# Types of System Patterns

## 1. What is a Centralized System?

- A centralized system is a system in which an individual, a group of people or a corporate entity holds the entire control over the functionality of the system.
- Online social applications like Facebook, Twitter, Quora are examples of centralized systems.
- We, end users don’t have a say in the architectural design, feature availability or the functionality of the applications. We don’t decide how the systems should operate. We have no control over the data.
- Corporate entities hold the rights to modify or delete our data without any permission.
- A centralized system has its risks, for instance, single point of failure. If the company goes out of business all our data is gone, for ever. And this has happened in the past.

---

**What is the Difference between Centralized & the Monolithic?**

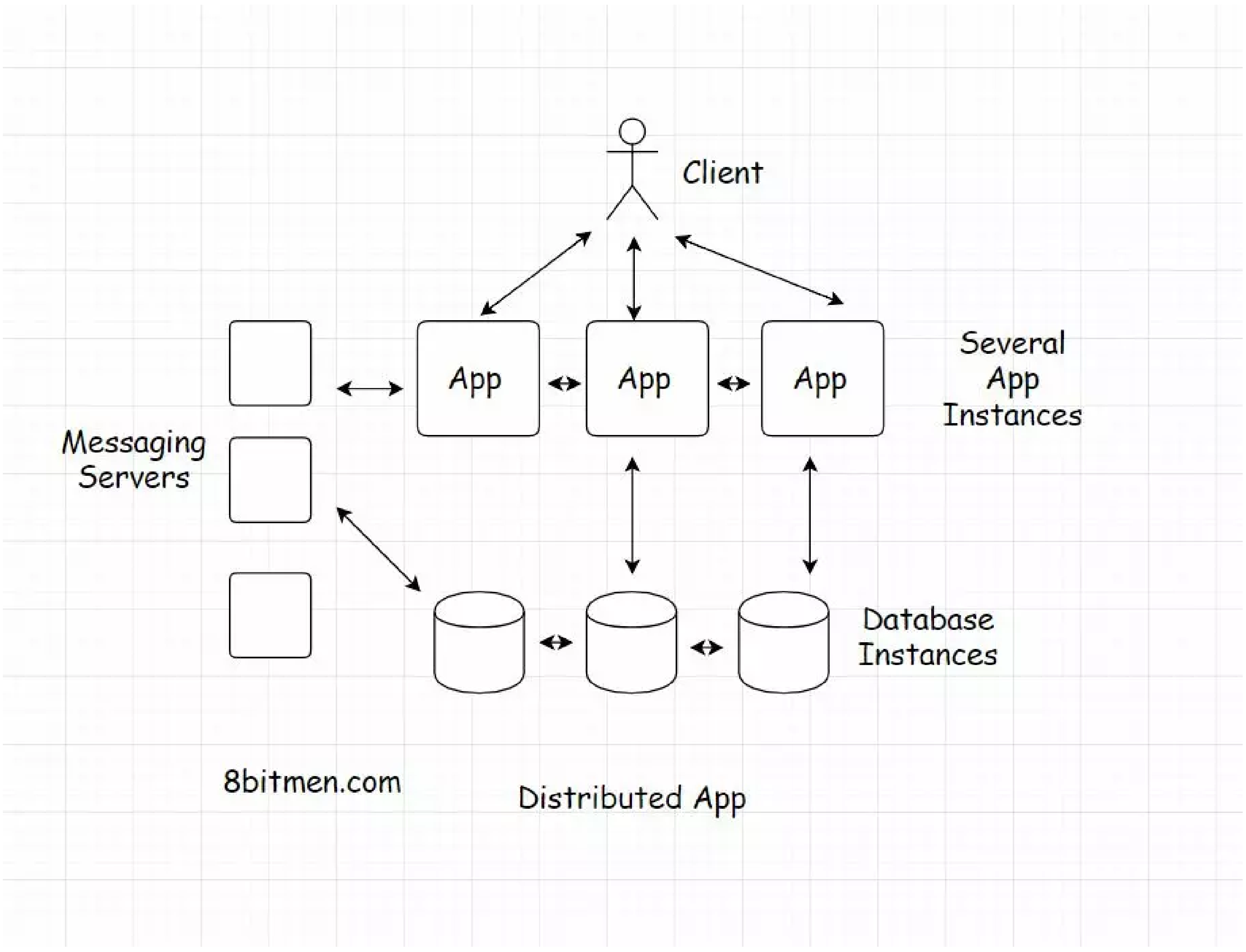
`The term centralized is used when we talk about control. Where does the control lie? Twitter, Facebook, as stated earlier hold all the control by themselves hence they are centralized systems. Unlike [peer to peer](https://scaleyourapp.com/p2p-peer-to-peer-networks-oversimplified-everything-you-should-know/) networks like BitTorrent or a cryptocurrency like Bitcoin, where the control lies equally with every user/node in the system/network.`

`On the other hand, monolithic is strictly used in the context of software architecture. It has nothing to do with the control.`

---

## 2.What is a Distributed System & the Need For it?

**A distributed system is a system which consists of several servers, a cluster of servers be it backend, messaging or database running together to perform one single task.**



- Several servers running together distribute the load amongst them reducing the latency of the app. They also help in tackling some of the critical issues like *single point of failure*, data loss prevention measures like *data replication* etc.

- Servers in data centres are also distributed geographically, nearer to the end-user, across continents, further reducing the latency of the app. I can go on about distributed systems for hours but that's not the primary focus of this article. I need to talk about the differences. I believe, this much amount of information is enough to get a gist of what it is.

---

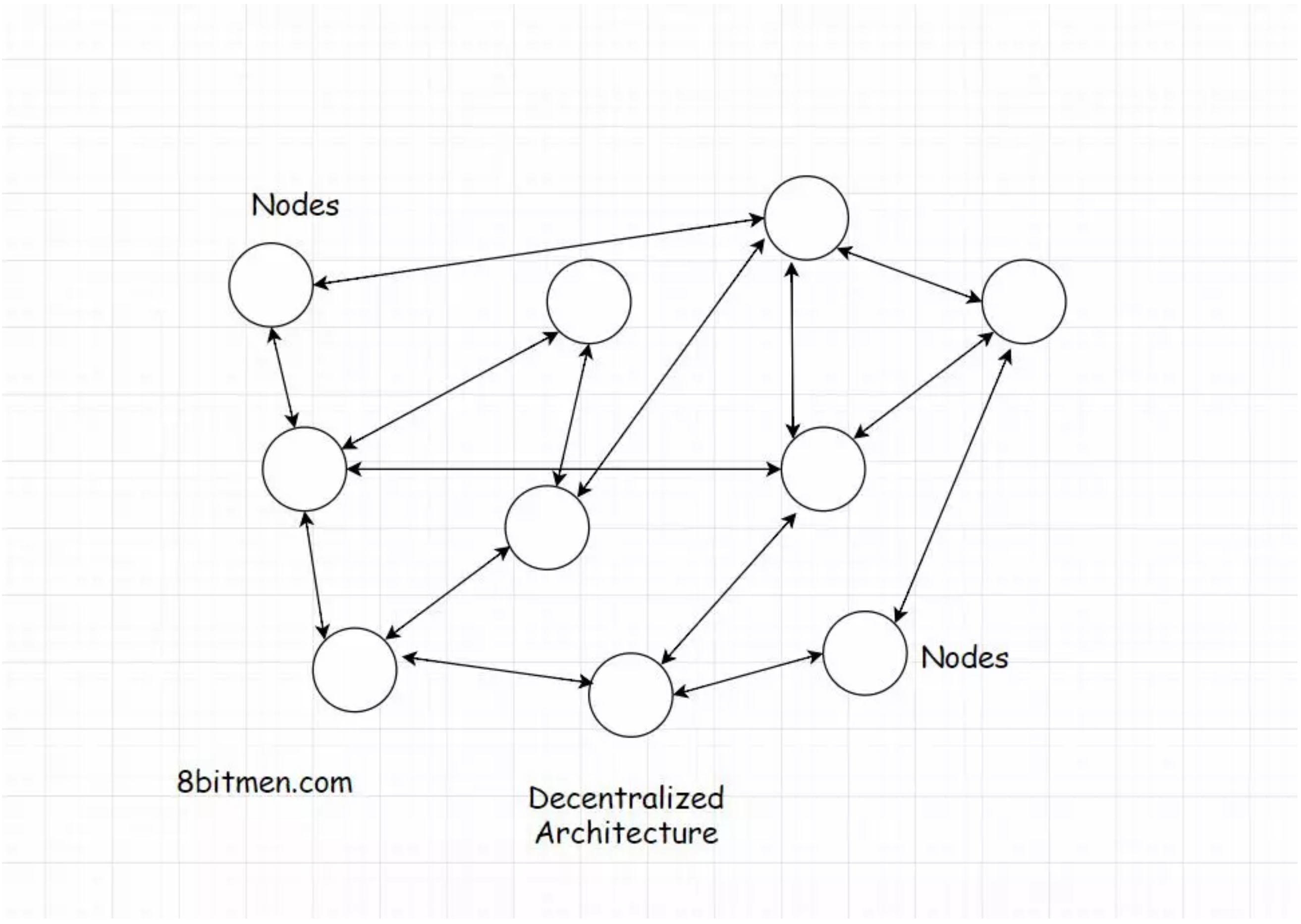
## 3.What is a Decentralized System?

- Decentralized simply means not centralized. *The control, as opposed to with a single entity, lies with the end users.* BitTorrent a peer to peer network is an ideal example of this.
  Where the peers are responsible for the availability of content on the network. More peers seed the data, higher is the availability & download speed.

- Decentralized systems just like the distributed architecture have no single points of failure. Even if several nodes go down, the network as a whole is still up.

- There is no single entity control, so there is zero possibility of the network going down anytime unless all the nodes go down simultaneously which is a rare possibility when we have systems connected all over the globe.

- This kind of architecture is almost infinitely scalable, unlike a centralized system in which scalability depends upon the resources of the organization in charge._



## Difference between a Decentralized and Distributed Network

- When we use the word decentralized it’s more in the context of control. Like a peer to peer network. It’s decentralized, no central authority. Every node acts as both client & server. On the other hand, when we speak of distributed, we do that primarily in the context of system scalability. *The difference is very subtle*. A distributed system can also be implemented in a centralized system to process computational heavy tasks.

- Speaking from my experience, I encountered the term distributed, in the context of a distributed cloud network, a cluster of servers hosted by centralized massive organizations like Google Cloud, Amazon Web services etc. They have a private distributed network of servers.

- I came across the term decentralized in the context of peer to peer networks, blockchain.

- Also, a distributed system comes along with several other features like data replication, intelligent fault tolerance policies, high availability etc.Their primary mission is to keep the system as a whole, running.

- Again decentralized, it primarily about the control.

---

#ARCITECTUREPATTERNS

## What is an Architectural Pattern?

> An **architectural pattern** is a general, reusable solution to a commonly occurring problem in software architecture within a given context. Architectural patterns are similar to software design pattern but have a broader scope.

1. **Layered pattern**
2. **Client-server pattern**
3. **Master-slave pattern**
4. **Pipe-filter pattern**
5. **Broker pattern**
6. **Peer-to-peer pattern**
7. **Event-bus pattern**
8. **Model-view-controller pattern**
9. **Blackboard pattern**
10. **Interpreter pattern**

---

## 1. Layered pattern

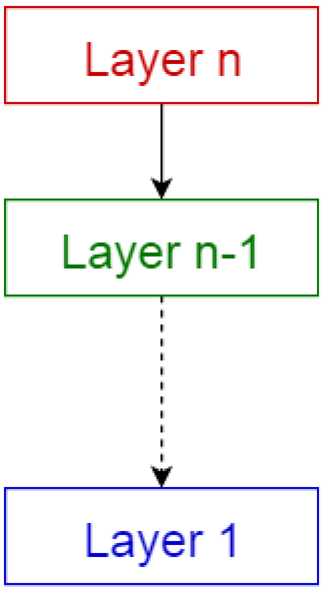
This pattern can be used to structure programs that can be decomposed into groups of subtasks, each of which is at a particular level of abstraction. Each layer provides services to

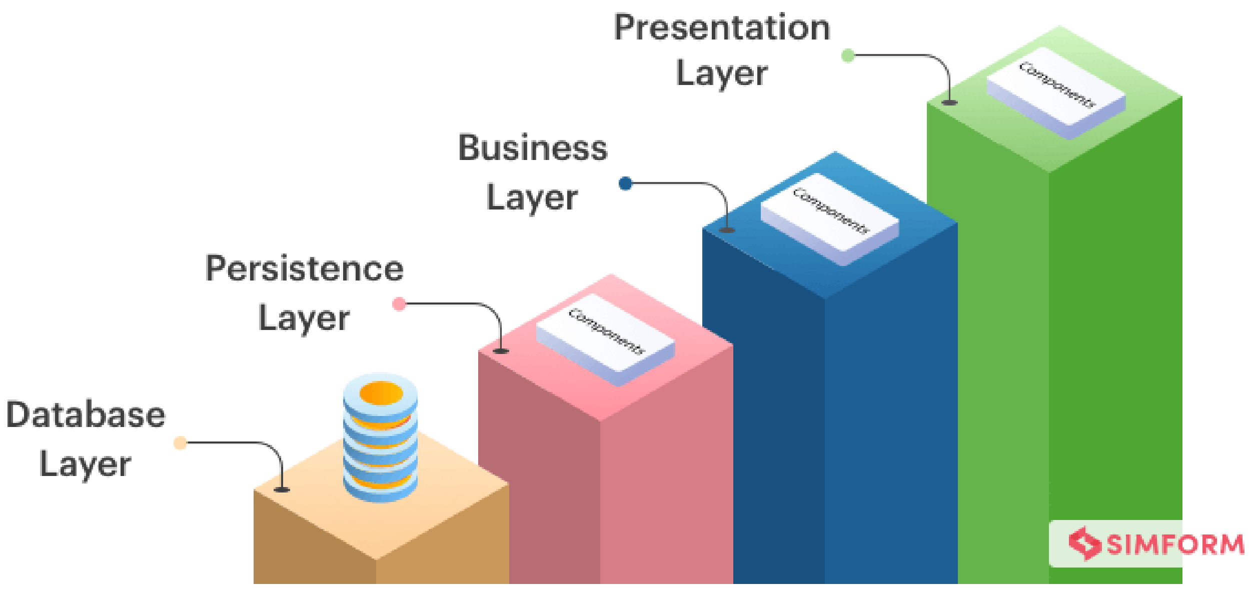
the next higher layer.
The most commonly found 4 layers of a general information system are as follows.

- **Presentation layer** (also known as **UI layer**)
- **Application layer** (also known as **service layer**)
- **Business logic layer** (also known as **domain layer**)
- **Data access layer** (also known as **persistence layer**)

## Usage

- General desktop applications.
- E commerce web applications.





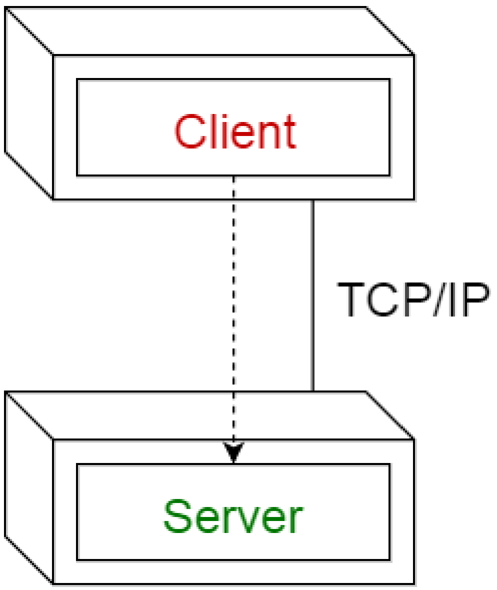
Layered pattern

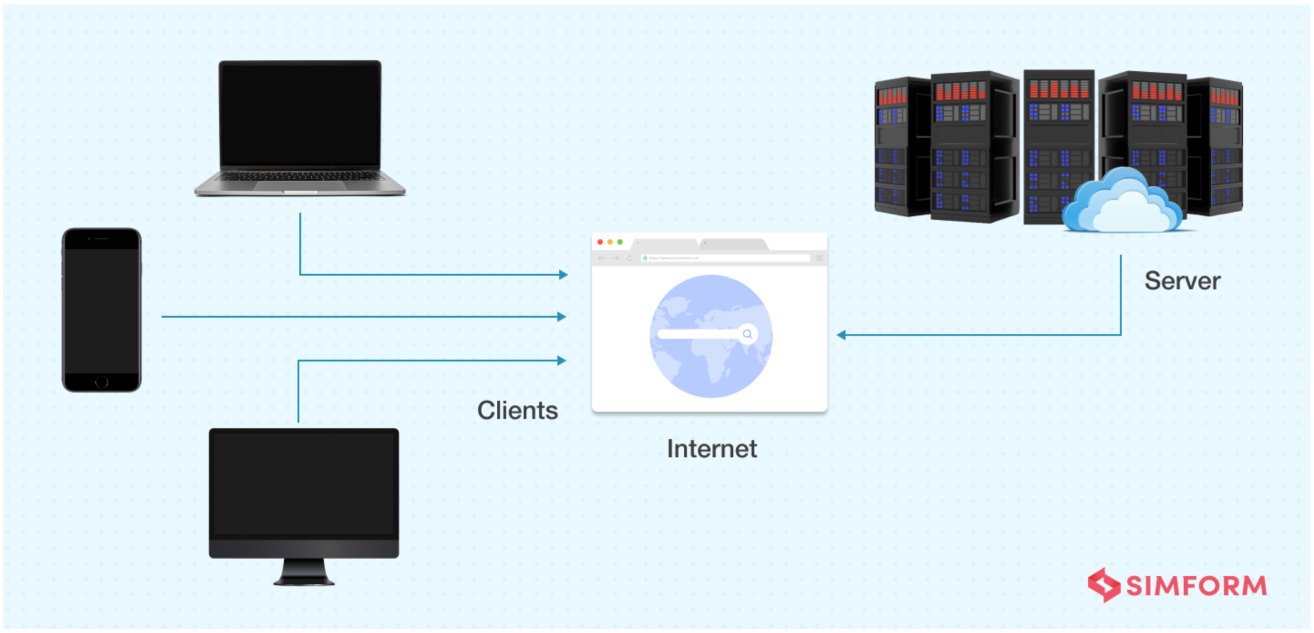
## 2. Client-server pattern

This pattern consists of two parties; a **server** and multiple **clients**. The server component will provide services to multiple client components. Clients request services from the server and the server provides relevant services to those clients. Furthermore, the server continues to listen to client requests.

### Usage

- Online applications such as email, document sharing and banking.





Client-server pattern

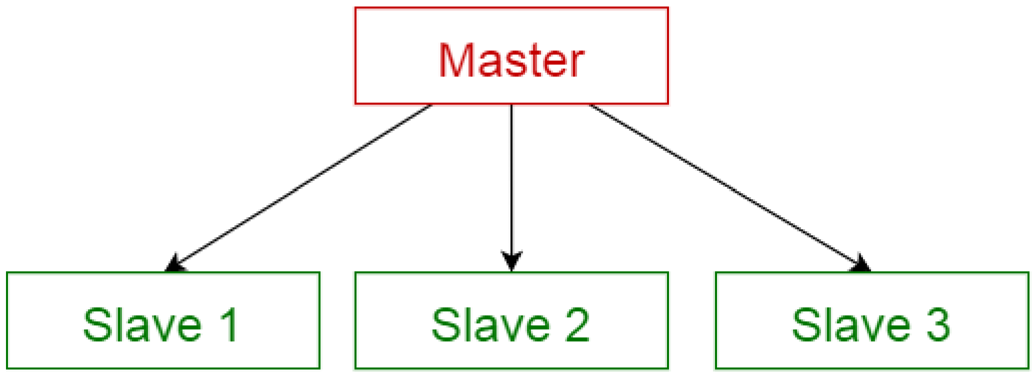
---

## 3. Master-slave pattern

This pattern consists of two parties; **master** and **slaves**. The master component distributes the work among identical slave components, and computes a final result from the results which the slaves return.

## Usage

- In database replication, the master database is regarded as the authoritative source, and the slave databases are synchronized to it.
- Peripherals connected to a bus in a computer system (master and slave drives).



Master-slave pattern

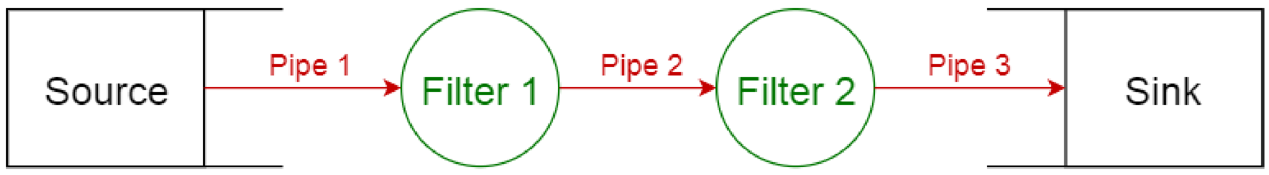
---

## 4. Pipe-filter pattern

This pattern can be used to structure systems which produce and process a stream of data. Each processing step is enclosed within a **filter** component. Data to be processed is passed through **pipes**. These pipes can be used for buffering or for synchronization purposes.

## Usage

- Compilers. The consecutive filters perform lexical analysis, parsing, semantic analysis, and code generation.
- Workflows in bioinformatics.
- It can be used for applications facilitating a simple, one-way data processing and transformation.



Pipe-filter pattern

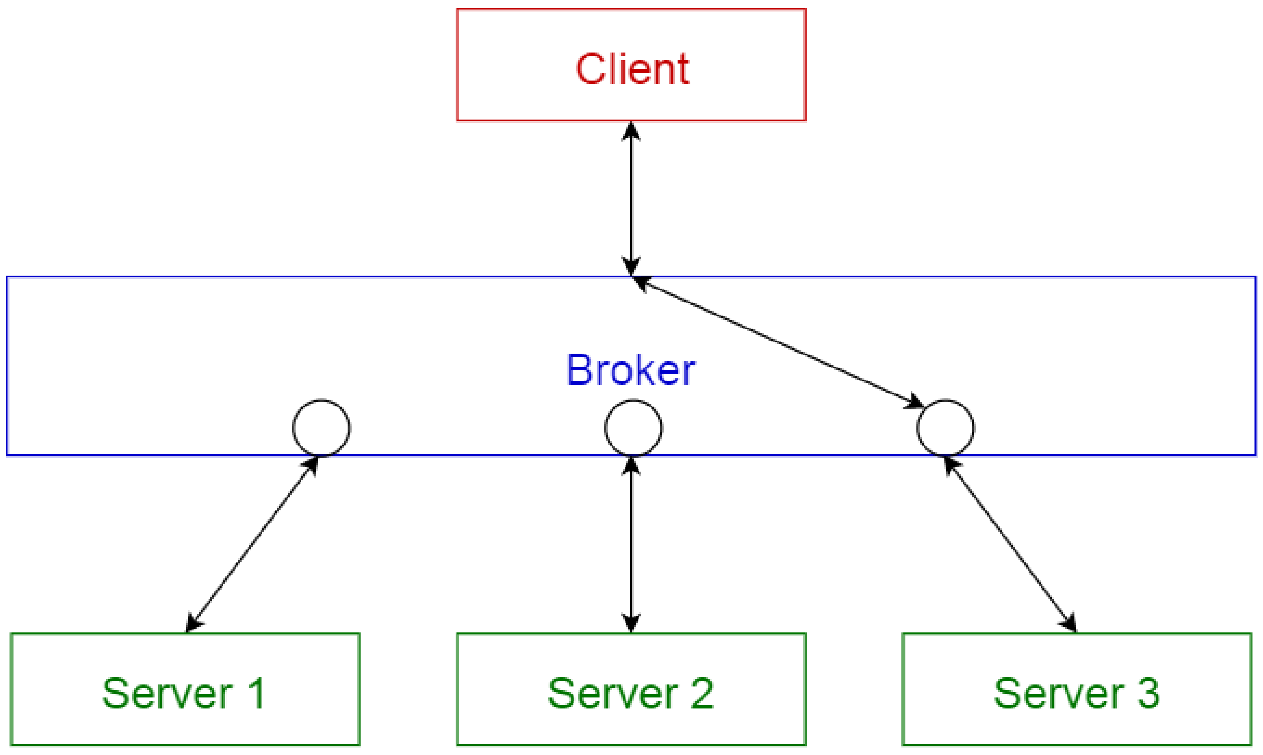
## 5. Broker pattern

This pattern is used to structure distributed systems with decoupled components. These components can interact with each other by remote service invocations.
A **broker** component is responsible for the coordination of communication among **components**.

Servers publish their capabilities (services and characteristics) to a broker. Clients request a service from the broker, and the broker then redirects the client to a suitable service from its registry.

### Usage

- Message broker software such as Apache ActiveMQ, Apache Kafka, RabbitMQ and JBoss Messaging.



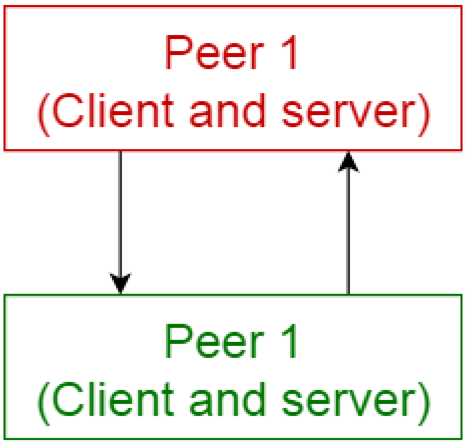
Broker pattern

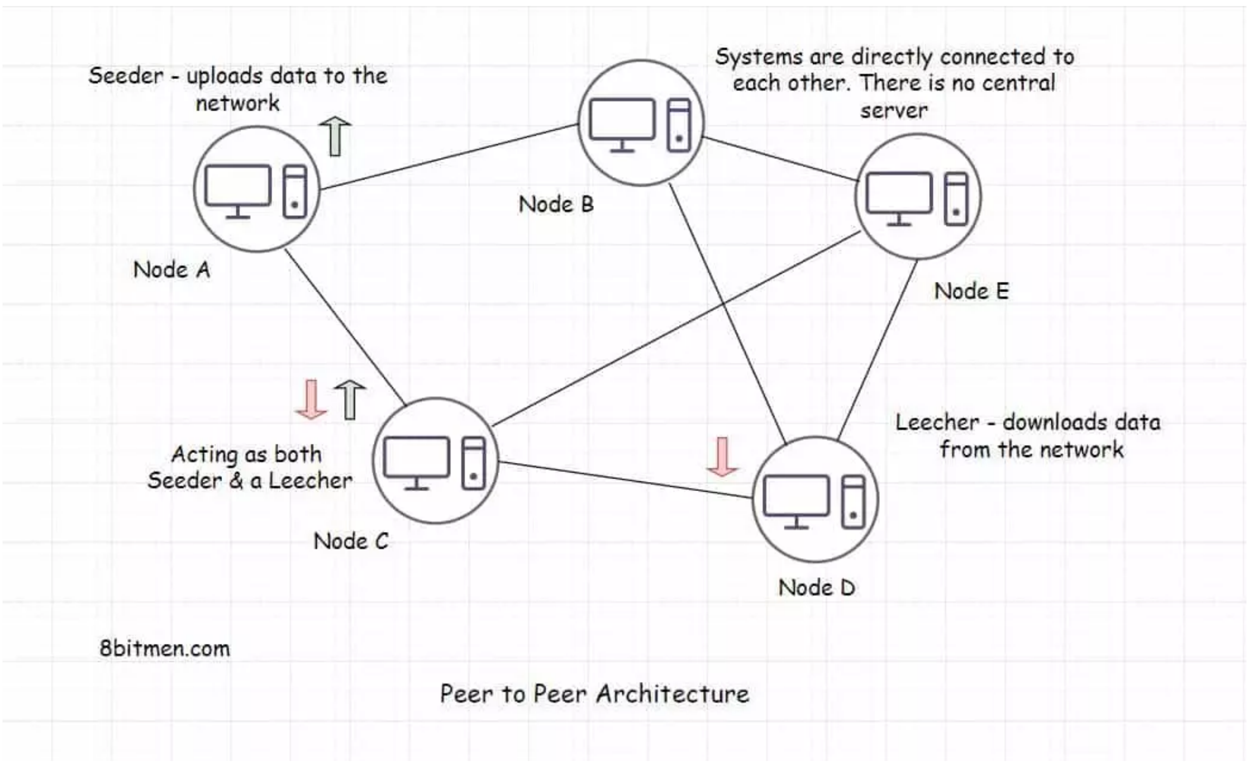
## 6. Peer-to-peer pattern

In this pattern, individual components are known as **peers**. Peers may function both as a **client**, requesting services from other peers, and as a **server**, providing services to other peers. A peer may act as a client or as a server or as both, and it can change its role dynamically with time.

## Usage

- File-sharing networks such as Gnutella and G2)
- Multimedia protocols such as P2PTV and PDTP.
- Cryptocurrency-based products such as Bitcoin and Blockchain




Seeder - uploads data to the network
Systems are directly connected to each other. There is no central server
Node A
Node B
Node E
Acting as both Seeder & a Leecher
Leecher - downloads data from the network
Node C
Node D
8bitmen.com
Peer to Peer Architecture

Peer-to-peer pattern

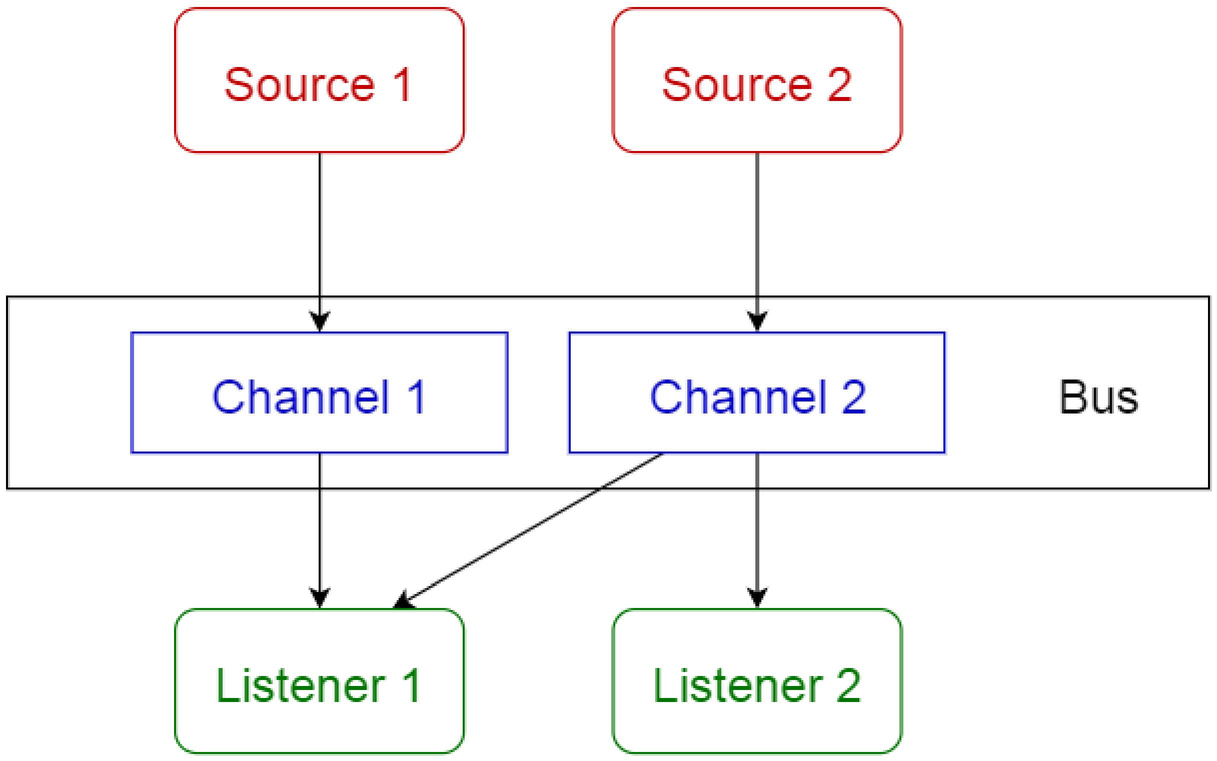
---

## 7. Event-bus pattern

This pattern primarily deals with events and has 4 major components; **event source**, **event listener**, **channel** and **event bus**. Sources publish messages to particular channels on an event bus. Listeners subscribe to particular channels. Listeners are notified of messages that are published to a channel to which they have subscribed before.

## Usage

- For applications where individual data blocks interact with only a few modules.
- Helps with user interfaces.
- Android development
- Notification services



Event-bus pattern

---

## 8. Model-view-controller pattern

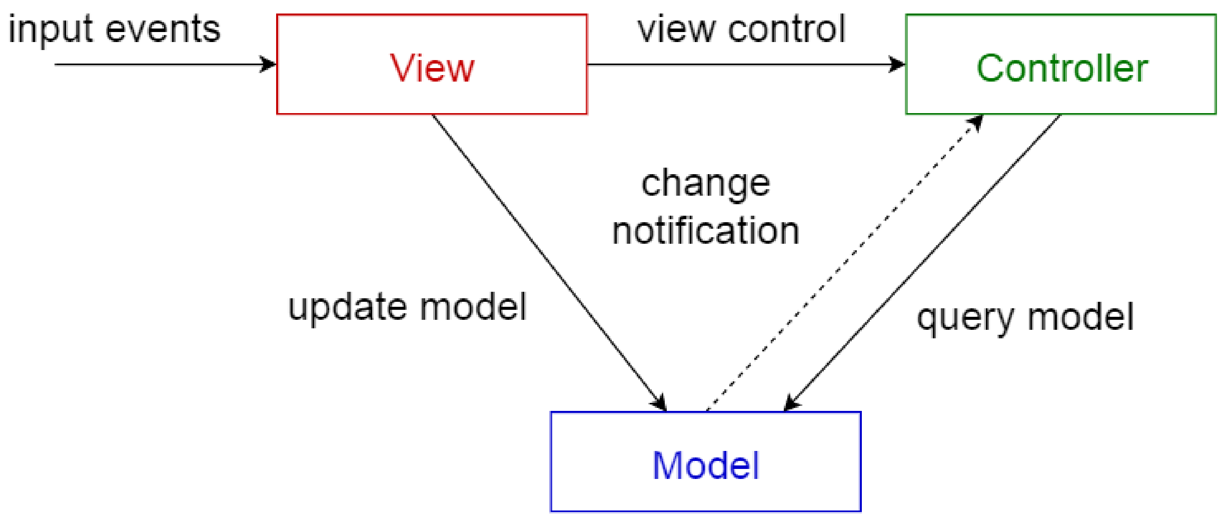
This pattern, also known as MVC pattern, divides an interactive application in to 3 parts as,

1. **model** — contains the core functionality and data
2. **view** — displays the information to the user (more than one view may be defined)
3. **controller** — handles the input from the user

This is done to separate internal representations of information from the ways information is presented to, and accepted from, the user. It decouples components and allows efficient code reuse.

### Usage

- Architecture for World Wide Web applications in major programming languages.
- Web frameworks such as Django and Rails.



Model-view-controller pattern

---

## 9. Blackboard pattern

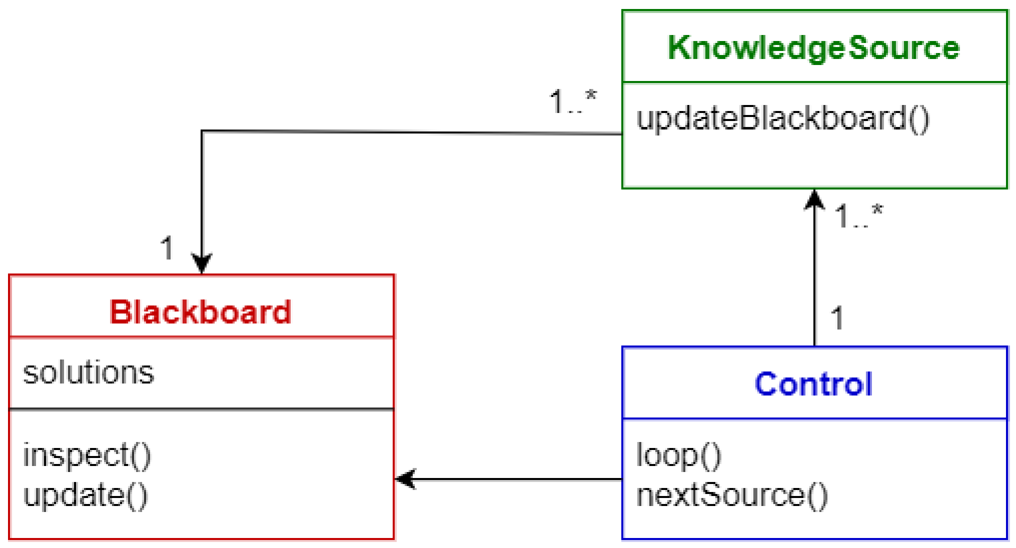
This pattern is useful for problems for which no deterministic solution strategies are known. The blackboard pattern consists of 3 main components.

- **blackboard** — a structured global memory containing objects from the solution space
- **knowledge source** — specialized modules with their own representation
- **control component** — selects, configures and executes modules.

All the components have access to the blackboard. Components may produce new data objects that are added to the blackboard. Components look for particular kinds of data on the blackboard, and may find these by pattern matching with the existing knowledge source.

### Usage

- Speech recognition
- Vehicle identification and tracking
- Protein structure identification
- Sonar signals interpretation.



Blackboard pattern

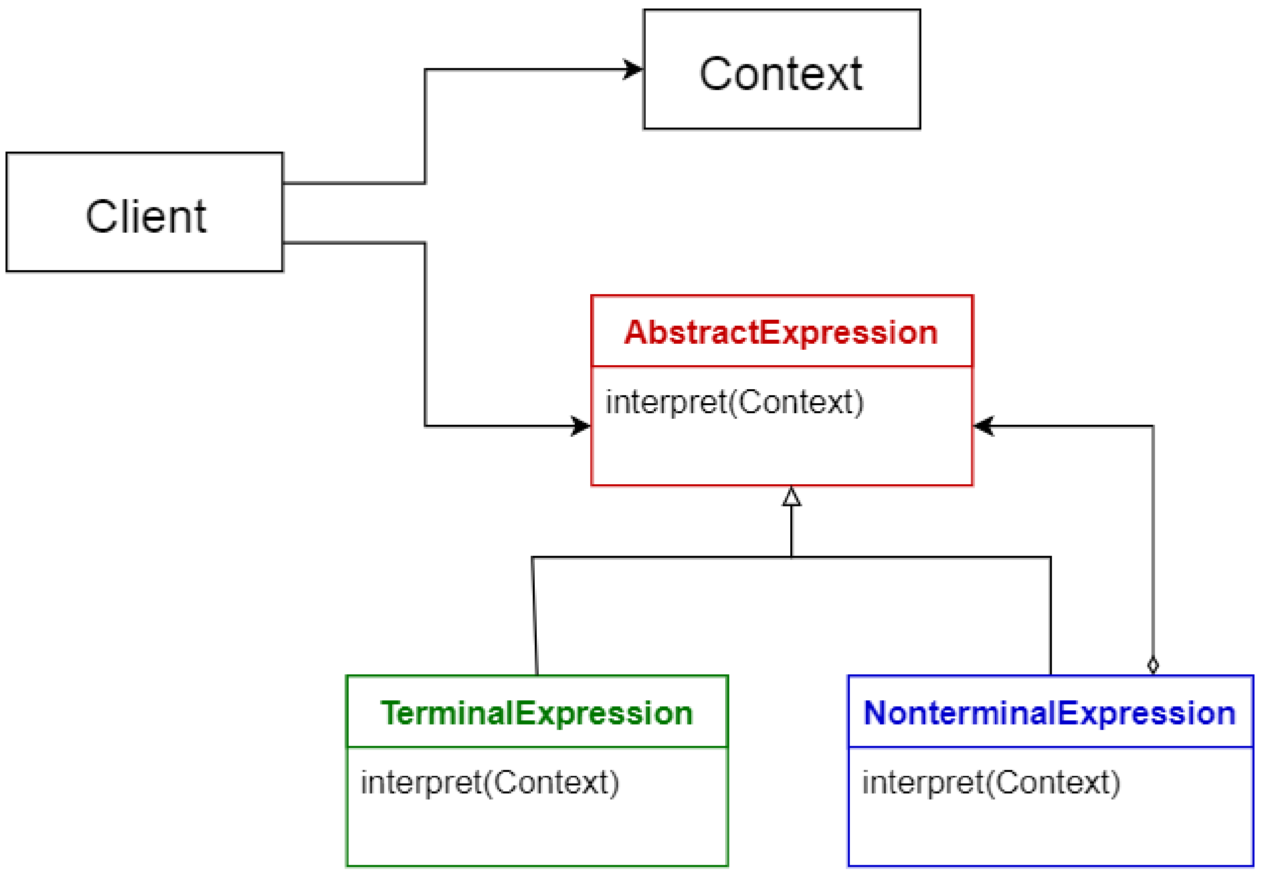
---

## 10. Interpreter pattern

This pattern is used for designing a component that interprets programs written in a dedicated language. It mainly specifies how to evaluate lines of programs, known as sentences or expressions written in a particular language. The basic idea is to have a class for each symbol of the language.

### Usage

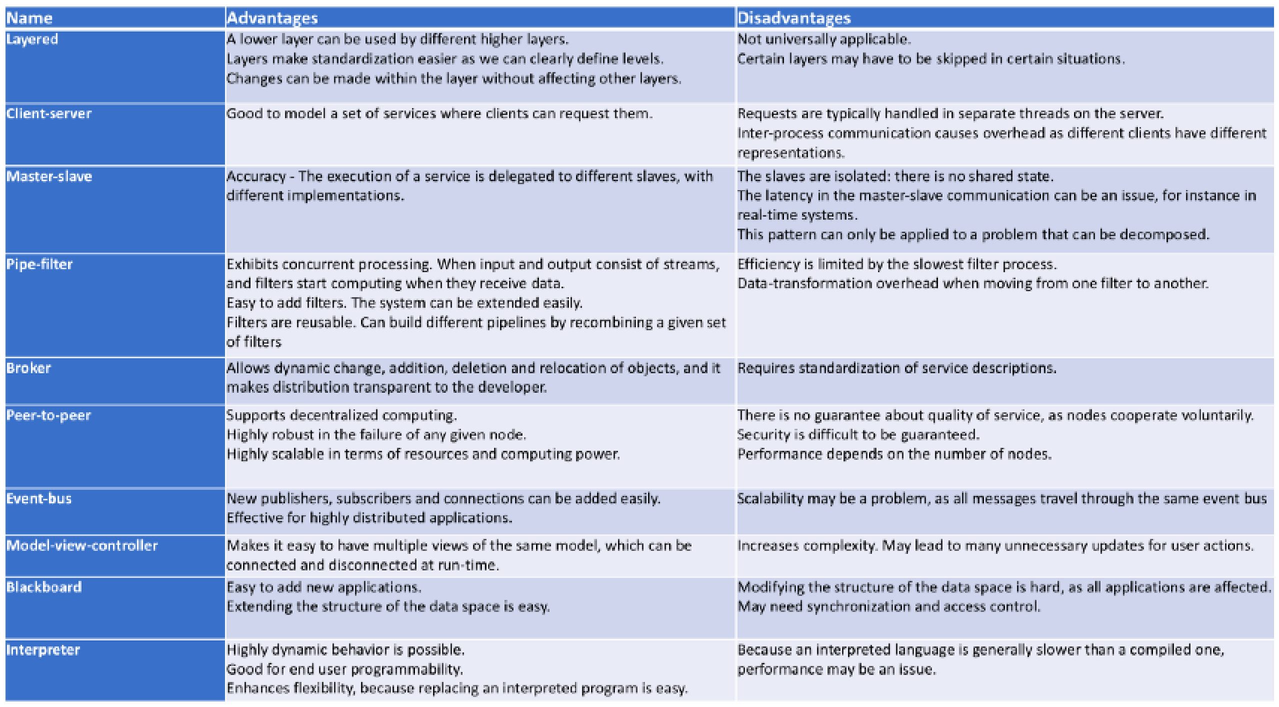
- Database query languages such as SQL.
- Languages used to describe communication protocols.



Interpreter pattern

## Advantage and Disadvantage Comparison of Architectural Patterns

The table given below summarizes the pros and cons of each architectural pattern.



| Name | Advantages | Disadvantages |
|---|---|---|
| Layered | A lower layer can be used by different higher layers.<br>Layers make standardization easier as we can clearly define levels.<br>Changes can be made within the layer without affecting other layers. | Not universally applicable.<br>Certain layers may have to be skipped in certain situations. |
| Client-server | Good to model a set of services where clients can request them. | Requests are typically handled in separate threads on the server.<br>Inter-process communication causes overhead as different clients have different representations. |
| Master-slave | Accuracy - The execution of a service is delegated to different slaves, with different implementations. | The slaves are isolated: there is no shared state.<br>The latency in the master-slave communication can be an issue, for instance in real-time systems.<br>This pattern can only be applied to a problem that can be decomposed. |
| Pipe-filter | Exhibits concurrent processing. When input and output consist of streams, and filters start computing when they receive data.<br>Easy to add filters. The system can be extended easily.<br>Filters are reusable. Can build different pipelines by recombining a given set of filters | Efficiency is limited by the slowest filter process.<br>Data-transformation overhead when moving from one filter to another. |
| Broker | Allows dynamic change, addition, deletion and relocation of objects, and it makes distribution transparent to the developer. | Requires standardization of service descriptions. |
| Peer-to-peer | Supports decentralized computing.<br>Highly robust in the failure of any given node.<br>Highly scalable in terms of resources and computing power. | There is no guarantee about quality of service, as nodes cooperate voluntarily.<br>Security is difficult to be guaranteed.<br>Performance depends on the number of nodes. |
| Event-bus | New publishers, subscribers and connections can be added easily.<br>Effective for highly distributed applications. | Scalability may be a problem, as all messages travel through the same event bus |
| Model-view-controller | Makes it easy to have multiple views of the same model, which can be connected and disconnected at run-time. | Increases complexity. May lead to many unnecessary updates for user actions. |
| Blackboard | Easy to add new applications.<br>Extending the structure of the data space is easy. | Modifying the structure of the data space is hard, as all applications are affected.<br>May need synchronization and access control. |
| Interpreter | Highly dynamic behavior is possible.<br>Good for end user programmability.<br>Enhances flexibility, because replacing an interpreted program is easy. | Because an interpreted language is generally slower than a compiled one, performance may be an issue. |

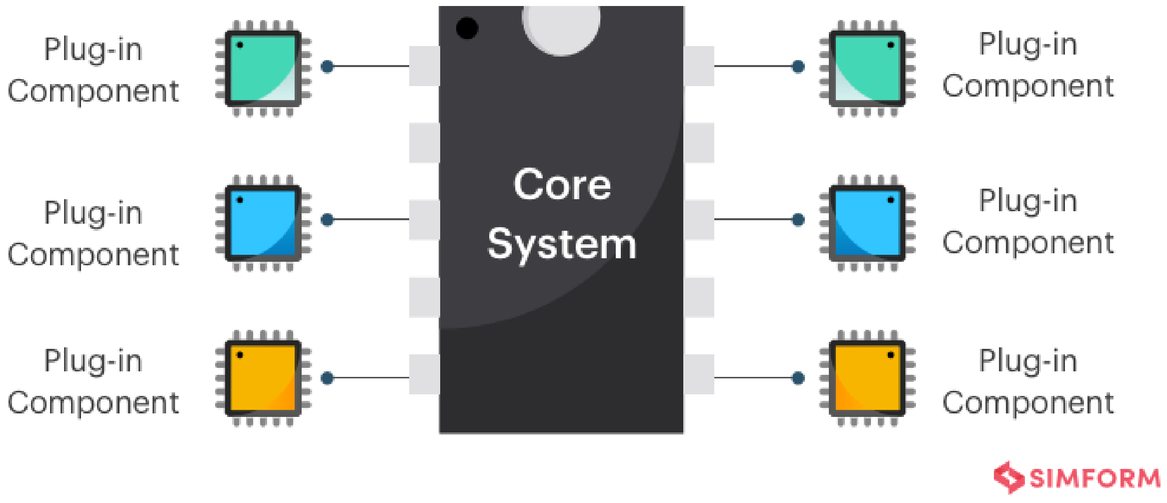
## Microkernel Architecture Pattern

- This architecture pattern consists of two types of components – a core system and several plug-in modules. While the core system works on minimal functionality to keep the system operational, the plug-in modules are independent components with specialized processing.

```
Taking the example of a task scheduler application, the microkernel contains all the
logic for scheduling and triggering tasks, while the plug-ins contain specific
tasks. As long as the plug-ins adhere to a predefined API, the microkernel can
trigger them without having to know the implementation details.
```

**Usage:**

- Applications that have a clear segmentation between basic routines and higher-order rules.
- Applications that have a fixed set of core routines and dynamic set of rule that needs frequent updates.



## Microservices Architecture Pattern(IMPORTANT)

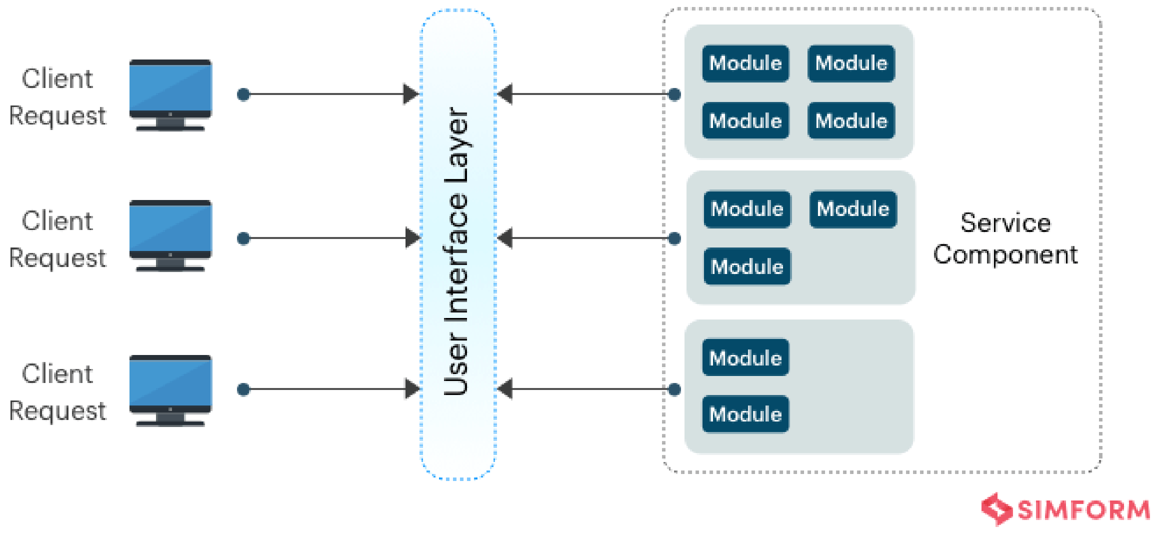
- Microservices architecture pattern is seen as a viable alternative to monolithic applications and service-oriented architectures. The components are deployed as separate units through an effective, streamlined delivery pipeline. The pattern's benefits are enhanced scalability and a high degree of decoupling within the application.

- Owing to its decoupled and independent characteristics, the components are accessed through a remote access protocol. Moreover, the same components can be separately developed, deployed, and tested without interdependency on any other service component.

```
Netflix is one of the early adopters of the microservice architecture pattern. The
architecture allowed the engineering team to work in small teams responsible for the
end-to-end development of hundreds of microservices. These microservices work
together to stream digital entertainment to millions of Netflix customers every day.
```

**Usage:**

- Businesses and web applications that require rapid development.
- Websites with small components, data centers with well-defined boundaries, and remote teams globally.


Client Request
Client Request
Client Request
User Interface Layer
Module
Module
Module
Module
Module
Module
Module
Module
Module
Service Component
SIMFORM