



CHAPTER 2

ARCHITECTURES

BY ANKU JAISWAL




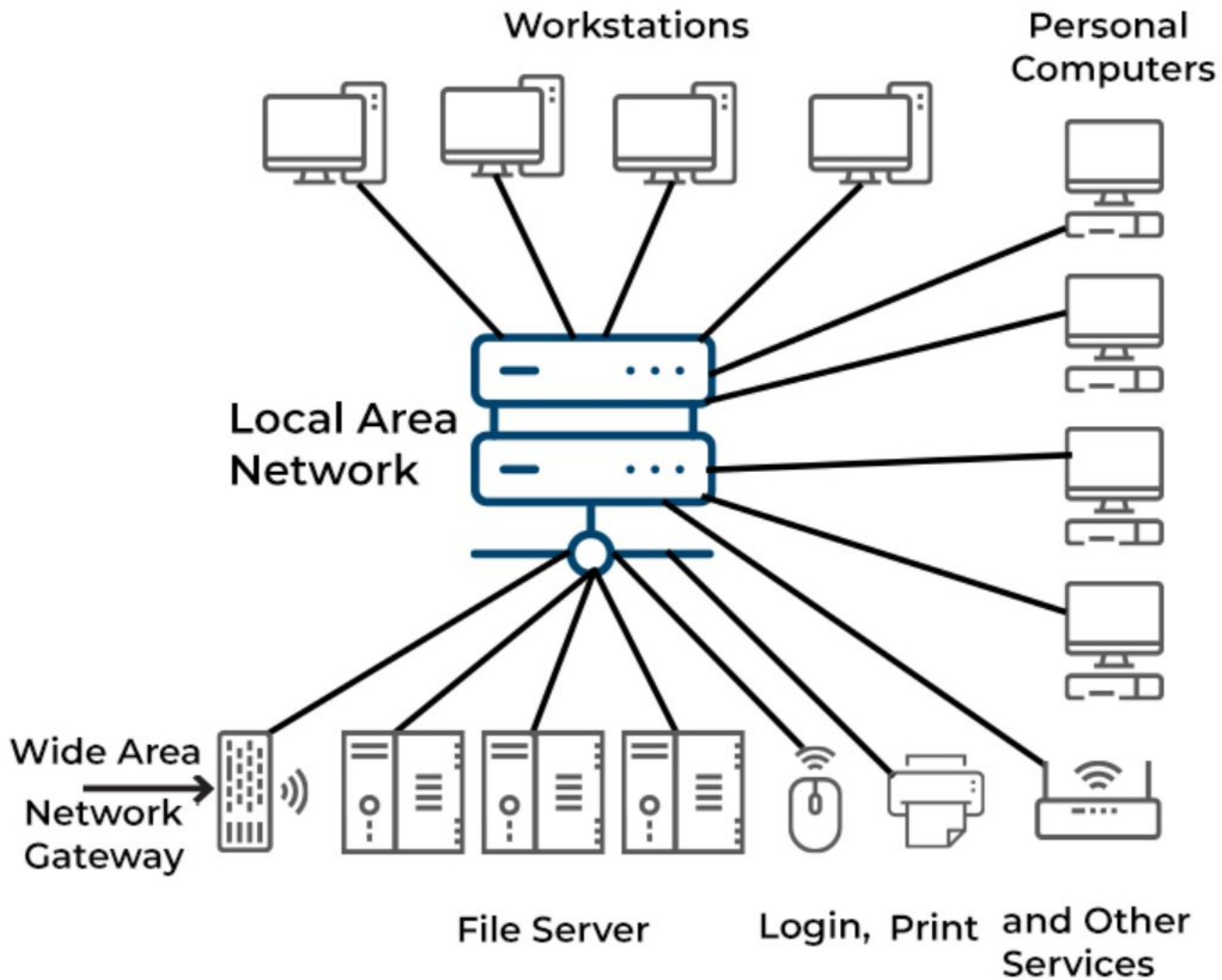
Contents

- 2.1 ARCHITECTURAL STYLES
 - 2.2. MIDDLEWARE ORGANIZATION
 - 2.3. SYSTEM ARCHITECTURES
 - 2.4. EXAMPLE ARCHITECTURE
- 





OVERVIEW

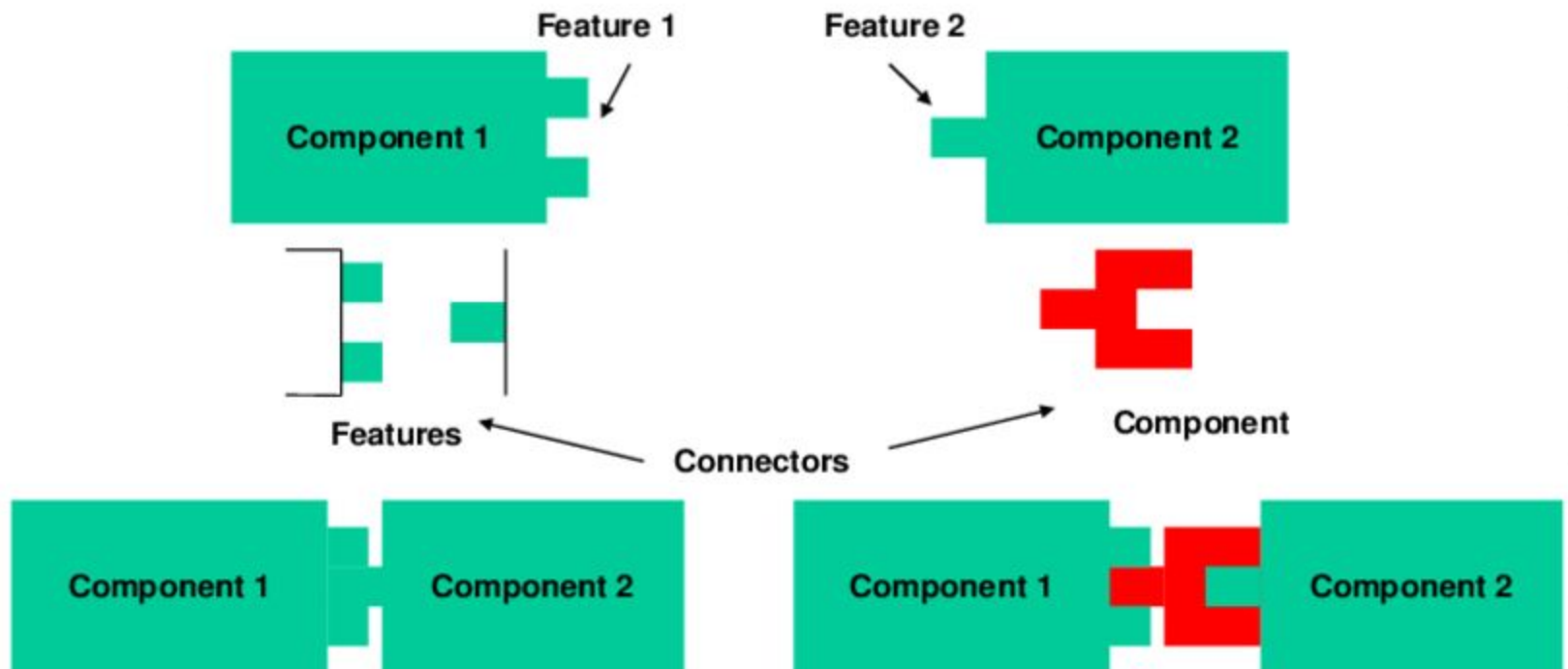
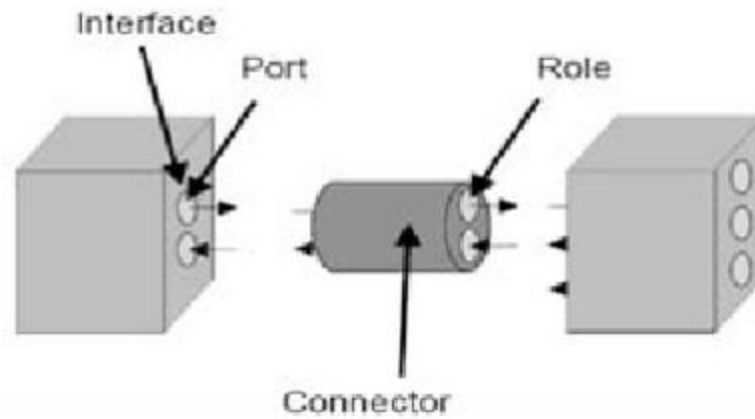
- Distributed systems are often complex pieces of software of which the components are by definition dispersed across multiple machines.
 - To master their complexity, it is crucial that these systems are properly organized.
- 



Distributed System Architecture


- Distributed system architectures are bundled up with components and connectors.
- Components can be individual nodes or important components in the architecture
- Connectors are the ones that connect each of these components.

- 
- 
- Component: A modular unit with well-defined interfaces; replaceable; reusable
 - Connector: A communication link between modules which mediates coordination or cooperation among components







So the idea behind distributed architectures is to:

- have these components presented on different platforms
 - where components can communicate with each other over a communication network in order to achieve specific objectives.
- 




2.1 ARCHITECTURAL STYLES

- In **Distributed Systems**, architectural styles define how components (clients, servers, services, or processes) interact and coordinate to achieve the system's goals.
 - Each style offers different **scalability, fault-tolerance, and communication mechanisms**.
- 




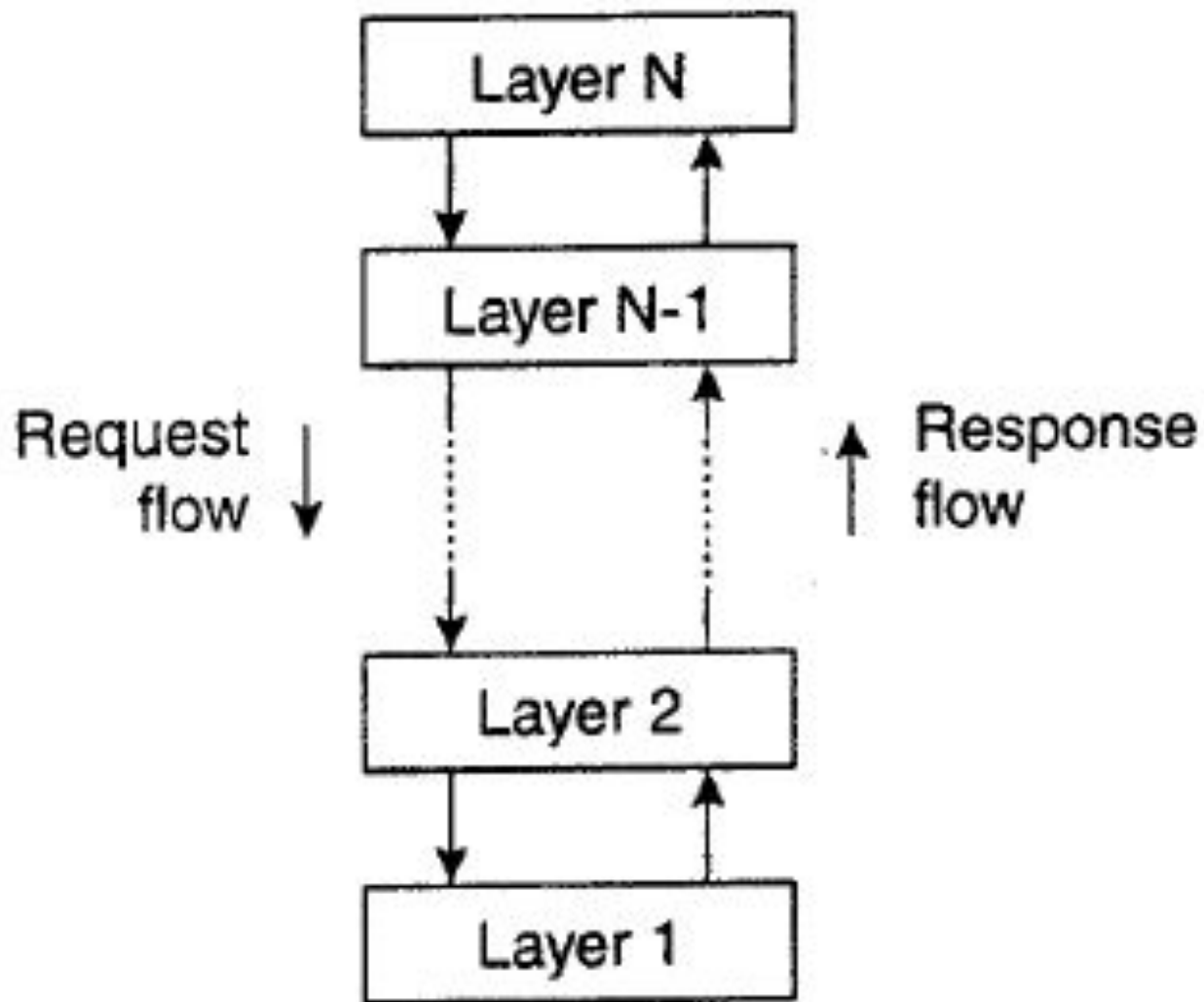
Using components and connectors, we can come to various configurations, which, in turn have been classified into architectural styles

- Layered architectures
 - Object-based architectures
 - Data-centered architectures
 - Event-based architectures
 - Service Oriented Architecture
- 




1. Layered Architecture

- A **layered architecture in distributed systems** is a design approach where the system is divided into multiple layers, each responsible for specific functions.
 - These layers are arranged hierarchically, where each layer provides services to the layer above it and uses services from the layer below it.
 - A well known example for this is the OSI model that incorporates a layered architecture when interacting with each of the components.
- 



(a)

- 
- The layers on the bottom provide a service to the layers on the top.
 - The request flows from top to bottom, whereas the response is sent from bottom to top.
 - The advantage of using this approach is that, each layer can be easily replaced or modified without affecting the entire architecture.



+-----+

| Application | (User-facing apps: banking, social media, etc.)

+-----+

| Middleware | (RPC, Messaging, Distributed objects, Transparency)

+-----+

| Operating System | (Resource management, process scheduling, security)

+-----+

| Communication Net | (Protocols, routing, addressing)

+-----+

| Hardware | (Servers, storage, network devices)

+-----+

1. **Application Layer**

- Closest to the end-user.
- Provides services like email, banking apps, e-commerce, social media, etc.
- Communicates with the middleware layer for resource access.

2. **Middleware Layer**

- Acts as a “glue” between applications and lower layers.
- Provides **transparency** (location, replication, concurrency, and failure transparency).
- Examples: Remote Procedure Calls (RPC), Message-Oriented Middleware (MOM), CORBA, Java RMI.

3. **Operating System Layer**

- Manages local resources (CPU, memory, I/O devices).
- Provides process scheduling, file systems, communication protocols, and security.
- Ensures applications can run on different hardware.



1. **Communication Layer (Networking Layer)**

- Handles data transfer between distributed nodes.
- Provides services like addressing, routing, error handling, and reliable message delivery.
- Protocols: TCP/IP, HTTP, gRPC.




2. **Hardware Layer**

- Physical devices (servers, PCs, mobile phones, IoT devices).
- Includes network infrastructure like routers, switches, and data centers.

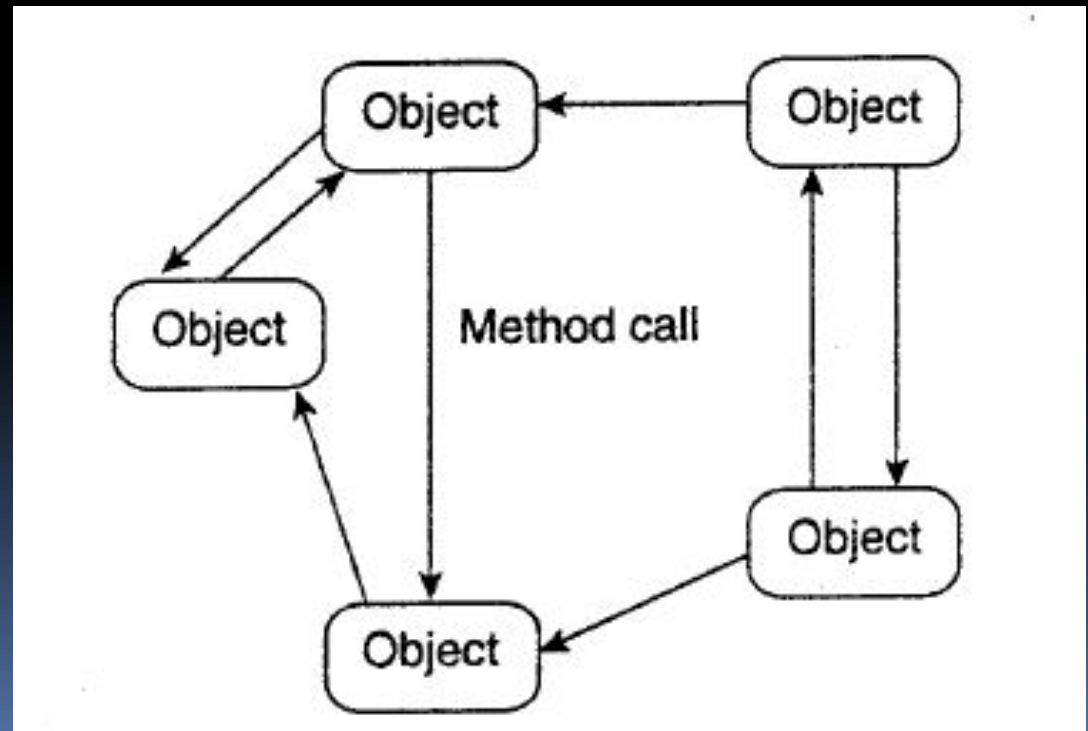


Advantages

- **Modularity** → Easier to develop, test, and maintain.
 - **Scalability** → Each layer can be upgraded without affecting others.
 - **Transparency** → Users do not need to know about underlying complexities.
 - **Reusability** → Common services (e.g., authentication, messaging) can be reused.
- 

2. Object-based Architectures

- In an **object-based architecture**, the entire distributed system is structured around **objects**.
- Each **object** represents an entity (data + behavior) with **methods** that can be invoked remotely or locally.



- 
- Communication between components happens via **Remote Method Invocation (RMI)** or similar mechanisms, instead of low-level message passing.
- 



Features

Encapsulation → Each object hides its internal state and exposes services through methods.

Transparency → Users don't need to know whether an object is local or remote.

Remote Object Invocation → Objects can invoke methods on other objects located on different nodes.



Naming Service → Objects are accessed through unique identifiers or directory services.

Reusability → Objects can be reused across different applications.



Components

Client Objects

- Objects that request services.
- Example: A "Student" object requesting "Course" details.

Server Objects

- Objects that provide services.
- Example: A "CourseDatabase" object serving requested course data.



Object Request Broker (ORB)

- Middleware that enables client objects to find and communicate with server objects.
- Example: CORBA (Common Object Request Broker Architecture).



Working

A client object calls a method on a remote object.

ORB/middleware locates the object across the network.



The request is transmitted, executed on the server object, and the result is returned.



Example

CORBA (Common Object Request Broker Architecture)

Java RMI (Remote Method Invocation)

Microsoft DCOM (Distributed Component Object Model)





Advantages

Natural fit for **object-oriented programming** (Java, C++, Python).

Provides **location transparency** (local vs remote objects look the same).

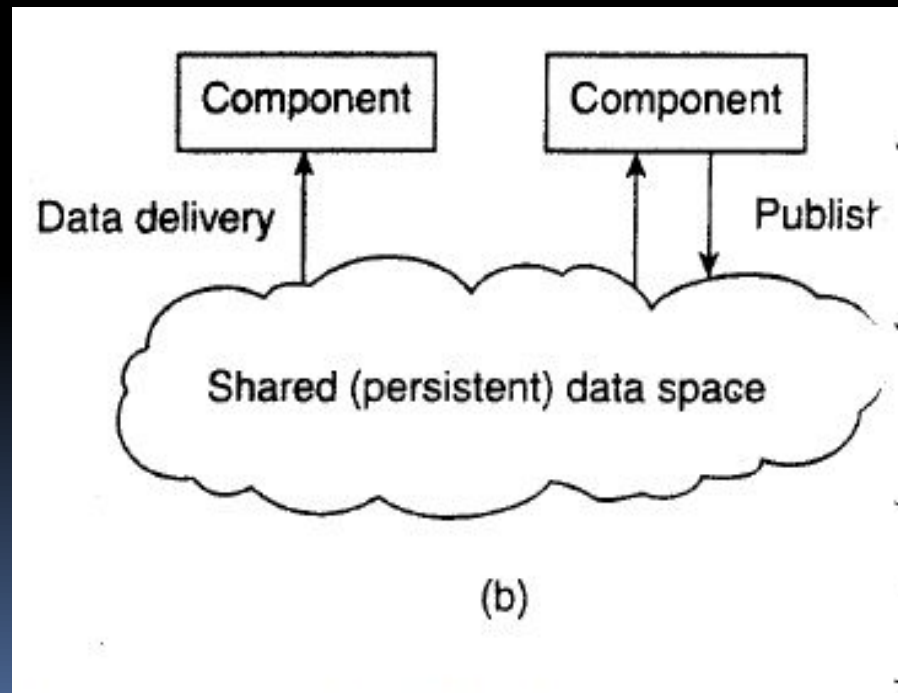
Encourages **modularity and reusability**.

Easier to model **real-world entities** as objects.



3. Data-centered architectures


- Evolve around the idea that processes communicate through a common (passive or active(backup)) repository





In a **data-centered architecture**, the main focus is on **shared data** that acts as the central point of the system.

All components (clients, servers, applications) interact **through the central data store** instead of directly communicating with each other.



The architecture ensures that data is **consistent, available, and synchronized** across the distributed system.




Working

All data is stored in a **central database/repository**.

Clients and applications query this repository.

Used in DBMS, data warehouses, version control systems (e.g., Git).





Components

Clients (Consumers/Users)

- Access or update the data.
- Example: A mobile app fetching bank account details.

Servers (Producers/Providers)

- Process data requests and manage the data store.
- Example: A transaction server updating balances.



Central Data Store (Repository / Blackboard)

- Stores all data in a consistent and structured way.
- Ensures concurrency control, integrity, and recovery.



Example

Banking systems → All branches update and read from a central database.

Airline reservation systems → Seat availability is updated in a central repository.

GitHub/Git → Central repository for code.

Big Data Systems → Hadoop Distributed File System (HDFS).





Advantages

Consistency → All users see the same updated data.

Central control → Easier to manage security, integrity, and backup.

Scalability → Centralized systems can be distributed with replication.



Reusability → Multiple applications can use the same data repository.

Event-based Architectures

An **event-based architecture** is a distributed design style where components communicate through **events** rather than direct calls.

Events are significant occurrences (e.g., “*Order Placed*”, “*Payment Successful*”).

Components are **loosely coupled**:

- **Event Producers (sources)** generate events.
- **Event Consumers (listeners/handlers)** subscribe and react to those events.
- An **Event Bus/Middleware** routes events between them.

Communication is **asynchronous** → producers and consumers do not block each other.



Components

Event Producer → Generates events (e.g., a sensor, web app, service).

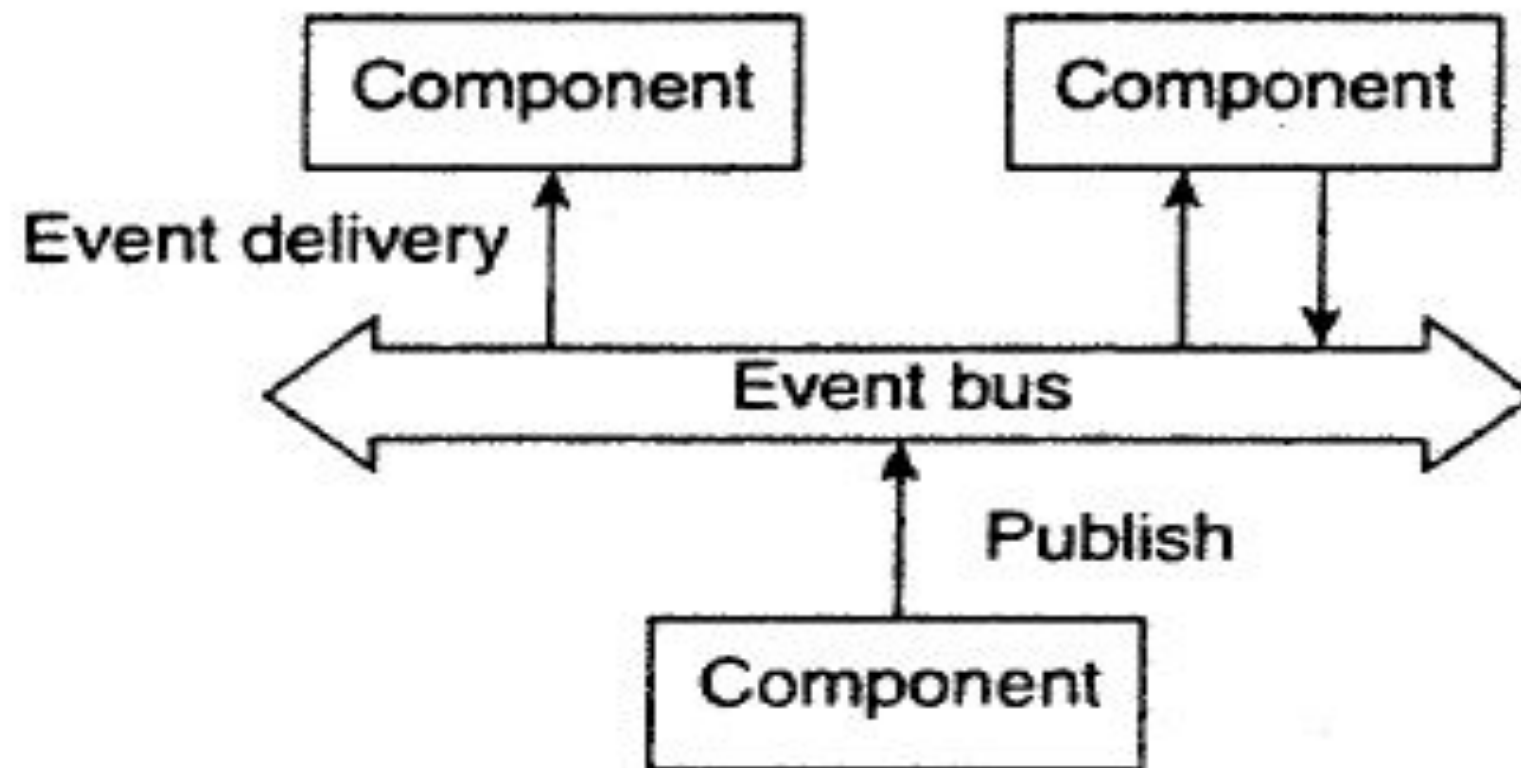
Event Consumer → Reacts to events (e.g., notification service, database updater).

Event Channel / Event Bus → Middleware for delivering events (e.g., Kafka, RabbitMQ).



Event Notification → Tells consumers that an event occurred.

Event Processing → Consumers perform actions based on the event.






Working

A producer generates an event (e.g., “*User registered*”).

The event is published to an **event channel/bus**.

All subscribed consumers receive the event and act accordingly (e.g., send welcome email, update user database, assign rewards).





Example


- **E-commerce:**

- Event: *Order Placed*.
- Consumers: Payment service, Inventory service, Shipping service, Notification service.

- **Social Media:**

- Event: *User uploaded photo*.
- Consumers: Newsfeed updater, Notification service, Recommendation engine.

- **IoT:**

- Event: *Temperature exceeds limit*.
 - Consumers: Cooling system, Alarm system, Dashboard updater.
- 




Advantages

Loose coupling → Producers and consumers don't depend on each other.

Scalability → Easy to add new consumers without modifying producers.

Real-time responsiveness → Suitable for live systems.




Flexibility → Multiple services can react differently to the same event.




Service Oriented Architecture

Service-Oriented Architecture (SOA) is a software design pattern where complex applications are built from independent, self-contained, and loosely coupled software components called services.

(For example, a retail enterprise might have an “Inventory Service,” “Customer Management Service,” and “Shipping Service”)




These services perform specific business or operational functions, communicate over a network using standardized protocols (like HTTP or SOAP), and can be reused across different applications.




The goal of SOA is to create flexible, scalable, and interoperable systems by promoting modularity, reusability, and ease of integration, especially in large enterprise environments.

SOA like a **restaurant**:

- Each section (kitchen, billing, delivery) provides a **service**.
 - Customers (applications) use these services without worrying about how they are implemented.
- 




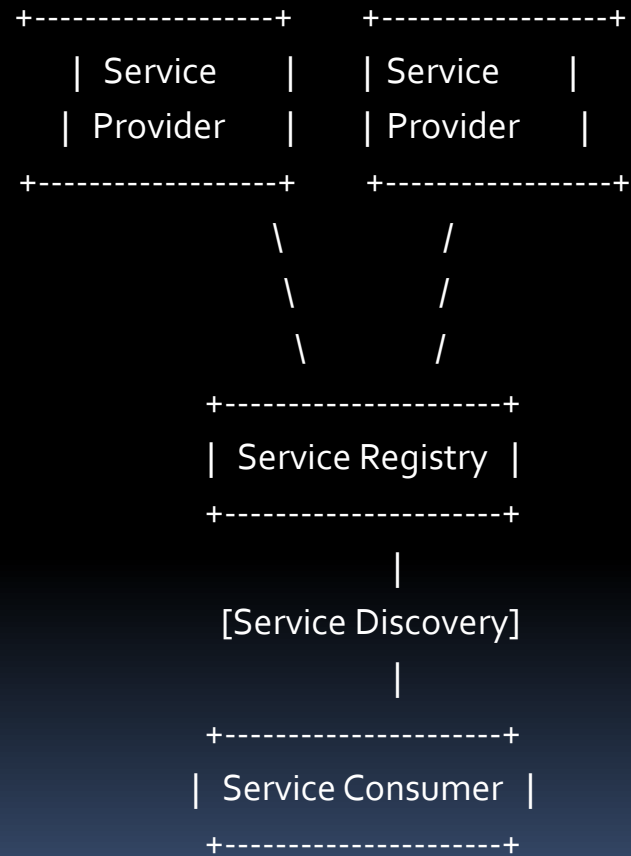
Components of SOA

1. **Service Provider** → Creates and publishes a service.
 2. **Service Registry** → Directory where available services are listed.
 3. **Service Consumer (Client)** → Discovers and invokes services from the registry.
 4. **Service Bus / Middleware** → Connects providers and consumers (sometimes called **Enterprise Service Bus, ESB**).
- 



Working


1. A **service provider** publishes its service (e.g., “Payment Service”) to a **registry**.
 2. A **service consumer** searches the registry for needed services.
 3. The consumer binds to the service and invokes it through standardized protocols.
- 






2.2 SYSTEM ARCHITECTURES

Deciding on software components, their interaction, and their placement leads to an instance of a software architecture, also called a system architecture





2.2 SYSTEM ARCHITECTURES

- 2.2.1 Centralized Architectures
 - 2.2.2 Decentralized Architectures
 - 2.2.3 Client - Server Architecture
 - 2.2.4 Peer to Peer Architecture
- 



CENTRALIZED NETWORK

WHAT IS CENTRALIZATION?

In a centralized network, there is a central authority that governs and handles the network.

ADVANTAGES

- Command chain
- Reduced costs
- Consistent output



DISADVANTAGES

- Not 100% Trustable
- Single point of failure
- Scalability limitation



DECENTRALIZED NETWORK

WHAT IS DECENTRALIZATION?

In a decentralized network, there is no central authority that governs and handles the network.

ADVANTAGES

- Full control
- Immutable data
- High security



DISADVANTAGES

- Costly
- Misuse of authority
- Volatility



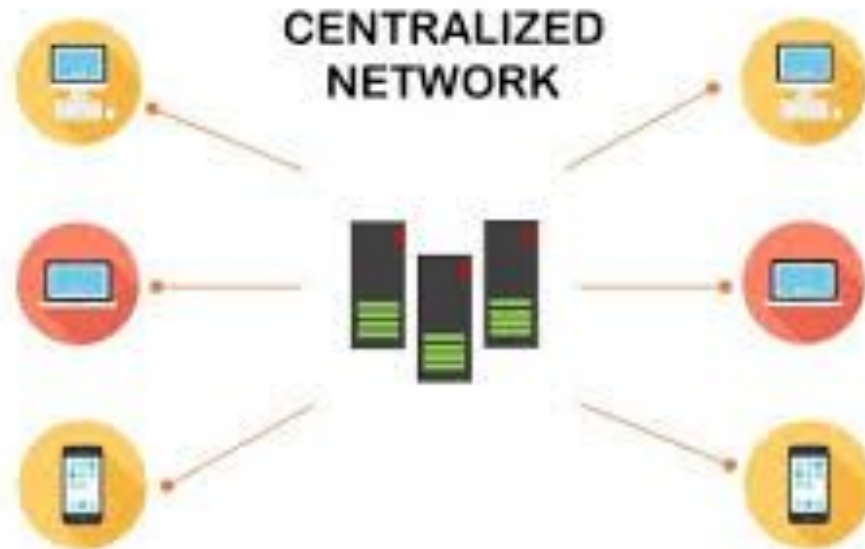
CENTRALIZED VS. DECENTRALIZED

	CENTRALIZED	DECENTRALIZED	
Third-Party Involvement	Yes	No	
Control	Full control stays with the central authority	Control stays with the user itself	
Hackable	More prone to hacks and data leaks	Less prone to hacks and data leaks	
Single Point of Failure	Yes	No	
Ease of Use	Intuitive and easy to use	Not easy to use	
Exchange Fees	Higher fees	Less fees	




2.2.1 Centralized Architecture

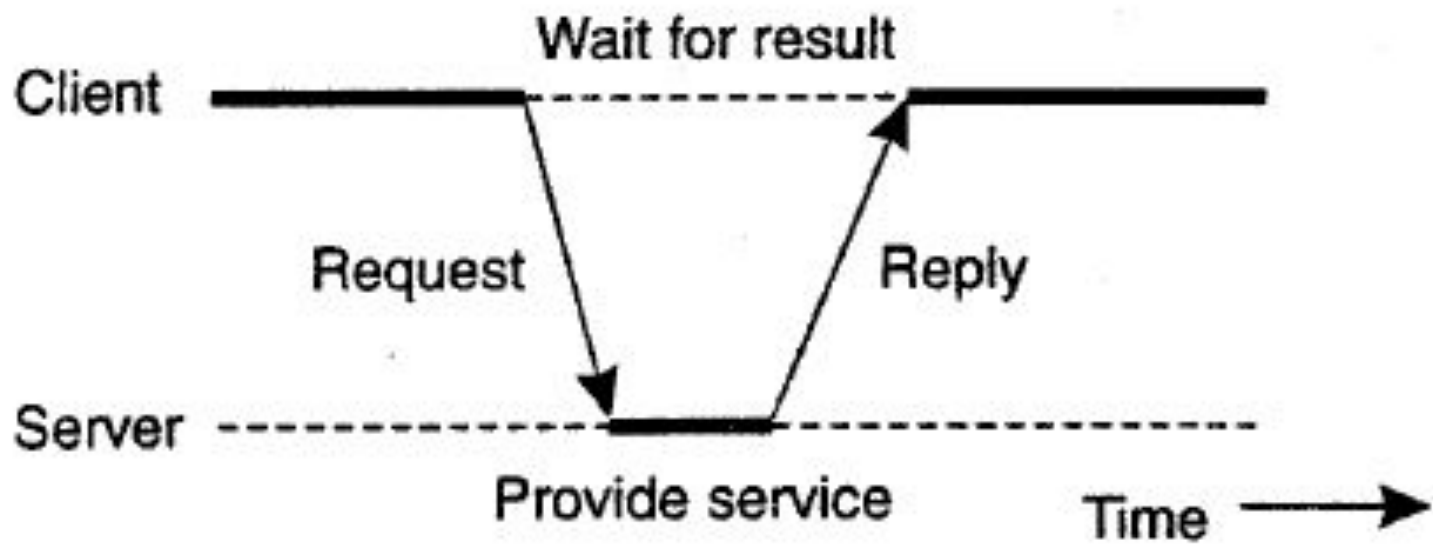
- A **centralized architecture** is a system design where **all major processing, control, and data storage** are managed by a **single central unit (server or mainframe)**. Other devices (clients or terminals) depend on this central system for resources, processing power, and decision-making.






Client/ Server

- A server provides a particular service to client.
 - 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.
- 






Characteristics

1. **Single Control Point:** All decisions and management happen in one central system.
 2. **Data Centralization:** Data is stored in one location (database/server).
 3. **Thin Clients:** Client devices have minimal processing power.
 4. **High Dependence:** System performance depends on the central node.
 5. **Simplified Management:** Easier to update and manage since everything is controlled centrally.
- 



Example

- **Banking Systems** (central server manages all branches' data)
 - **Airline Reservation Systems**
 - **University Database Systems**
 - **Mainframe-based Enterprise Applications**
- 



Advantages

- **Simplified Control & Administration:** Easy to manage security, updates, and backups.
- **Consistency of Data:** Since data resides in one place, duplication and inconsistency are minimized.
- **Efficient Resource Utilization:** Central server handles all heavy processing.
- **Ease of Maintenance:** Upgrades or fixes are done in one location.



Disadvantages

Single Point of Failure: If the central server crashes, the entire system stops.

Scalability Limitations: Server performance may degrade with many clients.



Network Dependency: Clients rely on constant network connectivity.

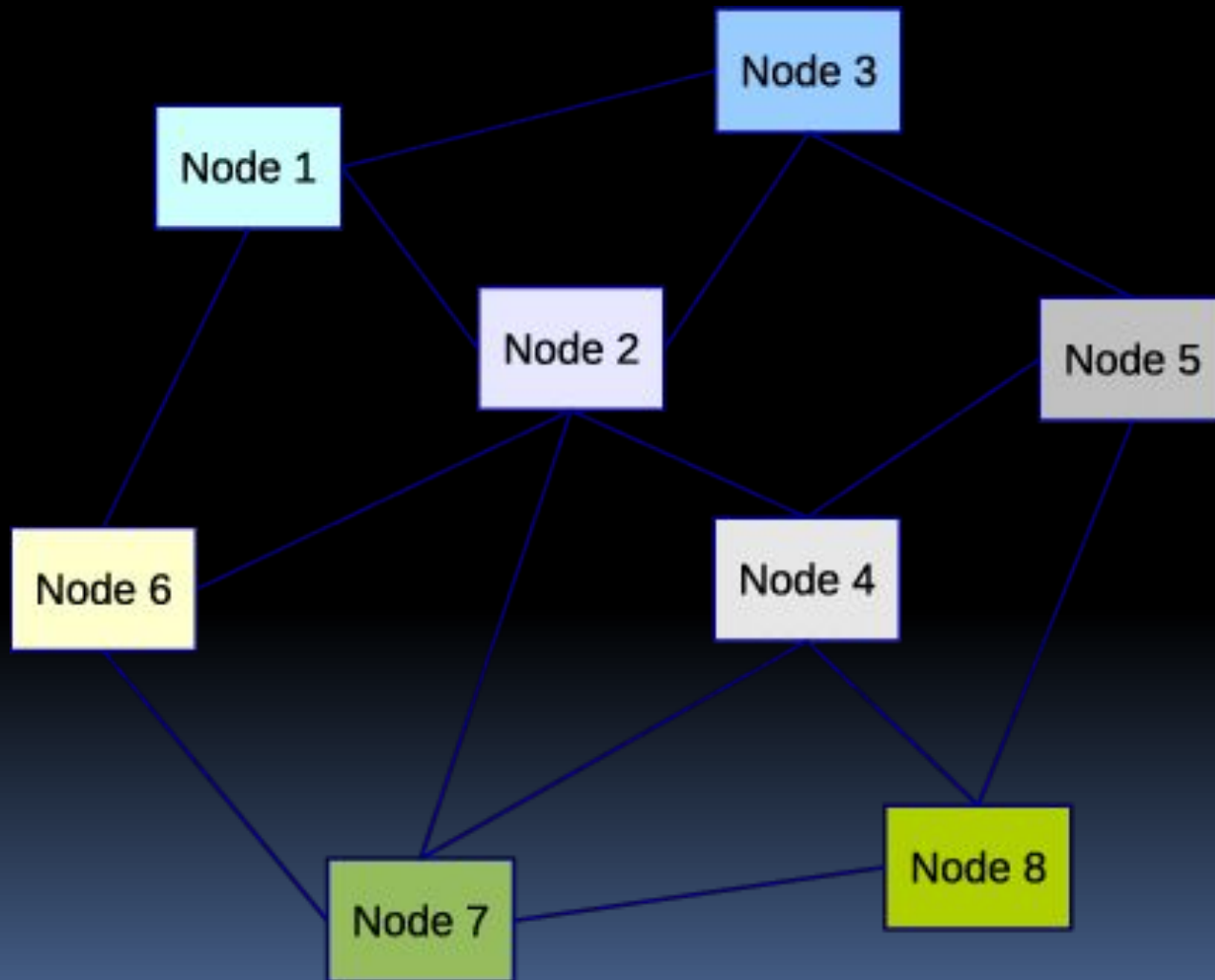
Potential Bottlenecks: Heavy traffic can slow down response time.

2.2.2 Decentralized Architecture

Decentralized Architecture refers to a system design in which control, decision-making, and management are distributed across multiple nodes, rather than being concentrated in a central authority or server.

This structure contrasts with centralized systems, where a single point of control governs the entire system.

In a decentralized architecture, there is no single failure point, which provides higher resilience, scalability, and fault tolerance.





Features of Decentralized Architecture

Distributed Control

Scalability

Redundancy and Fault Tolerance

Security



Transparency

Autonomy of Nodes




Example

Blockchain Networks (e.g., Bitcoin, Ethereum)

Peer-to-Peer (P2P) File Sharing (e.g., BitTorrent)


Decentralized Cloud Systems (e.g., IPFS, Storj)

Multi-Branch Banking Systems (each branch processes transactions locally but syncs data periodically)





Advantage

- **Reliability & Fault Tolerance:** Failure of one node does not affect the whole system.
 - **Scalability:** System performance can increase by adding more nodes.
 - **Reduced Bottlenecks:** No central server means less congestion.
 - **Autonomy:** Each node can make decisions locally.
 - **Security & Privacy:** Data is distributed, reducing the risk of total data compromise.
- 



Disadvantage



Complex Management: Harder to maintain and coordinate multiple nodes.

Data Consistency Issues: Synchronizing updates across nodes can be difficult.

Higher Cost: More resources and infrastructure required.

Communication Overhead: Increased traffic due to peer coordination.



- 
- 
- For example, distributed systems such as cloud computing, peer-to-peer networks, and client-server applications are overlay networks because their nodes run on top of the Internet
 - The Internet was originally built as an overlay upon the telephone network while today the telephone network is increasingly turning into an overlay network built on top of the Internet

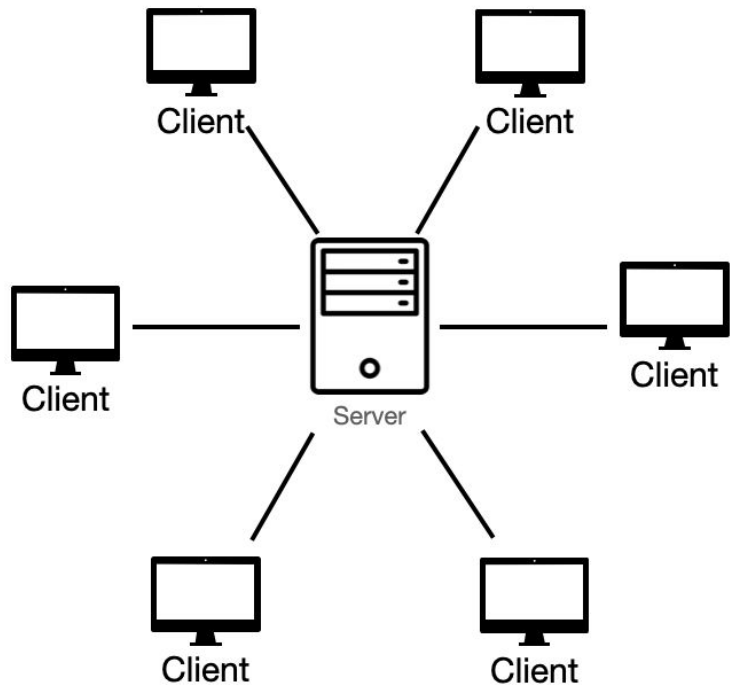


Client–Server Architecture

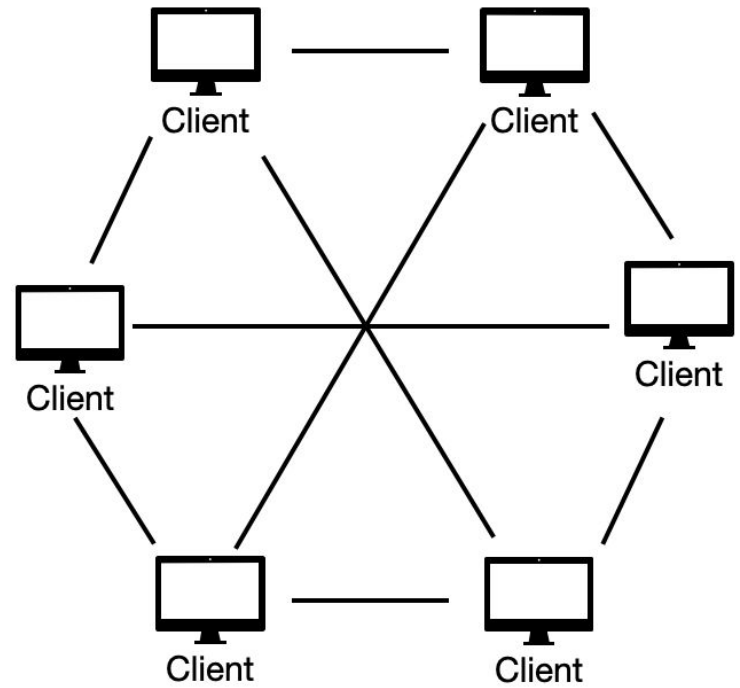
A **Client–Server Architecture** is a network design where **multiple client devices** (computers, smartphones, etc.) request and receive services from a **centralized server**.

The **server** provides resources or services (like data, applications, or files), and the **clients** use them.






Client Server Architecture



P2P Architecture



How It Works

- The **client** sends a request to the **server** (e.g., a web browser requesting a webpage).
 - The **server** processes the request and sends back a response (e.g., HTML data).
- 




Example

Web applications: Browser (client) ↔ Web server
(Apache/Nginx)

Email: Outlook (client) ↔ Mail server (SMTP/IMAP)

Databases: Application (client) ↔ MySQL/PostgreSQL (server)





Advantage

Centralized control and management

Easy to update or secure (since data is on the server)



Scalable with powerful server hardware



Disadvantage

Server overload if too many clients connect

Single point of failure (if server crashes, service stops)



Maintenance cost of server hardware/software



Peer-to-Peer (P2P) Model


In the **Peer-to-Peer model**, all nodes (peers) in the network act as **both clients and servers**.

Each peer can **request** and **provide** services or resources directly to other peers.






How It Works

- Peers communicate directly without a central server.
 - Each peer can share files, data, or computing power.
- 



Examples

- **File sharing:** BitTorrent, eMule
 - **Cryptocurrency networks:** Bitcoin, Ethereum
 - **VoIP applications:** Skype (earlier versions)
- 



Advantages

No central server — more resilient and fault-tolerant

Efficient sharing of resources (bandwidth, storage)




Scalable — more peers can mean more power



Disadvantages

Security and trust issues (no central authority)


Difficult to manage and monitor



Inconsistent performance (depends on peers' capability and uptime)




MIDDLEWARE ORGANIZATION

- In distributed system various heterogeneous devices are spread all over world.
 - Distributed system can achieve this consistency through a common layer to support the underlying hardware and OS.
 - This common layer is called Middleware.
- 

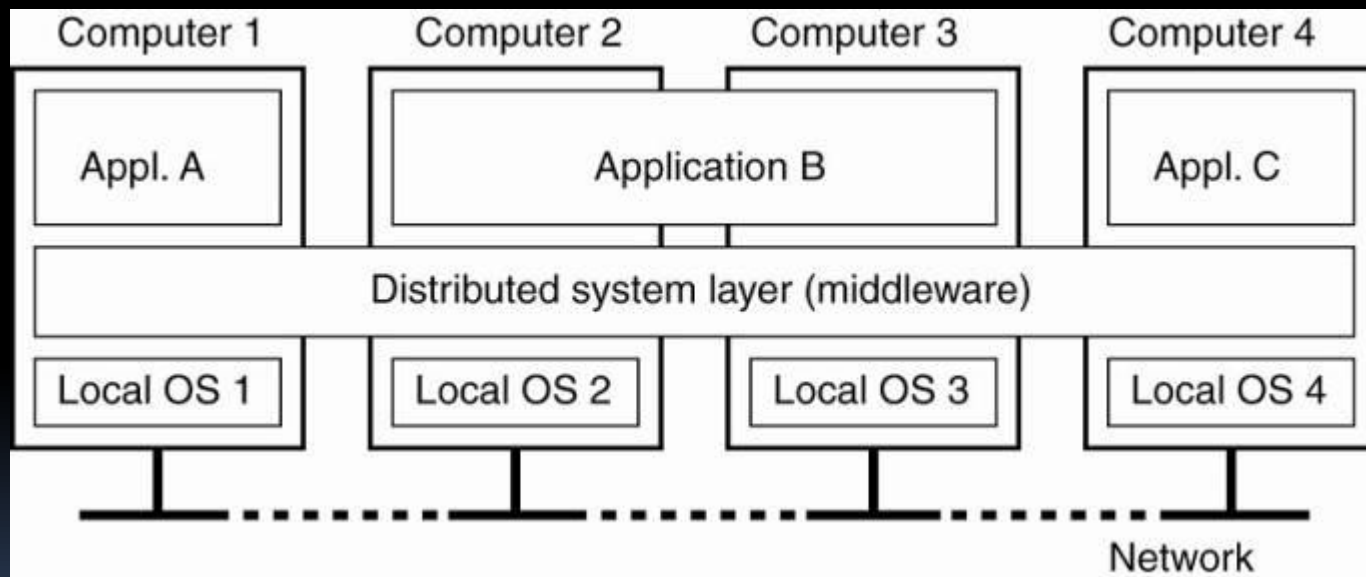


Middleware is software that acts as a bridge or intermediary between different software applications or components within a distributed computing system.

It enables communication and data management between disparate systems, helping them work together seamlessly.




Middleware typically operates "in the middle" between the operating system and the applications, providing a common platform for communication, authentication, data processing, and other services.





Function of Middleware

- Hides the details of distributed applications
 - Hides the heterogeneity of hardware, operating systems and protocols
 - Provides uniform and high-level interfaces used to make interoperable, reusable and portable applications
 - Provides a set of common services that enhances collaboration between applications
- 

Type of Middleware

Message-Oriented Middleware (MOM)

Supports sending and receiving messages between distributed systems asynchronously.

RabbitMQ, Apache Kafka, IBM MQ

Object Request Broker (ORB)

Allows programs to send requests and receive responses in an object-oriented way.

CORBA (Common Object Request Broker Architecture)

Database Middleware

Provides connectivity and access to databases.

ODBC (Open Database Connectivity), JDBC (Java Database Connectivity)

Remote Procedure Call (RPC) Middleware

Enables executing procedures on remote machines as if they were local.

Sun RPC, gRPC