Java™ System Design

INTERVIEW CHALLENGER

ACE THE JAVA INTERVIEWS BY MASTERING THE FUNDAMENTALS OF SYSTEM DESIGN

RAFAEL CHINELATO DEL NERO

Java Systems Design Interview Challenger

Pass Java System Design interviews by understanding the fundamentals of a cloud system's components. Don't try to learn all the tools, focus on concepts instead.

Rafael Chinelato del Nero

This book is for sale at
http://leanpub.com/java_systems_design_challenger

This version was published on 2024-05-01

Contents

Preface

The Systems Design interview is a difficult one. You need the knowledge and the strategy. You need to know how to handle it. Otherwise, the chances to pass in such an interview are very low. Even if you are knowledgeble.

Understanding the fundamentals of a cloud system is a great step to be better equipped for the interview. That's what we will see in this book, also, we will explore strategies to pass in this interview more easily.

Motivation

The Systems Design interview is very popular nowadays. This happened because the big companies are doing that with candidates and as it commonly happens, all the other companies are doing the same for interviews.

Therefore, studying Systems Design makes you a better software engineer and also prepares you for a job interview.

Another motivation to write this book is to show you that failing in this interview doesn't mean we are bad software engineers. It means that we don't know how the game is played and to beat any game, we need to know the rules.

In this book I will show you the rules of the game and also the fundamental knowledge you need to be able to learn technologies much faster. To be a fast learner in the area of software development is crucial because new technologies are launched every day.

Since our brains have many neuroconnections when we have the basic knowledge of any subject, our brains make the connection

much faster. That's why mastering the fundamentals is powerful. We make the connections very fast because all technologies are actually based on fundamentals.

Objectives

The main objective of this book is to give you a solid foundational knowledge about Systems Design so you can learn any new technology faster.

Programmers are not what the market wants anymore. Nowadays, companies are looking for software engineers, this means, a developer who can analyze a problem and come up with effective solutions.

The ability of articulating a solution, creating good quality code, talking to coworkers, preparing documentation with architecture solutions, designing database, create performant and scalable systems are the skills companies are looking for.

Some years ago, knowing a Java framework was enough for the market but the scenario has changed. That's why the most powerful skill any software can have is the ability to learn, the ability to be a software engineer.

If you study a little bit the life of one of the greatest engineers on history, Leonardo da Vinci, you will realize that one of his most powerful traits was his curiosity. He was curious on understanding even why we yawn when we are sleepy.

As Da Vinci, we need to have a similar mindset, we need to be curious about technologies and at the same time focused. Because we don't really have time to learn random things, we want to spend quality time with our families and deliver high-quality work.

Therefore, keep that in mind, be curious to learn, never think you know everything that will never happen. We have to accept that in

our profession more than others we will learn our whole life. We must see that as a good thing because if we are learning something new it means we are alive. If we stop learning we are dying, there is nothing else to do in life.

Keep this mantra, Never Stop Learning.

Scope and Limitations

We will explore the most important fundamentals of Systems Design. We will see:

- Concepts and technologies to make services communicate with the client
- Technologies to make Microservices scalable
- Manipulating data in the database
- Ways to debug and monitor cloud services
- How to make Microservices resilient to failure
- How to make Microservices performant
- How to make data transfering fast for special cases
- How to make Microservices more secure
- Techniques and technologies to handle Big Data

Notice that it's impossible to explain all of those topics in dept in just one book. For most of those technologies andconcepts we would be able to create a whole book.

However, remember that it's impossible to know all of those technologies in depth, we just need to know approximately 20% so we can solve 80% of the problems. Don't think you must know all of those technologies because that will never happen.

We must keep in mind that it's useless to know all of those technologies in depth, instead, we must know enough of them so we can solve important problems, so we can create something useful.

Otherwise, there is no point on learning a lot and applying nothing. Always learn something with an action in mind.

Target Audience

Systems Design is a complex topic because it involves understanding systems as a whole. Therefore, this book was created mostly for intermediate or senior developers.

However, this book can be consumed by developers who are begginers also because we explore a lot of fundamental knowledge that can be learned without prior knowledge. Even if the Systems Design interview is not something a begginer developer will do, it's good to know their principles.

Special Features

Unlike the other Systems Design books, this book is more focused on doing Systems Design with Java. Therefore, you will see Java as much as possible in the book.

Java has a huge ecosystem and has many technologies to use in the cloud. The frameworks Spring, Quarkus, Micronaut and others are highly focused on the cloud. Some of the Systems Design principles will mention those technologies since in most companies they will be used.

However, behind those technologies there are the Systems Design principles and that's what we will focus on.

Systems Design Interview - How it works

During the Systems Design interview we will need to design a system with a vague information such as “design Amazon”, or “design Netflix”, or “design Discord”. To accomplish that we need to have the basic knowledge of how systems work.

Other than that, we must assume we will design the application in the simplest way in our questions because usually in a Systems Design interview we have only 45 minutes.

We also have to understand what we are designing, we can’t start designing the system in a complicated way without making the right questions, that will make us fail right away. Instead, we make strategic questions to make the System design simple and always ask the interviewer for feedback to understand if the design is correct.

To have a general understanding of this process, let’s see how we would design some applications we use in our day-to-day lives.

We also need the fundamental knowledge of the parts of a system. In the book we will explore first the fundamentals and then in the last chapters we will see how e can explore famous systems.

Author's Credentials

Rafael del Nero is a Java Champion with more than 15 years of experience as a software engineer. He moved from Brazil Sao Paulo to Dublin in Ireland in 2017 to work as a software engineer in Europe. He worked mostly with Java, Spring and Jakarta EE with cloud projects.

He worked with highly-skilled software engineers so he could learn key techniques to help you to become also a highly-skilled software engineer. Rafael is non-stop learner and his focus is to help Java developers to earn more and work less so they enjoy high-quality time with their families.

1 - Introduction

The Systems design interview became very popular in the market nowadays. After the big techs Google, Meta, Microsoft, Amazon and other companies started using it, all the other companies are doing the same. That's why knowing how the systems design interview works is key for you to get a good job.

How Does a Systems Design Interview Work?

The interviewee will give you a vague question such as design Youtube or design Instagram, then it's up to you to make the right questions to explain how you will build the system.

It's very important to assume what the system should be like, therefore, every question you make assume the simplest. Remember that on those interviews you will have something around 1 hour. Accept that you won't be able to create a very robust system in this but it will be enough to get your knowledge tested on the subject.

Depending on what you decide to use on the Systems Design interview, you might be challenged by the interviewer to explain why you made that technology choice. If you talk about a technology, it's better for you to pick one that you know very well.

Differently than the algorithms interview, the answer you give to build a system is subjective. This means you will have to know why you choose the technologies and what are the trade offs.

Key Considerations for a Systems Design Interview

In a Systems Design interview, there are several key considerations that you should focus on. These considerations demonstrate your understanding of building scalable, performant, reliable, and secure systems. Here are some important design considerations:

Scalability: How will the system handle a growing number of users, data, and traffic? Consider horizontal scaling, load balancing, and partitioning strategies.

Performance: How will the system handle high volumes of concurrent requests? Consider optimizing database queries, caching strategies, content delivery networks (CDNs), and efficient use of resources.

Reliability: How will the system ensure high availability and fault tolerance? Consider redundancy, replication, failover mechanisms, and disaster recovery strategies.

Security: How will the system protect user data, prevent unauthorized access, and mitigate potential vulnerabilities? Consider authentication, authorization, encryption, secure API endpoints, and protection against common attacks like cross-site scripting (XSS) and SQL injection.

Data storage and retrieval: How will user-generated content be stored and retrieved efficiently? Consider database choices, data partitioning, indexing strategies, and caching mechanisms.

Data consistency: How will the system maintain data consistency across different components and replicas? Consider consistency models, distributed transactions, and conflict resolution strategies.

Communication and messaging: How will different components of the system communicate with each other? Consider message

queues, pub-sub systems, event-driven architectures, and communication protocols.

Third-party integrations: How will the system integrate with external services or APIs? Consider authentication, rate limiting, and handling failures or API changes.

Analytics and monitoring: How will the system collect and analyze user activity data? Consider logging, monitoring, and analytics tools to gain insights into system performance and user behavior.

Cost optimization: How will the system optimize costs associated with infrastructure, storage, and data transfer? Consider efficient resource utilization, auto-scaling, and leveraging cloud services.

These considerations will help you demonstrate a holistic understanding of system design principles and showcase your ability to design robust and scalable solutions. Remember to prioritize these considerations based on the specific requirements and constraints of the system you are designing.

Key Characteristics You Should Know for a Systems Design Interview

You will need to make the necessary questions to understand what you will want to prioritise for your system.

Usually, the system you will design will be a cloud system. Nowadays, you have to keep in mind all of the following system characteristics:

Availability: Availability refers to the ability of a system or service to remain operational and accessible without interruptions. High availability is important for critical systems used in healthcare, finance, and transportation, and is achieved through redundancy, failover mechanisms, and other strategies that minimize downtime.

Latency: refers to the time delay between when an action is performed and when it is actually executed or completed. It can be caused by various factors such as slow internet connection or processing time of a device. For example, when you click on a link on a website, it may take a few seconds for the page to load because of latency. In general, lower latency means faster response times, while higher latency means slower response times.

Throughput: refers to the amount of data that can be transferred over a network or processed by a device in a given amount of time. It is often measured in bits or bytes per second.

Higher throughput means that more data can be transferred or processed in a shorter amount of time, which can result in faster performance.

For example, with low latency and high internet speed the higher network throughput which means that files can be downloaded or uploaded faster. While a higher processing throughput means that a computer can perform more tasks in a given time period.

Redundancy: means having backup systems or components to ensure that if one fails, another can take over. It's done to increase reliability and minimize downtime. For example, in a hard drive, data may be stored in multiple locations. In a network, redundant systems are set up so that if one fails, another can take over its tasks.

Consistency: means maintaining a uniform behavior or performance over time. It involves following a set standard or pattern. It can be seen in data accuracy across multiple systems, quality and performance standards in products, and a consistent tone in communication. Consistency establishes trust and reliability in a product, system, or service.

Components You Should Know for a Systems Design Interview

To create performant, scalable, resilient cloud systems, you need to know about the basic components. You might be asked about those components during the interview. You need to at the very least know what those components do. Let's see some examples:

Load balancer: helps distribute network traffic across multiple servers to prevent overloading and improve performance. It works as a mediator between the client and server, ensuring requests are evenly distributed. Load balancers detect server failures and redirect traffic to healthy servers. They're commonly used in web applications and distributed systems to optimize resource utilization and improve availability.

Proxy: is a server or software that acts as an intermediary between a client and a server. It receives requests from clients and forwards them to the server, then sends the response back to the client. Proxies can be used to filter requests, cache responses, or provide anonymity for clients. They're commonly used in web applications to improve performance, security, and privacy.

Cache: is a temporary storage location that stores frequently accessed data to reduce the time it takes to access it. It serves as a quick lookup for commonly used data, so that it doesn't have to be fetched from the original source every time it's requested. Caches can be found in various forms such as web browser caches, CPU caches, and disk-based caches. They're commonly used to improve performance and reduce latency in computer systems.

Rate limiting: is a technique used to control the rate of traffic sent or received by a network or application. It sets a limit on the number of requests that can be made within a certain time frame to prevent overloading and ensure stable performance. Rate limiting can be used to prevent abuse, protect against attacks, and optimize

resource utilization. It's commonly used in web applications, APIs, and network devices.

Leader election: is a process used in distributed systems to select a leader from a group of nodes. The leader is responsible for making decisions and coordinating actions among the nodes. If the leader fails or goes offline, the other nodes can initiate a new leader election to select a new leader. Leader election is commonly used in systems like Apache Zookeeper and etcd to ensure that a single node is responsible for managing the system at any given time.

Technologies You Should Know for a Systems Design Interview

When you talk about technologies in a systems design interview it might be your time to shine. The more you know a certain technology, also the more you can explain why this technology will be efficient, which will likely impress the interviewer.

Usually, the interviewer will make you questions about the technology you chose, then it's a bonus for you if you can really nail the explanation. If you can give an acceptable explanation without much depth that is acceptable as well because what the interviewer is really testing is your skills to build a system with the correct requirements.

Example of technologies:

Zookeeper: is a distributed open-source software system that is used for coordinating and managing large distributed systems. It provides a centralized service for maintaining configuration information, naming, providing distributed synchronization, and group services. Zookeeper allows multiple servers to work together as a unified system and helps to ensure that data is consistent and up-to-date across all servers. It is commonly used in distributed systems

such as Hadoop and Kafka to manage configuration information and coordinate tasks among multiple nodes.

etcd: is a distributed key-value store used for shared configuration and service discovery. It's open-source and developed by CoreOS. It's based on the Raft consensus algorithm, which ensures data consistency across all servers. It's commonly used in container orchestration systems like Kubernetes.

Ngnx: (pronounced "engine-x") is an open-source web server software used for serving web pages and applications. It's designed to be fast, lightweight, and scalable. Ngnx can handle high-traffic websites and is often used as a reverse proxy, load balancer, or HTTP cache. It's commonly used in modern web architectures to improve performance and reliability.

Redis: is an open-source, in-memory data structure store that is used as a database, cache, and message broker. It supports a wide range of data structures including strings, hashes, lists, and sets. Redis is designed for high performance and scalability and is often used in real-time applications such as gaming, messaging, and analytics. It's commonly used in modern web architectures to speed up data access and improve application performance

Kafka: is an open-source distributed streaming platform used for building real-time data pipelines and streaming applications. It's designed to be fast, scalable, and fault-tolerant. Kafka allows for the processing of large amounts of data in real time and provides a messaging system for communication between different parts of an application. It's commonly used in modern data architectures to process and analyze large streams of data.

Summary

The systems design interview is not an easy one, you need to know how to make the right questions, not over-engineer your

system, and how to build it effectively. You need to know about technologies and how it works. It's useful to practice with a friend a systems design interview and also watch some examples of those interviews to get a grasp of how it works. Ultimately, it's crucial to do as many real systems design interviews as possible.

2 - Fundamentals of Availability

Knowing about fundamental concepts of availability is crucial for systems design. But high availability comes with a cost. One important point we must remember when designing a system is that every system design decision will have a trade-off. There is no silver bullet, but there are technologies that will solve specific problems.

Therefore, if the system you are designing does critical operations with payment, for example, you need to make systems design decisions that will make it highly available.

Why Systems Availability is Important?

Imagine you are using Slack in your company, and you need even more this tool to communicate because you are working remotely. Then the service goes down suddenly for many hours, and you cannot communicate with your team. That would likely delay your work delivery.

Imagine if the Amazon e-commerce website goes down for 1 hour. How many millions would they lose? Also, how would their reputation be harmed in this process?

Stripe is another excellent example; they are a SaaS (Software as a Service) that performs payments for companies. If Stripe goes down, their customers will lose money.

Let's list the main reasons why availability is essential:

Business Continuity: Availability is vital for businesses as it ensures uninterrupted operations by keeping systems and services operational. This high availability is essential for maintaining productivity, meeting customer demands, and preventing financial losses due to downtime.

Customer Satisfaction: Customers expect reliable and accessible services. When systems are consistently available, customers can access products, services, or information without inconvenience or delays. High availability enhances the customer experience, builds trust, and fosters customer satisfaction.

Revenue Generation: Many businesses use their systems to generate revenue. For example, E-commerce platforms, online services, and digital marketplaces require continuous availability to process transactions and serve customers. Downtime can lead to lost sales opportunities and revenue. High availability ensures that revenue-generating systems are accessible, minimizing the risk of financial losses.

Data Integrity and Security: Availability is closely tied to data integrity and security. Systems with high availability typically incorporate measures such as data replication, backups, and disaster recovery mechanisms. These safeguards protect against data loss, ensure data integrity, and help maintain the confidentiality of sensitive information.

Compliance and Legal Requirements: Businesses must adhere to regulations and legal requirements regarding system availability and data protection in specific industries. Non-compliance can lead to penalties, legal consequences, and reputational damage. Organizations can meet these obligations and demonstrate their commitment to data privacy and security by maintaining availability.

Competitive Advantage: Availability can be a competitive differentiator. Businesses that provide reliable and accessible services have an advantage over competitors experiencing frequent down-

time or performance issues. Customers are more likely to choose and remain loyal to businesses that offer dependable and available systems.

Availability Legal Agreement

When we develop web systems for other customers, we need to agree on the application's availability. Customers need the application to be available for their businesses. That's why we have to create agreement documents to build trust with our customers, and, of course, if what we wrote in the document is not met, there will be penalties. Therefore, it's essential to make sure we provide the agreed availability.

Now let's see how those documents and metrics work.

SLA – Service Level Agreement

SLA stands for Service Level Agreement. It is a contractual agreement between a service provider and a customer that defines the expected level of service and outlines the responsibilities of both parties. Companies use SLAs in various business arrangements, mainly IT services, cloud computing, telecommunications, and outsourcing.

The purpose of an SLA is to establish clear expectations and provide a framework for measuring and managing the performance of services. It typically includes the following components:

Service Description: The SLA begins by defining the services we provide. Therefore, we must include a detailed description of the service's scope, features, and functionality. It outlines what the service provider will deliver to the customer.

Service Levels and Metrics: The SLA specifies the expected service performance levels, such as availability, response time, resolution time, or throughput. It also defines the metrics and measurement methods we will use to assess and monitor the service performance.

Responsibilities and Roles: The SLA clearly outlines the responsibilities of both the service provider and the customer. It defines who is accountable for what tasks, actions, and deliverables. This agreement ensures clarity and alignment in terms of roles and expectations.

Performance Targets: The SLA sets specific performance or service level targets (SLTs) that the service provider must meet. For example, an SLA might state that the service should be available 99.9% of the time or that response times should be within a specific timeframe.

Reporting and Monitoring: The SLA specifies the reporting and monitoring mechanisms we will use to track the service provider's performance. It may include requirements for regular reporting, key performance indicators (KPIs), performance dashboards, or other monitoring tools.

Escalation and Dispute Resolution: The SLA includes handling issues, escalations, and dispute resolution procedures. It outlines the steps to follow when there are service disruptions, breaches of the SLA, or disagreements between the parties.

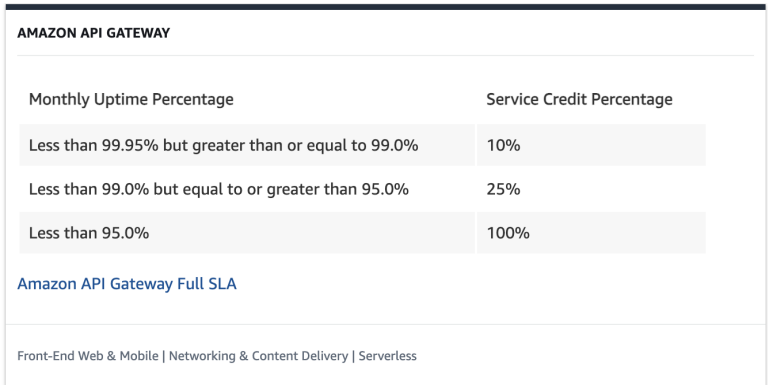
Remedies and Penalties: The SLA may outline remedies and penalties that we will impose if the service provider fails to meet the agreed-upon service levels. These could include financial penalties, service credits, or other forms of compensation.

The SLA document is a legal agreement showing our applications will be available. If the company does not meet the SLA, there are consequences. The company providing the service might pay a fee.

We can memorize some of the information in this document. Knowing the general idea will help when we need to provide a

cloud solution and agree on how available the application will be.

Let's have a look at the SLA from Amazon API Gateway: Let's list the main reasons why availability is essential:



AMAZON API GATEWAY

Monthly Uptime Percentage	Service Credit Percentage
Less than 99.95% but greater than or equal to 99.0%	10%
Less than 99.0% but equal to or greater than 95.0%	25%
Less than 95.0%	100%

Amazon API Gateway Full SLA

Front-End Web & Mobile | Networking & Content Delivery | Serverless

Figure 1. Amazon API Gateway SLA

You can check it further in the following link: https://aws.amazon.com/api-gateway/sla

SLO – Service Level Objective

SLO stands for Service Level Objective. It is a target or goal set by a service provider to define the desired level of performance or quality for a particular service. SLOs are typically defined within the context of a Service Level Agreement (SLA) and provide specific, measurable metrics that the service provider aims to achieve.

Here are some critical aspects of SLOs:

Performance Metrics: SLOs are defined using specific performance metrics relevant to the service provided. These metrics can vary depending on the nature of the service, but common

examples include availability, response time, throughput, error rate, or latency. The selection of appropriate metrics depends on the service's objectives and the customer's expectations.

Quantifiable Targets: SLOs set quantifiable targets or thresholds for the defined performance metrics. For example, an SLO for a web application's response time might state that 95% of requests and we should get a response within 200 milliseconds. These targets provide a clear benchmark against which we can measure the service provider's performance can be measured.

Measurable and Monitorable: SLOs should be measurable and monitorable, meaning there should be mechanisms to collect relevant data and track performance against the defined targets. This often involves using monitoring tools, analytics, or performance measurement systems to monitor the service's performance continuously.

Alignment with Customer Expectations: We design SLOs to align the service provider's performance with the customer's expectations. They consider the needs, priorities, and requirements of the customer, ensuring that the service meets or exceeds their desired level of performance. SLOs help set clear expectations and provide a basis for evaluating the service provider's performance.

Continuous Improvement: SLOs are not static but evolve. As technology advances, customer expectations change, and business requirements evolve, we must review and update SLOs to reflect new targets and goals. Continuous monitoring and analysis of performance data help identify areas for improvement and drive ongoing optimization of the service.

SLI – Service Level Indicator

SLI stands for Service Level Indicator. It is a quantitative measurement or metric that provides objective data about the performance

or quality of a service. SLIs are used to monitor and evaluate the actual performance of a service and are often defined within the context of a Service Level Agreement (SLA) or Service Level Objective (SLO).

SLIs are essential for objectively measuring and evaluating the performance of a service. They provide a quantifiable and objective basis for assessing the service's quality, ensuring that it meets the agreed-upon targets and aligns with customer expectations. By monitoring SLIs, service providers can identify areas for improvement, make data-driven decisions, and continuously enhance the service's performance.

Nines of Availability

One of the most important metric in the SLA is the uptime agreement. If we are talking about critical systems, we need to have high availability. The gold standard for the market is to have 5 nines, this means only 5.26 minutes of downtime during the year.

However, if we don't need the application to be available all the time, three nines will be more than sufficient. There is a trade-off when choosing to go for five nines, therefore, it's better to make choose a suitable availability for the system.

As mentioned, those are the most common availability percentages applications in the market use:

Availability	Percentage Downtime
99.9% ("three nines")	8.77 hours
99.99% ("four nines")	52.60 minutes
99.999% ("five nines")	5.26 minutes

Now let's see all the other nines in the following image:

Availability %	Downtime per year[note 1]	Downtime per quarter	Downtime per month	Downtime per week	Downtime per day (24 hours)
90% ("one nine")	**36.53 days**	**9.13 days**	**73.05 hours**	**16.80 hours**	**2.40 hours**
95% ("one nine five")	18.26 days	4.56 days	36.53 hours	8.40 hours	1.20 hours
97% ("one nine seven")	10.96 days	2.74 days	21.92 hours	5.04 hours	43.20 minutes
98% ("one nine eight")	7.31 days	43.86 hours	14.61 hours	3.36 hours	28.80 minutes
99% ("two nines")	**3.65 days**	**21.9 hours**	**7.31 hours**	**1.68 hours**	**14.40 minutes**
99.5% ("two nines five")	1.83 days	10.98 hours	3.65 hours	50.40 minutes	7.20 minutes
99.8% ("two nines eight")	17.53 hours	4.38 hours	87.66 minutes	20.16 minutes	2.88 minutes
99.9% ("three nines")	**8.77 hours**	**2.19 hours**	**43.83 minutes**	**10.08 minutes**	**1.44 minutes**
99.95% ("three nines five")	4.38 hours	65.7 minutes	21.92 minutes	5.04 minutes	43.20 seconds
99.99% ("four nines")	**52.60 minutes**	**13.15 minutes**	**4.38 minutes**	**1.01 minutes**	**8.64 seconds**
99.995% ("four nines five")	26.30 minutes	6.57 minutes	2.19 minutes	30.24 seconds	4.32 seconds
99.999% ("five nines")	**5.26 minutes**	**1.31 minutes**	**26.30 seconds**	**6.05 seconds**	**864.00 milliseconds**
99.9999% ("six nines")	**31.56 seconds**	**7.89 seconds**	**2.63 seconds**	**604.80 milliseconds**	**86.40 milliseconds**
99.99999% ("seven nines")	**3.16 seconds**	**0.79 seconds**	**262.98 milliseconds**	**60.48 milliseconds**	**8.64 milliseconds**
99.999999% ("eight nines")	**315.58 milliseconds**	**78.89 milliseconds**	**26.30 milliseconds**	**6.05 milliseconds**	**864.00 microseconds**
99.9999999% ("nine nines")	**31.56 milliseconds**	**7.89 milliseconds**	**2.63 milliseconds**	**604.80 microseconds**	**86.40 microseconds**

Figure 2. Nines of Availability

Source from Wikipedia in the following link: https://en.wikipedia.org/wiki/High_-availability

Redundancy

Redundancy refers to the duplication of critical components, systems, or processes within a larger system or organization. It involves having backup or redundant elements in place to ensure continued operation and mitigate the impact of failures or disruptions. The purpose of redundancy is to enhance reliability, fault tolerance, and resilience.

Here are a few key points about redundancy:

Backup and Failover: Redundancy often involves having duplicate components or systems that can take over the functionality of the primary component or system in the event of a failure. For example, in a computer network, redundant network switches or servers can act as backup devices and seamlessly take over if the primary ones

fail.

Fault Tolerance: Redundancy enhances fault tolerance by minimizing the impact of failures. When redundant components or systems are in place, the failure of one element does not lead to a complete system failure. Instead, the redundant element can step in and maintain system operation, ensuring continuity and minimizing downtime.

Reliability and Resilience: Redundancy improves the overall reliability and resilience of a system. By having redundant elements, the system becomes less vulnerable to single points of failure. It can withstand failures, disruptions, or malfunctions without significant impact, ensuring that critical functions and services remain available.

Data Redundancy: In the context of data storage and backup, redundancy refers to storing multiple copies of data across different storage devices or locations. This ensures that if one copy is lost or corrupted, there are still additional copies available for recovery. Redundant data storage helps protect against data loss and increases data reliability.

Cost and Complexity: Implementing redundancy often comes with additional costs and complexity. Redundant components or systems require additional resources, such as hardware, infrastructure, or maintenance. Managing and synchronizing redundant elements can also introduce complexity in system design and maintenance.

Redundancy is a commonly used strategy in various domains, including information technology, telecommunications, power distribution, transportation, and disaster recovery. It helps organizations ensure continuous operation, minimize downtime, and increase the reliability and resilience of their systems.

Passive redundancy

Imagine you are a student who needs to submit a very important assignment online. The submission system is crucial for you to complete your task successfully. In this case, passive redundancy is like having a backup plan to ensure that your assignment gets submitted even if something goes wrong with the primary submission system.

Here's how it works:

Primary Submission System: The primary submission system is the main system you use to submit your assignment. It's the system you rely on and interact with initially.

Backup Submission System: The backup submission system is like a spare or backup option that is ready to take over if the primary system fails or has a problem. It's not actively processing submissions, but it's there as a standby.

Standby Mode: The backup submission system is in a standby mode, patiently waiting for any issues to occur with the primary system. It's not actively processing submissions or doing anything until it's needed.

Failover Process: If the primary submission system encounters a problem or fails, a failover process is triggered. This process involves activating the backup system and diverting your assignment submission to it.

Continued Service: Once the failover process is completed, the backup submission system takes over seamlessly, allowing you to submit your assignment without any disruption. From your perspective as a student, it appears as if nothing went wrong, and your assignment gets successfully submitted.

Recovery Time: In case of a failure or problem with the primary system, the recovery time is the time it takes for the backup system to become active and start accepting submissions. During this time,

you may experience a brief delay or interruption, but the backup system ensures that the service is restored as quickly as possible.

Active Redundancy

Imagine that you have an assignment submission system that utilizes active redundancy to ensure a smooth and uninterrupted submission process. Here's how it works:

Active Components: In active redundancy, there are multiple active components or systems working simultaneously to handle the assignment submissions. These components are actively processing and delivering services concurrently.

Load Balancing: The active components distribute the workload evenly among themselves through load balancing. This ensures that each component shares the processing load and can handle a portion of the submissions efficiently.

Seamless Failover: If one of the active components fails or experiences an issue, the workload is automatically shifted to the remaining active components without any interruption or impact on the submission process. The failover process is seamless and transparent to you as a student.

High Availability: With active redundancy, the submission system maintains high availability, meaning it remains operational and accessible even if one or more active components encounter problems. The remaining active components continue to process submissions, ensuring that the service remains uninterrupted.

Increased Performance: Active redundancy can also improve the performance of the system. By distributing the workload among multiple active components, the overall processing capacity and throughput are increased. This allows for faster and more efficient handling of assignment submissions.

Redundancy Monitoring: In an actively redundant system, there is continuous monitoring of the health and performance of the active components. This monitoring allows for proactive detection of any issues or failures, enabling quick remediation and failover processes to ensure uninterrupted service.

In this scenario, active redundancy in the assignment submission system ensures that multiple components are actively working together to handle submissions. If one component fails, the others seamlessly take over the workload to ensure continuous availability and smooth operation. This redundancy configuration enhances reliability, fault tolerance, and performance, providing you with a reliable and efficient submission experience.

What is the Cost of Highly-Available Systems?

The cost of high-availability can vary depending on several factors, such as the specific requirements of the system, the level of redundancy needed, the technologies utilized, and the scale of the infrastructure. Here are some aspects to consider when evaluating the cost of high-availability:

Hardware and Infrastructure: High-availability often requires redundant hardware components and infrastructure to ensure continuous operation. This may include redundant servers, storage systems, network equipment, and power supplies. The cost of these components can vary significantly based on the required capacity, performance, and reliability.

Software and Licensing: High-availability solutions may involve specialized software, such as load balancers, clustering software, or failover mechanisms. The cost of these software licenses can add to the overall expense.

Network Connectivity: Redundant network connectivity is crucial for high-availability systems. It may involve multiple internet service providers (ISPs), redundant network links, or even geographically diverse data centers. The cost will depend on the chosen providers, the required bandwidth, and the complexity of the network setup.

Data Replication and Storage: High-availability often involves replicating data across multiple systems or data centers to ensure redundancy and eliminate single points of failure. The cost will depend on the amount of data, the replication method used (synchronous or asynchronous), and the storage infrastructure required.

Monitoring and Management: Effective high-availability systems require robust monitoring and management tools to detect issues, perform failovers, and maintain system health. The cost may include licensing fees for monitoring software, employing dedicated staff, or outsourcing these tasks to a managed service provider.

Staffing and Expertise: Maintaining a high-availability system usually requires skilled staff with expertise in system administration, network management, and troubleshooting. The cost will depend on the size of the team required and their level of expertise.

Downtime Impact: It's important to consider the potential cost of downtime and the impact it may have on your business. High-availability solutions aim to minimize downtime, which can help avoid financial losses due to service disruptions, loss of productivity, or damage to the organization's reputation.

Summary

Availability is a crucial characteristic for systems design. That will make a big difference in the technologies we will use. To have a highly available system will require a lot more resources, it will be

more expensive and the complexity will be higher. Let's review the key points of availability:

- The more available is the system, the more expensive and complex it will be. Nines of availability mean the availability percentage in the year for an application.
- 3 Nines have the yearly downtime of 8.77 hours.
- 4 Nines have the yearly downtime 52.60 minutes.
- 5 Nines have the yearly downtime of 5.26 minutes.
- SLA means Service Level Agreement and is an agreement document between the service provider and the customer.
- SLO means Service Level Objective which contains the objectives within the - SLA document with real metrics of availability.
- SLI means Service Level Indicator which contains the real data of how the available is the service and how well it is performing.
- To make an application highly-available it's necessary to have redundancy. - This means that services have to be replicated, also the load balancer. Otherwise, we will have one point of failure.

Redundancy involves having backup or duplicate systems, components, or resources in place.

- Redundancy ensures the availability, continuity, and reliability of the system or process.
- Redundancy can be applied to various areas, such as data storage, power supply, and networking.
- Redundancy is commonly used in critical systems, industries, and infrastructure where failure can have significant consequences.
- Redundancy requires careful planning, design, and maintenance to balance the costs and benefits effectively.

- Redundancy is part of a broader strategy for achieving fault tolerance, resilience, and system reliability.
- Active redundancy refers to the implementation of duplicate components or systems that operate simultaneously, with one actively serving as a backup to immediately take over in the event of a failure in order to ensure continuous operation and minimize downtime.
- Passive redundancy refers to the presence of duplicate components or systems that are not actively operating but can be activated manually or automatically when needed as a backup in case of failure.

3 - Client Server Model

In a systems design interview, we need to know how the client-server model work to use the right components when designing the system. That's why we will explore further how this model works since it's by far the most used in applications. It's also the foundation of how computers communicate with each other on the internet is the client server model.

Client-server Model Diagram

Before diving through the main concepts of the client-model diagram, let's have a look on how this works. The client-server model is the way our interactions with websites work. First we will write a URL in the browser and then there will be a request to query the IP (Internet Protocol) address with the DNS (Domain Name System)

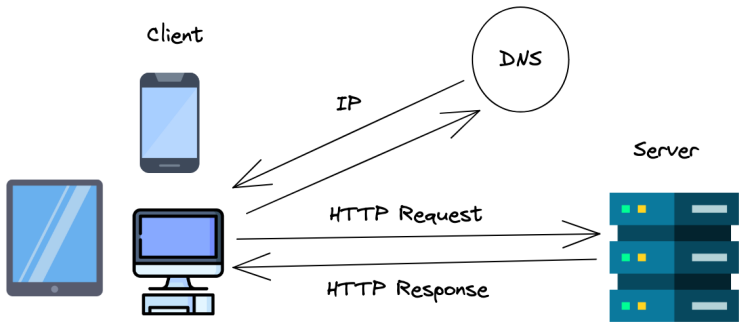


Figure 3. Client Server Model

What is IP?

IP (Internet Protocol) is the address of a computer on the internet. Each device has its own unique IP or address so it's possible to exchange data on the internet.

There are two versions of the IP protocol currently in use: IPv4 and IPv6. IPv4 is the older version and uses a 32-bit address format, while IPv6 is the newer version and uses a 128-bit address format. IPv6 was developed to provide a larger address space to accommodate the growing number of devices connected to the internet.

You can test IP addresses using a tool called "netcat". To use netcat, you can run the command on the terminal:

```
1  nc -l 8081
```

Then use the following command in another terminal to connect to the IP address and port:

```
1  nc 127.0.0.1 8081
```

What is a port?

A port is a communication endpoint that allows different processes or services running on a device to send and receive data over a network. Ports are identified by a number between 0 and 65535 and are used to differentiate between different types of network traffic.

Each port is associated with a specific process or service, and when data is sent to a device, the receiving device uses the port number

to route the data to the corresponding process or service. Firewalls and network security devices often use port numbers to control access to specific services or protocols.

A port is always used with an IP so we can make an analogy with real physical addresses. The IP will be the street address and the port will be the location number.

What is DNS?

The DNS is a legible name for humans to access websites. Any website name you are able to access on the internet is a DNS, for example, google.com, javachallengers.com are DNS. We could use directly the IP address to access a website but probably no one would remember that.

That's why a there will be a DNS query to find out the IP address of a website or server because that's the only way the server will ultimately understand.

To convert a DNS to an IP address, you can use the "dig" command on a Unix-based operating system. You can give a try the following:

```
1  dig javachallengers.com
```

The output will return the IP:

```
1  javachallengers.com.    300 IN  A   172.67.169.6
2  javachallengers.com.    300 IN  A   104.21.27.59
```

When you send data over the internet using HTTP, you need to specify the port number to be used in addition to the IP address. The IP address is like a street name, and the port number is like a building number.

What is HTTP?

HTTP, or Hypertext Transfer Protocol, is the protocol used for transferring data over the internet. It defines how information is formatted and transmitted between web servers and web browsers, allowing for the retrieval and display of web pages and other online resources. HTTP can be seen as a common language or pattern that computers will understand each other in a network.

When you type a URL into your web browser, the browser sends an HTTP request to the web server hosting the website. The server responds with an HTTP response, which contains the requested resource, such as an HTML document or image file.

HTTP is a stateless protocol, which means that it doesn't keep track of previous requests or responses. This makes it efficient for transferring small amounts of data, but it can become slow and inefficient for larger data transfers.

HTTP is the foundation of the World Wide Web and is used by millions of websites and web applications every day.

What Applications use the Client-server Model?

The client-server model is a widely used architecture for designing and implementing networked applications, and it is used in a wide range of applications across different industries. Here are some examples of applications that use the client-server model:

Web browsers and web servers: When you browse the internet, your web browser acts as a client, sending requests to web servers to retrieve web pages and other resources. The web servers then respond with the requested content, which is displayed in your browser.

Email clients and SMTP servers: Email clients such as Microsoft Outlook and Apple Mail use the Simple Mail Transfer Protocol (SMTP) to send and receive email messages. The email client acts as the client, and the SMTP server acts as the server.

File transfer applications: Applications such as FTP (File Transfer Protocol) and SFTP (Secure File Transfer Protocol) use the client-server model to transfer files between computers.

Database applications: Database management systems (DBMS) use the client-server model to allow multiple users to access and manipulate a database simultaneously. The DBMS acts as the server, and the client applications send requests to the server to retrieve or modify data.

Online games: Many online games use the client-server model to manage the game state and enable players to interact with each other in real-time.

These are just a few examples of the many applications that use the client-server model.

Summary

The client-server model is present everywhere on systems. Understanding the fundamentals of this model is crucial for a systems design interview. The vast majority of the systems you will design in such interview will have the client server model. Sometimes the interviewer will ask you about IP, DNS, HTTP. That's why it's fundamental to understand those concepts.

4 - Protocols

A protocol is a way for computers to communicate. An analogy to that could be the language we use to communicate with each other. A computer will understand only a set of pre-defined protocols.

There are many network protocols. Let's see the most important ones in depth and the others that could be more crucial to develop systems. We will explore it in a more shallow way.

IP – Internet protocol

IP stands for Internet Protocol. It's a set of rules that govern how data is transmitted over the internet. Every device connecting to the internet has an IP address, a unique numerical identifier assigned to it. This address allows data to be sent over the internet from one device to another.

We represent IP addresses in either IPv4 or IPv6 format. IPv4 addresses consist of four numbers separated by periods, while IPv6 addresses are more extended and use hexadecimal notation.

The IP protocol also governs how data is broken up into packets, how packets are addressed and routed, and how they are reassembled into the original data at their destination. All internet communication, including email, web browsing, and file transfers, rely on the IP protocol.

TCP

Transmission Control Protocol is a communication protocol that reliably transmits data over the internet. It breaks up data into smaller packets, establishes a connection between devices, detects and corrects errors, and regulates data flow between devices to ensure that data is transmitted and received correctly, without errors or loss. It's like a postal service that guarantees that letters are sent and received correctly and can fix any mistakes during delivery.

HTTP

HTTP (Hypertext Transfer Protocol) is the protocol used for transferring data over the World Wide Web. It allows your web browser to communicate with web servers and retrieve web pages, images, videos, and other resources. When you type a URL into your web browser, it sends an HTTP request to the web server asking for the resource you want to access.

The server then responds with an HTTP response containing the requested resource or an error message if the resource cannot be found or any other error happens. HTTP is a client-server protocol that involves communication between a client (your web browser) and a server (the web server hosting the resource you want to access).

HTTP Request Header

When sending an HTTP request to a server, we need to inform what is the URL or endpoint we want to access in the server and the request method to send information. If it's a more complicated

request, then we might need to authenticate, pass a specific type of information.

In the example, the client is sending a GET request to the server to retrieve the file "example.html." The request header contains the request method (GET), the requested resource ("/example.html"), the HTTP version (HTTP/1.1), and the hostname of the server being requested (www.example.com). Additionally, the request header includes information about the client making the request, such as the user agent (Firefox web browser on a Windows 10 computer) and the types of content it can accept.

```
GET /example.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:\
89.0) Gecko/20100101 Firefox/89.0
Accept: text/html,application/xhtml+xml,application/xml;q\
=0.9,image/webp,*/*;q=0.8
HTTP/1.1 200 OK
Date: Sat, 24 Jul 2023 16:30:00 GMT
Server: Apache/2.4.6 (Red Hat Enterprise Linux)
Content-Type: text/html; charset=UTF-8
Content-Length: 1234
Connection: close
```

Let's see now an example of an HTTP request sending an authorization Bearer token:

```
GET /API/v1/users/123 HTTP/1.1
Host: example.com
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ\
9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF
0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_ad
Qssw5c
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:\
```

```
89.0) Gecko/20100101 Firefox/89.0
Accept: application/json
```

The Authorization field contains the authorization token required to access the resource. In this case, it's a Bearer authentication scheme, and the token is a JSON Web Token (JWT) that includes the user ID, name, and timestamp.

Additionally, the request header includes information about the client making the request, such as the user agent (Firefox web browser on a Windows 10 computer) and the types of content it can accept (JSON in this case).

The HTTP response header includes the HTTP version (HTTP/1.1), the date and time the response was sent, the server software being used (Apache/2.4.6 on Red Hat Enterprise Linux), the type of content being returned (JSON), the character encoding (UTF-8), the length of the content being returned (256 bytes), and the authorization token required to access the resource. The response body would contain the requested data in JSON format.

HTTP Methods

HTTP defines a set of methods, also known as verbs or actions, that describe the desired action to be performed on a resource identified by a URI (Uniform Resource Identifier).

Here are the most commonly used HTTP methods:

GET: retrieves information or data from the server specified by the URL. A practical example is to get information from a user of a system.

POST: sends data to the server to create or update a resource. Even though it can be used for creation or update, it's more common to use the post method to create a register in the database, for example.

The post method is also non-idempotent, which means that every request you make to the server might create many registers.

PUT: sends data to the server to replace an existing resource or create a new one. The PUT method is more common to be used to update a register in the database. Another important point is that the put method is idempotent, meaning we can send as many requests as we want, and the results will always be the same when sending the same data. Also, the put method will send the whole data even if it wasn't changed. Obviously, the code implementation on the server side has also to be congruent with the PUT method to be idempotent.

DELETE: deletes the specified resource from the server. We should use it to delete a register in the database, for example. The DELETE method is also considered idempotent because, let's suppose we are deleting the user with the id of '1', and then after this user is deleted, we can invoke the same endpoint as many times as we want, and the response will be the same, nothing will be deleted because the user is already deleted.

PATCH: sends data to the server to update or modify an existing resource. It's used for partial updates; this means that only the data that was changed will be processed.

HEAD: retrieves the header information associated with a resource without actually retrieving the resource itself.

OPTIONS: returns the HTTP methods, headers, and other options supported by the server for a specified resource.

HTTP Response Codes

There are some HTTP response codes that you will see very often when developing software. Let's see some of them:

200 - OK: This response code means that the request was successful and the server has returned the requested data.

201 – Created: This response code means that the server has successfully created a new resource as a result of the request.

202 – Accepted: This response code means that the request has been accepted for processing, but the processing has not yet been completed.

204 – No Content: This response code means that the server has successfully processed the request, but there is no data to return to the client.

301 – Moved Permanently: This response code means that the requested resource has been moved permanently to a new location, and the client should update its records accordingly.

302 – Found: This response code means that the requested resource has been temporarily moved to a new location, and the client should use the new location for future requests.

304 – Not Modified: This response code means that the requested resource has not been modified since the last time the client requested it, and the server is instructing the client to use its cached version of the resource.

401 – Unauthorized: This response status code indicates that the requested resource is restricted and requires authentication. In other words, the server is telling the client that they need to provide valid credentials (such as username and password) in order to access the requested resource. This status code is often used for web pages that require users to log in before they can access the content. If the client does not provide valid credentials, the server will return a 401 Unauthorized status code, indicating that access is denied.

403 – Forbidden: This response code means that the client does not have permission to access the requested resource.

404 – Not Found: This response code means that the requested resource could not be found on the server.

500 – Internal Server Error: Those are more serious errors from an HTTP response. It's a very generic error and it might be related to the case that the server is down. The response codes from 500 onwards are related to infrastructure such as network or any other issue from the server where the application is deployed and it shouldn't happen very often.

If you want to see all the HTTP response codes and have fun, I recommend the following website: https://http.cat

Other Internet Protocols

Simple Mail Transfer Protocol (SMTP) – used to send and receive email messages.

Post Office Protocol (POP) – used to retrieve email messages from a mail server.

Internet Message Access Protocol (IMAP) – used to access email messages stored on a mail server.

Transmission Control Protocol (TCP) – used to establish and maintain connections between devices on a network.

User Datagram Protocol (UDP) – used for low-latency and loss-tolerating connections between applications.

Domain Name System (DNS) – used to translate domain names to IP addresses.

Dynamic Host Configuration Protocol (DHCP) – used to assign IP addresses to devices on a network automatically.

Simple Network Management Protocol (SNMP) – used to manage and monitor network devices.

Summary

Understanding the basics of network protocols is crucial to ace the systems design interview. Before using the significant amount of technologies we have nowadays, understanding the fundamentals will dramatically accelerate your learning.

IP, HTTP, and TCP are the most used network protocols. As a software engineer, you must master HTTP since we always use it with web applications.

5 - Effective API Design

Besides algorithms and Systems design interviews in many companies, we might also have a API (Application Programming Interface) design interview. The knowledge of APIs is crucial in the cloud era because Microservices communicate via API.

Therefore, developer teams will often have to align an API contract so that services communicate effectively. Without further ado, let's explore the main concepts of an API further!

What is an API?

An API, or Application Programming Interface, is like a messenger, allowing different software applications to talk to each other and share information. It serves as a point of connection between two systems, enabling them to interact and exchange data in a standardized way.

Think of it this way: Imagine you want to order food from a restaurant. The menu acts as an API. It provides a list of available dishes, their descriptions, and prices. As the client, you can select what you want to order and communicate your choices to the restaurant. As the server, the restaurant receives your request, prepares the food accordingly, and delivers it back to you. In this analogy, the menu is the API that defines how you and the restaurant can communicate and exchange information effectively.

Similarly, in software, an API defines a set of rules and network protocols that specify how different applications can interact, what data can be exchanged, and what operations can be performed. It simplifies the development process by providing a standardized

interface, allowing developers to integrate different systems seamlessly and leverage the functionalities provided by external services or libraries.

What is REST?

REST (Representational State Transfer) is an architectural style rather than a protocol or a standard. It provides a set of principles and constraints for designing networked applications and APIs.

REST is based on a few key concepts:

Resources: In REST, a resource is an entity or piece of information that can be identified by a unique URL, often referred to as a "resource endpoint" or "resource URI." For example, a resource could be a user profile, a product listing, or a blog post.

HTTP Verbs: RESTful APIs use HTTP verbs (GET, POST, PUT, DELETE, etc.) to perform operations on resources. Each HTTP verb has a specific meaning: GET is used to retrieve data, POST is used to create new resources, PUT is used to update existing resources, and DELETE is used to remove resources.

Stateless Communication: REST is stateless, meaning that each request from a client to a server should contain all the necessary information for the server to understand and process the request. The server does not store any client-specific information between requests. Therefore, it allows for scalability and simplicity in the architecture.

Uniform Interface: REST promotes a uniform interface between clients and servers, meaning the API follows consistent conventions. It includes using standardized HTTP methods, URLs to identify resources, and standard media types (such as JSON or XML) for data representation.

Hypermedia as the Engine of Application State (HATEOAS):

REST APIs can include hyperlinks within the responses, allowing clients to discover and navigate related resources. HATEOAS enables a self-descriptive API where clients can dynamically explore and interact with the available resources and actions.

RESTful APIs follow these principles to provide a scalable, stateless, and uniform approach to building web services. It emphasizes simplicity, interoperability, and leveraging the existing standards and infrastructure of the web, making it widely adopted and well-suited for distributed systems.

API Most Used Protocols

We use the following protocols when communicating one service to the other via API. Even though we use the most REST APIs, we have many different APIs. Simply put, an API is a connection point between systems where there is communication. Let's explore those protocols:

HTTP (Hypertext Transfer Protocol): HTTP is the primary protocol used for communication between clients and servers on the web. It is the foundation of the REST architectural style and is widely supported by various programming languages and frameworks. HTTP provides a simple, standardized way to request and exchange data between clients and servers.

HTTPS (Hypertext Transfer Protocol Secure): HTTPS is the secure version of HTTP that incorporates encryption and authentication mechanisms using SSL/TLS protocols. It ensures that data transmitted between clients and servers is encrypted, providing confidentiality and integrity. HTTPS is essential for secure communication and is commonly used in APIs that handle sensitive data or require authentication.

WebSockets: WebSockets is a communication protocol that enables real-time, bidirectional, and full-duplex communication be-

tween clients and servers. It provides a persistent connection for instant data updates and event-driven interactions. WebSockets are commonly used in APIs that require real-time updates, such as chat applications, collaborative tools, or live data streaming.

MQTT (Message Queuing Telemetry Transport): MQTT is a lightweight publish-subscribe messaging protocol designed for efficient communication between devices and servers in IoT (Internet of Things) applications. It is used for sending and receiving messages between devices with low bandwidth and limited resources. MQTT is commonly used in APIs for IoT platforms and applications.

gRPC (Google Remote Procedure Call): gRPC is a high-performance, open-source framework developed by Google. It uses HTTP/2 as the underlying transport protocol and Protocol Buffers for data serialization. gRPC enables efficient and fast communication between services, making it popular for building APIs in microservices architectures.

AMQP (Advanced Message Queuing Protocol): AMQP is a messaging protocol that enables reliable message-oriented communication between applications. It provides a standardized way to send, receive, and exchange messages across different systems and platforms. AMQP is commonly used in enterprise messaging systems and APIs for reliable messaging.

Webhooks: Webhooks are a way for applications to provide real-time event notifications to other systems or applications. Instead of polling or periodically checking for updates, a webhook allows a server to send a POST request to a specified endpoint whenever a specific event occurs. Webhooks are often used for event-driven architectures and integrating systems that require immediate notifications.

SOAP (Simple Object Access Protocol): SOAP is a protocol that allows communication between applications over a network. It uses XML for message formatting and can operate over various

protocols, such as HTTP, SMTP, or TCP. SOAP APIs provide a more complex and rigid interface than REST APIs but offer features like built-in security, transaction support, and reliable messaging.

These protocols are widely used in various API implementations and cover a range of use cases, from traditional HTTP-based APIs to real-time communication and messaging protocols. The protocol choice depends on the API's specific requirements and characteristics and the supported client platforms.

The API design must be defensible

In an API design interview, we don't need to create a perfect design, but we must be able to defend what we designed. It often happens in real situations as well because we need to agree on an API contract with other teams, and we must tell them why we are creating the API that way.

It's essential to have a flexible API because external teams will use this API, coupling their code with it. The more situations we plan with the API, the better. We also need to remember that there is no need to fulfill requirements that one day we might have, and we can't plan too much in the future. Otherwise, the API will get too complicated for nothing in most cases.

API Versioning

API versioning is a practice in software development where different versions of an API (Application Programming Interface) are created and maintained to ensure compatibility and manage changes over time. It allows developers to introduce new features, modify existing functionality, and deprecate or remove certain parts of an API without disrupting existing consumers.

Here are a few common approaches to API versioning:

URL Versioning: In this approach, the version number is included in the URL itself. For example:

```
https://api.example.com/v1/users
https://api.example.com/v2/users
```

This means that if a user is already using the v1 endpoint, the user can keep using it even if developers created the v2 endpoint. That enables the customer to migrate to the other version whenever it's convenient.

The version is specified as part of the URL path, allowing clients to access a specific API version.

Query Parameter Versioning: In this approach, the URL includes the version number as a query parameter. For example:

```
https://api.example.com/users?version=1
https://api.example.com/users?version=2
```

The version is specified using the "version" query parameter, allowing clients to request a specific version of the API.

Header Versioning: In this approach, the version number is included as a header in the HTTP request. For example:

```
http
Copy
GET /users HTTP/1.1
Host: api.example.com
Accept: application/json
X-API-Version: 1
```

The version is specified using a custom header, such as "X-API-Version". Each version of an API represents a distinct set of functionality and behavior. When making changes to an API, developers typically follow versioning best practices to ensure backward

compatibility and minimize disruptions for existing clients. These practices may include:

- Introducing new API endpoints or modifying existing ones while preserving the old endpoints.
- Providing clear documentation and communication about the changes introduced in each version.
- Supporting multiple versions of the API concurrently, allowing clients to migrate to newer versions at their own pace.
- Deprecating and eventually removing outdated or unsupported versions of the - API after a certain period.
- API versioning is crucial for maintaining stability, allowing developers to evolve their APIs while providing a consistent experience for consumers and minimizing the impact of changes on existing integrations.

Stripe API

Stripe has their APIs if we want to use their payment services on our websites. Let's see the example they have for the retrieve customer endpoint:

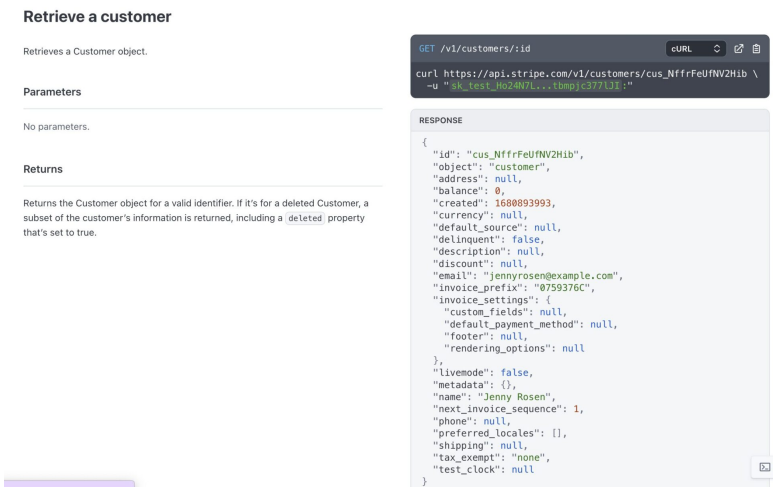


Figure 4. Retrieve Customer API Stripe

To understand better how the Stripe API works, read more in their documentation in the following link https://recurly.com/developers/api/v2021-02-25/index.html#section/Getting-Started.

Twitter (X):

To use the Twitter API, first, getting the logged user's token is required, and then invoking any API method is possible. Let's explore the tweets count API endpoint:

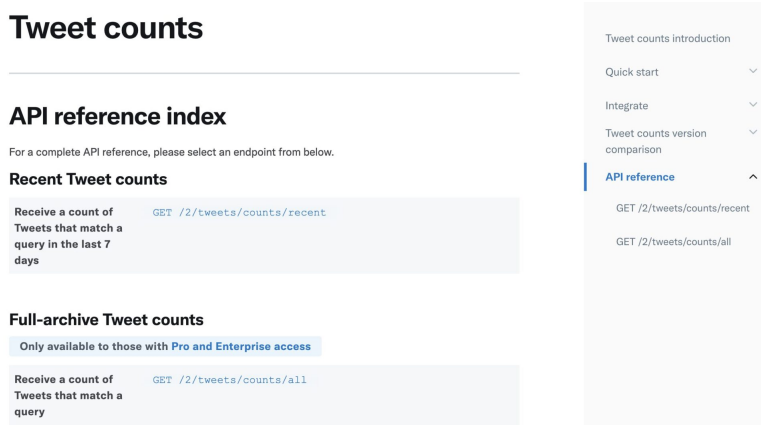


Figure 5. Tweet count API

To further explore the Twitter (X) API, take a look at the full documentation here.

Subscription Platform Recurly:

To use a SaaS platform to manage subscriptions, we have Recurly. Instead of implementing a subscription system, we can use what Recurly provides us.

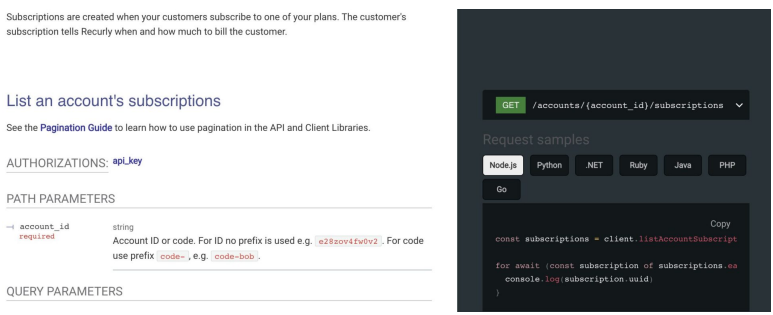


Figure 6. Recurly API Reference

To better understand how Recurly works, take a look at the documentation here.

Summary

We explored an API's main concepts and clarified what an API really is. Let's now recap the key points of what we learned about APIs:

- API stands for Application Programming Interface.
- An API defines a set of rules and protocols that allow different software applications to communicate and interact with each other.
- APIs enable developers to access and use the functionality of existing software components or services without needing to understand their internal workings.
- APIs provide a standardized way for applications to request and exchange data or perform specific actions.
- APIs can be used to retrieve data from a server, submit data to a server, or perform operations on a remote system.
- APIs can be categorized into different types, such as web APIs (HTTP-based APIs), library APIs (programming language-specific APIs), and operating system APIs.
- APIs often use common data formats for data exchange, such as JSON (JavaScript Object Notation) or XML (eXtensible Markup Language).
- APIs should be well-documented, providing clear instructions on how to use the API, including available endpoints, request/response formats, authentication methods, and any limitations or usage restrictions.
- APIs can be versioned to manage changes and ensure backward compatibility.
- APIs should have appropriate security measures in place, such as authentication and authorization mechanisms, to protect sensitive data and prevent unauthorized access.
- APIs play a crucial role in modern software development, enabling integration between different systems, fostering

interoperability, and promoting the development of third-party applications and services.

6 - Storage and Databases

Since the beginning of my career, I had to deal with databases by inserting, updating, searching, and deleting. Databases concepts are crucial for every developer to understand because they are present in nearly all applications.

Nowadays, with Microservices, this knowledge is even more important. That's because every Microservice may have different needs, and using the right database makes all the difference.

Let's see the components of a database so it's easier for you to learn new databases and do well in a systems design interview.

Computer Storage

Computer storage is the digital space where data is saved and stored in a computer system. It allows you to keep and access data even after you turn off your computer.

There are two main types of computer storage:

Primary storage: It is also known as memory and includes Random Access Memory (RAM) and Cache. The computer uses RAM to store data that is currently being used, while Cache is used to store frequently accessed data to speed up processing.

Secondary storage includes hard disk drives (HDDs), solid-state drives (SSDs), and external drives. These storage devices can store large amounts of data, ranging from documents to music and videos, and can be accessed anytime.

Storage capacity is usually measured in bytes, kilobytes (KB), megabytes (MB), gigabytes (GB), terabytes (TB), and petabytes (PB). The higher the storage capacity, the more data a computer can store.

It's important to regularly back up your data to prevent loss in case of a computer failure or other incidents. Cloud storage is becoming increasingly popular as a secure and convenient way to store data remotely.

Data in Disk

Most databases will insert data in a disk, meaning the data will be persisted, and even if there is an outage, the data can be retrieved. Let's see some of the databases that are persistent on the disk.

Databases such as Postgres, Oracle, MySQL, and SQL Server, MongoDB store data on disk

Data in Memory

Databases that make use of cache are databases that store data in memory. They are also called In-memory databases (IMDBs). Some examples of in-memory databases are Redis, Hazelcast, Ehcache, ArangoDB, H2, Memcached, and Apache Ignite.

Those databases will significantly enhance performance when the same data must be retrieved many times. The drawback is that the data will be lost if there is an outage.

Databases

A database collects organized data stored and managed in a computer system. It is designed to efficiently store, retrieve, and manage large amounts of data and can be used by various applications and users.

In a database, data is typically organized into tables of rows and columns. Each row represents a unique record, while each column represents a specific attribute or information about the record. The data is stored in a structured way, which makes it easy to search, sort, and analyze.

Databases are used in various applications, including business, education, research, healthcare, and more. They are essential for managing and organizing large amounts of data and support many important functions, such as customer relationship management, inventory management, financial reporting, etc.

RDBMS (Relational Database Management System) Relational databases such as Postgres, MySQL, and Oracle will have the data organized in tables that relate to each other using constraints. Those databases are highly reliable for consistency, meaning that you will always get the latest data, and it's also possible to have high availability since those databases can be replicated in different nodes (cloud machines).

ACID Transactions

ACID is a set of properties that ensure reliable and consistent database transactions.

Atomicity: Transactions are treated as a single unit of work. Either all the changes in a transaction are applied, or none of them are. There are no partial or incomplete transactions. This is

very important for financial transactions. Suppose you need to withdraw money. Then the system checks if money is available and will give you the money. You won't lose the money if any problem occurs during this transaction.

Consistency: Transactions bring the database from one valid state to another. The data in the database follow predefined rules or constraints, ensuring it is always valid. An example is the foreign key, which ensures a register will be inserted in two tables about the primary key from another table. Therefore, the consistency principle won't allow this process if we try to delete the register with the primary key.

Isolation: Transactions are executed independently and do not interfere with each other. Even if multiple transactions happen simultaneously, each transaction is isolated and will be executed in order.

Durability: Once a transaction is successfully committed, its changes are permanent and will survive future failures or system crashes. In other words, the data will persist in the disk and be available even if an outage happens.

In simple terms, ACID ensures that transactions are treated as a whole, maintain data integrity, work independently, and are resilient to failures. These properties are important for reliable and consistent database operations.

NoSQL

NoSQL stands for Not Only SQL, and nowadays, there are many famous of them. The great advantage of most NoSQL databases is that performance and amount of data are higher than relational databases. On the other hand, it's not as good to have sophisticated table relationships or be as consistent as a relational database.

Here are some key principles and features of NoSQL databases:

Schemaless: NoSQL databases are schemaless, meaning we can't have a structure for tables. This flexibility allows dynamic and evolving data structures without migrations or schema alterations.

Scalability and Performance: NoSQL databases handle massive data and high read/write workloads. They provide horizontal scalability by allowing data to be distributed out-of-the-box across multiple servers or clusters, enabling efficient scaling as data volumes and user traffic increase.

High Availability: NoSQL databases often replicate data across multiple nodes, ensuring that if one node fails, the data remains accessible from other nodes. This replication provides high availability and fault tolerance in distributed environments.

Data Models: NoSQL databases offer various data models, including key-value stores, document stores, column-family stores, and graph databases. Each data model caters to different use cases and provides additional capabilities for organizing and retrieving data.

Key-Value Stores: These databases store data as a collection of key-value pairs, allowing fast lookups and writes. Examples include Redis and Amazon DynamoDB.

Document Stores: Document databases store semi-structured or unstructured data as documents, typically in JSON or XML format. They provide flexibility in data representation and retrieval. Examples include MongoDB and Couchbase.

Column-Family Stores: Columnar databases store data in columns rather than rows, making them efficient for handling large amounts of structured data. Examples include Apache Cassandra and HBase.

Graph Databases: Graph databases focus on modeling and querying relationships between data entities. They are suitable for use cases involving complex relationships and network analysis. Examples include Neo4j and Amazon Neptune.

Use Cases: NoSQL databases find frequent use in scenarios that

demand high scalability, real-time data processing, and the management of large volumes of unstructured or semi-structured data. They commonly feature in web applications, real-time analytics, content management systems, and Internet of Things (IoT) applications.

Remember that NoSQL databases do not eliminate the need for relational databases. Relational databases are still well-suited for applications that require complex transactions, strong consistency, and pre-defined structured data. NoSQL databases provide an alternative approach that prioritizes scalability, flexibility, and performance in large-scale distributed systems.

CAP Theorem

With the advancement in cloud technologies, now it's possible to use a database in multiple machines (also called nodes). This means that it's possible to have massive scaling with data nowadays. There is even a term for that which is big data.

With relational databases, it's only possible to scale vertically, making one machine where the database is running more powerful and performant. This is expensive, though.

Using NoSQL databases and cloud technologies makes it possible to scale vertically, which is cheaper. Data will be synchronized between all machines, and data will be processed in parallel. With modern cloud technologies, this is possible to accomplish.

The CAP theorem is a graphic that demonstrates what we want to prioritize for your database. We can choose two of the characteristics between consistency, availability, and partition tolerance.

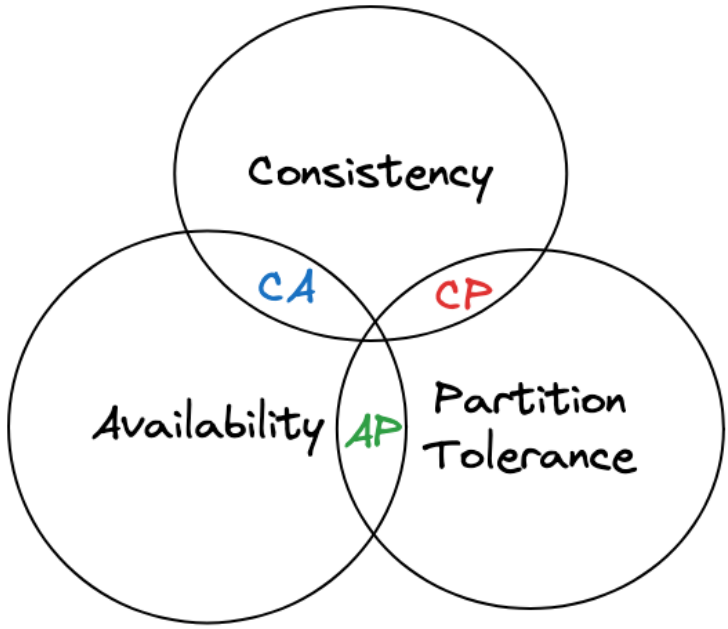


Figure 7. Cap Theorem

Let's explore the explanation from each of the database characteristics.

Consistency means that the data will be the data we want to retrieve, not stale. When we have a database replicated in many nodes (cloud machines), there are many processes in parallel to make the data consistent and reliable to what was inserted in the database. Therefore, if your use case requires you to always return the latest data in the database, you must prioritize consistency. Consistency is required for bank operations. For example, the data must be the latest always; otherwise, money can be lost.

Availability means that your database must be up and running no matter if other nodes are down. Your databases must have a mechanism to create other nodes if a node is down. Your database

must also have a replication strategy to reflect data in other nodes. If high availability is a must-have for you, you must think of a strategy to replicate nodes even in different countries so that if the servers from a country go down, you will have a backup server to run and replicate your database.

Partition Tolerance: Partition tolerance refers to the ability of a distributed system to continue functioning and providing consistent and available services despite network partitions or communication failures between nodes. Network partitions occur when nodes in a distributed system are separated and cannot communicate with each other because of an outage or network issues. Partition tolerance is crucial for systems that span multiple data centers or rely on unreliable network connections.

Now let's see the possible CAP theorem combination of characteristics and which databases use them:

CA (Consistency and Availability: Postgres, MySQL, Vertica, and Spanner are examples of databases that focus on consistency and availability. This means we will always have up-to-date data for those databases in one or multiple nodes. MySQL and Postgres can also replicate data in multiple nodes, therefore, being more available if one of the nodes goes down. It also guarantees the latest data even when using multiple nodes.

CP (Consistency and Partition Tolerance): MongoDB, Redis, MemcacheDB, and HBase are good examples of databases with consistency and partition tolerance. Data consistency is the top priority in a CP (Consistency and Partition tolerance) system, even in network partitions. Consistency ensures that all nodes have the same data at the same time. However, prioritizing consistency may lead to temporary unavailability during network partitions. CP systems, such as financial systems, are used in applications where data consistency is critical. The trade-off between consistency and availability varies, and different distributed systems offer different consistency and availability guarantees.

AP (Availability and Partition Tolerance): Apache Cassandra, Amazon DynamoDB, CouchDB, and Apache HBase are databases that use AP. AP systems prioritize high availability and the ability to function despite network partitions. In these systems, maintaining data consistency takes a backseat to ensure system uptime and responsiveness. They allow temporary inconsistencies in data and focus on eventual consistency, which means that data will be synchronized between nodes after some time or conflict resolution mechanisms. AP systems, such as distributed caching and real-time collaborative applications, are essential when availability and fault tolerance are available. While they may still provide some level of consistency, it is typically weaker or eventual rather than strong consistency.

When to use Relational Database in a Systems Design Interview?

The relational database is very effective when organizing complex data and having a relation between tables. Let's see some of the key points on when we should use a relational database:

Structured data: when the data needs to be pre-defined in multiple table relationships with constraints. Medium Volume: when there is no necessity to store massive data ACID: When we need consistency, robustness, and reliability in the stored data Reasons to avoid Relation Database:

It doesn't scale horizontally as well as NoSQL. This means that we have to optimize only one server where the database is running to scale the database to have vertical scaling. It's not effective to store massive data. There is no flexibility in inserting data.

When to Use NoSQL Database in a Systems Design Interview?

NoSQL is effective when we need to insert massive data. It's flexible to store structured or unstructured data and scalable and fast.

Unstructured data: even though NoSQL is flexible to be used with structured and unstructured data, it doesn't have the same power to structure data as relational databases. It's quite slow to perform complex queries. We must accept that we will store and retrieve data from one table with NoSQL. **Flexibility**: since the data from NoSQL is unstructured, there is the flexibility to insert whatever data we want in a table. This might be useful if the data is random. **Large data volume**: when it's necessary to store massive data in the database, NoSQL is a good choice. **Scales Horizontally**: In the cloud era, this is very important since it's possible to have multiple nodes of the NoSQL database in different machines from a cluster. This means that NoSQL is highly scalable. **Cheaper**: Since horizontal scaling is cheaper than vertical scaling, NoSQL has an advantage over relational databases. Reasons to avoid NoSQL:

It doesn't support ACID.

It's not good for complex relational database models since queries will be very slow. There will be a learning curve since there is no standard with NoSQL, depending on the technology you choose.

Summary

When deciding what database technology to use, understanding what comes before it is useful. In a Systems design interview,

we will have to understand fundamentals so it's faster to learn a technology. You have to know if you want your database to store information on disk for persistence or in memory for performance. Nowadays, there are so many technologies that knowing the fundamentals saves us a lot of time because we won't need to learn them in depth because they implement the fundamentals.

7 - Proxy

We must know what a proxy is to design cloud systems and pass in the systems design interview. Proxy is powerful and useful to create an intermediary between the client and server.

Let's see what types of proxies we have, how they can be helpful, and the technologies we can use.

Forward Proxy The forward proxy is an intermediary server between the client and the server. We will usually use a forward proxy to hide the IP for the client. Also, remember that a forward proxy will serve the client, not the server.

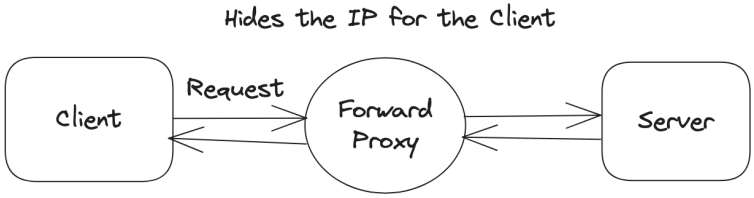


Figure 8. Forward Proxy

Let's see in more detail how a forward proxy works:

Client Request: A client sends a request to access a resource, such as a website, to the forward proxy server.

Forward Proxy Reception: The forward proxy server receives the client's request.

Target Server Connection: The forward proxy server establishes a connection with the target server on behalf of the client.

Forwarding the Request: The forward proxy server forwards the client's request to the target server. Target Server Response: The target server processes the request and generates a response.

Forward Proxy Response: The forward proxy server receives the response from the target server.

Response Delivery to Client: The forward proxy server sends the response back to the client that originally made the request. By using a forward proxy, clients direct their requests to the proxy server instead of directly communicating with external servers. This provides several benefits:

Privacy and Anonymity: The forward proxy server masks the client's IP address, enhancing privacy and providing a level of anonymity.

Content Filtering: The forward proxy server can be configured to filter and control access to specific websites or types of content based on predefined rules or policies.

Caching: The forward proxy server can cache frequently requested resources, reducing bandwidth usage and improving response times for subsequent requests.

Bandwidth Control: The forward proxy server can apply bandwidth control measures, limiting the data clients can receive or send.

Security: The forward proxy server can act as a firewall, inspecting incoming and outgoing traffic and providing an additional layer of security by blocking malicious or unauthorized requests.

Forward proxies are commonly used in corporate networks, educational institutions, and organizations to control internet access, enforce security policies, and optimize network usage. They allow clients to access external resources indirectly through the proxy server, providing various functionalities and controls in the process.

Reverse Proxy

A reverse proxy is a server or software application that acts as an intermediary between clients and servers, accepting client requests and forwarding them to the appropriate backend servers. It operates on behalf of the server, meaning that the client is usually unaware of a reverse proxy.

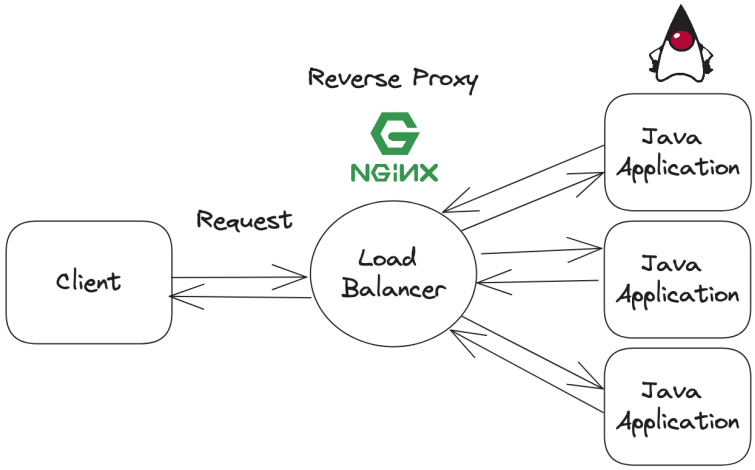


Figure 9. Reverse Proxy

Here's how a reverse proxy works:

Client Request: A client sends a request to access a resource, such as a web page, to the reverse proxy server.

Reverse Proxy Reception: The reverse proxy server receives the client's request.

Backend Server Selection: The reverse proxy determines which backend server or servers should handle the request. This selection can be based on various factors such as load-balancing algorithms, server availability, or specific routing rules.

Forwarding the Request: The reverse proxy forwards the client's request to the selected backend server, acting as an intermediary.

Backend Server Response: The backend server processes the request and generates a response.

Reverse Proxy Response: The reverse proxy server receives the response from the backend server.

Response Delivery to Client: The reverse proxy sends the response back to the client that originally made the request. Using a reverse proxy, clients interact with the proxy server instead of directly communicating with the backend servers. This offers several benefits:

Load Balancing: The reverse proxy can distribute incoming requests across multiple backend servers, balancing the load and optimizing resource utilization.

Caching: The reverse proxy can cache frequently requested resources, reducing the load on backend servers and improving response times for subsequent requests.

Security: The reverse proxy can act as a security barrier, providing an additional layer of protection by filtering and inspecting incoming traffic before forwarding it to backend servers.

SSL Termination: The reverse proxy can handle SSL encryption and decryption, offloading the resource-intensive task from backend servers.

Content Delivery: The reverse proxy can serve static content directly, relieving backend servers from handling such requests and improving overall performance.

Reverse proxies are commonly used in web applications, content delivery networks (CDNs), and server farms to enhance performance, scalability, and security. They enable efficient resource allocation, improve response times, and provide additional layers of protection for backend servers.

Forward Proxy VS Reverse Proxy

The main difference between a forward proxy and a reverse proxy lies in the direction of the communication flow and the parties involved. Here are the key distinctions:

Forward Proxy

Communication Flow: In a forward proxy setup, the client directs its requests to the proxy server, and the proxy server forwards those requests to the servers on behalf of the client.

Client Perspective: The client is aware of the forward proxy's existence and explicitly configures its network settings to use the proxy server.

Anonymity and Privacy: Forward proxies can provide privacy and anonymity to clients by masking their IP addresses from external servers.

Content Filtering: Forward proxies can be configured to filter and control access to specific websites or types of content based on predefined rules or policies.

Caching: Forward proxies can cache frequently requested resources, reducing bandwidth usage and improving response times for subsequent requests.

Example Use Case: A corporate network might use a forward proxy to control internet access, monitor and filter employee web activity, and provide caching for improved performance.

Reverse Proxy

Communication Flow: In a reverse proxy setup, the client sends requests to a reverse proxy server, which then forwards those

requests to the appropriate backend servers handling the requested resources.

Client Perspective: The client is generally unaware of the reverse proxy's existence and communicates with the reverse proxy as if it were the intended server.

Load Balancing: Reverse proxies often distribute incoming requests across multiple backend servers to balance the load and optimize resource utilization.

SSL Termination: Reverse proxies can handle SSL encryption and decryption, offloading the resource-intensive task from backend servers.

Security: Reverse proxies act as a security barrier, filtering and inspecting incoming traffic before forwarding it to backend servers to protect against malicious requests.

Example Use Case: A website or application with multiple servers might use a reverse proxy to distribute traffic (load-balancing), improve scalability, provide SSL termination, and enhance security.

In summary, a forward proxy primarily acts on behalf of clients, facilitating their requests to external servers. In contrast, a reverse proxy primarily acts on behalf of backend servers, receiving requests from clients and forwarding them to the appropriate servers.

Proxy Technologies

Now that we understand the fundamental differences between a forward proxy and reverse proxy let's see the top technologies used in the market:

Nginx: Nginx, primarily known as a web server, can also function as a forward proxy. It is lightweight, scalable, and efficient. Nginx supports HTTP, HTTPS, and other protocols, and it can be configured for caching and load balancing.

Squid: Squid is a widely used open-source caching proxy server that supports HTTP, HTTPS, FTP, and other protocols. It offers caching, content filtering, access control, and authentication features.

Apache HTTP Server: Apache HTTP Server has a built-in proxy module that allows it to act as a forward proxy. It supports HTTP, HTTPS, FTP, and more. Apache offers features like caching, load balancing, and authentication. Apache HTTP Server can also be configured to function as a reverse proxy using its mod_proxy module. It can distribute incoming requests, perform load balancing, and provide SSL/TLS termination.

HAProxy: HAProxy is a high-performance TCP/HTTP load balancer and proxy server. While commonly used as a reverse proxy, it can also function as a forward proxy. HAProxy supports HTTP, HTTPS, and TCP protocols, and it excels in handling high loads and providing advanced load-balancing features.

Envoy Proxy: Envoy is a modern, cloud-native proxy that offers advanced features such as dynamic service discovery, load balancing, circuit breaking, and observability. It is designed for managing microservices architectures and containerized environments.

Conclusion

When designing cloud systems, we will use a Proxy all the time. We will usually use a forward proxy to mask the IP from the client. Also, the forward proxy will act on behalf of the client.

In most of the Microservices, we will use a reverse proxy. A reverse proxy focuses on the server side. The most common use cases for a reverse proxy are load balancer, circuit breaker, dynamic service discovery, and observability.

Forward Proxy:

- Client directs requests to the proxy server.
- Proxy server forwards requests to external servers.
- Clients know the forward proxy and configure network settings accordingly.
- Provides privacy and anonymity by masking client IP addresses.
- Can filter and control access to specific content.
- Supports caching to improve performance.
- Commonly used for controlling internet access in corporate networks.

Reverse Proxy:

- Client sends requests to the reverse proxy server.
- Reverse proxy server forwards requests to backend servers.
- Clients are generally unaware of the reverse proxy's existence.
- Handles load balancing by distributing requests to multiple backend servers.
- Performs SSL termination, offloading encryption/decryption from backend servers.
- Acts as a security barrier by filtering and inspecting incoming traffic.
- Used to improve scalability, enhance security, and optimize resource utilization.

8 - Latency and Throughput

Latency and throughput are crucial concepts for a Systems Design interview. Since building Microservices in the cloud is the new normal in the market, knowing basic network concepts make a big difference. We will often see those terms when designing a system in the cloud.

Let's explore those concepts further so you get sharp for the Systems Design interview and learn to build systems in the cloud more effectively.

Latency

Latency is the time it takes for data to travel from one point to another in a system. We measure latency in nanoseconds (ns), microseconds (μs), milliseconds (ms), or seconds (s).

Latency is significant because it affects the performance of a system. For example, if a website has high latency, it will take longer for users to load and interact with it.

What can affect latency?

Several factors can affect latency, including:

The distance between the two points: The further apart the two points are, the higher the latency will be. The type of network: A wired network typically has lower latency than a wireless network.

The amount of traffic on the network: If there is a lot of traffic, latency will be higher. What can reduce latency? Latency can be reduced by using a wired network, reducing the distance between the two points, and using a network that is not congested.

Simulation of Latency Time for Data Reading

Let's roughly estimate the is the latency to transfer one megabyte of data:

Reading Data from	Time in Milliseconds
SSD (solid-state drive)	1 ms
Via Network (If it's in the same country or close enough)	10 ms
HDD (hard disk drive)	20ms
Via Network From Ireland to the US	150ms

Notice that sending 1 MB via a network is faster than reading data from HDD. Also, sending data from one continent to the other is much slower. Your systems design decisions have to take that in consideration if your system needs fast response.

For example, the latency from Ireland to the US is roughly 150 ms for each MB. This is relatively fast for a single packet of data. However, if you send a large file, the latency will increase, and the transfer will take longer.

Why latency is important

Latency is important for gaming or any application that requires quick responses. In a game, for example, it is important to have low latency so that the game is responsive and players can react quickly. If the latency is too high, it can make the game unplayable.

For websites, latency might not be as important as availability. This is because users can usually wait a few seconds for a website to load. However, if a website is unavailable, users cannot access it.

In general, latency is important to consider when designing and building a system. You can design a fast and reliable system by understanding the different factors that affect latency.

Throughput

Throughput is the rate at which data is successfully transferred from one point to another. We typically measure throughput in bits per second (bps), kilobits per second (Kbps), or megabytes per second (Mbps).

Throughput is crucial because it determines how quickly we can transfer data between two devices. For example, if you are downloading a file from the internet, your internet connection's throughput will determine how long it takes to download the file.

Some factors can affect throughput, including:

The type of connection: A wired connection typically has a higher throughput than a wireless connection. The distance between the two devices: The further apart the two devices are, the lower the throughput. The amount of traffic on the network: If there is a lot of traffic, the throughput will be lower. You can improve throughput using a wired connection, reducing the distance between the two devices and network traffic.

Bandwidth

The bandwidth is the amount of data we can transfer per unit of time. For example, imagine we have a pipe with water running through it. If the pipe is large inside, more water can pass through. If the pipe is narrower, less water can pass through. In this case, the bandwidth is the maximum amount of water that can go through the pipe.

We measure bandwidth in bits per second (bps) in computer networks. The higher the bandwidth, the more data we can transfer per unit of time.

Bandwidth is essential for several reasons. For example, if you are streaming a video online, you need a high bandwidth connection to avoid buffering. Similarly, if you are downloading a large file, you will need a high bandwidth connection to complete the download quickly.

Several factors can affect bandwidth, including the type of connection you have, the amount of traffic on the network, and the distance between you and the server you are connecting to.

If you are experiencing problems with slow internet speeds, you can do several things to try to improve your bandwidth. For example, you can use a different type of connection, such as a cable modem or a fiber optic connection. You can also try connecting to a different server, or you can reduce the amount of traffic on your network.

Bandwidth is necessary to understand and distinguish throughput to design your system.

Analogy Explanation from Throughput, Latency, and Bandwidth

Now that we understand throughput, latency, and bandwidth, let's see an analogy from those terms in a pipe and what they represent in the context of networks.

In the pipe, the water is the data, the pipe distance represents the latency, the thickness of the pipe is the bandwidth, and the water that goes through the pipe is the throughput:

Figure 10. Throughput and Bandwidth pipeline

Throughput vs. Bandwidth

Bandwidth and throughput are two terms that are often used interchangeably, but they actually have different meanings.

Bandwidth is the maximum data transfer rate that a network can support. It is measured in bits per second (bps). Throughput, on the other hand, is the actual data transfer rate that is sent over a network. It is also measured in bits per second (bps).

Bandwidth is determined by the physical characteristics of the network, such as the type of cable used and the number of nodes on the network. Throughput, on the other hand, is affected by a number of factors, including the amount of traffic on the network, the type of applications that are being used, and the distance between the sender and receiver.

It is important to understand the difference between bandwidth and throughput because it can help you to choose the right network for your needs. If you need to transfer large amounts of data quickly, then you will need a network with a high bandwidth. However, if you are only transferring small amounts of data, then a network with a lower bandwidth may be sufficient.

Here is a table that summarizes the key differences between bandwidth and throughput:

Feature	Bandwidth	Throughput
Definition	Maximum data transfer rate	Actual data transfer rate
Unit of measurement	Bits per second (bps)	Bits per second (bps)
Determinants	Physical characteristics of the network	Amount of traffic, applications, distance between sender and receiver
Importance	Important for choosing the right network	Important for understanding network performance

Summary

To understand throughput, latency and bandwidth is very important when designing a system in the cloud. In AWS or any other cloud vendor we will see those terms all the time. When working with Microservices, those terms will show up very often and not knowing what are those terms will make things far more difficult to

understand a tool such as AWS or monitoring tools such as Datadog or Dynatrace.

9 - Load Balancer

A load balancer actively distributes incoming network traffic across multiple servers or resources, optimizing resource utilization, scalability, and the availability of applications or services. It acts as a traffic manager, evenly distributing requests based on factors such as server health, workload, and predefined algorithms.

Load balancers operate at different network layers, such as the transport layer (Layer 4) or application layer (Layer 7), and provide functionalities like session persistence, SSL termination, and health monitoring. They play a critical role in ensuring high availability, scalability, and efficient resource utilization in modern IT infrastructures, enhancing overall performance and user experience.

Let's see a diagram of how a Load Balancer operates:

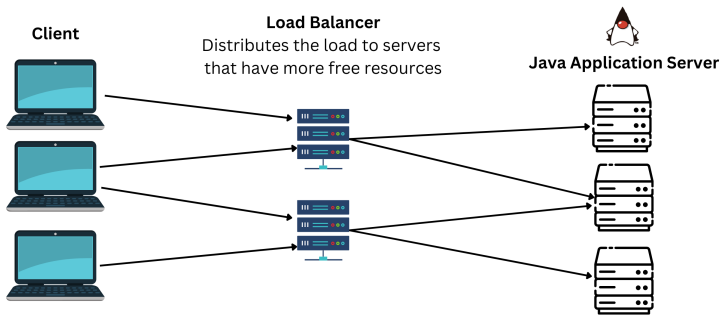


Figure 11. Load Balancer

Load Balancer in Practice

To show a real example of load balancer in action, let's see the server IP changing on each request we make when using the dig command for the Amazon website:

```
dig amazon.com
```

```
;; ANSWER SECTION:
amazon.com.             281     IN      A       52.94.236.248
amazon.com.             281     IN      A       205.251.242.103
amazon.com.             281     IN      A       54.239.28.85

;; Query time: 45 msec
;; SERVER: 192.168.1.1#53(192.168.1.1)
;; WHEN: Fri Sep 08 22:44:47 IST 2023
;; MSG SIZE  rcvd: 87
```

Figure 12. Amazon Dig 1

Notice that every time we use the dig command to the amazon.com website a different server is accessed. Notice that in the first response we have 52.94.236.248 and in the second one we have 205.251.242.103.

Also, obiviously the Amazon webserver uses a load balancer since there is a massive amount of requests to its website.

Types of Load Balancer

Several load balancers are available, each with its own characteristics and use cases. Here are some common types of load balancers:

Layer 4 Load Balancer: A Layer 4 load balancer operates at the OSI model's transport layer (Layer 4). It makes load-balancing decisions based on information such as source IP address, destination IP address, and ports. We often use Layer 4 load balancers for distributing traffic across multiple servers based on network-level information.

Layer 7 Load Balancer: A Layer 7 load balancer operates at the OSI model's application layer (Layer 7). It can make load-balancing decisions based on more advanced criteria, such as specific URLs, HTTP headers, or cookies. Layer 7 load balancers can perform

content-based routing and distribute traffic based on application-level information.

Hardware Load Balancer: A hardware load balancer is a physical appliance designed for load balancing. It often provides high-performance load-balancing capabilities and can handle large amounts of traffic. Software engineers typically deploy hardware load balancers on-premises data centers.

Software Load Balancer: A software load balancer is a load-balancing solution implemented in software running on general-purpose servers or virtual machines. Software load balancers can be deployed on-premises or in cloud environments, offering flexibility and scalability.

Application Delivery Controller (ADC): An ADC is a specialized load balancer that offers additional features beyond essential load balancing, such as SSL termination, caching, compression, and application-layer security. ADCs are used in enterprise environments to optimize application performance and improve security.

Cloud Load Balancer: Cloud service providers, such as Amazon Web Services (AWS) and Microsoft Azure, offer load-balancing services designed for their respective cloud platforms. These cloud load balancers are often fully managed services and provide scalability, high availability, and integration with other cloud services.

DNS Load Balancer: DNS load balancing involves distributing traffic across multiple IP addresses associated with a single domain name. DNS load balancers respond to DNS queries with different IP addresses based on load-balancing algorithms. It can be an effective method for distributing traffic across multiple servers or regions. These are some of the common types of load balancers used in various environments. The choice of load balancer depends on the application requirements, scalability needs, deployment environment, and budget.

Microservices Load Balancer

When it comes to load balancing in microservices architectures, there are several types of load balancers commonly used to distribute traffic across the microservices. Here are the main types:

Reverse Proxy Load Balancer: A reverse proxy load balancer acts as an intermediary between clients and microservices. It receives incoming requests and forwards them to the appropriate microservice based on predefined routing rules. Reverse proxy load balancers often provide additional features such as SSL termination, request/response modification, and traffic management capabilities.

Service Mesh Load Balancer: A service mesh load balancer is a specialized load balancing component within a service mesh architecture. Service meshes, like Istio or Linkerd, provide features such as intelligent routing, load balancing, and traffic control for microservices. The service mesh load balancer operates at the sidecar level, facilitating communication between microservices and ensuring efficient traffic distribution.

Container Orchestrator Load Balancer: Container orchestrators like Kubernetes often include built-in load balancing capabilities. They use a load balancer component, such as Kubernetes Service or an Ingress controller, to distribute traffic to microservices running within containers. These load balancers can dynamically adapt to changes in the containerized environment, scaling and routing traffic based on the current state of the cluster.

DNS Load Balancer: DNS-based load balancing involves using DNS records to distribute traffic across multiple instances of microservices. Clients resolve a domain name to multiple IP addresses representing different instances of the microservice. DNS load balancing can use strategies like Round Robin or Weighted Round Robin to distribute traffic across the resolved IP addresses.

Client-Side Load Balancer: In a client-side load balancing approach, the load balancing logic resides within the client application itself. The client is responsible for selecting an appropriate microservice instance to send requests to. Client-side load balancers often employ dynamic service discovery mechanisms to obtain information about available microservices and make load balancing decisions based on factors like latency, health, or proximity.

These are the primary types of load balancers commonly used in microservices architectures. The choice of load balancer depends on factors such as the specific requirements of the microservices environment, the level of control and flexibility needed, and the tools or platforms being used to manage and orchestrate the microservices.

Weighted Round Robin Load balancer

A Weighted Round Robin (WRR) load balancer is a type of load balancing algorithm that distributes incoming traffic across multiple servers or backend resources based on predefined weights. In a WRR load balancer, each server is assigned a weight that represents its capacity or performance capability. The weight determines the proportion of traffic that each server receives during load balancing.

Here's how a Weighted Round Robin load balancer typically works:

Server Weight Assignment: Each server in the load balancer pool is assigned a weight value. The weight value can be an integer or a relative value representing the server's capacity or performance. Servers with higher weights are allocated a larger share of the traffic.

Load Balancing Algorithm: When a new request arrives, the WRR load balancer follows a round-robin scheduling algorithm to

distribute the traffic. However, the selection of the next server to receive the request is influenced by the assigned weights.

Traffic Distribution: The load balancer maintains a pointer or index that keeps track of the current server in the rotation. The load balancer selects the server using the round-robin algorithm, but the server with the highest weight value is selected more frequently than servers with lower weights.

Weight Adjustment: The weight assigned to each server can be adjusted dynamically based on factors such as server health, resource utilization, or performance metrics. This allows for dynamic load balancing based on real-time conditions.

The key advantage of using a Weighted Round Robin load balancer is the ability to allocate traffic more intelligently based on the capacity or performance of individual servers. This approach ensures that servers with higher capacities receive a larger share of the traffic, leading to better utilization of resources and improved overall system performance.

It's worth noting that Weighted Round Robin is just one of several load balancing algorithms available. Other algorithms, such as Least Connections, IP Hash, or Least Response Time, may be more suitable depending on the specific requirements and characteristics of the application or environment.

Summary

In summary, load balancers are critical components in modern computing infrastructure that optimize the distribution of network traffic, enhance application performance, and ensure high availability. By evenly allocating traffic across multiple servers or backend resources, load balancers prevent overloading, reduce response times, and deliver a seamless user experience.

They play a key role in scaling applications, handling high traffic loads, and adapting to changing demands, all while maintaining the reliability and availability of the system. Load balancers are indispensable tools for organizations seeking to optimize resource utilization, improve application responsiveness, and provide reliable and efficient services to their users.

10 - Hashing

What is Hashing?

When implementing code or designing a system, hashing will be there. The fundamental knowledge of the hashing concept will make a big difference to design high quality systems and pass in the systems design interview.

In simple words, hashing in computer science is a way to quickly convert big pieces of information, like words or numbers, into smaller codes. It's like using a special function or formula to create a secret fingerprint or unique identifier for the information. This helps in organizing and finding data faster, securing passwords, and other useful tasks in computer systems.

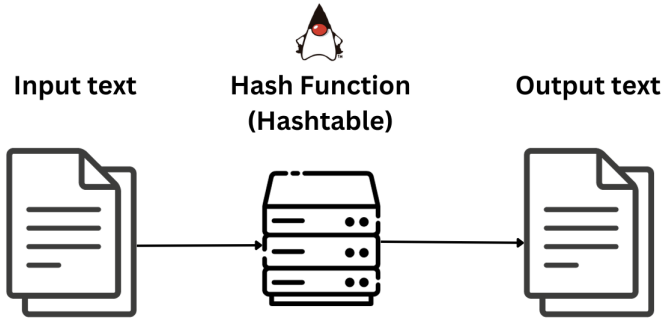


Figure 13. Hash Function Flow

Where to Use Hashing?

We use Hashing in various areas. Here are a few simple examples:

Data retrieval: In Java and other programming languages, the key value data structure will use a hash algorithm. In Java, we use the HashSet or Hashtable classes for key value store. Then, we use a hash algorithm in the hashCode method to retrieve data more quickly.

Password storage: When you create an account on a website or an app, your password is often hashed and stored in a database. Hashing ensures that even if someone gains unauthorized access to the database, they can't quickly retrieve the original passwords. When you log in, your entered password is hashed again, and the system checks if it matches the stored hash.

Digital signatures: Hashing plays a crucial role in verifying the

authenticity and integrity of digital documents. A hash function is used to create a unique fingerprint of a document. If a single character in the document changes, the hash value will be completely different. This fingerprint can be used to ensure that the document has not been tampered with.

File integrity checks: Hashing ensures the integrity of files during transmission or storage. By calculating and comparing hash values before and after transferring a file, you can verify if the file has been modified or corrupted.

Caching: We commonly use Hashing in caching mechanisms. A cache is a temporary storage that holds frequently accessed data to improve performance. Hashing allows quick lookup and retrieval of cached data based on a unique key. By using hash functions, caches can efficiently store and retrieve data, reducing the need to access slower storage systems.

Database indexing: Databases often use hashing to create indexes for efficient retrieval. Hash-based indexes store keys and corresponding pointers to data records. This enables fast lookup operations, especially for equality-based searches, where the hash value is a direct pointer to the desired data.

Cryptographic applications: Hash functions are fundamental in cryptographic algorithms. They are used to create digital signatures, ensure message integrity, and provide secure storage. Hash functions in cryptography are designed to be one-way, meaning it is extremely difficult to reverse-engineer the original input from the hash value.

Data deduplication: Hashing is employed in data deduplication techniques to identify and eliminate duplicate data. By hashing the content of files or data blocks, identical instances can be easily identified and eliminated, leading to optimized storage utilization.

Load balancing: Hashing is utilized in load-balancing algorithms to distribute incoming requests across multiple servers. A hash function is applied to the request data, and the result is used to

determine which server should handle the request. This helps evenly distribute the workload and improves system performance.

Digital Forensics: Hashing plays a significant role in digital forensics for data integrity verification, file identification, and password hash analysis. Hash values are computed for original files and used to detect changes or tampering during the forensic process. Hash functions are also employed to analyze password hash databases recovered from compromised systems.

Caching and Memoization: Hashing is used in caching mechanisms to store and retrieve frequently accessed data. By using the hash value of a key, the cache can quickly determine if the requested data is already available. Hashing is also used in memoization, where the results of expensive function calls are stored in a cache based on their input parameters.

These are just a few examples of where hashing can be used. The versatility, efficiency, and security properties of hashing make it an essential tool in various domains, including data management, security, cryptography, and performance optimization.

Practical Java Hash Example

Let's see how the HashSet class uses hash in practice.

The following is the hashCode implementation of the String class. You don't need to memorize or fully understand the following code but this is good for you to have an idea of what is hashCode. As mentioned before, a hashCode is used to enable data retrieval more quickly. It's much faster to check one code instead of all the values of an object.

```
public final class String implements java.io.Serializable,
            Comparable<String>, CharSequence, Constable, \
ConstantDesc {

 public int hashCode() {
        if (h == 0 && !hashIsZero) {
            h = isLatin1() ? StringLatin1.hashCode(value)
                           : StringUTF16.hashCode(value);
            if (h == 0) {
                hashIsZero = true;
            } else {
                hash = h;
            }
        }
        return h;
    }

}
```

Let's see how the hashCode is printed:

```

  public static void main(String[] args) {
    String name = "Duke";
    System.out.println(name.hashCode());
  }

}
```

Output: 2141643

Server Selection

Imagine you have a big website that gets lots of visitors, and you want to distribute the workload across multiple servers to ensure

smooth performance. Server selection hashing is a technique used to decide which server should handle each request.

Here's how it works. Let's say you have a bunch of servers available, labeled Server A, Server B, Server C, and so on.

When a user sends a request to your website, you want to determine which server should handle that request.

To make this decision, you use a hash function. A hash function takes some input, like the user's IP address or a unique identifier, and produces a fixed-length hash value.

The hash value is then used to map the request to one of the servers. For example, if the hash value falls within a certain range, you might assign the request to Server A. If it falls within another range, you might assign it to Server B, and so on.

The idea behind server selection hashing is that the same input will always produce the same hash value. So, if a particular user sends multiple requests, they will consistently be directed to the same server, ensuring that their session remains intact.

By using server selection hashing, you can evenly distribute the incoming requests across multiple servers. This helps balance the workload, prevent overloading of individual servers, and ultimately provide a better user experience on your website.

Types of Hashing

Hashing is a fundamental concept in computer science and cryptography, used to transform data into a fixed-size value (hash value or hash code) that represents the original data. There are several types of hash functions and hashing algorithms available, each with its own characteristics and areas of application. Here are some commonly used types of hashing:

Cryptographic Hash Functions: These hash functions are primarily designed for data integrity and security. They produce a fixed-size hash value (typically 128, 160, 256, or 512 bits) and have properties like pre-image resistance, second pre-image resistance, and collision resistance. Examples of cryptographic hash functions include MD5 (Message Digest Algorithm 5), SHA-1 (Secure Hash Algorithm 1), SHA-256, and SHA-3.

Non-Cryptographic Hash Functions: These hash functions are generally faster but do not provide the same level of security as cryptographic hash functions. They are commonly used for hash tables, data indexing, checksums, and other non-security-critical applications. Examples include MurmurHash, Jenkins Hash, and Pearson Hash.

Message Digest Algorithms: These are a type of hash function that takes an input message of arbitrary length and produces a fixed-size hash value. Message Digest Algorithms are commonly used in digital signatures, password storage, and data integrity checks. Examples include MD5, SHA-1, and SHA-2.

Perfect Hash Functions: A perfect hash function is a hash function that maps distinct keys to unique hash values without any collisions. It ensures that no two different keys will have the same hash value. Perfect hash functions are useful in situations where efficient lookup of items in a set or dictionary is required.

Bloom Filters: A Bloom filter is a probabilistic data structure that uses a set of hash functions to test whether an element is a member of a set. Bloom filters provide fast membership tests but may have a small probability of false positives. They are commonly used in caching, spell-checking, and network routing.

Cryptographic Key Derivation Functions: These functions are used to derive cryptographic keys from a given input, such as a password or passphrase. Key derivation functions are designed to be computationally expensive and slow down potential attackers in brute-forcing passwords. Examples include PBKDF2 (Password-

Based Key Derivation Function 2) and bcrypt.

These are just a few examples of the types of hashing algorithms and functions available. The choice of a particular hashing algorithm depends on the specific requirements of the application, such as security, speed, and memory constraints.

Consistent Hashing

Imagine you have a big website that uses multiple servers to handle user requests. Consistent hashing is a technique used to distribute the load evenly across these servers while minimizing disruptions when servers are added or removed.

Here's how it works:

Instead of dividing the servers into fixed ranges like in regular hashing, consistent hashing uses a circular ring-like structure. Each server is represented by a point on the ring. These points are distributed evenly around the ring.

When a request comes in, a hash function is used to calculate a hash value for that request.

The hash value is then mapped onto the ring. Starting from that point, you move clockwise on the ring until you find the nearest server or the next available server point.

This server is responsible for handling the request. So, requests with similar hash values will be directed to the same or nearby servers.

The key idea behind consistent hashing is that when a server is added or removed, only a small portion of the keys or requests need to be remapped. This minimizes the disruption to the overall system. With regular hashing, adding or removing a server could require remapping a significant portion of the keys, which can be time-consuming and inefficient.

By using consistent hashing, you can achieve load balancing across servers while maintaining stability when servers are added or taken out of the system.

I hope this explanation helps you understand consistent hashing in simple terms!

Rendezvous Hashing

Rendezvous hashing, also known as highest random weight (HRW) hashing, is a technique used to determine which node or server should handle a particular request based on a predetermined set of nodes. It aims to evenly distribute the workload while maintaining stability when nodes are added or removed from the system.

Here's how it works. Imagine you have a set of nodes or servers labeled A, B, C, and so on.

When a request comes in, a hash function is applied to both the request and each node in the system.

For each node, the hash function produces a hash value. The node with the highest hash value is chosen as the owner of that request.

The idea behind rendezvous hashing is that each request "rendezvous" with the node that generates the highest hash value for that request. It's like the request and the node "meet" at the point with the highest value.

If a node is added or removed from the system, only the requests that would have been assigned to that specific node need to be remapped to a different node. This minimizes the disruption caused by node changes.

The strength of rendezvous hashing lies in its ability to evenly distribute the workload across nodes. Since the hash function takes both the request and node into account, it provides a balanced

assignment of requests, ensuring that no single node becomes overwhelmed with requests while others remain idle.

Rendezvous hashing is particularly useful in scenarios where the set of nodes is relatively stable, and the focus is on load balancing and minimizing remapping when nodes change.

SHA Detailed Explanation

SHA (Secure Hash Algorithm) is a cryptographic hash function that takes an input (usually a message or data) and produces a fixed-size output called a hash value or digest. The purpose of the hash function is to ensure data integrity and provide a unique representation of the input.

There are several versions of SHA, such as SHA-1, SHA-256, SHA-384, and SHA-512, each with different output sizes. The most commonly used version is SHA-256, which produces a 256-bit hash value.

Here's a high-level explanation of how SHA hashing works:

Pre-processing: The input data is processed to meet specific requirements. Padding is added to the input to make it a multiple of a fixed block size. Additional information, such as the length of the input, may also be appended.

Message Digest Initialization: The hash algorithm initializes a set of variables, called the initial hash values, which serve as the starting point for the hashing process. These values are predefined for each SHA variant.

Message Digest Computation: The input data is divided into multiple blocks, and the hash algorithm processes each block in sequence. For each block, logical and arithmetic operations are performed to transform the data and update the hash values.

Output: Once all blocks have been processed, the final hash value, the message digest, is obtained. The message digest is a fixed-length representation of the input data. Even a tiny change in the input data will likely produce a substantially different message digest.

SHA hashing is designed to have several properties:

Deterministic: Given the same input, the hash function will always produce the same output. This allows for easy verification of data integrity. **Fast computation**: SHA hashing algorithms are designed to be efficiently computed on modern computer systems. **Pre-image resistance**: Given a hash value, it should be computationally infeasible to determine the original input that produced that hash value. **Collision resistance**: It should be extremely difficult to find two inputs producing the same hash value.

SHA hashes are widely used in various applications, such as password storage, digital signatures, and data integrity verification. They provide a reliable way to verify the integrity and authenticity of data without revealing the original input. However, it's worth noting that some older versions of SHA, such as SHA-1, have known vulnerabilities and are no longer considered secure for specific applications. Using the newer and more secure SHA-256 or SHA-3 algorithms is generally recommended when possible.

Difference Between Hashing and Cryptography

Hashing and cryptography are related concepts but serve different purposes in computer science and information security. Here are the key differences between hashing and cryptography:

Purpose

Hashing: Hashing is primarily used for data integrity and verification. It takes an input (message or data) of any size and produces a fixed-size output called a hash value or digest. The main purpose of hashing is to ensure that the data has not been tampered with or modified during transmission or storage.

Cryptography: Cryptography involves the encryption and decryption of data to ensure confidentiality, integrity, authenticity, and non-repudiation. It encompasses various techniques, including encryption algorithms, key management, digital signatures, and more. Cryptography aims to protect data from unauthorized access and ensure secure communication.

Output

Hashing: The hashing output is a fixed-size hash value or digest, typically represented as a sequence of characters or numbers. The hash value uniquely represents the input data, and even a small change in the input will produce a significantly different hash value.

Cryptography: The output of cryptography depends on the specific cryptographic algorithm being used. It can involve encryption, where the original data is transformed into an unreadable form (ciphertext), and decryption, where the ciphertext is converted back to its original form (plaintext). Cryptography does not necessarily produce fixed-size outputs like hashing.

Reversibility

Hashing: Hash functions are designed to be one-way functions, meaning retrieving the original input data from the hash value is

computationally infeasible. Hash functions are irreversible, and the original data cannot be obtained from the hash value.

Cryptography: Cryptographic algorithms can be reversible, allowing the original data to be retrieved from the encrypted form using the appropriate decryption key. Encryption and decryption are complementary operations in cryptography.

Key Usage

Hashing: Hash functions do not typically use keys. The same input data will always produce the same hash value. Hashing is deterministic and does not involve any secret information.

Cryptography: Cryptographic algorithms often involve using keys. Encryption requires a key to transform the original data into ciphertext, and decryption requires a corresponding key to revert the ciphertext to plaintext. The security of cryptographic systems relies on keeping the keys secret.

Security Goals

Hashing: The main security goal of hashing is data integrity. Hash functions are designed to detect even minor changes in the input data by producing a different hash value. They are not designed to provide confidentiality or protect against unauthorized access.

Cryptography: Cryptography encompasses multiple security goals, including confidentiality (ensuring the data remains secret), integrity (detecting data tampering), authenticity (verifying the identity of the sender), and non-repudiation (preventing the sender from denying their actions). Cryptographic techniques are designed to provide a comprehensive set of security services.

Difference Between Hashing and Encoding

Hashing and encoding are techniques used to transform data, but they serve different purposes and have distinct characteristics. Here's a comparison between hashing and encoding:

Purpose

Hashing: The primary purpose of hashing is data integrity and security. Hash functions are designed to generate a fixed-size hash value (digest) that represents the original data. Hashing verifies data integrity, detects changes or tampering, and securely stores passwords.

Encoding: Encoding is primarily used for data representation and transformation. It is used to convert data from one format to another, such as converting text to binary or encoding special characters for safe transmission.

Reversibility

Hashing: Hashing is a one-way process. Once data is hashed, it is computationally difficult (ideally, practically impossible) to reverse the process and obtain the original data from the hash value. Hash functions are designed to be irreversible, ensuring that the original data cannot be easily derived from the hash.

Encoding: Encoding is typically a reversible process. The encoded data can be decoded back to its original form using the appropriate decoding algorithm or scheme. The encoding and decoding operations are designed to be symmetrical.

Data Loss

Hashing: Hashing is a lossy process, meaning that the original data is not recoverable from the hash value. Hash functions condense the input data into a fixed-size output, discarding any excess information. Due to this property, hash functions can generate the same hash value for different input data (collisions).

Encoding: Encoding is generally a lossless process, ensuring that the original data can be fully recovered from the encoded representation. The encoding scheme maintains all the information of the original data during the transformation.

Security

Hashing: Hashing is commonly used for security purposes. Cryptographic hash functions, specifically designed for security, provide properties like pre-image resistance, second pre-image resistance, and collision resistance. Hashing is used in password storage, digital signatures, and data integrity checks.

Encoding: Encoding is not primarily designed for security. While some encoding schemes, like Base64, can provide a level of obfuscation, they are not intended to protect data from deliberate attacks or ensure data integrity.

Examples

Hashing: Examples of hashing algorithms include MD5, SHA-1, SHA-256, and bcrypt. These algorithms are commonly used for data integrity checks, password storage, and cryptographic protocols.

Encoding: Examples of encoding schemes include Base64, UTF-8 encoding, URL encoding, and HTML encoding. These schemes are

used to transform data between different representations or ensure safe data transmission.

In summary, hashing is a one-way, irreversible process primarily used for data integrity and security, while encoding is a reversible data representation and transformation process. Hashing focuses on generating fixed-size hash values for security purposes, while encoding preserves the original data and allows for recovery.

Summary

In summary, hashing is a useful tool in computer science that helps ensure the integrity of data, securely store passwords, and quickly retrieve information. By converting data into fixed-size hash codes, hashing allows us to verify if data has been tampered with, protect passwords from being easily stolen, and efficiently locate information in large datasets. Although there are some limitations and potential vulnerabilities, hashing remains an essential and widely used technique in various applications.

11 - Cache

Mastering the fundamentals of cache for Systems Design interviews is crucial since this knowledge will be required very often. The cache will significantly increase performance when designing your systems, even more nowadays when systems are in the cloud. Also, the cache makes an even more significant difference when there are many users. If more than 1 million users access part of the system, the cache will make a massive difference in performance.

Let's then explore the most important cache concepts, in what situations we should use them, and the technologies that implement them.

Eviction Cache Policies

Eviction cache policies determine which items are evicted or removed from the cache when it reach its capacity limit. Here are some commonly used eviction cache policies:

LRU (Least Recently Used)

LRU (Least Recently Used) is a caching algorithm that determines which items or data elements should be evicted from a cache when it reaches capacity. The idea behind LRU is to prioritize removing the least recently accessed items from the cache, keeping the most recently accessed items in the cache because they are more likely to be reaccessed.

LRU General Use Cases

We widely use the LRU (Least Recently Used) caching algorithm in various applications and scenarios where efficient caching of frequently accessed data is crucial. Here are some everyday use cases for LRU caching:

Web Caching: LRU is commonly employed to store frequently accessed web pages, images, or other resources in memory. Keeping the most recently accessed content in the cache reduces the load on backend servers and improves response times for subsequent requests.

Database Query Result Caching: In database systems, we can use LRU caching to cache the results of frequently executed database queries. LRU helps avoid repeated database access and speeds up query response times.

File System Caching: File systems often utilize LRU caching to cache frequently accessed files or file blocks. LRU reduces disk I/O operations and improves overall file system performance.

CPU Cache Management: CPUs employ LRU-based cache replacement policies to manage the cache hierarchy effectively. The CPU cache can minimize cache misses and improve overall processor performance by prioritizing the most recently accessed data.

Network Packet Caching: We can use LRU caching to cache frequently accessed network packets or data chunks in networking applications. LRU can enhance network performance by reducing the need for repeated data transmission.

API Response Caching: APIs that serve frequently requested data can benefit from LRU caching. By storing and doing the most recently accessed API responses, the caching layer can reduce the load on backend systems and improve the API's response time.

Compiler Optimization: During the compilation process, compilers often employ LRU caching to store frequently used intermediate

representations or symbol tables. LRU can speed up subsequent compilation stages and improve overall compilation time.

These are just a few examples of using LRU caching in various domains and applications. The primary goal is to reduce latency, improve response times, and optimize resource utilization by keeping frequently accessed data readily available in a cache.

LRU Real-World Use Cases

Let's see some use cases where we could use LRU from famous systems we use daily.

Youtube

Let's explore potential use cases where we can use LRU (Least Recently Used) caching algorithms on YouTube:

Video Thumbnail Caching: YouTube could utilize LRU caching to store and serve frequently accessed video thumbnails. Thumbnails are often displayed in search results, recommended videos, and video player previews. Caching the most recently accessed thumbnails can significantly reduce the backend systems load and improve thumbnail retrieval responsiveness.

Recommended Videos: YouTube's recommendation system could utilize LRU caching to store and serve recommended videos based on a user's viewing history and preferences. Caching recommended videos can improve recommendation response times and reduce the computational load on the recommendation engine.

Instagram

User Profile Caching: Instagram could cache frequently accessed user profiles to improve the response times when displaying pro-

file information, posts, followers, and other profile-related data. Caching user profiles allows for faster retrieval and reduces the load on backend systems, enhancing the user experience.

User Stories: Instagram may cache user stories, including images, videos, stickers, and interactive elements. Caching stories helps ensure smooth playback and reduces the load on the backend systems when displaying stories to users.

Netflix

Let's explore some potential use cases where LRU (Least Recently Used) caching algorithms could be applied on Netflix:

Movie and TV Show Metadata: Netflix could cache frequently accessed movie and TV show metadata, including titles, descriptions, genres, cast information, and ratings. Caching this information improves the responsiveness of search results, recommendation algorithms, and content browsing, reducing the need for repeated database queries.

Video Playback Data: Netflix may cache video playback data, such as user progress, watch history, and subtitle preferences. Caching this data allows users to resume watching a video from where they left off and maintain their preferred settings without requiring frequent backend server communication.

Least Frequently Used (LFU)

LFU (Least Frequently Used) is an eviction cache policy that aims to remove the items accessed the least number of times from the cache. It assumes that items frequently accessed are more likely to be accessed in the future.

Here's how LFU eviction works:

Tracking Access Frequency: LFU maintains a frequency count for each item in the cache. An item's access frequency count is incremented whenever accessed or requested.

Eviction Decision: When the cache reaches its capacity limit and a new item needs to be inserted, LFU examines the access frequency counts of the items in the cache. It identifies the item with the least access frequency or the lowest count.

Eviction of Least Frequently Used Item: The item with the lowest access frequency count is evicted from the cache to make room for the new item. If multiple items have the same lowest count, LFU may use additional criteria, such as the time of insertion, to break ties and choose the item to evict.

Updating Access Frequency: After the eviction, LFU updates the access frequency count of the newly inserted item to indicate that it has been accessed once.

LFU is based on the assumption that items that have been accessed frequently are likely to be popular and will continue to be accessed frequently in the future. By evicting the least frequently used items, LFU aims to prioritize keeping the more frequently accessed items in the cache, thus maximizing cache hit rates and improving overall performance.

However, LFU may not be suitable for all scenarios. It can result in the eviction of items that were accessed heavily in the past but are no longer relevant. Additionally, LFU may not handle sudden changes in access patterns effectively. For example, if an item was rarely accessed in the past but suddenly becomes popular, LFU may not quickly adapt to the new access frequency and may prematurely evict the item.

Therefore, it's important to consider the specific characteristics of the workload and the behavior of the cached data when choosing an eviction cache policy like LFU.

LFU General Use Cases

Here are some potential use cases where the LFU (Least Frequently Used) eviction cache policy can be beneficial:

Web Content Caching: LFU can be used to cache web content such as HTML pages, images, and scripts. Frequently accessed content, which is likely to be requested again, will remain in the cache, while less popular or rarely accessed content will be evicted. This can improve the responsiveness of web applications and reduce the load on backend servers.

Database Query Result Caching: In database systems, LFU can be applied to cache the results of frequently executed queries. The cache can store query results, and LFU ensures that the most frequently requested query results remain in the cache. This can significantly speed up subsequent identical or similar queries, reducing the need for repeated database accesses.

API Response Caching: LFU can be used to cache responses from external APIs. Popular API responses that are frequently requested by clients can be cached using LFU, improving the performance and reducing the dependency on external API services. This is particularly useful when the API responses are relatively static or have limited change frequency.

Content Delivery Networks (CDNs): LFU can be employed in CDNs to cache frequently accessed content, such as images, videos, and static files. Content that is heavily requested by users will be retained in the cache, resulting in faster content delivery and reduced latency for subsequent requests.

Recommendation Systems: LFU can be utilized in recommendation systems to cache user preferences, item ratings, or collaborative filtering results. By caching these data, LFU ensures that the most relevant and frequently used information for generating recommendations is readily available, leading to faster and more

accurate recommendations.

File System Caching: LFU can be applied to cache frequently accessed files or directory structures in file systems. This can improve file access performance, especially when certain files or directories are accessed more frequently than others.

Session Data Caching: In web applications, LFU can cache session data, such as user session variables or session-specific information. Frequently accessed session data will be kept in the cache, reducing the need for repeated retrieval and improving application responsiveness.

It's important to note that the suitability of LFU for a specific use case depends on factors such as access patterns, data popularity, and cache capacity. Each system should evaluate its requirements and consider the pros and cons of different eviction cache policies to determine the most appropriate approach.

Real Use Cases from LFU

Let's see some use cases where we could use LFU from famous systems we use daily.

Youtube

YouTube could use LFU caching for storing and serving frequently accessed video metadata, such as titles, descriptions, thumbnails, view counts, and durations. The LFU cache would prioritize caching the metadata of popular and frequently viewed videos, as users are more likely to watch them.

Instagram

User Feed and Explore Recommendations: Instagram could utilize LFU caching for storing and serving user feeds and exploring recommendations. The user feed displays posts from accounts that users follow, while the explore section suggests content based on the user's interests and browsing history.

Netflix

Netflix could utilize LFU caching for storing and serving frequently accessed movie or TV show metadata, such as titles, descriptions, genres, and cast information. The LFU cache would prioritize caching the metadata of popular and frequently watched content, as they are more likely to be requested by users.

Write Through Cache

A write-through cache is a caching technique that ensures data consistency between the cache and the underlying storage or memory. In a write-through cache, whenever data is modified or updated, the changes are simultaneously written to both the cache and the main storage. This approach guarantees that the cache and the persistent storage always have consistent data.

Here's how the write-through cache works:

Data Modification: When a write operation is performed on the cache, the modified data is first written to the cache.

Write to Main Storage: After the data is written to the cache, it is immediately propagated or "written through" to the main storage or memory. This ensures that the updated data is permanently stored and persisted.

Data Consistency: By writing through to the main storage, the write-through cache ensures that the cache and the underlying storage always have consistent data. Any subsequent read operation for the same data will retrieve the most up-to-date version.

Read Operations: Read operations can be serviced by either the cache or the main storage, depending on whether the requested data is currently present in the cache. If the data is in the cache (cache hit), it is retrieved quickly. If the data is not in the cache (cache miss), it is fetched from the main storage and then stored in the cache for future access.

Advantages of Write-Through Cache

Data Consistency: The write-through cache maintains data consistency between the cache and the main storage, ensuring that modifications are immediately reflected in the persistent storage.

Crash Safety: Since data is written to the main storage simultaneously, even if there is a system crash or power failure, the data remains safe and up-to-date in the persistent storage.

Read Performance: Read operations can benefit from the cache, as frequently accessed data is available in the faster cache memory, reducing the need to access the slower main storage.

Simplicity: Write-through cache implementation is relatively straightforward and easier to manage than other caching strategies.

Disadvantages of Write-Through Cache

Write Latency: The write operation in the write-through cache takes longer as it involves writing to both the cache and the main

storage, which can increase the overall latency of write operations.

Increased Traffic: The write-through cache generates more write traffic on the storage subsystem, as every write operation involves updating both the cache and the main storage.

Limited Write Performance Improvement: Write-through caching doesn't significantly enhance write performance, as the speed of the main storage still limits the writes.

Write-through caching is commonly used in scenarios where data consistency is critical, such as in databases and file systems, where immediate persistence of updated data is essential.

Time-To-Live (TTL): This policy associates a time limit with each item in the cache. When an item exceeds its predefined time limit (time-to-live), it is evicted from the cache. TTL-based eviction ensures that cached data remains fresh and up-to-date.

Random Replacement (RR): This policy randomly selects items for eviction from the cache. When the cache is full, and a new item needs to be inserted, a random item is chosen and evicted. Random replacement does not consider access patterns or frequencies and can lead to unpredictable cache performance.

Memoization

Memoization is a technique used in computer programming to optimize the execution time of functions by caching the results of expensive function calls and reusing them when the same inputs occur again. It is a form of dynamic programming that aims to eliminate redundant computations by storing the results of function calls in memory.

Storing data in a Map is a good example. In the recursive Fibonacci algorithm, we can use a Map to store (cache) repeated Fibonacci numbers to avoid processing a number already computed.

FIFO Cache

FIFO (First In, First Out) cache is a cache eviction policy that removes the oldest inserted item from the cache when it reaches its capacity limit. In other words, the item in the cache for the longest duration is the first to be evicted when space is needed for a new item. Remember, in a real-world queue, if you are the first person in the queue, you will be attended first.

Here's how FIFO cache eviction works:

Insertion of Items: When a new item is inserted into the cache, it is placed at the end of the cache structure, becoming the most recently added item.

Cache Capacity Limit: As the cache reaches its capacity limit and a new item needs to be inserted, the cache checks if it is already full. If it is full, the FIFO policy comes into play.

Eviction Decision: In FIFO, the item that has been in the cache the longest (i.e., the oldest item) is selected for eviction. This item is typically the one at the front or beginning of the cache structure, representing the first item that entered the cache.

Eviction of Oldest Inserted Item: The selected item is evicted from the cache to make room for the new item. Once the eviction occurs, the new item is inserted at the end of the cache, becoming the most recently added item.

FIFO cache eviction ensures that items that have been in the cache for a longer time are more likely to be evicted first. It is a simple and intuitive eviction policy that can be implemented without much complexity.

However, FIFO may not always be the best choice for cache eviction, as it does not consider the popularity or frequency of item accesses. It may result in the eviction of relevant or frequently accessed items that were inserted earlier but are still in demand.

Therefore, it's essential to consider the specific characteristics of the workload and the access patterns of the cached data when selecting an eviction cache policy like FIFO.

LIFO

LIFO (Last In, First Out) cache is a cache eviction policy that removes the most recently inserted item from the cache when it reaches its capacity limit. In other words, the item that was most recently added to the cache is the first one to be evicted when space is needed for a new item.

Here's how LIFO cache eviction works:

Insertion of Items: When a new item is inserted into the cache, it is placed at the top of the cache structure, becoming the most recently added item.

Cache Capacity Limit: As the cache reaches its capacity limit and a new item needs to be inserted, the cache checks if it is already full. If it is full, the LIFO policy comes into play.

Eviction Decision: In LIFO, the item that was most recently inserted into the cache is selected for eviction. This item is typically the one at the top of the cache structure, representing the last item that entered the cache.

Eviction of Most Recently Inserted Item: The selected item is evicted from the cache to make room for the new item. Once the eviction occurs, the new item is inserted at the top of the cache, becoming the most recently added item.

LIFO cache eviction is straightforward and easy to implement since it only requires tracking the order of insertions. It is commonly used in situations where the most recent data has a higher likelihood of being accessed or where temporal locality of data is important.

However, LIFO may not always be the best choice for cache eviction, as it can result in the eviction of potentially relevant or popular items that were inserted earlier but are still frequently accessed. It may not effectively handle access pattern changes or skewed data popularity over time.

Therefore, it's essential to consider the specific characteristics of the workload and the behavior of the cached data when choosing an eviction cache policy like LIFO.

Weighted Random Replacement (WRR)

This policy assigns weights to cache items based on their importance or priority. Low-weight items are more likely to be evicted when the cache is full. WRR allows for more fine-grained control over cache eviction based on item importance.

Size-Based Eviction: This policy evicts items based on size or memory footprint. When the cache reaches its capacity limit, the things that occupy the most space are evicted first. Size-based eviction ensures efficient utilization of cache resources.

What is stale Data?

Stale data refers to data that has become outdated or no longer reflects the current or accurate state of the information it represents. Staleness can occur in various contexts, such as databases, caches, or replicated systems. Here are a few examples:

Caches: In caching systems, stale data refers to cached information that has expired or is no longer valid. Caches are used to store frequently accessed data to improve performance. However, if

the underlying data changes while it is cached, the cached data becomes stale. For example, if a web page is cached, but the actual content of the page is updated on the server, the cached version becomes stale until it is refreshed.

Replicated Systems: In distributed systems with data replication, stale data can occur when updates are made to the data on one replica but not propagated to other replicas. As a result, different replicas may have inconsistent or outdated versions of the data.

Databases: Stale data can also occur in databases when queries or reports are based on outdated data. For example, if a report is generated using data from yesterday, but new data has been added since then, the report will contain stale information.

Managing stale data is important to ensure data accuracy and consistency. Techniques such as cache expiration policies, cache invalidation mechanisms, and data synchronization strategies in distributed systems are used to minimize the impact of stale data and maintain data integrity.

Cache Technologies

There are several technologies and frameworks that implement caching to improve performance and reduce latency. Here are some commonly used technologies for caching:

Memcached: Memcached is a distributed memory caching system that stores data in memory to accelerate data retrieval. It is often used to cache frequently accessed data in high-traffic websites and applications.

Redis: Redis is an in-memory data structure store that can be used as a database, cache, or message broker. It supports various data structures and provides advanced caching capabilities, including persistence and replication.

Apache Ignite: Apache Ignite is an in-memory computing platform that includes a distributed cache. It allows caching of data across multiple nodes in a cluster, providing high availability and scalability.

Varnish: Varnish is a web application accelerator that acts as a reverse proxy server. It caches web content and serves it directly to clients, reducing the load on backend servers and improving response times.

CDNs (Content Delivery Networks): CDNs are distributed networks of servers located worldwide. They cache static content such as images, videos, and files, delivering them from the nearest server to the user, reducing latency and improving content delivery speed.

Squid: Squid is a widely used caching proxy server that caches frequently accessed web content. It can be deployed as a forward or reverse proxy, caching web pages, images, and other resources to accelerate content delivery.

Java Caching System (JCS): JCS is a Java-based distributed caching system that provides a flexible and configurable caching framework. It supports various cache topologies, including memory caches, disk caches, and distributed caches.

Hazelcast: Hazelcast is an open-source in-memory data grid platform that includes a distributed cache. It allows caching of data across multiple nodes and provides features such as high availability, data partitioning, and event notifications.

These are just a few examples, and there are many other caching technologies available, each with its own features and use cases. The choice of caching technology depends on factors such as the application's requirements, scalability needs, and the specific use case.

Buffer VS Cache

Buffer and cache are both used in computing and storage systems to improve performance, but they serve different purposes and operate at different levels. Here's a comparison between buffer and cache:

Buffer

Purpose: A buffer is a temporary storage area that holds data during the transfer between two devices or processes to mitigate any differences in data transfer speed or data handling capabilities.

Function: Buffers help smooth out the flow of data between different components or stages of a system by temporarily storing data and allowing for asynchronous communication.

Data Access: Data in a buffer is typically accessed in a sequential manner, processed or transferred, and then emptied to make room for new data.

Size: Buffer sizes are usually fixed and limited, and they are often small in comparison to the overall data being processed or transferred.

Examples: Buffers are commonly used in I/O operations, network communication, and multimedia streaming to manage data flow between different components or stages. Youtube and Netflix are good examples of the use of buffering.

Cache

Purpose: A cache is a higher-level, faster storage layer that stores copies of frequently accessed data to reduce access latency and improve overall system performance.

Function: Caches serve as a transparent intermediary between a slower, larger storage (such as a disk or main memory) and the requesting entity, providing faster access to frequently used data.

Data Access: Caches use various algorithms and policies to determine which data to store and evict, aiming to keep the most frequently accessed data readily available.

Size: Cache sizes are usually larger compared to buffers, and they can vary in size depending on the system's requirements and available resources.

Examples: Caches are widely used in CPUs (processor cache) to store frequently accessed instructions and data, web browsers (web cache) to store web pages and resources, and storage systems (disk cache) to store frequently accessed disk blocks.

In summary, buffers are temporary storage areas used to manage data transfer or communication between components, while caches are faster storage layers that store frequently accessed data to reduce latency and improve performance. Buffers focus on data flow and synchronization, while caches focus on data access optimization.

Summary

We saw the trade-offs from each cache and when to use them in a real-world situation. Now, let's see the key points from the cache concept:

- Caching is a technique that stores frequently accessed or computationally expensive data in a faster and more accessible layer.
- It aims to improve performance by reducing the time and resources required to retrieve data.

- Caching can be applied in various domains, from web applications and databases to content delivery and distributed computing.
- Write-through cache ensures data consistency by simultaneously writing modifications to both the cache and the main storage.
- Memoization is a form of caching that stores the results of expensive function calls and reuses them when the same inputs occur again.
- Caching techniques include write-through cache, write-back cache, LRU cache, FIFO cache, random replacement cache, LFU cache, ARC, Belady's optimal algorithm, and multi-level caching.
- Each caching technique has its advantages and trade-offs in terms of performance, eviction policies, and memory usage.
- Caching is used to optimize performance, reduce latency, and improve scalability in data-intensive systems.
- Caching can be particularly effective for repetitive computations, recursive algorithms, and dynamic programming solutions.
- The choice of caching technique depends on factors such as access patterns, resource availability, and specific performance requirements.

12 - Specific Storage Paradigms

When designing systems, we need to know the different types of storage paradigms to manipulate data optimally. When we need to design a system that will use graph DB, spatial DB, blob, or time series DB, we need to know in what situation we can apply them.

Therefore, let's explore these data types to ensure we use the right technology for the use case.

Blob Store

Blob (Binary Large Objects) storage is a way to store large amounts of unstructured data, such as images, videos, documents, and other files. It's called "blob" storage because it treats the data as a single unit, or "blob," without organizing it into traditional folders or file structures.

Think of it like a big container where you can put all kinds of things without worrying about how they are organized. Each item you put in the container is called a blob, with a unique name or identifier.

We often use Blob storage by accessing websites or applications to store and retrieve files. For example, if you have a photo-sharing app, you can use blob storage to keep all the user-uploaded photos. Each photo is stored as a separate blob, and the app can easily retrieve and display them when needed.

Blob storage is flexible and scalable, meaning you can add or remove blobs as needed and handle enormous amounts of data. It

is also accessible over the internet to store and retrieve blobs from anywhere in the world.

Blob storage is a simple and effective way to store and manage various types of files without worrying too much about their organization or structure.

When to Use Blob?

We use Blob storage in the following scenarios:

Storing and serving media files: Blob storage is an excellent choice for storing images, videos, audio files, and other media assets. It provides a scalable and efficient solution for serving these files to websites, applications, or content delivery networks (CDNs). Blob storage can handle media files' high throughput and bandwidth requirements, ensuring fast and reliable access.

Backup and disaster recovery: Blob storage is well-suited for backup and disaster recovery. It allows you to store large amounts of data securely and durably. You can use blob storage to create backups of your critical data, ensuring that it is protected and can be restored.

Big data and analytics: When dealing with big data and analytics workloads, blob storage can serve as a cost-effective and scalable storage solution. You can store raw data, logs, and other input files in blob storage and then process and analyze that data using various tools and frameworks such as Apache Spark or Hadoop.

Archiving and long-term storage: We often use Blob storage for long-term archival of data that needs to be retained for regulatory or compliance purposes. It provides a cost-effective option for storing data that may not be accessed frequently but must be preserved for a specified retention period.

Content management and distribution: If you have a content

management system or need to distribute files to multiple locations or users, blob storage can be a convenient solution. It allows you to centralize your content storage and efficiently distribute files to different endpoints or users.

It's important to note that blob storage is particularly suitable for unstructured data, where the organization and structure of the data are not critical. If you have structured data that requires complex querying or relational database functionality, a different storage solution, such as a database, may be more appropriate.

Where to Store Blog Data?

Blob data can be stored in various places depending on your requirements and preferences. Here are some standard options for storing blob data:

Cloud Storage Services: Many cloud providers offer blob storage services as part of their cloud offerings. For example, Amazon S3 (Simple Storage Service), Microsoft Azure Blob Storage, and Google Cloud Storage are popular choices. These services provide scalable, durable, and highly available storage solutions specifically designed for storing blob data. They offer features like data redundancy, access controls, and integration with other cloud services.

On-Premises Storage Systems: If you prefer more control over your data and infrastructure, you can set up and manage your storage systems. Having more control over your data involves deploying dedicated storage hardware and software solutions, such as Network Attached Storage (NAS) or Storage Area Networks (SAN). On-premises storage offers direct control and can be customized to your needs but requires maintenance, hardware investment, and infrastructure management.

Hybrid Storage Solutions: Sometimes, organizations choose a combination of cloud and on-premises storage to create a hybrid

storage environment. This approach allows you to take advantage of the scalability and flexibility of cloud storage for specific data while keeping sensitive or critical data on-premises for security or compliance reasons. Hybrid storage solutions often involve data synchronization and management tools to ensure seamless integration between storage locations.

Object Storage Systems: Object storage systems like OpenStack Swift or Ceph provide a distributed and scalable platform for storing blob data. These systems can be deployed in a private cloud or on-premises infrastructure, and they offer features like data redundancy, fault tolerance, and horizontal scalability. Object storage systems are particularly suitable for large-scale deployments and scenarios where high availability and durability are crucial.

The choice of where to store blob data depends on factors like scalability requirements, data security and compliance, budget considerations, and the level of control you want over your storage infrastructure. Evaluating the features, capabilities, and pricing of different storage options is essential to determine the best fit for your specific needs.

Time Series DB

A time series database (TSDB) is a specialized system that efficiently stores, manages, and analyzes timestamped data. It is optimized for handling large volumes of data points collected regularly over time. Time series data typically includes metrics, measurements, or events that are recorded with a corresponding timestamp.

A time series database mainly used for debugging Microservices in real-time. We can look at the dashboard whenever things go wrong to see what happened. Information such as Microservices latency, throughput, and availability of a Microservice will be easily

trackable.

Here are some key characteristics and features of time series databases:

Timestamped Data: Time series databases store data points with associated timestamps, representing when the data was collected or recorded. The timestamps allow for chronological ordering and efficient retrieval and data analysis over specific time ranges.

High Write and Query Performance: Time series databases are designed to handle high write throughput and provide fast query performance over vast amounts of data. They use various techniques like indexing, compression, and data partitioning to optimize data ingestion and retrieval operations.

Scalability: Time series databases are built to scale horizontally, allowing for the efficient storage and processing of massive amounts of time series data. They can handle data growth and increasing workloads by distributing data across multiple nodes or clusters.

Data Retention Policies: Time series databases often include mechanisms to define data retention policies. These policies specify how long data should be retained in the database and may involve automatic data pruning or archiving to manage storage space efficiently.

Aggregation and Analysis: Time series databases provide built-in functions and tools for aggregating, summarizing, and analyzing time-based data. They can perform operations like downsampling (reducing data resolution over time), interpolation, and complex analytical functions tailored for time series analysis.

Query Flexibility: Time series databases support a range of querying capabilities, including range-based queries to retrieve data within specific time intervals, filtering based on tags or attributes, and advanced analytical queries for anomaly detection, forecasting, or pattern recognition.

Integration with Visualization Tools: Many time series databases

integrate data visualization and analysis tools. Those tools allow users to easily create charts, graphs, and dashboards to visualize time series data and gain insights from the stored information.

Time series databases find applications in various domains, such as IoT sensor data, financial markets, monitoring and observability systems, log analysis, scientific research, etc. They provide efficient storage, retrieval, and analysis of time-based data, enabling organizations to make informed decisions, detect patterns, and derive valuable insights from their time series datasets.

Time Series DB Technologies

Time series database (TSDB) technologies are available, each with features and capabilities. Here are some popular TSDB technologies:

InfluxDB: InfluxDB is an open-source TSDB designed for high write and query performance. It offers a SQL-like query language and supports efficient storage and retrieval of time series data. InfluxDB provides features like data downsampling, retention policies, continuous queries, and integration with visualization tools.

Prometheus: Prometheus is an open-source TSDB primarily used for monitoring and observability. It is designed to collect and store time series data related to metrics and monitoring events. Prometheus offers a flexible query language, efficient storage, and powerful alerting capabilities.

TimescaleDB: TimescaleDB is an open-source relational database built on top of PostgreSQL. It extends PostgreSQL with time series-specific features, making it suitable for handling large volumes of timestamped data. TimescaleDB provides SQL support, automatic data partitioning, compression, and retention policies.

OpenTSDB: OpenTSDB is a distributed and scalable TSDB built on top of Apache HBase. It provides a simple API for storing and

retrieving time series data. OpenTSDB offers features like data compaction, data roll-ups, and support for distributed architectures.

Graphite: Graphite is an open-source TSDB and visualization tool. It is often used for monitoring and graphing time series data. Graphite supports storing numeric time series data and provides a query language for data retrieval.

KairosDB: KairosDB is an open-source TSDB built on top of Apache Cassandra. It offers high write and query performance, scalability, and fault tolerance. KairosDB supports data roll-ups, downsampling, and distributed storage.

Druid: Druid is an open-source distributed columnar data store that can be used as a TSDB. It is designed for real-time analytics and provides efficient storage and querying capabilities for time series data. Druid supports aggregations, filtering, and advanced analytics.

Each TSDB technology has its strengths, and the choice depends on factors such as scalability requirements, performance needs, data retention policies, integration capabilities, and the specific use case or application. Evaluating the features, performance characteristics, and community support of different TSDB technologies is essential to select the most suitable one for your requirements.

Graph DB

A graph database, in simple terms, is a type of database that uses a graph data model to represent and store data. It focuses on the relationships between different entities rather than just the entities themselves.

In a graph database, we organize data as nodes (also known as vertices) and relationships (also known as edges) connecting these nodes. Nodes represent entities, such as people, objects, or

concepts, while relationships represent connections or associations between these entities.

Here's an analogy to help understand graph databases:

Imagine you have a social network like Facebook. In a traditional relational database, you would have separate tables for users, friendships, posts, comments, etc. Each table would store data related to a specific entity or relationship.

In a graph database, you would represent each user as a node. The friendships between users would be represented by relationships connecting the corresponding nodes. Each post, comment, or other entity can also be represented as nodes, with relationships capturing their connections.

The benefit of using a graph database is that it allows for efficient and flexible traversal of relationships between entities. You can easily navigate the graph to find connections, explore paths, or analyze patterns. Graph databases handle complex queries involving traversing multiple relationships and uncovering insights from the underlying network structure.

Graph databases find applications in various domains, such as social networks, recommendation systems, fraud detection, knowledge graphs, and network analysis. They provide a powerful and intuitive way to model, query, and analyze data by emphasizing the relationships and connections between entities.

Graph DB Technologies

Several graph database technologies are available that implement the graph data model and provide efficient storage and querying capabilities for graph data. Here are some popular graph database technologies:

Neo4j: Neo4j is a widely used and mature graph database. It

is known for its native graph storage and processing capabilities. Neo4j allows for the efficient storage and retrieval of nodes, relationships, and properties. It supports a query language called Cypher, which is specifically designed for querying and manipulating graph data.

Amazon Neptune: Amazon Neptune is a fully managed graph database service provided by Amazon Web Services (AWS). It is compatible with the popular property graph model and supports querying using the Gremlin query language and Apache TinkerPop framework. Neptune provides high availability, scalability, and integration with other AWS services.

JanusGraph: JanusGraph is an open-source, distributed graph database that supports massive scalability and high availability. It is the underlying storage layer built on Apache Cassandra or Apache HBase. JanusGraph supports the property graph model and provides advanced graph traversal and indexing capabilities.

TigerGraph: TigerGraph is a scalable, high-performance graph database for handling complex graph analytics. It supports a native parallel graph processing engine and provides a graph query language called GSQL. TigerGraph offers real-time analytics, machine learning integration, and distributed processing capabilities.

ArangoDB: ArangoDB is a multi-model database that supports graph, key-value, and document data models. It provides graph database functionality with support for storing and querying graph data. ArangoDB offers a query language called AQL (ArangoDB Query Language), which supports graph traversal and pattern-matching queries.

When to Use Graph DB?

Graph databases are particularly well-suited for certain situations where relationships and connections between entities play a signif-

icant role. Here are some scenarios in which using a graph database can be advantageous:

Relationship-Centric Data: A graph database shines when your data model heavily emphasizes the relationships and connections between entities. Graph databases represent and query complex relationships, such as social networks, recommendation systems, fraud detection, network analysis, and supply chain management.

Graph Traversal and Path Finding: A graph database provides efficient and expressive traversal capabilities if your application requires traversing and analyzing relationships between entities. Graph databases can quickly navigate paths, identify patterns, and perform graph algorithms like shortest path calculations, graph clustering, and centrality analysis.

Unknown or Evolving Data Schemas: Graph databases are schema-flexible, allowing you to add new node types, relationship types, or properties on the fly. This flexibility is advantageous when dealing with evolving data schemas, where the structure and relationships of the data may change over time or vary across different entities.

Real-Time Recommendations and Personalization: Graph databases are well-suited for recommendation systems that rely on user-item relationships. By leveraging the relationships in the graph, you can efficiently generate personalized recommendations, identify similar items, and discover connections between users based on their preferences and behaviors.

Knowledge Graphs and Semantic Data: Graph databases are widely used for building knowledge graphs, which capture and represent complex relationships between entities in a specific domain. Knowledge graphs enable advanced semantic querying, intelligent search, and reasoning capabilities to extract insights and provide context-aware information.

Complex Data Integration: When you need to integrate and consolidate data from multiple sources with diverse data structures,

a graph database can help. Graph databases provide a unified view of the data by representing relationships between disparate data sources, enabling efficient querying and analysis across the integrated data.

Scalability and Performance: Graph databases can efficiently handle large-scale graph data and perform complex queries on massive graphs. They provide optimized storage and indexing structures that enable fast traversal and retrieval of connected data, making them suitable for applications that require high performance and scalability.

It's important to note that while graph databases excel in scenarios that emphasize relationships, there may be better choices for all types of data and use cases. It's crucial to evaluate your requirements, data model, and query patterns to determine if a graph database fits your application.

Spatial DB

A spatial database, in simple terms, is a type of database that is designed to store and manage spatial or geographic data. It allows for storing, indexing, querying, and analyzing data with a spatial or geographic component.

Spatial data refers to information that represents the physical location, shape, or extent of objects or phenomena on Earth. It can include data like coordinates (latitude and longitude), addresses, boundaries, distances, and shapes of geographic features such as cities, buildings, roads, or natural landmarks.

A spatial database provides specific features and capabilities to handle spatial data effectively. These features typically include:

Spatial Data Types: Spatial databases support specialized data types to represent and store spatial information. These data types,

such as points, lines, polygons, or geometries, allow for the precise representation of spatial objects.

Spatial Indexing: Spatial indexing techniques are used to optimize the storage and retrieval of spatial data. These indexes enable efficient querying and spatial analysis by organizing the data to support fast spatial searches, nearest-neighbor queries, and spatial joins.

Spatial Queries: Spatial databases provide query languages or extensions that allow for the execution of spatial queries. Spatial queries can involve:

- Finding objects within a specific area.
- Determining distances between objects.
- Calculating intersections.
- Performing spatial analysis.
- **Geospatial Functions**: Spatial databases often include built-in functions and operators that enable spatial analysis and processing. These functions can perform operations like buffering, overlay analysis, spatial joins, and transformations, allowing for advanced spatial computations.

Integration with GIS Tools: Spatial databases can integrate with Geographic Information System (GIS) tools and software. This integration enables the visualization, analysis, and manipulation of spatial data using specialized GIS applications.

Spatial databases find applications in various domains, including urban planning, transportation management, environmental analysis, location-based services, and natural resource management. They provide a structured and efficient way to store and analyze spatial data, allowing organizations to make informed decisions and gain insights from their geographic information.

Technologies to Use Spatial DB

There are several technologies available for building and working with spatial databases. Here are some popular technologies commonly used for spatial databases:

PostGIS: PostGIS is an open-source spatial extension for the PostgreSQL relational database management system. It supports spatial data types, indexing, and spatial functions to store, query, and analyze spatial data within PostgreSQL. PostGIS is widely used and provides a robust set of spatial capabilities.

Oracle Spatial: Oracle Spatial is a spatial extension of the Oracle Database. It offers a range of spatial features and functions, including support for spatial data types, spatial indexing, spatial operators, and spatial analysis. Oracle Spatial is used in enterprise-level applications that require high-performance spatial processing.

Microsoft SQL Server Spatial: Microsoft SQL Server includes spatial extensions that enable the storage and analysis of spatial data. It provides spatial data types, indexing, and functions to work with spatial data. SQL Server Spatial is commonly used in Microsoft-based environments for spatial data management.

GeoServer: GeoServer is an open-source server-side software for sharing and serving geospatial data. It supports multiple spatial database backends, including PostgreSQL/PostGIS, Oracle Spatial, and Microsoft SQL Server Spatial. GeoServer allows you to publish spatial data as web services adhering to open geospatial standards.

MongoDB: MongoDB is a NoSQL document database that also supports spatial data. It offers geospatial indexing and querying capabilities, allowing you to efficiently store and retrieve spatial data. MongoDB's geospatial features are suitable for applications that require flexible document-based data storage combined with spatial capabilities.

Elasticsearch: Elasticsearch is a distributed search and analytics

engine that can also handle spatial data. It provides:

- Geospatial indexing and querying features.
- Making it useful for applications that require full-text search. Analytics.
- Spatial querying capabilities.
- QGIS: QGIS is an open-source desktop GIS software with features for managing and analyzing spatial data. While not a database itself, QGIS can connect to various spatial databases, including PostGIS, Oracle Spatial, and others, allowing you to visualize, manipulate, and analyze spatial data stored in these databases.

These are just a few examples of technologies designed explicitly for spatial databases. The choice of a spatial database technology depends on factors such as your application requirements, existing technology stack, performance needs, scalability, and the level of spatial functionality and support you require.

When to Use Spatial DB?

Spatial databases are beneficial when storing, managing, analyzing, and disseminating data with a spatial or geographic component. Here are some scenarios where using a spatial database is beneficial:

Geographic Information Systems (GIS): Spatial databases are commonly used in GIS applications where capturing, storing, analyzing, and visualizing geographic data is essential. GIS applications involve tasks like mapping, spatial analysis, routing, and spatial decision-making, all of which can benefit from the capabilities provided by a spatial database.

Location-Based Services (LBS): Spatial databases are fundamental to location-based services that provide information, recommendations, or services based on the user's location. Examples include mapping applications, ride-sharing services, real-time navigation, geofencing, proximity-based marketing, and asset-tracking systems.

Urban Planning and Infrastructure Management: Spatial databases are crucial in urban planning and infrastructure management. They facilitate storing and analyzing data related to land use, zoning, transportation networks, utility systems, and environmental factors. Spatial databases enable planners and decision-makers to assess the impact of proposed changes, optimize resource allocation, and manage urban infrastructure effectively.

Environmental Analysis and Natural Resource Management: Spatial databases are valuable for managing and analyzing environmental and natural resource data. They can store and analyze data on biodiversity, land cover, ecological habitats, water resources, and climate patterns. Spatial databases enable environmental scientists, conservationists, and resource managers to make informed decisions and assess the impact of various factors on ecosystems.

Retail Site Selection: Spatial databases can assist in retail site selection by analyzing demographic data, foot traffic patterns, competitor locations, and other spatial factors. Retail companies can leverage spatial databases to identify optimal store locations, understand customer behavior, and optimize their market presence.

Emergency Management and Disaster Response: Spatial databases are critical in emergency management and disaster response scenarios. They can store and analyze data related to emergency services, evacuation routes, hazard zones, and resource allocation. Spatial databases enable planners and responders to make informed decisions and coordinate their efforts during emergencies.

Transportation and Logistics: Spatial databases are beneficial in managing transportation networks, optimizing logistics operations,

and route planning. They can store and analyze data on road networks, traffic patterns, vehicle tracking, and delivery routes. Spatial databases enable efficient logistics planning, real-time monitoring, and route optimization for transportation companies.

These are just a few examples of situations where spatial databases are valuable. Any scenario that involves managing, analyzing, and querying data with a spatial component can benefit from the capabilities provided by a spatial database.

In summary, a spatial database is a specialized database designed to handle spatial or geographic data. It provides features like spatial data types, indexing, querying, and analysis capabilities to manage and work with spatial information effectively.

Summary

The article discussed four distinct types of databases: Blob storage, time series databases, graph databases, and spatial databases.

Blob storage is a reliable and scalable solution for storing unstructured data, such as media files, offering efficient cloud storage and content delivery capabilities.

Time series databases, explicitly designed for timestamped data, excel in managing and analyzing time-based information, making them essential for IoT applications, monitoring systems, and financial analytics.

Graph databases, on the other hand, specialize in representing and querying complex relationships, making them valuable for social networks, recommendation systems, fraud detection, and supply chain management.

Lastly, spatial databases focus on storing and analyzing spatial or geographic data, enabling efficient management of location-based services, urban planning, and environmental analysis.

Each type of database offers unique features and strengths, catering to specific data storage and processing requirements. Understanding the distinct characteristics of these databases empowers organizations to select the most suitable option based on their particular use cases.

Combining databases might be necessary in specific scenarios to address complex data management needs, allowing for a more holistic approach to data storage and analysis.

13 - Database Replica and Sharding

Most Microservices must scale databases, and the concepts of database replica and database sharding are vital to understanding when to use them for systems design and systems design interviews.

Database replica will replicate the database in different servers to keep the database more available and avoid losing data. It's even possible to replicate servers in different countries for disaster recovery.

Database sharding will break the database into separate partitions and give faster data access. Those partition shardings can be distributed across different servers, improving performance, scalability, and availability.

Database Replica

Database replication is the process of creating and maintaining copies of a database on multiple servers, known as replicas. Replication is commonly used in distributed systems to improve data availability, enhance performance, and ensure data durability. It involves synchronizing data changes from a source database to one or more replica databases in a controlled and consistent manner.

Here's an overview of how database replication works:

Replication Topology: A replication topology defines the relationship between the source database and replica databases. Different

replication topologies exist, including master-slave, master-master, and multi-level replication.

Source Database: The source database is the primary database that holds the original and authoritative copy of the data. It is responsible for processing read and write operations and propagating data changes to the replicas.

Replica Databases: Replica databases are copies of the source database. They receive and apply data changes from the source database to keep their data in sync. Replica databases can be located on the same server as the source database or distributed across different servers for improved performance and availability.

Data Replication: Data replication involves capturing and transmitting data changes from the source database to the replicas. When a write operation (such as an insert, update, or delete) occurs on the source database, we record the change in a replication or transaction log.

Replication Process: The replication process reads the changes recorded in the replication log and applies them to the replica databases. Depending on the database system and replication technology, we do database replication through various mechanisms, such as log-based, statement-based, or trigger-based.

Consistency and Synchronization: Maintaining data consistency and synchronization is crucial in database replication. Depending on the replication method, we can use various techniques to ensure we apply data changes to the replicas in the same order and with the same consistency as the source database.

Types of Database Replica

There are several database replicas, each with its characteristics and purposes. Here are some common types:

Full Replication: The entire database is copied to multiple servers, creating identical copies of the original database. Any changes made to the original database are propagated to all replicas. This type of replication provides high availability and fault tolerance, as any replica can take over if the primary database fails.

Snapshot Replication: Snapshot replication involves taking periodic snapshots of the database and distributing them to replicas. The snapshots capture the state of the database at a specific point in time, and subsequent changes are not propagated to the replicas. This type of replication is useful when read-only access to a particular database version is required.

Transactional Replication: Transactional replication involves replicating individual database transactions from the primary database to the replicas. Each transaction is applied to the replicas in the same order it was executed on the primary database. This type of replication is commonly used in scenarios where real-time data synchronization is required.

Merge Replication: Merge replication allows multiple replicas to independently modify the database and then merge the changes into a consolidated version. This type of replication is useful in scenarios where multiple users need to work with their local copies of the database and periodically synchronize changes with the central database.

Peer-to-Peer Replication: Peer-to-peer replication involves multiple databases acting as publishers and subscribers. Each database can independently make changes, which are propagated to other databases in the network. This type of replication is beneficial for distributed systems where data needs to be shared and synchronized across multiple locations or nodes.

Log Shipping: Log shipping involves shipping transaction log backups from the primary database server to one or more standby servers. The standby servers restore the transaction logs and apply them to maintain an up-to-date copy of the database. This type of

replication is often used for disaster recovery purposes.

These are just a few examples of the types of database replicas. The choice of replication method depends on factors such as the desired level of data consistency, availability requirements, network topology, and specific use cases.

When to Use Database Replica?

A database replica is a copy of a database created and maintained for various purposes, such as improved performance, high availability, and disaster recovery. Here are some scenarios where using a database replica is beneficial:

Load Balancing and Performance: If your application has a high volume of read operations, you can distribute the read workload across multiple replicas. The primary database server can focus on handling write operations by offloading read queries to replicas, improving overall performance.

High Availability: Database replicas can provide fault tolerance and ensure high data availability. If the primary database server fails, one replica can take over as the new primary server, minimizing downtime and ensuring continuous operation.

Disaster Recovery: Replication can be part of a disaster recovery strategy. By maintaining a replica in a separate geographic location or on a different server, you can quickly switch to the replica in case of a catastrophic event, ensuring data integrity and minimizing data loss.

Scaling: We can use replication to scale the database infrastructure horizontally. We can add more replicas to handle the increased demand as the workload increases. This approach allows you to scale read capacity independently from the primary database server.

Backup and Reporting: We can use replicas for performing backups and generating reports without impacting the performance of the primary database. We can offload resource-intensive tasks from the primary server by running backup jobs or generating reports on replicas.

It's important to note that while database replication offers benefits, it also introduces considerations such as additional complexity, data consistency challenges, and potential latency between the primary and replica databases.

Therefore, it's crucial to carefully design and configure database replication based on your specific requirements and workload characteristics.

Real World Use Cases for Database Replica

Here are some real-world use cases where using a database replica can be beneficial:

E-commerce Applications: In e-commerce applications, quick and reliable access to product information is essential. Using database replicas, you can distribute read traffic to handle a high volume of product searches, inventory lookups, and catalog browsing while ensuring the primary database remains available for write operations.

Content Delivery Networks (CDNs): CDNs serve static content to users from edge servers located in various geographical regions. To minimize latency and improve user experience, CDNs often replicate their databases across multiple locations. This allows the edge servers to retrieve content from nearby replicas, reducing the round-trip time for database queries.

Financial Systems: Financial institutions require high availability

and data integrity. Organizations can use database replicas to ensure uninterrupted access to financial data and provide failover capabilities in case of system failures. Replicas can also be used for reporting and analysis, allowing financial analysts to generate insights without impacting the primary system.

Social Media Platforms: Social media platforms handle massive user-generated content and interactions. By utilizing database replicas, these platforms can distribute the read load across multiple replicas to provide fast access to user profiles, posts, comments, and other social content.

Analytics and Business Intelligence: Organizations often have separate transactional systems and analytics databases. Replicating data from the transactional database to an analytics database allows for efficient data analysis without impacting the operational system's performance. This setup enables businesses to generate reports, perform complex queries, and gain insights from historical data.

Mobile Applications: Mobile apps frequently require real-time access to data, and network conditions can be unpredictable. Mobile applications can provide a responsive user experience with reduced latency by using local replicas on mobile devices or in close proximity to the users.

These are just a few examples, and the use of database replicas can be beneficial in various other scenarios depending on the application's specific requirements and the organization's needs.

Technologies for Database Replica

There are several technologies and approaches commonly used for implementing database replication:

Master-Slave Replication: This is one of the simplest forms of database replication. It involves a primary/master database server

that accepts both read and write operations and one or more secondary/slave servers that replicate data from the primary server. Read operations can be distributed to the slave servers, offloading the read workload from the primary server.

Master-Master Replication: In this setup, multiple database servers act as both master and slave, allowing read and write operations on each server. Changes made on one server are replicated on other servers, ensuring data consistency. Master-master replication provides higher availability and load-balancing capabilities.

Asynchronous Replication: Asynchronous replication allows the primary database server to continue processing transactions without waiting for the replicas to acknowledge the changes. Replicas receive and apply changes from the primary server at their own pace, introducing a slight delay in data synchronization.

Synchronous Replication: In synchronous replication, the primary server waits for replicas to acknowledge the changes before confirming the transaction's success. It ensures that data changes are immediately applied to replicas, providing stronger consistency guarantees but potentially impacting the primary server's performance.

Log Shipping: Log shipping involves copying and applying transaction logs from the primary server to the replica servers. The logs are periodically shipped and applied to keep the replica databases up to date. We commonly use this approach in disaster recovery scenarios.

Database Clustering: Clustering involves multiple interconnected database servers operating as a single system. Clusters typically provide high availability, load balancing, and automatic failover capabilities. Various clustering technologies exist for different database systems, such as MySQL Cluster, Oracle RAC, or Microsoft SQL Server Always On Availability Groups.

It's worth noting that different database management systems

(DBMS) offer their native replication features, such as MySQL replication, PostgreSQL streaming replication, or SQL Server Always On Availability Groups. Additionally, third-party replication tools and solutions are available in the market, such as Oracle Data Guard, VMware vSphere Replication, or Amazon RDS Multi-AZ deployment for AWS. The choice of technology depends on the specific DBMS and requirements of the application or organization.

Database Sharding

Sharding is a database partitioning technique used to distribute and store data across multiple database servers, known as shards. It is primarily employed in large-scale, high-traffic systems to improve performance, scalability, and availability.

In a traditional database setup, we store in a single server. As the amount of data and the number of concurrent users increases, this can lead to performance bottlenecks and limit the system's ability to handle the load. Sharding addresses this issue by horizontally partitioning the data into smaller subsets and distributing them across multiple servers.

Here's a simplified explanation of how sharding works:

Data Partitioning: The database is divided into smaller logical partitions called shards. Each shard contains a subset of the data based on a defined partitioning strategy. For example, you might partition data based on a specific range of values (e.g., customer IDs) or use a hashing algorithm that evenly distributes the data.

Shard Distribution: We distribute the shards across multiple database servers. Each server is responsible for storing and managing one or more shards. The number of servers and shards can vary based on the application's requirements.

Query Routing: A client application must determine which shard contains the relevant information when it wants to read or write

data. Based on the partitioning strategy, a query router or load balancer sits between the client and the database servers and redirects the queries to the appropriate shard.

Data Consistency: Maintaining data consistency is a crucial aspect of sharding. There are different approaches to handle consistency, depending on the system requirements. A common practice is to store related data within the same shard. Furthermore, we can employ distributed transactions or eventual consistency mechanisms to synchronize data across shards.

Real World Use Cases for Database Sharding

Database sharding is suitable in various real-world situations where the data volume and workload requirements of a system exceed the capabilities of a single database server. Here are some specific real-life scenarios where sharding can be beneficial:

E-commerce Applications: E-commerce platforms often handle a large volume of data, including product catalogs, customer information, and transaction records. Sharding can distribute the data across multiple servers, ensuring high performance and scalability during peak shopping seasons.

Social Networking Platforms: Social media platforms generate enormous amounts of user-generated content, such as posts, images, and videos. Sharding can help handle the massive data load and user concurrency, enabling efficient storage, retrieval, and real-time interactions.

Gaming Applications: Online gaming applications often deal with high user concurrency, real-time interactions, and large datasets. We can use sharding to distribute player data, game states, and matchmaking information across multiple servers to ensure smooth gameplay and fast response times.

IoT Data Management: Internet of Things (IoT) applications generate vast amounts of data from connected devices. We can use sharding to handle the scale and velocity of incoming data, enabling efficient storage, analysis, and processing of IoT data.

Big Data Analytics: Sharding can be beneficial in distributed big data analytics systems. By sharding data based on specific criteria (e.g., time ranges or geographical regions), queries and analysis can be performed in parallel across multiple shards, improving query performance and enabling scalable data processing.

Content Management Systems: Content-heavy applications, such as news websites or multimedia platforms, may experience high traffic and require efficient storage and retrieval of content. Sharding can help distribute content across servers, enabling faster content delivery and reducing the load on individual servers.

SaaS Applications: Software-as-a-service (SaaS) providers often serve multiple customers with diverse data requirements. Sharding can isolate customer data, ensuring data privacy and providing scalability as the number of customers grows.

Evaluating your application's specific requirements, anticipated data growth, and performance needs is essential before deciding to implement database sharding. Furthermore, it is crucial to consider data consistency, query complexity, and shard management during the planning and implementation.

Benefits of Sharding

Improved Performance: Sharding allows database operations to be distributed across multiple servers, enabling parallel processing and reducing the load on individual servers. Parallel processing can result in improved query response times and increased throughput.

Scalability: Sharding allows scaling a database system horizontally by adding more servers as the data or workload grows. Each shard

can be hosted on a separate server, allowing the system to handle increased data volume and user concurrency.

High Availability: Sharding enhances fault tolerance and availability by distributing data across multiple servers. If one server or shard goes down, the remaining shards can continue to serve requests, reducing the impact of failures on the overall system.

Challenges of Sharding

Data Distribution Complexity: Sharding introduces database design and application logic complexity. It requires careful data partitioning and proper query routing to maintain data integrity and consistency.

Joining Data: Sharding complicates the joining of data from different shards because it distributes the data across multiple servers. Complex queries that involve various shards may need extra coordination and processing.

Shard Management: Managing the distribution, scaling, and rebalancing of shards as the system evolves can be complex. It requires monitoring the data distribution, adding or removing shards, and redistributing data without disrupting the system's operation.

It's important to note that sharding is only sometimes necessary or suitable for every application. We typically use sharding in scenarios where the data and workload exceed the capabilities of a single server.

Types of Database Sharding

Database sharding is a technique used to horizontally partition a database into smaller, more manageable pieces called shards.

Each shard contains a subset of the data, allowing for improved scalability and performance. Here are some common types of database sharding:

Hash-based Sharding: In hash-based sharding, a hash function is applied to a shard key (such as a primary key or a specific field) to determine which shard the data should be assigned to. The hash function distributes the data evenly across the shards based on the key's value. This approach ensures a relatively even distribution of data but can make it challenging to perform range-based queries efficiently.

Range-based Sharding: Range-based sharding involves dividing the data based on a predefined range of values. For example, a shard may contain data for a specific range of customer IDs or time periods. Range-based sharding allows for efficient range queries, but it requires careful planning to ensure an even distribution of data and to handle data growth and rebalancing.

List-based Sharding: List-based sharding involves explicitly assigning specific data values or criteria to different shards. For example, you could assign customers from a specific region to one shard and customers from another region to a different shard. This approach provides more control over shard distribution but requires manual configuration and maintenance.

Directory-based Sharding: Directory-based sharding maintains a separate directory or metadata layer that maps data to the appropriate shard. The directory contains information about the shard location for each data item based on predefined rules or configurations. This approach provides flexibility and ease of management but introduces an additional layer of complexity and potential performance overhead.

Database-specific Sharding: Some databases, such as MongoDB and Apache Cassandra, provide built-in sharding capabilities. These databases offer proprietary sharding mechanisms tailored to their specific architecture and features. For example, MongoDB

uses a sharding method called range-based partitioning, while Cassandra uses a consistent hashing algorithm combined with virtual nodes.

It's worth noting that the choice of sharding method depends on various factors, including the nature of the data, query patterns, scalability requirements, and the capabilities provided by the database management system being used. Additionally, some systems may employ a combination of sharding techniques to optimize performance and accommodate different types of data.

Technologies for Database Sharding

There are several technologies and approaches available for implementing database sharding. The choice of technology depends on factors such as the specific database system, the requirements of the application, and the scalability needs. Here are some commonly used technologies for implementing database sharding:

Database-specific Sharding: Some databases provide built-in sharding capabilities. For example, MongoDB has a built-in sharding feature called "MongoDB Sharding" that allows you to distribute data across multiple servers. Similarly, other NoSQL databases like Cassandra and Couchbase also provide native sharding capabilities.

Consistent Hashing: Consistent hashing distributes data across multiple decentralized shards. It allows for easy scalability and rebalancing of data. Tools like Apache Kafka and Redis Cluster use consistent hashing for sharding.

Virtual Sharding: Virtual sharding involves using a middleware layer or proxy between the application and the database servers. It abstracts the sharding logic from the application and handles the routing of queries to the appropriate shard. Tools like Vitess

and ProxySQL provide virtual sharding capabilities for MySQL databases.

Shared-Nothing Architecture: In a shared-nothing architecture, each database server has its dedicated resources and does not share memory, disks, or processors with other servers. This approach simplifies sharding, as data can be partitioned and distributed across servers without complex coordination. Tools like Google Cloud Spanner and CockroachDB implement shared-nothing architectures with built-in sharding capabilities.

Distributed SQL Databases: Distributed SQL databases provide a distributed and horizontally scalable SQL database solution. They typically provide automatic sharding and replication capabilities while still offering SQL compatibility. Examples of distributed SQL databases include YugabyteDB, CockroachDB, and TiDB.

Cloud-based Database Sharding: Many cloud providers offer managed database services with built-in sharding capabilities. For example, Amazon RDS for MySQL and PostgreSQL, Azure SQL Database, and Google Cloud Spanner provide sharding features as part of their managed database offerings.

It's important to note that the suitability of these technologies depends on various factors, including the specific requirements of your application, the database system, and the scalability needs. We must thoroughly evaluate and test different technologies before selecting the best fit for the use case.

Summary

Let's recap the concepts of database replica and database sharding.

Database Replica:

A copy of a database used for improved performance, high availability, and disaster recovery. Distributes read workload across

replicas, allowing the primary server to focus on write operations. Provides fault tolerance and continuous operation in case of primary server failure. Supports disaster recovery by maintaining replicas in separate locations. Enables scaling of read capacity independently from the primary server. Can be used for backups and generating reports without impacting the primary server.

Database Sharding:

- Divides a database into smaller, independent shards to distribute data and workload.
- Improves performance by parallelizing data access and query execution.
- Allows horizontal scaling by adding more shards as the data and workload grows.
- Reduces contention and improves the overall throughput of the system.
- Requires careful planning and consideration to ensure data distribution and query routing.
- Can be used in scenarios with large data volumes, high write rates, and complex queries.
- Both database replica and sharding are techniques used to enhance database performance, scalability, and availability, but they address different aspects of database management. Database replica focuses on data redundancy, high availability, and workload distribution, while database sharding focuses on horizontal scalability and performance optimization by partitioning data.

14 - Leader Election

We use the concept of leader election more often than we think with MicroServices and Systems Design. In the Systems Design interview, it's also a crucial concept to know because that will make a difference when designing a highly scalable system.

The good news is that we don't need to know the behind-the-scenes algorithms or the ins and outs of leader election. We need to know instead when to apply this concept to build scalable MicroServices.

By knowing this fundamental, it will be much easier to master technologies and tools that implement this concept. Mastering fundamentals is the biggest hack to learn anything faster as a software engineer. Being curious is also very powerful to be a fantastic software engineer.

What is Leader Election?

Leader election is a process in distributed systems where a group of nodes or processes choose a single leader to coordinate their actions. The leader is responsible for making decisions and managing the overall behavior of the group.

The purpose of leader election is to ensure that there is a designated leader at all times, even in the presence of failures or changes in the system. Having a leader helps achieve coordination, efficient communication, and consistency among the nodes.

Here's a simple explanation of how leader election works:

Initial State: In the beginning, all the nodes in the system are equal, and there is no designated leader.

Election Process: When a node detects no leader or the current leader has failed, it initiates the election process. The node sends a message or a signal to all other nodes indicating its desire to become the leader.

Election Messages: Upon receiving an election message, a node compares the priority or identifier of the sender with its own. The node with the highest priority becomes a candidate for the leader position.

Acknowledgment: If a node receives an election message from a node with a higher priority than itself, it acknowledges the message and withdraws its candidacy. This process ensures that only the node with the highest priority continues in the election.

Election Result: Eventually, only one node will remain as the candidate with the highest priority. This node declares itself the leader and broadcasts a victory message to all other nodes.

Leader Acknowledgment: Upon receiving the victory message, all other nodes recognize the new leader and acknowledge its leadership.

Coordination: Once the leader is elected and acknowledged, it assumes the responsibility of coordinating the actions of the nodes in the system. It may initiate further communication, assign tasks, or make decisions on behalf of the group.

It's important to note that leader election algorithms can vary depending on the specific requirements and characteristics of the distributed system. Various approaches and protocols, such as the Bully Algorithm, Ring Algorithm, or Paxos, can be used to implement leader election in different scenarios.

Challenges of Leader Election

Leader election in distributed systems can pose several challenges. Here are some common challenges that may arise during the leader election process:

Network Communication: Leader election relies on the nodes communicating with each other to exchange messages and determine the leader. However, the network connections between nodes may be unreliable, resulting in message delays, packet loss, or network partitions. These communication issues can lead to inconsistencies and difficulties in reaching a consensus on the leader.

Node Failures: In a distributed system, nodes can fail due to hardware or software issues. When a node fails during the leader election process, it can disrupt the process and potentially lead to an incorrect or inconsistent leader selection. Handling node failures and ensuring fault tolerance is a critical challenge in leader election.

Concurrent Elections: In some cases, multiple nodes may initiate leader election simultaneously due to network delays or failures. Concurrent elections can cause conflicts and result in the selection of multiple leaders, leading to inconsistent behavior and coordination issues within the system. Coordinating and resolving concurrent elections is challenging to ensure a single, stable leader.

Scalability: As the number of nodes in a distributed system increases, the complexity of leader election also grows. Scalability becomes a challenge as the overhead of communication and coordination increases with a larger number of nodes. Leader election algorithms need to be designed to handle the scalability requirements of the system.

Timing and Synchronization: Leader election algorithms often rely on timing and synchronization assumptions. However, in distributed systems, nodes may have different clocks or experience

clock drift, making it challenging to establish a common notion of time. Ensuring accurate timing and synchronization among nodes is crucial to avoid inconsistencies and conflicts in leader elections.

Byzantine Failures: In some cases, nodes in a distributed system may exhibit malicious or faulty behavior, known as Byzantine failures. These nodes can intentionally send incorrect or misleading messages during the leader election process, leading to incorrect leader selection or system disruption. Protecting against Byzantine failures is a challenge in ensuring the integrity and correctness of leader elections.

Addressing these challenges requires careful design and implementation of leader election algorithms, considering network conditions, fault tolerance, concurrency, scalability, and security factors. Various distributed consensus protocols, such as Paxos or Raft, have been developed to tackle these challenges and provide reliable leader election mechanisms in distributed systems.

Technologies for Leader Election

Several technologies and algorithms are available for implementing leader election in distributed systems. Here are some commonly used ones:

ZooKeeper: ZooKeeper is a popular open-source distributed coordination service that provides primitives for implementing leader election. It offers a high-performance and reliable platform for managing distributed systems. ZooKeeper uses the ZAB (ZooKeeper Atomic Broadcast) protocol, which ensures consistency and fault tolerance in leader elections.

Apache Kafka: Apache Kafka is a distributed streaming platform that we can use for leader election scenarios. Kafka uses a distributed consensus protocol called Apache Kafka's Controller Election Protocol (KCEP) to elect a leader among the Kafka brokers.

The elected leader manages metadata and coordinates the activities of the Kafka cluster.

etcd: etcd is a distributed key-value store that we can often use as a coordination service in distributed systems. It provides a simple API for storing and retrieving data in a distributed manner. We can leverage etcd for leader election by utilizing its watch feature to monitor changes in a specific key representing the leader position.

Apache Mesos: Apache Mesos is a distributed systems kernel that enables the efficient sharing of resources across clusters. Mesos includes a built-in leader election mechanism called the Master-Slave architecture. In this architecture, multiple Mesos masters compete to be the leader, and the elected leader manages resource allocation and scheduling decisions for the Mesos cluster.

Consul: Consul is a service mesh and service discovery tool that also offers leader election capabilities. It uses the Raft consensus algorithm to elect a leader among a cluster of nodes. The elected leader can then manage coordination and synchronization tasks within the system.

Raft: Raft is a consensus algorithm that can be used for leader election in distributed systems. It ensures strong consistency and fault tolerance in the presence of failures. Raft divides the nodes into three roles: leader, follower, and candidate. Through a series of message exchanges and voting, a leader is elected among the candidates.

Paxos: Paxos is another consensus algorithm commonly used for leader election. It provides a fault-tolerant approach to ensure consistency and agreement among distributed nodes. Paxos elects a leader through a series of message exchanges and voting rounds, and the elected leader coordinates the actions of the nodes.

These technologies and algorithms provide different features, trade-offs, and capabilities for leader election in distributed systems. The choice of technology depends on the specific requirements, scale, and characteristics of the system being built.

Summary

Now that we explored the key points from the leader election concept let's recap:

- Leader election is a process used in distributed systems to select a single node as the leader or coordinator.
- It ensures that there is a designated leader responsible for making decisions and coordinating actions within the distributed system.
- Leader election algorithms are designed to handle scenarios where the current leader fails, becomes unavailable, or needs to be replaced. Various leader election algorithms exist, such as the Bully algorithm, the - - Ring algorithm, the Paxos algorithm, and the Raft algorithm.
- Leader election algorithms typically involve a set of nodes communicating and exchanging messages to determine the most suitable candidate for the leader position.
- Nodes may use criteria such as their priority, availability, or a unique identifier to establish the leader.
- The leader is responsible for tasks like resource allocation, task distribution, and consistency maintenance in the distributed system.
- Leader election is crucial for maintaining the system's resilience, fault tolerance, and overall stability. Once a leader is elected, it typically remains in power until it fails, loses connectivity, or a new leader election is triggered due to specific conditions. Leader election algorithms often incorporate mechanisms to handle scenarios like network partitions, where the distributed system may split into multiple groups with different leaders.

15 Peer-to-peer

A peer-to-peer network will help a file be downloaded by multiple machines much faster, so P2P is a crucial concept to consider for Systems Design and Systems Design interviews.

A peer-to-peer (P2P) network is a computer network where all the connected devices, called peers, share resources directly without needing a central server. In a P2P network, each device acts as a client and a server, meaning it can request and provide resources simultaneously.

Think of it like a group of friends sitting in a circle, where everyone can share things directly. Instead of going through a central authority or relying on a single person to distribute resources, each friend can share their resources and access resources from others in the group.

For example, in a file-sharing P2P network, if you have a file that someone else wants, they can download it directly from your computer, and vice versa. This direct sharing of resources among peers allows for decentralized and distributed file sharing, communication, or any other type of collaboration within the network.

We often use P2P networks for applications like file sharing (e.g., BitTorrent), decentralized cryptocurrencies (e.g., Bitcoin), torrent downloads, and communication platforms (e.g., Zoom). They offer advantages such as increased scalability, fault tolerance, and the ability to operate without relying on a single point of failure.

What is the problem Peer-To-Peer Networks Solve?

If 10 machines download 100MB from one machine with an upload speed of 10 Mbps, and each machine has an internet speed of 100 Mbps with no latency, we can calculate the time it would take for all the machines to complete the download.

First, let's convert the file size and network speeds to the same units:

100MB = 800 Mb (1 byte = 8 bits) 10 Mbps = 10 Mb/s (megabits per second) 100 Mbps = 100 Mb/s (megabits per second)

To calculate the time required for a single machine to download the file: Time = File Size / Download Speed Time = 800 Mb / 100 Mb/s Time = 8 seconds

Since there are 10 machines, and each machine takes 8 seconds to download the file, we can calculate the total time required for all machines to finish downloading:

Total Time = Time per Machine * Number of Machines Total Time = 8 seconds * 10 Total Time = 80 seconds

Therefore, it would take a total of 80 seconds for all 10 machines to download 100MB from the machine with an upload speed of 10 Mbps, assuming no latency or network bottlenecks.

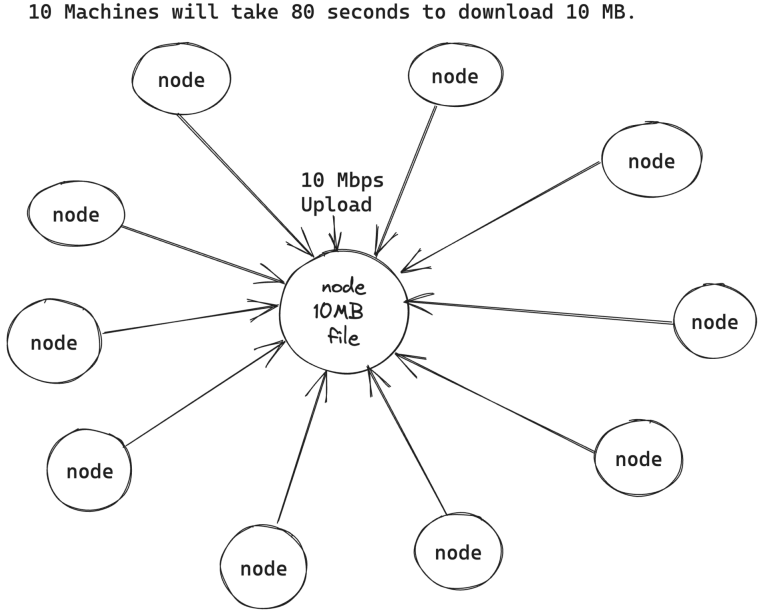


Figure 14. 10 Machines Download 10MB

Solution from Peer-to-Peer Network

The overall download time can be significantly reduced if we apply a peer-to-peer strategy, where each machine can simultaneously upload and download files to/from other machines.

In a peer-to-peer scenario, all machines can share the workload by distributing the file among themselves. Let's consider the following approach:

- Each machine divides the 100MB file into smaller chunks, for example, 10MB each.
- Each machine simultaneously uploads its assigned chunk to multiple machines.

- Each machine concurrently downloads the chunks assigned to it from other machines.
- Assuming there is no latency or network bottlenecks, and each machine maintains its upload speed of 10 Mbps and download speed of 100 Mbps, we can calculate the time required for the overall download.

Since each machine uploads 10MB (80 Mb) and its upload speed is 10 Mbps, the upload time for each machine would be:

- Upload Time per Machine = Upload Size / Upload Speed
- Upload Time per Machine = 80 Mb / 10 Mbps
- Upload Time per Machine = 8 seconds

Since there are 10 machines, they can all upload their chunks simultaneously. Therefore, the upload time remains the same.

Now, each machine needs to download the remaining 90 chunks (900MB) from other machines. Since each machine has a download speed of 100 Mbps, the download time for each machine would be:

- Download Time per Machine = Download Size / Download Speed
- Download Time per Machine = 900 Mb / 100 Mbps
- Download Time per Machine = 9 seconds

Again, all machines can download their assigned chunks simultaneously. Therefore, the download time remains the same.

In this scenario, the upload time is 8 seconds, and the download time is 9 seconds per machine. Since these tasks can be performed concurrently, the overall time required for all machines to finish downloading will be equivalent to the longest individual task, which is 9 seconds.

Therefore, with the peer-to-peer strategy, it would take approximately 9 seconds for all 10 machines to download the 100MB file, assuming no latency or network bottlenecks.

Types of Peer-to-Peer Network

Peer-to-peer (P2P) networks are a type of distributed network architecture where participants in the network, called peers, can directly communicate and share resources without needing a centralized server. There are several types of P2P networks, including:

Pure P2P Network: In a pure P2P network, all peers have equal capabilities and can act as clients and servers. Each peer can initiate and respond to requests for resources or services from other peers, and no central authority or server is controlling the network.

Hybrid P2P Network: A hybrid P2P network combines elements of both P2P and client-server architectures. It usually includes a central server or a set of super-peers that provide indexing, discovery, or other coordination services to facilitate the functioning of the network. Peers in the network can still communicate directly with each other. Still, the central server or super-peers help with tasks such as peer discovery, resource indexing, or maintaining network stability.

Structured P2P Network: In a structured P2P network, peers organize themselves into a specific structure or overlay network. This structure enables efficient resource discovery and routing by maintaining consistent lookup tables or distributed hash tables (DHTs). Examples of structured P2P networks include Chord, CAN (Content-Addressable Network), and Pastry.

Unstructured P2P Network: In an unstructured P2P network, there is no specific organization or structure among the participating peers. Peers connect randomly or through some form of ad hoc communication, and resource discovery relies on flooding or random-walk-based search algorithms. Examples of unstructured P2P networks include Gnutella and Freenet.

Overlay P2P Network: An overlay P2P network is built on top of an existing network infrastructure, such as the Internet. It uses

the underlying network's communication capabilities to establish connections between peers. The overlay network provides an additional layer of abstraction and enables peers to discover and communicate with each other. Examples of overlay P2P networks include BitTorrent and Skype.

These are some common types of P2P networks, each with its own advantages and use cases. The choice of network type depends on scalability, efficiency, resource requirements, and the specific application or use-case requirements.

Technologies that Use Peer-to-peer

Software engineers have several technologies and frameworks at their disposal to implement peer-to-peer (P2P) networks. Here are a few commonly used ones:

Libp2p: Libp2p is a modular networking stack designed specifically for P2P applications. It provides a set of protocols, libraries, and tools that enable developers to build decentralized and distributed systems. Libp2p supports various transport protocols, peer discovery mechanisms, and secure communication channels.

Uber Kraken: Kraken is a P2P-powered Docker registry that focuses on scalability and availability. It is designed for Docker image management, replication, and distribution in a hybrid cloud environment. With pluggable backend support, Kraken can easily integrate into existing Docker registry setups as the distribution layer.

Kraken has been in production at Uber since early 2018. In our busiest cluster, Kraken distributes more than 1 million blobs per day, including 100k 1G+ blobs. Kraken distributes 20K 100MB-1G blobs at its peak production load in under 30 sec.

Figure 15. Kraken visualization

Source image from: https://github.com/uber/kraken

WebRTC: WebRTC (Web Real-Time Communication) is a web standard that enables real-time communication between browsers and applications. It provides peer-to-peer capabilities for audio, video, and data streaming. WebRTC can be utilized to establish direct connections between peers in a P2P network.

ZeroMQ: ZeroMQ is a lightweight messaging library that facilitates high-performance asynchronous messaging between components or nodes in a distributed system. It supports various messaging patterns, including publish-subscribe, request-reply, and pipeline, which can be leveraged to implement P2P communication.

Kademlia: Kademlia is a distributed hash table (DHT) algorithm commonly used in P2P networks. It provides a decentralized key-value storage mechanism and efficient lookup operations. By implementing the Kademlia protocol, software engineers can build scalable and fault-tolerant P2P networks.

BitTorrent: BitTorrent is a widely used protocol for peer-to-peer file sharing. It enables efficient distribution and downloading of files across a network by dividing them into small pieces and allowing peers to exchange those pieces. Software engineers can

leverage the BitTorrent protocol to implement P2P file-sharing systems.

IPFS: IPFS (InterPlanetary File System) is a distributed file system that combines ideas from P2P networks and distributed hash tables. It provides a content-addressable storage model where files are identified by their cryptographic hashes. IPFS allows for decentralized and resilient file storage and retrieval.

Blockchain: Blockchain technology, popularized by cryptocurrencies like Bitcoin and Ethereum, can also be used to implement P2P networks. Blockchain provides a distributed and decentralized ledger for recording and validating transactions or other types of data. It can enable secure and transparent peer-to-peer interactions.

These are just a few examples of technologies that software engineers can use to implement P2P networks. The choice of technology depends on the specific requirements and goals of the P2P application being developed.

Gossip Protocol

A gossip protocol is a type of peer-to-peer (P2P) communication protocol that allows information to spread across a network in an efficient and decentralized manner. It is inspired by the way rumors or gossip spread among individuals in a social network. In a gossip protocol, each peer periodically selects a random set of peers to exchange information with, propagating the information throughout the network.

Here's how a gossip protocol typically works in a P2P network:

Initial Information Dissemination: When a new piece of information (e.g., an update, a message, or a resource) enters the network, a peer initiates the gossip process by sharing the information with a few randomly selected peers.

Peer Selection: Each peer, upon receiving new information, randomly selects a subset of peers from its list of known neighbors. The size of the subset can vary depending on the specific protocol.

Information Exchange: The selected peers exchange information with each other. This can involve sending the entire information or a subset of it, depending on the protocol design.

Propagation and Replication: The process of information exchange continues iteratively. Each peer, upon receiving new information, becomes a source for propagating that information to its selected peers. This process leads to the rapid dissemination of information across the network.

Convergence: Over time, as the gossip protocol progresses, the information is disseminated to a large portion of the network, and peers converge toward having consistent information.

Gossip protocols offer several advantages in P2P networks:

Decentralization: Gossip protocols operate without relying on a central authority or server. Peers communicate directly with each other, enabling decentralized information dissemination.

Scalability: Gossip protocols are highly scalable since the dissemination process occurs in a distributed manner. The workload is distributed among peers, allowing the network to handle large-scale systems.

Fault Tolerance: Gossip protocols are resilient to failures or changes in the network. If a peer fails or leaves the network, the information can still propagate through other paths.

Efficiency: Gossip protocols are typically efficient in terms of network bandwidth and latency. Peers exchange information with

a small subset of other peers, reducing the overall communication overhead.

Gossip protocols have been used in various applications, including distributed databases, distributed systems, content distribution networks, and peer-to-peer file-sharing networks. Examples of gossip protocols include the Epidemic protocol, Push-Sum protocol, and Plumtree protocol.

Summary

Peer-to-peer network is a powerful solution to download a large amount of data very fast for many machines. Therefore, to remember the key points, let's recap what we learned:

- A peer-to-peer (P2P) network is a decentralized network where participants, called peers, can directly connect and interact with each other without relying on a central authority.
- Peers in a P2P network can act both as consumers and providers of resources, services, or information.
- P2P networks facilitate direct communication and data exchange among peers, allowing them to share files, collaborate, or participate in decentralized systems.
- Peers in a P2P network can discover and connect to each other using various mechanisms like peer discovery protocols or distributed hash tables (DHTs).
- P2P networks can be structured in different ways, such as unstructured networks, where peers connect randomly, or structured networks, where peers connect based on a specific topology or protocol.
- P2P networks often employ protocols and algorithms designed for efficient resource sharing, such as distributed file sharing, distributed computing, or decentralized cryptocurrency transactions.

- P2P networks can offer advantages like increased resilience, scalability, and fault tolerance compared to centralized systems.
- Examples of P2P networks include file-sharing systems like BitTorrent, communication systems like Skype, and blockchain networks like Bitcoin and Ethereum.
- Building robust P2P networks requires addressing challenges like peer discovery, routing, data consistency, security, and incentivization mechanisms.

Please note that this is a simplified summary of the P2P network concept, and there are many variations and intricacies within different P2P architectures and implementations.

16 - Periodic Pooling and Streaming

We often use Periodic pooling and streaming in our day-to-day jobs, and it's also crucial to know what those concepts are for Systems Design.

In Jenkins, we often configure pooling to build our apps periodically. Kafka uses streaming by providing a distributed and fault-tolerant platform that allows applications to publish, subscribe, and process streams of records in real-time.

What is Periodic Pooling?

Pooling, also known as polling, is a technique in software development where a client periodically checks a server or a data source for updates or changes. Here are some advantages of using polling in software development:

Simplicity: Polling doesn't require complex event-driven architectures or real-time streaming infrastructure. Polling can be easily implemented using standard HTTP requests or similar communication protocols.

Compatibility: We use polling in various environments and technologies. It is compatible with most programming languages and frameworks. Additionally, it can work with various types of data sources, such as databases, APIs, or file systems.

Control over Request Frequency: With polling, you control the frequency of requests made to the server or data source. You can adjust the polling interval based on the specific requirements of

your application. This flexibility allows you to balance the trade-off between real-time updates and the load on the server.

Support for Non-Real-Time Updates: Polling suits scenarios where real-time updates are not critical. If the data or information you are retrieving doesn't need to be immediately up-to-date, polling can be an effective solution. It allows you to retrieve updates at a reasonable interval without the need for continuous streaming or event-driven architectures.

Lower Complexity on the Server-Side: Polling puts less burden on the server or data source than real-time streaming or event-driven systems. The server only needs to respond to requests when polled by the client rather than maintaining persistent connections or handling continuous data streams.

Compatibility with Caching: The client can cache the response received from the server during a polling request and use it for subsequent requests until new data is available. Caching can reduce server load and improve performance by serving responses from the cache instead of making frequent requests.

It's important to note that polling also has some limitations. It can introduce latency as the client needs to wait for the next polling interval to receive updates. It can also result in unnecessary network traffic if updates are infrequent or the client polls too frequently. In scenarios where real-time updates are critical, or network efficiency is a concern, alternative approaches like streaming or event-driven architectures may be more appropriate.

In general, polling can be a simple and effective solution for scenarios where real-time updates are not required, and periodic checks for updates are sufficient.

What is the Problem that Streaming Solve?

Streaming solves several problems in software engineering, particularly in the context of data processing and real-time applications. Here are some critical problems that streaming helps address:

Real-time data processing: Traditional batch processing approaches have limitations when processing and analyzing data in real-time. Streaming allows for continuous data processing as it arrives, enabling real-time insights and actions. Streaming is crucial in applications that require immediate processing and response, such as fraud detection, monitoring systems, or real-time analytics.

Scalability: Streaming architectures provide scalability by distributing data processing across multiple nodes or partitions. Instead of processing data centrally, streaming systems can scale horizontally by adding more processing nodes as the workload increases. Scaling allows applications to handle large volumes of data and accommodate fluctuating traffic or demand without sacrificing performance.

Low-latency processing: Streaming systems enable low-latency processing by reducing data ingestion and processing time. Unlike batch processing, which typically operates on fixed intervals or large datasets, streaming processes data incrementally as it arrives. This near real-time processing minimizes the delay between data generation and analysis, making it suitable for applications that require immediate or near-immediate responses.

Continuous data integration: Streaming facilitates the integration of diverse data sources in real-time. It allows for the seamless ingestion and processing of data from various systems, devices, or sensors, enabling a unified view of the data. Streaming is valuable in scenarios where data is generated from multiple sources

and needs to be processed and analyzed together, such as IoT applications or real-time monitoring systems.

Fault tolerance and resilience: Streaming systems are designed to handle failures and ensure data integrity. They often provide data replication, fault detection, and automatic recovery mechanisms. In the event of a failure, streaming systems can continue processing data without significant interruptions, minimizing the impact on applications and maintaining data consistency.

Event-driven architecture: Streaming enables event-driven architectures, allowing applications to respond to real-time events. Instead of relying on polling or manual triggers, streaming systems deliver events to subscribing applications in real time. This approach improves responsiveness, agility, and scalability, as applications can process events as they occur. Streaming technologies like Apache Kafka, Apache Flink, and Amazon Kinesis are commonly used for event-driven architectures, enabling seamless communication and integration between components in distributed systems.

Technologies that Use Periodic Pooling

Various technologies and frameworks use periodic pooling to check for updates or perform tasks at regular intervals. Some examples include:

HTTP Polling: Web applications often use periodic HTTP polling to check for updates or fetch new data from a server at regular intervals. Clients request the server at predetermined intervals to retrieve the latest information.

Database Polling: In some scenarios, applications may use periodic polling to check for new or updated records in a database. This check can be helpful in tasks such as data synchronization or detecting changes in a database table.

Messaging Systems: Messaging systems like RabbitMQ or Apache ActiveMQ can periodically poll to check for new messages in a queue or topic. Consumers periodically poll the messaging system to retrieve and process any new messages.

Job Scheduling: Job scheduling frameworks like Quartz or cron-based systems use periodic pooling to execute scheduled tasks at predefined intervals. The scheduler periodically checks for pending jobs and triggers their execution based on the configured schedule.

Monitoring and Health Checks: Systems monitoring tools or health check frameworks often employ periodic pooling to check the status or health of various components or services. We perform these checks regularly to ensure the system is functioning correctly.

These are just a few examples of technologies that utilize periodic pooling to check for updates, perform tasks, or monitor resources at regular intervals. The specific use cases and implementation details can vary depending on the requirements of the application or system being developed.

How Does Streaming Work?

Streaming can be facilitated through socket connections. Here's a simplified explanation of how streaming works using socket-based communication:

Server-Side Setup: The streaming process begins with setting up a server that listens for incoming connections. The server establishes a socket, a communication endpoint, and binds it to a specific IP address and port.

Client-Side Connection: On the client side, a connection is established to the server by creating a socket and providing the server's IP address and port number. The client socket connects to the server socket, forming a connection between the two.

Data Transmission: Once the connection is established, data can be transmitted from the server to the client in a streaming fashion. The server can continuously send data in smaller chunks or packets to the client over the open socket connection.

Data Reception: On the client side, the socket continuously receives the incoming data packets as the server transmits them. The client can process or consume the received data immediately without waiting for the entire data set.

Real-time Processing: As the client receives the data packets, it can perform real-time processing on the received data. The processing can include tasks such as data analysis, transformations, storage, or any other application-specific operations.

Continuous Communication: The data transmission and reception between the server and the client continue in a constant and ongoing manner. The server keeps sending data packets, and the client keeps receiving and processing them as they arrive.

Socket-based communication provides a reliable and efficient way to establish a streaming connection between a server and a client. It allows for bidirectional communication, enabling both data transmission from the server to the client and potential feedback or responses from the client to the server.

It's important to note that while socket-based communication is a common method for implementing streaming, there are also other technologies and protocols specifically designed for streaming, such as HTTP-based streaming protocols (e.g., HLS or DASH) or message queue systems (e.g., Apache Kafka or RabbitMQ). The choice of technology depends on the specific requirements and characteristics of the streaming application.

What Is the Use Case of Streaming?

Streaming is well-suited for various use cases in software engineering where real-time data processing, continuous data integration, or immediate response is required. Here are some everyday use cases where streaming is beneficial:

Real-time Analytics: Streaming is valuable for real-time analytics applications where data is continuously generated, and immediate insights are needed. Examples include monitoring systems, fraud detection, stock market analysis, social media sentiment analysis, and network traffic analysis. Streaming enables the processing and analysis of incoming data as it arrives, allowing organizations to make timely decisions based on up-to-date information.

Internet of Things (IoT): Streaming is essential in IoT applications, where many devices generate a continuous stream of sensor data. Streaming enables real-time processing and analysis of IoT data, enabling applications like smart home automation, industrial monitoring and control, predictive maintenance, or environmental monitoring. By processing IoT data in real-time, organizations can respond to events, trigger actions, or detect anomalies promptly.

Log Processing and Monitoring: Streaming is valuable for processing and analyzing log data in real-time. Applications can continuously stream logs from various sources, such as servers, applications, or network devices, and perform real-time analysis to detect errors, performance issues, or security breaches. Streaming log data allows organizations to proactively monitor systems, identify problems, and take immediate corrective actions.

Real-time Collaboration and Messaging: Streaming is useful in building real-time collaboration and messaging applications. Examples include chat applications, collaborative document editing, multiplayer gaming, or video conferencing. Streaming enables instant message delivery, real-time updates, and synchronized communication between multiple users or devices.

Event-driven Architectures: Streaming is foundational for event-driven architectures, where applications respond to events or triggers in real-time. Streaming allows applications to process and react to events as they occur, enabling event-driven workflows, notifications, or automated actions based on specific events. Event-driven architectures are relevant in domains like workflow automation, event processing systems, or event-driven microservices.

Continuous Data Integration: Streaming is beneficial for integrating and processing data from multiple sources in real-time. For example, streaming can capture data from various systems, databases, or APIs in a data pipeline and perform real-time transformations, aggregations, or enrichments. Streaming gives organizations a unified and up-to-date view of their data, crucial for applications like real-time dashboards, data synchronization, or data-driven decision-making.

These are just a few examples of how streaming can be applied in software engineering. The suitability of streaming depends on the application's specific requirements, the need for real-time processing, and the nature of the data being handled.

Technologies that Use Streaming

Several technologies and frameworks leverage streaming as a fundamental concept for real-time processing and delivering data. Here are some notable examples:

Apache Kafka: Kafka is a distributed streaming platform that provides a high-throughput, fault-tolerant, and scalable solution for handling real-time data streams. It is widely used for building event-driven architectures, real-time analytics, and data pipeline applications.

Apache Flink: Flink is an open-source stream processing framework that enables distributed, fault-tolerant, and low-latency data

processing of streaming data. It supports event time processing, windowing, state management, and complex event processing.

Apache Spark Streaming: Spark Streaming is a component of the Apache Spark ecosystem that allows processing and analyzing real-time data streams. It provides high-level abstractions for stream processing and integrates with Spark's batch processing, machine learning, and graph processing capabilities.

Amazon Kinesis: Amazon Kinesis is a managed service by Amazon Web Services (AWS) for handling real-time streaming data at scale. It offers capabilities to collect, process, and analyze streaming data from various sources, such as logs, IoT devices, and clickstreams.

Apache Storm: Storm is a distributed real-time computation system allowing stream processing and real-time analytics. It provides fault-tolerant processing of streams with scalable and reliable data processing capabilities.

Confluent Platform: Confluent Platform is built around Apache Kafka and provides an enterprise-ready streaming platform. It offers additional features and tooling for managing and monitoring Kafka clusters and integrating them with other data systems.

Microsoft Azure Stream Analytics: Azure Stream Analytics is a real-time service on the Microsoft Azure cloud platform. It enables processing and analyzing streaming data from various sources, including IoT devices, social media, and logs.

These technologies empower developers to efficiently handle and process real-time data streams, enabling use cases such as real-time analytics, event-driven architectures, IoT data processing, and more.

Summary

Either the concept of pooling or streaming is crucial when designing systems. That's because tools like Jenkins use pooling, and Kafka uses streaming, for example. Once we master those concepts, it's much easier to understand the tools that implement them. To remember the key points, let's recap:

Pooling:

- Periodically checking a server or data source for updates or changes.
- Simple approach.
- Compatibility with various technologies.
- Control over request frequency.
- Suitable for non-real-time updates.
- Places less burden on the server side.
- It can work well with caching mechanisms.

Streaming:

- Continuous flow of data in real-time.
- Data is transmitted and processed incrementally.
- Enables real-time processing and analysis.
- Suitable for real-time analytics, IoT, log processing, and collaboration applications.
- Supports event-driven architectures.
- Provides immediate insights and actions.
- Requires specialized infrastructure and protocols.
- Handles large volumes of data.
- Enables synchronized communication and real-time collaboration.

17 - Static and Dynamic Configuration

Static and dynamic configurations are two approaches used to set up and modify the settings of cloud services.

Static configuration involves defining the configuration parameters before deploying the service. It includes specifying values in configuration files or through the service's management console. The static configuration provides predictability and ease of deployment since we establish the settings upfront, and we can replicate them across environments. However, it may not be suitable for scenarios requiring frequent changes.

Dynamic configuration allows for on-the-fly modification of settings without redeployment or service interruption. It enables real-time adjustments to parameters while the service is running, offering flexibility and adaptability to changing conditions.

Dynamic configuration is achieved through mechanisms like configuration APIs or environment variables. It allows for quick response to requirements, workload variations, and fine-tuning of the service's behavior. We often use a combination of static and dynamic configuration approaches to balance stability and responsiveness in cloud services.

Pros and cons of Static Configuration

Static configuration refers to the process of setting up and defining the configuration parameters of a system or application before it is deployed or executed. In static configuration, the configuration

values are typically specified in configuration files or directly in the application's source code.

Static configuration often establishes a system or application's initial settings and behavior. These configuration parameters can include options such as database connection details, network settings, logging levels, feature toggles, and various other parameters that define the behavior and functionality of the system.

The advantages of static configuration include:

- **Predictability**: Since the configuration values are set before deployment, the system's behavior is consistent and predictable across different environments.
- **Ease of Deployment**: Static configuration allows for easy system deployment since the configuration values are already defined and can be easily replicated across different instances or environments.
- **Security**: Configuration files can be secured and restricted to authorized users, preventing unauthorized access or modification of critical system settings.
- **Version Control**: Configuration files can be maintained in version control systems, allowing for easy tracking of changes and rollbacks to previous configurations if needed.

However, the static configuration has some limitations:

- **Limited Flexibility**: Static configuration may not be suitable for scenarios where dynamic or runtime changes to configuration values are required. For example, if a system needs to adapt to changing network conditions or load balancing requirements, static configuration may not be the best approach.
- **Configuration Drift**: In complex systems with multiple instances or environments, keeping the configuration consistent across all deployments can be challenging and may lead to configuration drift if not properly managed.

To overcome the limitations of static configuration, dynamic configuration approaches can be used, where configuration values can be modified and updated at runtime without redeployment. It allows for greater flexibility and adaptability in changing environments.

Example of static Configuration in Java Let's see a simple example of a static configuration using Spring that reads a YML configuration file and prints the values. Remember that we are only able to make the changes reflected when the deploy the application:

1 – Create the application.yml file

```
myapp:
  staticValue: Hello, World!
```

2 - To configure Spring to read the YML file, we need to create a configuration Java class

```
import org.springframework.context.annotation.Configurati\
on;

@Configuration
public class MyAppConfig {
}
```

3 – Using the static configuration with the annotation @Value from Spring

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class MyBean {
    @Value("${myapp.staticValue}")
    private String staticValue;

    public void printStaticValue() {
        System.out.println(staticValue);
    }
}
```

4 – Now, we can print the value of the static configuration

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.          \
                                          AnnotationConfigA
pplicationContext;

public class MyApp {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfig\
ApplicationContext.
     (MyAppConfig.class);
        MyBean myBean = context.getBean(MyBean.class);
        myBean.printStaticValue(); // Output: Hello, Worl\
d!
    }
}
```

Pros and Cons of Dynamic Configuration

Dynamic configuration in software systems offers certain advantages and disadvantages. Let's explore the pros and cons.

Pros of Dynamic Configuration

Flexibility and Adaptability: Dynamic configuration allows for on-the-fly modifications to system settings without requiring a restart or redeployment. This flexibility enables the system to adapt to changing requirements or operational conditions more efficiently.

Real-time Configuration Updates: Dynamic configuration enables real-time updates to system settings, which can be advantageous in scenarios where immediate adjustments are necessary. It allows for fine-grained control over various aspects of the system without interrupting its operation.

Environment-specific Configuration: Dynamic configuration supports the ability to customize settings based on different environments (e.g., development, staging, production). This flexibility simplifies the deployment process and ensures that the system behaves optimally in each environment.

Centralized Configuration Management: Dynamic configuration often involves a centralized management system or tool. This centralized approach allows for easier management, versioning, and auditing of configuration settings, promoting better governance and control.

Cons of Dynamic Configuration

Increased Complexity: Dynamic configuration introduces additional complexity to the system. Managing real-time updates,

synchronization, and consistency across distributed systems can be challenging and may require careful design and implementation.

Potential for Configuration Errors: With dynamic configuration, there is a higher risk of introducing configuration errors or inconsistencies. Frequent updates and modifications increase the chances of misconfigurations, leading to system instability or unexpected behavior.

Runtime Dependencies: Dynamic configuration may introduce dependencies on external systems or services responsible for managing and distributing configuration settings. If these dependencies fail or experience downtime, it can impact the availability and stability of the system.

Security Concerns: Dynamic configuration systems can be potential targets for security breaches. Unauthorized access to configuration settings or tampering with them can have severe consequences. Therefore, robust security measures must be implemented to safeguard sensitive configuration data.

It's important to carefully evaluate the specific requirements and constraints of a system before deciding to adopt a dynamic configuration. While it offers benefits like flexibility and real-time updates, it also introduces complexities and potential risks that must be managed effectively.

Example of Dynamic Configuration with Java

An example that is widely used in the market for dynamic configuration is the Spring config server. Spring config server is a lib that allows Microservices to retrieve configurations dynamically from the Spring config server. It means that we have all the configurations in a separate service, enabling us to change those configurations dynamically without impacting other Microservices.

Let's see a code example to configure the config server.

Set up the Spring Config Server

1 – Create a new Spring Boot project and include the necessary dependencies in your pom.xml file

```
<!-- For Spring Config Server -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

2 – Annotate your main application class with @EnableConfigServer to enable the Config Server functionality

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootA\
pplication;
import org.springframework.cloud.config.server.EnableConf\
igServer;

@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(ConfigServerApplication.cla\
ss, args);
    }
}
```

3 – Configure the server's properties in application.properties

```
server.port=8888
spring.cloud.config.server.git.uri=https://github.com/you\
r-username/your-config-repo.git
Spring.cloud.config.server.git.search-paths=configs/{appl\
ication}
```

4 – Create a configuration file in the Git repository

Assuming you have a Git repository with the URL https://github.com/your-username/your-config-repo.git, create a configuration file named myapp.properties in the repository.

```
greeting.message=Hello from Config Server!
```

5 – Create a client application

Create a new Spring Boot project for the client application and include the necessary dependencies in your pom.xml file:

```
<!-- For Spring Config Client -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

6 – Configure the client application's properties in bootstrap.properties

```
spring.application.name=myapp
Spring.cloud.config.uri=http://localhost:8888
```

7 – Create a simple service that uses the configuration property

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;

@Service
public class GreetingService {

    @Value("${greeting.message}")
    private String greetingMessage;

    public String getGreetingMessage() {
        return greetingMessage;
    }
}
```

8 – Start the Config Server and the client application Start the Config Server application first. Then, start the client application. The client application will connect to the Config Server and retrieve the configuration dynamically.

Access the configuration property in the client application:

9 – Use the GreetingService bean to access the configuration property in a controller or any other Spring bean

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestContro\
ller;

@RestController
public class GreetingController {

    private final GreetingService greetingService;

    public GreetingController(GreetingService greetingSer\
vice) {
        this.greetingService = greetingService;
```

```
13      }
14
15      @GetMapping("/greeting")
16      public String getGreeting() {
17          return greetingService.getGreetingMessage();
18      }
19  }
```

When you access the /greeting endpoint of the client application (e.g., http://localhost:8080/greeting), it will fetch the greeting.message property from the Config Server dynamically and return the corresponding value ("Hello from Config Server!").

That's a basic example showcasing the usage of Spring Config Server with a client application. The Config Server retrieves configuration from a Git repository, and the client application fetches the configuration dynamically from the Config Server without requiring redeployment.

Summary

One key point to remember when designing systems is that there is no one-size-fits-all solution. Static configuration is not better than dynamic configuration or vice-versa. It will depend on the use case. If the configuration changes very often or there is a specific requirement where configurations must be changed without the need for a redeploy, then dynamic configuration is better. But if the configuration doesn't change very often, static configuration works fine, and it's less complex to maintain.

To remember the key points, let's recap:

Static Configuration:

- Configuration settings are defined and set during deployment or startup.

- Settings remain unchanged until the next deployment or restart.
- Provides stability and consistency to the application.
- Suitable for scenarios where configuration changes are infrequent.
- May require manual intervention and downtime to update settings.

Dynamic Configuration:

- Configuration settings can be modified during runtime without redeployment or restart.
- Enables real-time adjustments to application behavior.
- Offers flexibility and adaptability to changing requirements or dynamic environments.
- Useful for scenarios that require frequent updates or immediate changes.
- Can be achieved through mechanisms like configuration servers or live reloading.

18 - Rate Limiting

Rate limiting is a technique used in computer systems to control the speed at which requests or actions can be made within a specific timeframe. It limits the number of requests occurring within a given period, preventing abuse or overload. It's like a traffic signal on the internet, ensuring a smooth and controlled data flow.

Rate limiting prevents excessive or malicious usage by enforcing a maximum limit on the number of actions that can happen within a specific time window. For example, it can restrict the number of login attempts a user can make within a minute or limit the number of API requests a program can send per hour. By implementing rate limiting, system administrators can protect their resources, maintain system stability, and promote fair usage of their services.

Throughout my career, the situation of malicious users attacking our services has already happened, performing a massive amount of requests and making the service very slow. To solve this problem, we use rate limiting. So, rate limiting is a fundamental concept to remember when developing Microservices and exposing APIs.

Rate Limiting Use Cases

Rate limiting is beneficial in various situations where controlling traffic flow or actions within a system is necessary. Here are some common scenarios where rate limiting is advantageous:

API Management: When exposing an API to external users or third-party developers, rate limiting helps ensure fair usage and prevent abuse. It allows you to restrict the number of requests that can be made in a given period, preventing one user or application

from overwhelming the API and affecting its performance or availability. It's even possible to prevent users from certain regions to not access the API with a more strict rate limiting rule.

Security and Protection: Rate limiting can effectively protect against brute-force attacks or other malicious activities. For example, the Denial-of-Service (DoS) attack which is when an attacker overwhelms a target system with excessive traffic or requests, making it unavailable to legitimate users. Using rate limiting, we limit the number of login attempts within a specific time frame, making it harder for attackers to guess passwords or gain unauthorized access.

System Stability: Rate limiting helps maintain system stability and prevents overload. Controlling the rate of incoming requests or actions ensures that the system resources are not overwhelmed and can handle the load effectively. This is particularly relevant for high-traffic websites or applications where a sudden influx of requests can cause performance degradation or downtime.

Resource Conservation: Rate limiting can be used to conserve resources, especially in cases where limited resources are allocated to each user or application. By setting a limit on the number of actions or requests, you can prevent excessive use of resources and ensure fair distribution among users.

Monetization and Subscription Models: If you offer services through a subscription model or have different tiers of access, rate limiting can be used to enforce usage limits based on the subscription level. This helps control access to features or restrict usage based on the subscription plan, ensuring that users adhere to their allotted usage quotas.

In summary, rate limiting is useful in scenarios involving API management, security protection, system stability, resource conservation, and enforcing usage limits for monetization or subscription models. It helps maintain fairness, protect against abuse, and ensure the efficient use of system resources.

Rate Limiting Example with Spring

Now, let's see an example of rate-limiting with the most popular Java frameworks, Spring!

1 – Let's declare the Spring dependencies into the source code.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

2 – We create now a filter class that intercepts every request, then we create a Queue with the ipAddress and will block the IP if more than 100 requests are made within 5 minutes. In case the rate limiting condition is met, we will send the HttpStatus "too many requests" to the user which is the code 429:

```
public class RateLimitFilter extends OncePerRequestFilter\
 {

    private static final int MAX_REQUESTS = 100;
    private static final Duration TIME_WINDOW = Duration.\
ofMinutes(5);
    private static final Map<String, Queue<Long>> request\
Map = new ConcurrentHashMap<>();

    @Override
    protected void doFilterInternal(HttpServletRequest re\
quest, HttpServletResponse response, FilterChain filterCh
ain) throws ServletException, IOException {
```

```
14          String ipAddress = request.getRemoteAddr();
15          Queue<Long> requestQueue = requestMap.computeIfAb\
16  sent(ipAddress, k -> new ConcurrentLinkedQueue<>());
17
18          long now = System.currentTimeMillis();
19          requestQueue.add(now);
20
21          while (!requestQueue.isEmpty() && now - requestQu\
22  eue.peek() > TIME_WINDOW.toMillis()) {
23              requestQueue.poll();
24          }
25
26          if (requestQueue.size() <= MAX_REQUESTS) {
27              filterChain.doFilter(request, response);
28          } else {
29              response.setStatus(HttpStatus.TOO_MANY_REQUES\
30  TS.value());
31              response.getWriter().write("Rate limit exceed\
32  ed");
33          }
34      }
35  }
```

3 – In the basic web configuration from Spring, we include our RateLimitFilter class as a filter. The method addFilterAfter will intercept every HTTP request, will execute the RateLimitFilter logic and will invoke the API method only if the rate limiting is not exceeded.

```
@Configuration
@EnableWebSecurity
public class RateLimitingConfig extends WebSecurityConfig\
urerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Ex\
ception {
        http
            .authorizeRequests()
            .antMatchers("/api/**")
            .permitAll()
            .and()
            .addFilterAfter(rateLimitFilter(), UsernamePa\
sswordAuthenticationFilter.class);
    }

    @Bean
    public RateLimitFilter rateLimitFilter() {
        return new RateLimitFilter();
    }
}
```

4 – Finally, by accessing the following API more than 100 times within the time frame of 5 minutes, the "too many requests" HTTP status will be returned and will block requests of the user IP:

```
@RestController
@RequestMapping("/api")
public class ApiController {

    @GetMapping("/resource")
    @PreAuthorize("hasRole('USER')")
    public String getResource() {
        // Your API logic here
    }
}
```

Summary

Rate limiting is a crucial concept to keep in mind when developing Microservices. Ideally, we shouldn't wait an attack to use rate limiting, we should make our Microservices API prepared to avoid attacks such DoS (Denial-of-Service). Therefore, let's see the key points of rate limiting:

- Rate limiting is a technique used to control and limit the number of requests or actions that can be performed within a specific timeframe.
- It helps prevent abuse, misuse, and overloading of systems or APIs by limiting the rate at which requests can be made.
- Rate limiting can protect against various issues, including DoS attacks, brute force attacks, and excessive resource consumption.
- It sets thresholds on the number of requests allowed per user, IP address, or other identifiers, within a defined time window.
- When the rate limit is exceeded, further requests may be rejected or delayed, or an error response may be returned.
- Rate limiting can be implemented at different levels, such as network level, server level, or application level.

- It is commonly used in web applications, APIs, and microservices to ensure fair usage, maintain system stability, and protect against abuse or unauthorized access.
- Rate limiting strategies can include fixed quotas, sliding windows, token buckets, or leaky buckets, depending on the specific requirements and use cases.
- Proper configuration and tuning of rate limiting parameters are crucial to balance security, user experience, and system performance.
- Rate limiting can be combined with other security measures, such as authentication, authorization, and anomaly detection, for comprehensive protection against various threats.

19 - Logging and Monitoring

Logging and monitoring were important with monoliths, but they're far more critical with Microservices. That's because debugging with Microservices is difficult since services are deployed independently.

With a robust tool for logging and monitoring, we can quickly know what happened by checking what occurred with a specific user. Also, we can measure metrics from services such as latency, errors, performance, etc.

Logging

Logging is the process of recording events or messages that happen when a software application or system is running. It's crucial because it helps developers and administrators understand how the application behaves, find and fix issues or errors, and learn about system performance and usage.

For example, if a user purchased a product but didn't receive the product, how could we figure this out without logs? By logging in to the system, we can quickly check the time the customer purchased the product to see if the purchase was executed, and then we can deliver the product.

Another scenario is that a bug might appear in one cloud service for unknown reasons. The quickest action we can take to troubleshoot this issue is to check the logs. Sometimes, we can discover the problem immediately by looking at the logs.

By logging events, they can review the recorded information to investigate problems, track the sequence of events that caused an issue, and find ways to make the application better.

Let's see the key components of logging:

Log Messages: Log messages are the information that is recorded during the logging process. They typically include timestamps, severity levels (e.g., debug, info, warning, error, etc.), source or module names, and the actual content or description of the event or message.

Logging Framework/Library: A logging framework or library is a software component that provides the necessary tools, APIs, and utilities to facilitate logging in an application. Examples of popular logging frameworks include Log4j, Logback, and Python's logging module. These frameworks offer features like log-level configuration, formatting, rotation, and output destinations (e.g., console, file, database, etc.).

Log Outputs: Log outputs are the destinations where log messages are written or stored. Common log outputs include console output (displaying log messages in the console or terminal), log files (writing logs to files on disk), and remote log servers or databases (sending logs to a centralized logging system). The choice of log outputs depends on the specific requirements of the application or system.

Log Levels: Log levels are used to categorize log messages based on their severity or importance. Common log levels include:

DEBUG: Detailed information for debugging purposes. **INFO**: General information about the application's execution. **WARN**: Indication of potential issues or warnings that may require attention. **ERROR**: Signifies errors or exceptional conditions that may impact the application's functionality. **FATAL/CRITICAL**: Critical errors that may lead to the termination of the application.

Developers can set the desired log levels to control which messages

are recorded based on their importance. For example, in a production environment, the log level may be set to ERROR or higher to focus on critical issues. In contrast, in a development or testing environment, a lower log level like DEBUG or INFO may be used to capture more detailed information.

Logging Best Practices

Using meaningful log messages: Log messages should provide enough information to understand the context of the event or error and aid in troubleshooting.

Proper log levels: Assigning appropriate log levels to messages ensures we don't miss important events and that the logs are not cluttered with excessive information.

Contextual information: Including relevant contextual information in log messages, such as user IDs, request IDs, or error codes, can assist in tracing the flow of execution and identifying specific instances of an issue.

Log rotation and archival: Managing log files by implementing log rotation strategies helps prevent log files from growing indefinitely and consuming excessive disk space. Archiving old logs is helpful for compliance, auditing, or historical analysis.

Security and privacy considerations: Care should be taken to avoid logging sensitive information, such as passwords or personally identifiable data, to prevent potential security breaches or privacy violations.

Logging is a crucial aspect of software development and system administration that enables monitoring, troubleshooting, and analysis of applications and systems. By capturing and reviewing log messages, developers and administrators can gain insights into the behavior of their software and identify and resolve issues efficiently.

Monitoring

Monitoring means keeping a close watch on a system or application to ensure it works well. It involves checking and analyzing data in real-time or at set times to find unusual things happening and understand how the system is doing.

To accomplish that, we usually use software with specific dashboards and data to quickly understand what is happening in the system.

Monitoring ensures software, servers, networks, and other system parts work correctly. It helps find and fix problems before they become significant issues, reduces downtime, and best uses resources.

Let's see some key aspects of monitoring:

Metrics and Data Collection: Monitoring involves collecting relevant metrics and data points from different sources, such as system logs, performance counters, network traffic, and application-specific indicators. These metrics include CPU usage, memory utilization, network latency, response times, error rates, etc.

Monitoring Tools and Platforms: Various monitoring tools and platforms are available to facilitate data collection, visualization, and analysis. These tools range from simple command-line utilities to sophisticated monitoring systems with dashboards, alerts, and advanced analytics capabilities. Examples include Prometheus, Grafana, Nagios, and Datadog.

Real-time Monitoring: Real-time monitoring focuses on capturing and analyzing data as it happens, providing immediate insights into the system's current state. It enables quick identification and response to critical issues, such as service outages or performance bottlenecks.

Historical Monitoring: Historical monitoring involves analyzing collected data to identify trends, patterns, and long-term perfor-

mance characteristics. It helps in capacity planning, trend analysis, and predicting future resource requirements.

Alerting and Notifications: Monitoring systems often include alerting mechanisms that notify designated individuals or teams when predefined thresholds or conditions are met or exceeded. We can send alerts via email or SMS or integrate them with collaboration tools like Slack, allowing quick response and resolution of issues.

Visualization and Dashboards: Monitoring tools visually represent collected data through dashboards and charts. These visualizations make understanding and interpreting complex data sets easier, enabling stakeholders to assess the system's health and performance quickly.

Performance Optimization: Monitoring data can identify performance bottlenecks, resource inefficiencies, or areas for optimization. By analyzing the collected metrics, administrators and developers can make informed decisions and implement improvements to enhance the system's overall performance and efficiency.

In general, monitoring is essential in ensuring the proper functioning, performance, and availability of software applications and systems. It involves collecting and analyzing data in real-time or over a period to detect issues, identify trends, and optimize performance. Organizations can proactively address problems, maintain stability, and make informed decisions to improve their IT infrastructure by monitoring systems.

Metrics

Metrics refer to the specific measurements or data points that are collected and analyzed to assess the performance, health, and behavior of a system or application. These metrics provide quantitative information about various aspects of the system and are

used to track its performance, detect anomalies, and make informed decisions.

Metrics can include a wide range of data, depending on the specific system or application being monitored. Some common examples of metrics in monitoring include:

Performance Metrics: These metrics measure a system or application's speed, efficiency, and responsiveness. Examples include response time, throughput, CPU usage, memory utilization, and network latency.

Availability Metrics: These metrics indicate the uptime and availability of a system or service. For example, we can have metrics such as uptime percentage, downtime duration, or the number of service outages.

Error and Exception Metrics: These metrics track the occurrence and frequency of system errors, exceptions, or failures. Examples include error rates, exception counts, or specific error codes.

Resource Utilization Metrics: These metrics measure the usage and allocation of system resources such as CPU, memory, disk space, and network bandwidth. They provide insights into resource consumption patterns and can help optimize resource allocation.

User Experience Metrics: These metrics measure the quality and satisfaction of user interactions with a system or application. Examples include page load time, click-through rates, conversion rates, and user feedback ratings.

Security and Compliance Metrics: These metrics assess a system or application's security posture and compliance adherence. They can include metrics related to security incidents, vulnerability scans, access controls, or regulatory compliance checks.

Monitoring tools and platforms collect and analyze these metrics over time to generate reports, visualizations, and alerts. They help stakeholders understand the system's performance, identify issues,

and make data-driven decisions to improve its reliability, efficiency, and user experience.

Tracing

Tracing refers to capturing and analyzing the flow of execution and interactions within a system or application. It involves tracking the path of requests or transactions as they traverse various components or services.

It allows for detailed visibility into how they are processed and where potential issues or bottlenecks may arise.

For example, if a Microservice creates a request for another Microservice, we will have the tracing from the request trace. So, if there is a request for 5 Microservices, we will get the tracing of all of those Microservices.

Tracing provides a comprehensive view of the end-to-end journey of a request or transaction, including the different services or modules it traverses, the time taken at each step, and any errors or delays encountered along the way. This visibility helps identify performance bottlenecks, diagnose issues, and optimize system behavior.

Here are key aspects of tracing in monitoring:

Distributed Tracing: Distributed tracing focuses on capturing and correlating information across multiple components or services involved in processing a request. It allows for tracking the path of a request as it moves through different systems, enabling the identification of performance issues or dependencies between services.

Tracing Instrumentation: Tracing requires adding instrumentation to the code of an application or system to capture relevant information at different stages of execution. Tracing can involve

adding trace statements or using specialized tracing libraries or frameworks that automatically capture essential data points, such as timestamps, method invocations, or network calls.

Trace Data and Context: Tracing generates data that includes unique identifiers for requests or transactions, as well as contextual information about each step, such as timestamps, service names, and metadata. This data is collected and organized to form a trace, representing the complete journey of a request or transaction.

Trace Visualization and Analysis: We typically visualize traces in tools or platforms that provide insights into the behavior and performance of a system. Visualization tools can display the flow of requests, highlighting potential bottlenecks or errors. Analysis features allow drilling down into individual traces to examine specific steps, timings, and dependencies.

Performance Optimization: Tracing data can be used to identify performance issues and bottlenecks within a system. By analyzing trace information, developers and administrators can pinpoint areas contributing to slow response times, high latency, or resource inefficiencies. It also enables them to make targeted improvements and optimizations to enhance overall system performance.

Tracing complements monitoring by providing detailed information about the execution and behavior of specific transactions or requests. It helps in understanding the end-to-end flow of a system, diagnosing issues, and optimizing performance and reliability.

Technologies to Use Logging and Monitoring

Several technologies and tools are available for logging and monitoring in software development and system administration. Here are some commonly used ones:

Logging Technologies

Log4j: A widely used Java-based logging framework that allows developers to log events and messages at different levels of severity. **Serilog**: A versatile logging library for .NET that supports structured logging and allows logs to be stored in various formats and destinations. **Winston**: A popular logging library for Node.js that provides flexible logging options and supports various transports, such as console, file, or external services.

Monitoring Technologies

Prometheus: An open-source monitoring system that collects metrics from targets using a pull model and provides powerful querying, alerting, and visualization capabilities.

Grafana: A widely used open-source platform for visualizing and analyzing time-series data from various data sources, including Prometheus, Elasticsearch, and InfluxDB.

Nagios: A robust open-source monitoring tool that enables monitoring of hosts, services, and network devices, and provides alerting and reporting features.

Datadog: A cloud-based monitoring and analytics platform that offers comprehensive monitoring, alerting, and visualization capabilities for infrastructure, applications, and logs.

Cloud Monitoring Services

Amazon CloudWatch: A monitoring and observability service provided by Amazon Web Services (AWS) that collects and analyzes metrics, logs, and events from various AWS resources and applications. **Google Cloud Monitoring**: A monitoring and observability service offered by Google Cloud Platform (GCP) that

provides monitoring, alerting, and visualization capabilities for GCP resources and applications. **Azure Monitor**: A comprehensive monitoring solution provided by Microsoft Azure that collects and analyzes telemetry data from Azure resources, applications, and infrastructure.

Logging and Monitoring Aggregation

ELK Stack (Elasticsearch, Logstash, Kibana): A popular open-source stack for centralized logging and monitoring. It uses Elasticsearch for log storage and searching, Logstash for log processing and ingestion, and Kibana for log visualization and analysis.

Splunk: A powerful log management and analysis platform that allows organizations to collect, index, and analyze data from various sources, including logs, metrics, and events. These are just a few examples of the many logging and monitoring technologies available. The choice of technologies depends on factors such as the programming language, the infrastructure being used, specific requirements, and the scale of the system.

Summary

Logging and monitoring are necessary for every cloud service to give us more control over our applications. As software engineers, we need to know what logging and monitoring technologies we should use and how to use them.

Logging:

- Log4j, Serilog, Winston are popular logging technologies.
- Logging records events and messages during application execution.

- Helps understand application behavior, diagnose issues, and improve performance.
- Captures information for monitoring, debugging, analysis, and auditing.

Monitoring:

- Prometheus, Grafana, Nagios, Datadog are common monitoring technologies.
- Continuous observation and measurement of system/application aspects.
- Detects anomalies, identifies trends, and provides insights into system health.
- Ensures proper functioning, performance, and availability of software and infrastructure.
- Proactively identifies and addresses issues, minimizes downtime, and optimizes resource utilization.

Cloud Monitoring Services:

- Amazon CloudWatch, Google Cloud Monitoring, Azure Monitor.
- Collects and analyzes metrics, logs, and events from cloud resources and applications.

Logging and Monitoring Aggregation:

- ELK Stack (Elasticsearch, Logstash, Kibana) for centralized logging and monitoring.
- Splunk for log management and analysis.
- Choice of technologies depends on programming language, infrastructure, and specific requirements.

20 - Message Broker

Pub-Sub (Publish-Subscribe) is a messaging pattern used in event-driven systems with message brokers. It simplifies communication by separating publishers, who generate events, from subscribers, who consume them.

To make this simple, let's make an analogy with an email subscription service. When we register to an email list, we will receive their messages. In this case, the email list will be the publisher, we will be the subscribers, and the means (downloaded a book, mini-course, mini-quiz) you chose to get subscribed in the list is the topic. Depending on the topic you register for, you will receive different emails.

This decoupling allows for scalable and adaptable systems, as publishers and subscribers are unaware of each other. Publishers emit events without knowing who will receive them, while subscribers express interest in specific events. The Pub-Sub system acts as a mediator, efficiently distributing events to the relevant subscribers.

In this article, we delve into the fundamentals of Pub-Sub, exploring its benefits and real-world applications. We discuss how Pub-Sub facilitates loose coupling, enabling components to be added or removed dynamically without impacting the overall system. We also examine features such as event categorization and selective consumption, which enhance efficiency and reduce unnecessary processing. Whether you're new to event-driven systems or seeking to optimize your existing architecture, understanding Pub-Sub will empower you to build scalable and resilient applications. Join us as we uncover the simplicity and power of Pub-Sub in streamlining event-based communication.

Publisher

A message broker publisher is like a sender or a source of messages. It is responsible for creating and sending messages to the message broker. Think of it as someone who writes a letter and puts it in an envelope to be delivered through a postal service. The publisher decides what information or events to send and when. Once the publisher sends a message to the message broker, it becomes available for consumption by interested subscribers.

The other example is a newsletter. If you are subscribed to the newsletter from Java Challengers, you will receive emails according to your needs. In this case, Java Challengers is the publisher, and you are the subscriber.

Subscriber

A message broker subscriber is like a receiver or a listener for messages. It is interested in specific types of messages and wants to be notified whenever they are available. Think of it as someone eagerly waiting for a letter to arrive in their mailbox. The subscriber informs the message broker about the types of messages it wants to receive, and the message broker delivers those messages to the subscriber when they become available. The subscriber can then process or react to the received messages according to its specific needs or requirements.

As mentioned before, if you are subscribed to a newsletter service, you will be the subscriber and will be receiving messages from the publisher whenever available.

Topic

A message broker topic is like a channel or category that messages can be organized into. It helps to group related messages together based on a common theme or subject. Think of it as different sections or topics in a newspaper where articles of similar content are grouped together. When a publisher sends a message to the message broker, it assigns the message to a specific topic. Subscribers can then choose to subscribe to one or more topics they are interested in. This way, when a message is published to a topic, all subscribers interested in that topic will receive the message.

Topics allow for selective message consumption, ensuring that subscribers only receive the messages they are interested in and ignore the rest.

Multiple Topics

Multiple topics refer to the ability to organize messages into different categories or channels based on their subject or content. Instead of having a single topic, a message broker allows for the creation and management of multiple topics.

Think of it like a bookstore with different sections for various genres such as fiction, non-fiction, science fiction, and mystery. Each section represents a different topic. Similarly, in a message broker, multiple topics serve as distinct channels for messages to be published and consumed.

Publishers can choose the appropriate topic for each message they send, ensuring that it aligns with the subject matter. Subscribers, on the other hand, have the flexibility to subscribe to one or more topics based on their interests. By subscribing to specific topics, subscribers receive only the messages relevant to those topics, filtering out irrelevant information.

This capability of multiple topics allows for efficient message distribution, as it enables publishers to target specific audiences and subscribers to receive only the messages they are interested in. It also promotes modularity and scalability, as new topics can be added or removed without impacting the overall messaging system.

Message Broker Sharding

Sharding is a technique commonly used in distributed systems to horizontally partition data across multiple nodes or databases. It is primarily employed to improve scalability and performance by distributing the data workload across multiple resources.

When it comes to message brokers, sharding can be applied to achieve similar benefits. By sharding the message broker, the overall system can handle a higher volume of messages and increased throughput.

In a sharded message broker setup, the messages are partitioned across multiple message broker instances or shards. Each shard is responsible for storing and managing a subset of the messages. This allows for parallel processing and distribution of the message load.

Sharding can be implemented in different ways depending on the specific message broker technology being used. Some message brokers provide built-in sharding capabilities, while others may require manual partitioning and distribution of messages across multiple instances.

One common sharding approach is based on message topics or queues. Messages with the same topic or belonging to the same queue are directed to a specific shard. This ensures that related messages are stored together and can be efficiently processed by the corresponding shard.

Sharding a message broker can enhance scalability and performance by allowing message processing to be distributed across

multiple nodes or instances. It enables the system to handle higher message volumes, increases throughput, and improves overall system resilience. However, sharding also introduces additional complexities, such as managing shard assignments, ensuring data consistency, and handling shard failures.

It's worth noting that the decision to shard a message broker depends on the specific requirements and characteristics of the messaging workload. Sharding is typically employed in scenarios where the message volume exceeds the capacity of a single message broker instance or when high availability and fault tolerance are critical.

Idempotent Operation

An idempotent operation in the context of a message broker refers to an action or operation that can be safely repeated multiple times without altering the final result. In simpler terms, performing the same operation multiple times produces the same outcome as performing it just once. Idempotent operations are crucial in message broker systems to ensure data consistency and prevent unintended side effects. They are particularly important in scenarios where message delivery or processing can be unreliable or subject to duplication.

By designing the consumer logic to handle duplicate messages gracefully, idempotent operations help maintain system integrity. This is often achieved by using unique identifiers or sequence numbers to track and identify duplicate messages, allowing the consumer to detect and discard duplicates to avoid unintended modifications or inconsistencies in the system. Implementing idempotent operations in a message broker system provides reliability and consistency in message processing, making the system more resilient to potential message duplication or reprocessing scenarios and ensuring data integrity throughout the process.

Non-idempotent operation

In a non-idempotent message broker, the operations or actions performed by the broker are not inherently designed to be repeatable without altering the final outcome. This means that if the same operation is executed multiple times, it can result in different or unintended effects.

In such a scenario, duplicate or repeated messages can lead to undesired consequences or inconsistencies within the system. For example, if a non-idempotent message broker processes a duplicate message, it may perform the associated action multiple times, leading to data duplication, incorrect state changes, or unintended side effects.

To address the challenges of non-idempotent message processing, additional measures need to be taken. This may involve implementing deduplication mechanisms or introducing additional checks and validations to prevent the processing of duplicate messages. These measures help ensure that the non-idempotent operations are handled in a controlled and deterministic manner, minimizing the risks associated with duplicate messages.

Designing and implementing a non-idempotent message broker requires careful consideration of the application's specific requirements and the potential impact of duplicate message processing. It is essential to establish safeguards and mechanisms to handle duplicates appropriately and maintain data consistency and system integrity.

Deduplication

Deduplication in a message broker is the process of identifying and removing duplicate messages to ensure that each message is processed and delivered only once. It involves assigning unique

identifiers to incoming messages and comparing them with previously processed messages. If a duplicate message is detected, it is discarded or ignored. Deduplication helps maintain data consistency, prevents unintended side effects caused by duplicate processing, and improves the reliability and efficiency of the message broker system.

Order of Messages

In a message broker, the order of messages refers to the sequence in which they are delivered and processed. Maintaining message order can be important when messages need to be processed in a specific sequence. However, it can be challenging in distributed systems. Message brokers aim to preserve message order within a channel or topic, but factors like network latency and parallel processing can cause variations.

Some brokers offer features to prioritize order, but in some cases, scalability and performance may take precedence over strict message ordering. Overall, maintaining message order depends on system requirements and trade-offs between ordering, scalability, and performance.

Guaranteed to be Delivered at Least Once

A "guaranteed to be delivered at least once" message broker ensures that messages sent through it will reach the intended recipients without being lost or missed. It uses acknowledgments and retries to make sure messages are received. The broker stores messages and tries again if there are failures or network issues. This reliability is

important for critical processes like financial transactions or real-time events. However, it may introduce some delays. In summary, this type of message broker provides confidence that messages will be reliably delivered.

Dead-letter Queue

A dead letter queue (DLQ) is a special queue in messaging systems designed to handle messages that cannot be successfully processed or delivered. When a message encounters an error or fails to be processed, it is moved to the dead letter queue for further analysis and resolution.

The purpose of the DLQ is to capture problematic messages without disrupting the normal flow. Messages end up in the DLQ when they exceed retry limits, cannot be routed correctly, or encounter processing errors. Administrators or developers can then review and analyze the messages in the DLQ, identify the causes of failure, resolve any issues, and potentially reprocess the messages or take appropriate corrective actions. By using a dead letter queue, messaging systems can effectively manage and address problematic messages, prevent data loss, and ensure reliable message processing.

Message Broker Technologies

There are several popular message broker technologies available, each offering different features and capabilities to facilitate reliable and efficient messaging. Here are brief explanations of some commonly used message broker technologies:

Apache Kafka: Apache Kafka is a distributed streaming platform known for its high-throughput, fault-tolerant, and scalable architecture. It provides persistent, publish-subscribe messaging, where

data is stored in a distributed commit log. Kafka supports real-time event streaming, fault tolerance, and horizontal scalability, making it suitable for use cases such as real-time data processing, event-driven architectures, and data integration.

RabbitMQ: RabbitMQ is a widely adopted open-source message broker that implements the Advanced Message Queuing Protocol (AMQP). It supports various messaging patterns like publish-subscribe, request-reply, and work queues. RabbitMQ offers features such as message acknowledgments, flexible routing, and message persistence. It is known for its ease of use, extensibility, and support for multiple programming languages.

Apache ActiveMQ: Apache ActiveMQ is an open-source message broker that supports the Java Message Service (JMS) API and other protocols like MQTT and STOMP. It provides reliable messaging, message persistence, and clustering capabilities. ActiveMQ is feature-rich and widely used in enterprise applications for reliable messaging and integration scenarios.

Amazon Simple Queue Service (SQS): Amazon SQS is a fully managed message queuing service provided by Amazon Web Services (AWS). It offers reliable, scalable, and highly available message queuing with no upfront infrastructure management required. SQS supports both standard and FIFO (First-In-First-Out) queues, provides message durability, and integrates well with other AWS services.

Microsoft Azure Service Bus: Azure Service Bus is a cloud-based messaging service offered by Microsoft Azure. It provides features such as queuing, publish-subscribe messaging, and message sessions. Azure Service Bus supports multiple protocols and offers capabilities like message ordering, transactions, and dead-lettering.

Summary

Message broker technologies facilitate reliable and efficient messaging between applications and systems.

Key concepts in message brokers include:

- Publish-Subscribe: Messages are published to topics or channels and consumed by interested subscribers.
- Message Queues: Messages are sent to queues and consumed by consumers in a sequential manner.
- Message Routing: Messages are directed to specific destinations based on predefined rules or criteria.
- Message Persistence: Messages are stored durably to survive system failures or restarts.
- Acknowledgments: Senders receive acknowledgments from brokers upon successful message delivery.
- Retry Mechanisms: Failed message deliveries are retried to ensure successful processing.
- Dead Letter Queues: Messages that cannot be processed or delivered are moved to a special queue for analysis and resolution.
- Popular message broker technologies include Apache Kafka, RabbitMQ, Apache ActiveMQ, Amazon SQS, and Azure Service Bus.
- Message brokers are used in various scenarios such as real-time data processing, event-driven architectures, and enterprise integration.

21 - Map Reduce

Map-reduce operations are fundamental to both functional programming and big data. They are crucial in efficiently transforming and performing specialized operations on values. In this article, we will explore the inner workings of map-reduce, its practical applications, and its impact on data processing.

Map Reduce Operation Real-world Example Map-reduce operations have become a cornerstone of data processing in various real-world applications. Let's consider an example in the context of a large e-commerce platform.

Imagine a scenario where a popular online marketplace needs to analyze customer behavior and preferences based on their purchase history. The platform has millions of transactions recorded in its database, making it challenging to extract meaningful insights efficiently.

To tackle this problem, we need to use a map-reduce approach. In the mapping phase, the system distributes the purchase records across multiple nodes or servers. Each node applies a mapping function to extract relevant information from the raw data, such as customer ID and purchased items.

The system consolidates and aggregates the mapped data in the subsequent reduce phase. It performs operations like counting the number of purchases per customer, calculating average order value, or identifying frequently bought items. This reduction process significantly reduces the data volume and complexity.

The e-commerce platform can efficiently process and analyze vast transaction data using map-reduce operations. It can gain insights into customer preferences, identify popular products, and make

data-driven business decisions, such as targeted marketing campaigns or personalized recommendations.

Idempotent Map Reduce Operations

Idempotent map-reduce operations refer to operations we can apply multiple times without changing the final result beyond the initial application.

In simpler terms, imagine you have a set of data, and you perform a map-reduce operation on it. If you apply the same map-reduce operation again to the already processed data, the result will remain the same. It won't change or produce any additional effects.

This property is called idempotence, and it ensures that repeating the operation doesn't have any unintended consequences or alter the outcome. It provides reliability and predictability in data processing.

For example, let's consider a map-reduce operation that calculates the sum of a set of numbers. If you apply this operation once, you'll get a specific sum. If you apply it again to the already computed sum, it will still yield the same result without any changes. The operation is idempotent because repeating it doesn't modify the final sum.

Idempotent map-reduce operations are valuable in distributed systems, fault-tolerant scenarios, or when dealing with unreliable or repeated data processing. They ensure that executing the operation multiple times doesn't introduce inconsistencies or alter the desired outcome, providing stability and reliability in data processing pipelines.

Java Map Reduce Functions

You can also use the Stream API to perform MapReduce-like operations on data collections in Java. The Stream API provides a functional programming style for processing data declaratively. Here's an example of how you can use the Stream API to implement Map and Reduce operations in Java:

Map Operation: The map operation transforms each element of a stream into another element. In the context of MapReduce, it corresponds to the Map function. You can use the map() method of the Stream class to perform the mapping. Here's an example:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> squaredNumbers = numbers.stream()
                                      .map(n -> n * n)
                                      .collect(Collectors\
.toList());
```

In this example, the map() method takes a lambda expression that squares each stream element. The resulting stream is collected into a list.

Reduce Operation:

The reduce operation combines the elements of a stream into a single result. In the context of MapReduce, it corresponds to the Reduce function. You can use the reduce() method of the Stream class to perform the reduction. Here's an example:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
int sum = numbers.stream()
                 .reduce(0, (a, b) -> a + b);
```

In this example, the reduce() method takes an initial value (0) and a lambda expression that combines two elements. The reduction is performed by summing all the elements of the stream.

The Stream API provides various other methods for performing operations such as filtering, sorting, and grouping. Combining these methods allows you to implement more complex MapReduce-like operations in Java using the Stream API.

Distributed File System

In map-reduce, a Distributed File System (DFS) is crucial in providing the underlying storage infrastructure for data processing in a distributed environment.

A Distributed File System is a file system that spans across multiple machines or nodes in a cluster, allowing for the storage and retrieval of large amounts of data. It provides a unified view of the distributed storage resources and abstracts away the complexities of handling data across different nodes.

The primary goal of a Distributed File System is to provide high availability, fault tolerance, scalability, and efficient data access and retrieval. It achieves these goals through various techniques, such as data replication, partitioning, and distributed metadata management. Now, let's connect the concept of a Distributed File System to the map-reduce framework:

Data Storage: In a map-reduce workflow, we usually store the input data in a Distributed File System. The data is divided into blocks and distributed across the storage nodes in the cluster. We replicate each block to ensure data durability and availability. This distributed storage allows for parallel data processing across the cluster.

Input Splits: The map-reduce framework divides the input data into smaller units called input splits. Each input split corresponds to a block or a portion of a block stored in the Distributed File System. We then assign these input splits to different map tasks, which can be executed in parallel on different nodes.

Data Locality: One of the critical advantages of a Distributed File System in the map-reduce context is data locality. Since the input data is already distributed across the storage nodes, the map tasks can be executed on the nodes where the data is stored. This reduces data transfer overhead and improves overall performance by minimizing network traffic.

Intermediate Data: During the map phase, the map tasks generate intermediate key-value pairs. These intermediate results are temporarily stored on the local disks of the map tasks. The Distributed File System provides a reliable and distributed storage layer for this intermediate data, ensuring fault tolerance and efficient data access.

Shuffle and Reduce: In the shuffle phase of map-reduce, the framework transfers the intermediate data from the map tasks to the reduce tasks based on the keys. This data movement relies on the Distributed File System to efficiently transfer and organize the intermediate data across the network.

To summarize, a Distributed File System provides the foundation for storing and accessing data in a distributed environment. It enables efficient data storage, retrieval, and parallel processing in the map-reduce framework. The Distributed File System allows for high availability, fault tolerance, and scalability, making it an essential component in large-scale data processing systems.

Example of Hadoop Using DFS

The Hadoop Distributed File System (HDFS) is an example of a system commonly used in big data processing. It provides distributed storage, fault tolerance, scalability, and efficient data processing capabilities. HDFS divides data into blocks and distributes them across a cluster, replicating each block for fault tolerance. It manages metadata and is optimized for streaming large-scale

data. HDFS is used in various applications like distributed data processing, log/event data storage, and data warehousing.

Situations to Use Map-Reduce

A Map-Reduce solution would be good in the following situations: Large-Scale Data Processing: When you have massive data that needs to be processed efficiently, Map-Reduce offers a scalable and parallel processing approach. It enables processing data in parallel across multiple nodes, allowing faster and more efficient data processing.

Batch Processing: Map-Reduce is well-suited for batch processing workloads where data is processed in large batches or jobs. It can handle tasks that can be divided into smaller units and executed independently, making it suitable for scenarios where data processing can be done in parallel.

Distributed Computing: If you have a distributed computing environment with multiple nodes or servers, Map-Reduce provides a framework for distributed data processing. It allows you to leverage the computing power of multiple machines to process data in parallel, enabling faster processing times.

Fault Tolerance: Map-Reduce frameworks like Hadoop MapReduce offer built-in fault tolerance mechanisms. If a node or server fails during processing, the framework can automatically reroute the task to a different node and continue processing without data loss or interruption.

Scalability: Map-Reduce solutions can scale horizontally by adding more nodes to the cluster. As the data volume or processing requirements increase, additional nodes can be added to handle the workload. This scalability allows for efficient processing of large datasets and accommodates growth in data processing needs.

Data Preprocessing and Transformation: We often use Map-Reduce for data preprocessing and transformation tasks. It allows you to perform operations like filtering, sorting, aggregating, and transforming data in parallel. This is particularly useful when working with large datasets that need to be processed before further analysis or modeling.

Log Processing and Analysis: Map-Reduce is commonly used for processing and analyzing large volumes of log data. It can efficiently handle log files generated by systems, applications, or devices, extracting relevant information and performing aggregations or calculations on the log data.

Overall, a Map-Reduce solution is beneficial when processing large-scale data, performing batch processing, leveraging distributed computing capabilities, ensuring fault tolerance, achieving scalability, and handling preprocessing or transformation tasks. It provides a scalable and parallel processing approach that efficiently handles significant amounts of data.

Map Reduce Technologies

Several popular technologies and frameworks are available for implementing the MapReduce programming model. Here are some of the widely used ones:

Apache Hadoop: Hadoop is an open-source framework that provides a distributed computing platform for processing and analyzing large datasets. It includes the Hadoop Distributed File System (HDFS) for distributed storage and the MapReduce processing model. Hadoop is written in Java and is known for its scalability, fault tolerance, and flexibility.

Apache Spark: Spark is an open-source, distributed computing framework that provides an alternative to MapReduce. It offers in-memory processing, which makes it faster than traditional MapRe-

duce. Spark supports various programming languages such as Java, Scala, Python, and R and provides rich libraries for data processing, machine learning, graph processing, and more.

Apache Flink: Flink is an open-source, stream processing framework that also supports batch processing. It provides low-latency, high-throughput processing of streaming data and supports event time processing, exactly-once semantics, and stateful computations. Flink supports multiple programming languages and provides a flexible API for expressing complex data processing pipelines.

Apache Storm: Storm is an open-source, real-time stream processing framework. It is designed for processing large volumes of data in real time and provides fault tolerance and horizontal scalability. Storm processes data continuously, making it suitable for use cases like real-time analytics, fraud detection, and IoT data processing.

Apache Tez: Tez is an open-source framework built on top of Hadoop YARN. It provides an optimized execution engine for running data processing tasks in a distributed manner. Tez aims to improve the performance of Hadoop MapReduce by optimizing resource utilization and reducing the overhead of launching separate MapReduce jobs.

Google Cloud Dataflow: Dataflow is a fully managed service offered by Google Cloud Platform (GCP) for processing and analyzing data in both batch and streaming modes. It provides a unified programming model based on the Apache Beam SDK, which allows you to write data processing pipelines in various programming languages. Dataflow handles the underlying infrastructure and automatically scales resources based on the workload.

These are just a few examples of the map reduce technologies available. Each technology has its strengths and is suited for different use cases. The choice of technology depends on factors such as data volume, processing requirements, programming language preference, and the ecosystem of tools and libraries available.

Conclusion

Here's a summary of the critical concepts of MapReduce in bullet points:

MapReduce is a programming model for processing and analyzing large datasets in a distributed manner. It consists of two main phases: Map and Reduce.

- In the Map phase, data is divided into chunks and processed in parallel by multiple map tasks.
- Each map task takes a set of input key/value pairs and produces intermediate key/value pairs.
- The intermediate key/value pairs are then shuffled and sorted to group them by key.
- In the Reduce phase, the grouped intermediate key/value pairs are processed by multiple reduce tasks in parallel.
- Each reduce task takes a group of intermediate key/value pairs with the same key and produces a final output.
- The output of the reduce tasks is typically aggregated or combined to produce a final result.
- MapReduce provides fault tolerance by automatically handling failures and rerunning failed tasks on other nodes.
- It is designed to scale horizontally by adding more nodes to the cluster to process larger datasets.
- MapReduce is commonly used in distributed computing frameworks like Apache Hadoop, Apache Spark, and Apache Flink.
- It allows developers to write parallelizable and scalable data processing tasks without managing the underlying distributed infrastructure.

These bullet points provide a concise overview of the key concepts of MapReduce. It's important to note that there are additional details and considerations when implementing MapReduce in specific technologies or frameworks.

22 - Security with HTTPS

Security is not asked much in Systems Design interviews, but we need to know the difference between HTTP and HTTPS to avoid common hack attacks.

This article won't go deep into security because it's a vast topic. However, as software engineers, we need to know the basics of security to prevent systems from being hacked because of a basic mistake.

Therefore, let's explore the essence of HTTP and HTTPS to know why they're important and why we should use HTTPS instead of HTTP.

The Problem with HTTP

HTTP (Hypertext Transfer Protocol) is the foundation of communication on the World Wide Web. While HTTP has been widely successful in enabling the exchange of information between clients (such as web browsers) and servers, it does have a few limitations and shortcomings:

Lack of security: HTTP is inherently insecure because it does not encrypt the transmitted data. That's because any information sent over HTTP, including sensitive data like passwords or credit card details, can be intercepted and read by malicious actors. HTTPS (HTTP Secure) addresses this issue by adding a layer of encryption using SSL/TLS protocols.

Lack of state: HTTP is a stateless protocol, which means that each request from a client to a server is independent and does not carry any context or memory of previous requests. This poses challenges

when building web applications that require maintaining session information or user authentication across multiple requests. Developers often use techniques like cookies or server-side sessions to work around this limitation.

Performance overhead: HTTP relies on a request-response model, where each client request results in a separate response from the server. This introduces overhead due to the repeated establishment and termination of connections. Additionally, the text-based nature of HTTP messages can lead to larger payload sizes, which can impact performance, especially on low-bandwidth or high-latency networks.

Limited flexibility: HTTP has a fixed set of methods (GET, POST, PUT, DELETE, etc.) that define the operations that can be performed on resources. While these methods cover many common use cases, they might be insufficient for more complex interactions or custom requirements. This limitation has led to the development of additional protocols, such as WebDAV and RESTful APIs, to extend the capabilities of HTTP.

Lack of real-time communication: HTTP is primarily designed for request-response communication, which makes it less suitable for real-time or bidirectional communication scenarios, such as instant messaging or live streaming. To address this, alternative protocols like WebSockets or technologies like Server-Sent Events (SSE) have been introduced to enable real-time communication over HTTP.

It's worth noting that while HTTP has its limitations, it has been the foundation of the modern web and has undergone significant improvements over the years. Many of these limitations have been addressed through various extensions, protocols, and best practices, making HTTP a widely adopted and reliable protocol for web communication.

Man in the Middle Attack

As mentioned, HTTP is insecure, making it vulnerable to the man-in-the-middle attack.

To understand this attack, imagine you want to visit a website using your web browser. Usually, your browser communicates directly with the website to exchange data. However, in a Man-in-the-Middle attack, an unauthorized third party intercepts and manipulates the communication between your browser and the website.

Here's how it works:

Normal Communication: In a regular HTTP connection, your browser sends requests to a website, which responds with the requested data. This communication happens directly between your browser and the website's server.

Interception: In a Man-in-the-Middle attack, an attacker positions themselves between your browser and the website. An attacker can intercept the request through various means, such as by compromising a Wi-Fi network or by gaining control over a network router.

Impersonation: Once the attacker is in the middle, they can impersonate your browser and the website. The attacker pretends to be your browser when communicating with the website and pretends to be the website when communicating with your browser. This allows them to intercept and manipulate the data being exchanged.

Data Manipulation: With control over the communication, the attacker can read, modify, or even inject their own data into the messages passing between your browser and the website. For example, they could steal sensitive information like login credentials or credit card details by capturing them before they reach the legitimate website.

Transparent Relay: To make the attack less noticeable, the attacker typically relays the intercepted data between your browser and the website in real-time. This means that your browser may still receive responses from the website, creating the illusion of a normal connection, while the attacker is secretly manipulating the data in between.

MitM attacks can be particularly dangerous because they allow attackers to eavesdrop on sensitive information or carry out fraudulent activities without the knowledge of the victim or the website. That's why it's important to use secure protocols like HTTPS, which encrypts the communication between your browser and the website, making it much harder for attackers to perform successful MitM attacks.

Remember to exercise caution when connecting to public Wi-Fi networks or when accessing websites that handle sensitive information. Always look for the secure padlock icon and use websites with HTTPS to help protect against Man-in-the-Middle attacks.

HTTPS Process

Let's learn first basic security elements so HTTPS makes sense. We will understand first:

- Symmetric Encryption Key – one key cryptography
- Asymmetric Encryption Key – two keys cryptography
- SSL certificate (Secure socket layer) – certificate for the Certificate authority
- TLS Handshake – estabilish a secret code for communication
- Certificate Authority – authority that registers the domain

Symmetric Encryption key

In cryptography, a symmetric key is a type of encryption where the same key is used for both the encryption and decryption of data. It's like having a single key that can lock and unlock a door.

Here's how it works:

Key Generation: To use symmetric key encryption, a secret key is generated. This key is a string of bits or characters that can be randomly generated or derived from a password or passphrase.

Encryption: When you want to encrypt a message or data using a symmetric key, you take the plain text (the original, unencrypted data) and the secret key. Using an encryption algorithm, the secret key is applied to the plain text to produce the encrypted data, also known as the cipher text.

Decryption: To decrypt the encrypted data and retrieve the original plain text, you use the same secret key that was used for encryption. Applying the key with a decryption algorithm reverses the encryption process, transforming the cipher text back into the plain text.

Key Sharing: One challenge with symmetric key encryption is securely sharing the secret key between the sender and the recipient. If an attacker intercepts the key during transmission, they can also decrypt the encrypted data. Secure key distribution methods, such as using secure channels or exchanging keys in person, are crucial to protect against unauthorized access.

Symmetric key encryption is generally faster than other encryption methods, such as asymmetric key encryption, because the algorithms used are computationally less complex. It's commonly used for encrypting large amounts of data and for securing communication channels.

However, one limitation of symmetric key encryption is that the same key needs to be securely shared between the sender and the

recipient. This can be a challenge, especially when communicating over untrusted networks or between multiple parties.

AES Symmetric Encryption

AES is a widely used symmetric encryption algorithm that is used to secure sensitive information. It is considered one of the most secure encryption algorithms available today.

Here's how it works:

Key Generation: To use AES, a secret key is generated. The key is a string of bits or characters that must be kept secret and shared only between the sender and the recipient.

Block Encryption: AES operates on fixed-size blocks of data, typically 128 bits (16 bytes) in length. The plain text (the original, unencrypted data) is divided into these blocks. Each block is then encrypted independently using the secret key.

Encryption Rounds: AES uses a series of encryption rounds to transform the plain text block into the encrypted form, known as the cipher text. These rounds involve several operations, including substitution, permutation, and mixing of the data, based on the key.

Key Expansion: The original secret key is expanded to create a set of round keys, which are used in the encryption rounds. This key expansion process ensures that each round uses a different key, increasing the security of the encryption.

Decryption: To decrypt the cipher text and retrieve the original plain text, the same secret key is used in reverse. The cipher text is divided into blocks, and each block is decrypted independently using the secret key and the inverse of the encryption operations performed in AES.

AES provides a high level of security due to its robust encryption process and the use of multiple rounds. It is resistant to various

known attacks, making it suitable for a wide range of applications, including securing data in transit and at rest.

The strength of AES lies in the key length used, with 128-bit, 192-bit, and 256-bit key sizes being the most common. The larger the key size, the stronger the encryption, but also the more computationally intensive the encryption and decryption processes become.

Asymmetric Encryption Key

In cryptography, asymmetric key encryption (also known as public-key encryption) is a method that uses two separate but mathematically related keys: a public key and a private key. We commonly use an asymmetric key on GitHub for the initial SSH (Secure Shell) handshake to access resources.

Now, let's see how it works:

Key Generation: First, a user generates a pair of keys: a public key and a private key. The keys are mathematically linked, but it is computationally infeasible to derive one key from the other.

Public Key Distribution: The user shares their public key freely with others. The public key is used to encrypt data and verify digital signatures. It can be openly distributed without compromising the security of the encryption.

Encryption: When someone wants to send a secure message to the user, they use the recipient's public key to encrypt the message. The encryption process uses the public key to transform the original message into an encrypted form, known as the cipher text.

Private Key Decryption: Only the user in possession of the private key can decrypt the cipher text. The private key is kept secret and should not be shared. To decrypt the encrypted message, the user applies their private key, which reverses the encryption process and retrieves the original plain text.

One of the significant advantages of asymmetric key encryption is that the public key can be freely distributed, allowing anyone to encrypt messages for the intended recipient. However, decryption can only be done using the private key, which remains secret and known only to the recipient.

Asymmetric key encryption is commonly used for various purposes, including secure communication, digital signatures, and key exchange protocols. It enables secure communication even when the sender and recipient have no prior shared secret.

That's a simplified explanation of asymmetric key encryption! It's a cryptographic method that uses two related but distinct keys: a public key for encryption and a private key for decryption.

SSL certificate (Secure socket layer)

An SSL certificate is like a digital passport that confirms the identity of a website and enables secure communication between your web browser and the website. It helps ensure that the information you exchange with the website remains private and cannot be easily intercepted or tampered with by attackers.

Here's how it works:

Website Identity: When a website wants to obtain an SSL certificate, it goes through a process to prove its identity. This involves providing information about the website and its ownership to a trusted certificate authority (CA).

Certificate Issuance: The certificate authority verifies the website's information and if everything checks out, it issues an SSL certificate specific to that website. The certificate contains the website's public key, its domain name, and other details.

Certificate Installation: The website installs the SSL certificate on its web server. This enables the server to encrypt and decrypt

information using the public and private keys associated with the certificate.

Secure Communication: When you visit a website with an SSL certificate, your web browser checks the certificate to ensure it's valid. It verifies if a trusted CA has signed the certificate and if it has not expired. This step helps confirm that you are indeed connecting to the genuine website.

Encryption: Once the certificate is validated, your browser and the website initiate an encrypted connection. They use the website's public key from the certificate to encrypt the data you send and receive. This encryption ensures that even if someone intercepts the data, they won't be able to understand it without the corresponding private key, which only the website possesses.

Using an SSL certificate, websites can establish a secure connection with your browser, protecting sensitive information such as login credentials, credit card details, and other personal data from unauthorized access.

You can identify a website that has an SSL certificate by looking for the padlock icon in your browser's address bar or seeing "https://" at the beginning of the website's URL, where the "s" stands for "secure."

That's a simplified explanation of an SSL certificate! It's an important tool in ensuring secure communication and protecting your sensitive information online.

Certificate Authority

A Certificate Authority (CA) is a trusted entity that issues digital certificates to verify the authenticity of websites, servers, or individuals. It plays a crucial role in establishing secure communication over the Internet.

When a website wants to use HTTPS and secure its communication with users, it must obtain an SSL/TLS certificate from a trusted CA. The CA verifies the identity and ownership of the website before issuing the certificate. The certificate contains information about the website, including its domain name, public key, and the CA's digital signature.

When a user visits a website secured with HTTPS, their web browser checks the validity of the SSL/TLS certificate presented by the website. The browser verifies the certificate's digital signature and checks its authenticity by cross-referencing it with a list of trusted CA certificates pre-installed in the browser or operating system. If the certificate is valid and issued by a trusted CA, the browser establishes an encrypted connection with the website.

The role of a CA is crucial in maintaining the trust and security of the certificate ecosystem. CAs are responsible for verifying the identity of the certificate requestor before issuing a certificate, ensuring that only legitimate entities receive valid certificates. They follow industry standards and security practices to safeguard the integrity and confidentiality of the certificate issuance process.

However, it's important to note that CAs can make mistakes or be compromised, leading to fraudulent or malicious certificates being issued. In recent years, there have been instances where CAs were compromised and unauthorized certificates were issued. To mitigate this risk, browser vendors and operating system manufacturers maintain a list of trusted CAs and regularly update it to remove compromised or untrustworthy CAs.

In summary, a Certificate Authority is a trusted entity that issues digital certificates to validate the identity and authenticity of websites, servers, or individuals, enabling secure online communication.

Certificate Authority Examples

Let's explore a few examples of well-known certificate authorities:

Let's Encrypt: Let's Encrypt is a free, automated, and open certificate authority. It aims to make it easy for website owners to obtain and install SSL/TLS certificates. Let's Encrypt is widely used and supported by many hosting providers and web browsers.

DigiCert: DigiCert is a leading global certificate authority that provides a wide range of SSL/TLS certificates and related security solutions. They offer certificates for various purposes, including website encryption, code signing, and document signing.

Sectigo (formerly Comodo CA): Sectigo is a prominent certificate authority offering a comprehensive range of SSL/TLS certificates, including domain validation, organization validation, and extended validation certificates. They also provide other security solutions such as secure email and code signing certificates.

GlobalSign: GlobalSign is a trusted certificate authority that offers a variety of SSL/TLS certificates, including wildcard certificates, multi-domain certificates, and extended validation certificates. They cater to different types of organizations, from small businesses to large enterprises.

GoDaddy: GoDaddy is a well-known provider of domain registration and web hosting services, and they also offer SSL/TLS certificates. They provide a range of certificate options and support for various needs, including single-domain, multi-domain, and wildcard certificates.

These are just a few examples of certificate authorities, and there are many other trusted CAs available in the market. When choosing a certificate authority, it's important to consider factors such as their reputation, compatibility with web browsers and devices, customer support, and pricing.

TLS handshake

Imagine you want to have a secure conversation with someone over the internet. The TLS handshake is like establishing a secret code between you and the other person so that you can communicate securely.

Here's how it works:

Client Hello: The client (your web browser, for example) starts the handshake by sending a "hello" message to the server. This message includes information like the supported encryption algorithms and other details.

Server Hello: The server responds with its own "hello" message. It selects the strongest encryption algorithm that both the client and server support. The server also sends its digital certificate, which includes a public key.

Certificate Validation: The client checks the server's digital certificate to make sure it's valid. It verifies the certificate's authenticity by checking if it's been signed by a trusted certificate authority (CA). This step ensures that the client is communicating with the genuine server.

Key Exchange: The client generates a random session key, which is a secret code that will be used to encrypt and decrypt the messages during the session. The client encrypts this session key using the server's public key obtained from the certificate and sends it back to the server.

Server Key Decryption: The server receives the encrypted session key from the client. It uses its private key (which is paired with the public key in the certificate) to decrypt the session key.

Session Established: Now, both the client and server have the same session key. They use this key to encrypt and decrypt their messages during the session. This ensures that the communication

is secure and cannot be easily intercepted or tampered with by attackers.

Once the handshake is complete, the client and server can start exchanging encrypted data, such as web pages, securely. The TLS handshake typically happens once at the beginning of a session, and a new handshake is performed if the session needs to be re-established or if a new connection is made.

HTTPS

HTTPS, or Hypertext Transfer Protocol Secure, is a protocol used for secure communication over the internet. It ensures that the data sent between a web browser and a website is encrypted and cannot be easily intercepted or tampered with by malicious actors.

When you visit a website that uses HTTPS, your browser establishes a secure connection with the website's server. This connection is encrypted, meaning that any data transmitted between your browser and the server is encoded in a way that only the intended recipient can understand it.

The encryption is achieved using SSL (Secure Sockets Layer) or its successor TLS (Transport Layer Security) protocols. These protocols use cryptographic algorithms to encrypt the data and verify the identity of the website.

Here's a simplified explanation of how HTTPS works:

Client Request: You type in a website's address (e.g., https://www.example.com) in your browser.

Server Response: Your browser sends a request to the server asking for the website's content.

SSL/TLS Handshake: The server responds by sending its SSL/TLS certificate, which contains a public key. Your browser uses this public key to initiate the handshake process.

Secure Connection Established: Your browser generates a random symmetric encryption key and encrypts it using the server's public key. This encrypted key is sent to the server.

Encrypting Data: Now, your browser and the server have established a secure connection. Any data transmitted between them, such as web pages, form submissions, or personal information, is encrypted using the symmetric encryption key.

Data Exchange: Encrypted data is exchanged between your browser and the server. Even if someone intercepts this data, they won't be able to read its contents without the encryption key.

Secure Session: The secure connection remains active until you close your browser or navigate away from the website.

The use of HTTPS provides several advantages. It helps protect sensitive information, such as login credentials, credit card details, and personal data, from being intercepted and stolen. It also ensures the integrity of the data, as any tampering with the encrypted content would be detected.

In summary, HTTPS is a secure version of the HTTP protocol that encrypts data during transmission, providing a safer and more private browsing experience for users.

Summary

Let's recap the concepts we explored in this article so we remember the content more easily:

HTTP (Hypertext Transfer Protocol):

- Protocol used for transmitting data over the internet.
- Operates over unencrypted connections, making it vulnerable to eavesdropping and tampering.

HTTPS (Hypertext Transfer Protocol Secure):

Secure version of HTTP that uses encryption to protect data transmitted over the internet. Adds a layer of security by using SSL/TLS protocols. TLS (Transport Layer Security):

- Cryptographic protocol that provides secure communication over a network.
- Encrypts data and ensures its integrity during transmission.
- Successor to SSL (Secure Sockets Layer) and commonly used for secure communication on the web.

SSL (Secure Sockets Layer):

- Cryptographic protocol used for secure communication over a network.
- Predecessor to TLS and widely used in the past for securing web connections.
- Provides encryption, data integrity, and authentication.

Certificate Authority (CA):

- Trusted entity that issues digital certificates.
- Verifies the identity of individuals, organizations, or servers.
- Certificates are used for authentication and encryption in SSL/TLS.

Symmetric Key:

- Encryption method that uses a single shared key for both encryption and decryption.
- Fast and efficient for bulk data encryption.
- Requires secure key distribution between communicating parties.

Asymmetric Key:

- Encryption method that uses a pair of mathematically related keys: a public key and a private key.
- Public key is used for encryption, while the private key is kept secret and used for decryption.
- Enables secure key exchange and digital signatures.

AES (Advanced Encryption Standard):

- Symmetric encryption algorithm widely used for secure data transmission.
- Provides strong encryption and is considered secure for most practical purposes.
- Adopted as a standard by the U.S. government and used globally.

23 - Design Amazon

During a Systems design interview, we have approximately 45 minutes to design an entire system. Obviously, it's impossible to design a system in depth. Instead, we need to think about the system at a high level.

We also have to make specific questions assuming we will have only the essencial features without too much complexity, otherwise, it will be impossible to even complete the design.

With that in mind let's see what questions we should make, how to design a high-level Amazon e-commerce, the database model so that we know better what to do in such interview.

Ask clarifying questions

When designing an Amazon-like e-commerce system for a systems design interview, you can ask the following clarifying questions to gather more information and understand the requirements:

Expected User Base: What is the expected number of concurrent users and daily active users? Understanding the scale of the system will help determine the required infrastructure and scalability considerations.

Geographical Scope: Is the e-commerce platform targeting a specific region or operating globally? This will impact decisions related to language support, currency conversion, and content delivery.

Product Diversity: What types of products will be sold on the platform? Are they physical goods, digital products, or both?

The nature of the products will influence the system's handling of inventory, shipping, and fulfillment.

Third-Party Sellers: Will the platform allow third-party sellers to offer their products? If yes, what are the requirements for onboarding and managing these sellers? This will affect the system's architecture, user management, and payment processing.

Availability and Performance: What are the desired availability and performance targets for the system? Is there a specific service level agreement (SLA) to meet? This information will guide decisions related to redundancy, load balancing, and caching.

Security Considerations: What security measures should be implemented to protect user data, prevent unauthorized access, and secure transactions? Understanding the desired security level will influence decisions related to authentication, authorization, and data encryption.

Payment Methods: What payment methods will be supported? Will the system integrate with various payment gateways or use a custom payment processing solution? This will impact the payment gateway integration and the handling of financial transactions.

Mobile Support: Will the system require a dedicated mobile application or have responsive web design for mobile devices? Understanding the mobile requirements will influence decisions related to user interface design and API development.

Personalization and Recommendations: Will the system provide personalized product recommendations based on user behavior and preferences? Understanding the need for personalization will influence decisions related to user profiling, recommendation algorithms, and data storage.

Internationalization and Localization: Will the system support multiple languages, currencies, and regional preferences? This will impact decisions related to content management, localization of product information, and international shipping.

Analytics and Reporting: What types of analytics and reporting capabilities are required? Are there any specific metrics or performance indicators that need to be tracked? Understanding the reporting needs will guide decisions related to data collection, storage, and analysis.

Remember, these questions are just a starting point, and the interviewer may provide additional context or constraints that will inform your design further.

DB Model Design

Before designing our DB model, we must keep in mind that we won't be able to design a very robust DB model. The following model is a simplified version with the essencial tables for essential features:

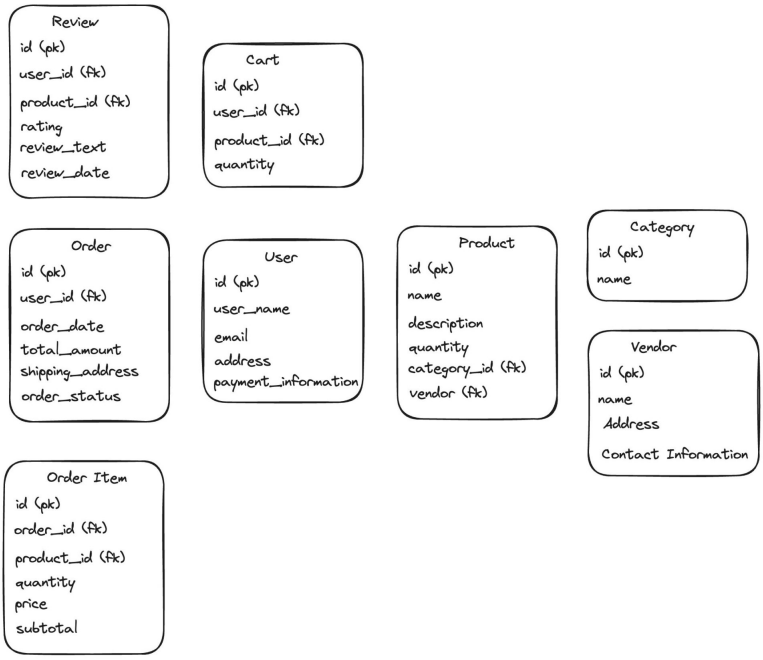


Figure 16. Amazon E-Commerce DB Design

Highlevel Systems design

Designing an Amazon-like e-commerce system for a systems design interview requires careful consideration of various components and their interactions. Here's a high-level overview of the system architecture:

Client Interface: Users interact with the system through a web or mobile interface. The interface allows them to search for products, view product details, add items to their cart, and proceed to checkout.

Load Balancer: Incoming requests from clients are distributed

across multiple web servers using a load balancer. The load balancer ensures optimal resource utilization and high availability.

Web Servers: The web servers handle client requests and generate dynamic web pages. These servers communicate with other components to fetch product information, user data, and process transactions.

Product Catalog: The product catalog stores information about the available products. It includes details like product name, description, price, images, and other attributes. The catalog can be stored in a relational database or a NoSQL store for scalability.

User Management: The system manages user accounts, authentication, and authorization. User information such as names, addresses, and payment methods can be stored in a user database. Authentication mechanisms like OAuth or JSON Web Tokens (JWT) can be used to secure user sessions.

Shopping Cart: The shopping cart holds the selected items for each user during their shopping session. It can be implemented using a combination of server-side and client-side storage mechanisms like cookies or sessions.

Order Processing: When a user checks out, the system processes the order. It verifies the availability of items, calculates the total cost, and generates an order confirmation. The order details are stored in the database, and inventory is updated accordingly.

Payment Gateway: The payment gateway handles payment processing. It securely communicates with external payment providers to authorize and process credit card transactions or other payment methods.

Inventory Management: The inventory management system tracks the availability of products. It ensures that products are not oversold and updates the inventory as orders are placed or canceled. Inventory can be stored in a separate database or integrated with the product catalog.

Reviews and Ratings: Users can provide reviews and ratings for products. The system stores this information and displays it on the product details page.

Caching and Content Delivery: To improve performance, caching mechanisms like Content Delivery Networks (CDNs) can be used to cache static content such as product images, CSS, and JavaScript files. This reduces the load on the web servers and improves response times.

Analytics and Monitoring: The system collects data on user interactions, sales, and performance metrics. This data can be used for generating reports, monitoring system health, and making data-driven decisions.

Search Functionality: An efficient search mechanism is crucial for an e-commerce system. Implementing a search engine or utilizing search platforms like Elasticsearch can help provide fast and accurate search results.

It is important to note that this is a high-level overview, and the actual implementation details would require more in-depth analysis and consideration of specific requirements and constraints.

Summary

This should be enough for a Systems design interview. Of course, you will have to go with the flow of the interview but the base design for Amazon e-commerce is what we discussed.

Now, let's see the most important points to get prepared for the systems design interview. The same principles to design Amazon e-commerce are applied for other systems. Let's recap them:

- Designing an Amazon-like e-commerce system requires careful consideration of various components and their interactions.

- The system architecture includes a client interface, load balancer, web servers, product catalog, user management, shopping cart, order processing, payment gateway, inventory management, reviews and ratings, caching and content delivery, analytics and monitoring, search functionality, and more.
- Clarifying questions should be asked to gather information about the expected user base, geographical scope, product diversity, third-party seller integration, availability and performance targets, security considerations, payment methods, mobile support, personalization and recommendations, internationalization and localization, and analytics and reporting.
- The database design for the e-commerce system includes tables for users, products, orders, order items, reviews, shopping carts, categories, and vendors/manufacturers.
- Optimization techniques such as indexing, normalization, and data partitioning can be considered.
- The design should focus on scalability, efficiency, data integrity, and security.
- The specific design decisions may vary based on requirements and constraints, and it's important to discuss and validate design choices during the interview.

24 - Design Discord

We must make strategic questions to design Discord for a Systems Design interview and assume it's simple because the interview usually takes 45 minutes.

When we ask questions regarding Discord, we need to take notes of the essential features we need to consider so we can organize our ideas on how to design Discord.

In case you don't know Discord, it's a communication tool similar to Slack, where we can interact in different groups and channels, restrict channel access, send direct messages, and send a message to a group.

We need cloud knowledge to design a system and know what technologies we should use. Let's first explore which questions we should ask to design Discord for a Systems Design interview.

Clarifying Questions to Design Discord

When designing Discord for a systems design interview, here are some clarifying questions you may want to ask:

- We will design only the essential features from Discord, right? Should we design the system to support the creation of Discord servers, direct messages, group messages, and group access?
- What is the expected number of concurrent users and servers should the system support?

- Are there any specific performance requirements, such as response time or throughput?
- Should the design be optimized for read-heavy or write-heavy workloads?
- Should I consider specific security and privacy considerations?
- Should the design support voice and video calls or focus solely on text-based communication, or will we stick to the basics?
- Is there a need for real-time updates and notifications?
- Are there any specific integrations or third-party services that the design should accommodate?
- What are the expected growth projections for the user base and usage patterns over time?
- Should the design be scalable across multiple regions or data centers?
- These questions will help you gather important information about the requirements and constraints of the system, allowing you to design a more accurate and effective solution. Remember to clarify any ambiguous requirements and discuss potential trade-offs with the interviewer.

Cloud Technologies for Discord

When designing Discord for a systems design interview, you can consider using the following cloud components:

Virtual Machines (VMs): VMs can be used to host the Discord application servers, which handle user authentication, message processing, and other business logic.

Load Balancers: Load balancers distribute incoming traffic across multiple Discord application servers, ensuring scalability and high availability. To explore Load Balancer, check out the following

article, Master the Principles of Load Balancer for Systems Design Interview.

Database Service: You can use a managed database service like Amazon RDS, Google Cloud SQL, or Azure Database for hosting user profiles, messages, and other persistent data.

Object Storage: Cloud object storage services like Amazon S3, Google Cloud Storage, or Azure Blob Storage can be used for storing media files, such as user avatars and attachments.

Content Delivery Network (CDN): A CDN can help optimize the delivery of static assets, such as images and emojis, to users across different regions.

Message Queues: Message queues such as Amazon SQS, Google Cloud Pub/Sub, or Azure Service Bus can be used to decouple components and handle asynchronous message processing, such as sending notifications or processing background tasks. If you are unfamiliar with message queues, check out the following article Message Brokers: Publish Subscribe Communication for Systems Design.

Caching Service: A caching service like Redis or Memcached can be employed to cache frequently accessed data, improving system performance and reducing database load. To learn more about cache, check out the following article Mastering the Fundamentals of Cache for Systems Design Interview.

Real-time Communication Infrastructure: To handle voice and video calls, you may need to utilize real-time communication infrastructure, such as WebRTC or a specialized service like Twilio.

Monitoring and Logging: Cloud monitoring and logging services like Amazon CloudWatch, Google Cloud Monitoring, or Azure Monitor can be used to gain insights into system health, performance, and troubleshooting. To learn more about monitoring and logging take a look at the following article Logging and Monitoring to Debug Cloud Services.

Containerization: You can use containerization technologies like Docker and container orchestration platforms like Kubernetes to manage and scale the Discord application containers.

Remember to evaluate each component's pros and cons based on the Discord system design's specific requirements and constraints.

Discord Implementation Technologies

Discord is primarily developed using various technologies, including JavaScript for the client-side application and Go for the server-side infrastructure.

The client-side application of Discord, which runs in web browsers and desktop applications, is built using JavaScript, HTML, and CSS. It utilizes frameworks and libraries like React for the user interface components and Redux for state management.

On the server side, Discord uses Go, a programming language known for its performance and concurrency capabilities, to build the backend infrastructure. Go is used for handling real-time communication, managing user authentication, processing messages, and managing the overall system architecture.

While Java can be used for building large-scale applications, Discord's development stack primarily consists of JavaScript and Go.

Basic Database model from Discord

Let's look at how we can design a simple database model for Discord. It's the same principle here when we are asking questions. We need to make it as simple as possible. Only the most essential

tables are needed because we must explain where we use the following tables in the interview.

The following simplified DB model is intuitive; for each server, we can have many channels; in a channel, we can have many messages from many users, and we also have information from users inside a Discord server:

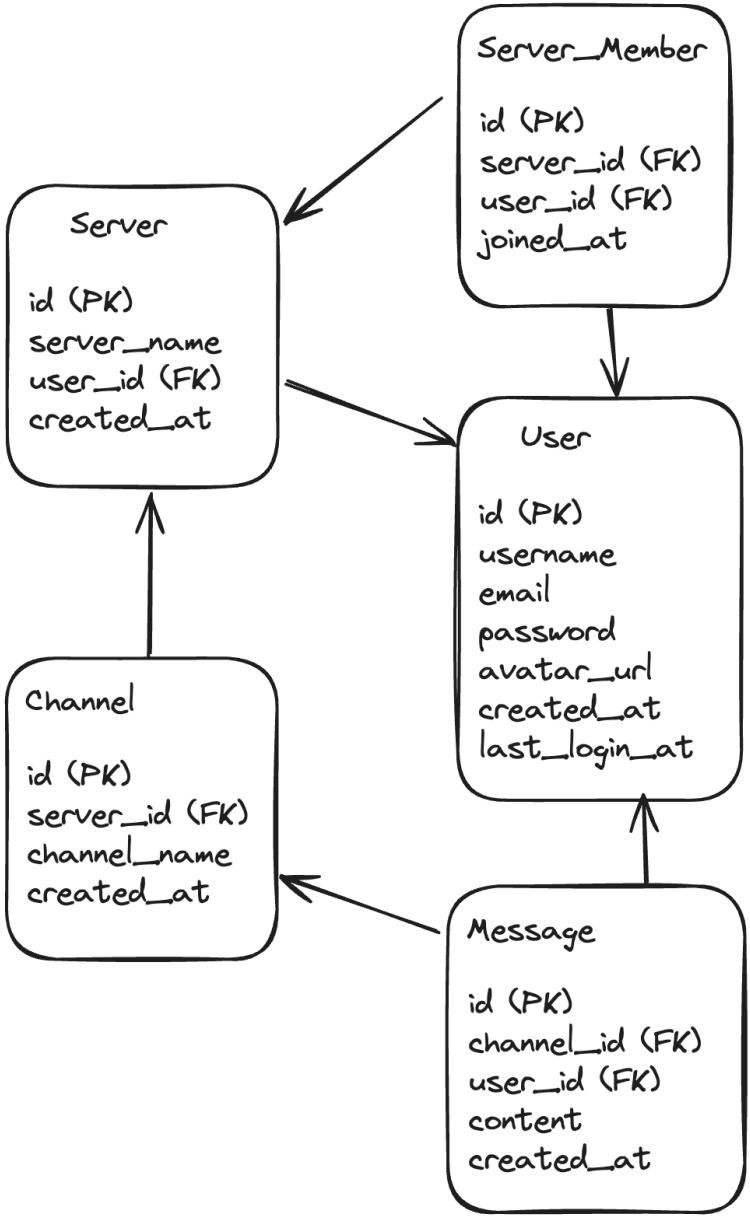


Figure 17. Discord Database Model

Summary

Clarifying Questions to Design Discord:

- Main features and functionalities
- Number of concurrent users and servers
- Performance requirements
- Read-heavy or write-heavy workload
- Security and privacy considerations
- Voice and video call support
- Real-time updates and notifications
- Integrations or third-party services
- User base growth projections
- Scalability across regions or data centers
- Cloud Technologies for Discord:

Virtual Machines (VMs)

- Load Balancers
- Database Service
- Object Storage
- Content Delivery Network (CDN)
- Message Queues
- Caching Service
- Real-time Communication Infrastructure
- Monitoring and Logging
- Containerization

Technologies Discord was developed With:

- Client-side: JavaScript, HTML, CSS, React, Redux
- Server-side: Go

Basic Database Model for Discord:

- User table
- Server table
- Server_Member table
- Channel table
- Message table

25 - Design Instagram

// To Update

26 - Design Netflix with Java

To design Netflix in a Systems Design interview, we must make questions assuming we will only design the most important features. Otherwise, we won't be able to finish the design in 45 minutes.

Making questions is challenging during a Systems design interview. Therefore, let's see some questions we can ask during an interview.

Netflix Overall Concepts to Keep in Mind for Systems Design Interview Before asking good questions to design Netflix in a Systems Design interview, we need to keep in mind core concepts that will make our Microservices robust and reliable.

If Microservices are still something confusing for you, take a look at this article: The Ultimate Guide of Microservices Technologies for Java Developers

Now let's explore the high-level requirements to design a robust Netflix system.

Scalability: The system should be able to handle a massive number of users and content streams simultaneously.

High Availability: Implement mechanisms to ensure that the service remains accessible and operational even in the face of failures or heavy traffic.

Content Delivery Network (CDN): Utilize a CDN to distribute content closer to end-users, reducing latency and improving performance.

Microservices Architecture: Design the system as a collection of loosely coupled services that can be independently developed,

deployed, and scaled. Service Discovery: Implement a service discovery mechanism to allow services to locate and communicate with each other effectively.

Caching: Utilize caching strategies to store frequently accessed data and reduce the load on backend systems.

Fault Tolerance: Employ mechanisms such as circuit breakers and retries to handle failures gracefully and prevent cascading failures.

Data Partitioning: Partition data across multiple storage nodes to distribute the load and improve performance.

Analytics and Monitoring: Incorporate monitoring tools and analytics to gain insights into system performance, user behavior, and content preferences.

Security: Implement robust security measures to protect user data, prevent unauthorized access, and ensure content rights management.

Recommendation Algorithms: Design intelligent recommendation systems to personalize content recommendations based on user preferences and behavior.

Remember that designing a system like Netflix requires a comprehensive understanding of system design principles, scalability, performance optimization, and various technologies. These bullet points provide a high-level overview, and each point can involve a deeper complexity and considerations.

Design Netflix Clarifying Questions

Remember that when preparing questions for a Systems Design interview, always assume something. For example, in the first question here, we will assume that we will develop only the basic features. Designing only the main features will help in the interview.

Therefore, let's explore some questions about designing Netflix for the Systems design interview:

- I assume we will have only the essential features, such as video streaming and personalized recommendations, right?
- What is the expected user base and traffic volume that the Netflix system handles?
- Are there any specific performance or latency requirements for delivering video content?
- Should the system support personalized recommendations for users? If yes, what factors should be considered in generating recommendations?
- Are there any geographic constraints or internationalization requirements - to consider?
- What is the expected availability and fault tolerance level for the Netflix system?
- What are the security requirements and measures that need to be implemented in the system?
- Should the system support multiple devices and platforms? If yes, which ones?
- Are there any budget or cost considerations that should be taken into account during the system design?
- Are there any existing systems or third-party services that the Netflix system should integrate with?
- What are the expected growth rate and scalability requirements for the system?
- Should the system support concurrent streaming for multiple users on the same account?
- Should any content licensing or copyright restrictions be considered?
- How will the system handle user authentication and authorization?
- Should the system provide real-time analytics and monitoring capabilities?

- How will the system handle user feedback, reviews, and ratings?
- Are there any constraints or limitations on the infrastructure or resources that can be used?
- Is there any specific timeline or deadline for implementing the Netflix system?

Netflix High-LevelTechnologies Technologies

Let's see a simplified design for the Netflix system that you can use for a systems design interview:

User Interface (UI): Cloud services like AWS or Azure can host the user interface components of Netflix. For example, the front-end web application can be hosted on AWS Elastic Beanstalk or Azure App Service. Therefore, this allows for easy scalability, automatic load balancing, and simplified deployment of the UI across different devices and platforms.

Content Delivery System: To optimize content delivery, Netflix can leverage a global content delivery network (CDN) offered by cloud providers. AWS CloudFront or Azure CDN can be used to distribute and cache content across edge locations worldwide. Therefore, this ensures that content is delivered from the nearest edge server to the user, reducing latency and improving streaming performance.

Video Encoding and Transcoding: We can use Cloud-based encoding and transcoding services for efficient video processing. AWS Elemental MediaConvert or Azure Media Services can handle the encoding and transcoding video content into various formats and bitrates. These services offer scalability, cost-effectiveness, and support for adaptive streaming protocols like HLS or DASH.

Recommendation Engine: We can use Cloud providers' machine learning services for the recommendation engine. AWS offers Amazon Personalize, a fully managed service that provides personalized recommendations based on user data. Azure provides Azure Personalizer, which offers similar capabilities. These services allow training machine learning models on vast amounts of data and generating personalized recommendations at scale.

By leveraging cloud services for these components, Netflix can benefit from scalability, reliability, and cost-efficiency. Cloud providers offer managed services that handle the underlying infrastructure, allowing the Netflix engineering team to focus on application logic and user experience. Additionally, cloud services offer APIs and integration capabilities that enable seamless integration with other parts of the system.

Design Netflix DB Model

Designing a database model for a complex system like Netflix requires careful consideration of the data requirements and relationships. Here's a simplified example of a database model for Netflix:

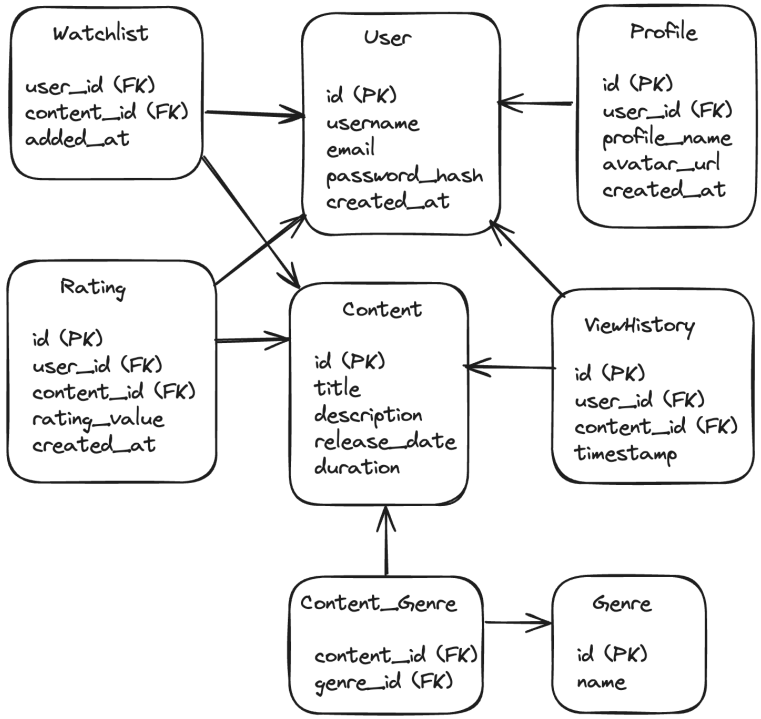


Figure 18. Netflix DB Design

General Technologies for Netflix Systems Design

Cloud Technologies:

Leverage cloud platforms like AWS or Azure for scalability and flexibility. Utilize AWS EC2 for scalable computing power. Benefit from AWS S3 for object storage. Explore serverless computing with AWS Lambda.

Database:

Combine relational databases (e.g., PostgreSQL, MySQL) and NoSQL databases (e.g., Cassandra, MongoDB). Relational databases for structured data, NoSQL databases for unstructured or semi-structured data.

Strategies:

Implement distributed caching with Redis or Memcached for efficient data retrieval and reduced latency. Employ a microservices architecture for modularity and scalability. Use service discovery mechanisms like Netflix Eureka or HashiCorp Consul for inter-service communication.

Java Technologies to Develop Netflix

Netflix uses several Java technologies in its development. Here are some key Java technologies commonly used by Netflix:

Java Programming Language: The core programming language used by Netflix for developing various platform components.

Spring Framework: Netflix heavily relies on the Spring framework for building scalable and robust Java applications. Spring provides features like dependency injection, inversion of control, and modular application development.

Netflix Open Source Software (OSS) projects: Netflix has developed and open-sourced several Java libraries and frameworks that developers widely use in the industry. Some notable examples are:

Netflix Hystrix: A library for implementing fault tolerance and resilience patterns, such as circuit breakers and bulkheads.

Netflix Ribbon: A client-side load balancing library that enables efficient and dynamic routing of requests between microservices.

Netflix Eureka: A service discovery framework that allows services to locate and communicate with each other in a distributed system.

Netflix Zuul: A gateway service that provides routing, filtering, and load-balancing capabilities for API requests.

Apache Kafka: Netflix uses Apache Kafka, a distributed event streaming platform, to handle large-scale data processing and real-time streaming of events within their architecture.

Apache Cassandra: Netflix employs Apache Cassandra, a highly scalable and distributed NoSQL database, to store and manage large volumes of data efficiently.

JUnit and Mockito: These testing frameworks are commonly used by Netflix developers to ensure the quality and reliability of their Java code.

Summary

To design Netflix in a Systems Design interview is not an easy task, we need to be objective and design the system in the simplest way as possible.

We need to assume the Netflix design is simple with specific questions. We must know how to design the DB model and what cloud technologies we will use.

Let's recap some of the core concepts we have to keep in mind to design a robust system like Netflix:

- Designing Netflix in a Systems Design interview requires focusing on the most important features within a limited time frame.
- Core concepts to consider when designing Netflix include scalability, high availability, CDN utilization, microservices architecture, service discovery, caching, fault tolerance, data partitioning, analytics and monitoring, security, and recommendation algorithms.

- The simplified design for the Netflix system includes the User Interface (UI), Content Delivery System, Video Encoding and Transcoding, and Recommendation Engine.
- Technologies to use in the design include cloud services for hosting the UI, CDN for content delivery, cloud-based encoding and transcoding services, and machine learning services for recommendation engines.
- The database model for Netflix involves a combination of relational and NoSQL databases.
- General technologies for Netflix systems design include cloud technologies, databases, distributed caching, microservices architecture, and service discovery mechanisms.
- Key Java technologies used by Netflix include the Java programming language, Spring Framework, Netflix OSS projects (Hystrix, Ribbon, Eureka, Zuul), Apache Kafka, Apache Cassandra, JUnit, and Mockito.

Let's recap important questions to ask:

- What are the essential features required for the system?
- What is the expected user base and traffic volume?
- What are the performance and latency requirements?
- Are personalized recommendations needed?
- Are there any geographic constraints to consider?
- What are the availability and fault tolerance requirements?
- What are the security requirements?
- Which devices and platforms should be supported?
- Are there any budget and cost considerations?
- Is integration with existing systems or third-party services necessary?
- What are the expected growth rate and scalability requirements?
- Will concurrent streaming be supported?
- Are there any content licensing and copyright restrictions?

- How should user authentication and authorization be handled?
- Is real-time analytics and monitoring required?
- How will user feedback and ratings be managed?
- Are there any infrastructure and resource constraints?
- Are there specific timeline or deadline considerations?

28 Next Steps

Now that you know the basics of how a Systems Design interview works, you need to practice it with a friend. This interview is a tricky one because designing a system in 45 minutes is very hard.

However, if we use the right strategy and design a simple system, the interview will be much easier.

Also, keep in mind that every technology you say during the interview, the interviewer might ask you why you are using this specific technology. So, everytime you mention a technology, make sure it's the technology you know the most.

Your main goal though is to create a reliable system that will work with the given requirements.

Recap of Key Concepts

In this book we explored the main concepts of Systems Design so we are able to design a Systems on the spot. Now, let's recap the main concepts we explored:

- Fundamentals of Availability: The principles and practices that ensure a system or service is consistently accessible and operational for users.
- Client-Server Model: A computing architecture where a client device requests services or resources from a server, which provides the requested functionality or data.
- Protocols: Rules and conventions that govern communication between different systems, specifying how data is transmitted, received, and interpreted.
- Effective API Design: Designing application programming interfaces (APIs) that are intuitive, efficient, and easy to use, promoting developer productivity and enabling seamless integration between software components.
- Storage Databases: Systems that organize and store structured data, providing efficient data retrieval, storage, and management capabilities.
- Proxy: An intermediary server that acts as a gateway between clients and other servers, facilitating various functions such as caching, load balancing, and security.
- Latency and throughput: Latency refers to the delay between a request and a response, while throughput measures the amount of data processed within a given time period.
- Load Balancer: A device or software that distributes network traffic across multiple servers or resources, optimizing resource utilization and enhancing system scalability and availability.

- Hashing: The process of converting data into a fixed-length value or key using a hashing algorithm, commonly used for data indexing, lookup, and ensuring data integrity.
- Cache: A high-speed storage component that stores frequently accessed data, reducing the need to fetch data from slower storage systems and improving system performance.
- Specific Storage: Tailored storage solutions designed to meet specific requirements, such as specialized databases for handling specific data types or use cases.
- Database Replica and Sharding: Replication involves creating multiple copies of a database for redundancy and fault tolerance, while sharding involves partitioning a database into smaller, more manageable pieces to distribute the workload.
- Leader election: The process by which a distributed system selects a leader or coordinator node to provide centralized control and coordination among the participating nodes.
- Peer-to-peer: A decentralized network architecture where participating nodes or devices interact directly with each other, sharing resources and responsibilities without the need for a central server.
- Periodic Polling and Streaming: Mechanisms for data retrieval in which periodic polling involves regularly querying a data source for updates, while streaming involves continuous data delivery in real-time.
- Static and Dynamic Configuration: Static configuration involves predefined settings that remain unchanged until modified, while dynamic configuration allows for runtime modifications and adjustments to system behavior.
- Rate Limiting: Imposing restrictions on the number of requests or data transmitted within a specific timeframe to prevent abuse, ensure fair resource allocation, and maintain system stability.
- Logging and Monitoring: Practices for recording and analyzing system events, errors, and performance metrics to diagnose issues, track system health, and support troubleshooting.

- Message Broker: A middleware component that facilitates communication and coordination between distributed systems or components by enabling asynchronous message passing.
- Map-Reduce: A programming model and data processing technique for distributed computing, where large data sets are divided, processed in parallel, and then combined to produce a final result.
- Security with HTTPS: The use of the Hypertext Transfer Protocol Secure (HTTPS) to encrypt data transmitted between a client and server, ensuring confidentiality and integrity of the communication.

Having this general view about Systems Design is the key to succeed in the interview.

Do Interviews - Practice, Practice, Practice

A best way to practice those interviews is by doing the interviews. Of course, it might be frustrating to practice that in a real interview because being rejected sucks. But we need to know how to deal with rejection and understand that there is always a reason.

The problem is not to do the interview and be rejected, the problem is to not do the interview because of the fear of bein rejected. Doing the interview is an effective way learn how the game works in practice.

We also have to remember that by doing those interviews and failing on them doesn't mean we are bad developers. It means intead that we don't know how the game for this interview works exactly. That's why we need to study, practice until we pass on those interviews.

Join Communities and Networks

You can find a developer at work who wants to get better on interviews. You can offer to interview your coworker and then you can ask your coworker to interview you. That's a great practice that will help you both to grow in your careers.

I also recommend you to join Java communities so you can learn from other experienced Java developers.

Also, because you finished the book, I will give you access to the Java Challengers Discord community where you can ask developers to interview you: https://discord.gg/MV2BTvav

Don't forget the reciprocity principle, offer developers to interview them, then, ask them to interview you also. You help them and they will help you.

Diversify Your Knowledge

It's not enough to learn something as complex as Systems Design in only one book. My suggestion is to diversify your knowledge and keep learning on different resources such as videos, tutorials, documentations, technologies.

You will have to continue your journey being curious, learning more about cloud components and most importantly, to really absorb those concepts you can share your knowledge about specific technologies or create your own system using those concepts.

Talking about Systems Design with other developers is also great to connect the dots.

Seek Feedback

We must fight our egos to learn how to take feedback. If we don't accept feedback we stop growing. We must train ourselves to be able to understand where we must improve so we can work on those points.

At the same time, we have to be careful to not take destructive feedback, you can easily identify that when you have a feedback with no suggestions of action you can take to improve that.

But usually, if you are getting feedback from someone you trust, it will be a good feedback. You have to be careful to not take the feedback emotionally, instead, take it as a gold to help you improve and be better.

Take the Challenger Developer Course

Getting a new job is not just about Systems Design, it's also about having a strong mindset so you are disciplined to take action in a strategic way.

By applying some techniques I will show you in the Challenger Developer course you might even skip the interview process.

How can you even skip the interview process? By sharing your knowledge someway! When you share your knowledge, other people will trust you can solve their problems.

You will also gain another level of clarity of what technologies you must study and how to stay motivated for your next steps in your career.

Learn more about the course in the following page: https://javachallengers.com/challenger-developer-course-algorithms-book

Conclusion

As I mentioned before, an important principle for you to take for life is to **never stop learning**. After reading this book, adopt a routine to learn constantly. That will be a powerful key for your growth because the more we learn, more our knowledge coumpounds.

Darren Hardy said:

Compound interest is the eighth wonder of the world. He who understands it, earns it… he who doesn't… pays it.

This concept of compounding is very true for our knowledge as software engineers. The more we learn, the easier it is to learn new things. Keep that in mind and reserve at least 1 hour every day to learn something new.

I hope you got great value from the book and don't forget to join the Java Challengers community on Discord! I will mention the link again to make it easy for you: https://discord.gg/MV2BTvav

As always, break your limits!