SYSTEM DESIGN CONCEPTS

- Say one server is receiving requests and fulfilling responses of one user , but now when more users start to come we need to scale it. So the easiest thing to do is add more RAM or upgrade CPU . This is called as **VERTICAL SCALING.** Its easy but is limited.
- Better approach is to add replicas of servers so that each srver can handle a subset of requests , this is known as **HORIZONTAL SCALING.** More powerful as we can scale infinetly and we don't even need good machines ,it also adds fault tolerancy and redundancy. If one server goes down then other servers can continue to fulfil requests. Downside of this approach is that it is more complicated. First of all how do we ensure that one server wont get overloadedwhile the others are idle.
- So we use a **LOAD BALANCER ,** which is a server known as a reverse proxy, it directs incoming requests to appropriate servers, An algorithm like round robin can be used . Or algorithms like hashing can be used.
  Id%server_count
- If servers are located all around the world then a load balancer can be used to route a request to a nearest location.
- **Content delivery Networks(CDNs),** If we are just serving Static files like images or videos html,css or JS then we can use a CDN. It is a network of servers located all over the world. CDNs don't run any application logic, they work by taking files hosted on our server(origin server) and copying to the CDN servers. This can be done by push or pull basis.
- CDNs are just one technique for caching. **Caching** is about creating copies of data so that it can be fetched faster. Making network requests can be expensive so our browser requests cache data on to our disk. But even that can be expensive , so computers copy it to our RAM memory, but reading memory can be expensive so OS will copy a subset into L1,L2 and L3 CPU cache.
- How do computers communicate with each other? Every computer is assigned a IP address which uniquely identifies a device. **TCP**/IP -> some set of rules or protocols that decides how we send data over the internet. Reliable protocol. HTTP and web sockets are built on TCP.
- When we click on say google.com how does it know what ip address it belongs to , so for that we have **the DNS(Domain Name System)** which is a largely decentralised service which is used to translate a domain to its ip address. So once we do that , our computer system will cache it so that it doesn't need to make the DNS query every single time .
- But why do we use http://google.com , so here why do we always use http? Application level protocol like http which users like us use , It follows the client server model. Where the client will initiate a request which includes 2 parts , one is the request header (it tells us where the package is going and whom it is from and some other meta data) and second part is the request body which is the package content.
- Even with HTTP we have multiple API patterns which we can follow like REST, or GRAPHQL or gRPC.
- **REST** -> Standardization around the http APIs making them stateless and consistent guidelines , like a successful request should include a 200 code and bad request would return a 400 code.
- **Web sockets** -> application level protocol. Consider chat apps were when someone sends a text we receive it immedietly , but if we were to implement a http then we would have to make use of polling where we periodically make a request to check if there was a new message available . But unlike http , websocket **support bidirectional communication**. So whenever we send a message it is immediately pushed to the device.

- And for storing all of these data we have SQL or RDBMS like MySQL, but why do we need DB when we can store everything in a text file ?
- With DBs we can more effeiciently store data with Data Structure like B-trees and we have fast retrieval of data as the data is organised into tables and rows.
- RDBMS are mostly ACID compliant which means it offers:

  A – Atomicity – Every transaction is All of Nothing

  C- Consistency – Foreign keys and other restrictions will always be enforced

  D- Durability – data is stored on disk, so even if the machine restarts the data is still there.

  I – Isolation – different concurrent transactions wont interfere with each other

- So this consistency restriction lead to NoSQL , as consistency makes DBs hard to scale.
- NoSQL drop this constraint and relations altogether .
- Popular NOSQL is MongoDB which uses JSON and documents.
- So if we don't want FK constraint that means we can break up the Database and scale it horizontally with different machines. This technique is called **sharding.**
- But how will we know which data to keep in which machine ? SO we make use of the shard key. But sharding can be complicated so simple approach can be replication.
- **Repliaction** types –

  Leader – Follower

  Leader - Leader

- If we want our DBs only to read , then we can make read only copies of our DBs which is called the leader-follower. Here every write will get sent to the leader DB who sends it to the followers , but every read is sent to the follower or the leader.
- There is also a leader-leader replication where either of them can read/write , but this can lead to inconsistent data.
- So for example we can have a Replica for every region in the world , it can be complex to keep the replicas in sync. So the **CAP theorm** was created to weigh trade-offs between the replicated design.

  C – Consistency

  A – Availability

  P – Partition (Network)

  Given a network partition in DB we can only "Pick 2 of 3".
- **Message queues** are jus like DBs as they have durable storage and can be replicated for redundancy and can be sharded for scalability.
- But queues have many use cases , If our server was receiving more data than it can process then we introduce a queue , so that our data can be persisted before its processed.
- So doing so we also get a added benefit that diff parts of our app can be decoupled.