Networks

Distrbuted Systems

Parallel Computing

These are used to build large-scale systems.

When building large scale systems, we have to make them distributed by design.

On the server side, we will need brilliant engineering.

Requests on the server should never fail.

Even if there is hardware failure, we should have multiple copies of same server in the system.

Changing the number of copies should be easy so that, as per our requirement, we can add or remove servers.

We should make sure that the information on each server is the same.

Important consideration is to distribute the load fairly.

Requests should be responded to in a timely fashion.

This requires optimization of our networks and databases.

**Lecture 2**:

**Horizontal vs Vertical Scaling**:

Suppose we write some code and other people want to use this code.

We can expose our code through some protocol running on the internet.

We expose our code through an API.

If the code runs then instead of storing the output in a file or db, it gives a response.

A request is sent to our API.

We cannot afford to let the service down due to power failure, etc.

We should host the services on the cloud.

Cloud is a set of computers that somebody provides to us for money.

We can do a remote login to a computer/desktop on the cloud.

For example, AWS

We can now focus only on the business requirements.

If suppose the machine on which our code is hosted is unable to handle multiple requests/connections.

One solution is to buy a bigger machine.

2nd solution is to buy more machines.

The ability to handle more requests by buying a bigger machine or more machines is called as **scalability**.

We can handle more requests by throwing more money at the problem.

First solution is called vertical scaling. The machine is larger and can process the requests faster.

2nd solution is called horizontal scaling. Requests can be randomly distributed among machines that we have.

**Horizontal Scaling**:

1. Load Balancing is required.

2. If one machine fails, then we can redirect the requests to the other ones. (Resilient).

3. Communication between the servers will be over the network. Network calls are slow. Its I/O.

Remote procedure calls (RPC).

4. Issue of Data inconsistency. Suppose there is a transaction in which machine 3 sends data to 4, and 4 to 5, 5 to 1, etc. Data is complicated to maintain. There is some sort of lose transactional guarantee.

5. **Scales well**. The amount of servers that we throw at the problem is almost linear in terms of how many users are added.

**Vertical Scaling**:

1. There is no load to balance on a single machine.

2. Single point of failure.

3. There is inter-process communication here. Faster

4. Consistent

5. Hardware limit.

We use both in real world. Take good qualities of both.

In vertical, the cache is going to be consistent and there are no dirty reads and dirty writes.

Hybrid solution is horizontal scaling only.

Initially we can vertical scale as much as we like. Later on, when the users start trusting us, we should probably go for horizontal scaling.

Is it scalable?

Is it resilient?

Is it consistent?

There is always going to be some **trade offs**.

We design a system which is going to meet the requirements

**Lecture 3**:

Let’ s take an example of a restaurant. Suppose there is 1 chef.

Optimise processes and increase throughput using the same resource: In computing this is called Vertical scaling.

Prepare before hand at non-peak hours.

Resilience: capability to recover from difficulties

Hire a back up chef.

**Trade Off**: a balance achieved between two desirable by incompatible features.

Keep backups and avoid single points of failures.

For computers, it like a Master slave architecture.

We have a master and slave chef.

If the business keeps on growing, we can make the backup chef as a full time or in fact hire more chefs.

Hire more resources which maps to Horizontal scaling.

Now suppose there are 2 chefs for pizza and 1 chef for making garlic bread.

Build on the strengths. Route all garlic bread orders to chef 2 and pizza orders to chef 1 and 3.

Anytime we need to change the recipe for garlic bread, we notify this to chef 2.

And if we want to get information about about an order related to garlic bread, we can ask chef 2.

We can make a team of chefs according to their strengths.

Suppose there is 1 team of 3 chefs for garlic bread and 2 teams for pizzas of 3 and 4 chefs each.

We are scaling the garlic bread team at a different rate compared to pizzas team and also dividing responsibilities.

We have Microservices architecture.

We have all the responsibilities well defined. There is nothing outside the business use case that we handle.

**What if there is an electricity outage in the shop or the shop loses its license** ?

We can have a backup shop.

We should be able to route a pizza request to a particular shop depending on the situation.

**Distributed System**

Any orders which are very close to a shop can be served by that shop.

Partitioning

Customer sends a request to a cental authority which routes the request to a pizza store depending on the time it takes for a pizza to be delivered.

As long as central authority gets real time updates, the business can make profit.

This central authority which routes requests in a better way is called a **Load Balancer**.

The system is now fault tolerant.

How can we make it **flexible to change**?

The shops don’t care whether delivery agent comes or the customer comes to take the order.

Delivery agents and pizza shops are independent.

Instead of same managers managing pizza shops and delivery agents, we want to separate them out.

Decoupling the system

We can handle separate systems more efficiently.

We want to log everything. We want to see at what time something happened and see what are the next events.

And we want to understand these events and make sense out of them.

Analytics

Auditing

Reporting

Machine Learning

Logging and Metrics calculations.

We have to keep the system extensible.

As a backend engineer, we don’t want to rewrite all this code again and again to serve a different purpose.

The delivery agent does not need to know that they are delivering a pizza.

Order overload – Recruitment

Complexity – Separation of Concerns

Mishaps – Fault Tolerance

We just discussed high level design.

Low level design

Has to do with how we will code the stuff.

**Lecture 4**: Load Balancing

**Consistent Hashing**: Related to concept of hashing objects.

Suppose we run an algorithm on our machine and a customer wants to use it and can pay for it.

We have technical specifications.

Our machine is like a server.

A server is something which serves requests.

When a client is connecting with our machine with his machine, he sends a request.

We send back a response with the required data.

Suppose we start to make money, now we can afford to buy new machine.

Now if there is new client, where should we send the request ?

We want to balance the load on all of these servers.

Servers need to process the requests (**load**).

Concept of trying to balance the load evenly on all servers is called load balancing.

Concept of consistent hashing helps to balance the load.

RequestID 🡪 0 to M-1

RequestID is sent to the server.

We hash this request id.

h(r1) = m1

Suppose there are n servers.

m1 % n maps to a server number.

If hash function is uniformly random, then we can expect uniform load.

**What happens if we need to add more servers**?

For some reason, people are hitting our servers a lot.

For the previous request, the sever number might change.

If we previously had 5 buckets and now we have 4 buckets, then the

Cost of the change in this is 100 which is actually the entire search space (M).

In practice, the request id is never or rarely random.

The request id encapsulates the information of the user.

This will help in sending a particular request or user to the same server.

Depending on the user id, we can send users to specific servers.

Instead of sending request again and again to same sevrer,

We can save the relevant information of the user in the cache of that server.

But due to the policy discussed, the whole system changes. Almost All users are sent to different places.

And almost all the useful cache information is dumped.

We want to avoid a huge change in the range of numbers that we are serving.

When we have 4 servers,

Suppose we take equal parts from all 4 servers such that their sum is 20% as the number of servers we now have is 5.

But the overall change should be minimum.

**Consistent Hashing**:

The problem is not really load balancing. The problem is adding and removing servers.