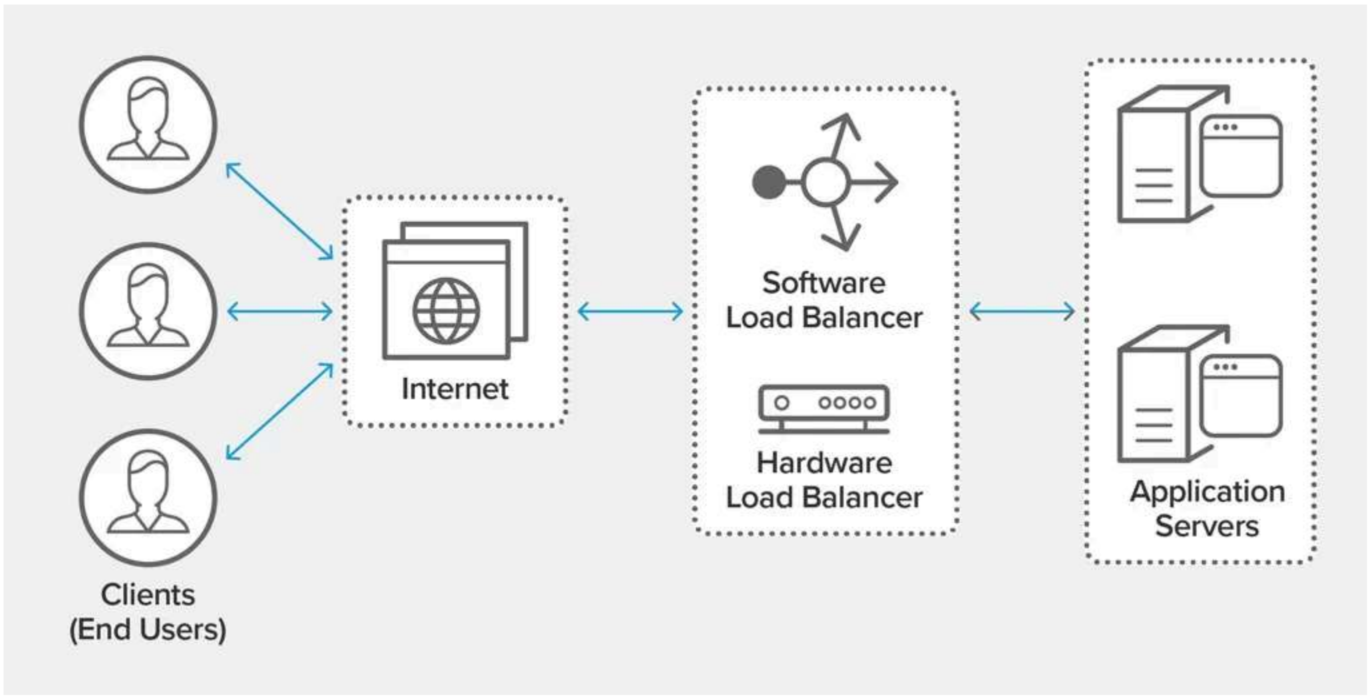
Load Balancing and Consistent Hashing

Kasun Dissanayake · Follow
Published in The Startup · 7 min read · Jul 12, 2020

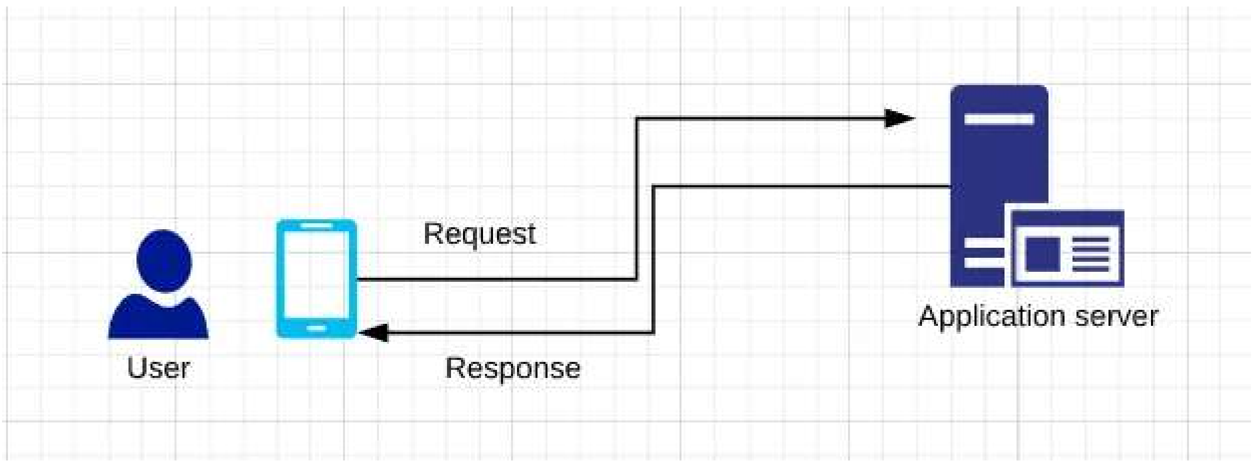
784 2



In this tutorial, we are going to learn about Load Balancing. This is an important thing that you need to know when you are building systems. Let me get a simple example for explaining the concept of Load

And assume that there is a person with a mobile phone. He can connect to your machine and use the **algorithm**. He only uses your algorithm and gets the result. And every time he does that he pays you some **money**. But keep the money, marketing, and other things aside. We have **technical specifications.** This algorithm needs to run. Then the customer to be happy.

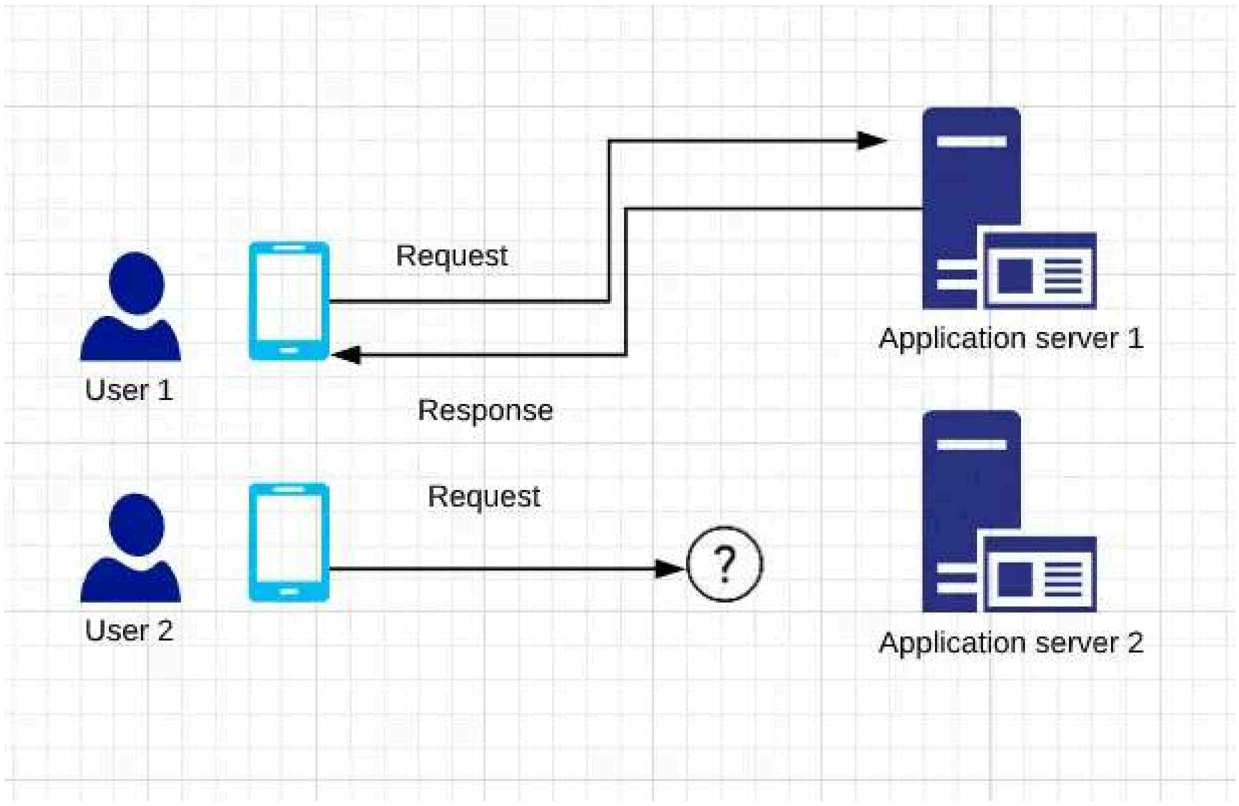
Now lets come into the **real-life scenario**. This algorithm is like a **server**. The server is something that servers **request**. When a person connecting through mobile technically he sends a request. Then your algorithm runs. You understand what he wants and the algorithm can be any algorithm. It can be a **face recognition** algorithm and sending an **image** as a response.



Request-Response

So your server gets **requests** and sends back the **response.** Assume that you have thousands and thousands of requests (**many users**) coming in. Now your computer cannot handle this anymore. So what you can do is **add another computer.** Because you are getting a lot of money from these people now you can afford to buy a new **computer or a server.** If we can do this there is one problem.

Where do you send the requests, If there are multiple Users?



Should the **User 2** request go to **Application Server 1** or **Application Server 2**?

At the very minimum level if you have **N servers** you want in general balance a load of all these servers. Now imaging all of these servers carrying a load. The requests are the things they need to process. So the server has the **Load.**

So the taking **N servers** and trying to balance the load among of all them is called **Load Balancing.** The concept of **Hashing** will help to implement this concept.

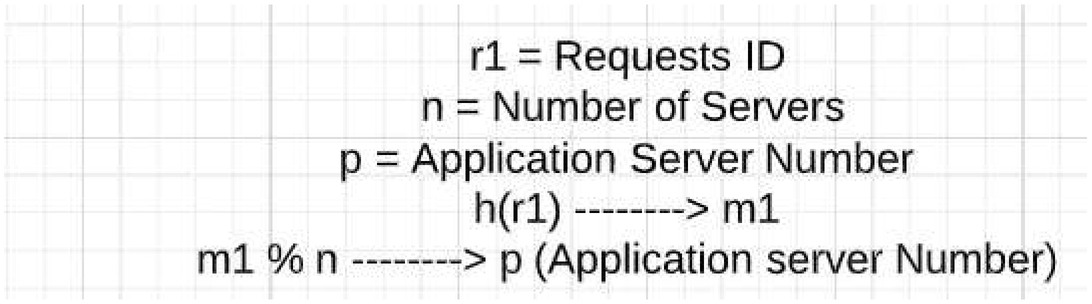
Distribute weight using Hashing

So you want evenly distribute the weight across all servers. Assume there is a **Request ID** for each and every request. And RequestID you can expect **uniformly random.** When Mobile(Client) sends a request to the server it randomly generates a **number(0 to m1 number)**. You can hash the **Request ID using a hash function(h(r1)).**

For example, a hash function can be used to map random numbers to some **fixed number between 0 … m1**. Given any number, it will always try to map it to any integer between 0 to m1.

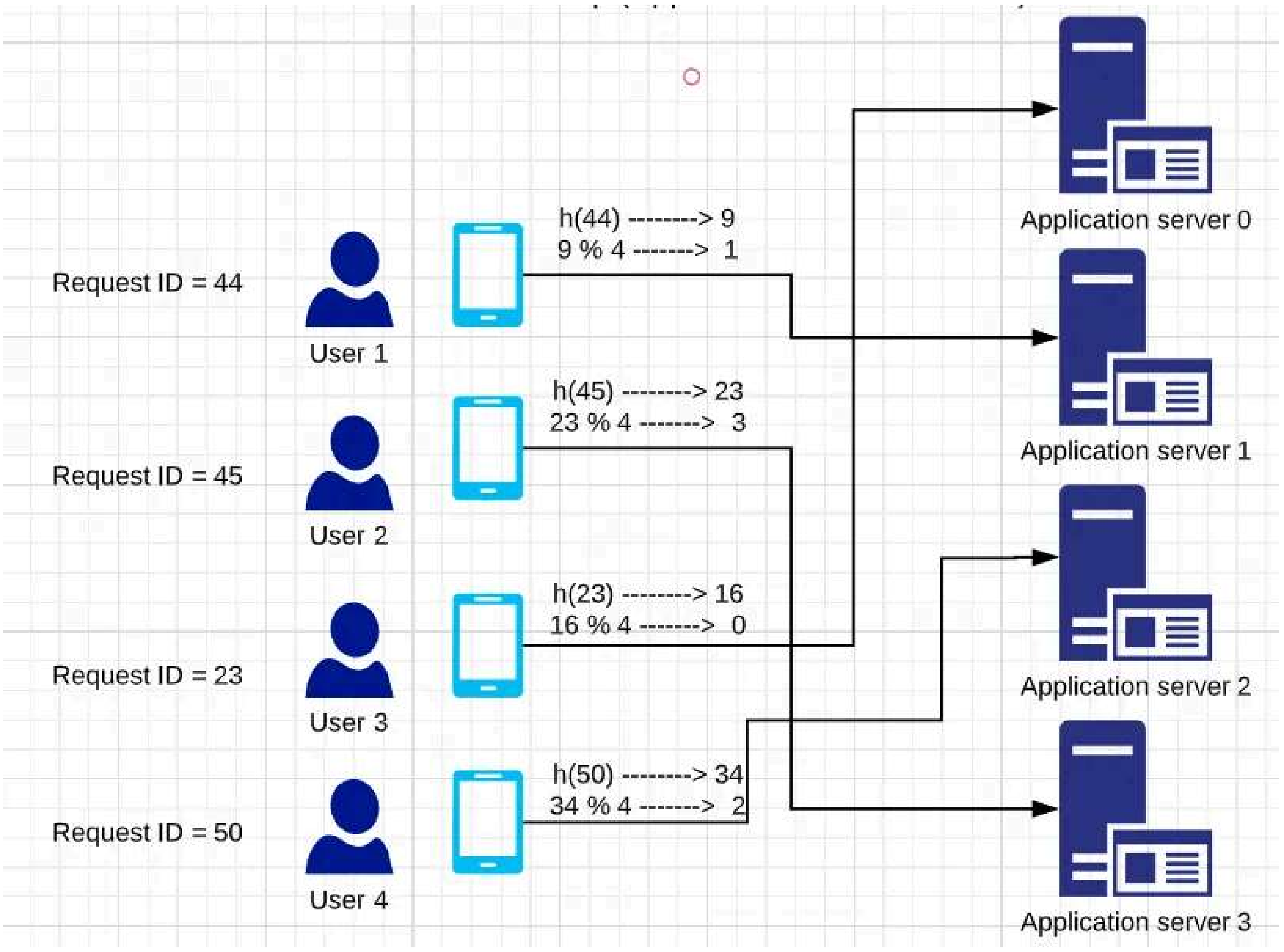
> *Suppose m1 is 50. Then for example, for any hash function will always return a value between 0 to 50.*

When you hash it you get a particular **Value(m1).** This Value/Number can be mapped to a particular server after getting the remainder with **n servers.**



r1 = Requests ID
n = Number of Servers
p = Application Server Number
h(r1) --------> m1
m1 % n --------> p (Application server Number)

Example :



Here your hash function is uniformly random. You can expect all the servers to have a uniform load. So each of the servers will have

```
(Number of Requests)/n
```

load and the **Load Factor** is 1/n.

So everything perfect and that's all we need to do. But what happens if we need to add more servers if the users hitting more and more requests. Assume we need to add one more server to this system. Now User **1, 2, 3, and 4 requests** assign to **different servers**. As an example, **User 1** Request ID is **44.** Now the hash value is **9.** Since there are **5 servers** now the value of the remainder is **4.** Which means that User Request goes to Application **Server 4.** Like this, all the Request path to servers will change.

So the cost of the change in this is **high**. Why such a big deal?. In practice, the **Request ID** is never Random or it is rarely Random. Request ID usually encapsulates some of the information of the User. For example **User ID.**

```
h(Encapsulate information with User ID) ----> m1
```

If the **Request ID** is the same then the hash value will be the **same** again and again since we are using the **same** hashing algorithm.

```
m1%n ----> p
```

If the **hash value(m1)** is the same then the request going send to the **same** server again and again. If I am being sent to the **same** server again and again why should I so that **all the time**?. Why not store it in the **local cache**?

Depending on the **User ID** we can send requests to a specific server(Ex: User 1 request should go to Server 1) and once we sent them you can store relevant information in the **cache** on those servers.

In the above caching mechanism what going to happen is the entire system changes all Users now need to go to different servers and all the useful cache information you had is going to useless because the numbers of the servers which you are serving completely changed. This is called **Rehashing.**

The solution is Consistent Hashing

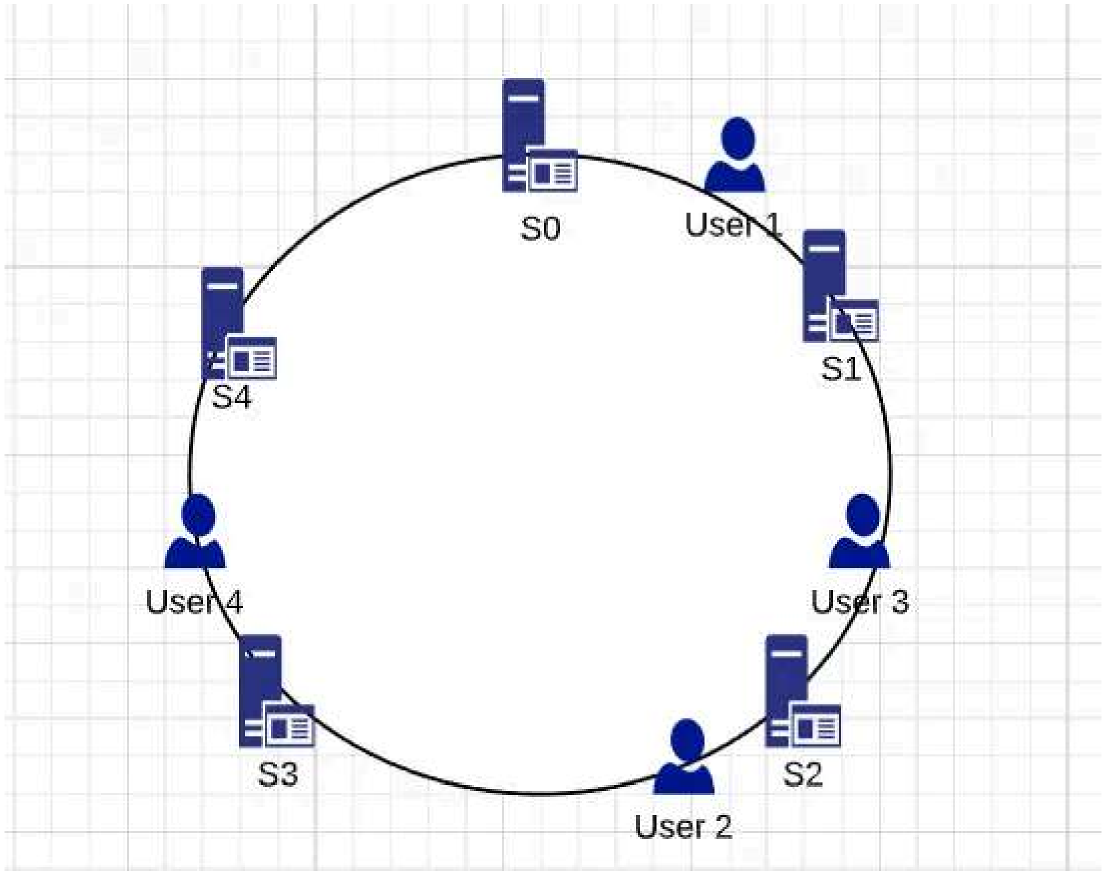
Consistent hashing solves the problem of rehashing by providing a distribution scheme which does not directly depend on the number of servers.

Consistent Hashing is a **distributed hashing scheme** that operates independently of the number of servers or objects in a **distributed hash table** by assigning them a position on an **abstract circle, or *hash ring***. This allows servers and objects to scale without affecting the overall system.

Suppose our hash function output range in between **zero** to **2**32** or **INT_MAX**, then this range is mapped onto the hash ring so that values are wrapped around. All keys and servers are hashed using the same hash function and placed on the edge of the circle. To find out which server to ask for a given key or store a given key, we need to first locate the key on the circle and move in a clockwise direction until we find a server.

Let's use the above example and place them on the hash ring. In this case, the **minimum** value on the circle is **0** and the **maximum** value is **50**.

Example:



Now we have 5 servers. According to consistent hashing rule, **User 1 is on server S1, User2 is on S3, User 3 is on S2 and User 4 is on S4 server.**

Inconsistent hashing when a server is removed or added then the only key from that server is relocated. For example, if server **S3** is removed then, all **keys from server S3** will be moved to **server S4** but keys stored on server **S4, S0, S1**, and **S2** are **not relocated**. But there is one problem when server **S3** is removed then keys from **S3** are not equally distributed among remaining servers **S0, S1, S2, and S4**. They were only assigned to server **S4** which will increase the **load** on server **S4.**

To evenly distribute the load among servers when a server is added or removed, it creates a fixed number of replicas (known as virtual nodes) of each server and distributed it along the circle. So instead of server labels S1, S2, S3, and S4, we will have **S00 S01…S09, S10 S11…S19, S20 S21…S29, S30**

S31…S39 and S40 S41…S49. The factor for a number of replicas is also known as ***weight,*** depends on the situation.

All keys which are mapped to replicas **Sij** are stored on server **Si.** To find a key we do the same thing, find the position of the key on the circle, and then move forward until you find a server replica. If server replica is Sij then the key is stored in server **Si.**

Suppose server **S3** is **removed**, then all **S3** replicas with labels **S30 S31 … S39** must be removed. Now the objects keys adjacent to **S3X** labels will be automatically re-assigned to **S4X** and **S2X.** All keys originally assigned to **S4** and **S2** will not be moved.

Similar things happen if we add a server. Suppose we want to add a server S5 as a replacement of S3 then we need to add labels **S50 S51 … S59**. In the ideal case, one-third of keys from **S4** and **S2** will be reassigned to **S5.**

> **In general, only the K/N number of keys are needed to remapped when a server is added or removed. K is the number of keys and N is the number of servers (to be specific, maximum of the initial and final number of servers)**

I hope you get an idea of **Load Balancing and Consistent Hashing.** See you in the next tutorial.

Thank You!

Load Balancing Load Balance Consistent Hashing Hashing Rehashing

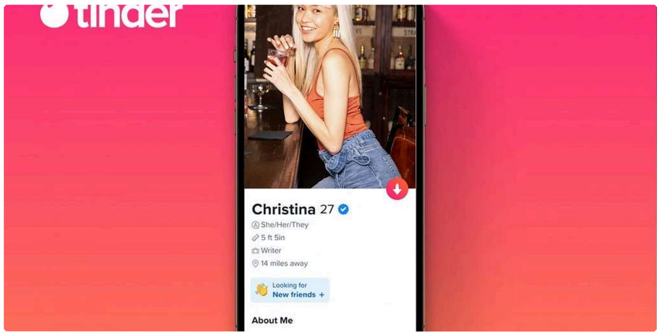
Written by Kasun Dissanayake

Follow

2.5K Followers · Writer for The Startup

Senior Software Engineer at IFS R & D International || Former Software Engineer at Pearson Lanka || Former Associate Software Engineer at hSenid Mobile

More from Kasun Dissanayake and The Startup



Kasun Dissanayake

Tinder — Fully explained System Design and Architecture

In this article, I am going to explain about the interesting topic you all want to know. That is...

16 min read · Nov 14, 2023

609 2

Michael Lim in The Startup

What Sam Altman's Prediction About The $1B One-Person Business Mod...

If you're under 39, you've got a massive advantage.

· 5 min read · Feb 25, 2024

5.8K 155

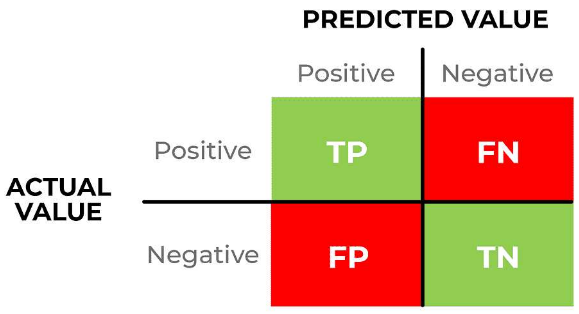
Desiree Peralta in The Startup

This Is the Smartest Move You Can Make With Crypto and Bitcoin Righ...

How to take advantage of the most you can this harvest season without losing everythin...

· 9 min read · Apr 12, 2024

1.2K 24



Kasun Dissanayake in Towards Dev

Machine Learning Algorithms(6)-Metrics for Binary Classification

Classification metrics let you assess the performance of machine learning models bu...

6 min read · Nov 9, 2023

414 1

See all from Kasun Dissanayake

See all from The Startup

Recommended from Medium

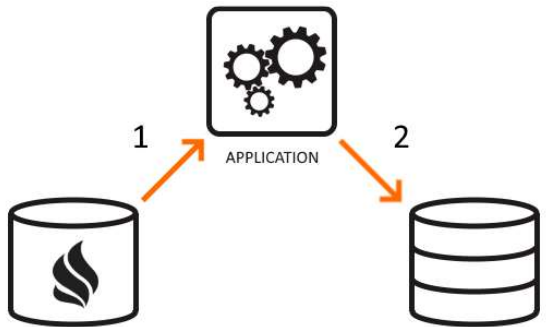
Muttineni Sai Rohith in CodeX

MongoDB Sharding | Sharding vs Replication

As the size of the data increases, a single machine may not be sufficient to store the…

4 min read · Mar 3, 2024

11



Barak Lagziel

In-Memory vs. Distributed Caching: A Comparative Look with Caffeine

In the fast-paced world of software development, caching is a cornerstone…

7 min read · Feb 2, 2024

4

Lists

Staff Picks

632 stories · 928 saves

Stories to Help You Level-Up at Work

19 stories · 583 saves

Self-Improvement 101

20 stories · 1698 saves

Productivity 101

20 stories · 1573 saves

Mohit Sharma

Caching

Caching is a technique used in computing to temporarily store frequently used data or...

19 min read · Dec 26, 2023

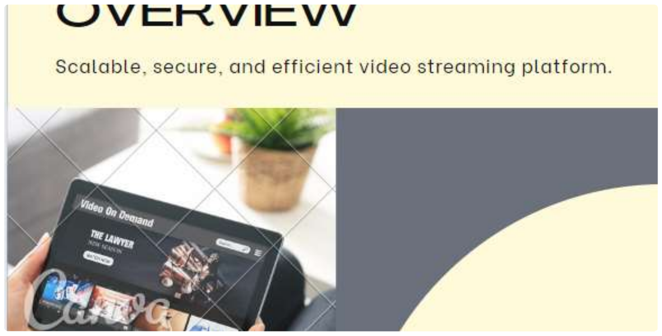
System Design By CHK

System Design — A Deep Dive into the Food Ordering System

In a world where digital interfaces seamlessly connect consumers with their desires, the...

6 min read · Dec 16, 2023

148 5

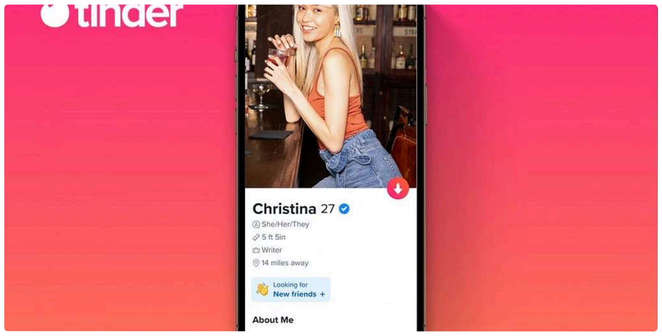


Kabeer Mahmood

High-Level System Architecture of Video Streaming App...

Hello everyone! In this article, we will take an in-depth look at the possible high-level...

4 min read · Apr 17, 2024



Kasun Dissanayake

Tinder — Fully explained System Design and Architecture

In this article, I am going to explain about the interesting topic you all want to know. That is...

16 min read · Nov 14, 2023

2

609 2

See more recommendations