System Design

System design is the process of defining the architecture, components, modules, interfaces, and data for a software system or application to meet specified requirements. It involves deciding how the system will be structured, how its various parts will interact, and how it will perform its intended functions efficiently and reliably.

Before starting designing you need to keep in mind-
Requirements Analysis: Understand the functional and non-functional requirements of the system.
Architecture Design: Decide on the overall system architecture, such as monolithic, microservices, or hybrid. Identify key components or modules and their relationships.
Data Design: Decide on data storage technologies (RDBMS, NoSQL databases, etc.).
Component Design: Break down the system into smaller components or modules. Consider how components will communicate with each other.

Performance Optimization and Scalability: Plan for scalability by considering load balancing, replication, and distribution without degrading the performance.

System Design fundamentals

Performance tuning

Improve existing code base

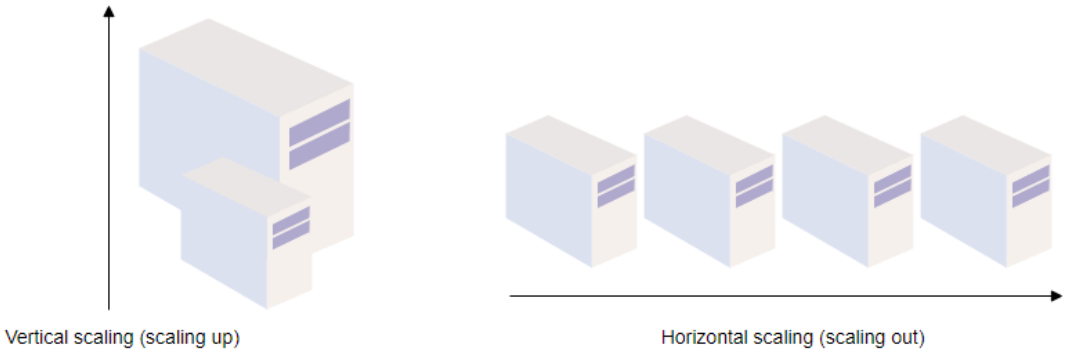
1. Refactoring
2. Convert unnecessary synchronous functions to asynchronous.
3. Improve code logic
4. Caching

Caching

https://medium.com/codex/server-side-caching-in-web-applications-a9145be1cfa0

Horizontal and vertical scaling

Scalability refers to an application's ability to handle and withstand an increased workload without sacrificing latency.
Vertical: update the specs/ power of the server
Horizontal: add more server
When to use which one? Cost and user

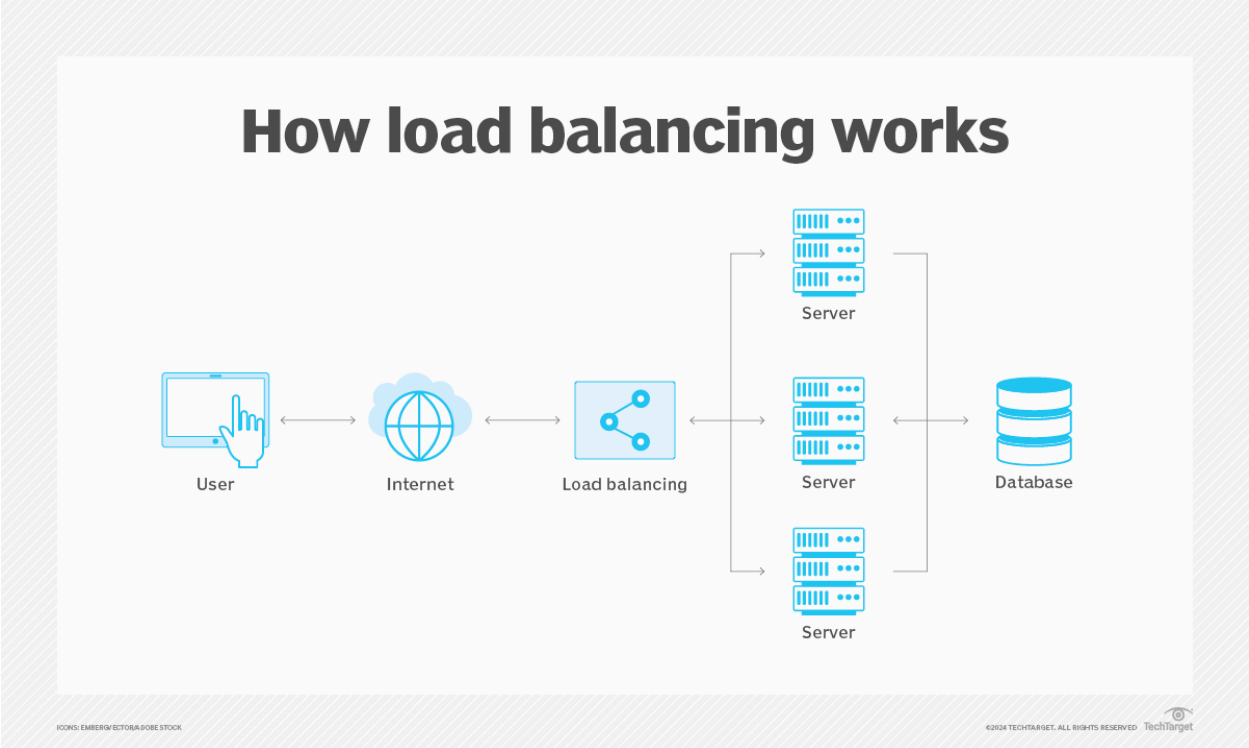


Load balancing:

https://aws.amazon.com/what-is/load-balancing/#:~:text=Load%20balancing%20is%20the%20method,a%20fast%20and%20reliable%20manner.

How load balancing works

Load balancers handle incoming requests from users for information and other services. They sit between the servers that handle those requests and the internet. Once a request is received, the load balancer first determines which server in a pool is available and online and then routes the request to that server. During times of heavy loads, a load balancer acts promptly and can dynamically add servers in response to spikes in traffic. Conversely, load balancers can drop servers if demand is low.



Types of load balancers

Hardware load balancers

A hardware-based load balancer is a hardware appliance that can securely process and redirect gigabytes of traffic to hundreds of different servers. You can store it in your data centers and use virtualization to create multiple digital or virtual load balancers that you can centrally manage.

Pros

- They provide fast throughput, as the software is run on specialized processors.
- These load balancers offer better security, as they're handled only by the organization and not by any third party.
- They come with a fixed cost at the time of purchase.

Cons

- Hardware load balancers require extra staff and expertise to configure and program them.
- They can't scale when a set limit on several connections has been reached. When this happens, connections are either refused, dropped or degraded, and the only option is to purchase and install additional machines.
- They're more expensive, as the cost of their purchase and maintenance is higher. Owning a hardware load balancer might require hiring consultants to manage it.

Software load balancers

Software-based load balancers are applications that perform all load balancing functions. You can install them on any server or access them as a fully managed third-party service.

Pros

- They offer the flexibility to adjust to the changing needs and requirements of a network.
- By adding more software instances, they can scale beyond the initial capacity.
- They offer cloud-based load balancing, which provides off-site options that can operate on an elastic network of servers. Cloud computing also offers options with various combinations, such as hybrid with in-house locations. For example, a company could have the main load balancer on premises, and the backup load balancer could be in the cloud.

Cons

- When scaling beyond capacity, software load balancers might cause an initial delay. This usually happens when the load-balancer software is being configured.
- Since they don't come with a fixed upfront cost, software load balancers can add ongoing costs for upgrades.

Comparison of hardware balancers to software load balancers

Hardware load balancers require an initial investment, configuration, and ongoing maintenance. You might also not use them to full capacity, especially if you purchase one only to handle peak-time traffic spikes. If traffic volume increases suddenly beyond its current capacity, this will affect users until you can purchase and set up another load balancer.

In contrast, software-based load balancers are much more flexible. They can scale up or down easily and are more compatible with modern cloud computing environments. They also cost less to set up, manage, and use over time.

Cloud-based load balancing

Uses the cloud as its underlying infrastructure to balance cloud computing environments.
Network load balancing. This is the fastest load-balancing option available. It operates on Layer 4 of the OSI model and uses network layer information to transport network traffic.
HTTP Secure load balancing. This enables network administrators to distribute traffic based on information coming from the HTTP address. It's based on Layer 7 and is one of the most flexible load-balancing options.
Internal load balancing. This is similar to network load balancing, but it can also balance traffic distribution across the internal infrastructure.

Load-balancing algorithms

https://aws.amazon.com/what-is/load-balancing/#seo-faq-pairs#what-are-lb-algo
A set of rules that a load balancer follows to determine the best server for each of the different client requests. Load-balancing algorithms fall into two main categories.

Static load balancing

Static load balancing algorithms follow fixed rules and are independent of the current server state. The following are examples of static load balancing.

Round-robin method

In the round-robin method, an authoritative name server does the load balancing instead of specialized hardware or software. The name server returns the IP addresses of different servers in the server farm turn by turn or in a round-robin fashion.

Example Scenario:

Imagine you have three servers: Server A, Server B, and Server C.

- **Request 1** → Assigned to **Server A** (pointer moves to B)
- **Request 2** → Assigned to **Server B** (pointer moves to C)
- **Request 3** → Assigned to **Server C** (pointer moves back to A)
- **Request 4** → Assigned to **Server A** (and so on)

Weighted round-robin method

In weighted round-robin load balancing, you can assign different weights to each server based on their priority or capacity. Servers with higher weights will receive more incoming application traffic from the name server.

IP hash method

In the IP hash method, the load balancer performs a mathematical computation, called hashing, on the client IP address. It converts the client IP address to a number, which is then mapped to individual servers.

Dynamic load balancing

Dynamic load balancing algorithms examine the **current state** of the servers **before distributing traffic**. The following are some examples of dynamic load balancing algorithms.

Least connection method

The load balancer checks which servers have the fewest active connections and sends traffic to those servers. This method assumes that all connections require equal processing power for all servers.

Benefits: Effective for balancing loads in environments with varying connection lengths and server capabilities.

Use Case: Suitable for applications with persistent connections, like chat services or video streaming.

Weighted least connection method

Weighted least connection algorithms assume that some servers can handle more active connections than others. Therefore, you can assign different weights or capacities to each server, and the load balancer sends the new client requests to the server with the least connections by capacity.

Benefits: Better at balancing traffic in environments with heterogeneous servers (e.g., servers with different CPU or memory capacities).

Use Case: Environments where some servers can handle more traffic than others.

Least response time method

The response time is the total time that the server takes to process the incoming requests and send a response. The least response time method combines the server response time and the active connections to determine the best server. Load balancers use this algorithm to ensure faster service for all users.

Benefits: Ensures that users are directed to servers that can respond the fastest, improving overall performance.

Use Case: Time-sensitive applications where fast response times are critical, like real-time financial trading platforms.

Resource-based method

In the resource-based method, load balancers distribute traffic by analyzing the current server load. Specialized software called an agent runs on each server and calculates usage of server resources, such as its computing capacity and memory. Then, the load balancer checks the agent for sufficient free resources before distributing traffic to that server.

Benefits: Helps distribute load based on actual server performance, ensuring no single server becomes overburdened.

Use Case: Complex applications that consume significant processing power, such as data analysis tools or machine learning applications.

Consistent Hashing (with Load Awareness)

https://medium.com/swlh/load-balancing-and-consistent-hashing-5fe0156035e1

What is CONSISTENT HASHING and Where is it used?

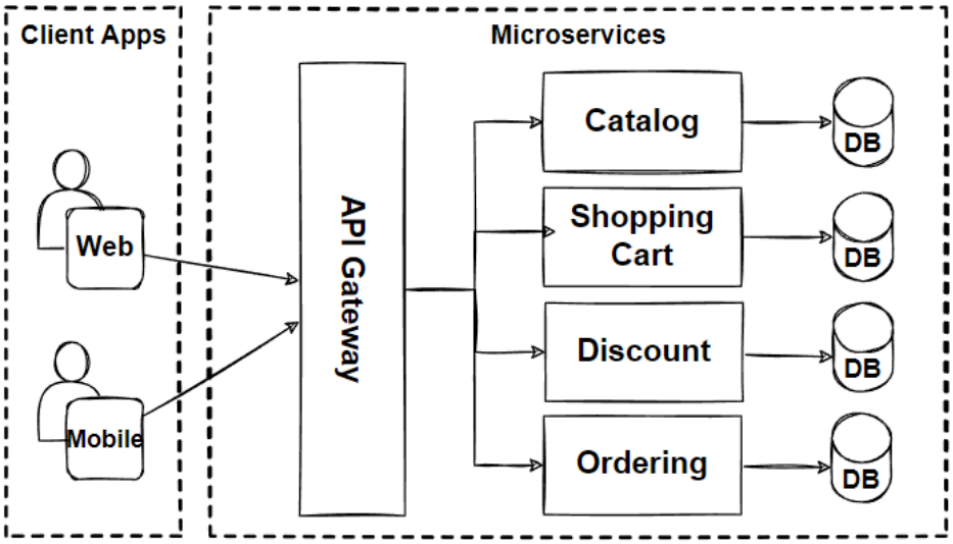
Although consistent hashing is mainly used for data distribution, some advanced implementations incorporate server load awareness. This combination helps distribute new client requests to servers based on their load and hash values.

Benefits: Balances traffic while maintaining stability when servers are added or removed.

Use Case: Distributed systems that require scalable load balancing, such as distributed databases or caching systems.

Monolithic and microservice architecture

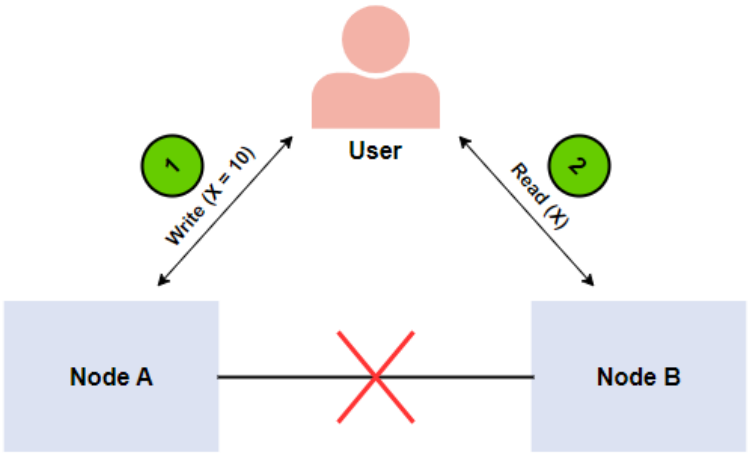
Monolithic: Static content, business logic and database layer. if all of them are in one server. Divide the business logic.



Microservice: structures an application using loosely coupled services. It divides a large application into a collection of separate, modular services. These modules can be independently developed, deployed, and maintained.

CAP theorem

The CAP theorem states that a distributed system can only provide two of three properties simultaneously: **consistency, availability, and partition tolerance**.



There's a network failure that results in a network partition between the two nodes in the system. An end-user performs a write request, and then a read request. Let's examine a case where a different node of the system processes each request. In this case, our system has two options:

It can fail at one of the requests, breaking the system's **availability**

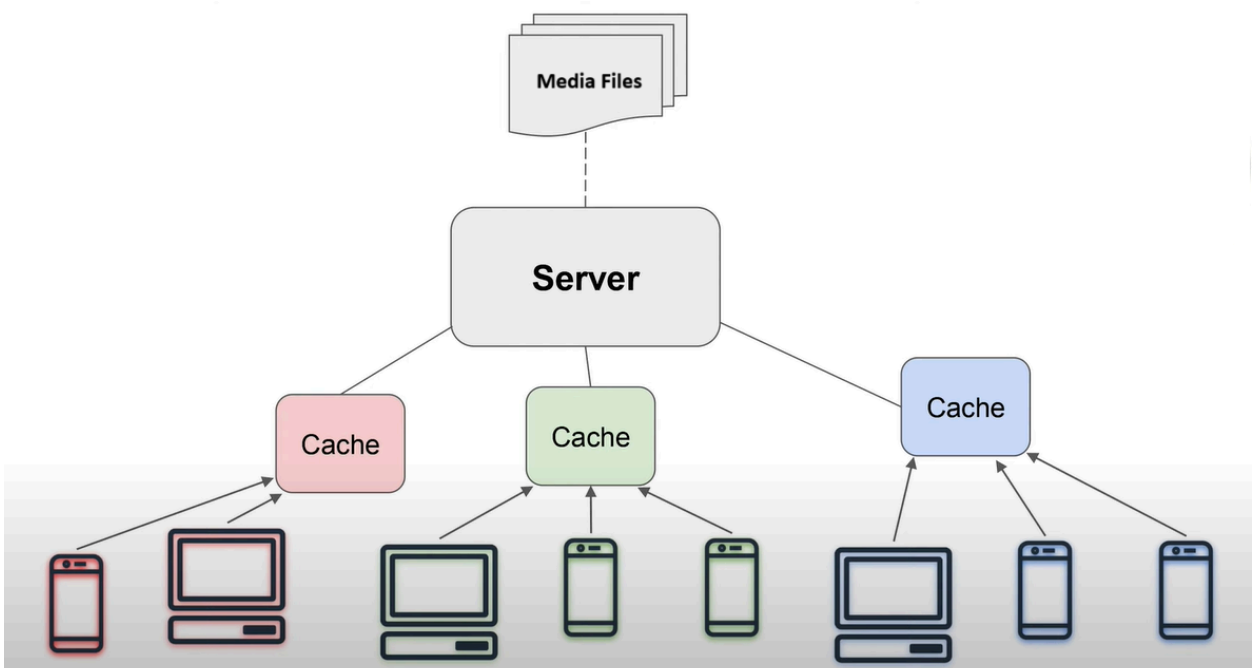
It can execute both requests, returning a stale value from the read request and breaking the system's **consistency**

Message queues

A message queue is a queue that routes messages from a source to a destination, or from the sender to the receiver. It follows the FIFO (first in first out) policy. asynchronous behavior, which allows modules to communicate with each other in the background without hindering primary tasks.

Example: Apache Kafka started in 2011 as a messaging system for LinkedIn. It can handle trillions of records per day. Kafka is a distributed system consisting of servers and clients that communicate through a TCP network protocol. rabitMQ, kinesis, flink.

CDN (Content Delivery Network)



CDN stores Static Files (Image, Video, Pdf). When to refresh the cache? ➔ TTL (Time To Live). Suppose, TL is fixed to 7 days (can be varied), now after 7 days, CDN will contact the main server to check whether the data has been modified or not. If it's modified, then CDN will get the updated info.

Two types of CDN -

Push CDN

Main server sends the updated data to the Proxy server?CDN whenever there's a change. Advantages: Users can get updated content all the time. Suitable for - Sites with small traffic. If the Server has fewer load, Push CDN is good. Also, Static files frequently update

Pull CDN

Fetch new content from server when the user first requests it. TTL determines for how long content is cached.

Advantages: Main server er load kome jacche. Age ekta content change hoilei main server ke ekta response pathano lagtesilo proxy server/CDN ke. Suitable for - Sites with high traffic.

Disadvantages: Suppose ekta image er TTL 7 days. But user oita 5 days por delete kore dilo. Still proxy server e image ta aro 2 days theke jabe, and oi region er onno user ra shei expired content dekhbe.

Pros:

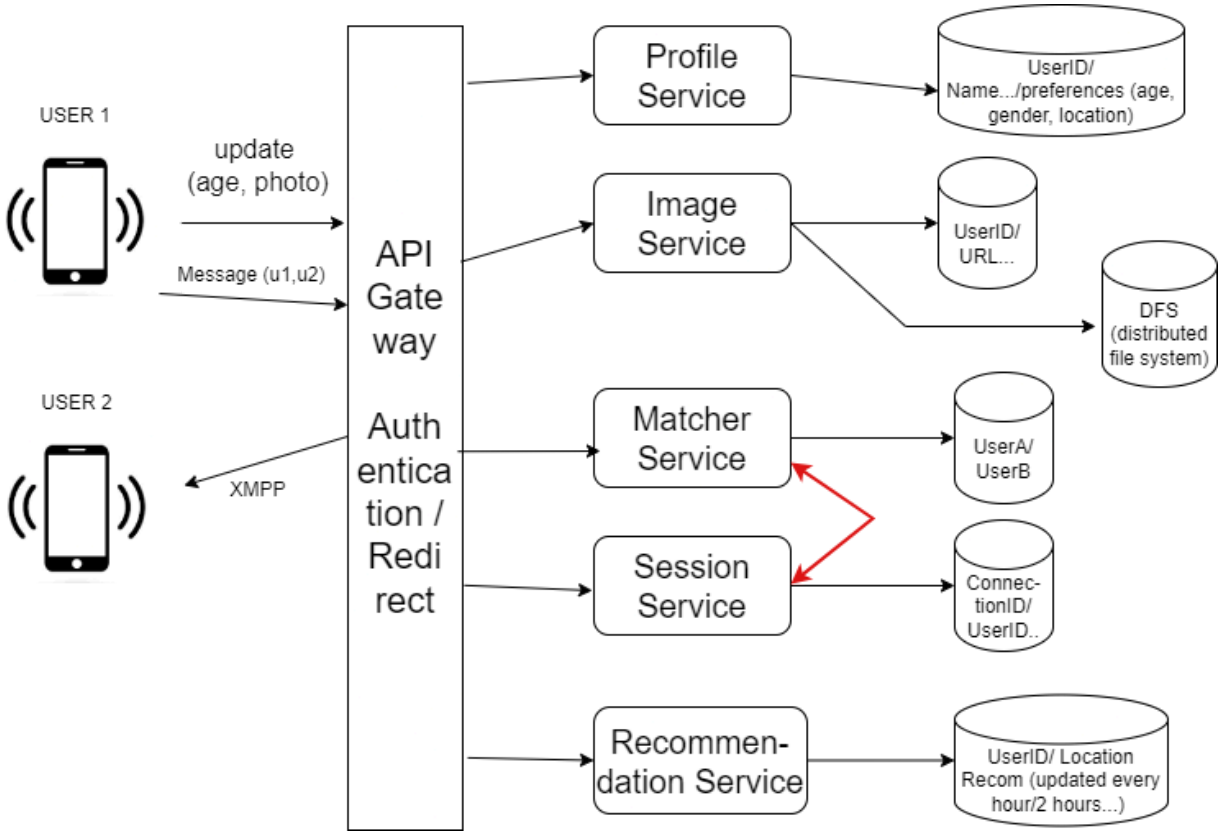
- Faster ⇒ Less Round trip time
- Regulations ⇒ Area wise content selection
- Relevance ⇒ Since this is a cache, it can't store all the movies that you have in your file store behind, you're going to keep only relevant data in these local caches.
- Less load ⇒ Age main server e millions of request ashto. Kintu ekhon jotogula proxy server, totogula request ashe.
- Security ⇒ CDN is provided by Amazon, Cloudflare, etc. which provide additional firewalls/ security.

Cons.

- Cost
- Require Changing URLs ⇒ For general user, we can store images in CDN/ local regional server. What about celebrities? We need to store the images in multiple CDNs, so need multiple URLs
- Expired Content ⇒ Disadvantage of Pull CDN

System design of Tinder:

Functional Requirement	Non-functional Requirement
1. Create a profile (Store preference, profile image) 2. Recommend match 3. Note down matches 4. DM (if 2 users don't get matched, then they can't communicate.)	1. The system should have strong consistency for swiping. If a user swipes "yes" on a user who already swiped "yes" on them, they should get a match notification. 2. The system should scale to lots of daily users / concurrent users (20M daily actives, ~100 swipes/user/day on average). 3. The system should load the potential matches stack with low latency (e.g. < 300ms). 4. The system should avoid showing user profiles that the user has previously swiped on.



Store Profile (preference: age, gender, location; max 5 image)

API Gateway: The client only talks to the gateway. authenticate and redirect to the correct service based on the request.

How to store Images?

Based on the requirements, choose either file or blob.

Blob: DB⇒ ACID; Much easier to do full backups through the index.

File: store image URL; if your application uploads a large number of high-quality images, DB backups become huge; FS can be accessed via a CDN without allowing the user's request to go through your Application and Database layers. Store / retrieve easy.

➔ File vs Blob

Mutability- We don't usually change part of the image, we simply create a separate file and discard the old image. If we need to change only part of an image, then BLOB ok, else File is better.

Transaction - It's not a secured transaction, even if the system crashes, the DB inconsistency won't be a big issue. Hence, File is a better choice.

Indexes - We are not searching based on images.. So no need for indexing

Access Control - BLOB provides it better, however we can set up equally good File system as well. It's just a bit tedious.

File better cz- Cheaper, Faster a bit (storing large object separately - Image service), Static - so apply CDN.

Note Down Matches

Table e userID A , userID B thakbe. That means A , B er sathe match korse.
Why is it necessary? Bcz ekbar jehetu match korse, ekhon ami ar A ke B er kase recommend korbo na.

Now if a user reinstalls the app, then matcher service will pull up all the matches, profile/image service will have the data. The only information we'll lose is the number of left/right swipes. It's not something important. We can show the same recommendation multiple times.

DM

Normally we use client-server communication (http)- clients send req msg to server but not vice versa. But in the case of chat, u1 sends msg to server, how will u2 get the msg? One way to constantly ask the server if there's any msg. Extremely inefficient. Another option is peer to peer protocol (xmpp). All are equal. Server can send msg to clients.

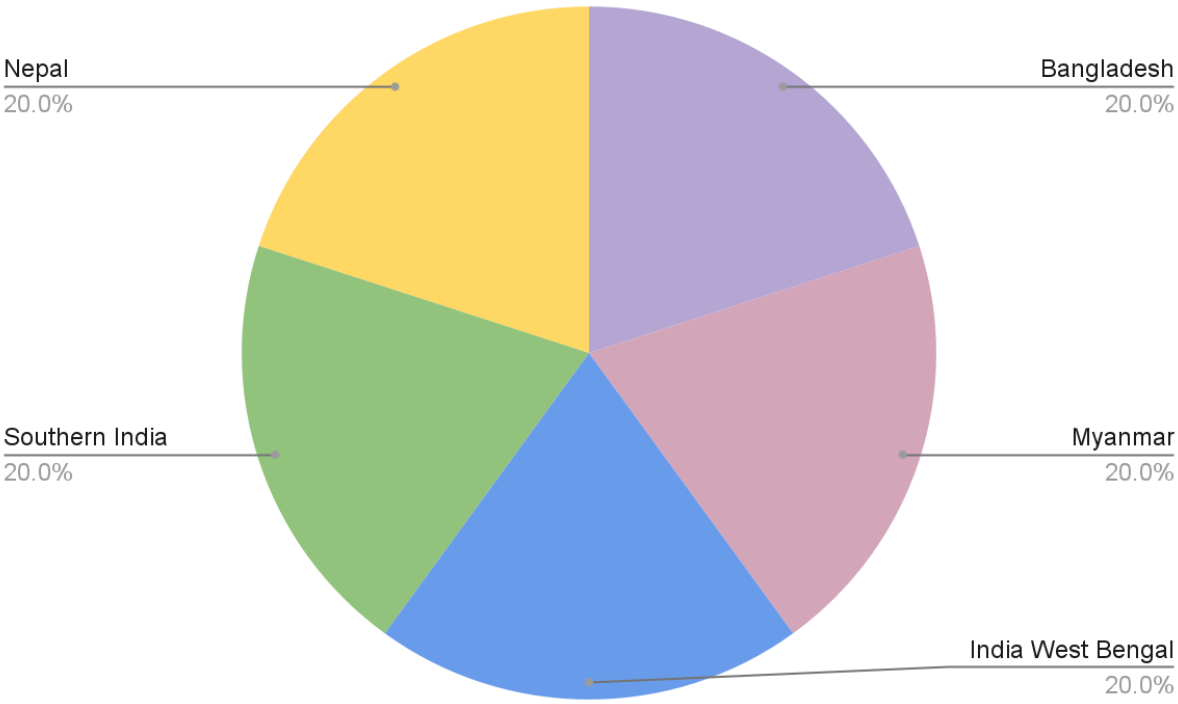
Session Service

However, there are a lot communication among user at the same time. **How to know who is communicating with whom?** XMPP uses a connection (TCP/ Web-socket). Use a session service (store connection ID [TCP connection ID] and user ID) . first time, user 1 sends msg (msg(u1,u2)) and session service finds u2 and store its id+ connectionID. Then user 1 sends a msg req and session service checks who else using the same connection ID and sends the msg. can store msg in session service as well. But before sending any msg, 1st chech the matcher service to verify whether they can DM or not.

Recommendation Engine:

The biggest problem with recommendation will be to figure out who are the users close to me? So I can figure out quite easily which genders I'm interested in. So, the profile service can contain preferences- age, gender, location etc. Using this info, we need to sort and search. We can use indexing, but where to put that index? You cannot have multiple indexes. You cannot have the data sorted in multiple ways in one database table. So in this case, what happens is you need to use a NoSQL database like **Cassandra**, which is good at querying for this kind of data type. You just replicate the data in multiple places, and depending on the query, you build a table on that query, and then you can have an efficient query.
What if a person is not very comfortable moving onto a distributed database? We can use the same concepts kind of on a relational database. And that requires something called **sharding/ Horizontal Partitioning** (details will be covered later).

Horizontal Partitioning: Divide the database based on the location. Each country/ region will be a different database table. So when someone is searching for a person, we can simply look into that country-specific table instead of the whole DB.



What if the user lives in the border region?
Then we need to perform join operations.

System design of Instagram:

Functional Requirements:

- Store/get image
- like/comment (support recursive comments)
- Follow someone
- News feed

Store Image - Use CDN

like/comment (support 1 layer recursive comments)

Design the DB tables:

Post/ Comment table:

POST

ID	UserID	Text	Image URL	TimeStamp

Likes table:

LIKES

ID	PostID	UserID	TimeStamp	Active	Type
				T/F	Comment/Post

PostID: it can be a comment/ post

Active: bhule like diye dile? Hence true/false.

Type: It can be a comment or a post. **like** ta post er naki comment er sheta janar jonno whole id er upor ekta string operation korte hobe, which is very expensive. Hence this field.

To count how many like/ comments a post has, we can run a query, but the problem is that instagram has a lot of active users at the same time. So to run this query more efficiently, we can create another table that'll keep track of the number of like/comment.

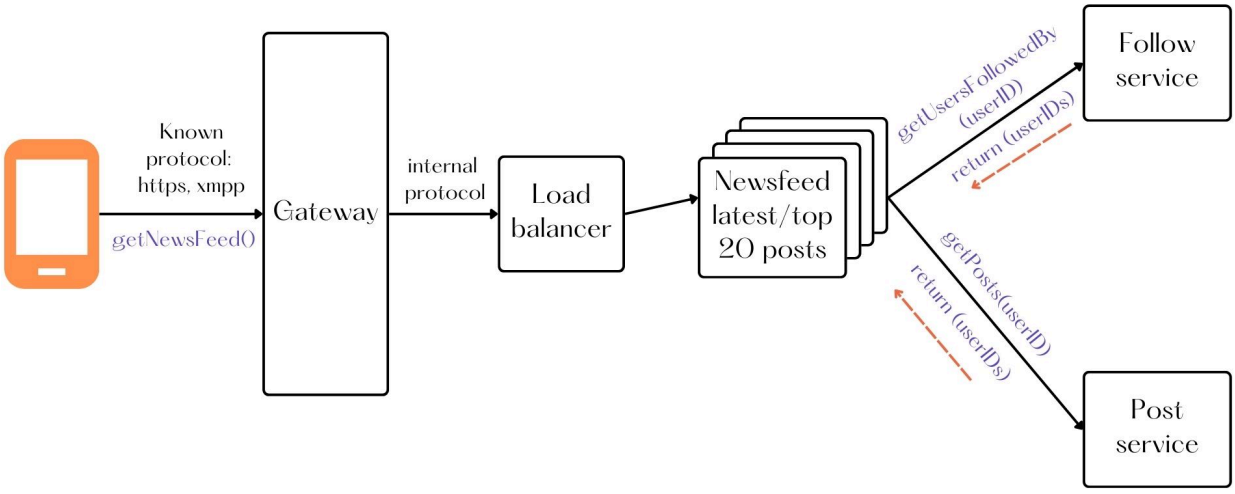
ACTIVITIES

ID	ParentID	Likes	Counts
	Post/Comment		

FOLLWOERS

ID	FollowerID	FolloweeID	TS

Overall System design:



The **Gateway Server** will receive the username, password, token, etc., from the user for authentication. Alternatively, users may simply send messages to each other (e.g., User1 to

User2). The **Gateway Server's** role is to authenticate users and route requests to the appropriate service.

Since Instagram is a mobile application, it needs a way to connect to our server side. Let's assume that there's a **gateway** which is going to be **encapsulating** a lot of things.

One important thing that the gateway can do is it can take external protocols. It can take http/xmpp protocols from client side and use another sort of internet protocol for the microservices. Why do this? -To increase security so that others don't know which protocol you are using.

A **Load Balancer** will determine which **News Feed Service** instance will handle a user's request.

Now, suppose we want to display **20 posts** at a time in the **News Feed**. The **News Feed Service** will send the **user ID** to the **Follow Service**. The **Follow Service** will return the list of **followers' IDs** to the **News Feed Service**.

Then, the **News Feed Service** will send this set of **user IDs** to the **Post Service**. The **Post Service** will return the posts from these users to the **News Feed Service**, which will then display them in the **news feed**.

There are more services similar to this. Example: image, like, comment, activity services.

Optimization:

➔ We will **limit** the number of posts retrieved when refreshing the news feed to a **maximum of 20 posts**.
➔ The **20 posts** displayed in the news feed can be **precomputed** in advance.

For example, we can **maintain a queue** for a group of **four users** (who follow each other or have some connection). When **User 2** post something in their news feed, the **queues of the other users** will get updated.

Next time, when those users refresh their news feed, the **precomputed posts** from the queue will be displayed instead of fetching them again. This process is called **News Feed Precomputation**. As a result, the **server load is significantly reduced**.

This **precomputation** can be stored in **two places**: **Cache & Database**.

- For **frequent users** who refresh their news feed often, we will store their precomputed feed in **cache** for faster access.
- For **less frequent users**, we will store their precomputed feed in the **database** to optimize resource usage.

Notification

Suppose a **user** makes a post, and we want to send **push notifications** to their **followers**. But what if the user is **Kylie Jenner**? We **can't send notifications to millions of followers at once**. What's the solution?

Batch Notification

- We can send notifications in **batches**, e.g.,
    - Every **10 seconds**, send **10,000 notifications**.
    - After the next **10 seconds**, send another **10,000 notifications**, and so on.

Polling

- The **user's device** will check (poll) for notifications **every 1 minute**.
- A notification will be sent **only when the user polls** (checks for new notifications).
- If the user is **offline**, we **don't need to worry** about sending notifications.
- Once they **come online**, they will receive the notifications.