SYSTEM DESIGN

High Level Design (HLD):

High Level Design in short HLD is the general system design means it refers to the overall system design. It describes the overall description/architecture of the application. It includes the description of system architecture, data base design, brief description on systems, services, platforms and relationship among modules. It is also known as macro level/system design

Sweet Shop example: the sweet shop owner is happy with his business, but he needs to expand his business. He is opening other outlets to expand his business, but this requires lot of money. So, a computer engineer walks-in with an idea of going ONLINE, so that people can directly place on the website and you can sell your sweets from the same shop. Once you have made enough money then you can go open another outlet.

Accepting this idea, the sweet shop owner asked his computer engineer friend for his help, so the software developer, bought a computer of capacity 2 GB RAM, 1 TB HDD. The IP address of the computer is 5.8.6.1 He hosted a website www.sweetshop.com under the domain name. He purchased a DNS of 5.8.6.1

Now, Sweet shop is online and all the requests from the user are being registered at the computer that Ashwin has. As the shop is online, so it is available for 24 X 7. But what if the HDD is full, or the system has to be restarted or there is any hardware damage? Such situations are called **'Single Point of Failure'**.

What if there are billions of users? In such cases, we can increase the resources of the computer like, RAM, HDD etc. This is called **vertical scaling**. Instead, if we buy more computers, then it is called **horizontal scaling**.

But the software developer opted for horizontal scaling and bought 3 more computer and introduced a load balancer.

Whenever user wants to go to the website, then he would be redirected to the load balancer and it will direct the user to computer.

S No	HIGH LEVEL DESIGN	LOW LEVEL DESIGN
01.	High Level Design refers to the overall system design.	Low Level Design refers to component-level design process.
02.	It is also known as macro level/system design.	It is also known as micro level/detailed design.
03.	It describes the overall description of the application.	It describes detailed description of each and every module.
04.	High Level Design expresses the brief functionality of each module.	Low Level Design expresses details functional logic of the module.
05.	It is created by solution architect.	It is created by designers and developers.
06.	Here in High Level Design the participants are design team, review team and client team.	Here in Low Level Design participants are design team, Operation Teams and Implementers.
07.	It is created first means before Low Level Design.	It is created second means after High Level Design.
08.	In HLD the input criteria are Software Requirement Specification (SRS).	In LLD the input criteria are reviewed High Level Design (HLD).
09.	High Level Solution converts the Business/client requirement into High Level Solution.	Low Level Design converts the High-Level Solution into Detailed solution.
10.	In HLD the output criteria are data base design, functional design and review record.	In HLD the output criteria are program specification and unit test plan.

In the DNS, their will have multiple load balancer's IP addresses so that even if one load balance is not working, the other load balancers would be working and distributing the users.

The next thing to understand is that, DNS will have multiple IP address. If we are in India, then DNS will do an optimization and will give us the Indian IP address because that would be faster.

The sweet shop owner's business has grown, but the work of the computer developer is built things and not on the maintenance or networking part. If Ashwin has four computers, he would put in his place and if the whole place has gone down then all the four computers can shut down. This would-be single point failure.

Amazons – AWS, Google – GCS, Windows – Microsoft Azure etc are cloud providers. They provide computers on rent. They will charge small amount of money and give us computers.

In AWS, we have **availability zone** and **regions**. So, the computers will be at different places and there will be no **single point failure** as the computers would be placed in different availability zones.

When we have a single computer, the things become very simple, but when we have multiple-servers, which is called distributed computing, it become tough. HOW?

A client goes to AWS and every time he logs in, they will check in the data base. The server will check the user name and password in the database and the server will give permission to login to AWS. This is simple. But when you have 3 computers and the same guy is logging in. The load balancer has set S1, S2 and S3 to the three computers. If the load balancers send the user details to S1 which has the details of the user, then he would successfully log-in. But if the load balancer sends the user input to S2 or S3 where the details of the clint are not registered or saved, then their would-be login failure. Such issues, are called **distributed computing problem**.

A husband keeps forgetting things due to which he never wished his wife on her birthday. He came up with a plan to write it in a diary and checked every day. As he was checking it every day, he wished his wife on the big day. This worked well, so he got the make-it-large idea.

He made a website and as it was getting popular, the request of 1 user per second was not working as many people were in the waiting line. He got stressed out as there is lot of money but there is no rest for him. Now, his clever-wife came up with an idea of making two separate phones with a load balancer and it will send the user either to you or to me. We will have our own separate diaries and will enter the details.

If the husband is sleeping then his wife can continue the work. The website is available now in every case.

Issue: the client A gave his details to the guy, but today his call got connected to his wife, who did not record his details. This brings **consistency issues**. As the data is not shared between them this occurs.

Every distributed system is **partition tolerant**, because if one system is not working still the service will be available.

To cover come the consistency issues, the couple decided to share the details. The moment he got the details the couple called each other and shared the details. If they are on call, then users will not be able to contact any of them. This brings **availability problem**.

This is called CAP Theorem; in a distributed system you cannot have all the three things.

CAP Theorem:

The cap theorem is a tool used to makes system designers aware of the trade-offs while designing networked shared-data systems. Cap has influenced the design of many distributed data systems. It made designers aware of a wide range of trade-offs to consider while designing distributed data systems. Over the years, the cap theorem has been a widely misunderstood tool used to categorize databases. There is much misinformation floating around about cap. Most blog posts on cap are historical and possibly incorrect.

It is important to understand cap so that you can identify the misinformation around it. The Cap Theorem applies to distributed systems that store state. Eric Brewer, at the 2000 symposium on principles of distributed computing (podc), conjectured that in any networked shared-data system there is a fundamental trade-off between consistency, availability, and partition tolerance. In 2002, Seth Gilbert and Nancy Lynch of MIT published a formal proof of Brewer's Conjecture. The theorem states that networked shared-data systems can only guarantee/strongly support two of the following three properties:

- Consistency — a guarantee that every node in a distributed cluster returns the same, most recent, successful write. Consistency refers to every client having the same view of the data. There are various types of consistency models. Consistency in cap (used to prove the theorem) refers to linearizability or sequential consistency, a very strong form of consistency.
- Availability — every non-failing node returns a response for all read and write requests in a reasonable amount of time. The key word here is ever. To be available, every node on (either side of a network partition) must be able to respond in a reasonable amount of time.
- Partition tolerant — the system continues to function and upholds its consistency guarantees in spite of network partitions. Network partitions are a fact of life. Distributed systems guaranteeing partition tolerance can gracefully recover from partitions once the partition heals.

https://mwhittaker.github.io/blog/an_illustrated_proof_of_the_cap_theorem/