# High Level System Design

# Introduction

- When we talk about system design, we generally divide it into two parts: *High Level* and *Low Level*.
- High level design do not include any code but think about how the system will function.
  - How are we going to scale it?

# Pizza shop

- We want to open a pizza shop. We start small with *one* chef and one or two types of pizza to sell.
- Let's say that our shop works great and there are a lot of customers.
  - More customers mean more orders, which is why our chef is getting a hard time supplying all that demand.
- How are we going to handle all these orders?
  - We can ask the chef to work more and harder!
    - He is not going to do that for nothing! It is possible that he can demand more money.
    - This is similar to a *computer*, or *processor*. By spending **more** money, we can create a more **powerful** machine.
    - This is called **vertical scaling**.

# Pizza shop > cont.

- We also want to optimize our processes and increase the throughput by using the same resources.
  - However, we don't want to spend more money, so we are going to do what we can in order to utilize our resouces.
  - Use them in a better way if that is possible.
- E.g. in cooking, there are a lot of stuff to prepare. You would not want to prepare them when the customer comes. That is why these preparation are done in some *spare* time, when there are not much customers.

# Preprocessing

- It is similar in a computing system. Sometimes the traffic is very low. That is the time we would want to do our preparation!
  - We prepare stuff at non-peak hours.
- That is called **preprocessing** and usually done by using **cronjobs**.

## Homework

Recently we learned about Command Line. Research what is a cronjob and how is it used.

# Single point of failure

- What would happen if the chef is sick?
  - There is no one to make pizzas.
  - When the chef is gone, the shop is done!
- This means that the *chef* is a **single point of failure**.
  - We don't want to have SPoF's in our system.
  - We need a *backup* chef!
- For similar cases, we need to have some system ready to go so that our shop is not closed.

# Horizontal scaling

- Sometimes we will have more work than usual. Yes, we can pay more to the chef to work more but the work may still increase.
- What can we do ?
  - We can hire **more** chefs!
  - This is called **horizontal scaling**.
- Here, we buy **more** machines of similar types to get more work done.

# Routing

- So, at this point we now have more than one chef. It is possible that they have **specialties**.
  - One can be a Chef #1 i a chef of *garlic bread* and the other can be *pizza* chef, Chef #2.
  - When the order comes, we want the system to work in such a way that if the order is garlic bread, it goes to Chef #1.
  - If it is pizza, it goes to Chef #2.
- This is a **routing** problem.

# Solutions to routing problem

- Random distribution of orders
  - Obviously, this is not efficient. Chefs are not going to be happy and they are not going to be able to do their best work.
  - Probably customers won't like it too.
- Therefore, this is not a good solution!
- What if we create *teams*?
  - Now, every team will be special teams who are experts at what they do!

# Microservice architecture

- We have a *team pizza* and *team garlic bread*. These teams are only going to be responsible from what they know.
- If there are any questions about garlic bread, we will only communicate with that team.
  - This is called **microservice architecture**.
  - For every team or *microservice* the responsibilities are defined.
- We will talk about **microservices** a lot later.

# Redundancy

- We talked about *single point of failure*.
  - If there is only one chef, we have a SPoF but the chef is not the only thing!
  - What if the electricity went out?
  - What if there is no water?
    - For these cases, we want **redundancy**.
  - We can buy a *water tank* ready incase the water is gone.
  - We can buy a *generator* if the electricity went out.
- However, in computer systems; we can think of creating a *different shop*.

# Redundancy

- We distribute our jobs into separate jobs.
  - We are going to build a new pizza shop at *another* place. It is going to be a *backup* service.
  - If anything happens to the original pizza place, we always have the other one!
- In addition, they are also going to work together, we it means we are also **expanding**.

# Distributed System

- However, this is a complex action.
  - Not only we are opening up a new place, if we want to work efficiently, we need to have **communication** between these shops.
  - This is called a **distributed system**.
- Creating a *distributed system* helps in terms of **fault tolerance** and **faster response times**.

# cont.

- At this point, we have two shops: $A$ and $B$. We also have an *application* where users can buy pizzas!
- Someone wants a pizza.
  - Here is a question: Which shop are we going to send the request to?

# Choosing where to route

- At this point, we have two shops: $A$ and $B$. We also have an *application* where users can buy pizzas!
- Someone wants a pizza.
  - Here is a question: Which shop are we going to send the request to? \pause
  - We are not going to ask that to customer!
  - We need to decide it which is the **best** solution for the customer. \pause
- The first thing comes to mind is *proximity*. \pause
  - Whichever the shop is the closest one, you can send the request to them.
  - Sometimes big companies with a lot of requests use local machines to improve their *response time*.

# Choosing where to route

- However, what we want is actually **faster response times**.
  - This is not guaranteed by proximity!
  - Maybe that shop has a lot of customers and very busy!
- That is why we need **parameters**.
  - The time passed from the request and delivery is a *parameter* here.
  - How can we solve this? /pause
  - We can ask every shop how many pizzas are waiting in the line!
  - Then we add two things: the time it takes to make a pizza and the delivery time.

# Load balancing

- After these calculations, naturally we will choose the **shortest** one.
  - However, how are we going to choose that?
  - Who is going to choose, who will **decide** it?
- We can use a **central authority** for this purpose. That central authority is called a **load balancer** and the action is called **load balancing**.

# Separation of responsibilities

- All these pizzas need to go to the customer. It is done by the *courier*.
  - We can also call them *delivery agents*.
  - These agents do not need to know **what** they are carrying because it is irrelevant.
    - It can be pizza, burgers, etc.
- It means they these agents are totally **independent** from the pizza shop.
- Here, we see a **separation of responsibilities**.
  - We do not manage the delivery.
  - This is called **decoupling** the system. We are *separating* our concerns therefore we can handle separate systems more efficiently.

# Logging and metrics

In order to improve efficiency of the system, we need to *store* certain information.

- *e.g.* If something is broken, we need to know about it.
- We also store some information about *how* the system works.
  - It is called **logging and metrics**.
- A good example to it can be *analytics*.
  - Many websites use analytics to store information about users:
    - Where they connect from?
    - What they do?
    - Which button they click?
  - These informations will help the system improve!

# Extensibility

- By *decoupling* stuff which are not actually related to each other (like courier) we make sure that the system is **extensible**.
  - Delivery agents don't need to know the type of food they carry.
  - They can be **decoupled**.
- By using services like these, we are going to make an extensible system.

# Horizontal vs Vertical Scaling

# Introduction

- We designed a system. There is a *server* and there are some code running on it.
  - That code is a *program*.
  - It takes an *input* and gives out *output*.
- This program is very functional and other people pay you money to be able to run that for themselves.
  - You can package and sell it.
    - Not the *standard* anymore.
    - It is also possible that you don't want to give your data to them.

# Story time

- Geohot (George Hotz).
  - Founder of comma.ai
  - Trying to make affordable autonomous cars.
- System works with a mobile phone.
  - During testing, they used **a lot of** camera data to train their machine learning systems.
  - He claims not not even *car companies* have that much data.
- In a recent podcast with Lex Fridman, he asked him if he thought of sharing this data for a price.

# How it works?

- When a program is working on a server, it can rely on some data which you don't want to give out.
- You can share the *application* which people can use to **use** the data for a price (*or free*) by using an API.
    - Now, you can share some data but limit the usage!
- It is possible for people to pay for your program, or system.

# Our system

- At this point, imagine that we have a system we share over the Internet. People are going to connect to that system.
  - They will **request** some information and the system will **respond** to them.
- Our system is *popular* and is being used by a lot of people.
  - Now, we need to think about additional stuff.
- We cannot have any **downtime**.
  - What will happen if there is a power loss?

# Cloud

- We can use **cloud** systems. That is what we do! Instead of storing the system in our own computer, it resides on a computer on a cloud.
  - These companies are experts in this, therefore we expect them to have preventions against these.
  - AWS, Digital Ocean, Heroku are some examples.
- When we say cloud, we still have a computer but we just don't have control physically.
  - We will connect to them *remotely* by using SSH.
- They usually use **virtualization**.
  - There is not a single machine allocated to us.

# What else?

- We solved the power problem. But we are now more popular than ever.
  - Our system is not able to **respond** to all these **requests** fast enough!
- This is a **scalability** problem.
- We can solve it in two ways:
  - Buying a **bigger** machine
  - Buying **more** machines.

# Vertical Scaling

- If we buy a **bigger** machine, we are going to be able to respond faster to requests.
- It can handle more requests in a given time.
- This is called **vertical scaling**

# Horizontal Scaling

- When we have **more** machines, we can *balance* the load between them.
  - We will have **faster** response to the requests.
  - The system will handle **more** requests in a given time.
- This is called **horizontal scaling**.

# Advantages & Disadvantages of Horizontal Scaling

- We have multiple machines and we divide the responsibility. However, **how** to divide it? We need **load balancing**.
- There is no **SPoF** because we have redundancy now. If a machine fail, we can easily distribute that responsibility.
- Since we have multiple machines, they need to communicate over the network. These are done as network calls, **RPC** (remote procedure calls) and it is slower compared to **IPC**.
    - Inter process communications

# Advantages & Disadvantages of Horizontal Scaling

- Data consistency can be a problem. We need **loose coupling**. When you have an atomic transaction, you need to lock all databases at all devices and it is impractical.
  - (We will see that when we talk about *microservice* architecture)
- It is hard to improve a single machine indefinitely. At some point, it is going to stop.
  - Then, you will increase the number of devices.

# Advantages & Disadvantages of Vertical Scaling

- You don't need **load balancing** because you are using a single machine.
- There is **SPoF**. If something happens to a machine, the system will go down.
- Because there is a single system, communication between processes will be done by **IPC** which is faster.
- There is no *consistency* problem.
- It is possible to hit a dead end in terms of increasing computational power.

# Verdict

We can see that both have advantages an disadvantages. In real life, both approaches are used. We take good qualities of both and combine them together.

It can be seen as a **hybrid** solution.

We end up with **big** machines and a lot of them.

# Load Balancing

# Introduction

- In horizontal scaling, we are adding more machines to the system.
- In order to **route** requests, we need a load balancer to determine which server to choose.
- It sits between the user and servers.
  - First, the user is going to connect to the load balancer.
    - DNS will point to it.
  - Then, the load balancer will send the request to the available server.

# Types of Load Balancers

## Round Robin

The simplest and one of the most popular ways to route requests to servers.

The first request go to the first server, the second request go to the second one. It will carry on like this.

This is a naive approach.

# Types of Load Balancers

## Load-based

- Load based balancers are more intelligent.
- Instead of just sending the request without checking the *availability*, the system checks whether the server is **suitable** or not.
  - Maybe the first server is doing some calculation and busy.
  - Then, the request is sent to another server.
- It can work in two ways:
  - **Connection based**
    - It will check the server with the *least* number of connections.
  - **Resource Based**
    - Monitor the machines, CPU, memory levels and decide on that.

# Summary

- Scalability, availability and flexibility.
- You can increase *scalability* by increasing the number of servers.
- You can increase *availability* because there are multiple machines that can take place of another if a problem occurs.
- You have *flexibility* because you can direct requests based on the load on the servers.

# Consistent Hashing

- When you have multiple servers, you need to determine which server is going to store which requests.
- There are some ways to do it. One of them is using *hash* functions and *hashmaps*.
- Let's draw how it works.
- We are going to take the hash of the request id and point it to the server. If we have $m$ servers, we are going to apply **mod m** so that we can point it to a server.

# Consistent Hashing

- However, this creates a problem if you want to add a new server to the system.
- We will see that previously a request was going to server $x$, but now the calculation points to server $y$.
  - We need to change almost everything.
  - It is very expensive.
- That is why people thought about this and came with a solution.

# Consistent Hashing

Like every request, we also have a *server id*. Instead of just hashing the *request id* we are also going to hash *server id*.

- In addition to this, we are going to think of our distributed system as a **ring**.
  - This means that every server is going to be on a *point* in this ring.
- We know that a circle (ring) consists of 360 degrees. We can easily say that there are 360 points in a circle. Remember the modular arithmetic we have done.
  - We are going to do something similar here.
- Drawing

# Consistent Hashing

- We are going to hash the server id, $hash(s_id)$. And we are going to take mod360 of it, so that it represents a *point* in the ring.
- Then, we do the same thing to the request id. $hash(r_id)$.
    - We are going to select the server which is the *next closest* to that point on the ring.
- This helps because now if we add a new server, instead of changing everything, we are only going to change those which are between the *previous* server and the *added* one.