TUTFLIX
Visit- TutFlix.ORG For More Premium Resources

00
Foreword

We are delighted that you have decided to join us in learning the system design interviews. System design interview questions are the most difficult to tackle among all the technical interviews. The questions require the interviewees to design an architecture for a software system, which could be a news feed, Google search, chat system, etc. These questions are intimidating, and there is no certain pattern to follow. The questions are usually very big scoped and vague. The processes are open-ended and unclear without a standard or correct answer.

Companies widely adopt system design interviews because the communication and problem-solving skills tested in these interviews are similar to those required by a software engineer's daily work. An interviewee is evaluated based on how she analyzes a vague problem and how she solves the problem step by step. The abilities tested also involve how she explains the idea, discusses with others, and evaluates and optimizes the system. In English, using "she" flows better than "he or she" or jumping between the two. To make reading easier, we use the feminine pronoun throughout this course. No disrespect is intended for male engineers.

The system design questions are open-ended. Just like in the real world, there are many differences and variations in the system. The desired outcome is to come up with an architecture to achieve system design goals. The discussions could go in different ways depending on the interviewer. Some interviewers may choose high-level architecture to cover all aspects; whereas some might choose one or more areas to focus on. Typically, system requirements, constraints and bottlenecks should be well understood to shape the direction of both the interviewer and interviewee.

The objective of this course is to provide a reliable strategy to approach the system design questions. The right strategy and knowledge are vital to the success of an interview.

This course provides solid knowledge in building a scalable system. The more knowledge gained from reading this course, the better you are equipped in solving the system design questions.

This course also provides a step by step framework on how to tackle a system design question. It provides many examples to illustrate the systematic approach with detailed steps that you can follow. With constant practice, you will be well-equipped to tackle system design interview questions.

01

Join the Community

We created a members-only Discord group. It is designed for community discussions on the following topics:

- System design fundamentals.
- Showcasing design diagrams and getting feedback.
- Finding mock interview buddies.
- General chat with community members.

Come join us and introduce yourself to the community, today! Use the link below.

Invite link: http://bit.ly/systemdiscord

03
Back-of-the-envelope Estimation

In a system design interview, sometimes you are asked to estimate system capacity or performance requirements using a back-of-the-envelope estimation. According to Jeff Dean, Google Senior Fellow, "back-of-the-envelope calculations are estimates you create using a combination of thought experiments and common performance numbers to get a good feel for which designs will meet your requirements" [1].

You need to have a good sense of scalability basics to effectively carry out back-of-the-envelope estimation. The following concepts should be well understood: power of two [2], latency numbers every programmer should know, and availability numbers.

Power of two

Although data volume can become enormous when dealing with distributed systems, calculation all boils down to the basics. To obtain correct calculations, it is critical to know the data volume unit using the power of 2. A byte is a sequence of 8 bits. An ASCII character uses one byte of memory (8 bits). Below is a table explaining the data volume unit (Table 1).

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

Table 1

Latency numbers every programmer should know

Dr. Dean from Google reveals the length of typical computer operations in 2010 [1]. 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

Table 2

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 a tool to visualize Dr. Dean's numbers. The tool also takes the time factor into consideration. Figures 2-1 shows the visualized latency numbers as of 2020 (source of figures: reference material [3]).

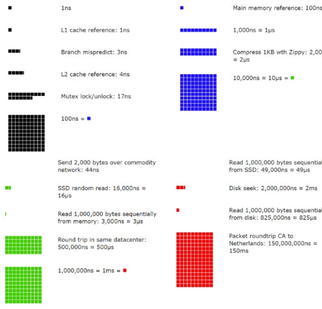

Figure 1

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 is the ability of a system to be continuously operational for a desirably long period of time. High availability 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) is a commonly used term for service providers. This is an agreement between you (the service provider) and your customer, and this agreement formally defines the level of uptime your service will deliver. Cloud providers Amazon [4], Google [5] and Microsoft [6] set their SLAs at 99.9% or above. Uptime is traditionally measured in nines. The more the nines, the better. As shown in Table 3, the number of nines correlate to the expected system downtime.

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
99.99999%	86.40 milliseconds	604.80	2.63 seconds	31.56 seconds

Table 3

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

Back-of-the-envelope estimation is all about the process. Solving the problem is more important than obtaining results. Interviewers may test your problem-solving skills. Here are a few tips to follow:

- Rounding and Approximation. It is difficult to perform complicated math operations during the interview. For example, what is the result of "99987 / 9.1"? There is no need to spend valuable time to solve complicated math problems. Precision is not expected. Use round numbers and approximation to your advantage. The division question can be simplified as follows: "100,000 / 10".
- Write down your assumptions. It is a good idea to write down your assumptions to be referenced later.
- Label your units. When you write down "5", does it mean 5 KB or 5 MB? You might confuse yourself with this. Write down the units because "5 MB" helps to remove ambiguity.
- Commonly asked back-of-the-envelope estimations: QPS, peak QPS, storage, cache, number of servers, etc. You can practice these calculations when preparing for an interview. Practice makes perfect.

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

Reference materials

[1] J. Dean.Google Pro Tip: Use Back-Of-The-Envelope-Calculations To Choose The Best Design:
http://highscalability.com/blog/2011/1/26/google-pro-tip-use-back-of-the-envelope-calculations-to-choo.html

[2] System design primer:
https://github.com/donnemartin/system-design-primer

[3] Latency Numbers Every Programmer Should Know:
https://colin-scott.github.io/personal_website/research/interactive_latency.html

[4] Amazon Compute Service Level Agreement:
https://aws.amazon.com/compute/sla/

[5] Compute Engine Service Level Agreement (SLA):
https://cloud.google.com/compute/sla

[6] SLA summary for Azure services:
https://azure.microsoft.com/en-us/support/legal/sla/summary/

04
A Framework For System Design Interviews

You have just landed a coveted on-site interview at your dream company. The hiring coordinator sends you a schedule for that day. Scanning down the list, you feel pretty good about it until your eyes land on this interview session - System Design Interview.

System design interviews are often intimidating. It could be as vague as "designing a well-known product X?". The questions are ambiguous and seem unreasonably broad. Your weariness is understandable. After all, how could anyone design a popular product in an hour that has taken hundreds if not thousands of engineers to build?

The good news is that no one expects you to. Real-world system design discussions take extremely complicated. For example, Google search is deceptively simple however, the amount of technology that underpins that simplicity is truly astonishing. If no one expects you to design a real-world system in an hour, what is the benefit of a system design interview?

The system design interview simulates real-life problem solving where two co-workers collaborate on an ambiguous problem and come up with a solution that meets their goals. The problem is open-ended, and there is no perfect answer. The final design is less important compared to the work you put in the design process. This allows you to demonstrate your design skill, defend your design choices, and respond to feedback in a constructive manner.

Let us flip the table and consider what goes through the interviewer's head as she walks into the conference room to meet you. The primary goal of the interviewer is to accurately assess your level. The last thing she wants is to give an inconclusive evaluation because the session has gone poorly and there are not enough signals. What is an interviewer looking for in a system design interview?

Many think that system design interview is all about a person's technical design skills. It is much more than that. An effective system design interview gives strong signals about a person's ability to collaborate, to work under pressure, and to resolve ambiguity constructively. The ability to ask good questions is also an essential skill, and many interviewers specifically look for this skill.

A good interviewer also looks for red flags. Over-engineering is a real disease of many engineers as they delight in design purity and ignore tradeoffs. They are often unaware of the compounding costs of over-engineered systems, and many companies pay a high price for that ignorance. You certainly do not want to demonstrate this tendency in a system design interview. Other red flags include narrow mindedness, stubbornness, etc.

In this chapter, we will go over some useful tips and introduce a simple and effective framework to solve system design interview problems.

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. However, there are steps and common ground to cover in every system design interview.

Step 1 - Understand the problem and establish design scope

"Why did the tiger rule?"

A head shot up in the back of the class.

"Yes, Jimmy?", the teacher responded.

"Because he was HUNGRY".

"Very good Jimmy."

Throughout his childhood, Jimmy has always been the first to answer questions in the class. Whenever the teacher asks a question, there is always a kid in the classroom who loves to take a crack at the question, no matter if he knows the answer or not. That is Jimmy.

Jimmy is an ace student. He takes pride in knowing all the answers fast. In exams, he is usually the first person to finish the questions. He is a teacher's top choice for any academic competition.

DON'T be like Jimmy.

In a system design interview, giving out an answer quickly without thinking gives you no bonus points. Answering without a thorough understanding of the requirements is a huge red flag as the interviewer is not a trivia contest. There is no right answer.

So, do not jump right in to give a solution. Slow down. 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 hard problem first. However, this approach is likely to lead you to design the wrong system. One of the most important skills as an engineer is to ask the right questions, make the proper assumptions, and gather all the information needed to build a system. So, do not be afraid to ask questions.

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. Here is a list of questions to help you get started:

- 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

If you are asked to design a news feed system, you want to ask questions that help you clarify the requirements. The conversation between you and the interviewer might look like this:

Candidate: Is this a mobile app? Or a web app? Or both?
Interviewer: Both.
Candidate: What are the most important features for the product?
Interviewer: 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?
Interviewer: 5000
Candidate: What is the traffic volume?
Interviewer: 10 million daily active users (DAU)
Candidate: Can feed contain images, videos, or just text?
Interviewer: It can contain media files, including both images and videos.

Above are some sample questions that you can ask your interviewer. It is important to understand the requirements and clarify ambiguities.

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 interviews love to talk and get involved.
- Draw box diagrams with two-tier clients/servers on the whiteboard or paper. This might include clients (mobile/web), APIs, web servers, data stores, cache, CDN, message queue, etc.
- Go back-of-the-envelope calculations to evaluate if your blueprint fits the scale constraints. Think out loud. Communicate with your interviewer if back-of-the-envelope is necessary before diving into it.

If possible, go through a few concrete use cases. This will help you frame the high-level design. It is also likely that the use cases would help you discover edge cases you have not yet considered.

Should we include API endpoints and database schema here? This depends on the problem. For large design problems like "Design Google search engine", this is a bit of too low level. For a problem like designing the backend for a multi-player poker game, this is a fair game. Communicate with your interviewer.

Example

Let us 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. All the details will be explained in the "Design A News Feed System" chapter.

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 1 and Figure 2 present high-level designs for feed publishing and news feed building, flows, respectively.

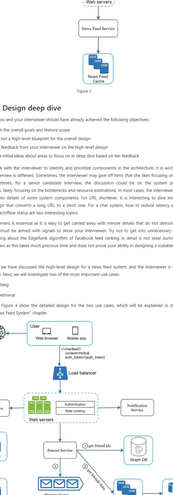

Figure 1

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. It is worth stressing that every interview is different. Sometimes, the interviewer may give off hints that she likes focusing on high-level designs. Sometimes, for a senior candidate interview, the discussion could be on the system performance (bandwidth/etc.) likely focusing on the bandwidth 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. You must be armed with signals to show your interviewer. 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.

Figure 3

Figure 4

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. Reiterating 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

System design interview questions are usually very broad, and 45 minutes or an hour is not enough to cover the entire design. Time management is essential. How much time should you spend on each step? The following is a very rough guide on distributing your time in a 45-minute interview session. Please remember this is a rough estimate, and the actual time distribution depends on the scope of the problem and the requirements from the interviewer.

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

05
Design A Rate Limiter

In a network system, a rate limiter is used to limit the number of client requests allowed to be sent over a specified period. If the number of requests sent exceeds the threshold defined by the limiter, the additional requests will be stopped. For example, an API that allows 2 requests per second, a request in excess of the limit will be stopped.

- A user can write to no more than 2 posts per second.
- The maximum number of accounts that can be created in a day is 10.
- A maximum of 5 pages can be liked per day by a user.

In this chapter, you are asked to design a rate limiter. Besides writing the design, we first look at the benefits of using an API rate limiter.

- Misbehaving clients or bots: Some clients are designed to send a large number of requests to a service. A rate limiter can be used to prevent misbehaving client from affecting the example. Twitter limits the number of tweets a user can make per day. Google Docs limits the number of times a user can open a document. Facebook limits the number of pages a user can like per day. Rate limiting ensures that a service is not affected by such misbehaving clients or bots.
- Security and Spam: Limiting access requests helps block spam and other security attacks. For example, Google limits the number of times a user can try to log into its accounts. Twitter limits the number of tweets that can be made per day from a single IP address. Facebook allows only a certain number of URLs to be added to a post per day. Rate limiting helps block spam and other security attacks based on the number of requests they generate.
- Control and manage traffic: Rate limiters can be used to control traffic. For example, a rate limiter may be used to ensure that a specified number of requests are made to a service per second. Or a rate limiter may be used to allocate a certain quota for a service per day or per hour. For example, Google APIs use rate limiting to serve different quotas to their API users.

Step 1 - Understand the problem and establish design scope

Question: What kind of rate limiter are we supposed to design? Is it a client side API limiter or a server side API limiter?
Answer: We are interested in a server side API limiter.

Question: Does the rate limiter need to be able to limit requests based on some criteria? For example, can we use IP address, user ID, or other properties to identify the type of client that is making a request?
Answer: Yes, the system should be flexible enough to support different sets of criteria to define how we limit requests.

Question: Does the system need to work in a distributed environment?
Answer: Yes, this is a very important question. We will talk about more details later.

Question: Do we need to inform users that their request has been throttled?
Answer: Yes, we need to inform users that their request has been throttled.

Question: Can we assume that the number of requests will be large? Do we need to support a large number of requests?
Answer: Yes, this is a common scenario for a company that implemented its organization scale!

Question: Should the system be a standalone service?
Answer: No, it should be a standalone service.

Question: Can we assume that the rate limiter will not affect the performance of the API servers?
Answer: Yes, we can assume that.

- Client side: Client side is a separate process or a module that implemented in organization scale!
- Server side: Server side is a separate process or a module that implemented in organization scale!
- Middleware: Middleware is a software that provides services to software applications.

Question: Do we need to inform users that their request has been throttled?
Answer: Yes, we need to inform users that their request has been throttled.

Question: Can we assume that the number of requests will be large? Do we need to support a large number of requests?
Answer: Yes, this is a common scenario for a company that implemented in organization scale!

Question: Should the system be a standalone service?
Answer: No, it should be a standalone service.

Question: Can we assume that the rate limiter will not affect the performance of the API servers?
Answer: Yes, we can assume that.

- Client side: Client side is a separate process or a module that implemented in organization scale!
- Server side: Server side is a separate process or a module that implemented in organization scale!
- Middleware: Middleware is a software that provides services to software applications.

Based on the client and server side implementations, there is an alternative way instead of putting a rate limiter in client side or server side, we can create a rate limiter middleware, and let the middleware provide rate limiting services.

Let us use an example to illustrate. If a company has backend services in AWS, AWS API Gateway provides rate limiting as a middleware feature. Cloudflare also provides rate limiting middleware services. We can also write our own middleware to integrate with our backend services.

- Client side implementation: If client side is available, rate limiting can be implemented on client side. This is a good option when client side is available, and it is not required to be compatible with all clients.
- Server side implementation: When it comes to implementation, a server side implementation is more common. It is also easier to write a server side rate limiter when we have control on the server side. With server side implementation, we can be sure that the client side is not under our control, and we can use server side implementation to block client requests that are not properly formatted, or requests that are not properly routed.
- Middleware: Middleware is a software that provides services to software applications. Middleware is a good option if we do not want to use our own servers to implement the rate limiter, or we want to use a cloud provider's rate limiter services.

Algorithms for rate limiting

Rate limiting can be implemented using different algorithms, each with its Pros and Cons. In this section and under, we will discuss different algorithms and their benefits and drawbacks. This will help us to decide which algorithm to use to implement our rate limiter.

- Token bucket
- Leaking bucket
- Fixed window counter
- Sliding window log
- Sliding window counter

Token bucket algorithm

The token bucket algorithm is widely used for rate limiting. It is simple, well understood and commonly used by internet companies. This algorithm is used by both Amazon and Stripe to rate limit their API requests.

- Token bucket: A token bucket is a container that holds predefined number of tokens.
- Refill rate: How many tokens are added into the token bucket per second.

Figure 5 illustrates how token consumption, refill, and token limiting logic work. In this example, the token bucket size is 4, and the refill rate is per 1 minute.

Figure 5

The token bucket algorithm takes two parameters:

- The bucket size: How many tokens are allowed in the bucket.
- The refill rate: How many tokens are put into the bucket every second.

How many buckets do we need? This varies, and it depends on the rate limiting rules. Here are a few examples:

- It is possible to have different buckets for different types of API endpoints. For example, a user can make 1 post per second, add 150 friends per day, and 5 million requests per day to the API endpoint.
- It is possible to have different buckets for different types of IP addresses. Let's assume a bucket is used for each IP address.
- It is possible to have different buckets for different types of API endpoints. Let's assume a bucket is used for each API endpoint.

Figure 6 illustrates how token consumption, refill, and token limiting logic work. In this example, the token bucket size is 4, and the refill rate is per 1 minute.

Figure 6

The token bucket algorithm takes two assumptions:

- The rate limiter can send enough tokens to the bucket every time a specified period of time passes.
- Each request consumes one token.

Let's look at two examples to illustrate the algorithm.

- Assume the refill rate is 2 token per minute, and the bucket size is 4 token. In this example, we need to add 2 tokens into the bucket every minute. We can also make 4 requests to the API endpoint before we need to wait for a while. For example, we can make 4 requests per second, or 2 requests per second, or 1 request per second.
- Assume the refill rate is 4 token per minute, and the bucket size is 4 token. In this example, we need to add 4 tokens into the bucket every minute. We can also make 4 requests to the API endpoint before we need to wait for a while.

Leaking bucket algorithm

The leaking bucket algorithm takes the following two parameters:

- Bucket size: It is equal to the queue size. The queue stores the requests to be processed in a fixed rate.
- Outflow rate: It defines how many requests can be processed in a fixed time, such as every second.

Depending on how many requests are added to the queue, we have four different cases:

- Exactly the same number of requests are added to the queue.
- Fewer requests are added than the queue size.
- Fewer requests are added than the queue size, and a certain number of requests are added to the queue.
- More requests are added than the queue size.

Figure 7 illustrates how leaking bucket algorithm works.

Figure 7

A large problem with this algorithm is that a burst of traffic at the edge of the window causes more requests than allowed quota to be processed. Figure 8 illustrates how this problem happens.

Figure 8

In Figure 8, the queue allows a maximum of 4 requests per minute, and the available quota resets at the fixed minutes. As shown in the chart, there are 4 requests in the first minute, and 4 requests in the second minute. It seems that everything is within the limit. However, the 4 requests in the second minute are from the 1st minute. This means that the system will process 8 requests in a fixed time, which is twice as many as the allowed quota.

Fixed window counter algorithm

Fixed window counter algorithm works as follows:

- The algorithm divides the timeline into fixed time windows and assign a counter for each window.
- Each request increments the counter by one for the window it falls into. If the counter reaches a predefined threshold within a window, the new request in the window will not be processed until a new time window is added.

Let us use a concrete example to see how it works. In Figure 9, the time unit is 1 second and the system allows a maximum of 3 requests in a fixed window. Assume the time unit is 1 second and the system allows a maximum of 3 requests in a fixed window, the number of requests in each window is illustrated in Figure 9.

Figure 9

A large problem with this algorithm is that a burst of traffic at the edge of the window causes more requests than allowed quota to be processed. Figure 10 illustrates how this problem happens.

Figure 10

In Figure 10, the system allows a maximum of 3 requests per minute, and the available quota resets at the fixed minutes. As shown in the chart, there are 3 requests in the first minute, and 3 requests in the second minute. It seems that everything is within the limit. However, the 3 requests in the second minute are from the 1st minute. This means that the system will process 6 requests in a fixed time, which is twice as many as the allowed quota.

Sliding window log algorithm

Sliding window log algorithm works as follows:

- The algorithm keeps a log for each request. For example, if we set the limit to be 60 per hour, the log for each request is kept in the cache, and it expires after 1 hour.
- When a request arrives, the system checks if the queue is full. If it is not, the request is added to the queue.
- Remove any outdated requests from the queue.
- Request is accepted if the queue is not full.

Figure 11 illustrates how the algorithm works.

Figure 11

Sliding window log algorithm has the following two parameters:

- Window size: How long the sorted set is kept for a user.
- Number of allowed requests per window size.

Assume the rate limiter is to keep 2 requests per minute for each user, Redis will create a sorted set with key = user_request_timestamp. Each request is a value in the sorted set. For example, if the first time a user request to the API is at 1:00:00 AM, a new key-value pair will be added to the sorted set. In this example, the key is 1:00:00 AM. If a request is kept in the sorted set, it will be accepted. To see if more requests can be accepted, we simply count the number of entries in the sorted set. If the number of entries in the sorted set is less than the allowed quota, a request is accepted. Otherwise, it is rejected.

Sliding window counter algorithm

Sliding window counter algorithm combines the fixed window counter and sliding window log algorithms. Let's assume the rate limiter is to allow a maximum of 7 requests per minute, and there were 5 requests in the previous minute and 3 requests in the current minute. For a new request, the sliding window counter algorithm uses the following formula to calculate the number of requests in the rolling window:

Number of requests in current window + number of requests in previous window * overlap percentage of the rolling window and previous window

Using this formula, we get 3 + 5 * 0.70 = 6.5 requests. Depending on the use case, we can round the number either up or down. In our example, we round down to 6.

Figure 12 illustrates the time windows comprised of 1 minute and the rolling window. For example, the current request arrival time is at the 1:00:30 mark, which is 70% of the previous minute.

Figure 12

Because the algorithm takes only 2 minutes into account, it requires very little memory. The algorithm works well when the traffic is not evenly distributed in the previous minute. For example, if the request arrival time is at the 1:00:30 mark, we can assume that the previous minute is filled with 5 requests. The previous minute may have had 1 request at the beginning of the minute and 4 requests at the end of the minute, or it may have had 4 requests at the beginning of the minute and 1 request at the end of the minute. We can also assume that the previous minute is filled with 5 requests. Based on the previous assumption, the sliding window counter algorithm works well.

Step 2 - Design deep dive

Let's look at the high level architecture for various rate limiting rules and how they can be implemented in distributed environment. We also discuss how to handle rate limiting in distributed environment in detail.

Rate limiting rules

Let's take example from rate limiting, a company's API has API private keys of the organization and each or some API endpoints have different rate limiting rules.

```
domain: messaging
descriptors:
  - key: message_type
    value: marketing
    rate_limit:
      unit: day
      requests_per_unit: 5
  - key: message_type
    value: customer_support
    rate_limit:
      unit: hour
      requests_per_unit: 100
```

Rate limiting rules are stored in configuration files in the cloud. In the example below, the rule states that a user can send at most 5 marketing messages per day and 100 customer support messages per hour.

Exceeding the rate limit

In a company's or client's API, when a HTTP request is sent to the server, it requires to the client. Depending on the use case, the response can be sent later or sent immediately. The response indicates that the request is rate limited (429 Too Many Requests). Depending on the use case, the response may be dropped or processed.

Rate limiter headers

Rate limiter headers are used to inform the client how many requests are allowed before the request is throttled. The rate limiter uses the following HTTP response headers:

- X-Ratelimit-Remaining: The remaining number of allowed requests within the current time window.
- X-Ratelimit-Limit: The number of calls the client can make per time window.
- X-Ratelimit-Retry-After: The number of seconds to wait until you can make a request again without getting throttled.

When a client sends too many requests, a 429 Too Many Requests error and X-Ratelimit-Retry-After header are returned to the client.

Detailed design

Figure 13 illustrates a detailed design of the system.

Client
Cache
Rate limiter
Rate limiter middleware
API servers
Rate limiter rules
Cache
Rate limiter middleware
API servers
Rate limiter rules
Cache
Rate limiter middleware
API servers
Rate limiter rules
Cache

06
Design Consistent Hashing

To achieve horizontal scaling, it is important to distribute requests/data efficiently and evenly across servers. Consistent hashing is a commonly used technique to achieve this goal. But first, let us take an in-depth look at the problem.

The rehashing problem

If you have n cache servers, a common way to balance the load is to use the following hash method:

serverIndex = hashKey % N, where N is the size of the server pools.

Let us use an example to illustrate how it works. As shown in Table 1, we have 4 servers and 8 string keys with their hashes.

key	hash	hash % 4
key0	10336617	1
key1	26145564	0
key2	19131146	2
key3	35862686	0
key4	34059265	1
key5	27581703	3
key6	30164879	2
key7	22535551	3

Table 1

To fetch the server where a key is stored, we perform the modulo operation (Step 3). For instance, hashKey0 % 4 = 1 means a client must contact server 1 to fetch the cached data. Figure 1 shows the distribution of keys based on Table 1.

serverIndex = hashKey % 4

Figure 1

This approach works well when the size of the server pool is fixed, and the data distribution is even. However, problems arise when new servers are added, or existing servers are removed. For example, if server 1 goes offline, the size of the server pool becomes 3. Using the same hash function, we get the same hash value for a key. But applying modulo operation gives us different server indexes because the number of servers is reduced by 1. We get the results shown in Table 2 by applying hash % 3.

key	hash	hash % 3
key0	10336617	0
key1	26145564	0
key2	19131146	1
key3	35862686	2
key4	34059265	1
key5	27581703	0
key6	30164879	1
key7	22535551	0

Table 2

Figure 2 shows the new distribution of keys based on Table 2.

serverIndex = hash % 3

Figure 2

As shown in Figure 2, most keys are redistributed, not just the ones originally stored in the offline server (server 1). This means that when server 1 goes offline, most cache clients will connect to the wrong servers to fetch data. This causes a storm of cache misses. Consistent hashing is an effective technique to mitigate this problem.

Consistent hashing

Quoted from Wikipedia: "Consistent hashing is a special kind of hashing such that when a hash table is re-sized and consistent hashing is used, only k/n keys need to be remapped on average, where k is the number of keys, and n is the number of slots. In contrast, in most traditional hash tables, a change in the number of array slots causes nearly all keys to be remapped."[1]

Hash space and hash ring

Now we understand the definition of consistent hashing, let us find out how it works. Assume SHA-1 is used as the hash function f, and the output range of the hash function is: $x_0, x_1, x_2, ..., x_n$. In cryptography, SHA-1's hash space goes from 0 to $2^{160} - 1$. That means x_0 corresponds to 0, x_n on corresponds to $2^{160} - 1$, and all the other hash values in the middle fall between 0 and $2^{160} - 1$. Figure 3 shows the hash space.

Figure 3

By collecting both ends, we get a hash ring as shown in Figure 4.

Figure 4

Hash servers

Using the same hash function f, we map servers based on server IP or name onto the ring. Figure 5 shows that 4 servers are mapped on the hash ring.

Figure 5

Hash keys

One thing worth mentioning is that hash function used here is different from the one in "the rehashing problem," and there is no modulo operation. As shown in Figure 6, 4 cache keys (key0, key1, key2, and key3) are hashed onto the hash ring.

Figure 6

Server lookup

To determine which server a key is stored on, we go clockwise from the key position on the ring until a server is found. Figure 7 applies this process. Going clockwise, key0 is stored on server 0; key1 is stored on server 1; key2 is stored on server 2 and key3 is stored on server 3.

Figure 7

Add a server

Using the logic described above, adding a new server will only require redistribution of a fraction of keys.

Quoted from Wikipedia: "If a new server is added, only key5 needs to be redistributed, s1, s2, and s3 remain on the same servers. Let us take a deep look at the key5. Before we add server 4, key5 is stored on server 0. Now key5 will be stored on server 4 because server 4 is the first server it encounters by going clockwise from key5's position on the ring. The other keys are not redistributed based on consistent hashing algorithm.

Figure 8

Remove a server

When a server is removed, only a small fraction of keys require redistribution with consistent hashing. In Figure 9, when server 1 is removed, only key1 must be remapped to server 2. The rest of the keys are unaffected.

Figure 9

When a server is removed, only a small fraction of keys require redistribution with consistent hashing. In Figure 9, when server 1 is removed, only key1 must be remapped to server 2. The rest of the keys are unaffected.

There are two important changes for the proposed approach:

1. Each server is mapped to multiple places on the ring. For instance, if servers are mapped to positions listed in Figure 11, most of the keys are stored on server 2. However, server 1 and server 3 have no data.

Figure 11

A technique called virtual node or replicas is used to solve these problems.

Virtual nodes

A virtual node refers to the real node, and each server is represented by multiple virtual nodes on the ring. In Figure 12, both server 0 and server 1 have 3 virtual nodes. The 3 is arbitrarily chosen and in real-world systems, the number of virtual nodes is much larger. Instead of using s0, we have s0_0, s0_1, and s0_2 to represent server 0 on the ring. Similarly, s1_0, s1_1, and s1_2 represent server 1 on the ring. With virtual nodes, each server is responsible for multiple partitions. Partitions (edges) with label of are managed by server 0. On the other hand, partitions with label of are managed by server 1.

Figure 12

To find which server a key is stored on, we go clockwise from the key's location and find the first virtual node encountered on the ring. In Figure 13, to find out which server s0 is stored on, we go clockwise from s0's location and find virtual node s1_1, which refers to server 1.

Figure 13

As the number of virtual nodes increases, the distribution of keys becomes more balanced. This is because the standard deviation gets smaller with more virtual nodes, leading to balanced data distribution. Standard deviation measures how data are spread out. The outcome of an experiment carried out by online research [2] shows that with one or two hundred virtual nodes, the standard deviation is between 5% (200 virtual nodes) and 10% (100 virtual nodes) of the mean. The standard deviation will be smaller when we increase the number of virtual nodes. However, more spaces are needed to store data about virtual nodes. This is a trade-off, and we can tune the number of virtual nodes to fit our system requirements.

Find affected keys

When a server is added or removed, a fraction of data needs to be redistributed. How can we find the affected range to redistribute that?

In Figure 14, server 4 is added onto the ring. The affected range starts from s1 (newly added node) and moves anticlockwise around the ring until a server is found (s0). Thus, keys located between s0 and s1 need to be redistributed to s4.

Figure 14

When a server (s1) is removed as shown in Figure 15, the affected range starts from s1 (removed node) and moves anticlockwise around the ring until a server is found (s2). Thus, keys located between s0 and s1 need to be redistributed to s2.

Figure 15

Wrap up

In this chapter, we had an in-depth discussion about consistent hashing, including why it is needed and how it works. The benefits of consistent hashing include:

- Minimized keys are redistributed when servers are added or removed.
- It is easy to scale horizontally because data are more evenly distributed.
- Mitigate 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. Consistent hashing helps to mitigate the problem by distributing the data more evenly.

Consistent hashing is widely used in real-world systems, including some notable ones:

- Partitioning component of Amazon's Dynamo database [3]
- Data partitioning across the cluster in Apache Cassandra [4]
- Discord chat application [5]
- Akamai content delivery network [6]
- Maglev network load balancer [7].

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

Reference materials

[1] Consistent hashing:
https://en.wikipedia.org/wiki/Consistent_hashing

[2] Consistent Hashing:
https://tom-e-white.com/2007/11/consistent-hashing.html

[3] Dynamo: Amazon's Highly Available Key-value Store:
https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

[4] Cassandra - A Decentralized Structured Storage System:
http://www.cs.cornell.edu/Projects/ladis2009/papers/lakshman-ladis2009.PDF

[5] How Discord Scaled Elixir to 5,000,000 Concurrent Users:
https://blog.discord.com/scaling-elixir-f9b8e1e7fb19

[6] CS168: The Modern Algorithmic Toolbox Lecture #1: Introduction and Consistent Hashing:
https://theory.stanford.edu/~tim/s16/l/l1.pdf

[7] Maglev: A Fast and Reliable Software Network Load Balancer:
https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/946928.pdf

07

Design A Key-value Store

Amazon Dynamo is a highly reliable and highly scalable key-value store in its internal distribution platform. Dynamo offers a simple interface for storing and retrieving data. Dynamo is used by many services at Amazon.

Facebook Cassandra is a distributed storage system for structured data. Cassandra is used by many services at Facebook, including Facebook Messenger, Facebook Search, and Facebook Ads. Cassandra is a key-value store that can be thought of as an extension to Dynamo. Cassandra is also used by many companies.

Redis is an open-source, in-memory key-value store. Redis is used by many companies. Redis is used by many companies.

Google Bigtable is a key-value store for storing large data. Bigtable. The Bigtable is a public implementation of Google Bigtable. Bigtable is used by many companies, including Google Cloud Platform.

Amazon DynamoDB is a key-value store for storing large data. Amazon DynamoDB is used by many companies.

Understand the problem and establish design scope

Key-value stores offer a simple interface for storing and retrieving data. Key-value stores are used by many companies. Key-value stores are used by many companies. Key-value stores are used by many companies, and scalability is the biggest challenge. Key-value stores are used by many companies.

- The key-value pairs are small in size (less than 10 KB).
- Ability to store big data.
- High availability: The system responds quickly even during failures.
- High scalability: The system can be scaled to support very large data set.
- Automatic scaling: The addition of servers should be automatic based on traffic.
- Tunable consistency.
- Low latency.

Single server key-value store

Key-value stores are used by many companies. Key-value stores are used by many companies. Key-value stores are used by many companies. Key-value stores are used by many companies. Key-value stores are used by many companies. Key-value stores are used by many companies.

Distributed key-value store

Key-value stores are used by many companies. Key-value stores are used by many companies. Key-value stores are used by many companies. Key-value stores are used by many companies. Key-value stores are used by many companies. Key-value stores are used by many companies.

CAP theorem

CAP theorem states that it is impossible to provide more than two of the following three guarantees in a distributed system: Consistency, Availability, Partition Tolerance. Let's clarify the definitions of the three characteristics.

Consistency: Consistency means all clients see the same data at the same time no matter which node they connect to.

Availability: Availability means any client which requests data gets a response even in the event of a node failure. Availability means that the system continues to function and remain responsive for business purposes in the event of such a failure.

Partition Tolerance: A partition is a communication break between two nodes. Partition tolerance means the system continues to operate despite network partitions.

Figure 1

Networks may often one-time distributed failures. Therefore, we have to choose between the two remaining characteristics.

CP (consistency and partition tolerance) systems: CP key-value stores supports consistency and partition tolerance at the cost of availability.

AP (availability and partition tolerance) systems: AP key-value stores supports availability and partition tolerance. Here, we have to define the consistency model as AP systems give up some consistency.

CA (consistency and availability) systems: CA key-value stores supports consistency and availability. Since network failure is unavoidable, a distributed system must tolerance network failure. Therefore, CA systems can not exist in real-world distributed systems.

Which one does Dynamo, Cassandra, Redis, and Zookeeper belong to? Dynamo is understood to be a data store of choice for many Amazon services requiring high availability. Cassandra offers tunable consistency. Redis is a data structure store. Zookeeper is a distributed coordination service.

Ideal characteristics

Let's discuss some ideal characteristics of a key-value store. Key-value stores are used by many companies. Key-value stores are used by many companies. Key-value stores are used by many companies. Key-value stores are used by many companies.

System components

In this section, we will discuss the following core components and technique used to build a key-value store:

- Data partition
- Data replication
- Consistency
- Inconsistency resolution
- Handling failures
- Tombstones
- Tunable consistency
- Handling permanent failures
- Anti-entropy
- Gossip protocol
- Membership and failure detection

Data partition

For small applications, it is feasible to keep the entire data in a single server. The disadvantage of this approach is that it is not feasible to fit the whole application into a single server. There are two challenges when partitioning the data:

- Distribute data across multiple servers evenly.
- Minimize data movement when nodes are added or removed.

Consistent hashing is a great technique to address these two challenges. Consistent hashing is a great technique to address these two challenges. Consistent hashing is a great technique to address these two challenges. Consistent hashing is a great technique to address these two challenges.

Data replication

To achieve high availability and reliability, data must be replicated asynchronously over N servers, where N is a configurable parameter. These N servers are chosen as follows: After a key is mapped to a server, we walk clockwise from that server to find N servers on the ring. For example, if N = 3, we will find 3 servers.

Consistency

Since data is replicated on multiple nodes, it must be synchronized across replicas. Quorum consensus can guarantee consistency for both read and write operations. Consider the following:

- N = 3 (the number of replicas)
- W = 2 (a write quorum of size 2 is a write operation is considered as successful when operation from at least 2 replicas are received)
- R = 2 (a read quorum of size 2 is a read operation is considered as successful when operation from at least 2 replicas are received)

Consider the following scenarios based on Figure 2:

Figure 2: Quorum consensus (N = 3, W = 2, R = 2)

W = 1 and R = 1 means that a write operation is considered as successful when operation from at least 1 replica are received. If a read operation is sent to three replicas, we will wait for one response. If the response is fast, it could be from a slow node. Setting W = 1 and R = 1 means that a write operation is considered as successful when operation from at least 1 replica are received. If a read operation is sent to three replicas, we will wait for one response. If the response is fast, it could be from a slow node.

W = 3 and R = 1 means that a write operation is considered as successful when operation from all 3 replicas are received. If a read operation is sent to three replicas, we will wait for one response. If the response is fast, it could be from a slow node.

W = 1 and R = 3 means that a write operation is considered as successful when operation from at least 1 replica are received. If a read operation is sent to three replicas, we will wait for three responses. If the response is fast, it could be from a slow node.

Strong consistency is usually achieved by forcing a replica not to accept new read/write operations until all replicas have agreed on current read/write operations. This approach is not ideal for high-availability systems. Since some replicas may be slow, read/write operations could be synchronized across all replicas. Strong consistency usually means that read/write operations are slow. Strong consistency usually means that read/write operations are slow.

Inconsistency resolution: versioning

Replication gives high availability and eventual consistency. Replication gives high availability and eventual consistency. Replication gives high availability and eventual consistency. Replication gives high availability and eventual consistency. Replication gives high availability and eventual consistency. Replication gives high availability and eventual consistency.

Versioning and vector clock are used to resolve inconsistency. Versioning means that a piece of data is assigned a new version when it is changed. Versioning means that a piece of data is assigned a new version when it is changed. Versioning means that a piece of data is assigned a new version when it is changed. Versioning means that a piece of data is assigned a new version when it is changed.

Let's assume that S1 and S2 are two replicas of a key-value store. Let's assume that S1 and S2 are two replicas of a key-value store. Let's assume that S1 and S2 are two replicas of a key-value store. Let's assume that S1 and S2 are two replicas of a key-value store.

Figure 3: Replication

In the example, the network was down and two clients started to write to two servers. Let's assume. Let's assume. Let's assume that the conflict of the two servers. Let's assume that the two servers. Let's assume that the two servers. Let's assume that the two servers. Let's assume that the two servers.

Let's assume that S1 and S2 are two replicas of a key-value store. Let's assume that S1 and S2 are two replicas of a key-value store. Let's assume that S1 and S2 are two replicas of a key-value store. Let's assume that S1 and S2 are two replicas of a key-value store.

Figure 4: Replication

Let's assume that S1 and S2 are two replicas of a key-value store. Let's assume that S1 and S2 are two replicas of a key-value store. Let's assume that S1 and S2 are two replicas of a key-value store. Let's assume that S1 and S2 are two replicas of a key-value store.

1. A client writes a data item D1 to the system. Let's assume that this write is handled by server Sx which now has the system vector clock [Sx, 1].
2. Another client reads the latest D1, updates it to D2, and writes back. All descendants from D1 of D2 overwrite D1. There is only one version in the system, D2. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx.
3. Another client reads the latest D1, updates it to D3, and writes back. Let's assume that this write is handled by server Sy. Let's assume that this write is handled by server Sy. Let's assume that this write is handled by server Sy. Let's assume that this write is handled by server Sy.
4. Another client reads the latest D2 and D3, resolves the conflict, and writes back. Let's assume that this write is handled by server Sz. Let's assume that this write is handled by server Sz. Let's assume that this write is handled by server Sz. Let's assume that this write is handled by server Sz.
5. Let's assume that a client reads D4 from Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx.

Handling failures

Let's assume that a client reads D4 from Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx.

Tombstones

Let's assume that a client reads D4 from Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx.

Tunable consistency

Let's assume that a client reads D4 from Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx.

Handling permanent failures

Let's assume that a client reads D4 from Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx.

Handling temporary failures

Let's assume that a client reads D4 from Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx.

Anti-entropy

Let's assume that a client reads D4 from Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx.

Gossip protocol

Let's assume that a client reads D4 from Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx.

Membership and failure detection

Let's assume that a client reads D4 from Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx.

Summary

Let's assume that a client reads D4 from Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx. Let's assume that this write is handled by server Sx.

References materials

[1] Amazon DynamoDB: https://aws.amazon.com/dynamodb/
[2] Cassandra: https://cassandra.apache.org/
[3] Redis: https://redis.io/
[4] Google Bigtable: https://cloud.google.com/bigtable/
[5] Amazon DynamoDB: https://aws.amazon.com/dynamodb/
[6] Cassandra: https://cassandra.apache.org/
[7] Redis: https://redis.io/
[8] Google Bigtable: https://cloud.google.com/bigtable/
[9] Amazon DynamoDB: https://aws.amazon.com/dynamodb/
[10] Cassandra: https://cassandra.apache.org/
[11] Redis: https://redis.io/
[12] Google Bigtable: https://cloud.google.com/bigtable/
[13] Amazon DynamoDB: https://aws.amazon.com/dynamodb/
[14] Cassandra: https://cassandra.apache.org/
[15] Redis: https://redis.io/
[16] Google Bigtable: https://cloud.google.com/bigtable/

08

Design A Unique ID Generator In Distributed Systems

In this chapter, you are asked to design a unique ID generator in distributed systems. Your first thought might be to use a primary key with the *auto_increment* attribute in a traditional database. However, *auto_increment* does not work in a distributed environment because a single database server is not large enough and generating unique IDs across multiple databases with minimal delay is challenging.

Here are a few examples of unique IDs:

```
+--------------------+
|  user_id           |
+--------------------+
|  1227238262110117894 |
+--------------------+
|  1241107244890095715 |
+--------------------+
|  1243643959492173824 |
+--------------------+
|  1247686581489692673 |
+--------------------+
|  1567981766075453448 |
+--------------------+
```

Figure 1

Step 1 - Understand the problem and establish design scope

Asking clarification questions is the first step to tackle any system design interview question. Here is an example of candidate-interviewer interaction:

Candidate: What are the characteristics of unique IDs?
Interviewer: IDs must be unique and sortable.

Candidate: For each new record, does ID increment by 1?
Interviewer: The ID increments by time but not necessarily only increments by 1. IDs created in the evening are larger than those created in the morning on the same day.

Candidate: Do IDs only contain numerical values?
Interviewer: Yes, that is correct.

Candidate: What is the ID length requirement?
Interviewer: IDs should fit into 64-bit.

Candidate: What is the scale of the system?
Interviewer: The system should be able to generate 10,000 IDs per second.

Above are some of the sample questions that you can ask your interviewer. It is important to understand the requirements and clarify ambiguities. For this interview question, the requirements are listed as follows:

- IDs must be unique.
- IDs are numerical values only.
- IDs fit into 64-bit.
- IDs are ordered by date.
- Ability to generate over 10,000 unique IDs per second.

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

Multiple options can be used to generate unique IDs in distributed systems. The options we considered are:

- Multi-master replication
- Universally unique identifier (UUID)
- Ticket server
- Twitter snowflake approach

Let us look at each of them, how they work, and the pros/cons of each option.

Multi-master replication

As shown in Figure 2, the first approach is multi-master replication.

Figure 2

This approach uses the databases' *auto_increment* feature. Instead of increasing the next ID by 1, we increase it by *k*, where *k* is the number of database servers in use. As illustrated in Figure 2, next ID to be generated is equal to the previous ID in the same server plus 2. This solves some scalability issues because IDs can scale with the number of database servers. However, this strategy has some major drawbacks:

- Hard to scale with multiple data centers
- IDs do not go up with time across multiple servers.
- It does not scale well when a server is added or removed.

UUID

A UUID is another easy way to obtain unique IDs. UUID is a 128-bit number used to identify information in computer systems. UUID has a very low probability of getting collusion. Quoted from Wikipedia, "after generating 1 billion UUIDs every second for approximately 100 years would the probability of creating a single duplicate reach 50%" [1].

Here is an example of UUID: *09c93e62-50b4-468d-bf8a-c07e1040bfb2*. UUIDs can be generated independently without coordination between servers. Figure 3 presents the UUIDs design.

Figure 3

In this design, each web server contains an ID generator, and a web server is responsible for generating IDs independently.

Pros:

- Generating UUID is simple. No coordination between servers is needed so there will not be any synchronization issues.
- The system is easy to scale because each web server is responsible for generating IDs they consume. ID generator can easily scale with web servers.

Cons:

- IDs are 128 bits long, but our requirement is 64 bits.
- IDs do not go up with time.
- IDs could be non-numeric.

Ticket Server

Ticket servers are another interesting way to generate unique IDs. Flicker developed ticket servers to generate distributed primary keys [2]. It is worth mentioning how the system works.

Figure 4

The idea is to use a centralized *auto_increment* feature in a single database server (Ticket Server). To learn more about this, refer to flicker's engineering blog article [2].

Pros:

- Numeric IDs.
- It is easy to implement, and it works for small to medium-scale applications.

Cons:

- Single point of failure. Single ticket server means if the ticket server goes down, all systems that depend on it will face issues. To avoid a single point of failure, we can set up multiple ticket servers. However, this will introduce new challenges such as data synchronization.

Twitter snowflake approach

Approaches mentioned above give us some ideas about how different ID generation systems work. However, none of them meet our specific requirements; thus, we need another approach. Twitter's unique ID generation system called "snowflake" [3] is inspiring and can satisfy our requirements.

Divide and conquer is our friend. Instead of generating an ID directly, we divide an ID into different sections. Figure 5 shows the layout of a 64-bit ID.

1 bit	41 bits	5 bits	5 bits	12 bits
0	timestamp	datacenter ID	machine ID	sequence number

Figure 5

Each section is explained below.

- Sign bit: 1 bit. It will always be 0. This is reserved for future uses. It can potentially be used to distinguish between signed and unsigned numbers.
- Timestamp: 41 bits. Milliseconds since the epoch or custom epoch. We use Twitter snowflake default epoch 1288834974657, equivalent to Nov 04, 2010, 01:42:54 UTC.
- Datacenter ID: 5 bits, which gives us $2 \wedge 5 = 32$ datacenters.
- Machine ID: 5 bits, which gives us $2 \wedge 5 = 32$ machines per datacenter.
- Sequence number: 12 bits. For every ID generated on that machine/process, the sequence number is incremented by 1. The number is reset to 0 every millisecond.

Step 3 - Design deep dive

In the high-level design, we discussed various options to design a unique ID generator in distributed systems. We settle on an approach that is based on the Twitter snowflake ID generator. Let us dive deep into the design. To refresh our memory, the design diagram is relisted below.

1 bit	41 bits	5 bits	5 bits	12 bits
0	timestamp	datacenter ID	machine ID	sequence number

Figure 6

Datacenter IDs and machine IDs are chosen at the startup time, generally fixed once the system is up running. Any changes in datacenter IDs and machine IDs require careful review since an accidental change in those values can lead to ID conflicts. Timestamp and sequence numbers are generated when the ID generator is running.

Timestamp

The most important 41 bits make up the timestamp section. As timestamps grow with time, IDs are sortable by time. Figure 7 shows an example of how binary representation is converted to UTC. You can also convert UTC back to binary representation using a similar method.

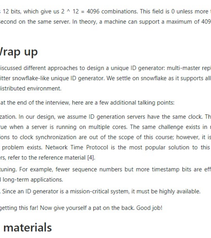

Figure 7

The maximum timestamp that can be represented in 41 bits is

$2 \wedge 41 - 1 = 2199023255551$ milliseconds (ms), which gives us: ~ 69 years = *2199023255551 ms / 1000 / 365 days / 24 hours/ 3600 seconds*. This means the ID generator will work for 69 years and having a custom epoch time close to today's date delays the overflow time. After 69 years, we will need a new epoch time or adopt other techniques to migrate IDs.

Sequence number

Sequence number is 12 bits, which give us $2 \wedge 12 = 4096$ combinations. This field is 0 unless more than one ID is generated in a milliasecond on the same server. In theory, a machine can support a maximum of 4096 new IDs per milliasecond.

Step 4 - Wrap up

In this chapter, we discussed different approaches to design a unique ID generator: multi-master replication, UUID, ticket server, and Twitter snowflake-like unique ID generator. We settle on snowflake as it supports all our use cases and is scalable in a distributed environment.

If there is extra time at the end of the interview, here are a few additional talking points:

- Clock synchronization. In our design, we assume ID generation servers have the same clock. This assumption might not be true when a server is running on multiple cores. The same challenge exists in multi-machine scenarios. Solutions to clock synchronization are out of the scope of this course; however, it is important to understand the problem exists. Network Time Protocol is the most popular solution to this problem. For interested readers, refer to the reference material [4].
- Section length tuning. For example, fewer sequence numbers but more timestamp bits are effective for low concurrency and long-term applications.
- High availability. Since an ID generator is a mission-critical system, it must be highly available.

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

Reference materials

[1] Universally unique identifier:
https://en.wikipedia.org/wiki/Universally_unique_identifier

[2] Ticket Servers: Distributed Unique Primary Keys on the Cheap:
https://code.flickr.net/2010/02/08/ticket-servers-distributed-unique-primary-keys-on-the-cheap/

[3] Announcing Snowflake:
https://blog.twitter.com/engineering/en_us/a/2010/announcing-snowflake.html

[4] Network time protocol:
https://en.wikipedia.org/wiki/Network_Time_Protocol

09
Design A URL Shortener

In this chapter, we will tackle an interesting and classic system design interview question: designing a URL shortening service like tinyurl.

Step 1 - Understand the problem and establish design scope

System design interview questions are intentionally left open-ended. To design a well-crafted system, it is critical to ask clarification questions.

Candidate: Can you give an example of how a URL shortener works?

Interviewer: Assume URL https://www.systemdesignfun.com/articles/system-design?id=abc123 is long. In the original URL, four service clients are able with shorter length: https://tinyurl.com/y7keocwj. If you click the alias, it redirects you to the original URL.

Candidate: What is the traffic volume?

Interviewer: 100 million URLs are generated per day.

Candidate: How long is the shortened URL?

Interviewer: As short as possible.

Candidate: What characters are allowed in the shortened URL?

Interviewer: Shortened URL can be a combination of numbers (0-9) and characters (a-z, A-Z).

Candidate: Can shortened URLs be deleted or updated?

Interviewer: For simplicity, let us assume shortened URLs cannot be deleted or updated.

Here are the basic use cases:

1.URL shortening: given a long URL => return a much shorter URL

2.URL redirecting: given a shorter URL => redirect to the original URL

3.High availability, scalability, and fault tolerance considerations

Back of the envelope estimation

- Write operation: 100 million URLs are generated per day.
- Write operation per second: 100 million / 24 /3600 = 1160
- Read operation: Assuming ratio of read operation to write operation is 10:1, read operation per second: 1160 * 10 = 11,600
- Assuming the URL shortener service will run for 10 years, this means we must support 100 million * 365 * 10 = 365 billion records.
- Assume average URL length is 100.
- Storage requirement over 10 years: 365 billion * 100 bytes = 36.5 TB

It is important for you to walk through the assumptions and calculations with your interviewer so that both of you are on the same page.

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

In this section, we discuss the API endpoints, URL redirecting, and URL shortening flows.

API Endpoints

API endpoints facilitate the communication between clients and servers. We will design the APIs REST-style. If you are unfamiliar with restful API, you can consult external materials, such as the one in the reference material [1]. A URL shortener primary needs two API endpoints.

1.URL shortening. To create a new URL shortener URL, a client sends a POST request, which contains one parameter: the original long URL. The API looks like this:

POST api/v1/data/shorten

- request parameter: {longUrl: longURLString}
- return shortURL

2.URL redirecting. To redirect a short URL to the corresponding long URL, a client sends a GET request. The API looks like this:

GET api/v1/shortURL

- Return longURL for HTTP redirection

URL redirecting

Figure 1 shows what happens when you enter a tinyurl into the browser. Once the server receives a tinyurl request, it changes the short URL to the long URL with 301 redirect.

Figure 1

The detailed communication between clients and servers is shown in Figure 2.

Figure 2

One thing worth discussing here is 301 redirect or 302 redirect.

301 redirect: A 301 redirect shows that the requested URL is "permanently" moved to the long URL. Since it is permanently redirected, the browser caches the response, and subsequent requests for the same URL will not be sent to the URL shortening service. Instead, requests are redirected to the long URL server directly.

302 redirect: A 302 redirect means that the URL is "temporarily" moved to the long URL, meaning that subsequent requests for the same URL will be sent to the URL shortening service first. Then, they are redirected to the long URL server.

Each redirection method has its pros and cons. If the priority is to reduce the server load, using 301 redirect makes sense as only the first request of the same URL is sent to URL shortening servers. However, if analytics is important, 302 redirect is a better choice as it can track click rate and source of the click more easily.

The most intuitive way to implement URL redirecting is to use hash tables. Assuming the hash table stores <shortURL, longURL> pairs, URL redirecting can be implemented by the following:

- Get longURL: longURL = hashTable.getShortURL()
- Once you get the longURL, perform the URL redirect.

URL shortening

Let us assume the short URL looks like this: www.tinyurl.com/{hashValue}. To support the URL shortening use case, we must find a hash function *fx* that maps a long URL to the "hashValue", as shown in Figure 3.

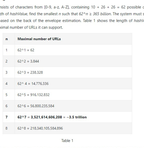

Figure 3

The hash function must satisfy the following requirements:

- Each longURL must be hashed to one hashValue.
- Each hashValue can be mapped back to the longURL.

Detailed design for the hash function is discussed in deep dive.

Step 3 - Design deep dive

Up until now, we have discussed the high-level design of URL shortening and URL redirecting. In this section, we dive deep into the following: data model, hash function, URL shortening and URL redirecting.

Data model

In the high-level design, everything is stored in a hash table. This is a good starting point, however, this approach is not feasible for real-world systems as memory resources are limited and expensive. A better option is to store <shortURL, longURL> mapping in a relational database. Figure 4 shows a simple database table design. The simplified version of the table contains 3 columns: id, shortURL, longURL

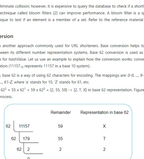

Figure 4

Hash function

Hash function is used to hash a long URL to a short URL, also known as hashValue.

Hash value length

The hashValue consists of characters from [0-9, a-z, A-Z], containing 10 + 26 + 26 = 62 possible characters. To figure out the length of hashValue, find the smallest n such that $62^n \geq 365$ billion. The system must support up to 365 billion URLs based on the back of the envelope estimation. Table 1 shows the length of hashValue and the corresponding maximum number of URLs it can store.

n	Maximum number of URLs
1	$62^1 = 62$
2	$62^2 = 3,844$
3	$62^3 = 238,328$
4	$62^4 = 14,776,336$
5	$62^5 = 916,132,832$
6	$62^6 = 56,800,235,584$
7	**$62^7 = 3,521,614,606,208 = 3.5$ trillion**
8	$62^8 = 218,340,105,584,896$

Table 1

When $n = 7$, $62^7 = \sim 3.5$ trillion, 3.5 trillion is more than enough to hold 365 billion URLs, so the length of hashValue is 7.

We will explore two types of hash functions for a URL shortener. The first one is "hash + collision resolution", and the second one is "base 62 conversion". Let us look at them one by one.

Hash + collision resolution

To shorten a long URL, we should implement a hash function that hashes a long URL to a 7-character string. A straightforward solution is to use well-known hash algorithms like CRC32, MD5, or SHA-1. The following table compares the hash results after applying different hash functions on this URL:
https://en.wikipedia.org/wiki/System_design

Hash function	Hash value (Hexadecimal)
CRC32	5d5d0d54
MD5	5a0250bd8ef9e01235b9d95b6e
SHA-1	2eaee797c6d051f6fc5fa5e6a9b6d7ed85c

Table 2

As shown in Table 2, even the shortest hash value (from CRC32) is too long (more than 7 characters). How can we shorten it?

The first approach is to collect the first 7 characters of a hash value; however, this method can lead to hash collisions. To resolve hash collisions, we can recursively append a new predefined string until no more collision is discovered. This process is explained in Figure 5.

Figure 5

This method can eliminate collision; however, it is expensive to query the database to check if a shortURL exists for every request. A technique called Bloom filters [2] can improve performance. A bloom filter is a space-efficient probabilistic technique to test if an element is a member of a set. Refer to the reference material [3] for more details.

Base 62 conversion

Base conversion is another approach commonly used for URL shorteners. Base conversion helps to convert the same number between its different number representation systems. Base 62 conversion is used as there are 62 possible characters for hashValue. Let us an example to explain how the conversion works: convert 11157 $_{10}$ to base 62 representation 2TX $_{62}$, represents 11157 is a base 10 system.

- From its name, base 62 is a way of using 62 characters for encoding. The mappings are: 0 –> 0, 1 –> 1, ... 9 –> 9, 10 –> a, ... 35 –> z, 36 –> A, ... 61 –> Z where 0 stands for 10, 2 stands for 61, etc.
- $11157_{10} = 2 \times 62^2 + 55 \times 62^1 + 59 \times 62^0 = (2, 55, 59) \rightarrow (2, T, X)$ in base 62 representation. Figure 6 shows the conversion process.

Figure 6

- Thus, the short URL is https://tinyurl.com/**2TX**.

Comparison of the two approaches

Table 3 shows the differences of the two approaches.

Hash + collision resolution	Base 62 conversion
Fixed short URL length.	Short URL length is not fixed. It goes up with the ID.
Does not need a unique ID generator.	This option depends on a unique ID generator.
Collision is possible and needs to be resolved.	Collision is not possible because ID is unique.
It's not possible to figure out the next available short URL because it doesn't depend on ID.	It is easy to figure out what is the next available short URL if ID increments by 1 for a new entry. This can be a security concern.

Table 3

URL shortening deep dive

As one of the core pieces of the system, we want the URL shortening flow to be logically simple and functional. Base 62 conversion is used in our design. We build the following diagram (Figure 7) to demonstrate the flow.

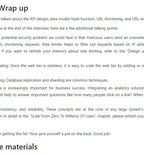

Figure 7

1. longURL is the input.
2. The system checks if the longURL is in the database.
3., 4., 5., 6. means the longURL was converted to shortURL before. In this case, fetch the shortURL from the database and return it to the client.
4. If not, the longURL is new. A unique ID (primary key) is generated by the unique ID generator.
5. Convert the ID to shortURL with base 62 conversion.
6. Create a new database row with the ID, shortURL, and longURL.
7. To make the flow easier to understand, we omit the following use cases:
- Assuming the input longURL is https://en.wikipedia.org/wiki/System_design
- Unique ID generator returns ID: 2009215674938.
- Convert the ID to shortURL using the base 62 conversion: ID (2009215674938) is converted to "zOe9ucu".
- Save ID, shortURL, and longURL to the database as shown in Table 4.

id	shortURL	longURL
2009215674938	zOe9ucu	https://en.wikipedia.org/wiki/System_design

Table 4

The distributed unique ID generator is worth mentioning. Its primary function is to generate globally unique IDs, which are used for creating shortURLs. In a highly distributed environment, implementing a unique ID generator is challenging. Luckily, we have already discussed a few solutions in the "Design A Unique ID Generator in Distributed System" chapter. You can refer back to it to refresh your memory.

URL redirecting deep dive

Figure 8 shows the detailed design of the URL redirecting. As there are more reads than writes, <shortURL, longURL> mapping is stored in a cache to improve performance.

Figure 8

The flow of URL redirecting is explained as follows:

1.A user clicks a short URL link: https://tinyurl.com/zOe9ucu

2. The load balancer forwards the request to a web servers.

3.If a shortURL is already in the cache, return the longURL directly.

4.If a shortURL is not in the cache, fetch the longURL from the database. If it is not in the database, it is likely a user entered an invalid shortURL.

5. The longURL is returned to the user.

Step 4 - Wrap up

In this chapter, we talked about the API design, data model, hash function, URL shortening, and URL redirecting.

If there is extra time at the end of the interview, here are a few additional talking points.

- Rate limiter: A potential security problem we could face is that malicious users send an overwhelmingly large number of URL shortening requests. Rate limiter helps to filter out requests based on IP address or other filtering rules. If you want to refresh your memory about rate limiting, refer to the "Design a rate limiter" chapter.
- Web server scaling: Since the web tier is stateless, it is easy to scale the web tier by adding or removing web servers.
- Database scaling: Database replication and sharding are common techniques.
- Analytics: Data is increasingly important for business success. Integrating an analytics solution to the URL shortener could help to answer important questions like how many people click on a link? When do they click the link? etc.
- Availability, consistency, and reliability: These concepts are at the core of any large system's success. We discussed them in detail in the "Scale From Zero To Millions Of Users" chapter, please refresh your memory on these topics.

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

Reference materials

[1] A RESTful Tutorial https://www.restapitutorial.com/index.html

[2] Bloom filter https://en.wikipedia.org/wiki/Bloom_filter

10
Design a Web Crawler

Author: **Yu Zhou**, Senior Software Engineer
Lead: **Yu Zhou**, Senior Software Engineer
Date: **Yu Zhou**, Senior Software Engineer

In this chapter, we focus on web crawler design as interesting and classic system design interview question.

A web crawler is known as a bot, a spider, or a web robot. It is a software that systematically browses various pages of a website in a methodical manner and collects specified data from these pages. In this chapter, we will create a web crawler that collects pages of English articles and stores them for later use. We don't need to worry about how to collect the pages, or what to do with the collected pages.

Figure 1

A crawler is used for many purposes:

1. Search engine indexing: This is the most common use case. A crawler collects web pages to create a local index for a search engine. For example, Googlebot is Google's web crawler. It crawls the web to build its search engine's index.
2. Web archiving: Libraries and organizations can use crawlers to archive static versions of websites for long-term preservation. Consider legal and cultural reasons for archival sites. Notable examples are the US Library of Congress and the European Union Web Archive.
3. Web data mining: Companies can use crawlers to collect data from the internet for analysis. For example, they can analyze trends, monitor sentiment, or gather competitive intelligence. For example, top financial firms use crawlers to scrape financial data from websites to inform their investment strategies and decisions.
4. Web monitoring: Crawlers can be used to monitor copyright and trademark infringements over the internet. For example, Digimarc utilizes crawlers to discover pirated works and report it.

Step 1 - Understand the problem and establish design scope

Crawler is used for many purposes, so we need to define the scope of our crawler.

Interviewer asks questions to clarify requirements and assumptions.

Candidate: Let's clarify the requirements. I'm not familiar with how search engines index data, but crawling is a simpler alternative. Please correct me if my assumptions are wrong.

Interviewer: Sounds good.

Candidate: Can I assume this is a text-only crawler? Multimedia files, such as images and videos, are also popular. Is this correct?

Interviewer: Yes, you're correct.

Candidate: How many pages should we crawl?

Interviewer: For this exercise, let's say 1 billion pages.

Candidate: What about the quality of the pages? Should we crawl pages with low quality or spam content?

Interviewer: Yes, any content is fair game.

Candidate: What about HTTP, HTTPS, and FTP? Do we need to support all these protocols?

Interviewer: Yes, you're correct.

Candidate: How should we handle URLs that are accessible via different protocols? For example, if a page is accessible via both HTTP and HTTPS, should we treat them as two separate pages?

Interviewer: No, you're correct.

Candidate: How long should we keep the crawler running? Should we store all the data indefinitely?

Interviewer: For this exercise, we don't need to consider long-term storage. A few days should be sufficient.

Candidate: What about handling dynamic content, such as JavaScript-rendered pages? Should we crawl these pages?

Interviewer: No, let's stick to static HTML pages for now.

Candidate: What about rate limiting? Should we be polite and respect robots.txt?

Interviewer: Yes, that's important. Please assume we're building a polite crawler.

Candidate: What about error handling? How should we handle HTTP errors, timeouts, and other failures?

Interviewer: Assume we can simply log errors and skip the problematic pages.

Candidate: Thank you. I'll proceed with the design based on these requirements.

Interviewer: Sounds good.

Figure 2

Big or small, web crawler should be scalable, robust, and extensible. Other requirements are also important, but less critical, depending on your specific use case. For example, if you are asked to design a crystal-focused project like a web crawler, error handling is less critical. We can keep it simple and skip complex error handling.

Here is a list of functionalities we plan to implement for this web crawler:

- Given a set of URLs, download all the web pages associated with those URLs.
- Extract URLs from these web pages.
- Add new URLs to the list of URLs to be downloaded.
- Repeat the above three steps until there is no more URL to download, or we have downloaded 1 billion pages.

Next, we make back of the envelope estimation to understand their functionalities. There, we examine the crawler from performance and scalability perspectives.

Scalability

Our web crawler needs to be able to scale to perform the crawl process. For example, to crawl all web pages from a website, one crawler process may be sufficient. For more complex tasks, multiple crawler processes may be needed.

To crawl the entire web, we need the crawler scalable to extracting data URLs of special web URL servers as a special handling. Consider using a message queue to decouple URL extraction from the crawling process. This allows for independent scaling of URL extraction and crawling. Consider implementing a distributed crawler system if needed to handle the scale of the web. For example, a single server might handle 100 crawler processes, while a large-scale crawler system might need dozens or hundreds of such servers to distribute the workload. Consider using cloud-based infrastructure or a distributed system to manage the crawler processes.

Performance

Web crawler's performance is crucial for efficiently gathering data. Consider factors like the number of concurrent connections, network latency, and the time taken to process each page. Optimize the crawler to maximize data retrieval while minimizing resource consumption and avoiding overloading the target websites.

URL Frontier

The URL frontier is a data structure that stores URLs to be downloaded and already downloaded URLs. The URL frontier is a data structure that stores URLs to be downloaded and also remembers which URLs have been crawled.

Since web crawl tasks are random access in nature, the URL frontier should support random access to any URL. The URL frontier should also support adding new URLs to the queue. Finally, the URL frontier should be persistent. Disk storage is used to ensure data persistence even if the crawler restarts.

URL Extractor

URL extractor parses and extracts URLs from HTML pages. Given a URL, the URL extractor downloads the page associated with the URL and extracts URLs from the HTML page.

Content Retriever

Content retriever downloads web pages from the URL extractor. For example, to crawl a page associated with the URL https://www.wikipedia.org/, the content retriever downloads the HTML page associated with https://www.wikipedia.org/.

Content Storage

Content storage is used to store HTML and other media content. Most of the content should be persistent. Disk storage is used for most of the content. Popular frequently used content may be cached in memory for faster retrieval.

Figure 3

Step 1 - 3 are given URLs to the URL Frontier.

Step 4 - 6 URL Extractor retrieves URLs from the URL Frontier.

Step 7 - 9 Content Retriever retrieves web pages from the URL extractor and stores it in Content Storage.

Step 10 - 14 Content is parsed and validated if it is passed to the "Content Seen?" component.

Step 15 - 19 Content is passed to the URL extractor if URL page is stored in the Storage.

Step 20 - 24 If it is the first time, the same process as a different URL has always been processed in the crawl process, the URL is added to the URL Frontier.

Step 25 - 27 If it is not the first time, the URL is processed before the same process occurred. The crawler is aware of the URL and it is not added to the URL Frontier.

Step 28 - 30 Content extractor extracts URLs from HTML pages.

Step 31 - 33 Filter URL filters URLs that are not of interest.

Step 34 - 36 Filter URL filters URLs that are known to be "URL Seen" component.

Step 37 - 39 URL Seen? component checks if a URL is seen before or if it is a processed before, and nothing needs to be done with it.

Step 40 - 42 If URL is not seen before, it is added to the URL Frontier.

Step 3 - Design deep dive

In just take-by-hand discussion the high-level design, there are will discuss the most important buildings components and technologies in detail.

- Depth-first search (DFS) vs Breadth-first search (BFS)
- URL frontier
- HTML fetcher
- Anti-spam
- Robustness
- Extensibility
- Detect and avoid problematic content

DFS vs BFS

Web crawl is modeled as a directed graph where web pages serve as nodes and hyperlinks (URLs) as edges. The crawl process can be viewed as traversing this graph, starting from a set of seed URLs. Two common algorithms for traversing directed graphs are DFS and BFS. However, DFS is usually not a good choice because the depth of DFS can be very deep.

The breadth of BFS is usually limited by the number of hyperlinks on a page, which is typically small. While BFS is often recommended for web crawl, it has two issues. First, it does not prioritize URLs based on their importance. Second, the implementation has two problems:

- Most links from the same web page are to the same server. This causes the crawler to send many parallel requests to the same server, which can be interpreted as a denial-of-service attack by the server. For example, if we download 1000 pages from www.facebook.com, it would be polite to space out these requests over a period. Otherwise we risk an overload of resources. Impolite crawling is prohibited.

Figure 4

- Standard DFS crawler will likely be assigned to a URL on a local server. The crawler will spend a lot of time crawling this server because it's recursively crawling all URLs on this server. The majority of URLs will be on the same server, so the URL frontier will be full of URLs from this server. This means other servers will be neglected.

URL Frontier

URL frontier is a data structure that stores URLs to be downloaded. It ensures that only one URL is downloaded from the URL frontier at an implemented concurrently to ensure politeness. URL prioritization, and Robustness. It also keeps track of URLs that have been downloaded to avoid crawling the same URL again. The frontier is a data structure that supports adding URLs and retrieving URLs for the crawl process.

Politeness

Crawling a set of URLs from a single server can be viewed as a series of HTTP requests, each requesting a URL on that server. Polite crawling means making these HTTP requests over time, with gaps between each request. This avoids overwhelming the server and thousands of requests every second to the same website.

We can implement politeness by maintaining a mapping from URL to download thread. Each download thread is a separate thread of execution that downloads URLs from a single website. For example, if we download 1 million pages from www.facebook.com, we might assign a single download thread to handle all requests to www.facebook.com. This ensures that all requests to www.facebook.com are made sequentially, with a delay between each request.

Figure 5

- Queue router: It ensures that each queue (URL) only contains URLs from the same host.
- Queue selector: Maps each host to a queue.
- Worker: Each worker thread is assigned to a single host.

Host	Queue
wikipedia.org	1
apple.com	2
...	...

Figure 6

- Queue 1 contains URLs from wikipedia.org.
- Queue 2 contains URLs from apple.com.
- Queue n contains URLs from www.facebook.com.

Figure 7 is a high-level diagram of various components we've talked about so far. The URL frontier is a queue, and URLs are dequeued from queue router. The queue selector assigns a queue to a download thread, and each worker thread is bound to a specific queue, fetching URLs from its assigned queue. The URL frontier's design ensures that only one worker thread accesses a specific queue at a time, effectively managing politeness by preventing overwhelming any single website with too many requests from multiple threads.

Figure 7

Priority

We can prioritize URLs based on their importance, such as their PageRank, website traffic, or other factors. The URL frontier can be implemented using a priority queue to prioritize URLs for crawling. The URL frontier can also be implemented using multiple queues, each representing a different priority level. The queue selector can then prioritize queues based on their priority when assigning them to worker threads.

Freshness

Web pages are constantly being added, deleted, and edited. A web crawler must periodically recrawl downloaded pages to maintain data freshness. The URL frontier can be used to prioritize URLs for recrawling. The crawler can also use techniques like sitemaps and RSS feeds to detect changes.

Storage for URL Frontier

The URL frontier is a data structure that stores the number of URLs in the frontier could be hundreds of millions. Putting everything in memory is neither durable nor scalable. Keeping everything on the disk is undesirable either because of slow read and write operations. Our compromise is a hybrid approach that combines memory and disk storage.

We maintain most URLs in disk queues, but only a few queues are in memory for quick access. This ensures that a crawl thread can quickly retrieve the URL data and crawling. On the disk, we organize URLs by priority and host, with separate queues for each combination. This allows for efficient management and prioritization of URLs from different hosts.

Figure 8

- Queue router: It ensures that each queue (URL) only contains URLs from the same host.
- Queue selector: Maps each host to a queue.
- Disk queues: Contains URLs stored on disk.
- Memory queues: Contains URLs kept in memory for quick access.
- Worker: Each worker thread is assigned to a single host.

Figure 9 presents the URL frontier design, and it contains two modules:

- Front queues: manage prioritization.
- Back queues: manage politeness.

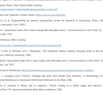

Figure 9

- Prioritizer: It takes URL as input and computes the priority.
- Front queue 1 and 2 queues have an assigned priority. Queues with high priority are selected with higher probability.
- Queue router: Routes URLs to queues with a mapping from queue to host name.

Figure 10 presents the URL frontier design, and it contains two modules:

- Front queues: manage prioritization.
- Back queues: manage politeness.

Step 4 - Wrap up

In this chapter, we first discussed the characteristics of a good crawler: scalability, politeness, extensibility, and robustness. Then, we proposed a design and discussed the implementation. Building a scalable web crawler is a time-consuming and resource-intensive endeavor. Many challenges remain to be addressed even after a basic system is built. Here are a few other talking points, if you are asked to discuss more details:

- Server-side rendering: Numerous websites employ JavaScript to dynamically generate HTML pages on the server-side. This poses a challenge for our crawler, as it must parse HTML pages to extract URLs for crawling. To address this, we can perform server-side rendering before crawling. This approach allows the crawler to access the fully rendered HTML content, including dynamically generated URLs.
- Cloud crawling: Leveraging cloud-based services or infrastructure can simplify the crawler's deployment and management. Cloud platforms offer scalable resources and can handle the infrastructure and maintenance tasks, allowing you to focus on the core functionality of the crawler.
- Machine learning: Incorporating machine learning techniques can enhance the crawler's intelligence and efficiency. For example, you can use machine learning to prioritize URLs based on their relevance and importance, or to detect and avoid spam or low-quality content.
- Content filtering: Implementing content filtering mechanisms can help you exclude certain types of content from being crawled, such as adult content, or content from specific domains or websites.
- Distributed crawling: For large-scale crawling, you can distribute the crawling task across multiple servers or cloud instances to parallelize the process and increase the overall crawling speed.
- Monitoring and analytics: Implementing monitoring and analytics tools can provide valuable insights into the crawler's performance, error rates, and other relevant metrics. This data can be used to optimize the crawler's performance and-identify-potential-issues-or-bottlenecks-in-the-crawling-process.

Reference materials

[1] Wikipedia: https://en.wikipedia.org/wiki/Web_crawler

[2] Scrapy: https://scrapy.org/

[3] Apache Nutch: https://nutch.apache.org/

[4] Crawlee: https://crawlee.dev/

[5] Colly: https://go-colly.org/

[6] Puppeteer: https://pptr.dev/

[7] Selenium: https://www.selenium.dev/

[8] Playwright: https://playwright.dev/

[9] Web Archive Standards: https://www.loc.gov/preservation/digital/formats/fdd/browse_list_archive.shtml

[10] Robots Exclusion Protocol: https://developers.google.com/search/docs/crawling-indexing/robots/intro

[11] Sitemaps: https://www.sitemaps.org/protocol.html

[12] Cloudflare: https://www.cloudflare.com/learning/bots/what-is-a-web-crawler/

[13] Apache Kafka: https://kafka.apache.org/documentation/

[14] Redis Streams: https://redis.io/docs/latest/develop/data-types/streams/

[15] Kleinberg, J. (1999). Authoritative sources in a hyperlinked environment. Proceedings of the ACM-SIAM symposium on Discrete algorithms. https://www.cs.cornell.edu/home/kleinberg/auth.pdf

[16] Page, L., Brin, S., Motwani, R., Winograd, T. The PageRank citation ranking: Bringing order to the web. Stanford InfoLab. http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf

[17] Barbosa, Berlin, Saxena, Gupta, Cho. (2010). Towards a refined vision of crawling. In Proceedings of the 19th International World Wide Web Conference. (WWW '10). ACM. https://www.ccs.neu.edu/home/vip/teach/DCS8350-F15-Project1-F-Paper-Crawlers.pdf

[18] Eda, Cho, Shi. (2011). Characterizing and supporting cross-lingual vocabulary growth. In Proceedings of the 19th International World Wide Web Conference. (WWW '11). ACM. https://www.cs.ucla.edu/~cho/papers/eda-www11.pdf

[19] Heydon, Allan. (1999). Mercator: A scalable, extensible web crawler. Compaq Systems Research Center. https://www.cs.cornell.edu/courses/cs685/2002fall/mercator.pdf

[20] Lee, Jeong, Cho, Grama, Jiang. (2009). Parallel Web Crawlers: Performance and Communication Issues. In Proceedings of the 18th International World Wide Web Conference. (WWW '09). ACM. https://www.cs.cornell.edu/courses/cs685/2002fall/parallelcrawlers.pdf

[21] Boldi, Marino, Santini, Vigna. (2009). BUBINGA: Towards an open-source scalable distributed web crawler. In Proceedings of the 18th International World Wide Web Conference. (WWW '09). ACM. https://www.cs.cornell.edu/courses/cs685/2002fall/bubinga.pdf

[22] Shkapenyuk, Suel. (2002). Design and implementation of a high-performance distributed web crawler. In Proceedings of the 18th International World Wide Web Conference. (WWW '02). ACM. https://www.cs.cornell.edu/courses/cs685/2002fall/shkapenyuk.pdf

[23] Zhuang, Mislove, Gummadi, Druschel. (2007). Bandwidth management in distributed crawling. In Proceedings of the 16th International World Wide Web Conference. (WWW '07). ACM. https://www.cs.cornell.edu/courses/cs685/2002fall/bandwidth.pdf

11
Design A Notification System

A notification system has always become a very popular feature for many applications in recent years. A notification can be pushed to iOS devices, Android devices, and even to a person's phone through SMS messages. It has become an indispensable part of our lives. In this chapter, you are asked to design a notification system.

A notification is more than just mobile push notification. These types of notification formats are mobile push notifications, SMS messages and Email. Figure 1 shows the different notification types.

Email
SMS
Mobile push
Notification Center
1243
Notification Center
Yesterday
Mark all as read
Clear all
Gothamis
Wed, 8:00 AM
Gotham Knights
Wed, 8:00 AM
It's 40% more than the "Arkham Knight" collection!
It's 40% more than the "Arkham Knight" collection!

12

Design A News Feed System

In this chapter, you are asked to design a news feed system. What it mean feed? According to the Facebook help page, "News feed is the constantly updating list of stories in the middle of your home page. News Feed includes status updates, photos, videos, links, app activity, and likes from people, pages, and groups that you follow on Facebook" [1]. To a popular interview question. Similar questions commonly asked are to design Facebook news feed, Instagram feed, Twitter timeline, etc.

Figure 1

Step 1 - Understand the problem and establish design scope

The first set of clarification questions are to understand what the interviewer has in mind when she asks you to design a news feed system. At the very least, you should figure out what features to support. Here is an example of candidate-interviewer interaction:

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

Candidate: What are the important features?
Interviewer: A user can publish a post and see her friends' posts on the news feed page.

Candidate: A user sees feed sorted by reverse chronological order or any particular order such as topic scores? For interest, posts from your close friends have higher scores.
Interviewer: To keep things simple, let us assume the feed is sorted by reverse chronological order.

Candidate: How many friends can a user have?
Interviewer: 5000

Candidate: What is the traffic volume?
Interviewer: 10 million DAU

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

Now you have gathered the requirements, we focus on designing the system.

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

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 and database. A post is populated to her friends' news feed.
- Newsfeed building: for simplicity, let us assume the news feed is built by aggregating friends' posts in reverse chronological order.

Newsfeed APIs

The news feed APIs are the primary ways for clients to communicate with servers. Those APIs are HTTP based that allow clients to perform actions, which include posting a status, retrieving news feed, adding friends, etc. We discuss two most important APIs: feed publishing API and news feed retrieval API.

Feed publishing API

To publish a post, a HTTP POST request will be sent to the server. The API is shown below:

POST /v1/me/feed

Params:

- content: content is the text of the post.
- auth_token: it is used to authenticate API requests.

Newsfeed retrieval API

The API to retrieve news feed is shown below:

GET /v1/me/feed

Params:

- auth_token: it is used to authenticate API requests.

Feed publishing

Figure 2 shows the high-level design of the feed publishing flow.

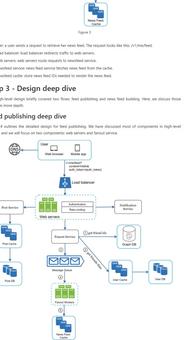

Figure 2

- User: a user can view news feeds on a browser or mobile app. A user makes a post with content "Hello" through API:
  /v1/me/feed?content=Hello&auth_token=auth_token
- Load balancer: distribute traffic to web servers.
- Web servers: web servers redirect traffic to different internal services.
- Post service: persist post in the database and cache.
- Fanout service: push new content to friends' news feed. Newsfeed data is stored in the cache for fast retrieval.
- Notification service: inform friends that new content is available and send out push notifications.

Newsfeed building

In this section, we discuss how news feed is built behind the scenes. Figure 3 shows the high-level design:

Figure 3

- User: a user sends a request to retrieve her news feed. The request looks like this: /v1/me/feed.
- Load balancer: load balancer redirects traffic to web servers.
- Web servers: web servers route requests to newsfeed service.
- Newsfeed service: news feed service fetches news feed from the cache.
- Newsfeed cache: store news feed IDs needed to render the news feed.

Step 3 - Design deep dive

The high-level design briefly covered two flows: feed publishing and news feed building. Here, we discuss those topics in more depth.

Feed publishing deep dive

Figure 4 outlines the detailed design for feed publishing. We have discussed most of components in high-level design, and we will focus on two components: web servers and fanout service.

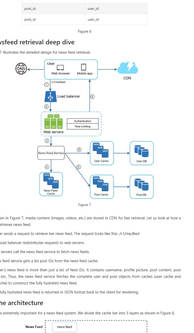

Figure 4

Web servers

Besides communicating with clients, web servers enforce authentication and rate limiting. Only users signed in with valid auth_token are allowed to make posts. The system limits the number of posts a user can make within a certain period, vital to prevent spam and abuse content.

Fanout service

Fanout is the process of delivering a post to all friends. Two types of fanout models are fanout on write (also called push model) and fanout on read (also called pull model). Both models have pros and cons. We explain their workflow and explore the best approach to support our system.

Fanout on write. With this approach, news feed is pre-computed during write time. A new post is delivered to friends' cache immediately after it is published.

Pros:

- If a user has many friends, fetching the friend list and generating news feeds for all of them are slow and time consuming. It is called hotkey problem.
- For inactive users or those rarely log in, pre-computing news feeds waste computing resources.

Fanout on read. The news feed is generated during read time. This is an on-demand model. Recent posts are pulled when a user feeds her newsfeed.

Pros:

- For inactive users or those who rarely log in, fanout on read works better because it will not waste computing resources on them.
- Data is not pushed to friends so there is no hotkey problem.

Cons:

- Fetching the news feed is slow as the news feed is not pre-computed.

We adopt a hybrid approach to get benefits of both approaches and avoid pitfalls in them. Since fetching the news feed fast is crucial, we use a push model for the majority of users. For celebrities or users who have many friends/followers, we let followers pull news content on-demand to avoid system overload. Consistent hashing is a useful technique to mitigate the hotkey problem as it helps to distribute requests/data more evenly.

Let us take a close look at the fanout service as shown in Figure 5.

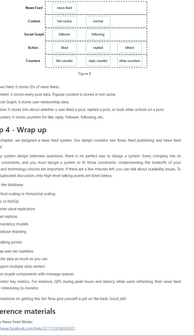

Figure 5

The fanout service works as follows:

1. Fetch friend IDs from the graph database. Graph databases are suited for managing friend relationship and friend recommendations. Interested readers wishing to learn more about this concept should refer to the reference material [2].

2. Get friends info from the user cache. The system then filters out friends based on user settings. For example, if you mute someone, her posts will not show up on your news feed even though you are still friends. Another reason why posts may not show is that a user could selectively share information with specific friends or hide it from other people.

3. Send friends list and new post ID to the message queue.

4. Fanout workers fetch data from the message queue and store news feed data in the news feed cache. You can think of the news feed cache as a <*user_id*, *set-ype*> mapping table. Whenever a new post is made, it will be appended to the news feed table as shown in Figure 6. The memory consumption can become very large if we keep all the entire history of news feed data. Fortunately, only the most recent news feeds are stored. To keep the memory size small, we set a configuration limit. The chance of a user scrolling through thousands of posts in news feed is slim. Most users are only interested in the latest content, so the cache miss rate is low.

5. Store <*post_id*, *user_id*> in news feed cache. Figure 6 shows an example of what the news feed looks like in cache.

post_id	user_id
post_id	user_id
post_id	user_id
post_id	user_id
post_id	user_id
post_id	user_id
post_id	user_id
post_id	user_id

Figure 6

Newsfeed retrieval deep dive

Figure 7 illustrates the detailed design for news feed retrieval.

Figure 7

As shown in Figure 7, media content (images, videos, etc.) are stored in CDN for fast retrieval. Let us look at how a client retrieves news feed.

1. A user sends a request to retrieve her news feed. The request looks like this: /v1/me/feed

2. The load balancer redistributes requests to web servers.

3. Web servers call the news feed service to fetch news feeds.

4. News feed service gets a list post IDs from the news feed cache.

5. A user's news feed is more than just a list of feed IDs. It contains username, profile picture, post content, post image, etc. Thus, the news feed service fetches the complete user and post objects from caches (user cache and post cache) to construct the fully hydrated news feed.

6. The fully hydrated news feed is returned in JSON format back to the client for rendering.

Cache architecture

Cache is extremely important for a news feed system. We divide the cache tier into 5 layers as shown in Figure 8.

Figure 8

- News Feed: it stores IDs of news feeds.
- Content: it stores every post data. Popular content is stored in hot cache.
- Social Graph: it stores user relationship data.
- Action: it stores info about whether a user liked a post, replied a post, or took other actions on a post.
- Counters: it stores counters for like, reply, follower, following, etc.

Step 4 - Wrap up

In this chapter, we designed a news feed system. Our design contains two flows: feed publishing and news feed retrieval.

Like any system design interview questions, there is no perfect way to design a system. Every company has its unique constraints, and you must design a system to fit those constraints. Understanding the trade-offs of your design and technology choices are important. If there are a few minutes left, you can talk about scalability issues. To avoid duplicate discussion, only high-level talking points are listed below.

Scaling the database:

- Vertical scaling vs Horizontal scaling
- SQL vs NoSQL
- Master-slave replication
- Read replicas
- Consistency models
- Database sharding

Other talking points:

- Keep web tier stateless
- Cache data as much as you can
- Support multiple data centers
- Loose coupple components with message queues
- Monitor key metrics. For instance, QPS during peak hours and latency while users refreshing their news feed are interesting to monitor.

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

Reference materials

[1] How News Feed Works:
https://www.facebook.com/help/327131014036297/

[2] Friend of Friend recommendations Neo4j and SQL Server:
http://mattsalmon.com/2014/03/03/friend-of-friend-recommendations-with-neo4j-and-sql-server/

13

Design A Chat System

In this chapter, we explore the design of a chat system. Which chat system should we build? Well, that's a broad question. Chat means different things to different companies.

It can go: personal vs. enterprise, business to consumer (B2C) or business to business (B2B), text only or rich media, group vs. one-to-one, or some other combination. Which one should we focus on? We should ask our friends and colleagues what they need in a chat system. Based on the feedback, we decide to focus on a B2B chat system for enterprise customers. Some of our enterprise customers are located in Asia and they want to reach out to their customers in China. We also learn that mobile is important to our customers, and that small and medium size businesses are our main target audience.

In this chapter, we design a B2B chat system that supports one-to-one and group chat, online presence, and mobile push notifications. We also discuss how to scale the system to support 1 million concurrent users, and how to extend the system to support other features such as photo sharing.

Step 1 - Understand the problem and establish design scope

It is not a good idea for the sake of it to rush into a design. As Confucius said, "Real knowledge is to know the extent of one's ignorance." Many things need to be considered before we can start a design. For example, we need to know what features are important for a chat system. Is it one-to-one chat or group chat? How do users make friends? How do we store user data? How many users are there? We may also need to know how many users we expect in the future, how the system should be scaled to meet business growth, and other requirements such as third-party integration, etc.

In short, a lot needed. At the very least, we need to understand the requirements and agree on the assumptions before we start our design. Here are some of the questions we may ask our product manager to clarify the requirements:

```
Candidate: What kind of chat system do we need? Is it a web app or mobile app? Or both?
Interviewer: It should support both 1-1 and group chat.
Candidate: What is the scale we are looking at?
Interviewer: Scale is important. The system should support 1 million concurrent users.
Candidate: How do people make friends in the chat system?
Interviewer: For simplicity, we can assume all users are in a single group. In reality, a user may have a
variable number of friends, and we may need to think about how that affects the design.
Candidate: What kind of features do you want? For example, do you care about features such as
attachments, group chat, online presence, etc.
Interviewer: For simplicity, let's assume it is a text only chat. It should support group chat and
online presence.
Candidate: Chat features are important for the chat app. Do you want to support attachments?
Interviewer: No, that is not a requirement for us at this time.
Candidate: How long should we store the chat history? Do we need to support data synchronization
across multiple devices?
Interviewer: We don't require data to be stored. We can assume it is a text only chat, and when a user
logs back into the app, he/she will be able to see the chat history they have not seen before.
Candidate: What is the expected traffic?
Interviewer: Our DAU is 50 million.
Candidate: Do we need to support push notifications?
Interviewer: Yes, we need to support push notifications when there is a new message.
```

Functional requirements are the foundation of what we build for a system. Different companies may have different requirements for a chat system. Some of us may want to build a chat system like Facebook Messenger, and others may want to build a chat system like Slack. We need to ask questions to clarify the requirements and make assumptions when the requirements are not clear. For example, we know we need to support one-to-one chat, but we don't know how the system should handle group chat. We need to make an assumption in this case, because it is a common feature for chat applications. Some other features we need to consider are: attachments, group chat, online presence, etc. Chat is a very broad topic, and we can easily spend all our time on it. It is important to agree on what we will focus on when we start our design.

Non-functional requirements are also important for us to understand before we start our design. For example, we need to know how many users we expect to use our system, how we should scale the system to meet business goals, and what latency we should aim for. Some of our non-functional requirements are:

```
Interviewer: Some of our enterprise customers are located in Asia and they want to reach out to their
customers in China. We also learn that mobile is important to our customers, and that small and
medium size businesses are our main target audience.
Interviewer: Latency should be as low as possible on the same continent.
Interviewer: Security is important. We should support at least 1-1 encryption.
Interviewer: Our company is growing fast. We should be able to easily add new features in the future.
Interviewer: Our company is growing fast. We should be able to easily add new features in the future.
Interviewer: Our company is growing fast. We should be able to easily add new features in the future.
```

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

In a chat system, clients can be either mobile or web applications. They can also be desktop applications. For simplicity, let's assume we only need to support web applications. Clients can communicate directly with each other, or they may communicate through a chat server. The chat server is responsible for many things. For example, it should support multiple client connections, route messages to the appropriate recipient, and notify users when someone is online or offline.

The chat system supports two types of data: messages and presence. The chat system should be able to send and receive messages, and it should be able to notify users when their friends are online, offline, or last seen.

Polling

Polling is a technique that lets a client periodically ask the server if there are messages. Depending on how frequently polling happens, it may cost us much in terms of server resources. Imagine a chat system where we need to poll the server for a new message every 1 second. It is a lot of wasted resources.

Client
Time elapsed
Client sends request
Server responds
Client sends request
Server responds
Client sends request
Server responds

14
Design A Search Autocomplete System

When searching for Google or shopping at Amazon, we can type in the search box, and a list of terms for the search will be presented to us. The feature is called search autocomplete, also known as type-ahead, search-as-you-type, or search-as-you-type. In this chapter, we design a search autocomplete system for Google or Amazon. Search autocomplete is an important feature of many products. The basic idea is that when we type in a search term, we will see a list of matching terms. Figure 14.1 shows an example.

Figure 1

Step 1 - Understand the problem and establish design scope

The first step is to ask questions to clarify requirements or to ask engaging questions to verify dependencies and assumptions.

Candidate: Is the matching process based on prefix or anywhere in the text?
Interviewer: It's only for beginning of a search query.
Candidate: How many autocomplete suggestions should the system return?
Interviewer: 5
Candidate: Should the system rank the system-wide popular suggestions?
Interviewer: Yes, popularity is a factor.
Candidate: Should the system support spell check?
Interviewer: No, spell check is beyond the scope.
Candidate: Do we need to support capitalization?
Interviewer: No, we can assume all the queries are lowercase alphanumeric characters.
Candidate: Do all other capitalization and special characters?
Interviewer: No, we can assume all other characters are invalid or special characters.
Candidate: How long should the system run?
Interviewer: It runs 24hr.
Candidate: Scale?
Interviewer: 10 million DAU.

Candidate: Requirements:
- The system should return 5 autocomplete suggestions based on the search query.
- The suggestions are a list of query strings that are autocomplete suggestions much about my full strength.
- The suggestions are based on the search query. It matches the search query when it matches the prefix of the search query. It's only for beginning of a search query.
- Relevant. Autocomplete suggestions should be relevant to the search term.
- Sorted. The system should return popular suggestions first. Popularity is a factor for ranking recently.
- Scalable. The system can handle high traffic volume.
- Highly available. The system should remain available and accessible even part of the system is offline, slow or encounter unexpected network errors.

Back of the envelope estimation

Candidate: Back of the envelope calculation:
- Assume 10 million DAU.
- Average person performs 10 search queries per day.
- 20 bytes of ASCII per query string.
- When a user enters a character, an API call is made. 1 character = 1 byte.
- Average a query contains 4 words, and each word contains 5 characters on average.
- That is 20 x 5 = 20 and 20 bytes.
- When a user enters a character, the system has to collect data in support of the handle for autocomplete suggestions. Assume 20% of the queries are new queries. Assume the system runs for 24hr and the data collection starts from 00:00 and ends at 24:00.

Daily Traffic	QPS: 10M * 10 / 24 / 3600 = 1157	Peak QPS: 2 * QPS = 2314	QWPD: 10M * 10 * 20 / 1000000 = 2000	Peak QWPD: 2 * QWPD = 4000

Table 1

Query service

Second and third frequency tables are known as Table 1 in top two fields.

- Frequency is the number of times a query has been searched.

Query	Frequency
tw	35
twitch	29
tyler perry	17
twitter	15
tyler the creator	12

Table 2

To get top 5 frequently searched queries, execute the following command:

```
select * FROM frequency table
WHERE query LIKE 'prefix%'
ORDER BY frequency DESC
LIMIT 5
```

For a prefix of length n, there are n table scans. This approach is slow when there are many rows in the table.

Step 2 - Design deep dive

In the high-level design, we discuss the following: data gathering, service and query, analytics. The high-level design is not optimal but it serves as a good starting point. In this section, we will deep dive into a few components and explore different options.

- Trie data structure
- Data gathering service
- Query service
- Scale the storage
- Analytics

Trie data structure

Relational database are used for storage in the high-level design. However, finding the top 5 matched prefixes is inefficient and slow. A more suitable data structure is needed to store the data. A commonly used data structure for prefix matching is a data structure called trie (pronounced as "try"). The idea of a trie is discussed in the following.

Understanding the trie data structure is essential for the interview preparation. Furthermore, trie is a data structure that is used in many real-world problems. It is also a popular question in coding interviews. A trie (also known as a prefix tree) is a tree-like data structure that stores a dynamic set of strings. Typically, it is used to store a set of words for fast retrieval. Trie can be used for storing a dictionary for English words.

Figure 2 shows a trie. In a trie, the root stands for an empty string. Each node stores a character and has up to 26 children, one for each character on the alphabet (assume only alphabet characters are allowed). To find all words that begin with "to", we start from the root, first node we encounter is "t", then we go to the second node "o", and we have reached the prefix "to". To find all other words in the subtree that are prefixed with "to", we need to pick all the children of the node "o" and continue the search in the same way.

Figure 2

Note: the letters in the trie are sorted in alphabetical order. The order is important because it allows us to make sure that we can get the top 5 results in the correct order. Letters are not lexicographically sorted in a regular trie.

Basic trie data structure contains the following elements: a node, and a hashtable. In a regular trie, a node is made up of 26 nodes, one for each letter of the alphabet. Instead of using 26 nodes, we use a hashtable to keep a set of children. In this case, we only need to store the letters we need. We can also add a boolean value to indicate if a node is the last character of a query string.

Figure 3

Data stored in a trie node can be quite numerous. For instance, we could store all the queries that start with a letter on the node. However, this could be very slow. For instance, it takes us 60,000 steps to collect all the queries that start with letter "t" in the worst case if there are 60,000 queries in the system. To make the query process much faster, we can store a maximum of 5 top queries on each node. We can prove that this is a fast enough approach later in the interview.

Based on the discussion, we modify the basic trie data structure and introduce the following elements:

Name	Description
children	A hashtable that contains all children of the node.
top_queries	A list of top 5 queries that are most frequently searched given this node.

Table 3

Now, we have all the ingredients to build a trie. We discuss how we can use trie to store and retrieve search queries.

- Create a root node. The root node is empty.
- Add a new query and its frequency into the trie.
  - Create a node for each character if it does not exist.
  - Update top_queries if it is a top 5 frequently searched query for the prefix.
- Find the prefix node. For example, if the prefix is "tw", we need to find the node representing "w" in the trie.
- Return top_queries stored on the prefix node.

Figure 4

Two more components of the algorithm is the cost of time spent on each step mentioned above **Step 1 – Step 4** in the high-level design.

The cost of operation is logarithmic because it is the cost because we need to traverse a tree to find the prefix node. To get top 5 results in the worst case scenario, there are two operations:

- Traverse all children nodes.
- Collect all top_queries from children nodes.

Let's look at these operations one by one.

Let's look at the time complexity of the first step. Assume the number of children for each node is a fixed number. For instance, there are 36 alphanumeric characters in the dataset. Consequent search time through 36 children is a fixed number. Therefore, the time complexity for the first step is 1.

To collect top_queries, we need to traverse all children in the tree, and collect top_queries from each node. The time complexity is proportional to the number of children in the tree. Since the number of children is a fixed number, the time complexity for the second step is also 1.

Limit the max length of a prefix

Assuming the longest search query ever typed in the system. Thus, it is safe to say a user will never type more than 50 characters in a search query. Therefore, we can limit the number of nodes in a path to 50. This limit ensures the algorithm will work efficiently even for very long search queries.

Cache top search queries at each node

Let's do a back of the envelope estimation. There are 5 UTF-8 bytes stored for each query of each trie. Suppose 2 to 3 UTF-8 bytes are needed to store a character in the English language, and we assume 2 bytes for simplicity. A maximum of 50 queries are stored in each node. Thus, it takes 5 * 50 = 250 bytes to store top_queries on each node.

Although we have ample space in our system, we do not want to waste too much space. Storing top_queries on every node is a fast approach, but it requires a lot of extra space. It is a good trade-off between space and time.

Aggregated time

Aggregated time is the time of aggregated search query. Time' field represents the start time of a week. Frequency' field represents the frequency of the query that was searched during the specified week.

query	time	frequency
tea	2019-10-01 (2019W40)	12000
tea	2019-10-08 (2019W41)	15000
tea	2019-10-15 (2019W42)	9000
tea	2019-10-22 (2019W43)	8000
tea	2019-10-29 (2019W44)	10000

Table 4

Aggregation. The rate of analytics log is usually very large. Look only at one of the logs? Across, the week is a good aggregation unit. We can aggregate the data on a weekly basis.

Let's look at the raw data and the aggregation time difficulties. The raw data is stored in analytics log. Each log entry contains one search query. Aggregation takes a certain time interval as its function and produces the aggregation. On the other hand, aggregating data on a daily basis, we can aggregate 1 day's worth of data on the next day. Generally, we can aggregate data on a minute, hourly, or daily basis, depending on the use case.

Aggregation time

Aggregation time is the time when the aggregation is performed. Let's say we aggregate data on a weekly basis, we can start aggregation at 00:00 on Sunday. Aggregation can take a long time to complete, and some of the queries may not be available until it is ready. Assuming we want to get the most recent aggregated results, we can start aggregation at a later time.

Query time

Let's consider the time complexity of aggregated weekly data. Time' field represents the start time of a week. Frequency' field represents the frequency of the query that was searched during the specified week.

query	time	frequency
tea	2019-10-01 (2019W40)	12000
tea	2019-10-08 (2019W41)	15000
tea	2019-10-15 (2019W42)	9000
tea	2019-10-22 (2019W43)	8000
tea	2019-10-29 (2019W44)	10000

Table 5

Weights. Recent data is more valuable than old data. We assign more weight to recent data.

Filter. We filter the aggregated data. For instance, we can exclude the query if the length is longer than 50 characters.

Top 25. We filter the aggregated message. The system only accepts the top 25 messages.

Top 5. We filter the aggregated message. The system only accepts 5 messages.

Let's look at the time complexity of the algorithm after applying these optimizations.

1. Fetch the prefix node. Time complexity: O(1)
2. Traverse the children of the prefix node. Time complexity: O(1)
3. Sort the candidates. Assume 2 children are left, time complexity: O(2 * log 2) = O(1)

Data gathering service

In our previous design, data is in real-time. Queries are constantly coming in. Data is updated in real-time. This approach is not practical when many queries are being typed in.

In a production system, it takes too much time to make a trie. Data is updated once every week. Thus, it is acceptable.

Let's suppose we are building a system from scratch for data gathering service. Assume the system receives 10 billion queries per day. The data gathering service should be able to reflect the most recent search queries.

Figure 5 shows the proposed data gathering service design.

Figure 5

In Figure 5, the system has one API server and one log watcher. The API server receives queries from users and stores the data in analytics log. The log watcher keeps watching the log and stores any new search queries in the analytics storage. The data in the storage is in the format of table 5.

Aggregators take the raw data from the storage and aggregate it on a weekly basis. The aggregated data is then stored in the aggregated data storage. Weekly trie builder takes the aggregated data and produces a new trie. The new trie is then persisted in the trie DB. Finally, the new trie is then used to replace the old one in the query service.

Query service

Second and third frequency tables are known as Table 1 in top two fields.

- Frequency is the number of times a query has been searched.

Query	Frequency
tw	35
twitch	29
tyler perry	17
twitter	15
tyler the creator	12

Table 2

To get top 5 frequently searched queries, execute the following command:

```
select * FROM frequency table
WHERE query LIKE 'prefix%'
ORDER BY frequency DESC
LIMIT 5
```

For a prefix of length n, there are n table scans. This approach is slow when there are many rows in the table.

Scale the storage

Storing data in a relational database is fast for finding autocomplete queries but slow if it is time to write the suggestions. Let's explore how we can scale the storage.

Since Google is the only organization language, a name may be short or based on the first character. There are some popular letters.

1. If we have one server for storage, we can store queries starting with "a" to "m" on the first server and "n" to "z" on the second server.
2. If we have two servers, we can store queries starting with "a" to "z" on the first server and "t" to "z" on the second server.

Following this logic, we can add queries to a server based on the first character. If we need to add more servers, we can add them to the end of the list. This approach works well because we can add new servers to the end of the list without affecting the existing servers. However, this approach does not work well if we want to add servers in the middle of the list.

We can further improve the approach where responsibility, which you notice that there are a lot more words that start with letter "c" than those that start with letter "x".

To manage the data distribution problem, we introduce a hashing layer. Hash distribution pattern and sharding mechanism are used to distribute the data across the servers. Suppose we have three servers, we use a hashing function to map the queries to the servers. For instance, we can use a hashing function that produces a hash value of 0 for servers 0, 1 for server 1, and 2 for server 2.

Figure 6

Step 4 - Wrap up

In this section, we discuss some extra talking points and some follow-up questions.

Interviewer: How do you scale your design to support multi-language?

To support other non-English queries, we store Unicode characters in the trie. Trie size is not feasible with Unicode characters. Suppose we want to support all the languages in the world, the number of children for each node could be as high as 150,000.

Interviewer: What if they want queries in a country or region?

Interviewer: We can significantly improve the efficiency in this scenario. To store the respective trie, we can store trie in each region.

Interviewer: How can we support the trending (real-time) search queries?

Interviewer: Assume search results are cached, we can queries updates frequent queries. For instance, changes will not take effect.

- If the system was not considered to be quickly the list will because that it is scheduled to run on weekly basis.
- If one is a trending event or topic, the trie is being hit.

Building a real-time search suggestion system is complicated and is beyond the scope of this course as we will only give a few pointers.

- Analytics log ingesting and query streaming.
- Change the sampling rate and log volume.
- Use different servers and different parameters for different queries. For instance, we can run 4 nodes for prefix length 1, 2 nodes for prefix length 2, and 1 node for prefix length 3 and longer.
- Use the Apache Streaming Ingestion Service, Apache Kafka, Flink, etc. Review all these topics respectively.
- Use Apache Hadoop MapReduce to process the logs.

Interviewer: Do you know how to test the autocomplete system? Let's do it.

Reference materials

[1] Trie - Wikipedia https://en.wikipedia.org/wiki/Trie
https://www.facebook.com/notes/facebook-engineering/the-life-of-a-typeahead-query/1411375214119
[2] Facebook Typeahead https://www.youtube.com/watch?v=1-3Bn-P8rO0
[3] Facebook Typeahead https://www.youtube.com/watch?v=kqeb5dfqqVg&feature=youtu.be&t=1m15s
[4] Facebook Typeahead https://www.youtube.com/watch?v=mlzxGeCJ8Tk&feature=youtu.be&t=1m30s
[5] Wikipedia: Top-k and Top-k-frequent Data Stream Distributed Trie Tables
[6] Wikipedia: Top-k and Top-k-frequent Trie
[7] Wikipedia: https://en.wikipedia.org/wiki/Hexagram
[8] Wikipedia: https://en.wikipedia.org/wiki/Unicode#Unicode_planes
[9] Apache Kafka: https://kafka.apache.org/
[10] Apache Hadoop: https://hadoop.apache.org/
[11] Apache Flink: https://flink.apache.org/
[12] Apache Spark: https://spark.apache.org/
[13] Apache Storm: https://storm.apache.org/
[14] Apache Ignite: https://ignite.apache.org/

15
Design YouTube

Similar to other system design interviews, we can start by clarifying the requirements and assumptions. Since interviews are open-ended, it's important to clarify what type of system we'll be designing. Here's what we'll do:

- Step 1 - Understand the problem and establish design scope
- Step 2 - Propose high level design and get buy in
- Step 3 - Design deep dive
- Step 4 - Wrap up

Step 1 - Understand the problem and establish design scope

In Chapter 9, Design Twitter, we've learned that a news feed system is mostly about how the message is delivered. Since YouTube is a video streaming platform, we'd like to know what features are important. Here's what we'll ask the interviewer:

- Features: We'll focus on the following features: upload a video, watch a video, and other functions such as adding comments, likes, and dislikes.
- Platform: We'll support mobile, web browser, and smart TV.
- Scale: We'll assume 5 million daily active users (DAU).
- Key statistics: Assume the average user watches 5 videos per day, and 10% of daily active users upload one video per day.
- Content: Videos can be either small or large, and they can be encrypted.
- Compression: Compression of videos is required.
- Maximum size: Uploaded videos can be up to 1 GB in size.
- Accessibility: Videos can be watched by users who have Internet access.
- Understanding the requirements is critical for designing the right system. We'll ask questions to understand the requirements and clarify any ambiguities. Here's what we've learned so far:

- Product: Like most other system design interviews, we can assume we're asked to work on the following functions: user sign up, video upload, and video streaming.
- Traffic: We have 5 million DAU and we assume users can watch 5 videos per day.
- 10% of daily active users upload 1 video per day. Assume average video size is 300 MB.
- We need to support live streaming.
- Storage: Assume the storage size for videos is 5 GB.
- Cloud services: Cloud services such as CDN and blob storage service will be used.
- Servers: How many servers do we need for video streaming, video processing, and so on.

Back of the envelope estimation

The purpose of back of the envelope estimation is to make sure we are on the same page with the interviewer regarding what we are trying to build. Here's what we'll do:

- Assume the average video size is 300 MB.
- 10% of daily active users upload 1 video per day. Assume there are 5 million daily active users (DAU).
- Assume the average video size is 300 MB.
- Storage for one day: 5 million * 10% * 300 MB = 150 TB
- Storage for one year: 150 TB * 365 = 54,750 TB
- Assume 20% of storage is used for popular videos that are replicated for fast streaming. Assume we use Amazon Web Services (AWS) S3 as our blob storage, the storage cost is around $0.0125 per GB for the first 50 TB and $0.0123 per GB for the next 450 TB. The storage cost for one year is around:
- 50 TB * $0.0125 per GB + 450 TB * $0.0123 per GB = 5,535 TB * 1,000 GB * $0.0125 = $69,187,500

Step 2 - Propose high level design and get buy in

In this section, we'll propose high level design and get buy in from the interviewer. It is important to collaborate with the interviewer at this stage. Here's what we'll do:

- Discuss various APIs that are required.
- Discuss the high level design for the following functions: video upload, video streaming, and other functions such as adding comments, likes, and dislikes.
- Discuss the following data models: Video and User.

Video upload flow

Figure 15-1 shows the high level design for the video upload flow. It consists of the following components:

- Client: A client to watch a video on a mobile phone, a web browser, or a smart TV.
- Load balancer: It distributes traffic evenly across the API servers.
- API servers: Most API requests go through the API servers.
- Metadata DB: It stores video metadata, such as video title, description, size, and video URL.
- Metadata cache: Some video metadata is cached for fast retrieval.
- Original storage: A blob storage system to store the original videos.
- Transcoding servers: Video transcoding is CPU intensive and time consuming. It is far better if a transcoding server is added to the system for video transcoding. Transcoding servers can be added easily as we need more CPU power.
- Transcoded storage: A blob storage system to store transcoded video files.
- CDN: Videos are cached in the CDN and they can be streamed from the closest edge server to the client.
- Completion queue: A message queue that stores information about video transcoding completion events.
- Completion handler: A worker that consumes events from the completion queue and updates the metadata database and cache when video transcoding is finished.

Figure 15-1 shows the high level design for the video upload flow. The explanations of some major components are provided below.

Client

Any client that can stream videos such as a mobile phone, a web browser, and a smart TV is supported.

Load balancer

It distributes traffic evenly across the API servers.

API servers

Most API requests go through the API servers.

Metadata DB

It stores video metadata, such as video title, size, and URL. Database sharding and multi-AZ (Availability Zone) are used to ensure data availability and data persistence.

Metadata cache

Some video metadata is cached for fast retrieval.

Original storage

A blob storage system is used to store the original videos. It's a binary and immutable object. We can think of it as a file that can be written once but read many times. It's a robust and reliable way to store binary data. Amazon Web Services (AWS) S3 (Simple Storage Service) is a popular cloud storage system that provides industry-leading scalability, data availability, security, and performance. We'll use S3 to store the original videos.

Transcoding servers

Video transcoding is CPU intensive and time consuming. We'll need to add a video transcoding server to the system for video transcoding. Transcoding servers can be added easily as we need more CPU power. We'll use S3 to store transcoded video files.

Transcoded storage

It's a blob storage system to store transcoded video files. It's similar to the original storage system, we'll use S3 to store transcoded video files.

CDN

Videos are cached in the CDN and they can be streamed from the closest edge server to the client.

Completion queue

It's a message queue that stores information about video transcoding completion events.

Completion handler

It's a worker that consumes events from the completion queue and updates the metadata database and cache when video transcoding is finished.

Video streaming flow

Figure 15-2 shows the high level design for the video streaming flow. It consists of the following components:

- Load balancer: It distributes traffic evenly across the API servers.
- API servers: Most API requests go through the API servers.
- Metadata DB: It stores video metadata, such as video title, description, size, and video URL.
- Metadata cache: Some video metadata is cached for fast retrieval.
- CDN: Videos are cached in the CDN and they can be streamed from the closest edge server to the client.

Figure 15-2 shows the high level design for the video streaming flow. The explanations of some major components are provided below.

Load balancer

It distributes traffic evenly across the API servers.

API servers

Most API requests go through the API servers.

Metadata DB

It stores video metadata, such as video title, size, and URL. Database sharding and multi-AZ (Availability Zone) are used to ensure data availability and data persistence.

Metadata cache

Some video metadata is cached for fast retrieval.

CDN

Videos are cached in the CDN and they can be streamed from the closest edge server to the client.

Other functions

Other functions such as adding comments, likes, and dislikes are similar to the video streaming flow. We'll skip the explanation.

Step 3 - Design deep dive

In this section, we'll dive deep into some of the components and discuss some important topics. Here's what we'll do:

- Discuss various video formats and codecs.
- Discuss the following topics: video transcoding architecture, streaming protocol, error handling, system optimization, and adding other features.

Video transcoding

Video transcoding is important because most raw video data stored in memory or disk takes a huge amount of bandwidth. The bandwidth requirement is a bottleneck for the performance of a video streaming system. Video transcoding is also important because most of the videos we record on our phones or other video recording devices are in a format that is not standard for video streaming. Video transcoding is also important because it allows us to stream a video to other devices such as a smart TV that have different specifications on bandwidth, resolution, and so on.

Video transcoding is a CPU intensive and time consuming operation. We need to add a video transcoding server to the system for video transcoding. Transcoding servers can be added easily as we need more CPU power. We'll use S3 to store transcoded video files.

Directed acyclic graph (DAG) model

To transcode a video, many different video processing tasks are needed. Different creators have different requirements for their videos, such as the following: some videos require a lot of anti-piracy checks, some videos require high definition, some videos require watermarks, and so on. To support various video processing requirements from different creators, we think it makes sense to allow creators to specify what video processing they want us to perform. We choose the directed acyclic graph (DAG) model to define video processing pipelines because it's easy to extend and it's easy to define the processing tasks. Figure 15-3 shows the DAG model for video processing.

Figure 15-3

- A node in the DAG model represents a task, such as video encoding, audio encoding, or thumbnail generation.
- Different types of media processing tasks can be defined in any arbitrary data structure.
- Tasks are generated by the system based on configurations stored in the configuration files.
- Artificial intelligence (AI) can be easily added to the system for enhanced video processing.
- Adding new processing tasks to the system is easy.

Figure 15-4 shows the video transcoding architecture. It's a combination of the message queue and DAG models. It's composed of the following components:

Figure 15-4

Preprocessor

- Video segmentation: The preprocessor splits video streams into chunks based on GOP (group of picture) alignment (2 pictures in the same group can not be played independently without the other one) and attaches a customizable GOP (group of picture) structure (up to 300 pictures in the same group) to each chunk. This is important to some video streaming platforms such as Twitch.
- Some old versions of popular video players may not be able to play newly encoded videos. We split the encoding work based on GOP (group of picture) alignment to support fault tolerance.
- The preprocessor stores the video segmentation data (such as the number of GOP (group of picture) and the storage location for chunks) in storage for other processing units to access.
- The preprocessor generates DAG (directed acyclic graph) based on the client's programming.
- The preprocessor stores the streaming manifest (what protocol, resolution, and bitrate are supported on the client side) in storage. The client side can download the streaming manifest from storage to see what protocol, resolution, and bitrate are supported.

DAG scheduler

- The DAG scheduler splits a DAG graph into stages of tasks and puts the stages in queues.

Figure 15-5 shows an example of how the DAG scheduler works. It splits the DAG in Figure 15-3 into two stages. The task in stage 1 can be executed in parallel because they are independent of each other. Stage 2 depends on stage 1 and its tasks will be executed after stage 1 is finished.

Figure 15-5

Resource manager

- Resource allocation: The resource manager allocates the task workers to run the tasks in stage 1 and stage 2 in Figure 15-5.
- Task worker assignment: The resource manager assigns task workers to run the tasks in stage 1 and stage 2 in Figure 15-5.
- Optimal resource allocation: The resource manager allocates the optimal resources to the task workers to run the tasks in stage 1 and stage 2 in Figure 15-5.

Task workers

- Task workers run tasks which are defined in the DAG graph.
- Different task workers may run different tasks.

Temporary storage

Various types of storage are used for the data of video transcoding operations. Metadata is small and frequently accessed by the task workers. Some video data such as the video segments and the DAG graph are temporary and only stay in the system for a short time. Other video data such as the encoded video frames are more persistent.

Encoded video

- The encoded video is the output of a video transcoding pipeline.

Figure 15-6 shows the final version of our video transcoding architecture. It's a combination of the message queue and DAG models. It's composed of the following components:

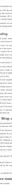

Figure 15-6

Video transcoding architecture is flexible and easy to extend. We can add new models to support new video processing algorithms. We can also add new video formats to support cross platform streaming.

Streaming protocol

- Streaming protocol is the most popular protocol for real-time streaming. The majority of live streaming traffic uses a streaming protocol.
- Streaming protocol works well for live streaming because it's very fast.
- Streaming protocol is not supported by some old versions of popular video players.
- Dynamic adaptive streaming over HTTP (DASH) is not supported by Apple.
- Apple HTTP Live Streaming (HLS) is a streaming protocol over HTTP that supports live video streaming.
- Apple HLS is supported by most video players.
- Apple HLS has longer latency than other streaming protocols.

System optimizations

- Speed optimization: place uploaded videos close to the user.
- We can upload a video to a server that is geographically far from the user. For example, we live in California and we upload a cat video to a server in India. It could take a very long time for the video to be uploaded.
- Slow upload can be avoided if we upload the video to a server that is geographically close to the user, then asynchronously copy the video to other data centers.
- The CDN architecture is a good fit for this.

Figure 15-7

Figure 15-7 shows how a video is uploaded to a close by CDN server and watched by users from other locations.

- Slow video streaming can be avoided if we stream videos from servers that are geographically close to the user.
- Videos can be cached in CDN servers close to the user.

Figure 15-8 shows how videos are cached in CDN servers close to the user.

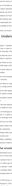

Figure 15-8

- Slow video playback can be avoided if we adapt bitrate according to network bandwidth, device resolution, and so on.
- Bitrate is the amount of data that is required to be streamed over a unit of time.
- Bitrate can be classified as either video bitrate or audio bitrate.
- Adaptive bitrate streaming is a technique that allows a video to be streamed at various bitrate quality levels.
- Adaptive bitrate streaming is CPU intensive and it requires a lot of storage.

Speed optimization: place uploaded videos close to user

Speed optimization: place uploaded videos close to user

- Speed optimization: parallel video uploading.
- Uploading a video as a single chunk may be prone to network failures and may not meet the latency requirement on the video size. Splitting videos into smaller chunks for uploading is a better approach.
- Splitting a video into smaller chunks allows for quick retry after failure.
- Splitting a video into smaller chunks can be uploaded in parallel.

Figure 15-9

- Speed optimization: parallel video processing.
- We can build a pipeline to stream the uploaded video from the client to a CDN server.
- The client splits a video into smaller chunks and uploads them to the CDN server.
- The CDN server receives the video chunks and it can start to transcode the video when it receives the first chunk.
- The CDN server receives the video chunks and it can start to transcode the video when it receives the first chunk.

Figure 15-10

- Speed optimization: place upload servers close to user.
- We can place upload servers close to user to improve upload speed.
- We can place streaming servers close to user to improve streaming speed.
- We can place CDN servers close to user to improve streaming speed.

Figure 15-11

- Other optimizations: we could ask the interviewer if she would like us to talk about other optimizations.
- Consider other video formats such as Facebook and Netflix.
- Consider adding more error checking and error handling if there are enough time.

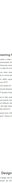

Figure 15-12

- Consider adding live streaming if there is enough time.
- Consider adding copyright protection such as Digital Rights Management (DRM) if there is enough time.
- Consider adding video recommendation if there is enough time.
- Consider adding analytics if there is enough time.

Error handling

For a large-scale system, errors can occur everywhere. It is hard, subtle and tedious to deal with failure cases. We need to think through what errors can happen and how we will handle them.

- Recoverable errors: for example, a video segment fails to transcode from a certain format. Most errors are recoverable with a retry if the error is not caused by bad video format.
- Non-recoverable errors: for example, the video format is malformed or unsupported. We return a proper error code for the client to understand the error.
- Client error: we need to handle various client errors such as the upload of the video fails, the video is corrupted while being streamed, the video is not in a supported format, and so on.
- Upload error: retry a few times and abandon the upload if the client still receives errors.
- Corrupted video: we can use checksums for both the uploaded video and the video chunks. We compare the checksum we received against the checksum we calculated to see if the video has been corrupted.
- Supporting various video formats: new video formats may be added to support new devices and platforms.
- Supporting new video codecs: new codecs may be added to reduce storage and bandwidth requirements.
- Supporting advanced video processing: the system may be modified to support advanced video processing such as adding new filters and enhancing the recommendation engine.

Step 4 - Wrap up

In this section, we recommend some additional topics that may be interesting to the interviewer.

- Rate limiting: it's easy for users to upload videos of various sizes and formats. Rate limiting can be applied at the API level.
- Security: we can employ various levels of security, including authentication and encryption, and we can also consider digital rights management (DRM) to protect copyrighted videos.
- Different business models: we can discuss various business models such as advertisements, premium videos, and so on.
- Scalability: we can discuss various ways to scale the system. For example, we need to scale the API servers, we need to scale the upload flow, we need to scale the streaming flow, we need to scale the database, we need to scale the message queue, and so on.
- Various performance optimizations: caching frequently accessed data is a good idea. We can also discuss which data is suitable for being stored on a content delivery network (CDN).
- Various kinds of other features: we can discuss various other features if we have time, such as adding a new feed, recommending videos based on user's watch history, finding trending videos, and so on.
- Monitoring the key metrics: it's important to monitor the key metrics such as CDN delivery latency, video transcoding time, and so on.
- Video ads: we can discuss various ways to place video ads, such as ad insertion in live streaming.

Reference materials

- Apple HTTP Live Streaming (HLS)
- Dynamic Adaptive Streaming over HTTP
- DRM Digital Rights Management
- Facebook Video Infrastructure
- Netflix Video Infrastructure
- NETFLIX VMAF: https://netflixtechblog.com/vmaf-the-journey-continues-44b51ee9ed12
- Twitch Video Infrastructure
- Vimeo Video Infrastructure
- Amazon Web Services (AWS) S3 (Simple Storage Service)
- Amazon Web Services (AWS) CDN
- AWS Elastic Load Balancing (Elastic Load Balancer)
- Amazon Web Services (AWS) Elastic Compute Cloud (EC2)
- Amazon Web Services (AWS) Relational Database Service (RDS)
- Amazon Web Services (AWS) Simple Queue Service
- Amazon Web Services (AWS) Simple Notification Service
- Amazon Web Services (AWS) Simple Email Service
- Amazon Web Services (AWS) Elastic Container Service
- Amazon Web Services (AWS) Identity and Access Management
- Amazon Web Services (AWS) Key Management Service
- Amazon Web Services (AWS) Elastic Block Store
- Amazon Web Services (AWS) Simple Notification Service
- Amazon Web Services (AWS) Simple Email Service
- Amazon Web Services (AWS) Elastic Container Service
- Amazon Web Services (AWS) Identity and Access Management
- Amazon Web Services (AWS) Key Management Service
- Amazon Web Services (AWS) Elastic Block Store
- Amazon Web Services (AWS) Virtual Private Cloud
- Amazon Web Services (AWS) Lambda
- Amazon Web Services (AWS) Aurora
- Amazon Web Services (AWS) DynamoDB
- Amazon Web Services (AWS) ElastiCache
- Amazon Web Services (AWS) Simple Workflow Service
- Amazon Web Services (AWS) Elastic MapReduce
- Amazon Web Services (AWS) Kinesis

16
Design Google Drive

In recent years, cloud storage services such as Google Drive, Dropbox, Microsoft OneDrive and Apple iCloud have gained tremendous popularity. Cloud storage services have many users worldwide.

In our design, consider a centralized Google Drive-like cloud storage service. In this design, Google Docs, or the Google Docs and spreadsheet software, Google Sheets, are not discussed specifically. Instead, we refer files in the future. For our design, we discuss how we can sync files across multiple devices, how we can notify other users when a file is edited, and how we can scale our infrastructure to 10 million users. We also discuss how we can make our system more reliable.

Step 1 - Understand the problem and establish design scope

Google Drive is a large-scale, complex system. It is impractical to ask questions to interview about the entire system. Instead, ask questions to focus on the most important features.

Candidate: Which platform do we need to support? Mobile? Web? Mac? Windows?
Interviewer: Let's start with a web app, and a mobile app.
Candidate: How do users add files to Google Drive?
Interviewer: Any file type can be uploaded.
Candidate: Can files be shared between users?
Interviewer: Yes, sharing files with other users is a key feature.
Candidate: How many users does the product have?
Interviewer: Two million users initially, and it will grow rapidly.
Candidate: How many files do we expect to be added to Google Drive every month?
Interviewer: 500 million files per month.
Candidate: How long do files need to be stored?
Interviewer: Files can be stored forever.
Candidate: What is the maximum file size allowed?
Interviewer: 50 GB is the maximum file size.
Candidate: Do we need to support multiple file versions?
Interviewer: Yes, existing versions should be stored.
Candidate: Do users need data stored in their home region?
Interviewer: Not in the initial design.

Based on the above conversation, we focus on the following:

- Add files
- Download files
- Sync files across different devices
- See file revisions
- Share files with other users
- Send notifications when a file is added, deleted, or edited with us.
- Support up to 2 million users
- Ease of maintenance, cost, and performance are important.
- Some commonly used features, such as file chunking, compression, and delta transfer are not in scope. We focus on the core design.

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

Instead of storing files in our system, we store files in a third party cloud storage platform. We use Amazon S3 for our system. Files can be uploaded to and downloaded from Amazon S3 over the Internet. We can use Google Drive web editors to create, edit, and view files in the cloud. We can also use Google Docs, Google Sheets, and Google Forms to create new files online. We can also use Google Docs, Google Sheets, and Google Forms to view and edit files in the cloud.

Google Docs, Google Sheets, and Google Forms are not in scope.

Our system is built on top of Google Cloud Platform.

Google Drive API is a RESTful API for building Google Drive integrations. Google Drive offers many Google Drive APIs. We use the following Google Drive APIs in our design:

- Files API: Manage files and folders in Google Drive.
- Changes API: Track changes made to files in Google Drive.
- Push notifications API: Receive notifications when files in Google Drive change.

Google Drive API supports JSON and HTTP. We use JSON as the data format for API responses. We use HTTP as our protocol for client-server communication.

APIs

1. Upload files API

To upload a file, we can use the Google Drive API's files.create method. This method uses the HTTP POST request method. The API endpoint is:

```
POST https://www.googleapis.com/upload/drive/v3/files?uploadType=resumable
```

2. Get file metadata

Request HTTP method: HTTP GET

Request URL:

```
GET https://www.googleapis.com/drive/v3/files/fileId?fields=parts
```

Request body: none

Response body: none

Example parameters:

```
GET https://www.googleapis.com/drive/v3/files/{file_id}?fields=parts
Content-Type: application/
Content-Length: 0
```

Request headers:

```
Content-Type: application/
Content-Length: 0
```

3. Get file metadata

Request HTTP method: HTTP GET

Request URL:

```
GET https://www.googleapis.com/drive/v3/files/fileId?fields=parts
```

Request body: none

Response body: none

Example parameters:

```
GET https://www.googleapis.com/drive/v3/files/{file_id}?fields=parts
Content-Type: application/
Content-Length: 0
```

Request headers:

```
Content-Type: application/
Content-Length: 0
```

4. Download file API

To download a file, we can use the Google Drive API's files.get method. This method uses the HTTP GET request method. The API endpoint is:

```
GET https://www.googleapis.com/drive/v3/files/{fileId}?alt=media
```

Request body: none

Response body: none

Example parameters:

```
GET https://www.googleapis.com/drive/v3/files/{file_id}?alt=media
Content-Type: application/
Content-Length: 0
```

Request headers:

```
Content-Type: application/
Content-Length: 0
```

5. Get changes API

To get changes, we can use the Google Drive API's changes.list method. This method uses the HTTP GET request method. The API endpoint is:

```
GET https://www.googleapis.com/drive/v3/changes?pageToken=pageToken&includeItemsFromAllDrives=true&supportsAllDrives=true&fields=kind,newStartPageToken,nextPageToken,changes(kind,type,time,removed,fileId,driveId,file(kind,mimeType,id,name,parents,trashed,shared,ownedByMe,capabilities(canCopy,canEdit),modifiedByMeTime,modifiedTime,webContentLink,webViewLink,iconLink,lastModifyingUser(displayName,emailAddress),size,md5Checksum,teamDriveId,driveId,hasThumbnail,thumbnailVersion,modifiedByMe,writersCanShare,viewedByMeTime,sha256
```

18

Nearby Friends

We talked about how to find nearby places in Proximity Service. In this section, we discuss how to find nearby friends. For simplicity, we assume that a user can see all nearby friends regardless of their preferences/settings. We also assume that geographical location is recorded every 30 seconds. When a friend is far away, the location change is not perceivable by another user. So, we assume we can ignore far away friends.

If we use the Proximity Service design, the server needs to calculate the distance between every friend and every active user. This is a very resource-intensive job. For example, if the average number of friends per user is 30, and there are 1 million active users at the same time, the server needs to perform 30 million distance calculations every location change. If friends' locations change every 30 seconds, the server needs to perform 1 million * 30 / 30 = 1 million distance calculations per second.

Step 1 - Understand the Problem and Establish Design Scope

Nearby friends: Nearby means a configurable number of miles between two users.

Location accuracy: Location accuracy is approximate. It is okay if it is off by a few miles.

Number of friends: The average number of friends per user is 30. The maximum number of friends is 5000.

Location data: We have latitude and longitude of a user's location. The location data is stored in the US.

Latency: The end-to-end latency should be low. It means it takes a few seconds for a friend to see the nearby friend's location change.

Location: Users do not want to share their location data with their friends. The location data is shared for the purpose of finding nearby friends.

Mobile phone: We only support mobile phone users. Other clients are out of scope.

Authentication and Consent: Assume authentication and consent are handled by other services. We focus on the business logic for this system.

Scale: Assume we have 10 million active users at the same time.

Functional requirements

1. User location data is used to find nearby mobile phone users.
2. When a friend is far away, it is no longer shown to the user. When a friend becomes nearby again, it is shown to the user.
3. Location data is stored in the US.

Non-functional requirements

1. Low latency: it should be fast to find nearby friends.
2. Reliability: the system should be reliable overall.
3. Eventual consistency: it is okay for a nearby friend's location to be stale by a few seconds.

Back-of-the-envelope estimation

Let's perform back-of-the-envelope estimation to see if our assumptions are reasonable and to determine what kind of computer system we need.

1. The average number of friends per user is 30.
2. The average radius of a nearby area is 5 miles.
3. The average number of active users per hour is 1 million. The system only keeps the latest location data of a user. We only need one location record per user in the cache at any given time.
4. Location data is updated every 30 seconds. On average, we receive 1 million / 30 = 33,000 location updates per second.
5. Assume the peak number of active DAU is 10 million. On average, there are 10 million / 30 = 300,000 location updates per second. We need to be able to handle 300,000 location updates per second.
6. Assume the peak number of active DAU is 20 million. We need to be able to handle 600,000 location updates per second.

Storage requirement:

1. Storage for user location data: 10 million * 100 = 1 GB
2. Storage for location history data: 1 billion * 100 = 100 GB
3. Storage for user profile and friend data: 1 billion * 1 KB = 1 TB

Step 2 - Propose High-Level Design and Get Buy-In

Peer-to-peer vs client-server

1. Peer-to-peer: each mobile phone connects to its peer mobile phones to receive their locations.
2. Client-server: each mobile phone connects to the backend server to receive locations of its friends.

Peer-to-peer

Peer-to-peer is not a good choice for a mobile app. It is hard to establish a reliable connection and battery consumption is a major concern on mobile phones. It is okay for a small number of users, but it is not scalable for a large number of users.

Figure 18.01 Peer-to-peer

Client-server

The following is a high-level design for client-server architecture. Most of the work is done by the backend servers.

Figure 18.02 Client-server

The following is a high-level design for the client-server architecture. We explain each component in the following sections.

Figure 18.03 High-level design

Load balancer

The load balancer is the entry point for the RESTful API and sits in the backend service cluster. It distributes traffic to the RESTful API servers and the websocket servers.

RESTful API server

The RESTful API server is the entry point for other services. It handles all the non-real-time tasks, such as adding/removing friends, updating user profile, etc. We will not discuss the RESTful API server in this chapter because it can be implemented using the request-response model, which is a well-known technology.

Websocket server

The websocket server is a stateful server. It handles the following tasks:

1. Receive location updates from mobile phone users.
2. Calculate the locations of a user's friends when receiving a location update.
3. Send location data of friends to the user.

Redis pub/sub server

Redis pub/sub server is used to manage pub/sub channels. We will discuss why we need pub/sub channels in the deep dive section.

Redis location cache

Redis location cache is a key-value store. It stores the most recent location data of each user. Cache entries have a 30 seconds TTL (Time To Live) because we do not need to keep the location data longer than 30 seconds. When a user's location is updated, the websocket server can get the user's friends' locations from this cache.

User database

The user database stores user profile data, such as name, profile image, etc.

Location history database

The location history database stores the location history of users' nearby friends. Cassandra is a good fit for this.

Nearby friends data

JSON guide: JSON (JavaScript Object Notation) is a lightweight data format for exchanging data over the Internet. It is easy to read and write for humans and machines. It is also easy for machines to parse and generate. It is a good fit for RESTful API.

Redis Geohash

Redis Geohash stores locations as keys in the sorted set. We derive Geohash from the latitude and longitude of a location. Geohash has 11 characters in our design.

Protocol buffers

Protocol buffers (Protobuf) is a binary protocol developed by Google. It is lightweight and fast. We need to define the message format for Protobuf. We will use Protobuf for the websocket protocol.

Websocket

Websocket is a TCP based protocol. It is a bidirectional protocol. It can be used for both client and server to initiate messages. We will use Websocket for the communication between the client and the websocket server.

Why do we use a stateless service for RESTful API?

As we have learned from Chapter 6 Design a key-value store, stateless servers are easy to scale because any request can be sent to any server. Websocket servers are stateful because we need to keep the websocket connection open. We will discuss why we need to be able to scale the websocket servers in the deep dive section.

Why do we use Redis pub/sub server?

Redis pub/sub server is used to manage pub/sub channels. We will discuss why we need pub/sub channels in the deep dive section.

Why do we use a key-value store to store the last known location?

We need to store the last known location of each user. The location data will be used to calculate the locations of a user's friends. We need a fast read/write ratio. A key-value store is a good fit because it is quite easy to scale, and it can provide sub-millisecond latency.

Why do we use Protobuf for the communication between the client and the websocket server?

Protobuf is a binary protocol. It is very lightweight and fast. It can reduce the payload size and parsing overhead. It is a good fit for our system. It is easy to define the message format for our system. We can use Protobuf for other communications in our system.

Why do we use Websocket for the communication between the client and the websocket server?

Websocket is a persistent TCP connection. It can be used for both client and server to initiate messages. It is a good fit for our system because our system needs to handle location updates from a friend and calculate and receive locations of nearby friends in the future.

Why do we use Geohash to store locations in Redis?

We need to store a user's location in the cache. We need to find nearby friends based on a user's location. We will discuss how we use Geohash to find nearby friends in the deep dive section.

Step 3 - Design Deep Dive

How RESTful API notifies websocket server?

Why do we need a service discovery?

Why do we need pub/sub channels?

How many pub/sub channels do we need?

How do we distribute messages from pub/sub channels to clients?

How do we send the initial data to a user?

How do we store and send location history data?

How do we scale the websocket servers?

How do we handle adding/removing friends?

How do we handle a user's friend list?

How do we handle a user's location change?

How do we handle a user's friend location change?

How do we handle adding/removing friends?

How do we handle a user's friend list?

How do we handle a user's location change?

How do we handle a user's friend location change?

Nearby random places

Alternative to Redis pub/sub

Alternative to Redis pub/sub

Alternative to Redis pub/sub

Step 4 - Wrap Up

Monitoring

Feature rollout

Internationalization

Supporting other features

Reference Materials

Chapter Summary

21
Metrics Monitoring and Alerting System

In this chapter, we design a scalable metrics monitoring and alerting system. Such a system is fundamental for tech companies. We start by discussing the requirements and the high-level design. We then explore the data model and query language. We also discuss the storage system, alerting system, and visualization system. We conclude with discussions on system operations, such as how to scale the system and how to monitor the system itself.

Step 1 - Understand the Problem and Establish Design Scope

Metrics can take many different forms in our new era. Metrics may be as simple as a number of hosts in a cluster or as complicated as the 95th percentile of HTTP response latency for a cluster of web servers over the last 10 minutes. Metrics may also be related to different things, such as hardware resources, high-level services, or business-related metrics. Our system should be able to accommodate different types of metrics. In this chapter, we focus on building a scalable system for monitoring and alerting on time-series metrics. Here are a few clarifying questions we ask the interviewer to narrow the scope of the system we are building.

Candidate: What kind of metrics do you want to monitor? Is it server metrics, web metrics, or something else?
Interviewer: It can be any kind of metrics. It is up to you to define what kind of metrics we want to monitor.
Candidate: How many metrics do you plan to monitor? Will the number of metrics be a scalability issue?
Interviewer: Yes, it could be a scalability issue. Your system should be able to accommodate a large number of metrics.
Candidate: What is the retention period?
Interviewer: Let's assume we want to keep data for two weeks.
Candidate: Can we reduce the resolution of metrics for storage purposes?
Interviewer: Yes, that is a good idea.
Candidate: What kinds of alerts do you want to send? Do you want to send alerts via email, text message, or phone call?
Interviewer: It should be able to support all three options.
Candidate: Do you want to use a third-party service to build this system?
Interviewer: No, we want to build our own system.

Functional requirements

In the interview, we have established that we are building a system that monitors metrics and generates alerts. The system should be able to support different types of metrics. It should also be able to apply different types of data retention and data resolution policies. We also need to be able to send alerts through different channels, such as email, text message, and phone call.

Non-functional requirements

- Scalability: The system should be scalable to accommodate a large volume of metrics.
- Low latency: The system should have low query latency for dashboards and alerts.
- Reliability: The system should be highly reliable to avoid missing critical alerts.
- Flexibility: It should be easy to add new types of metrics and alerts in the future.

Back of the envelope estimation

Let's assume we have 100 million daily active users (DAU) in a cloud service. Let's also assume we have 1 billion running hosts/containers/VMs. Let's also assume we have 1 hundred metrics per host/container/VM on average. We plan to keep data for two weeks. Here are some back of the envelope calculations:

- 1 billion hosts/containers/VMs × 100 metrics per host/container/VM = 100 billion metrics
- 100 billion metrics × 2 bytes per data point = 200 billion bytes
- 200 billion bytes × 86,400 seconds in a day × 14 days = 2,419,200,000,000 bytes = 2.4 TB

Step 2 - Propose High-Level Design and Get Buy-In

In this section, we discuss the following topics: metrics collection, data ingestion, data storage, and alerting.

Fundamentals

A time-series consists of a series of values for a metric over time.

Figure 21.1: Time-series

The data access pattern looks like this:

- Write heavy: thousands of data points are written every second. Only a few of them are read at any time.
- Spikes in data do not matter: we care about the overall trend. For example, the 95th percentile response time for a cluster of web servers over the last 10 minutes.

A time-series can be uniquely identified by its name and optional key-value pairs. For example, the following time-series name and key-value pairs uniquely identify a time-series. We use the following format for time-series throughout this book.

```
metric_name {key1=value1, key2=value2, ...}
```

For example, the following time-series name and key-value pairs represent the CPU load for the production server in the west region:

```
cpu.load {host=prod, region=west}
```

A time-series can be represented by the following two data structures:

```
struct {
    metric_name string,
    optional_key_value_pairs map<string, string>,
    timestamps array<int>,
    values array<double>
}
```

```
struct {
    metric_name string,
    optional_key_value_pairs map<string, string>,
    timestamps array<int>,
    values array<double>
}
```

Figure 21.2: Time-series data structure

The following table shows the CPU load for a production server instance. The data is collected every 1 minute and kept for 10 minutes.

timestamp	cpu.load {host=prod, region=west}
1630630800	0.5
1630630860	0.8
1630630920	0.7
1630630980	0.9
1630631040	0.5

Table 21.1: CPU load for a production server instance

The following table shows the memory usage for a production server instance. The data is collected every 1 minute and kept for 10 minutes.

timestamp	memory.free {host=prod, region=west}
1630630800	10240
1630630860	9930
1630630920	9820
1630630980	9710
1630631040	9600

Table 21.2: Memory usage for a production server instance

Data access pattern

Figure 21.3: Data access pattern

Figure 21.3 shows the data access pattern for our system. It is write heavy and read spikes. This is because many software or DevOps teams monitor recently added metrics for a period of time to see if these new metrics cause any "unusual" behavior. Once they have gained enough confidence that the new metrics do not cause any "unusual" behavior, they may only access the metrics when there is a production issue or when they need to generate a business report. For example, we may not deploy a new server for a few days if the CPU load of the new server instance shows a normal pattern. However, if we see a spike in CPU load, we may be alerted. Or if we see a spike in CPU load in the morning, we may not care about it at all because it happens every day at that time and it is a normal pattern.

Data retention

We may be able to persist almost all metrics for a few weeks if we monitor and scale our infrastructure properly. For the purpose of this interview, we may be able to persist all metrics for one year. After one year, we may replace some of the metrics with higher resolution metrics with lower resolution metrics. We may also be able to persist some of the metrics for a longer period of time if we find it useful to do so. For example, we may want to know if the CPU load of a server instance is always under 50% for the last year. We may also want to know how much disk space we will need in the next year to plan for storage. We may also want to persist some of the metrics for long periods of time to see long-term trends in business metrics.

High-level design

Figure 21.4: High-level design

The high-level design diagram shown in Figure 21.4 supports various use cases that we have mentioned in the requirements. It takes in metrics data from various sources, including cloud services, infrastructure elements, and hosts. Metrics collection, data ingestion, data storage, alerting, and visualization are essential components we need to build a robust system. We discuss the purpose of each component in the following paragraphs.

Metrics collection

For metrics collection, we need to collect metrics data from different sources. We discuss some of the options we have for collecting metrics data.

Figure 21.5: Metrics collection

For metrics collection, we can either pull metrics or push metrics to our system. We discuss the pull model and the push model in the following paragraphs.

Pull model

In the pull model, we need to deploy a metrics collector. The metrics collector will pull metrics from the underlying servers. The metrics collector may be a long-running binary or a small binary that runs on a server. We need to decide how we can get the list of all the server instances we need to collect metrics from. We can use a service discovery system such as Zookeeper to provide us with the list of server instances we need to collect metrics from. Service discovery is an effective way to manage a large number of server instances. It provides a mechanism for the metrics collector to find the server instances we need to collect metrics from. It is also highly configurable and scalable. We discuss the pull model in more detail in the following paragraphs.

Figure 21.6: Pull model

In the pull model, we need to deploy a metrics collector. The metrics collector will pull metrics from the underlying servers. The metrics collector may be a long-running binary or a small binary that runs on a server. We need to decide how we can get the list of all the server instances we need to collect metrics from. We can use a service discovery system such as Zookeeper to provide us with the list of server instances we need to collect metrics from. Service discovery is an effective way to manage a large number of server instances. It provides a mechanism for the metrics collector to find the server instances we need to collect metrics from. It is also highly configurable and scalable. We discuss the pull model in more detail in the following paragraphs.

Figure 21.7: Pull model with service discovery

Push model

In the push model, we need to deploy a metrics collector. The metrics collector will receive metrics from the underlying servers. The metrics collector may be a long-running binary or a small binary that runs on a server. We need to decide how we can get the list of all the server instances we need to collect metrics from. We can use a service discovery system such as Zookeeper to provide us with the list of server instances we need to collect metrics from. We discuss the push model in more detail in the following paragraphs.

Figure 21.8: Push model with service discovery

Pull or push

For most of the discussions, we will use the pull model because it is easier to add a server to collect metrics from. We also think the pull model is more suitable for our distributed environment. We can easily run our metrics collector on every server we maintain. We also think it is more reliable because we only need to make sure a server is running in order to pull metrics from it. We also think it is easier to add a new server to be monitored because we only need to make sure the metrics collector is running on the server. We also think it is easier to troubleshoot the metrics collector because we can access the metrics we collect from any server we want. We also think it is easier to develop a scalable system because we can choose any server we want to receive our pull requests. We also think it is easier to develop a highly available system because we do not need to push our metrics across a network. We also think it is easier to develop a secure system because we do not need to open a port on the server we want to collect metrics from. We also think it is easier to develop a scalable system because we can choose any server we want to receive our pull requests. We also think it is easier to develop a highly available system because we do not need to push our metrics across a network. We also think it is easier to develop a secure system because we do not need to open a port on the server we want to collect metrics from.

Metrics collector

We need to deploy a metrics collector to collect metrics from various sources. We discuss some of the options we have for deploying a metrics collector in the following paragraphs.

Figure 21.9: Metrics collector

- Metrics collector. It will receive metrics from Metrics collector and process them into a format that can be ingested into our data ingestion system.
- Data ingestion system. It will receive metrics from Metrics collector and store them in our data storage system.
- Data storage system. It will store metrics from Data ingestion system and provide them to Alerting system and Visualization system.
- Alerting system. It will receive metrics from Data storage system and send alerts to various messaging systems.
- Visualization system. It will receive metrics from Data storage system and provide them to various visualization systems.

Step 3 - Design Deep Dive

In this section, we discuss the following topics: data model, high-level architecture, query language, and storage layer.

Data model

The data model for our system is very important. It affects how our system is implemented. We want to make our data model simple, flexible, and extensible. We discuss the data model in the following paragraphs.

Metrics name

We need to be able to identify a metric by its name. We need to be able to query for all the metrics that have a certain prefix. We also need to be able to support hierarchical metrics. We discuss the metrics name in the following paragraphs.

Figure 21.10: Metrics name

Key-value pairs

We need to be able to identify a metric by its key-value pairs. We need to be able to query for all the metrics that have a certain key-value pair. We discuss the key-value pairs in the following paragraphs.

Figure 21.11: Key-value pairs

Metrics collection

For metrics collection, we need to collect metrics from various sources. We discuss some of the options we have for collecting metrics in the following paragraphs.

Figure 21.12: Metrics collection

- Metrics collection. It will receive metrics from various sources and process them into a format that can be ingested into our data ingestion system.
- Data ingestion system. It will receive metrics from Metrics collection and store them in our data storage system.
- Data storage system. It will store metrics from Data ingestion system and provide them to Alerting system and Visualization system.
- Alerting system. It will receive metrics from Data storage system and send alerts to various messaging systems.
- Visualization system. It will receive metrics from Data storage system and provide them to various visualization systems.

Data ingestion system

Data ingestion system is a system that receives metrics from Metrics collection and stores them in our data storage system. We discuss the data ingestion system in the following paragraphs.

Figure 21.13: Data ingestion system

Data storage system

Data storage system is a system that stores metrics from Data ingestion system and provides them to Alerting system and Visualization system. We discuss the data storage system in the following paragraphs.

Figure 21.14: Data storage system

Alerting system

Alerting system is a system that receives metrics from Data storage system and sends alerts to various messaging systems. We discuss the alerting system in the following paragraphs.

Figure 21.15: Alerting system

Now that we have discussed the high-level design, let's discuss the data model and query language.

Data model

The data model for our system is very important. It affects how our system is implemented. We want to make our data model simple, flexible, and extensible. We discuss the data model in the following paragraphs.

Metrics name

We need to be able to identify a metric by its name. We need to be able to query for all the metrics that have a certain prefix. We also need to be able to support hierarchical metrics. We discuss the metrics name in the following paragraphs.

Figure 21.16: Metrics name

Key-value pairs

We need to be able to identify a metric by its key-value pairs. We need to be able to query for all the metrics that have a certain key-value pair. We discuss the key-value pairs in the following paragraphs.

Figure 21.17: Key-value pairs

Metrics collection

For metrics collection, we need to collect metrics from various sources. We discuss some of the options we have for collecting metrics in the following paragraphs.

Figure 21.18: Metrics collection

- Metrics collection. It will receive metrics from various sources and process them into a format that can be ingested into our data ingestion system.
- Data ingestion system. It will receive metrics from Metrics collection and store them in our data storage system.
- Data storage system. It will store metrics from Data ingestion system and provide them to Alerting system and Visualization system.
- Alerting system. It will receive metrics from Data storage system and send alerts to various messaging systems.
- Visualization system. It will receive metrics from Data storage system and provide them to various visualization systems.

Data ingestion system

Data ingestion system is a system that receives metrics from Metrics collection and stores them in our data storage system. We discuss the data ingestion system in the following paragraphs.

Figure 21.19: Data ingestion system

Data storage system

Data storage system is a system that stores metrics from Data ingestion system and provides them to Alerting system and Visualization system. We discuss the data storage system in the following paragraphs.

Figure 21.20: Data storage system

Alerting system

Alerting system is a system that receives metrics from Data storage system and sends alerts to various messaging systems. We discuss the alerting system in the following paragraphs.

Figure 21.21: Alerting system

Now that we have discussed the high-level design, let's discuss the data model and query language.

Data model

The data model for our system is very important. It affects how our system is implemented. We want to make our data model simple, flexible, and extensible. We discuss the data model in the following paragraphs.

Metrics name

We need to be able to identify a metric by its name. We need to be able to query for all the metrics that have a certain prefix. We also need to be able to support hierarchical metrics. We discuss the metrics name in the following paragraphs.

Figure 21.22: Metrics name

Key-value pairs

We need to be able to identify a metric by its key-value pairs. We need to be able to query for all the metrics that have a certain key-value pair. We discuss the key-value pairs in the following paragraphs.

Figure 21.23: Key-value pairs

Metrics collection

For metrics collection, we need to collect metrics from various sources. We discuss some of the options we have for collecting metrics in the following paragraphs.

Figure 21.24: Metrics collection

- Metrics collection. It will receive metrics from various sources and process them into a format that can be ingested into our data ingestion system.
- Data ingestion system. It will receive metrics from Metrics collection and store them in our data storage system.
- Data storage system. It will store metrics from Data ingestion system and provide them to Alerting system and Visualization system.
- Alerting system. It will receive metrics from Data storage system and send alerts to various messaging systems.
- Visualization system. It will receive metrics from Data storage system and provide them to various visualization systems.

Data ingestion system

Data ingestion system is a system that receives metrics from Metrics collection and stores them in our data storage system. We discuss the data ingestion system in the following paragraphs.

Figure 21.25: Data ingestion system

Data storage system

Data storage system is a system that stores metrics from Data ingestion system and provides them to Alerting system and Visualization system. We discuss the data storage system in the following paragraphs.

Figure 21.26: Data storage system

Alerting system

Alerting system is a system that receives metrics from Data storage system and sends alerts to various messaging systems. We discuss the alerting system in the following paragraphs.

Figure 21.27: Alerting system

Now that we have discussed the high-level design, let's discuss the data model and query language.

Data model

The data model for our system is very important. It affects how our system is implemented. We want to make our data model simple, flexible, and extensible. We discuss the data model in the following paragraphs.

Metrics name

We need to be able to identify a metric by its name. We need to be able to query for all the metrics that have a certain prefix. We also need to be able to support hierarchical metrics. We discuss the metrics name in the following paragraphs.

Figure 21.28: Metrics name

Key-value pairs

We need to be able to identify a metric by its key-value pairs. We need to be able to query for all the metrics that have a certain key-value pair. We discuss the key-value pairs in the following paragraphs.

Figure 21.29: Key-value pairs

Metrics collection

For metrics collection, we need to collect metrics from various sources. We discuss some of the options we have for collecting metrics in the following paragraphs.

Figure 21.30: Metrics collection

- Metrics collection. It will receive metrics from various sources and process them into a format that can be ingested into our data ingestion system.
- Data ingestion system. It will receive metrics from Metrics collection and store them in our data storage system.
- Data storage system. It will store metrics from Data ingestion system and provide them to Alerting system and Visualization system.
- Alerting system. It will receive metrics from Data storage system and send alerts to various messaging systems.
- Visualization system. It will receive metrics from Data storage system and provide them to various visualization systems.

Data ingestion system

Data ingestion system is a system that receives metrics from Metrics collection and stores them in our data storage system. We discuss the data ingestion system in the following paragraphs.

Figure 21.31: Data ingestion system

Data storage system

Data storage system is a system that stores metrics from Data ingestion system and provides them to Alerting system and Visualization system. We discuss the data storage system in the following paragraphs.

Figure 21.32: Data storage system

Alerting system

Alerting system is a system that receives metrics from Data storage system and sends alerts to various messaging systems. We discuss the alerting system in the following paragraphs.

Figure 21.33: Alerting system

Now that we have discussed the high-level design, let's discuss the data model and query language.

Data model

The data model for our system is very important. It affects how our system is implemented. We want to make our data model simple, flexible, and extensible. We discuss the data model in the following paragraphs.

Metrics name

We need to be able to identify a metric by its name. We need to be able to query for all the metrics that have a certain prefix. We also need to be able to support hierarchical metrics. We discuss the metrics name in the following paragraphs.

Figure 21.34: Metrics name

Key-value pairs

We need to be able to identify a metric by its key-value pairs. We need to be able to query for all the metrics that have a certain key-value pair. We discuss the key-value pairs in the following paragraphs.

Figure 21.35: Key-value pairs

Metrics collection

For metrics collection, we need to collect metrics from various sources. We discuss some of the options we have for collecting metrics in the following paragraphs.

Figure 21.36: Metrics collection

- Metrics collection. It will receive metrics from various sources and process them into a format that can be ingested into our data ingestion system.
- Data ingestion system. It will receive metrics from Metrics collection and store them in our data storage system.
- Data storage system. It will store metrics from Data ingestion system and provide them to Alerting system and Visualization system.
- Alerting system. It will receive metrics from Data storage system and send alerts to various messaging systems.
- Visualization system. It will receive metrics from Data storage system and provide them to various visualization systems.

Data ingestion system

Data ingestion system is a system that receives metrics from Metrics collection and stores them in our data storage system. We discuss the data ingestion system in the following paragraphs.

Figure 21.37: Data ingestion system

Data storage system

Data storage system is a system that stores metrics from Data ingestion system and provides them to Alerting system and Visualization system. We discuss the data storage system in the following paragraphs.

Figure 21.38: Data storage system

Alerting system

Alerting system is a system that receives metrics from Data storage system and sends alerts to various messaging systems. We discuss the alerting system in the following paragraphs.

Figure 21.39: Alerting system

Now that we have discussed the high-level design, let's discuss the data model and query language.

Data model

The data model for our system is very important. It affects how our system is implemented. We want to make our data model simple, flexible, and extensible. We discuss the data model in the following paragraphs.

Metrics name

We need to be able to identify a metric by its name. We need to be able to query for all the metrics that have a certain prefix. We also need to be able to support hierarchical metrics. We discuss the metrics name in the following paragraphs.

Figure 21.40: Metrics name

Key-value pairs

We need to be able to identify a metric by its key-value pairs. We need to be able to query for all the metrics that have a certain key-value pair. We discuss the key-value pairs in the following paragraphs.

Figure 21.41: Key-value pairs

Metrics collection

For metrics collection, we need to collect metrics from various sources. We discuss some of the options we have for collecting metrics in the following paragraphs.

Figure 21.42: Metrics collection

- Metrics collection. It will receive metrics from various sources and process them into a format that can be ingested into our data ingestion system.
- Data ingestion system. It will receive metrics from Metrics collection and store them in our data storage system.
- Data storage system. It will store metrics from Data ingestion system and provide them to Alerting system and Visualization system.
- Alerting system. It will receive metrics from Data storage system and send alerts to various messaging systems.
- Visualization system. It will receive metrics from Data storage system and provide them to various visualization systems.

Data ingestion system

Data ingestion system is a system that receives metrics from Metrics collection and stores them in our data storage system. We discuss the data ingestion system in the following paragraphs.

Figure 21.43: Data ingestion system

Data storage system

Data storage system is a system that stores metrics from Data ingestion system and provides them to Alerting system and Visualization system. We discuss the data storage system in the following paragraphs.

Figure 21.44: Data storage system

Alerting system

Alerting system is a system that receives metrics from Data storage system and sends alerts to various messaging systems. We discuss the alerting system in the following paragraphs.

Figure 21.45: Alerting system

Now that we have discussed the high-level design, let's discuss the data model and query language.

Data model

The data model for our system is very important. It affects how our system is implemented. We want to make our data model simple, flexible, and extensible. We discuss the data model in the following paragraphs.

Metrics name

We need to be able to identify a metric by its name. We need to be able to query for all the metrics that have a certain prefix. We also need to be able to support hierarchical metrics. We discuss the metrics name in the following paragraphs.

Figure 21.46: Metrics name

Key-value pairs

We need to be able to identify a metric by its key-value pairs. We need to be able to query for all the metrics that have a certain key-value pair. We discuss the key-value pairs in the following paragraphs.

Figure 21.47: Key-value pairs

Metrics collection

For metrics collection, we need to collect metrics from various sources. We discuss some of the options we have for collecting metrics in the following paragraphs.

Figure 21.48: Metrics collection

- Metrics collection. It will receive metrics from various sources and process them into a format that can be ingested into our data ingestion system.
- Data ingestion system. It will receive metrics from Metrics collection and store them in our data storage system.
- Data storage system. It will store metrics from Data ingestion system and provide them to Alerting system and Visualization system.
- Alerting system. It will receive metrics from Data storage system and send alerts to various messaging systems.
- Visualization system. It will receive metrics from Data storage system and provide them to various visualization systems.

Data ingestion system

Data ingestion system is a system that receives metrics from Metrics collection and stores them in our data storage system. We discuss the data ingestion system in the following paragraphs.

Figure 21.49: Data ingestion system

Data storage system

Data storage system is a system that stores metrics from Data ingestion system and provides them to Alerting system and Visualization system. We discuss the data storage system in the following paragraphs.

Figure 21.50: Data storage system

Alerting system

Alerting system is a system that receives metrics from Data storage system and sends alerts to various messaging systems. We discuss the alerting system in the following paragraphs.

Figure 21.51: Alerting system

Now that we have discussed the high-level design, let's discuss the data model and query language.

Data model

The data model for our system is very important. It affects how our system is implemented. We want to make our data model simple, flexible, and extensible. We discuss the data model in the following paragraphs.

Metrics name

We need to be able to identify a metric by its name. We need to be able to query for all the metrics that have a certain prefix. We also need to be able to support hierarchical metrics. We discuss the metrics name in the following paragraphs.

Figure 21.52: Metrics name

Key-value pairs

We need to be able to identify a metric by its key-value pairs. We need to be able to query for all the metrics that have a certain key-value pair. We discuss the key-value pairs in the following paragraphs.

Figure 21.53: Key-value pairs

Metrics collection

For metrics collection, we need to collect metrics from various sources. We discuss some of the options we have for collecting metrics in the following paragraphs.

Figure 21.54: Metrics collection

- Metrics collection. It will receive metrics from various sources and process them into a format that can be ingested into our data ingestion system.
- Data ingestion system. It will receive metrics from Metrics collection and store them in our data storage system.
- Data storage system. It will store metrics from Data ingestion system and provide them to Alerting system and Visualization system.
- Alerting system. It will receive metrics from Data storage system and send alerts to various messaging systems.
- Visualization system. It will receive metrics from Data storage system and provide them to various visualization systems.

Data ingestion system

Data ingestion system is a system that receives metrics from Metrics collection and stores them in our data storage system. We discuss the data ingestion system in the following paragraphs.

Figure 21.55: Data ingestion system

Data storage system

Data storage system is a system that stores metrics from Data ingestion system and provides them to Alerting system and Visualization system. We discuss the data storage system in the following paragraphs.

Figure 21.56: Data storage system

Alerting system

Alerting system is a system that receives metrics from Data storage system and sends alerts to various messaging systems. We discuss the alerting system in the following paragraphs.

Figure 21.57: Alerting system

Now that we have discussed the high-level design, let's discuss the data model and query language.

Data model

The data model for our system is very important. It affects how our system is implemented. We want to make our data model simple, flexible, and extensible. We discuss the data model in the following paragraphs.

Metrics name

We need to be able to identify a metric by its name. We need to be able to query for all the metrics that have a certain prefix. We also need to be able to support hierarchical metrics. We discuss the metrics name in the following paragraphs.

Figure 21.58: Metrics name

Key-value pairs

We need to be able to identify a metric by its key-value pairs. We need to be able to query for all the metrics that have a certain key-value pair. We discuss the key-value pairs in the following paragraphs.

Figure 21.59: Key-value pairs

Metrics collection

For metrics collection, we need to collect metrics from various sources. We discuss some of the options we have for collecting metrics in the following paragraphs.

Figure 21.60: Metrics collection

- Metrics collection. It will receive metrics from various sources and process them into a format that can be ingested into our data ingestion system.
- Data ingestion system. It will receive metrics from Metrics collection and store them in our data storage system.
- Data storage system. It will store metrics from Data ingestion system and provide them to Alerting system and Visualization system.
- Alerting system. It will receive metrics from Data storage system and send alerts to various messaging systems.
- Visualization system. It will receive metrics from Data storage system and provide them to various visualization systems.

Data ingestion system

Data ingestion system is a system that receives metrics from Metrics collection and stores them in our data storage system. We discuss the data ingestion system in the following paragraphs.

Figure 21.61: Data ingestion system

Data storage system

Data storage system is a system that stores metrics from Data ingestion system and provides them to Alerting system and Visualization system. We discuss the data storage system in the following paragraphs.

23
Hotel Reservation System

Hotel reservation system is a system that enables customers to book hotel rooms online. It allows customers to view available rooms, make reservations, and manage their bookings.

Step 1 - Understand the Problem and Establish Design Scope

Step 1 - Understand the Problem and Establish Design Scope

For a hotel reservation system, it takes months for a customer to book a room. We can, for example, book a room 1 year in advance or 1 day in advance. Requirements are different for different use cases. We need to ask questions to understand the requirements. Here are some sample questions:

candidate: Can customers make a reservation for more than one room?
interviewer: Yes, customers can book up to 5 rooms.
candidate: How many people can a room accommodate?
interviewer: Each room can accommodate up to 3 people.
candidate: How long can customers book a room in advance?
interviewer: Customers can book a room up to 1 year in advance.
candidate: Can customers cancel a reservation?
interviewer: Yes, customers can cancel a reservation.
candidate: Do customers need to pay for a reservation?
interviewer: Customers can pay for a reservation when they check in.
candidate: Can customers modify a reservation?
interviewer: Yes, customers can modify a reservation.
candidate: What types of payment do you support?
interviewer: We support credit card payment.
candidate: Do we need to send notifications to customers?
interviewer: Yes, we need to send notifications to customers when they make a reservation, modify a reservation, or cancel a reservation.

Functional requirements

- Customers can book a room.
- Customers can view available rooms.
- Customers can modify a reservation.
- Customers can cancel a reservation.
- Customers can pay for a reservation when they check in.
- Customers can receive notifications when they make a reservation, modify a reservation, or cancel a reservation.

Non-functional requirements

- Support 10 hotels. Assume 1000 rooms per hotel. Assume 2 people per room.
- Support 5 million users.
- Support 1 million reservations per month.
- Support 100 transactions per second.

Back of the envelope estimation

- Assume 10% of the rooms are reserved on a daily basis.
- Assume 1 million reservations per month.
- Assume 10 million users.
- Assume 100 transactions per second.
- Assume 10% of the users are active on a daily basis.
- Assume 10 million users. Assume 1 million reservations per month. Assume 10% of the users are active on a daily basis. Assume 100 transactions per second.

Step 2 - Propose High-Level Design and Get Buy-In

- Hotel Service
- Payment Service
- Notification Service

API design

We use RESTful API to expose the service to client. For our service, we will use JSON to transfer data between client and server.

Hotel Service

POST /v1/hotels	Add a hotel.
GET /v1/hotels/{id}	Get details about a hotel.
POST /v1/hotels/{id}/rates	Add a rate for a hotel.
GET /v1/hotels/{id}/rates?startDate=yyyyMMdd&endDate=yyyyMMdd	Get available rates for a hotel.
POST /v1/hotels/{id}/reservations	Reserve a room.
GET /v1/reservations?firstName=xxx&lastName=xxx	Get reservations.
DELETE /v1/reservations/{id}	Cancel a reservation.
PUT /v1/reservations/{id}	Update a reservation.

Rate Service API

POST /v1/rates	Add a rate.
PUT /v1/rates/{id}	Update a rate.
GET /v1/rates?hotelId=xxx&roomTypeId=xxx	Get rates.

Payment Service API

POST /v1/payments	Execute a payment.
PUT /v1/payments/{id}	Update a payment.
GET /v1/payments?orderId=xxx	Get payment status by order.

Notice that we use plural nouns in API URLs. For example, we use /v1/hotels instead of /v1/hotel. Here is an example of the request and response for the API to add a hotel:

```
POST /v1/hotels
{
  "name": "Radisson",
  "address": "1 Main St",
  "city": "Chicago",
  "country": "USA",
  "type": "Resort",
  "rating": 5
}
```

```
HTTP/1.1 201 Created
Location: /v1/hotels/123
```

Data model

Data access patterns are important when we design a data model. We need to ask questions such as what kind of queries, how much data, and how often the data will be accessed.

Hotel Table

Hotel Rate Table

Room Type Table

High-level design

The high-level design is shown in the diagram below. Hotel, rate, and room type services are responsible for CRUD operations for hotel, rate, and room type. The payment service is responsible for payment processing. The reservation service is responsible for room reservation.

Hotel, rate, and room type services are responsible for CRUD operations for hotel, rate, and room type. The payment service is responsible for payment processing. The reservation service is responsible for room reservation. Notification service is responsible for sending notifications to customers. Hotel, rate, and room type databases are relational databases. Payment and reservation databases are also relational databases. We use a message queue to decouple the payment service from the reservation service. The payment service publishes payment events to the message queue, and the reservation service subscribes to payment events from the message queue. We use third-party payment gateways to process payments. We use third-party email gateways and third-party SMS gateways to send notifications to customers.

Hotel and Rate Service

Hotel and rate services are responsible for CRUD operations for hotel and rate. We need to ask questions such as what kind of queries, how much data, and how often the data will be accessed.

- Hotel and rate services are read-heavy services. Hotel and rate data are not updated frequently. We can use a cache to improve read performance.
- Hotel and rate services are internal services. We use gRPC to expose the services to other services. gRPC is a high-performance, open-source universal RPC framework. It is a binary protocol and is more efficient than JSON. It supports bidirectional streaming and multiplexing. It supports strong typing and code generation. It supports authentication and encryption. It supports load balancing and service discovery.
- Hotel and rate services are stateless services. We can scale them horizontally.
- Hotel and rate services are CRUD services. We can use a relational database to store hotel and rate data.

Here is the diagram for the hotel and rate services:

Hotel and rate services are read-heavy services. We can use a cache to improve read performance. We can use a cache-aside pattern to load data into the cache. We can use a write-through cache to update the cache when we update the database.

Reservation Service

Reservation service is responsible for room reservation. We need to ask questions such as what kind of queries, how much data, and how often the data will be accessed.

- Reservation service is a read-heavy service. We can use a cache to improve read performance.
- Reservation service is a stateless service. We can scale it horizontally.
- Reservation service is a CRUD service. We can use a relational database to store reservation data.
- Reservation service is a transaction service. We need to ensure data consistency.

Here is the diagram for the reservation service:

Reservation service is a read-heavy service. We can use a cache to improve read performance. We can use a cache-aside pattern to load data into the cache. We can use a write-through cache to update the cache when we update the database.

Payment Service

Payment service is responsible for payment processing. We need to ask questions such as what kind of queries, how much data, and how often the data will be accessed.

- Payment service is a write-heavy service. We can use a relational database to store payment data.
- Payment service is a stateless service. We can scale it horizontally.
- Payment service is a transaction service. We need to ensure data consistency.
- Payment service is a third-party service. We need to use third-party payment gateways to process payments.
- Payment service is a CRUD service. We can use a relational database to store payment data.
- Payment service is a third-party service. We need to use third-party payment gateways to process payments.

Here is the diagram for the payment service:

Payment service is a write-heavy service. We can use a relational database to store payment data. We need to use third-party payment gateways to process payments.

Notification Service

Notification service is responsible for sending notifications to customers. We need to ask questions such as what kind of queries, how much data, and how often the data will be accessed.

- Notification service is a write-heavy service. We can use a message queue to store notification data.
- Notification service is a stateless service. We can scale it horizontally.
- Notification service is a third-party service. We need to use third-party email gateways and third-party SMS gateways to send notifications to customers.

Here is the diagram for the notification service:

Notification service is a write-heavy service. We can use a message queue to store notification data. We need to use third-party email gateways and third-party SMS gateways to send notifications to customers.

Step 3 - Design Deep Dive

- Scalability
- Hotel search
- Concurrency
- Caching
- Monitoring
- Logging
- Security
- Rate limiting

Scalability

Hotel and rate services are read-heavy services. We can use a cache to improve read performance. We can use a cache-aside pattern to load data into the cache. We can use a write-through cache to update the cache when we update the database.

Reservation service is a read-heavy service. We can use a cache to improve read performance. We can use a cache-aside pattern to load data into the cache. We can use a write-through cache to update the cache when we update the database.

Payment service is a write-heavy service. We can use a relational database to store payment data. We need to use third-party payment gateways to process payments.

Notification service is a write-heavy service. We can use a message queue to store notification data. We need to use third-party email gateways and third-party SMS gateways to send notifications to customers.

Hotel search

Hotel search is a read-heavy service. We can use a cache to improve read performance. We can use a cache-aside pattern to load data into the cache. We can use a write-through cache to update the cache when we update the database.

Hotel search is a stateless service. We can scale it horizontally.

Hotel search is a CRUD service. We can use a relational database to store hotel data.

Hotel search is a third-party service. We need to use third-party search engines to process hotel search.

Hotel search is a third-party service. We need to use third-party search engines to process hotel search.

Concurrency

Concurrency is a key issue in a hotel reservation system. We need to ensure data consistency. We need to use third-party payment gateways to process payments.

Concurrency is a key issue in a hotel reservation system. We need to ensure data consistency. We need to use third-party payment gateways to process payments.

Caching

Caching is a key issue in a hotel reservation system. We need to ensure data consistency. We need to use third-party payment gateways to process payments.

Caching is a key issue in a hotel reservation system. We need to ensure data consistency. We need to use third-party payment gateways to process payments.

Monitoring

Monitoring is a key issue in a hotel reservation system. We need to ensure data consistency. We need to use third-party payment gateways to process payments.

Monitoring is a key issue in a hotel reservation system. We need to ensure data consistency. We need to use third-party payment gateways to process payments.

Logging

Logging is a key issue in a hotel reservation system. We need to ensure data consistency. We need to use third-party payment gateways to process payments.

Logging is a key issue in a hotel reservation system. We need to ensure data consistency. We need to use third-party payment gateways to process payments.

Security

Security is a key issue in a hotel reservation system. We need to ensure data consistency. We need to use third-party payment gateways to process payments.

Security is a key issue in a hotel reservation system. We need to ensure data consistency. We need to use third-party payment gateways to process payments.

Rate limiting

Rate limiting is a key issue in a hotel reservation system. We need to ensure data consistency. We need to use third-party payment gateways to process payments.

Rate limiting is a key issue in a hotel reservation system. We need to ensure data consistency. We need to use third-party payment gateways to process payments.

Step 4 - Wrap Up

- Hotel Service
- Payment Service
- Notification Service

Chapter Summary

Reference Material

- Hotel reservation system
- Design a hotel reservation system
- Hotel reservation system
- Hotel reservation system
-

24
Distributed Email Service

Email is one of the most complex and widely used internet services. In this chapter, we design a scalable email service that supports millions of users and billions of messages per day. We focus on the core flows, storage, and key trade-offs that make such a system reliable and secure.

Step 1 - Understand the Problem and Establish Design Scope

Email systems can mean many things to different people. For this chapter, we focus on a system that supports the core flows of sending, receiving, storing, and retrieving messages for millions of users.

Functional requirements

Sending: Users can send emails to one or more recipients, including CC and BCC.

Receiving: Users receive emails from other users and from external mail servers.

Storage: Emails must be stored reliably and durably, with support for folders or labels, metadata, and attachments.

Retrieval: Users can fetch their mail from multiple clients and devices, with consistent mailbox state.

Scale: The system must support millions of users and billions of messages per day.

Delivery guarantees: Messages should be delivered reliably with retries and eventual consistency. Duplicate delivery is acceptable if handled gracefully.

Non-functional requirements

Latency: Inbox loads and message retrieval should be fast enough for a responsive user experience.

Availability: The system should remain available even during partial failures, with graceful degradation rather than total outage.

Security and privacy: Support TLS for transport, authentication, spam and abuse protection, and access control. Protect user data at rest and in transit.

Back-of-the-envelope estimation

Assume 1 billion users, 10 billion messages per day, average message size of 50 KB, average attachment size of 500 KB, and 30 days of retention.

- Ingress: 10 billion messages × 50 KB ≈ 500 TB per day
- Attachment storage: 10 billion × 500 KB ≈ 5 PB per day
- Retention: 30 days ≈ 150 PB
- Metadata: 10 billion × 1 KB ≈ 10 TB per day
- Peak SMTP connections: 100 million concurrent connections
- Peak API requests: 100 million requests per second

Step 2 - Propose High-Level Design and Get Buy-In

Email is a store-and-forward system. Mail servers accept messages, queue them, retry when needed, and deliver them to the recipient's mailbox. We'll start with the protocols users and servers actually use, then sketch the core components that move and store mail.

Email protocols

SMTP: Simple Mail Transfer Protocol is the standard protocol for sending mail between servers and from clients to servers. It handles the envelope, recipients, and message transfer.

IMAP: Internet Message Access Protocol lets clients read and manage mail stored on a server. It supports folders, flags, and partial sync across devices.

POP3: Post Office Protocol version 3 is a simpler protocol that downloads mail to a single client. It's less flexible but still widely supported.

Traditional mail servers

Traditional mail servers store messages in local mailbox formats and deliver them directly between organizations. This worked well for small networks but scales poorly when you need high availability, spam filtering, and compliance.

Modern email architecture

Modern email systems separate ingress, storage, and retrieval into distinct services. This allows each component to scale independently and fail without taking the entire system down.

Core components

- Ingress: Accepts incoming mail from other servers and from clients, applies spam and policy checks, and queues for delivery.
- Storage: Persists mailboxes, folders, metadata, and attachments. Supports multi-region replication.
- Retrieval: Serves IMAP and API requests so clients can read, search, and sync mail.

Ingress

Ingress is the entry point for all incoming mail. It must accept connections from the internet, validate senders, filter spam and abuse, and queue messages for delivery. Ingress should be stateless and horizontally scalable so it can handle connection floods and DDoS attempts.

Key responsibilities

- Accept SMTP connections from other mail servers.
- Validate recipients and domains.
- Run spam, malware, and policy checks.
- Queue messages for durable delivery.
- Rate limit by sender, IP, and domain.

Storage

Storage must persist mailboxes, folders, metadata, and attachments reliably and durably. Most systems use a combination of a metadata store and blob storage, with multi-region replication for disaster recovery.

Key responsibilities

- Persist mailbox state and message metadata.
- Store attachments as blobs with deduplication.
- Replicate across regions for durability.
- Support quotas and retention policies.

Retrieval

Retrieval serves IMAP and API requests so clients can read, search, and sync mail. It must support multi-device access with consistent mailbox state.

Key responsibilities

- Serve IMAP connections for mail clients.
- Provide API access for web and mobile clients.
- Support search and filtering.
- Manage mailbox state across devices.

Step 3 - Deep Dive

Ingress: SMTP at scale

Ingress must accept connections from the internet, validate senders, and queue messages for delivery. Spam and abuse are the biggest challenges here. Most systems use layered checks before accepting mail.

Key mechanisms

- Connection limits per IP and per domain.
- Rate limits by sender and domain.
- Reputation checks against DNS blocklists.
- Spam and malware scanning.
- Policy checks for attachment types and sizes.
- Queue with retry and exponential backoff.

Storage: metadata and blobs

Most systems split storage into a metadata store and blob storage. Metadata holds mailboxes, folders, and message pointers. Blobs hold the actual message bodies and attachments.

Key design choices

- Metadata: Relational database or key-value store for mailbox state and message pointers.
- Blobs: Object storage for raw message bodies and attachments.
- Deduplication: Hash attachments to avoid storing duplicates.
- Quotas: Track per-user storage and enforce limits.
- Retention: Support TTL and legal hold policies.

Retrieval: IMAP and API

Retrieval serves mail clients that read, search, and sync mail. IMAP is the standard protocol for mail clients, and APIs support web and mobile apps.

Key operations

- Fetch mailbox state and message lists.
- Search by sender, recipient, subject, and date.
- Mark read/unread, flag, move, and delete.
- Push notifications for new mail.

Spam and abuse

Spam and abuse are the biggest challenges for any email system. Layered checks at ingress reject most bad mail before it enters the system.

Key mechanisms

- Connection limits per IP and per domain.
- Rate limits by sender and domain.
- Reputation checks against DNS blocklists.
- Spam and malware scanning.
- Policy checks for attachment types and sizes.
- Queue with retry and exponential backoff.

Delivery guarantees

Email delivery is store-and-forward with retries. Messages may be delayed but should not be lost. Duplicate delivery is acceptable if handled gracefully.

Key mechanisms

- Queue with durable storage.
- Retry with exponential backoff.
- Dead letter queue for permanent failures.
- Idempotent delivery to avoid duplicates.

Step 4 - Wrap Up

In this chapter, we proposed a scalable email service that supports millions of users and billions of messages per day. We covered the core protocols, ingress for accepting mail, storage for persisting messages, and retrieval for serving mail clients. We also discussed spam and abuse, delivery guarantees, and key trade-offs.

If time allows, we can discuss:

- Full-text search across mailboxes.
- Multi-region active-active deployment.
- Encryption at rest and in transit.
- Compliance and legal hold.
- Monitoring and alerting for delivery failures.

Chapter summary

References Materials

- RFC 5321 - Simple Mail Transfer Protocol
- RFC 5322 - Internet Message Format
- RFC 3501 - Internet Message Access Protocol
- RFC 1939 - Post Office Protocol - Version 3
- RFC 8314 - Cleartext Considerations and Implicit TLS for Email
- RFC 7208 - Sender Policy Framework (SPF)
- RFC 6376 - DomainKeys Identified Mail (DKIM) Signatures
- RFC 7489 - Domain-based Message Authentication, Reporting, and Conformance (DMARC)
- RFC 8461 - SMTP MTA Strict Transport Security (MTA-STS)
- RFC 8460 - SMTP TLS Reporting
- Amazon SES Developer Guide
- Google Workspace Admin Help - Email delivery basics
- Microsoft Exchange Server Architecture
- Postfix Architecture Overview
- Dovecot Architecture
- Stripe Engineering - How we scale our email infrastructure
- Cloudflare Blog - How we secure email at scale
- Mailgun Engineering - Lessons learned from building a mail platform
- Twilio SendGrid Engineering - Deliverability at scale
- Google Cloud Pub/Sub Documentation
- Amazon S3 Developer Guide
- Apache Kafka Documentation
- Redis Streams Documentation
- Prometheus Monitoring Documentation
- Grafana Dashboards Documentation