Distributed Systems

- 07-04-2022 - DK
- 08-04-2022 - INT
-

Agenda

- Distributed Systems
- Asynchronous/Synchronous - Communication
- MVC
- Design Patterns

Tue Hellstern © 2022

Distributed Systems

Distributed systems vs. Distributed computing

Distributed system

A set of computers which are **independent** and connected with an interconnection network. Components are **running** on **different machines** that communicate via messages and work together towards a single end.

Distributed computing

A method of computer processing in which different parts of a computer program are run on two or more computers that are communicating with each other over a network. Allows for better performance due to its simple scalability and builtin concurrency.
It also allows for flexible and resilient application that doesn’t have a single point of failure.

Services of distributed systems

Services of distributed systems

Mail Services

Games and Multimedia

Remote logging

File transferring and sharing

The simplest, and most *relatable*, example of a distributed system would be

The Internet!

Same concept can be applied to a plethora of problems - *machine learning, real time streaming* - anywhere there is a need for large-scale compute then distributed systems are a must.

Pros of distributed systems

- **Fault tolerance** - The system’s tasks are distributed across multiple machines, that means that in the event of a machine failure, another agent is available for the system to delegate the work to instead. **This prevents a single source of failure.**
- **Scalability** - Distributed systems follow best practices for scaling in that they are focused on *horizontally* scaling loads rather than *vertically*. This means that as the load increases, you simply add more machines to your system to do the work rather than increasing the memory and CPU of the old machines to handle the larger input.
- **Efficiency** - Since the system can be more performant than a *non-distributed* system, you have the flexibility to use fewer resources to get the same task done, even removing machines reactively to the input. This allows for a better cost fitting to usage in the cloud.

Cons of distributed systems

- **Messaging** - Since there are different machines processing at the same time, there is a large risk of overhead necessary to synchronize them and keep all the applications running appropriately and the data consistent across the network.
- **Reliability** - Fault tolerance ia a benefit of distributed systems, but the network management is often a behemoth task that is tuned particularly to your application and thus can be sensitive to changes or outages in the network.
- **Maintenance** - Running an application where the number of machines scales requires new architectural and managerial considerations. Traffic needs to be directed correctly to machines via a load balancer of some kind and every machine needs its own application and infrastructure monitoring, logging, delivery, and testing.

Asynchronous

Synchronous

Communication

Asynchronous communication

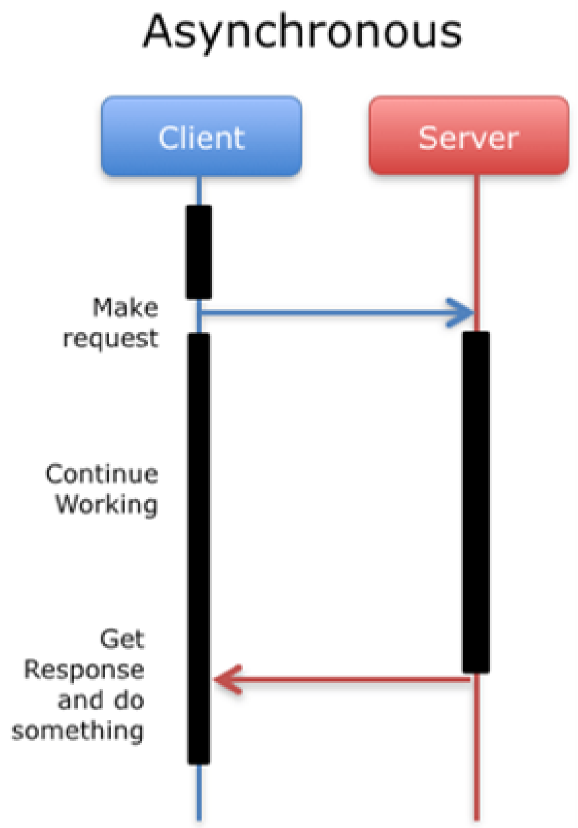
Sender is **not blocked**, the process may, **after sending** the message, **continue** immediately

Answers are optional

- The sender receives on occasion the result asynchronously
- The sender gets active on occasion
- Implementation uses in general queues

Features

- It is more complicated to implement, but more efficient
- **Loose coupling** of processes
- Lower error dependency
- Receiver does not need to be available to receive



Synchronous communication

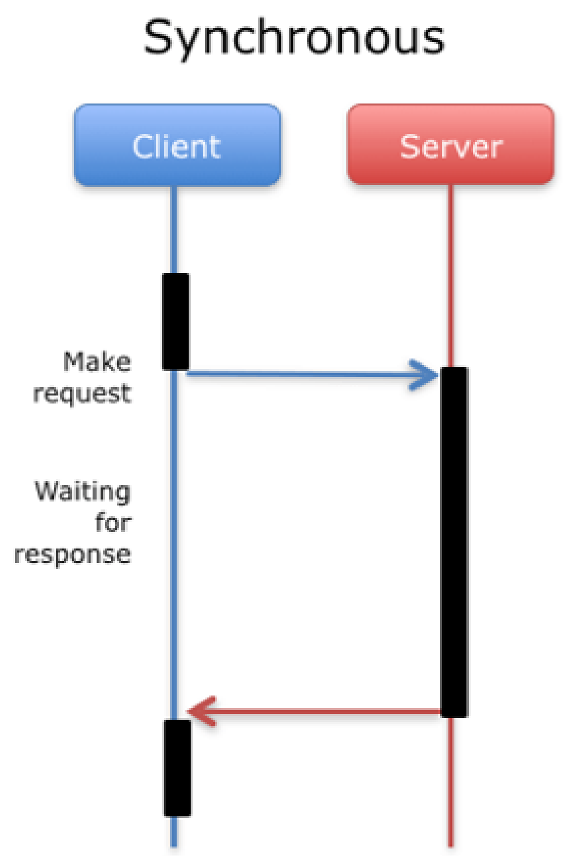
Transmitter and **receiver block** when executing the **Send** or **Receive** operation

Features

- Tight coupling between the transmitter and receiver with all its advantages and disadvantages
- High dependence especially in case of failure

Prerequisite

- Secure and fast network connections are available
- Receiving process is available



False assumptions

False assumptions

Developers making programs **sometimes make the mistake to assume things**. When this happens, the program run into problems, and this can have some problematic consequences.

- The network is reliable
- The network is secure
- The network is homogeneous
- The topology does not change
- Latency is zero
- Bandwidth is infinite
- Transport cost is zero
- There is one administrator

Discuss

What happens when the assumption is *not* true?

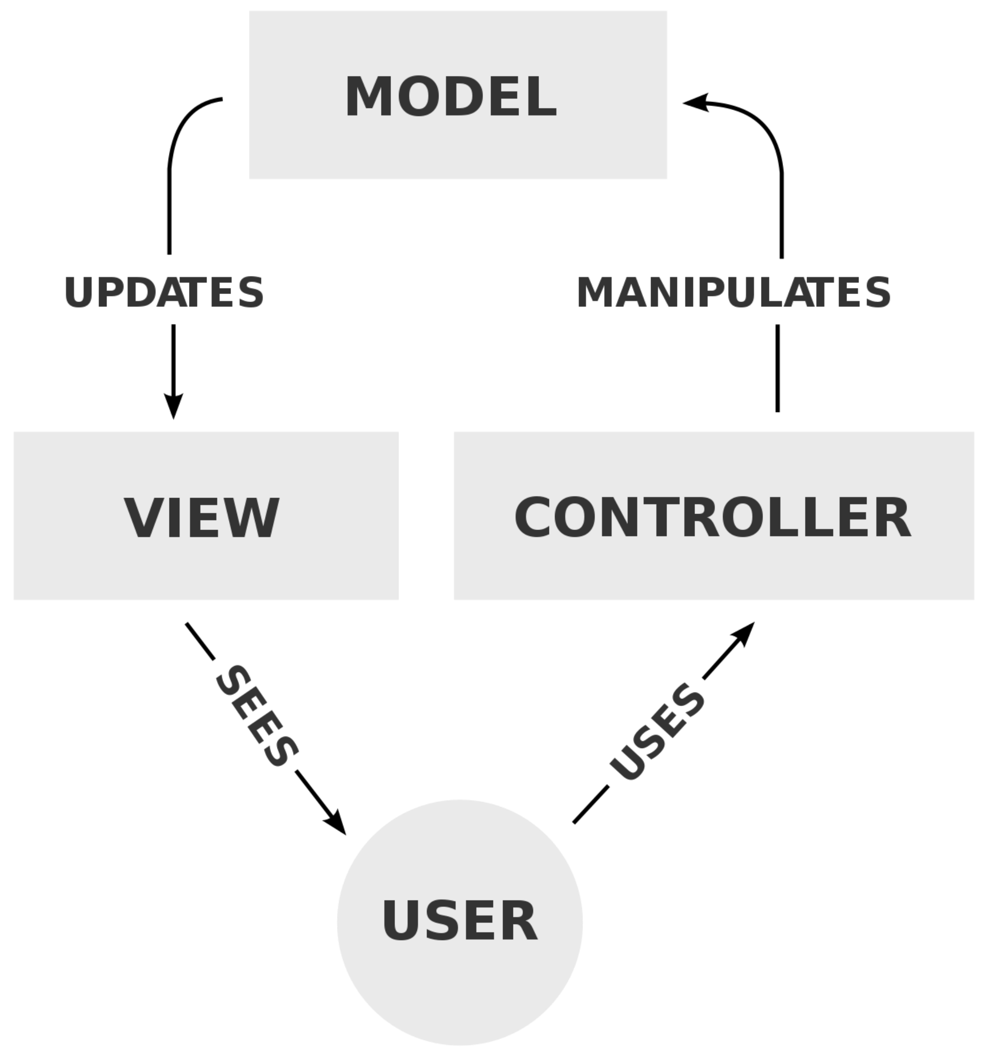
- Make a list of examples, to show you understand when it is *not* true
- Make a list of consequences, what’s the result in those situations

MVC

Model-View-Controller

MVC

Model–View–Controller is a **software design pattern** commonly used for developing user interfaces that divides the related program logic into three **interconnected** elements.



Python - HTML - MVC

When building a web app, you define what are known as routes.

Routes are, essentially, URL patterns associated with different pages. So when someone enters a URL, behind the scenes, the application tries to match that URL to one of these predefined routes.

There are four major components in play:

- Routes
- **M**odels
- **V**iews
- **C**ontrollers

Routes

Each route is associated with a controller - more specifically, a certain function within a controller, known as a **controller action**.

So when you enter a URL, the application attempts to find a **matching route**, and, if it's *successful*, it calls that **route's associated controller** action.

Python - Flask example

```
@app.route('/')
def main_page():
    pass
```

Here we establish the **/** route associated with the **main_page()** view function.

Models and Controllers

Within the controller action, **two main things** typically occur:

- The models are used to **retrieve** all of the necessary **data** from a database
- Data is passed to a view, which renders the **requested page**

The data retrieved via the models is generally added to a data structure (*like a list or dictionary*), and that structure is what’s sent to the view.

Python - Flask example

```
@app.route('/')
def main_page():
    """Searches the database for entries, then displays them."""
    db = get_db()
    cur = db.execute('select * from entries order by id desc')
    entries = cur.fetchall()
    return render_template('index.html', entries=entries)
```

Now within the view function, we **grab data from the database** and perform some basic logic.

This returns a list, which we assign to the variable entries, that is accessible within the **index.html** template.

Views

In the view, that structure of data is accessed and the information contained within is used to render the HTML content of the page the user ultimately sees in their browser.

Again, back to our Flask app, we can loop through the entries, displaying each one using the Jinja syntax:

HTML

```
{% for entry in entries %}
<li>
    <h2>{{ entry.title }}</h2>
    <div>{{ entry.text|safe }}</div>
</li>
{% else %}
<li><em>No entries yet. Add some!</em></li>
{% endfor %}
```

MVC Summary

MVC request process is as follows

- A user requests to view a page by entering a URL
- The application matches the URL to a predefined **route**
- The **controller action** associated with the route is called
- The controller action uses the **models** to retrieve all of the necessary data from a database, places the data in an array, and loads a **view**, passing along the data structure
- The **view** accesses the structure of data and uses it to render the requested page, which is then presented to the user in their browser

Links - MVC

- www.tutorialsteacher.com/mvc/mvc-architecture
- https://www.tutorialspoint.com/mvc_framework/mvc_framework_introduction.htm - Only the Introduction rest is ASP.net

Understanding MVC architecture

https://youtu.be/eTdVkgF_Slo

AILS ESSENTIAL TRAINING

MVC Architecture

Discuss how to understand the MVC pattern

Explain in your own words

- What does each part do?
- Think of the programs you have created in Python – what would you have to do to change it into MVC style?

Design Patterns

Well-known solutions to well-known problems - And that is called Design Patterns

Design Patterns

In software engineering, a design pattern is a **general repeatable solution** to a commonly **occurring problem** in software design.

A design pattern **isn't a finished design** that can be transformed directly into code.

It is a description or **template** for how to **solve a problem** that can be used in many different situations.

https://sourcemaking.com/design_patterns

Gang Of Four Design Patterns

Elements of Reusable Object-Oriented Software(1994) written by *Erich Gamma, Richard Helm*, *Ralph Johnson*, and *John Vlissides* is a book on software engineering highlighting the capabilities and pitfalls of object-oriented programming.

They have listed **23 classic software design patterns** which are influential even in the current software development environment.

The authors are often referred to as the **Gang of Four**.

Benefits of the design pattern

- Design patterns can **speed up the development** process by providing tested, proven development paradigms.
- Reusing the design patterns helps to **prevent** subtle issues that can cause major **problems** and it also improves code readability
- Design pattern provides **general solutions**, documented in a format that doesn’t specifics tied to a particular problem
- In addition to that patterns allows developers to **communicate** well-known, well-understood names for software interactions,
- Common design patterns can be **improved over time**, making them more robust than ad-hoc design
- A standard solution to a common programming problem enables large scale **reuse of software**

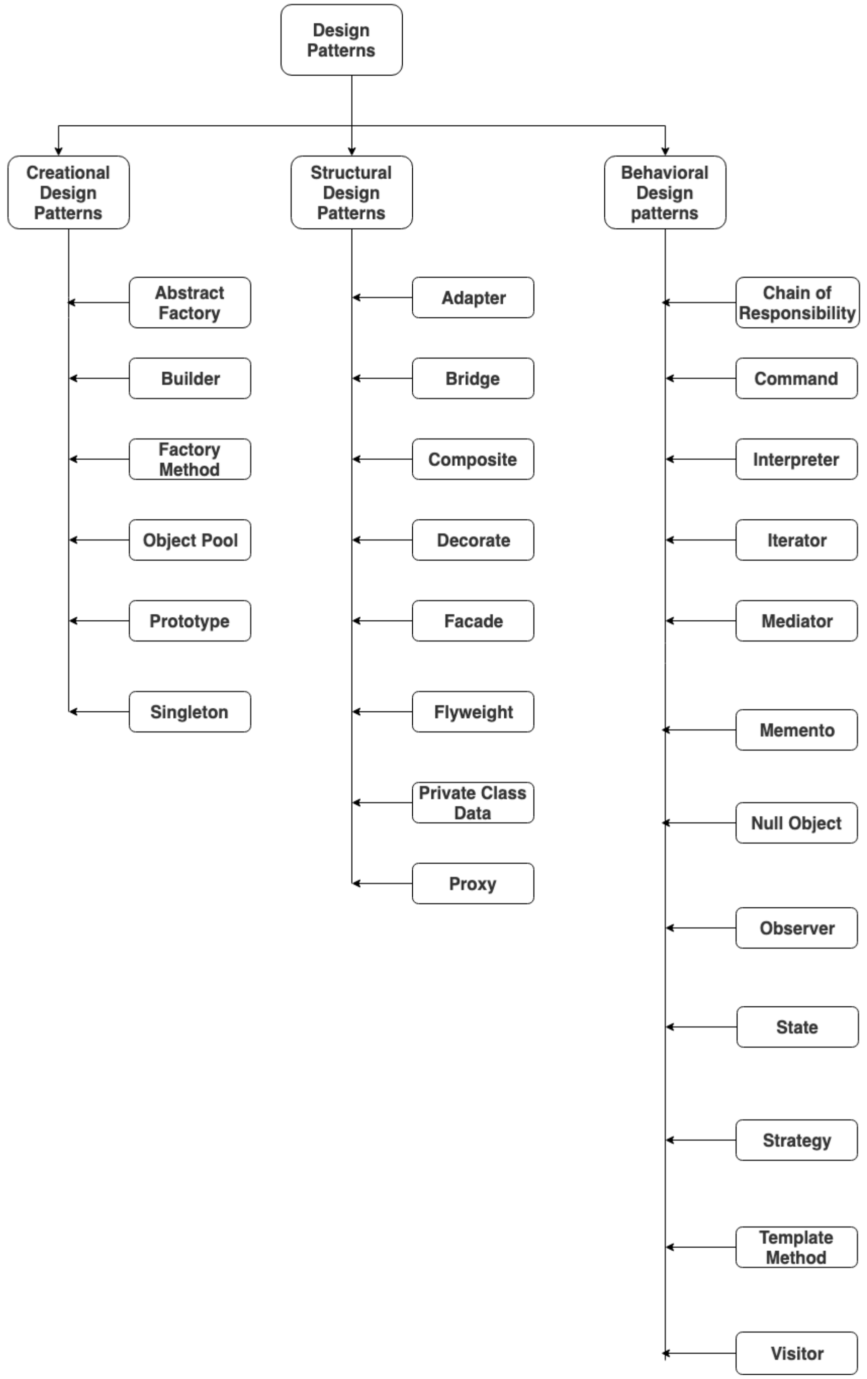
Types of Design Patterns

The 23 design patterns have been categorized into **3** verticals:

1. **Creational** - Deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.
2. **Structural** - Deal with easing the design by identifying a simple way to realize relationships among entities.
3. **Behavioural** - Deal with

Types of Design Patterns

- Creational
- Structural
- Behavioural



Uses of Design Patterns

Design patterns can **speed up the development** process by providing tested, proven development paradigms.

Effective software design requires considering issues that may not become visible until later in the implementation.

Reusing design patterns helps to **prevent** subtle issues that can cause major **problems** and **improves code readability** for coders and architects familiar with the patterns.

https://sourcemaking.com/design_patterns

Command Patten

The command pattern is handy in situations when, for some reason, we need to start by preparing what will be executed and then to execute it when needed.

The advantage is that encapsulating actions in such a way enables Python developers to add additional functionalities related to the executed actions, such as undo/redo, or keeping a history of actions and the like.

Code

Select one Design Pattern to understand together

You can use - https://sourcemaking.com/design_patterns - as a starting point.

Understand and debate in the group

This is not an easy assignment. - Don't panic if you can not understand all details. Try to grab the main idea and then use your common sense.