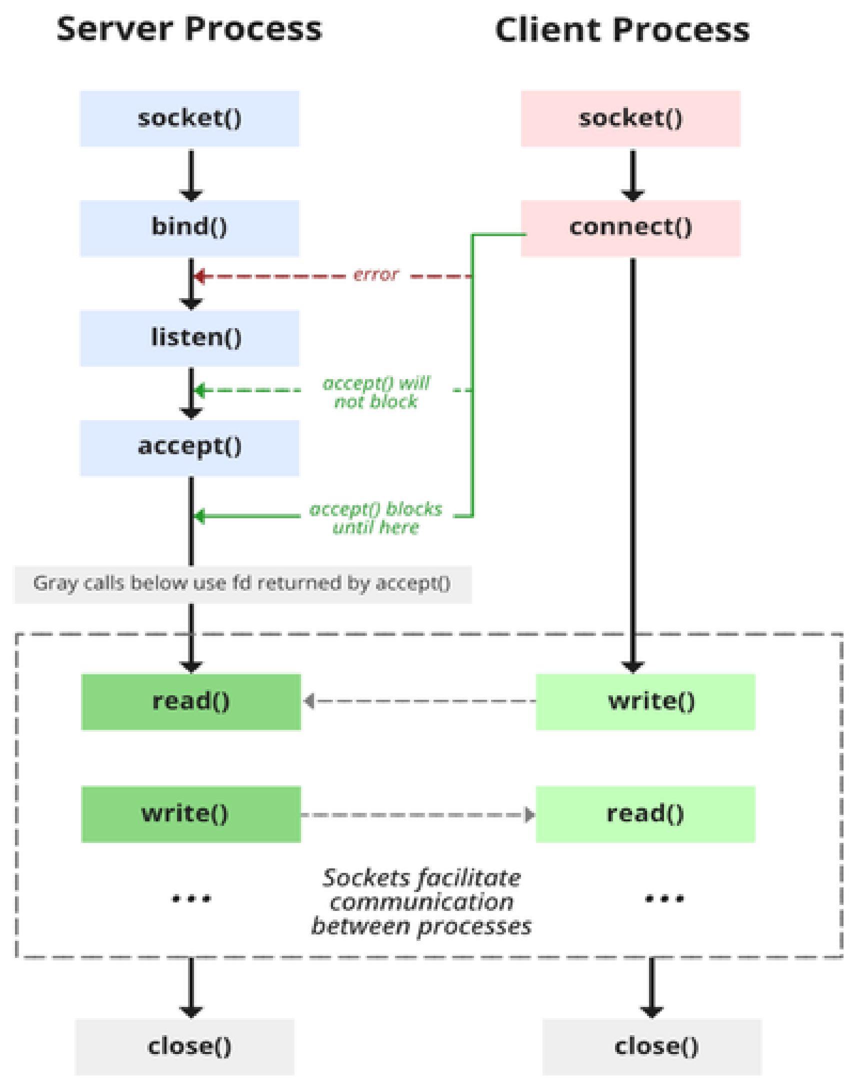
• Implemented by means of a simple connectionless protocol when the underlying network is fairly reliable as in many LAN.
• However, many client-server systems use a reliable connection oriented protocol in WAN.
E.g. TCP/IP for Internet
In this case, whenever a client requests a service, it first sets up a connection to the server before sending the request.

## What is Socket Programming?

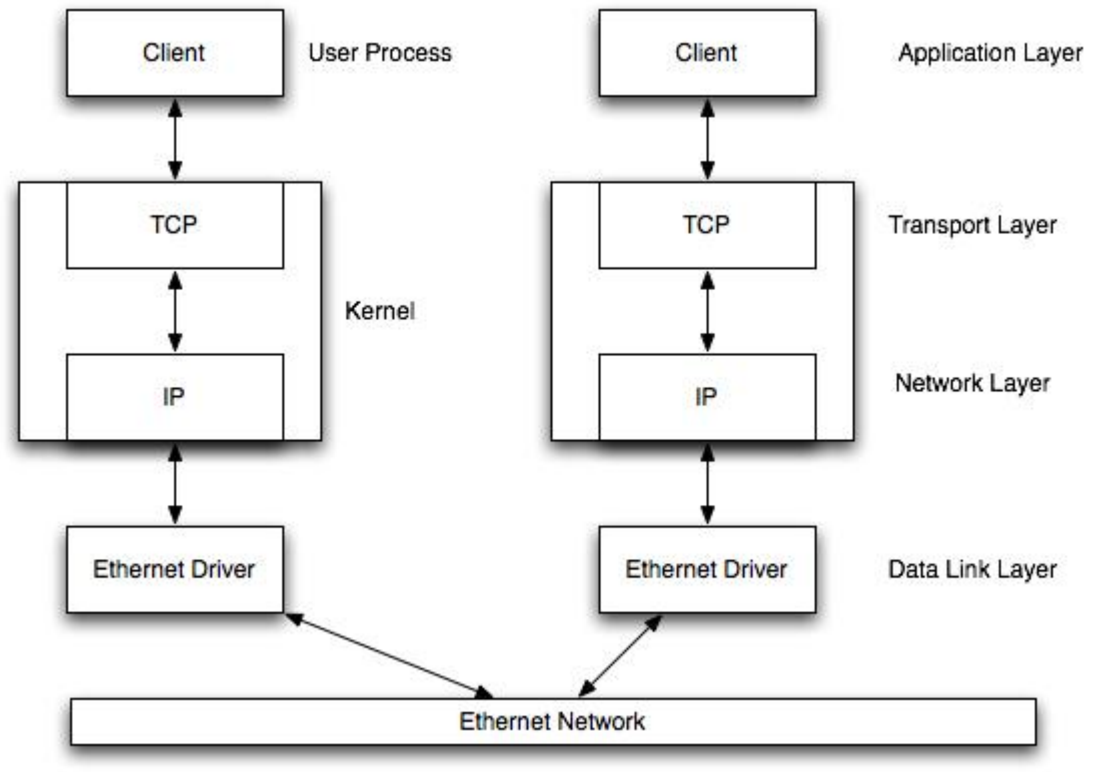
Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket(node) listens on a particular port at an IP, while the other socket reaches out to the other to form a connection. The server forms the listener socket while the client reaches out to the server.

**Socket programming in Java** allows different programs to communicate with each other over a network, whether they are running on the same machine or different ones. This method describes a very basic one-way Client and Server setup, where a Client connects, sends messages to the server and the server shows them using a socket connection. There is a lot of low-level stuff that needs to happen for these things to work but the Java API networking package (java.net) takes care of all of that, making network programming very easy for programmers.

***Note:** A "socket" is an endpoint for sending and receiving data across a network.*



State diagram for server and client model of Socket



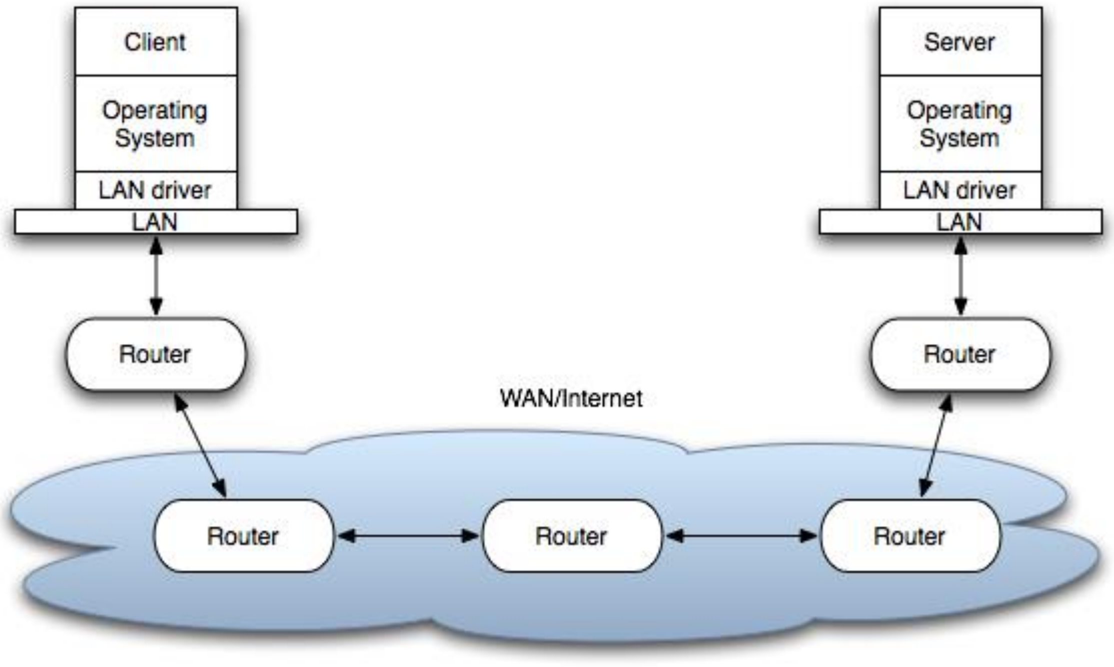
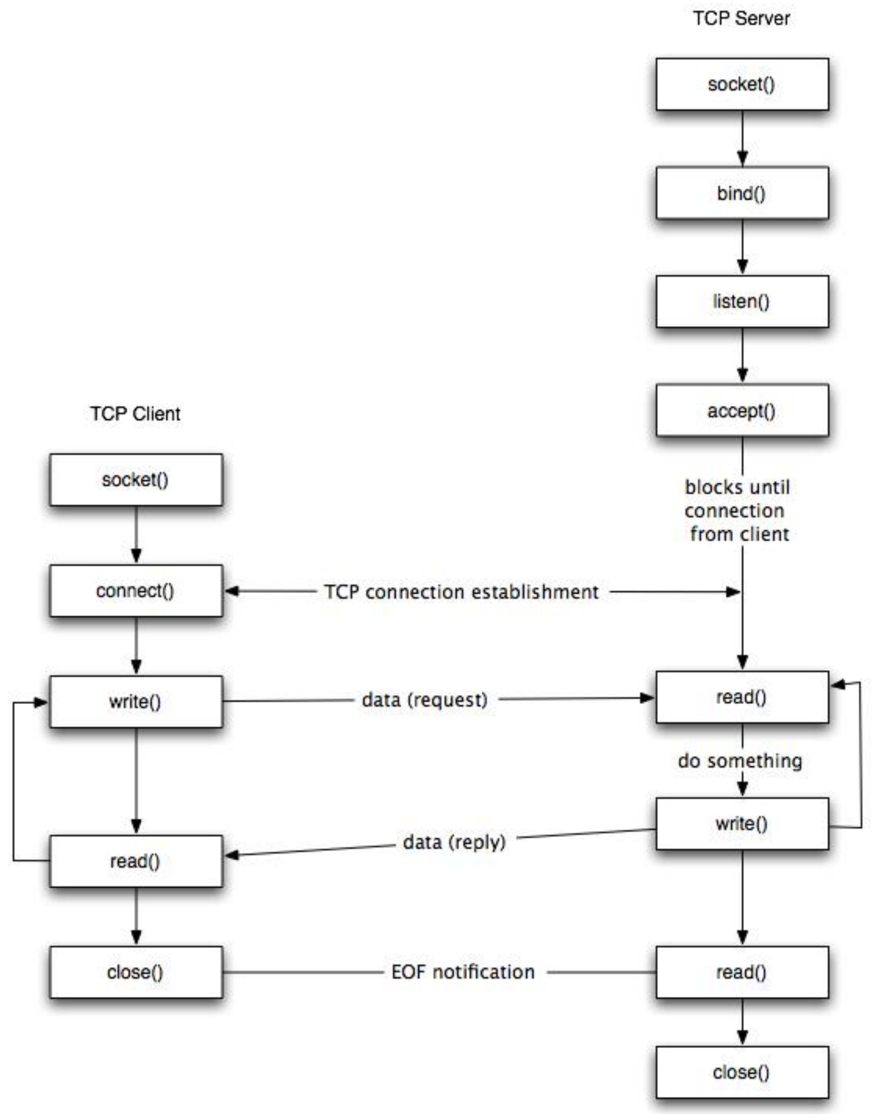


**Figure 1:** Client and server on the same Ethernet communicating using TCP/IP.

**Figure 2:** Client and server on different LANs connected through WAN/Internet.

## TCP Socket API

The sequence of function calls for the client and a server participating in a TCP connection is presented in Figure 3.



As shown in the figure, the steps for establishing a TCP socket on the client side are the following:

•Create a socket using the `socket()` function;

•Connect the socket to the address of the server using the `connect()` function;

•Send and receive data by means of the `read()` and `write()` functions.

The steps involved in establishing a TCP socket on the server side are as follows:

•Create a socket with the `socket()` function;

•Bind the socket to an address using the `bind()` function;

•Listen for connections with the `listen()` function;

•Accept a connection with the `accept()` function system call. This call typically blocks until a client connects with the server.

•Send and receive data by means of `send()` and `receive()`.

**Figure 3:** TCP client-server.

## Client-Side Programming

### 1. Establish a Socket Connection

To connect to another machine we need a socket connection. A socket connection means both machines know each other's IP address and TCP port. The java.net.Socket class is used to create a socket.

```
Socket socket = new Socket("127.0.0.1", 5000)
```

The first argument: The IP address of Server i.e. 127.0.0.1 is the IP address of localhost, where code will run on the single stand-alone machine.

The second argument: The TCP Port number (Just a number representing which application to run on a server. For example, HTTP runs on port 80. Port number can be from 0 to 65535)

### 2. Communication

To exchange data over a socket connection, streams are used for input and output:

Input Stream: Reads data coming from the socket.
Output Stream: Sends data through the socket.
Example to access these streams:

```
// to read data
InputStream input = socket.getInputStream();
// to send data
OutputStream output = socket.getOutputStream();
```

### 3. Closing the Connection

The socket connection is closed explicitly once the message to the server is sent.

Example: Here, in the below program the Client keeps reading input from a user and sends it to the server until "Over" is typed.

Server-Side Programming

1. Establish a Socket Connection

To create a server application two sockets are needed.

ServerSocket: This socket waits for incoming client requests. It listens for connections on a specific port.

Socket: Once a connection is established, the server uses this socket to communicate with the client.

2. Communication

Once the connection is established, you can send and receive data through the socket using streams.

The getOutputStream() method is used to send data to the client.

3. Close the Connection

Once communication is finished, it’s important to close the socket and the input/output streams to free up resources.

**Three tier architecture**
**Application Layering**

However, many individuals have urged a distinction between the three levels below, effectively adhering to the layered architectural approach we previously described, given that many client-server applications are intended to provide user access to databases:

•The user-interface level
•The processing level
•The data level

Everything required to connect with the user, such as display management directly, is contained at the user-interface level. Applications are often found at the processor level. The actual data that is used to make decisions is managed at the data level.

**The User Interface Level**

The user-interface level is often implemented by clients. Programs that let users interact with applications make up this level. The sophistication level across application programs differs significantly. A character-based screen is the most basic user-interface application.
Typically, mainframe environments have employed this kind of interface. One hardly ever speaks of a client-server setup in situations where the mainframe manages every aspect of interaction, including the keyboard and monitor.

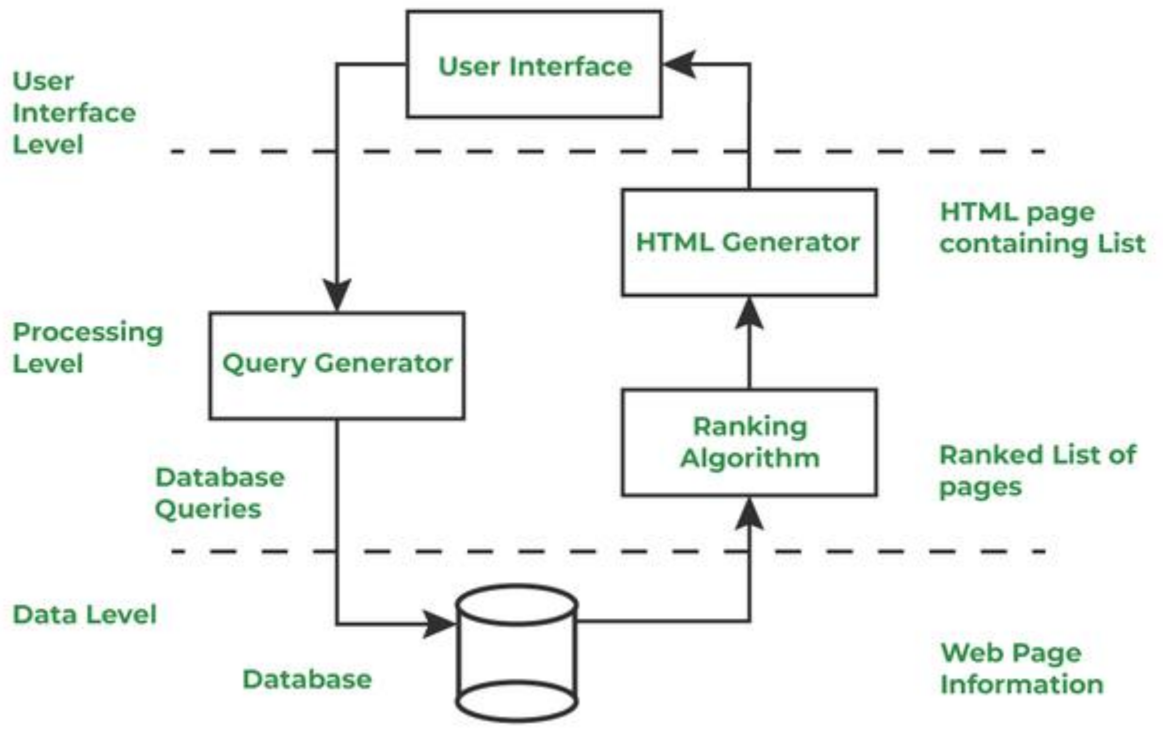
**The Processing Level**

This is the middle part of the architecture. This is a logical part of the system where all the processing actions are performed on the user interaction and the data level. It processes the requests from the user interface and performs certain operations.

**The Data Level**

The data level in the client-server model contains the programs that maintain the actual data on which the applications operate. An important property of this level is that data are often persistent, that is, even if no application is running, data will be stored somewhere for the next use. In its simplest form, the data level consists of a file system, but it is more common to use a full-fledged database. In the client-server model, the data level is typically implemented on the server side.

Example: Consider an Internet search engine. The user interface of a search engine is very simple: a user types in a string of keywords and is subsequently presented with a list of titles of Web pages. The back end is formed by a huge database of Web pages that have been prefetched and indexed. The core of the search engine is a program that transforms the user's string of keywords into one or more database queries. It subsequently ranks the results into a list and transforms that list into a series of HTML pages. Within the client-server model, this information retrieval part is typically placed at the processing level.



## What is Decentralized Architecture?

Decentralized architecture means a conceptual design of a system's components as elements that are mutually integrated and act according to the general principles of a system without a specific coordinative center.

•In such systems, the control and data are present in different nodes and each node is independent and can make decisions independently.

•This architecture improves the system's capability on scalability, fault tolerance and robustness in eliminating a single centralized point of failure and integrating peers to work together. Distribution is a usual practice in the blockchain networks, P2P systems, and distributed ledger technology field.

**Key Concepts in Decentralized Systems**

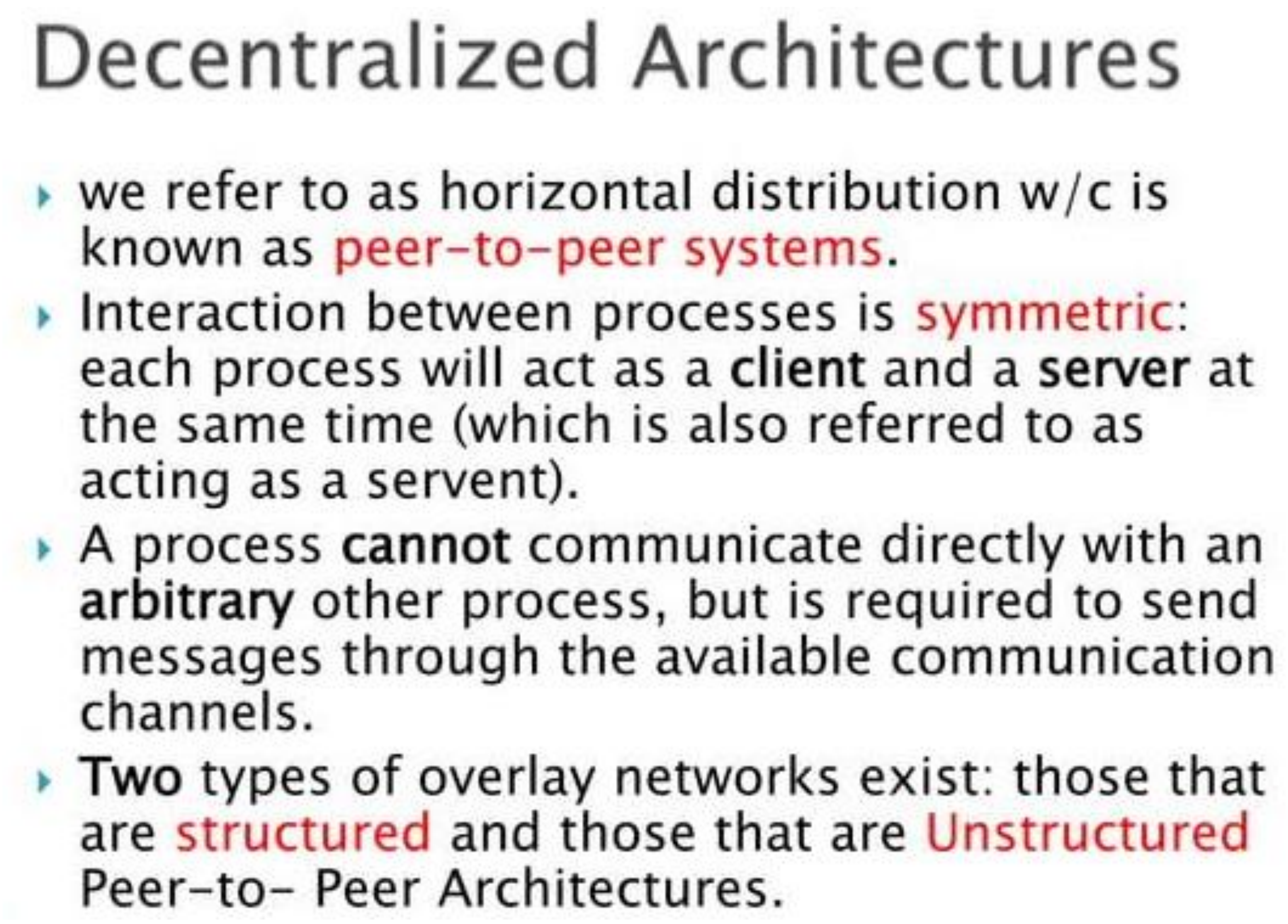
Below are the key concepts of distributed systems:

**Peer-to-Peer (P2P) Networks:** the distributed network in which most of the communicating entities have equal capabilities, and each peer is on an equal level, using the resources of other peers directly without any interference from the hub. These include; the Bit Torrent and the Block chain networks.

**Consensus Mechanisms:** A procedure applied to solve a consistency problem in distributed computing, to have a unique value or state among the participating processes or systems. They also apply decentralization procedures to make sure they are not only uniform and safe as well as trustworthy.

**Decentralized Autonomous Organizations (DAOs):** Organizations depicted by rules written in a program that is, clear; open for modification by organization members, and not dominated by central government. A smart contract is a digital contract that exists on a blockchain and DAOs are fully automated.

**Data Replication:** The write operation of creating backups of the data, which is committed to several nodes to warrant its accessibility and integrity. The consensus helps in providing a certain level of repetitiveness, and error tolerance within the decentralized applications.

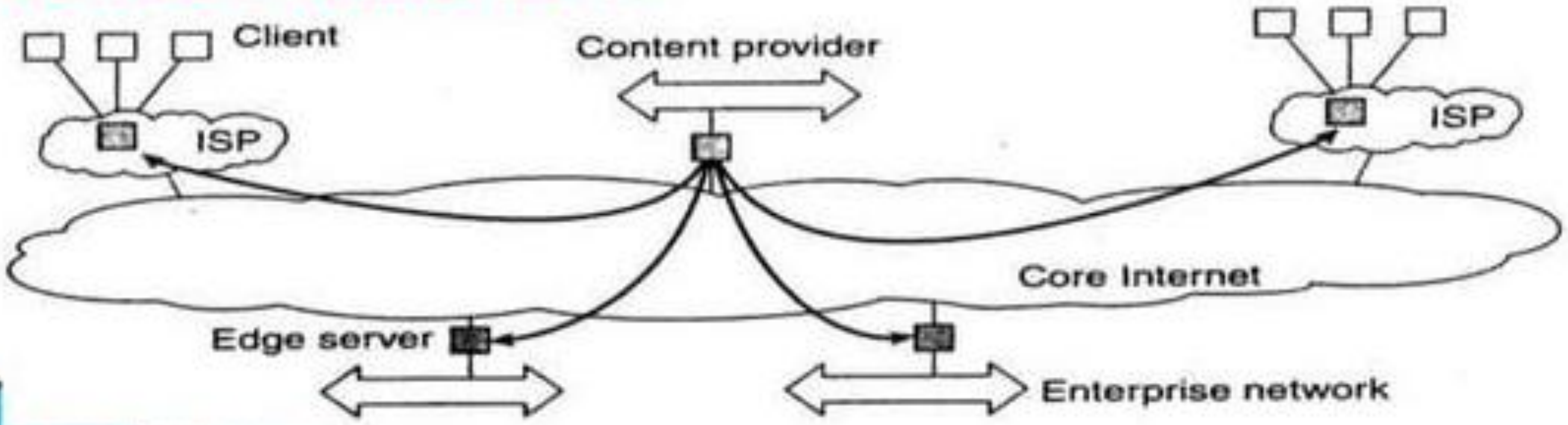


## Decentralized Architectures

- we refer to as horizontal distribution w/c is known as peer-to-peer systems.
- Interaction between processes is symmetric: each process will act as a **client** and a **server** at the same time (which is also referred to as acting as a servent).
- A process **cannot** communicate directly with an **arbitrary** other process, but is required to send messages through the available communication channels.
- **Two** types of overlay networks exist: those that are structured and those that are Unstructured Peer-to- Peer Architectures.

## Hybrid Architectures

- Distributed systems in which client-server solutions are combined with **decentralized** architectures.
- **Edge-Server Systems**
  Servers are placed "at the edge" of the network formed by enterprise networks and the actual Internet, for example, as provided by an Internet Service Provider (ISP).



## Client-Server Model

## Servers general design issues / Design Issues of Distributed System

Certainly! Here is a detailed breakdown of the key design issues of distributed systems:

1. Scalability

•**Challenges**

- **Handling Increased Load:** As the number of users or requests increases, the system must scale accordingly without performance degradation.
- **Geographic Distribution:** Ensuring performance across geographically dispersed locations.

•**Strategies to Achieve Scalability**

- **Horizontal Scaling:** Adding more nodes to the system.
- **Vertical Scaling:** Enhancing the capacity of existing nodes.
- **Sharding:** Dividing the database into smaller, more manageable pieces.

2. Reliability

•**Fault Tolerance**

- **Redundancy:** Using duplicate components to take over in case of failure.
- **Failover Mechanisms:** Automatically switching to a standby system when the primary system fails.

•**Redundancy and Replication**

- **Data Replication:** Storing copies of data on multiple nodes to ensure availability and reliability.
- **Consensus Algorithms:** Ensuring consistency among replicated data (e.g., Paxos, Raft).

## 3. Availability

- **Uptime and Downtime Considerations**
  - **High Availability Architectures:** Designing systems to minimize downtime.
  - **Monitoring and Alerting:** Using tools to detect and respond to issues promptly.
- **Techniques to Improve Availability**
  - **Load Balancers:** Distributing incoming requests across multiple servers.
  - **Geographic Redundancy:** Deploying servers in different geographic locations to avoid single points of failure.

## 4. Consistency

**Data Consistency Models**

**Strong Consistency:** Ensuring that all nodes see the same data at the same time.

**Eventual Consistency:** Allowing for temporary discrepancies between nodes, with eventual convergence.

**Trade-offs between Consistency and Availability (CAP Theorem)**

**CAP Theorem:** Understanding the trade-off between Consistency, Availability, and Partition Tolerance.

## 5. Latency

**Sources of Latency**

**Network Delays:** Time taken for data to travel across the network.

**Processing Delays:** Time taken for nodes to process requests.

**Minimization Techniques**

**Caching:** Storing frequently accessed data closer to the user.

**Data Compression:** Reducing the amount of data that needs to be transferred.

6. Load Balancing

- **Load Distribution Methods**
  - **Round Robin:** Distributing requests evenly across servers.
  - **Least Connections:** Directing traffic to the server with the fewest active connections.
- **Dynamic vs. Static Load Balancing**
  - **Dynamic:** Adapting to changing loads in real-time.
  - **Static:** Using predetermined load distribution strategies.

**7. Security**

- **Authentication and Authorization**
  - **Identity Verification:** Ensuring that users are who they claim to be.
  - **Access Control:** Restricting access to resources based on user roles.
- **Data Encryption and Secure Communication**
  - **Encryption:** Protecting data in transit and at rest.
  - **Secure Protocols:** Using HTTPS, SSL/TLS for secure communications.

**8. Architectural Design Patterns**

- **Client-Server Model**
  - **Centralized Servers:** Handling requests from multiple clients.
- **Peer-to-Peer Model**
  - **Decentralized Network:** Nodes act as both clients and servers.
- **Microservices Architecture**
  - **Service Decomposition:** Breaking down applications into smaller, independent services.
- **Service-Oriented Architecture (SOA)**
  - **Service Reusability:** Designing services to be reused across different applications.

## 9. Communication Issues

- **Network Protocols**
  - **TCP/IP:** Ensuring reliable, ordered, and error-checked delivery of data.
  - **UDP:** Providing faster, connectionless communication.
- **Message Passing vs. Shared Memory**
  - **Message Passing:** Communicating by sending messages between nodes.
  - **Shared Memory:** Direct access to a common memory space.
- **Synchronous vs. Asynchronous Communication**
  - Synchronous: Blocking operations until a response is received.
  - Asynchronous: Allowing operations to proceed without waiting for a response.

## 10. Data Management

- **Data Distribution and Partitioning**
  - **Horizontal Partitioning:** Distributing rows of a database across different nodes.
  - **Vertical Partitioning:** Distributing columns of a database across different nodes.
- **Database Replication**
  - **Master-Slave Replication:** One master node with multiple read-only slave nodes.
  - **Multi-Master Replication:** Multiple nodes capable of both read and write operations.
- **Handling Distributed Transactions**
  - **Two-Phase Commit:** Ensuring all nodes agree on a transaction’s outcome.
  - **Distributed Ledger Technologies:** Using blockchain for immutable and verifiable transactions.

## Types of Distributed Systems

Distributed Computing Systems

- ❑Clusters
- ❑Grids
- ❑Clouds

Distributed Information Systems

- ❑Transaction Processing Systems
- ❑Enterprise Application Integration

❑Internet

# Cluster Computing

## Definition

- ❑A collection of similar processors (PCs, workstations) running the same operating system, connected by a high-speed LAN.
- ❑Parallel computing capabilities using inexpensive PC hardware
- ❑Replace big parallel computers (MPPs)

## Types and Uses

### High Performance Clusters (HPC)

- ❑run large parallel programs
- ❑Scientific, military, engineering apps; e.g., weather modeling

### Load Balancing Clusters

- ❑Front end processor distributes incoming requests
- ❑server farms (e.g., at banks or popular web site)

### High Availability Clusters (HA)

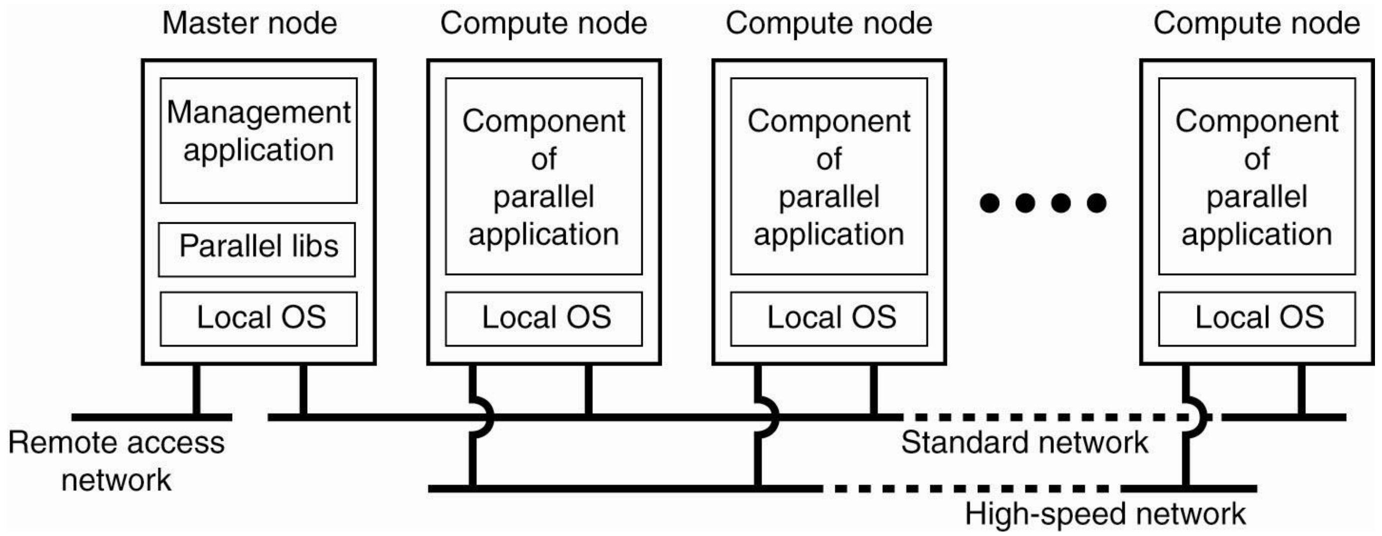
- ❑Provide redundancy – back up systems
- ❑May be more fault tolerant than large mainframes

## Clusters – Beowulf model

Linux-based

Master-slave paradigm

- One processor is the master; allocates tasks to other processors, maintains batch queue of submitted jobs, handles interface to users
- Master has libraries to handle message-based communication or other features (the middleware).



Master node
Compute node
Compute node
Compute node
Management application
Parallel libs
Local OS
Component of parallel application
Local OS
Component of parallel application
Local OS
Component of parallel application
Local OS
Remote access network
Standard network
High-speed network