System Design Basics

System is built considering these things in mind

**1. User**

**2. Requirements of users**

**3. Components of system**

Design  what those components are going to be and how they will interact with each other to fulfill the requirement of the users.

In order to build large scale systems, you need very good knowledge of technologies and software.

A simple Image Sharing App (a system) will be designed in this fashion 

**1. User Registration/SignIn service.**

**2. A Followers/Following DB Structure so that users can follow each other.**

**3. An API for the User to POST an image.**

**4. A GET API for other users to view the images posted.**

**5. An ML model served through an API for recommending people to follow other users.**

**Components of System **

**1. Logical Entites**

**2. Tangible Entites**

**Basics**:

If you have a server and then you user base grows, so more and more users are now using your code/api then the load increases and one machine is not able to handle that load. So, you need to scale. This is called scalability.

We take underlined features from both and then we make our solution.

Both of these are use in real life.

**Basics of System Design **

1: vertical scaling: optimise precision and increase through put with the same resources

2: pre-processing (e.g cron job) : prepare beforehand during non-pick hours

3: Backups: keep backups and avoid single point of failure

4: horizontal scaling: get more resources

5: micro service architecture

6: distributed system (portioning)

7: load distribution

8: Decoupling

9: Logging

10: extensible

**Scalability**

**Vertical Scaling:** Make the machine bigger by adding more resources to the machine such as RAM, processing power etc.

**Horizontal Scaling:** Add more machines. Each machine will have its own RAM, processing power etc.

**Comparison:**

* Horizontal Scaling will need some **load balancer**, but Vertical Scaling does not need that.
* Vertical Scaling has **single point of failure**; however, Horizontal Scaling is **resilient**.
* In Horizontal Scaling, we must communicate through the network which are network calls, also called **RPC** or **Remote Procedure Calls**; however, in Vertical Scaling we communicate through **inter-process communication**, which makes it faster.
* Horizontal Scaling has **data consistency** issues due to the presence of multiple machines; Vertical Scaling does not have this issue.
* Vertical Scaling has a **hardware limit** because there is a limit to how much resources we can add to a single machine, Horizontal Scaling will scale well as the number of users increase.

We try to create a hybrid solution using the good qualities from both types. We try to take machines as big as we can (depending upon the budget) (Vertical Scaling) and we take many machines instead of just one (Horizontal Scaling)

Let’s create a **Pizza Shop** example to understand the concept.

We only have one chef at the shop. With only one chef, how do we serve many users?

We can increase the speed and efficiency of the chef by paying more (**vertical scaling**)

We can prepare food beforehand at non-peak hours.

**Resilience:** if our chef calls in sick then we have a single point of failure. Let’s hire a backup chef, only when our main chef is not available. We avoid single point of failure using this technique. This makes the system resilience. This is like **master slave architecture.**

**Availability:** Hire more chefs (**horizontal scaling**). Keep backups.

Let’s expend our business.

Let’s say chef2 has expertise in making garlic bread, chef1, chef 3 have expertise in making pizza. **How would you route your requests?**

We route all the requests we receive for garlic bread to chef2 and pizza orders to chef1 and chef3. This way we are assigning requests to the chefs who have expertise in that item. Also, all the responsibility for garlic bread lies on chef2 and all the responsibility for pizza lies upon chef1 and chef3. This is **microservices architecture.**

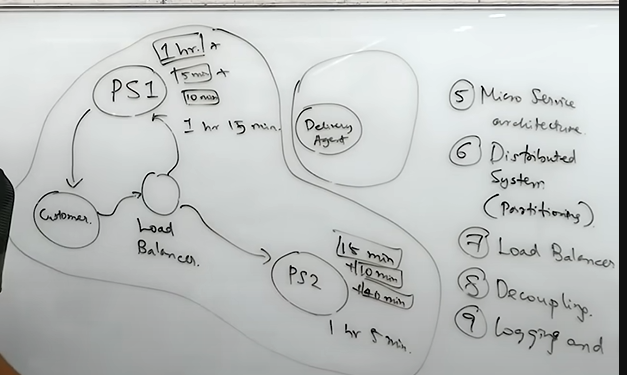
**What happens if you don’t have electricity for a day?**

You might lose business that day. To avoid this, we create another shop which can also deliver orders. Now, we need to route requests between the main shop and backup shop. Also, there needs to be some communication between the main and backup shop. This makes the system complex. This has now become a **distributed system.** The backup shop will be able to serve requests which are very close to it geographically, this is **partitioning.**

Let’s say you now have multiple main pizza shops. You need to route orders between those shops. To route, we will use the time required to deliver the customer his order as a parameter. This will tell if we should send the request to shop1 or shop2 or so on. This is **load balancing.**

The responsibility of delivery agent is to deliver the order. Shop manager’s responsibility is to make any order as fast as they can. The responsibilities are separated. This is **decoupling.**

Sometimes we may see issues in the system. Shop1’s oven broke down or delivery agent’s bike broke down etc. We need to record all the events to find issues and make system faster. This is **metrics.**

****

Problems we might face in the future:



This is **high level design (HLD).**

We also have **low level design (LLD). LLD** concerns with the coding up the HLD. We need to make classes, interfaces, functions, tables and many more.

**Load Balancing:** User sends a **request** to the **server** and server sends back a **response**. As the number of requests increases, we need to **scale up** the system.

If we now have many servers due to scaling up, now we also need to balance the load on these servers. **Load** means the processing/work the server must do. If we have N servers and we want to distribute the load among them, this is called **load balancing.**

**Hashing:** If we want to distribute the load evenly among all the servers, we can use consistent hashing. If user sends the request, we receive a **request id.** This id is expected to be unique and random.

Let’s assume that we have n servers, and we want to distribute the load among them. So, whatever id we receive we will find its mod with n, and this will give us a number. This number is then mapped to a server. We send this request to that server.

**Id%n = m; We send this request to mth server. n = number of servers, id = request id.**

If Ids are random then we can expect the load to be uniformly distributed.

If x is the total number of requests, then **each server will have x/n load and load factor is 1/n.**

Let’s say we add a new server now. New hash function is **id%(n+1).** With this change the requests that were previously going to a certain server will now go to a different server. for example, if we had 4 servers before that request with id = 10 was going to 10%4=2, but now, it will go to 10%5 = 0, like this a lot of requests will now go to a different server due to increase in the number of servers.

In practice, the request id is never random, it encapsulates some information about the user. So, if a user sends a request, it is mapped to a server using hashing. The server will save some of the information in local cache. This makes the retrieval very fast. Now, due to the addition of the new server, our hash function has changed and now we are sending the request to a different server. This new server has to query to DB to get the information. However, the previous server had that user information in cache. So, change in the hash function makes almost all the information stored in the cache of the server useless.

To avoid this issue what want to do is to not change the search space of the server very much. If server 1 is serving 0-100 user ids, then we might make it 0-90 and make the new server to serve the 10 ids. Same with server 2. Take some of the users from each server and make the new server serve these users. This will not make all the caches of these servers useless.

Let’s see how to do it:

Number of users, m = 100

number of servers, n = 4

hash function = user id % n

users served by s0, s1, s2, s3 = 25 each

s0 serves 0, 4, 8, 12 and so on

s1 serves 1, 5, 8, 13 and so on

s2 serves 2, 6, 9, 14 and so on

s3 serves 3, 7, 10, 15 and so on

Now, if we have 5 servers and we use the same hash function ie user id % n

Number of users, m = 100

number of servers, n = 5

hash function = user id % n

users served by s0, s1, s2, s3, s4 = 20 each

s0 serves 0, 5, 10, 15 and so on

s1 serves 1, 6, 11, 16 and so on

s2 serves 2, 7, 12, 17 and so on

s3 serves 3, 8, 13, 18 and so on

s4 serves 4, 9, 14, 19 and so on

If we focus, we can see that user 0 is still being served by s0 in both the cases. But user 4 is now being served by s4 instead of s0. This is the problem. If we calculate we see that many users are now being served by different servers. Which makes their caches less useful. To solve this problem, we use **consistent hashing.**

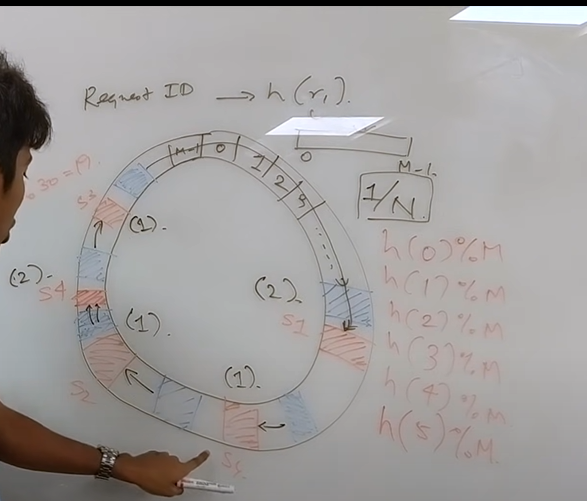
**Consistent Hashing:** The problem is not load balancing, the problem is how we add and remove servers.Instead of just hashing the user ids, we will first hash the server ids also. Hash function looks like this: **hash(s) % m.** s is the server id. m is the number of users or request ids. First hash the server id, get that value then use that value to hash the user id. Instead of a linear array, we create a circular kind of array. The request nearest to the server is served by it.

Requests are 0, 1, 2…, m-1. So, a total of m requests. We have 4 servers: s0, s1, s2, s3.

hash(0) % 30; hash on server 0 and request id or user id is 30.

Let’s assume hash(0) = 49 and then 49%30 = 19, so server0 is mapped to position 19. And so on for other servers as well. All the user ids near to 19 will be served by server0 and same will go for other servers as well.

When a new server is added, there is not much change to the users getting to some servers.



If a server goes down, there may be issues and some server might be overloaded with load.

We can solve this problem by adding more and more hash functions. If we have **k-hash functions** through which the user ids pass, then each server will have many points in the ring. Now, the servers are distributed over the request on the ring. Each server is on multiple points in the ring. If a server goes down, then its load will be distributed to the other server almost evenly.