# System Design Notes:

## Lec 1 scalability:

<https://youtu.be/xpDnVSmNFX0?list=PLMCXHnjXnTnvo6alSjVkgxV-VH6EPyvoX>

Scalability: The ability to handle more requests by buying more machines or buy bigger machines

Vertical Scaling: Buying BIGGER MACHINES

Horizontal scaling: Buying more Machines

|  |  |
| --- | --- |
| Horizontal scaling | Vertical Scaling |
| Load Balancing required | Not any |
| Resilient against single point of failure | Single point of failure |
| Remote procedure calls between two servers. Therefore, the network calls (RPC) are slower as it has more I/O ops in. | Inter Process Communication. Hence its faster. |
| Atomic operations cannot happen🡪 Loose coupling🡪 Data Inconsistency. | Consistent |
| Scales well as **users increases.** | Hardware Limit |

Questions?

Is it scalable? Is it resilient? and is it consistent?

## Lec 2 Scaling up a Pizza shop

<https://www.youtube.com/watch?v=SqcXvc3ZmRU&list=PLMCXHnjXnTnvo6alSjVkgxV-VH6EPyvoX&index=2>

Vertical scaling

How to optimize processes and increase throughput using the same resource 🡪 Vertical scaling

Preprocessing using cron Jobs

Some Jobs can be done before hand like preparing a pizza base at non peak hours.

Backup servers

If the pizza chef is sick and he is the only chef then we can call a backup chef. He can replace him for those days and we can pay him.

**Keep back ups to avoid single point of failure.**

**This is Master Slave architecture**

Horizontal scaling

We can also hire more chefs and even keep some back up as well.

**Hire more machines of similar type to get work done.**

Microservice

If there are two chefs, one is specialized in garlic bread and other is specialized in pizza. Then its better to channelize all the orders based on garlic bread to one chef and the pizza orders to pizza expert chef. This will also synchronize the process like changing the specific menu etc.

We can also increase it into two teams like Garlic bread team specialized in garlic bread orders. And pizza specialized team. This specialized team is also called Microservice.

Distributed Systems

What if the shop ran out of electricity? Or lost a license for one day? Don’t put all the eggs in one basket!

Buy a backup shop!

The issue here will be communication gap between both the shops.

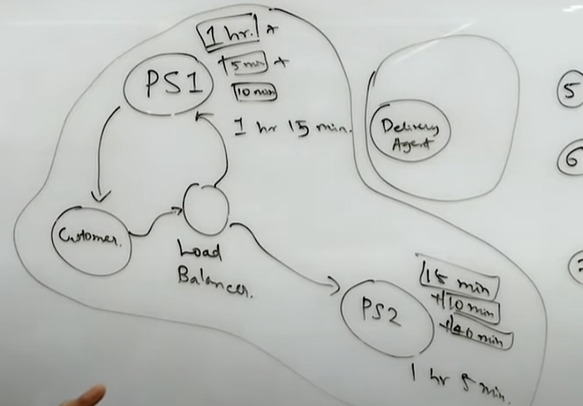
Load Balancing

Now a client doesn’t need to think which pizza shop it needs to call? Is it Pizza shop 1 or pizza shop 2. It will just call the central authority.

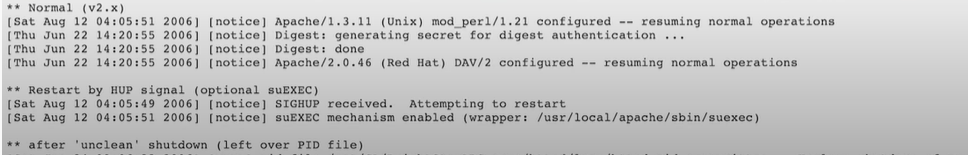
The central authority also called **Load balancer** will decide which pizza shop it needs to direct the order. It will compute all the time like delivery time, cooking time, queue time etc. and decide between the two shops.

Decoupling

Separating the delivery agents and managers for both the shops. **Separation of responsibilities.**



Logging and metrics calculation



Analytics, Auditing, Reporting, Machine Learning. In order to monitor each de coupled system. And then taking actions.

Extensibility

We don’t want to re write the code again and again to fulfill slightly different purpose. The delivery agent doesn’t need to know that they are delivering a pizza, it can be a burger tomorrow.

Low level system design

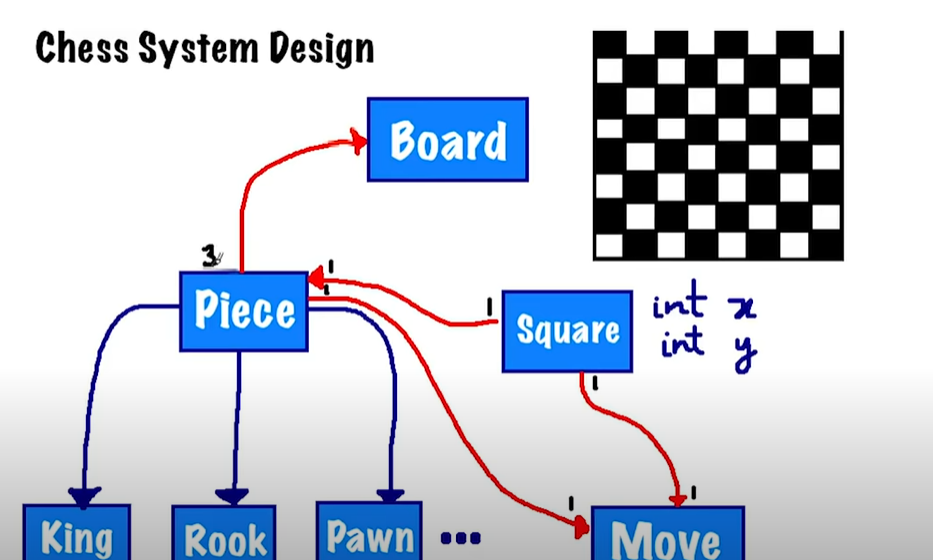
Related with code like classes and objects etc.

High Level Design

**Order Overload🡪Recruitment**

**Complexity🡪 Separation of concerns**

**Mishaps🡪 Fault Tolerance**



## Lec 3: Load Balancing:

<https://www.youtube.com/watch?v=K0Ta65OqQkY&list=PLMCXHnjXnTnvo6alSjVkgxV-VH6EPyvoX&index=3>

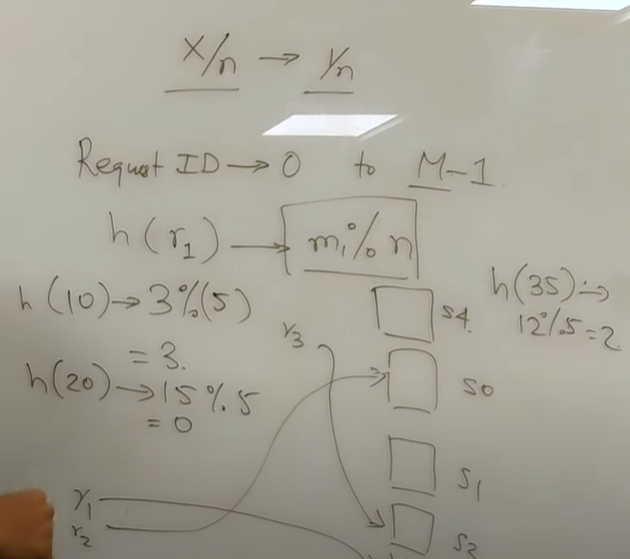
To evenly distribute the load amongst all the servers.

Consistent Hashing:

Let the requests be from ID: 0 to M-1;

We hash(r1)🡪m1 this maps to m1%n server.

Given that the hash function is uniformly random. The servers will have uniform load.



Now with the change in n (number of servers) Huge traffic of requests is directed to different channel. To reduce **This change, we need consistent hashing.**