# Topics

02 Scale From Zero To Millions Of Users

03 Back-of-the-envelope Estimation

O4 A Framework For System Design Interviews

13 Design A Chat System

Designing a Rate Limiter

15 Design YouTube

# 02 Scale From Zero To Millions Of Users

Designing a system that supports millions of users is challenging. Start with a single user and gradually scale it up to serve millions of users.

|  |  |
| --- | --- |
| Figure 2 | 1. Users access websites through domain names, such as api.mysite.com..  2. IP address is returned, 3. HTTP requests are sent directly to your web server.  4. The web server returns HTML pages or JSON response for rendering.  The traffic comes from two sources: web application and mobile application.   * Web application: it uses a combination of server-side languages (Java) to handle business logic, storage, etc., and client-side languages (HTML and JavaScript) for presentation. * Mobile application: HTTP protocol is the communication protocol between the mobile app and the web server. JSON is commonly used API response format to transfer data due to its simplicity. |
|  | Database added. Non-relational databases might be the right choice if:   * Your application requires super-low latency. * Your data are unstructured, or you do not have any relational data. * You only need to serialize and deserialize data (JSON, XML, YAML, etc.). * You need to store a massive amount of data. |

## Vertical scaling vs horizontal scaling

|  |  |
| --- | --- |
| Vertical scaling, referred to as “scale up”, means the process of adding more power (CPU, RAM, etc.) to your servers. When traffic is low, vertical scaling is a great option, and the simplicity of vertical scaling is its main advantage. Unfortunately, it comes with serious limitations.   * Vertical scaling has a hard limit. It is impossible to add unlimited CPU and memory to a single server. * Vertical scaling does not have failover and redundancy. If one server goes down, the website/app goes down with it completely. | Horizontal scaling, referred to as “scale-out”, allows you to scale by adding more servers into your pool of resources. Horizontal scaling is more desirable for large scale applications due to the limitations of vertical scaling. |

**Load balancer**

|  |  |
| --- | --- |
|  | A load balancer evenly distributes incoming traffic among web servers that are defined in a load-balanced set.  Users connect to the public IP of the load balancer directly. With this setup, web servers are unreachable directly by clients anymore. For better security, private IPs are used for communication between servers.  The load balancer communicates with web servers through private IPs.  A load balancer and a second web server are added, successfully solved no failover issue and improved the availability of the web tier. Details are explained below:   * If server 1 goes offline, all the traffic will be routed to server 2. This prevents the website from going offline. * If the website traffic grows rapidly, add more servers to the web server pool, and the load balancer automatically starts to send requests to them. |

**Database replication**

|  |  |
| --- | --- |
|  | The data tier? The current design has one database, so it does not support failover and redundancy. Database replication is a common technique to address those problems.  Database replication can be used in many DBMS, usually with a master/slave relationship between the original (master) and the copies (slaves).  A master database generally only supports write operations. A slave database gets copies of the data from the master database and only supports read operations. Most applications require a much higher ratio of reads to writes; thus, the number of slave databases in a system is usually larger than the number of master databases. |
| Advantages of database replication:   * Better performance: In the master-slave model, all writes and updates happen in master nodes; whereas, read operations are distributed across slave nodes. This model improves performance because it allows more queries to be processed in parallel. * Reliability: If one of your database servers is destroyed by a natural disaster, data is still preserved. You do not need to worry about data loss because data is replicated across multiple locations. * High availability: By replicating data across different locations, your website remains in operation even if a database is offline as you can access data stored in another database server.   what if one of the databases goes offline? The architectural design discussed can handle this case:   * If only one slave database is available and it goes offline, read operations will be directed to the master database temporarily. As soon as the issue is found, a new slave database will replace the old one. In case multiple slave databases are available, read operations are redirected to other healthy slave databases. A new database server will replace the old one. * If the master database goes offline, a slave database will be promoted to be the new master. All the database operations will be temporarily executed on the new master database. A new slave database will replace the old one for data replication immediately. In production systems, promoting a new master is more complicated as the data in a slave database might not be up to date. The missing data needs to be updated by running data recovery scripts. Although some other replication methods like multi-masters and circular replication could help, those setups are more complicated. | |

**Cache**

|  |  |
| --- | --- |
| This caching strategy is called a read-through cache.  Other caching strategies are available depending on the data type, size, and access patterns. | It is time to improve the load/response time. This can be done by adding a cache layer and shifting static content (JavaScript/CSS/image/video files) to the content delivery network (CDN).  A cache is a temporary storage area that stores the result of expensive responses or frequently accessed data in memory so that subsequent requests are served more quickly. Cache tier - is a temporary data store layer, much faster than the db. The benefits include better system performance, ability to reduce db workloads, and the ability to scale the cache tier independently. |

### <https://codeahoy.com/2017/08/11/caching-strategies-and-how-to-choose-the-right-one/>

### Considerations for using cache

Here are a few considerations for using a cache system:

* Decide when to use cache. Consider using cache when data is read frequently but modified infrequently. Since cached data is stored in volatile memory, a cache server is not ideal for persisting data. For instance, if a cache server restarts, all the data in memory is lost. Thus, important data should be saved in persistent data stores.
* Expiration policy. Once cached data is expired, it is removed from the cache. When there is no expiration policy, cached data will be stored in the memory permanently. It is advisable not to make the expiration date too short as this will cause the system to reload data from the database too frequently. Meanwhile, it is advisable not to make the expiration date too long as the data can become stale.
* Consistency: This involves keeping the data store and the cache in sync. Inconsistency can happen because data-modifying operations on the data store and cache are not in a single transaction. When scaling across multiple regions, maintaining consistency between the data store and cache is challenging.
* Mitigating failures: A single cache server represents a potential single point of failure (SPOF), if it fails, will stop the entire system from working. As a result, multiple cache servers across different data centers are recommended to avoid SPOF. Another recommended approach is to overprovision the required memory by certain percentages. This provides a buffer as the memory usage increases.
* Eviction Policy: Once the cache is full, any requests to add items to the cache might cause existing items to be removed. This is called cache eviction. Least-recently-used (LRU) is the most popular cache eviction policy. Other eviction policies, such as the Least Frequently Used (LFU) or First in First Out (FIFO), can be adopted to satisfy different use cases.

## Content delivery network (CDN)

|  |  |
| --- | --- |
|  | A CDN is a network of geographically dispersed servers used to deliver static content. CDN servers cache static content like images, videos, CSS, JavaScript files, etc.  Dynamic content caching is a relatively new concept. It enables the caching of HTML pages that are based on request path, query strings, cookies, and request headers. how CDN works at the high-level: when a user visits a website, a CDN server closest to the user will deliver static content. Intuitively, the further users are from CDN servers, the slower the website loads. For example, if CDN servers are in SF, users in LA will get content faster than users in Europe. 1. User A tries to get image.png by using an image URL.  2. If the CDN server does not have image.png in the cache, the CDN server requests the file from the web server or S3.  3. The origin returns image.png to the CDN server, which includes optional HTTP header Time-to-Live (TTL) which describes how long the image is cached.  4. The CDN caches the image and returns it to User A. The image remains cached in the CDN until the TTL expires.  5. User B sends a request to get the same image. 6. The image is returned from the cache as long as the TTL has not expired. |

### Considerations of using a CDN

* + - Cost: charged for data transfers in and out of the CDN. Caching infrequently used assets provides no significant benefits so you should consider moving them out of the CDN.
    - Setting an appropriate cache expiry: For time-sensitive content, setting a cache expiry time is important. The cache expiry time should neither be too long nor too short. If it is too long, the content might no longer be fresh. If it is too short, it can cause repeat reloading of content from origin servers to the CDN.
    - CDN fallback: You should consider how your website/application copes with CDN failure. If there is a temporary CDN outage, clients should be able to detect the problem and request resources from the origin.
    - Invalidating files: You can remove a file from the CDN before it expires by performing one of the following operations:
    - Invalidate the CDN object using APIs provided by CDN vendors.
    - Use object versioning to serve a different version of the object. To version an object, you can add a parameter to the URL, such as a version number. For example, version number 2 is added to the query string: image.png?v=2.

The design after the CDN and cache are added.

Diagram

Description automatically generated

## Stateless web tier

Time to consider scaling the web tier horizontally. For this, we need to move state (for instance user session data) out of the web tier. A good practice is to store session data in the persistent storage such as relational database or NoSQL. Each web server in the cluster can access state data from databases. This is called stateless web tier.

### Stateful architecture

Some key differences. A stateful server remembers client data (state) from one request to the next. A stateless server keeps no state information.

|  |  |
| --- | --- |
| Diagram  Description automatically generated | user A’s session data and profile image are stored in Server 1. To authenticate User A, HTTP requests must be routed to Server 1. If a request is sent to other servers like Server 2, authentication would fail because Server 2 does not contain User A’s session data. Similarly, all HTTP requests from User B must be routed to Server 2; all requests from User C must be sent to Server 3.  The issue is that every request from the same client must be routed to the same server. This can be done with sticky sessions in most load balancers; however, this adds the overhead. Adding or removing servers is much more difficult with this approach. It is also challenging to handle server failures. |



### Stateless architecture

|  |  |
| --- | --- |
| In this stateless architecture, HTTP requests from users can be sent to any web servers, which fetch state data from a shared data store. State data is stored in a shared data store and kept out of web servers. A stateless system is simpler, more robust, and scalable. | Move the session data out of the web tier and store them in the persistent data store. The shared data store could be a relational database, Memcached/Redis, NoSQL, etc. Autoscaling means adding or removing web servers automatically based on the traffic load. |

**Data centers**

|  |  |
| --- | --- |
| To improve availability and provide a better user experience across wider geographical areas, supporting multiple data centers is crucial.  Eample setup with two data centers. In normal operation, users are geoDNS-routed, also known as geo-routed, to the closest data center, with a split traffic of x% in US-East and (100 – x)% in US-West. geoDNS is a DNS service that allows domain names to be resolved to IP addresses based on the location of a user. | In the event of any significant data center outage, we direct all traffic to a healthy data center. In Figure 16, data center 2 (US-West) is offline, and 100% of the traffic is routed to data center 1 (US-East). |

Several technical challenges must be resolved to achieve multi-data center setup:

* Traffic redirection: Effective tools are needed to direct traffic to the correct data center. GeoDNS can be used to direct traffic to the nearest data center depending on where a user is located.
* Data synchronization: Users from different regions could use different local databases or caches. In failover cases, traffic might be routed to a data center where data is unavailable. A common strategy is to replicate data across multiple data centers. A previous study shows how Netflix implements asynchronous multi-data center replication.
* Test and deployment: With multi-data center setup, it is important to test your website/application at different locations. Automated deployment tools are vital to keep services consistent through all the data centers.

**Message queue**

To further scale our system, need to decouple different components of the system so they can be scaled independently. Messaging queue is a key strategy employed by many real-world distributed systems to solve this problem.

A message queue is a durable component, stored in memory, that supports asynchronous communication. It serves as a buffer and distributes asynchronous requests. The basic architecture of a message queue is simple. Input services, called producers/publishers, create messages, and publish them to a message queue. Other services or servers, called consumers/subscribers, connect to the queue, and perform actions defined by the messages.

Icon

Description automatically generated

Decoupling makes the message queue a preferred architecture for building a scalable and reliable application. With the message queue, the producer can post a message to the queue when the consumer is unavailable to process it. The consumer can read messages from the queue even when the producer is unavailable.

Consider the following use case: your application supports photo customization, including cropping, sharpening, blurring, etc. Those customization tasks take time to complete. web servers publish photo processing jobs to the message queue. Photo processing workers pick up jobs from the message queue and asynchronously perform photo customization tasks. The producer and the consumer can be scaled independently. When the size of the queue becomes large, more workers are added to reduce the processing time. However, if the queue is empty most of the time, the number of workers can be reduced.

A picture containing text

Description automatically generated

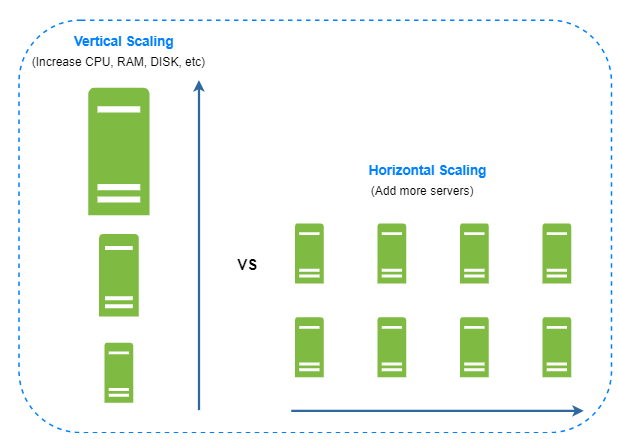
**Logging, metrics, automation**

|  |  |
| --- | --- |
|  | When working with a small website that runs on a few servers, logging, metrics, and automation support are good practices but not a necessity. However, now that your site has grown to serve a large business, investing in those tools is essential.  Logging: Monitoring error logs is important because it helps to identify errors and problems in the system. You can monitor error logs at per server level or use tools to aggregate them to a centralized service for easy search and viewing.  Metrics: Collecting different types of metrics help us to gain business insights and understand the health status of the system. Some of the following metrics are useful:   * Host level metrics: CPU, Memory, disk I/O, etc. * Aggregated level metrics: for example, the performance of the entire database tier, cache tier, etc. * Key business metrics: daily active users, retention, revenue, etc. |
| Automation: When a system gets big and complex, we need to build or leverage automation tools to improve productivity. CI is a good practice, in which each code check-in is verified through automation, allowing teams to detect problems early. Besides, automating your build, test, deploy process, etc. could improve developer productivity significantly.  1. The design includes a message queue, which helps to make the system more loosely coupled and failure resilient.  2. Logging, monitoring, metrics, and automation tools are included. | |

**Database scaling**

As the data grows every day, your database gets more overloaded. It is time to scale the data tier.

There are two broad approaches for database scaling: vertical scaling and horizontal scaling.



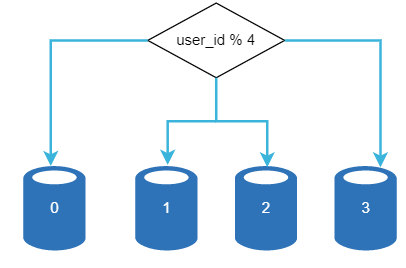
### Vertical scaling: also known as scaling up, is the scaling by adding more power (CPU, RAM, DISK, etc.) to an existing machine. Adding powerful db server could store and handle lots of data. For example, stackoverflow.com in 2013 had over 10 million monthly unique visitors, but it only had 1 master database. Some serious drawbacks:

* You can add more CPU, RAM, etc. to your database server, but there are hardware limits. If you have a large user base, a single server is not enough.
* Greater risk of single point of failures.
* The overall cost of vertical scaling is high. Powerful servers are much more expensive.

### Horizontal scaling: also known as sharding, is the practice of adding more servers. Figure 20 compares vertical scaling with horizontal scaling.

Sharding separates large databases into smaller, more easily managed parts called shards. Each shard shares the same schema, though the actual data on each shard is unique to the shard.

An example of sharded databases. User data is allocated to a database server based on user IDs. Anytime you access data, a hash function is used to find the corresponding shard. In our example, user\_id % 4 is used as the hash function. If the result equals to 0, shard 0 is used to store and fetch data. If the result equals to 1, shard 1 is used. The same logic applies to other shards.

 Diagram

Description automatically generated

The most important factor to consider when implementing a sharding strategy is the choice of the sharding key. Sharding key (known as a partition key) consists of one or more columns that determine how data is distributed. “user\_id” is the sharding key. A sharding key allows you to retrieve and modify data efficiently by routing database queries to the correct database. When choosing a sharding key, one of the most important criteria is to choose a key that can evenly distributed data.

Sharding is a great technique to scale the database but it is far from a perfect solution. It introduces complexities and new challenges to the system:

**Resharding data**: is needed when 1) a single shard could no longer hold more data due to rapid growth. 2) Certain shards might experience shard exhaustion faster than others due to uneven data distribution. When shard exhaustion happens, it requires updating the sharding function and moving data around. Consistent hashing is a commonly used technique to solve this problem.

**Celebrity problem**: also called a hotspot key problem. Excessive access to a specific shard could cause server overload. Imagine data for Katy Perry, Justin Bieber, and Lady Gaga all end up on the same shard. For social applications, that shard will be overwhelmed with read operations. To solve this problem, we may need to allocate a shard for each celebrity. Each shard might even require further partition.

**Join and de-normalization**: Once a database has been sharded across multiple servers, it is hard to perform join operations across database shards. A common workaround is to de-normalize the database so that queries can be performed in a single table.

Shard databases to support rapidly increasing data traffic. At the same time, some of the non-relational functionalities are moved to a NoSQL data store to reduce the database load.

|  |  |
| --- | --- |
|  | **Summary of how we scale our system to support millions of users:**   * Keep web tier stateless * Build redundancy at every tier * Cache data as much as you can * Support multiple data centers * Host static assets in CDN * Scale your data tier by sharding * Split tiers into individual services * Monitor your system and use automation tools   https://netflixtechblog.medium.com/ |

# 03 Back-of-the-envelope Estimation – System capacity

Estimate system capacity or performance requirements using a back-of-the-envelope estimation.

## Power of two

To obtain correct calculations, it is critical to know the data volume unit using the power of 2. Below is a table explaining the data volume unit.

|  |  |  |  |
| --- | --- | --- | --- |
| **Power** | **Approximate value** | **Full name** | **Short name** |
| 10 | 1 Thousand | 1 Kilobyte | 1 KB |
| 20 | 1 Million | 1 Megabyte | 1 MB |
| 30 | 1 Billion | 1 Gigabyte | 1 GB |
| 40 | 1 Trillion | 1 Terabyte | 1 TB |
| 50 | 1 Quadrillion | 1 Petabyte | 1 PB |

## Latency numbers every programmer should know

Some numbers are outdated as computers become faster and more powerful. However, those numbers should still be able to give us an idea of the fastness and slowness of different computer operations.

| **Operation name** | **Time** |
| --- | --- |
| L1 cache reference | 0.5 ns |
| Branch mispredict | 5 ns |
| L2 cache reference | 7 ns |
| Mutex lock/unlock | 100 ns |
| Main memory reference | 100 ns |
| Compress 1K bytes with Zippy | 10,000 ns = 10 µs |
| Send 2K bytes over 1 Gbps network | 20,000 ns = 20 µs |
| Read 1 MB sequentially from memory | 250,000 ns = 250 µs |
| Round trip within the same datacenter | 500,000 ns = 500 µs |
| Disk seek | 10,000,000 ns = 10 ms |
| Read 1 MB sequentially from the network | 10,000,000 ns = 10 ms |
| Read 1 MB sequentially from disk | 30,000,000 ns = 30 ms |
| Send packet CA (California) ->Netherlands->CA | 150,000,000 ns = 150 ms |

Notes

ns = nanosecond, µs = microsecond, ms = millisecond

1 ns = 10^-9 seconds

1 µs= 10^-6 seconds = 1,000 ns

1 ms = 10^-3 seconds = 1,000 µs = 1,000,000 ns

A Google software engineer built (2020) a tool to visualize Dr. Dean’s numbers. The tool also takes the time factor into consideration.

<https://colin-scott.github.io/personal_website/research/interactive_latency.html>

|  |  |
| --- | --- |
| Graphical user interface, application  Description automatically generated | By analyzing the numbers in Figure 1, we get the following conclusions:   * Memory is fast but the disk is slow. * Avoid disk seeks if possible. * Simple compression algorithms are fast. * Compress data before sending it over the internet if possible. * Data centers are usually in different regions, and it takes time to send data between them. |

## Availability numbers

High availability (continuously operate) is measured as a percentage, with 100% means a service that has 0 downtime. Most services fall between 99% and 100%.

A service level agreement (SLA) between the service provider (you) and customer, defines the level of uptime your service will deliver. Uptime is traditionally measured in nines. The more the nines, the better. <https://aws.amazon.com/compute/sla/> | <https://cloud.google.com/compute/sla>

<https://azure.microsoft.com/en-us/support/legal/sla/summary/>

| **Availability %** | **Downtime per day** | **Downtime per week** | **Downtime per month** | **Downtime per year** |
| --- | --- | --- | --- | --- |
| 99% | 14.40 minutes | 1.68 hours | 7.31 hours | 3.65 days |
| 99.99% | 8.64 seconds | 1.01 minutes | 4.38 minutes | 52.60 minutes |
| 99.999% | 864.00 | 6.05 seconds | 26.30 seconds | 5.26 minutes |
| 99.9999% | 86.40 milliseconds | 604.80 | 2.63 seconds | 31.56 seconds |

## Example: Estimate Twitter QPS and storage requirements

Please note the following numbers are for this exercise only as they are not real numbers from Twitter.

Assumptions:

* 300 million monthly active users.
* 50% of users use Twitter daily.
* Users post 2 tweets per day on average.
* 10% of tweets contain media.
* Data is stored for 5 years.

Estimations: Query per second (QPS) estimate:

* Daily active users (DAU) = 300 million \* 50% = 150 million
* Tweets QPS = 150 million \* 2 tweets / 24 hour / 3600 seconds = ~3500
* Peek QPS = 2 \* QPS = ~7000

We will only estimate media storage here.

* Average tweet size:
* tweet\_id 64 bytes
* text 140 bytes
* media 1 MB
* Media storage: 150 million \* 2 \* 10% \* 1 MB = 30 TB per day
* 5-year media storage: 30 TB \* 365 \* 5 = ~55 PB

## Tips

* Rounding and Approximation. It is difficult to perform complicated math operations during the interview. For example, Instead of complex “99987 / 9.1” to round and simply by “100,00/10”. Precision is not expected. Use round numbers and approximation to your advantage.
* Write down your assumptions. It is a good idea to write down your assumptions to be referenced later.
* Label your units. Label “5”, does it mean 5 KB or 5 MB? So Label “5 MB” helps to remove ambiguity.
* Commonly asked : QPS, peak QPS, storage, cache, number of servers, etc. You can practice these calculations when preparing for an interview.

<https://github.com/donnemartin/system-design-primer>

# O4 A Framework For System Design Interviews

## A 4-step process for effective system design interview

Every system design interview is different. A great system design interview is open-ended and there is no one-size-fits-all solution

### Step 1 - Understand the problem and establish design scope

Answering/Solution without a thorough understanding of the requirements or thinking is a huge red flag as the interview is not a trivia contest. Think deeply and ask questions to clarify requirements and assumptions. This is extremely important.

As an engineer, we like to solve hard problems and jump into the final design; however, this approach is likely to lead you to design the wrong system. Ask the right questions, make the proper assumptions, and gather all the information needed to build a system.

When you ask a question, the interviewer either answers your question directly or asks you to make your assumptions. If the latter happens, write down your assumptions on the whiteboard or paper. You might need them later.

What kind of questions to ask? Ask questions to understand the exact requirements.

* What specific features are we going to build? How many users does the product have?
* How fast does the company anticipate to scale up? What are the anticipated scales in 3 months, 6 months, and a year?
* What is the company’s technology stack? What existing services you might leverage to simplify the design?

#### Example ; Asked to design a news feed system, you want to ask questions that help you clarify the requirements. Some sample questions

**Candidate**: Is this a mobile app? Or a web app? Or both? **Interviewer**: Both.

**Candidate**: What are the most important features for the product? >> Ability to make a post and see friends’ news feed.

**Candidate**: Is the news feed sorted in reverse chronological order or a particular order? The particular order means each post is given a different weight. For instance, posts from your close friends are more important than posts from a group.  
**Interviewer**: To keep things simple, let us assume the feed is sorted by reverse chronological order.

**Candidate**: How many friends can a user have? >> 5000

**Candidate**: What is the traffic volume? >> 10 million daily active users (DAU)

**Candidate**: Can feed contain images, videos, or just text? >> It can contain media files, including both images and videos.

### Step 2 - Propose high-level design and get buy-in

In this step, we aim to develop a high-level design and reach an agreement with the interviewer on the design. It is a great idea to collaborate with the interviewer during the process.

* Come up with an initial blueprint for the design. Ask for feedback. Treat your interviewer as a teammate and work together. Many good interviewers love to talk and get involved.
* Draw box diagrams with key components on the whiteboard - like clients (mobile/web), APIs, web servers, data stores, cache, CDN, message queue, etc.
* Do back-of-the-envelope calculations to evaluate. Communicate with your interviewer if it is necessary before diving into it.

If possible, go through a few concrete use cases. This will help you frame the high-level design.

#### Example

Use “Design a news feed system” to demonstrate how to approach the high-level design. Here you are not required to understand how the system actually works.

At the high level, the design is divided into two flows: feed publishing and news feed building.

* Feed publishing: when a user publishes a post, corresponding data is written into cache/database, and the post will be populated into friends’ news feed.
* Newsfeed building: the news feed is built by aggregating friends’ posts in a reverse chronological order.

Figure present high-level designs for feed publishing and news feed building flows, respectively.

|  |  |
| --- | --- |
| Figure 1 | Figure 2 |

### Step 3 - Design deep dive

At this step, you and your interviewer should have already achieved the following objectives:

* Agreed on the overall goals and feature scope
* Sketched out a high-level blueprint for the overall design
* Obtained feedback from your interviewer on the high-level design
* Had some initial ideas about areas to focus on in deep dive based on her feedback

You shall work with the interviewer to identify and prioritize components in the architecture. Sometimes, the interviewer may give off hints that she likes focusing on high-level design. Sometimes, for a senior candidate interview, the discussion could be on the system performance characteristics, likely focusing on the bottlenecks and resource estimations. In most cases, the interviewer may want you to dig into details of some system components. For URL shortener, it is interesting to dive into the hash function design that converts a long URL to a short one. For a chat system, how to reduce latency and how to support online/offline status are two interesting topics.

Time management is essential as it is easy to get carried away with minute details that do not demonstrate your abilities. Try not to get into unnecessary details. For example, talking about the EdgeRank algorithm of Facebook feed ranking in detail is not ideal during a system design interview as this takes much precious time and does not prove your ability in designing a scalable system.

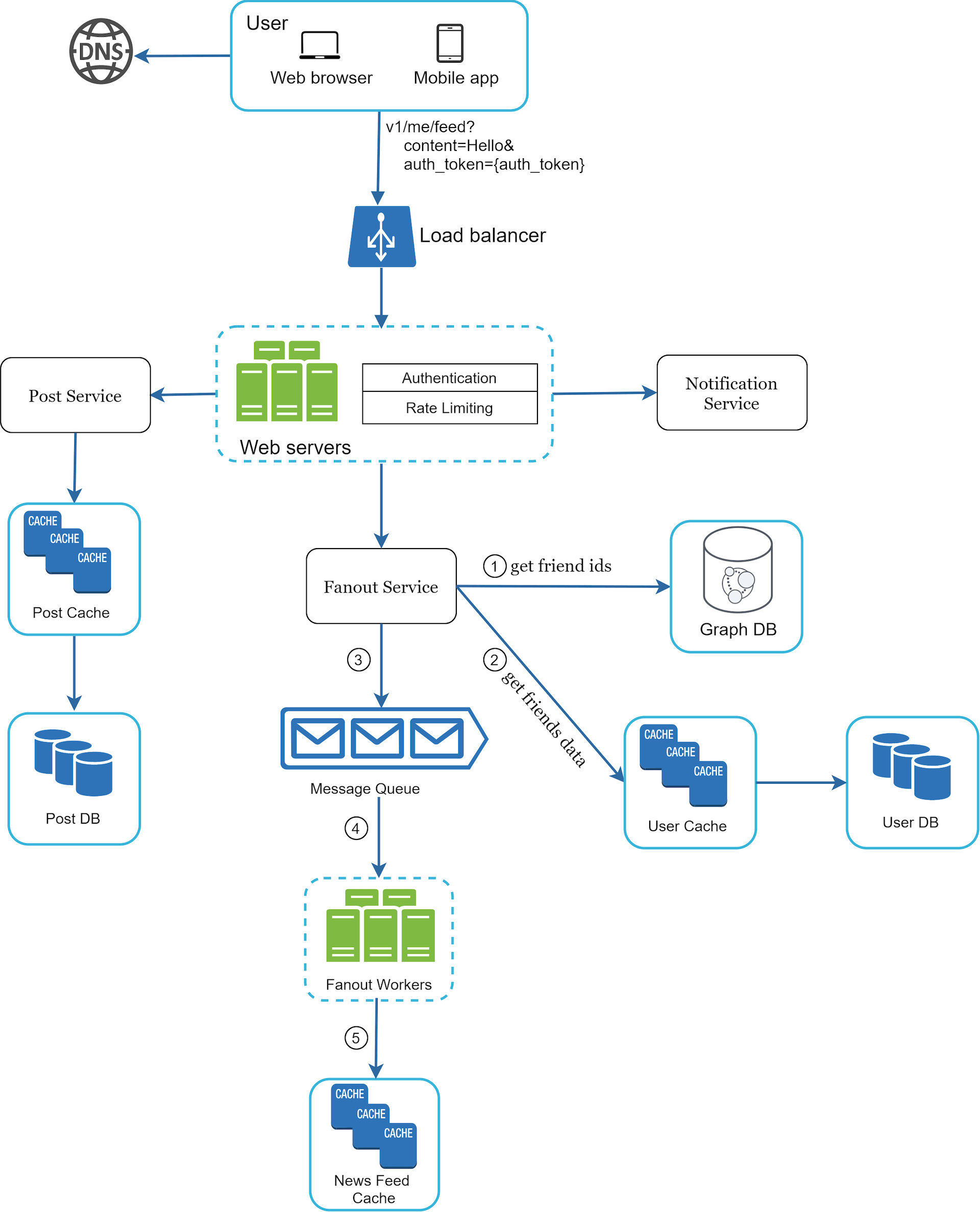
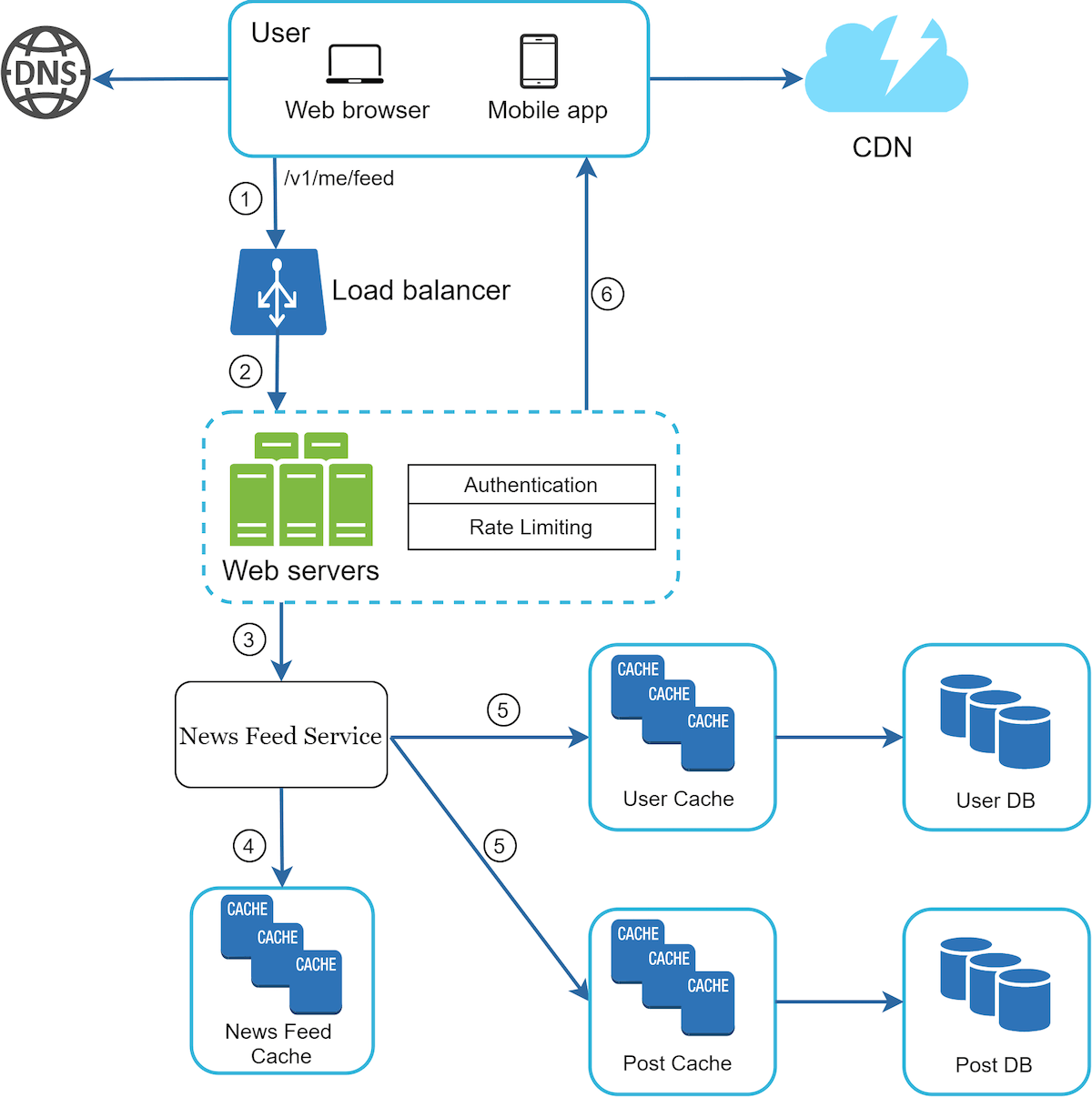
#### Example

At this point, we have discussed the high-level design for a news feed system, and the interviewer is happy with your proposal. Next, we will investigate two of the most important use cases:

1. Feed publishing

2. News feed retrieval

Figure 3 and Figure 4 show the detailed design for the two use cases, which will be explained in detail in the "Design A News Feed System" chapter.

### Step 4 - Wrap up

In this final step, the interviewer might ask you a few follow-up questions or give you the freedom to discuss other additional points. Here are a few directions to follow:

* The interviewer might want you to identify the system bottlenecks and discuss potential improvements. Never say your design is perfect and nothing can be improved. There is always something to improve upon. This is a great opportunity to show your critical thinking and leave a good final impression.
* It could be useful to give the interviewer a recap of your design. This is particularly important if you suggested a few solutions. Refreshing your interviewer’s memory can be helpful after a long session.
* Error cases (server failure, network loss, etc.) are interesting to talk about.
* Operation issues are worth mentioning. How do you monitor metrics and error logs? How to roll out the system?
* How to handle the next scale curve is also an interesting topic. For example, if your current design supports 1 million users, what changes do you need to make to support 10 million users?
* Propose other refinements you need if you had more time.

To wrap up, we summarize a list of the Dos and Don’ts.

Dos

* Always ask for clarification. Do not assume your assumption is correct.
* Understand the requirements of the problem.
* There is neither the right answer nor the best answer. A solution designed to solve the problems of a young startup is different from that of an established company with millions of users. Make sure you understand the requirements.
* Let the interviewer know what you are thinking. Communicate with your interview.
* Suggest multiple approaches if possible.
* Once you agree with your interviewer on the blueprint, go into details on each component. Design the most critical components first.
* Bounce ideas off the interviewer. A good interviewer works with you as a teammate.
* Never give up.

Don’ts

* Don't be unprepared for typical interview questions.
* Don’t jump into a solution without clarifying the requirements and assumptions.
* Don’t go into too much detail on a single component in the beginning. Give the high-level design first then drills down.
* If you get stuck, don't hesitate to ask for hints.
* Again, communicate. Don't think in silence.
* Don’t think your interview is done once you give the design. You are not done until your interviewer says you are done. Ask for feedback early and often.

### Time allocation on each step

The following is a very rough guide on distributing your time in a 45-minute interview session.

Step 1 Understand the problem and establish design scope: 3 - 10 minutes

Step 2 Propose high-level design and get buy-in: 10 - 15 minutes

Step 3 Design deep dive: 10 - 25 minutes

Step 4 Wrap: 3 - 5 minutes

# 13 Design A Chat System

A chat app performs different functions for different people. Nail down the exact requirements (1-on-1 or group chat). It is important to explore the feature requirements.

## Step 1 - Understand the problem and establish design scope

Type of chat app to design. 1-on-1 chat apps (Facebook Messenger, WeChat, and WhatsApp), office chat apps that focus on group chat ( Slack), or game chat apps (Discord), that focus on large group interaction and low voice chat latency.

Nail down what the interviewer has in mind exactly when she asks you to design a chat system. Some questions you might ask are as follows:

**Candidate**: What kind of chat app shall we design? 1 on 1 or group based? -> **Interviewer**: It should support both 1 on 1 and group chat.

**Candidate**: Is this a mobile app? Or a web app? Or both? -> Both.

**Candidate**: What is the scale of this app? A startup app or massive scale? -> It should support 50 million daily active users (DAU).

**Candidate**: For group chat, what is the group member limit? -> A maximum of 100 people

What features are important for the chat app? Can it support attachment? -> 1 on 1 chat, group chat, online indicator. The system only supports txt msgs.

**Candidate**: Is there a message size limit? -> Yes, text length should be less than 100,000 characters long.

**Candidate**: Is end-to-end encryption required? -> Not required for now but we will discuss that if time allows.

**Candidate**: How long shall we store the chat history? -> Forever.

In the chapter, we focus on designing a chat app like Facebook messenger, with an emphasis on the following features:

* A one-on-one chat with low delivery latency
* Small group chat (max of 100 people)
* Online presence
* Multiple device support. The same account can be logged in to multiple accounts at the same time.
* Push notifications

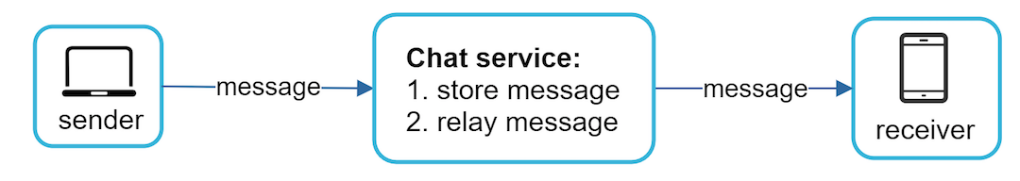
It is also important to agree on the design scale. We will design a system that supports 50 million DAU.

## Step 2 - Propose high-level design and get buy-in

In a chat system, clients can be either mobile applications or web applications. Clients do not communicate directly with each other. Instead, each client connects to a chat service, which supports all the features mentioned above. The chat service must support the following functions (fundamental operations):

* Receive messages from other clients.
* Find the right recipients for each message and relay the message to the recipients.
* If a recipient is not online, hold the messages for that recipient on the server until she is online.

Figure shows the relationships between clients (sender and receiver) and the chat service.



When a client intends to start a chat, it connects the chats service using one or more network protocols. Discuss the choice of protocols with the interviewer.

In Figure, when the sender sends a message to the receiver via the chat service, it uses the time-tested HTTP protocol, which is the most common web protocol. In this scenario, the client opens a HTTP connection with the chat service and sends the message, informing the service to send the message to the receiver. The keep-alive is efficient for this because the keep-alive header allows a client to maintain a persistent connection with the chat service. It also reduces the number of TCP handshakes. HTTP is a fine option on the sender side, and many popular chat applications such as Facebook used HTTP initially to send messages.

However, the receiver side is a bit more complicated. Since HTTP is client-initiated, it is not trivial to send messages from the server. Over the years, many techniques are used to simulate a server-initiated connection: polling, long polling, and WebSocket.

### Polling - is a technique that the client periodically asks the server if there are messages available. Depending on polling frequency, polling could be costly. It could consume precious server resources to answer a question that offers no as an answer most of the time.

A picture containing diagram

Description automatically generated

### Long polling

|  |  |
| --- | --- |
|  | Because polling could be inefficient, the next progression is long polling (Figure 4).  In long polling, a client holds the connection open until there are actually new messages available or a timeout threshold has been reached. Once the client receives new messages, it immediately sends another request to the server, restarting the process. Long polling has a few drawbacks:   * Sender and receiver may not connect to the same chat server. HTTP based servers are usually stateless. If you use round robin for load balancing, the server that receives the message might not have a long-polling connection with the client who receives the message. * A server has no good way to tell if a client is disconnected. * It is inefficient. If a user does not chat much, long polling still makes periodic connections after timeouts. |

### WebSocket

|  |  |
| --- | --- |
| Diagram  Description automatically generatedFigure 6 | WebSocket is the most common solution for sending asynchronous updates from server to client. Figure 5 shows how it works. WebSocket connection is initiated by the client. It is bi-directional and persistent. It starts its life as a HTTP connection and could be “upgraded” via some well-defined handshake to a WebSocket connection. Through this persistent connection, a server could send updates to a client. WebSocket connections generally work even if a firewall is in place. This is because they use port 80 or 443 which are also used by HTTP/HTTPS connections.  Earlier we said that on the sender side HTTP is a fine protocol to use, but since WebSocket is bidirectional, there is no strong technical reason not to use it also for sending. Figure 6 shows how WebSockets (ws) is used for both sender and receiver sides.  By using WebSocket for both sending and receiving, it simplifies the design and makes implementation on both client and server more straightforward. Since WebSocket connections are persistent, efficient connection management is critical on the server-side. |

### High-level design

it is important to note that everything else does not have to be WebSocket. In fact, most features (sign up, login, user profile, etc) of a chat application could use the traditional request/response method over HTTP.

As shown in Figure 7, the chat system is broken down into three major categories: stateless services, stateful services, and third-party integration.

|  |  |
| --- | --- |
| Figure 7 | Stateless Services - are traditional public-facing request/response services, used to manage the login, signup, user profile, etc. These are common features among many websites and apps. It’s sit behind a load balancer whose job is to route requests to the correct services based on the request paths. These services can be monolithic or individual microservices. Use service from market instead of build.  Service discovery - is to give the client a list of DNS host names of chat servers that the client could connect to. Stateful Service - The only stateful service is the chat service. The service is stateful because each client maintains a persistent network connection to a chat server. In this service, a client normally does not switch to another chat server as long as the server is still available. The service discovery coordinates closely with the chat service to avoid server overloading.Third-party integration - For a chat app, push notification is the most important third-party integration. It is a way to inform users when new messages have arrived, even when the app is not running. Proper integration of push notification is crucial. |
| Figure 8 | Scalability – On a small scale, all services listed above could fit in one server. Even at the scale we design for, it is in theory possible to fit all user connections in one modern cloud server. The number of concurrent connections that a server can handle will most likely be the limiting factor. In our scenario, at 1M concurrent users, assuming each user connection needs 10K of memory on the server (rough figure), it only needs about 10GB of memory to hold all the connections on one box. Propose a 1 server design, may raise a big red flag in the interviewer’s. However, it is perfectly fine to start with a 1server design. Make sure the interviewer knows this.  In Figure 8, the client maintains a persistent WebSocket connection to a chat server for real-time messaging.   * Chat servers facilitate message sending/receiving. * Presence servers manage online/offline status. * API servers handle everything including user login, signup, change profile, etc. * Notification servers send push notifications. * Finally, the key-value store is used to store chat history. When an offline user comes online, she will see all her previous chat history. |
| Storage - Two types of data exist in a typical chat system.The first is generic data, such as user profile, setting, user friends list. These data are stored in robust and reliable relational databases. Replication and sharding are common techniques to satisfy availability and scalability requirements. The second is unique to chat systems: chat history data. It is important to understand the read/write pattern.   * The amount of data is enormous for chat systems. A previous study reveals that Facebook messenger and Whatsapp process 60 billion messages a day. * Only recent chats are accessed frequently. Users do not usually look up for old chats. * Although very recent chat history is viewed in most cases, users might use features that require random access of data, such as search, view your mentions, jump to specific messages, etc. These cases should be supported by the data access layer. * The read to write ratio is about 1:1 for 1 on 1 chat apps.   Selecting the correct storage system that supports all of our use cases is crucial. We recommend key-value stores for the following reasons:   * K-V stores allow easy horizontal scaling. K-V stores provide very low latency to access data. * Relational databases do not handle long tail of data well. When the indexes grow large, random access is expensive. * Key-value stores are adopted by other proven reliable chat applications. For example, both Facebook messenger and Discord use key-value stores. Facebook messenger uses HBase, and Discord uses Cassandra. | |

### Data models

|  |  |
| --- | --- |
| Graphical user interface, application  Description automatically generated | *Message table for 1 on 1 chat - The primary key is*message\_id*, which helps to decide message sequence. We cannot rely on*created\_at*to decide the message sequence because two messages can be created at the same time.* |
| Figure 10 | Message table for group chat - The composite primary key is (channel\_id, message\_id). Channel and group represent the same meaning here. channel\_id is the partition key because all queries in a group chat operate in a channel. |

#### **Message ID**

Message\_id carries the responsibility of ensuring the order of messages. message\_id must satisfy the following two requirements:

* IDs must be unique.
* IDs should be sortable by time, meaning new rows have higher IDs than old ones.

The first idea that comes to mind is the “auto\_increment” keyword in MySql. However, NoSQL databases usually do not provide such a feature.

The second approach is to use a global 64-bit sequence number generator like Snowflake.

The final approach is to use local sequence number generator. Local means IDs are only unique within a group. The reason why local IDs work is that maintaining message sequence within 1-on-1 channel or a group channel is sufficient. This approach is easier to implement in comparison to the global ID implementation.

## Step 3 - Design deep dive

For the chat system, service discovery, messaging flows, and online/offline indicators worth deeper exploration.

### Service discovery

|  |  |
| --- | --- |
| Figure 11 | The primary role of service discovery is to recommend the best chat server for a client based on the criteria like geographical location, server capacity, etc.  Apache Zookeeper is a popular open-source solution for service discovery. It registers all the available chat servers and picks the best chat server for a client based on predefined criteria.  Figure shows how service discovery (Zookeeper) works.  1. User A tries to log in to the app.  2. The load balancer sends the login request to API servers.  3. After the backend authenticates the user, service discovery finds the best chat server for User A. In this example, server 2 is chosen and the server info is returned back to User A.  4. User A connects to chat server 2 through WebSocket. |

### Message flows

|  |  |
| --- | --- |
|  | It is interesting to understand the end-to-end flow of a chat system1 on 1 chat flow - Figure explains what happens when User A sends a message to User B. 1. User A sends a chat message to Chat server 1.  2. Chat server 1 obtains a message ID from the ID generator.  3. Chat server 1 sends the message to the message sync queue.  4. The message is stored in a key-value store.  5.a. If User B is online, the message is forwarded to Chat server 2 where User B is connected.  5.b. If User B is offline, a push notification is sent from push notification (PN) servers. 6. Chat server 2 forwards the message to User B. There is a persistent WebSocket connection between User B and Chat server 2. |
| *Figure 13* | Message synchronization across multiple devices user A has two devices: a phone and a laptop. When User A logs in to the chat app with her phone, it establishes a WebSocket connection with Chat server 1. Similarly, there is a connection between the laptop and Chat server 1.  Each device maintains a variable called cur\_max\_message\_id, which keeps track of the latest message ID on the device. Messages that satisfy the following two conditions are considered as news messages:   * The recipient ID is equal to the currently logged-in user ID. * Message ID in the key-value store is larger than cur\_max\_message\_id.   With distinct cur\_max\_message\_id on each device, message synchronization is easy as each device can get new messages from the KV store. |
| Figure 14Figure 15 | Small group chat flow Figure 14 explains what happens when User A sends a message in a group chat. Assume there are 3 members in the group (User A, User B and user C). First, the message from User A is copied to each group member’s message sync queue: one for User B and the second for User C. You can think of the message sync queue as an inbox for a recipient. This design choice is good for small group chat because:   * it simplifies message sync flow as each client only needs to check its own inbox to get new messages. * when the group number is small, storing a copy in each recipient’s inbox is not too expensive.   WeChat uses a similar approach, and it limits a group to 500 members. However, for groups with a lot of users, storing a message copy for each member is not acceptable.  On the recipient side, a recipient can receive messages from multiple users. Each recipient has an inbox (message sync queue) which contains messages from different senders. Figure 15 illustrates the design. |

### Online presence

Usually, you can see a green dot next to a user’s profile picture or username. This section explains what happens behind the scenes.

In the high-level design, presence servers are responsible for managing online status and communicating with clients through WebSocket. There are a few flows that will trigger online status change. Let us examine each of them.

|  |  |
| --- | --- |
| Figure 16 | User login - flow is explained in the “Service Discovery” section. After a WebSocket connection is built between the client and the real-time service, user A’s online status and *last\_active\_at* timestamp are saved in the KV store. Presence indicator shows the user is online after she logs in. |
| Diagram  Description automatically generated | User logout - When a user logs out, it goes through the user logout flow. The online status is changed to offline in the KV store. The presence indicator shows a user is offline. |
|  | User disconnection - When a user disconnects from the internet, the persistent connection between the client and server is lost. A naive way to handle user disconnection is to mark the user as offline and change the status to online when the connection re-establishes. However, this approach has a major flaw. It is common for users to disconnect and reconnect to the internet frequently in a short time. For example, network connections can be on and off while a user goes through a tunnel. Updating online status on every disconnect/reconnect would make the presence indicator change too often, resulting in poor user experience. We introduce a heartbeat mechanism to solve this problem. Periodically, an online client sends a heartbeat event to presence servers. If presence servers receive a heartbeat event within a certain time, say x seconds from the client, a user is considered as online. Otherwise, it is offline.  the client sends a heartbeat event to the server every 5 seconds. After sending 3 heartbeat events, the client is disconnected and does not reconnect within x = 30 seconds. The online status is changed to offline. |
| Figure 19 | Online status fanout - How do user A’s friends know about the status changes? Presence servers use a publish-subscribe model, in which each friend pair maintains a channel. When User A’s online status changes, it publishes the event to three channels, channel A-B, A-C, and A-D. Those three channels are subscribed by User B, C, and D, respectively. Thus, it is easy for friends to get online status updates. The communication between clients and servers is through real-time WebSocket. The above design is effective for a small user group. For instance, WeChat uses a similar approach because its user group is capped to 500. For larger groups, informing all members about online status is expensive and time consuming. Assume a group has 100,000 members. Each status change will generate 100,000 events. To solve the performance bottleneck, a possible solution is to fetch online status only when a user enters a group or manually refreshes the friend list. |

## Step 4 - Wrap up

WebSocket is used for real-time communication between the client and server. The chat system contains the following components: chat servers for real-time messaging, presence servers for managing online presence, push notification servers for sending push notifications, key-value stores for chat history persistence and API servers for other functionalities.

If you have extra time at the end of the interview, here are additional talking points:

* Extend the chat app to support media files such as photos and videos. Media files are significantly larger than text in size. Compression, cloud storage, and thumbnails are interesting topics to talk about.
* End-to-end encryption. Whatsapp supports end-to-end encryption for messages. Only the sender and the recipient can read messages. Interested readers should refer to the article in the reference materials [9].
* Caching messages on the client-side is effective to reduce the data transfer between the client and server.
* Improve load time. Slack built a geographically distributed network to cache users’ data, channels, etc. for better load time [10].
* Error handling.
* The chat server error. There might be hundreds of thousands, or even more persistent connections to a chat server. If a chat server goes offline, service discovery (Zookeeper) will provide a new chat server for clients to establish new connections with.
* Message resent mechanism. Retry and queueing are common techniques for resending messages.

<https://www.erlang-factory.com/upload/presentations/31/EugeneLetuchy-ErlangatFacebook.pdf>

# Designing a Rate Limiter

**What is Rate Limiting?**

|  |  |
| --- | --- |
| Diagram  Description automatically generated  Figure 1: Rate limiter rejects 3rd request | Rate limiting is used for security purposes as well as performance improvement. For example, a service can serve a limited number of requests per second. However, if a service is receiving a huge number of requests at a time, too much for the server to handle & the server might be shut down. To handle this type of problem, we would need some kind of limiting mechanism that would allow only a certain number of requests to the service.  Rate limiting is limiting some operations with some threshold. If those threshold values are crossed, the system will return errors.  Rate limiting can be implemented in tiers. E.g., a network request can be limited to 1 request per second, 4 requests/min, 10 requests/5 minutes. |

**Why do Need Rate Limiting?**

Rate limiting is needed to protect a system from being brought down by hackers or the DOS attack.

The denial of service (DOS) attack, is when hackers try to flood a system with many requests within a short period of time to shut the system down. Because a server has a limit of how many requests it can serve within a certain period of time. So, the system can’t handle the floods of requests properly.

We can rate-limit the system in various ways, like

Based on user id - limit a user’s request operation to a server for 5 operations/minute.

Based on IP address, region, etc.

Based on total more than 20K requests/minute. If the total user request exceeds that number in a minute, the system is gonna return errors.

**Requirements and Goals of the System -** The requirements are divided into two parts:

Functional Requirements:

* A client can send a limited number of requests to a server within a time window, e.g., 10 requests per second.
* The client should get an error msg if the defined threshold limit of request is crossed for a single server or across different combinations of servers.

Non-Functional Requirements:

* The system should be highly available since it protects our service from external attacks.
* Performance is an important factor for any system. Careful that rate limiter service should not add substantial latencies to the system.

**Memory Estimation:**

Let’s assume we are keeping all of the data in a hash-table. The key would be the hash value of user ID, and the value would be a structure of count and startTime, e.g., UserId {Count, StartTime} UserId {Count, StartTime}

Now, let’s assume ‘UserID’ takes 8 bytes and 2 bytes for ‘Count,’ which can count up to 65k, which is sufficient for our use case. Although the end time will need 4 bytes, we can store only the minute and second parts. The hour part is not necessary if we are not considering a window of an hour. Now, it will take 2 bytes. So, we need a total of 12 bytes to store a user’s data.

Now, if our hash-table has an overhead of 20 bytes for each record and we need to track 1 million users at any time, the total memory we need would be : (12 + 20) bytes \* 1 million => 32MB

If we need a 4-byte number to lock each user’s record to resolve our atomicity problems, we would require a total of 36MB memory.

**Domain Model/High-level Design:**

|  |  |
| --- | --- |
| Diagram  Description automatically generated | When a new request arrives from the client, the Server first asks the Rate Limiter to decide if it will be served or rejected. If the request is not rejected, then it’ll be passed to the internal API servers.  Rate Limiter’s responsibility is to decide which client request will be served and which request will be declined. |

**Fixed Window algorithm:**

|  |  |
| --- | --- |
| Diagram  Description automatically generated | In this algorithm, we may consider a time window from the start to end time unit. For example, our window period can be considered 0–60 seconds, a minute, irrespective of the time frame at which the client requests the server. We already have thought of a structure of count and startTime, e.g., UserId {Count, StartTime}  Figure I displays the scenario where a client can make two requests per minute. So, request 1 and request 2 are allowed by the rate limiter. But the third request is blocked as it can within 1 minute from the same user. This type of algorithm is named as Fixed Window algorithm. In this algorithm, a period is considered 1 minute( for this example) irrespective of the time frame at which the two API requests have been allowed.  After 1 minute of the first request from a specific client, the start time is reset for that user. This is a simple approach for rate limiters. But there are some drawbacks to this approach. |

**Drawback:**

It can allow twice the number of allowed requests per minute. Imagine the user sends two requests at the last second of a minute(12:01:59) and immediately makes two more requests at the first second(12:02:00) of the next minute. It may seem a minor problem for only two requests. But if the number of requests is much higher(for example, 50 requests) or the time frame is much lower, e.g., 15 seconds, it can cause a huge number of requests for a server. Especially we are considering here only one user. Imagine what will happen to a service that serves millions of users.

Besides, in a distributed system, the “read-and-then-write” behavior can create a race condition. For example, imagine if the client’s current request count is 2 and makes two more requests. If two separate processes serve each of these requests and concurrently read the Count value before either of them updated it, the process would result in the client not hitting the rate limit. Thus, we may need to implement locking each record.

**Sliding Window Algorithm:**

If we keep track of each request per user in a time frame, we may store the timestamp of each request in a Sorted Set in our ‘value’ field of hash-table. But, it would take a lot of memory. So, we can use a sliding window with the counters. Now, consider this if we keep track of request counts for each user using multiple fixed time windows.

For example, if we have an hourly window, when we receive a new request to calculate the limit, we can count requests each minute and calculate the sum of all counters in the past hour. This would reduce the memory we need for the set.

Suppose we rate-limit at 500 requests per hour with an additional limit of 10 requests per minute. This means that the client has exceeded the rate limit when the sum of the counters in the past hour exceeds the limit request(500).

For a smaller time frame, the client can’t send more than ten requests per minute. This would be a hybrid and reasonable consideration as we may think that none of the real users would send such frequent requests. Even if they do, they will get success with retries since their limits get reset every minute.

**Problems in Distributed Environment:**

|  |  |
| --- | --- |
| Diagram  Description automatically generated | A user is allowed to request 3 times within a minute. The scenario is that the user already made 2 requests and made two new requests within 2 seconds. And the requests from the user went to two different load balancers, then to two different rate limiter services. As the DB already contains the count value 2, both the Rate Limiter service gets the value below the threshold and permits the requests. So, we are now allowing 4 requests from the same user within a minute, which breaks the limit of the rate limiter. This is the inconsistency problem.  As a solution, we may use sticky session load balancing to ensure that one user’s request will always go to the same rate limiter service. But the problem here is that it’s not a fault-tolerant design. Because if the Rate limiter service is down, user requests can not be served by the system.  We can solve this problem using locks. Once a service uses counter data from the database, it will lock it. So, other services can not use it before the counter data is updated. But this procedure also has its own share of problems. It will add an extra latency for the locking. So, we need to decide between availability and performance trade-offs. |

**Caching:**

|  |  |
| --- | --- |
| Diagram  Description automatically generated | This system may get huge benefits from caching recent active users. Servers can quickly check if the cache contains the counter value for the user before hitting backend servers. This should be faster than going to the DB each time.  We may use the Write-back cache strategy by updating all counters and timestamps in cache only. Then, we can write to the database done at fixed intervals, e.g., 1 hour. This can ensure minimum latency added to the user’s requests, which should improve the Lock latency problem.  When the server reads data, it can always hit the cache first. This will be useful when the user has hit their maximum limit. The rate limiter reads data without any updates.  We may use the Least Recently Used (LRU) as a cache eviction policy for our system. |

**DoS attack:**

To prevent the system from a DoS attack, we can take other strategies to rate limit by IP address or users.

IP: limit requests per IP; It might not be a good way to differentiate between normal users and attackers. But, it’s still better to have some protection rather than not have anything at all.

The biggest problem with IP-based rate limiting is when multiple users share a single public IP. For example, in an internet cafe or smartphone, users are using the same gateway. Another problem could be, there are lots of IPv6 addresses available to a hacker from even one computer.

User: After user authentication, the system will provide the user a token which the user will pass with each request. This will ensure that we can rate limit an API that has a valid authentication token. But the problem is that we have to rate-limit the login API itself.

The weakness of this strategy would be that a hacker can perform a denial of service attack against a user by providing wrong credentials up to the limit; after that, the actual user may not be able to log in.

So, we can use both the approach together. However, this may result in more cache entries with more details per entry. So, we would need more memory and storage.

<https://ashchk.medium.com/>

# 15 Design YouTube

YouTube looks simple: content creators upload videos and viewers click play. Is it really that simple? Not really. There are lots of complex technologies underneath the simplicity. Let us look at some impressive statistics, demographics, and fun facts of YouTube in 2020.

* Total number of monthly active users: 2 billion. Number of videos watched per day: 5 billion.
* 73% of US adults use YouTube. 50 million creators on YouTube
* YouTube’s Ad revenue was $15.1 billion for the full year 2019, up 36% from 2018.
* YouTube is responsible for 37% of all mobile internet traffic. YouTube is available in 80 different languages.

## Step 1 - Understand the problem and establish design scope

Besides watching a video, you can do a lot more on YouTube. For example, comment, share, or like a video, save a video to playlists, subscribe to a channel, etc. It is impossible to design everything within a 45- or 60-minute interview. Thus, it is important to ask questions to narrow down the scope.

**Candidate**: What features are important? **Interviewer**: Ability to upload a video and watch a video.

**Candidate**: What clients do we need to support? >> Mobile apps, web browsers, and smart TV.

**Candidate**: How many daily active users do we have? >> 5 million

**Candidate**: What is the average daily time spent on the product? >> 30 minutes.

**Candidate**: Do we need to support international users? >> Yes, a large percentage of users are international users.

**Candidate**: What are the supported video resolutions? >> The system accepts most of the video resolutions and formats.

**Candidate**: Is encryption required? >> Yes

Any file size requirement for videos? >> Our platform focuses on small and medium-sized videos. The maximum allowed video size is 1GB.

Can we leverage some of the existing cloud infrastructures provided by Amazon, Google, or Microsoft? >> That is a great question. Building everything from scratch is unrealistic for most companies, it is recommended to leverage some of the existing cloud services.

In the chapter, we focus on designing a video streaming service with the following features:

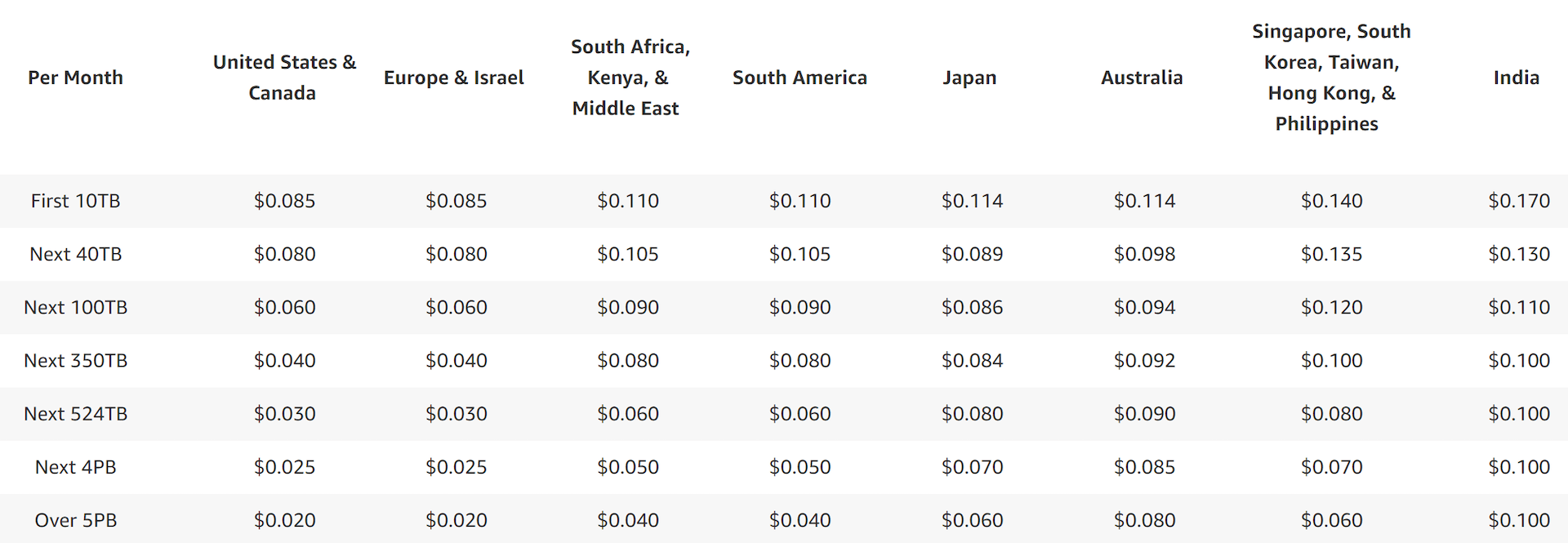
* Ability to upload videos fast, Smooth video streaming, Ability to change video quality
* Low infrastructure cost, High availability, scalability, and reliability requirements
* Clients supported: mobile apps, web browser, and smart TV

### Back of the envelope estimation

The following estimations are based on many assumptions, check the interviewer to make sure she is on the same page.

* Assume the product has 5 million daily active users (DAU). Users watch 5 videos per day.
* 10% of users upload 1 video per day. Assume the average video size is 300 MB.
* Total daily storage space needed: 5 million \* 10% \* 300 MB = 150TB
* CDN cost. When cloud CDN serves a video, you are charged for data transferred out of the CDN.
* Let us use Amazon’s CDN CloudFront for cost estimation (Figure 2). Assume 100% of traffic is served from the US. The average cost per GB is $0.02. For simplicity, we only calculate the cost of video streaming.
* 5 million \* 5 videos \* 0.3GB \* $0.02 = $150,000 per day.

From the rough cost estimation, we know serving videos from the CDN costs lots of money. We will discuss ways to reduce CDN costs in deep dive.



## Step 2 - Propose high-level design and get buy-in

Some readers might ask why not building everything by ourselves? Reasons are listed below:

* System design interviews are not about building everything from scratch. Within the limited time frame, choosing the right technology to do a job right is more important than explaining how the technology works in detail. For instance, mentioning blob storage for storing source videos is enough for the interview. Talking about the detailed design for blob storage could be an overkill.
* Building scalable blob storage or CDN is extremely complex and costly. Even large companies like Netflix or Facebook do not build everything themselves. Netflix leverages Amazon’s cloud services, and Facebook uses Akamai’s CDN .

|  |  |
| --- | --- |
| Figure 3 | At the high-level, the system comprises three components .  **Client**: You can watch YouTube on your computer, mobile phone, and smartTV.  **CDN**: Videos are stored in CDN. When you press play, a video is streamed from the CDN.  **API servers**: Everything else except video streaming goes through API servers. This includes feed recommendation, generating video upload URL, updating metadata database and cache, user signup, etc.  In the question/answer session, the interviewer showed interests in two flows:   * Video uploading flow * Video streaming flow |

### Video uploading flow

|  |  |
| --- | --- |
| Figure 4 | It consists of the following components:   * User: A user watches YouTube on devices. * Load balancer: A load balancer evenly distributes requests among API servers. * API servers: All user requests go through API servers except video streaming. * Metadata DB: Video metadata are stored in Metadata DB. It is sharded and replicated to meet performance and high availability requirements. * Metadata cache: For better performance, video metadata and user objects are cached. * Original storage: A blob storage system is used to store original videos. . * Transcoding servers: Video transcoding is also called video encoding. It is the process of converting a video format to other formats (MPEG, HLS, etc), which provide the best video streams possible for different devices and bandwidth capabilities. * Transcoded storage: It is a blob storage that stores transcoded video files. * CDN: Videos are cached in CDN. When you click the play button, a video is streamed from the CDN. * Completion queue: It is a message queue that stores information about video transcoding completion events. * Completion handler: This consists of a list of workers that pull event data from the completion queue and update metadata cache and database.   Examine how the video uploading flow works. The flow is broken down into two processes running in parallel.  a. Upload the actual video.  b. Update video metadata. Metadata contains information about video URL, size, resolution, format, user info, etc. |
| Figure 5 | Flow a: upload the actual video 1. Videos are uploaded to the original storage.  2. Transcoding servers fetch videos from the original storage and start transcoding.  3. Once transcoding is complete, the following two steps are executed in parallel:   * 3a. Transcoded videos are sent to transcoded storage. * 3b. Transcoding completion events are queued in the completion queue.   3a.1. Transcoded videos are distributed to CDN.  3b.1. Completion handler contains a bunch of workers that continuously pull event data from the queue.  3b.1.a. and 3b.1.b. Completion handler updates the metadata database and cache when video transcoding is complete.  4. API servers inform the client that the video is successfully uploaded and is ready for streaming. |
| Figure 6 | Flow b: update the metadataWhile a file is being uploaded to the original storage, the client in parallel sends a request to update the video metadata as shown in Figure 6. The request contains video metadata, including file name, size, format, etc. API servers update the metadata cache and database. |

### Video streaming flow

|  |  |
| --- | --- |
| Figure 7Figure shows a high level of design for video streaming. | Whenever you watch a video on YouTube, it usually starts streaming immediately and you do not wait until the whole video is downloaded. Downloading means the whole video is copied to your device, while streaming means your device continuously receives video streams from remote source videos. When you watch streaming videos, your client loads a little bit of data at a time so you can watch videos immediately and continuously.  Before we discuss video streaming flow, let us look at an important concept: streaming protocol. This is a standardized way to control data transfer for video streaming. Popular streaming protocols are:   * MPEG–DASH. MPEG stands for “Moving Picture Experts Group” and DASH stands for "Dynamic Adaptive Streaming over HTTP". * Apple HLS. HLS stands for “HTTP Live Streaming”. * Microsoft Smooth Streaming. * Adobe HTTP Dynamic Streaming (HDS).   You do not need to fully understand or even remember those streaming protocol names as they are low-level details that require specific domain knowledge. The important thing here is to understand that different streaming protocols support different video encodings and playback players. When we design a video streaming service, we have to choose the right streaming protocol to support our use cases.  Videos are streamed from CDN directly. The edge server closest to you will deliver the video. Thus, there is very little latency. |

## Step 3 - Design deep dive

### Video transcoding

When you record a video, the device (usually a phone or camera) gives the video file a certain format. If you want the video to be played smoothly on other devices, the video must be encoded into compatible bitrates and formats. Bitrate is the rate at which bits are processed over time. A higher bitrate generally means higher video quality. High bitrate streams need more processing power and fast internet speed.

Video transcoding is important for the following reasons:

* Raw video consumes large amounts of storage space. An hour-long high definition video recorded at 60 frames per second can take up a few hundred GB of space.
* Many devices and browsers only support certain types of video formats. Thus, it is important to encode a video to different formats for compatibility reasons.
* To ensure users watch high-quality videos while maintaining smooth playback, it is a good idea to deliver higher resolution video to users who have high network bandwidth and lower resolution video to users who have low bandwidth.
* Network conditions can change, especially on mobile devices. To ensure a video is played continuously, switching video quality automatically or manually based on network conditions is essential for smooth user experience.

Many types of encoding formats are available; however, most of them contain two parts:

* Container: This is like a basket that contains the video file, audio, and metadata. You can tell the container format by the file extension, such as .avi, .mov, or .mp4.
* Codecs: These are compression and decompression algorithms aim to reduce the video size while preserving the video quality. The most used video codecs are H.264, VP9, and HEVC.

### Directed acyclic graph (DAG) model

Transcoding a video is computationally expensive and time-consuming. Besides, different content creators may have different video processing requirements. For instance, some content creators require watermarks on top of their videos, some provide thumbnail images themselves, and some upload high definition videos, whereas others do not.

To support different video processing pipelines and maintain high parallelism, it is important to add some level of abstraction and let client programmers define what tasks to execute. For example, Facebook’s streaming video engine uses a directed acyclic graph (DAG) programming model, which defines tasks in stages so they can be executed sequentially or parallelly [8]. In our design, we adopt a similar DAG model to achieve flexibility and parallelism. Figure represents a DAG for video transcoding.

In Figure, the original video is split into video, audio, and metadata. Here are some of the tasks that can be applied on a video file:

* Inspection: Make sure videos have good quality and are not malformed.
* Video encodings: Videos are converted to support different resolutions, codec, bitrates, etc. Below Figure shows an example of video encoded files.
* Thumbnail. Thumbnails can either be uploaded by a user or automatically generated by the system.
* Watermark: An image overlay on top of your video contains identifying information about your video.

|  |  |
| --- | --- |
|  |  |

### Video transcoding architecture

|  |  |
| --- | --- |
| Diagram  Description automatically generated | The proposed video transcoding architecture that leverages the cloud services, is shown in Figure.  The architecture has six main components: preprocessor, DAG scheduler, resource manager, task workers, temporary storage, and encoded video as the output. |
| Figure 12    Figure 13  Figure 13 (source: [9]) | **Preprocessor :** The preprocessor has 4 responsibilities: 1. Video splitting. Video stream is split or further split into smaller Group of Pictures (GOP) alignment. GOP is a group/chunk of frames arranged in a specific order. Each chunk is an independently playable unit, usually a few seconds in length.  2. Some old mobile devices or browsers might not support video splitting. Preprocessor split videos by GOP alignment for old clients.  3. DAG generation. The processor generates DAG based on configuration files client programmers write. Figure 12 is a simplified DAG representation which has 2 nodes and 1 edge:  This DAG representation is generated from the two configuration files below (Figure 13):  4. Cache data. The preprocessor is a cache for segmented videos. For better reliability, the preprocessor stores GOPs and metadata in temporary storage. If video encoding fails, the system could use persisted data for retry operations. |
|  | **DAG scheduler** The DAG scheduler splits a DAG graph into stages of tasks and puts them in the task queue in the resource manager. Figure shows an example of how the DAG scheduler works.  As shown in Figure, the original video is split into three stages: Stage 1: video, audio, and metadata. The video file is further split into two tasks in stage 2: video encoding and thumbnail. The audio file requires audio encoding as part of the stage 2 tasks. |
| Diagram  Description automatically generated **Task workers -** run the tasks which are defined in the DAG. Different task workers may run different tasks( the above Figure).**Temporary storage** Multiple storage systems are used here. The choice of storage system depends on factors like data type, data size, access frequency, data life span, etc. For instance, metadata is frequently accessed by workers, and the data size is usually small. Thus, caching metadata in memory is a good idea. For video or audio data, we put them in blob storage. Data in temporary storage is freed up once the corresponding video processing is complete. Encoded video - is the final output of the encoding pipeline. Here is an example of the output: funny\_720p.mp4. | **Resource manager -** is responsible for managing the efficiency of resource allocation. It contains 3 queues and a task scheduler as shown in Figure.  * Task queue: It is a priority queue that contains tasks to be executed. * Worker queue: It is a priority queue that contains worker utilization info. * Running queue: It contains info about the currently running tasks and workers running the tasks.   •Task scheduler: It picks the optimal task/worker, and instructs the chosen task worker to execute the job.  The resource manager works as follows:   * The task scheduler gets the highest priority task from the task queue. * The task scheduler gets the optimal task worker to run the task from the worker queue. * The task scheduler instructs the chosen task worker to run the task. * The task scheduler binds the task/worker info and puts it in the running queue. * The task scheduler removes the job from the running queue once the job is done. |
|  |  |

### System optimizations - the system with optimizations, including speed, safety, and cost-saving.

#### Speed optimization: parallelize video uploading

|  |  |
| --- | --- |
| Text  Description automatically generated | Uploading a video as a whole unit is inefficient. We can split a video into smaller chunks by GOP alignment. |
| Figure 23 | This allows fast resumable uploads when the previous upload failed. The job of splitting a video file by GOP can be implemented by the client to improve the upload speed |

#### Speed optimization: place upload centers close to users

Another way to improve the upload speed is by setting up multiple upload centers across the globe. People in the US can upload videos to the NA upload center, and people in China can upload videos to the Asian upload center. To achieve this, we use CDN as upload centers.

#### Speed optimization: parallelism everywhere

|  |  |
| --- | --- |
| Diagram  Description automatically generated | Achieving low latency requires serious efforts. Another optimization is to build a loosely coupled system and enable high parallelism.  Our design needs some modifications to achieve high parallelism. Let us zoom in to the flow of how a video is transferred from original storage to the CDN. The flow is shown in Figure, revealing that the output depends on the input of the previous step. This dependency makes parallelism difficult. |
| Diagram  Description automatically generated | To make the system more loosely coupled, we introduced message queues as shown in Figure. Let us use an example to explain how message queues make the system more loosely coupled.   * Before the message queue is introduced, the encoding module must wait for the output of the download module.   After the message queue is introduced, the encoding module does not need to wait for the output of the download module anymore. If there are events in the message queue, the encoding module can execute those jobs in parallel. |

#### Safety optimization: pre-signed upload URL

To ensure only authorized users upload videos to the right location, we introduce pre-signed URLs as shown in Figure 27.

|  |  |
| --- | --- |
| Diagram  Description automatically generated | The upload flow is updated as follows:  1. The client makes a HTTP request to API servers to fetch the pre-signed URL, which gives the access permission to the object identified in the URL. The term pre-signed URL is used by uploading files to Amazon S3. Other cloud service providers might use a different name. For instance, Microsoft Azure blob storage supports the same feature, but call it “Shared Access Signature”.  2. API servers respond with a pre-signed URL.  3. Once the client receives the response, it uploads the video using the pre-signed URL. |

#### Safety optimization: protect your videos

Many content makers are reluctant to post videos online because they fear their original videos will be stolen. To protect copyrighted videos, we can adopt one of the following three safety options:

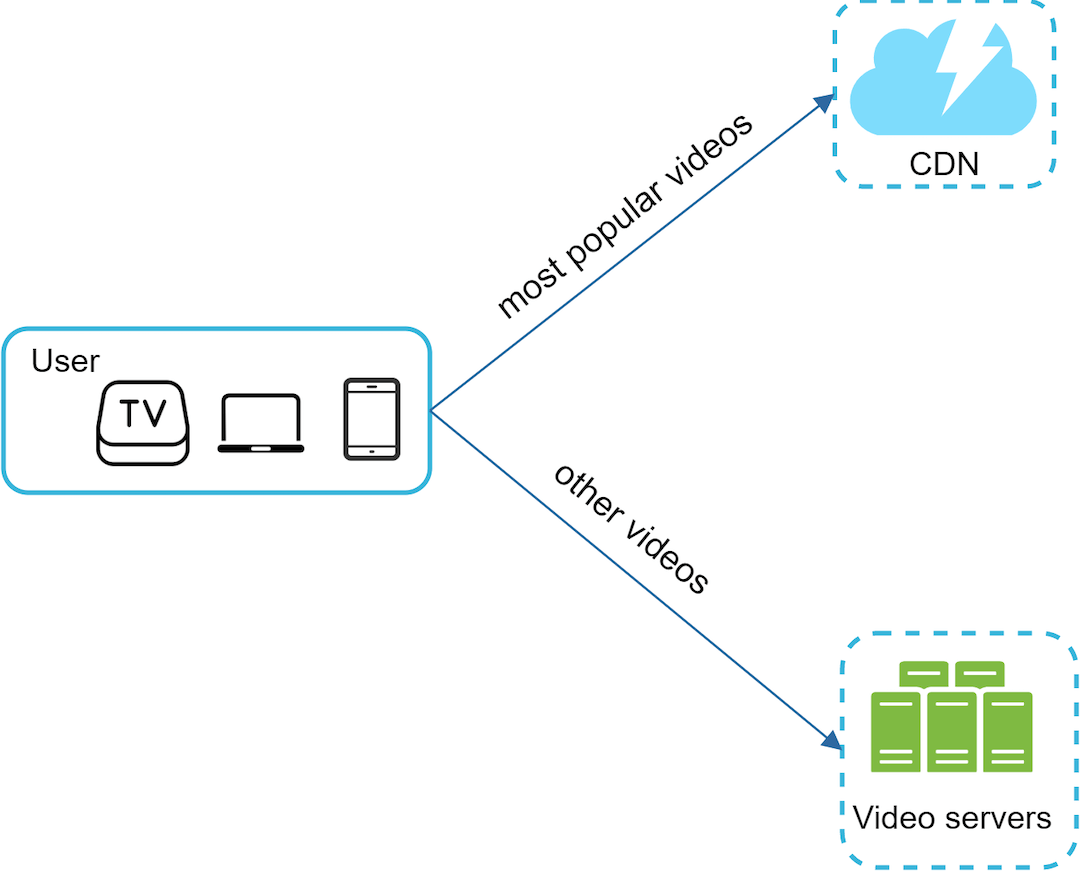
* Digital rights management (DRM) systems: Three major DRM systems are Apple FairPlay, Google Widevine, and Microsoft PlayReady.
* AES encryption: You can encrypt a video and configure an authorization policy. The encrypted video will be decrypted upon playback. This ensures that only authorized users can watch an encrypted video.
* Visual watermarking: This is an image overlay on top of your video that contains identifying information for your video. It can be your company logo or company name.

#### Cost-saving optimization

CDN is a crucial component of our system. It ensures fast video delivery on a global scale. However, from the back of the envelope calculation, we know CDN is expensive, especially when the data size is large. How can we reduce the cost?

Previous research shows that YouTube video streams follow long-tail distribution. It means a few popular videos are accessed frequently but many others have few or no viewers. Based on this observation, we implement a few optimizations:

1. Only serve the most popular videos from CDN and other videos from our high capacity storage video servers.



2. For less popular content, we may not need to store many encoded video versions. Short videos can be encoded on-demand.

3. Some videos are popular only in certain regions. There is no need to distribute these videos to other regions.

4. Build your own CDN like Netflix and partner with Internet Service Providers (ISPs). Building your CDN is a giant project; however, this could make sense for large streaming companies. An ISP can be Comcast, AT&T, Verizon, or other internet providers. ISPs are located all around the world and are close to users. By partnering with ISPs, you can improve the viewing experience and reduce the bandwidth charges.

All those optimizations are based on content popularity, user access pattern, video size, etc. It is important to analyze historical viewing patterns before doing any optimization.

### Error handling

For a large-scale system, system errors are unavoidable. To build a highly fault-tolerant system, we must handle errors gracefully and recover from them fast. Two types of errors exist:

* Recoverable error. For recoverable errors such as video segment fails to transcode, the general idea is to retry the operation a few times. If the task continues to fail and the system believes it is not recoverable, it returns a proper error code to the client.
* Non-recoverable error. For non-recoverable errors such as malformed video format, the system stops the running tasks associated with the video and returns the proper error code to the client.

Typical errors for each system component are covered by the following playbook:

* Upload error: retry a few times.
* Split video error: if older versions of clients cannot split videos by GOP alignment, the entire video is passed to the server. The job of splitting videos is done on the server-side.
* Transcoding error: retry.
* Preprocessor error: regenerate DAG diagram.
* DAG scheduler error: reschedule a task.
* Resource manager queue down: use a replica.
* Task worker down: retry the task on a new worker.
* API server down: API servers are stateless so requests will be directed to a different API server.
* Metadata cache server down: data is replicated multiple times. If one node goes down, you can still access other nodes to fetch data. We can bring up a new cache server to replace the dead one.
* Metadata DB server down:
* Master is down. If the master is down, promote one of the slaves to act as the new master.
* Slave is down. If a slave goes down, you can use another slave for reads and bring up another database server to replace the dead one.

## Step 4 - Wrap up

here are a few additional points:

* Scale the API tier: Because API servers are stateless, it is easy to scale API tier horizontally.
* Scale the database: You can talk about database replication and sharding.
* Live streaming: It refers to the process of how a video is recorded and broadcasted in real time. Although our system is not designed specifically for live streaming, live streaming and non-live streaming have some similarities: both require uploading, encoding, and streaming. The notable differences are:
* Live streaming has a higher latency requirement, so it might need a different streaming protocol.
* Live streaming has a lower requirement for parallelism because small chunks of data are already processed in real-time.
* Live streaming requires different sets of error handling. Any error handling that takes too much time is not acceptable.
* Video takedowns: Videos that violate copyrights, pornography, or other illegal acts shall be removed. Some can be discovered by the system during the upload process, while others might be discovered through user flagging.

## Reference materials

[1] YouTube by the numbers: <https://www.omnicoreagency.com/youtube-statistics/>

[2] 2019 YouTube Demographics: <https://blog.hubspot.com/marketing/youtube-demographics>

[7] Here’s What You Need to Know About Streaming Protocols: <https://www.dacast.com/blog/streaming-protocols/>

[8] SVE: Distributed Video Processing at Facebook Scale: <https://www.cs.princeton.edu/~wlloyd/papers/sve-sosp17.pdf>

[13] Content Popularity for Open Connect: <https://netflixtechblog.com/content-popularity-for-open-connect-b86d56f613b>

<https://medium.com/@nidhiupreti99/understanding-system-design-of-netflix-backend-architecture-and-cloud-services-b077162e45bc>