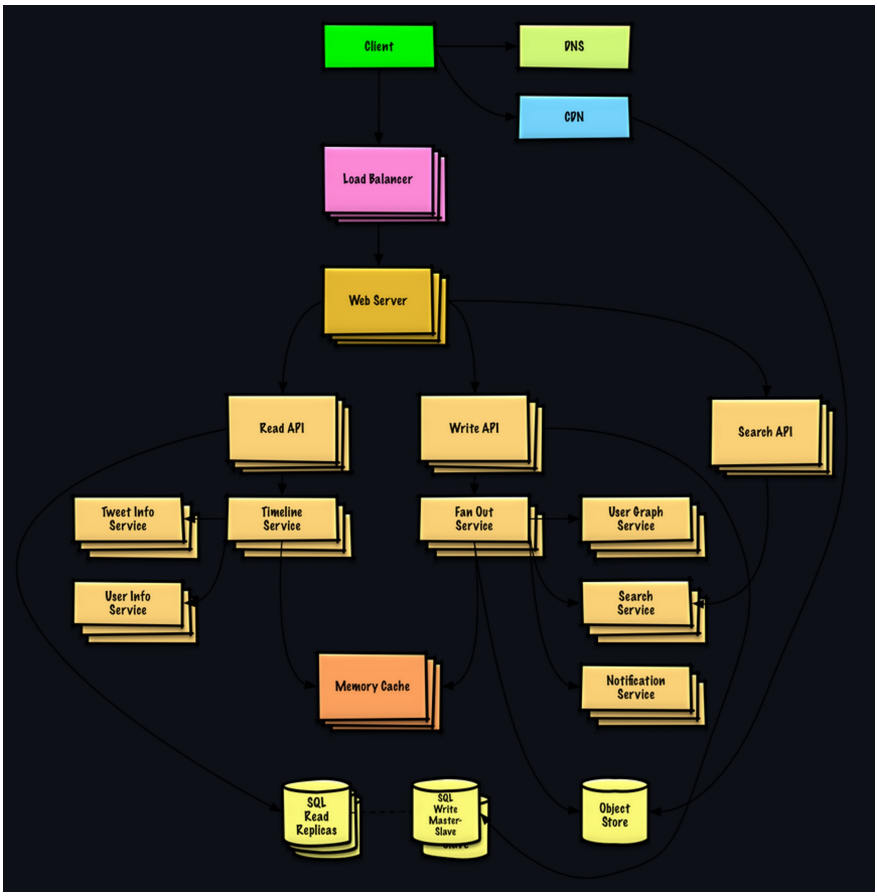
0 - Index

viernes, 26 de abril de 2024 13:41

https://github.com/donnemartin/system-design-primer?tab=readme-ov-file#index-of-system-design-topics



Desde <https://github.com/donnemartin/system-design-primer?tab=readme-ov-file#index-of-system-design-topics>

1 - Summary

viernes, 26 de abril de 2024 9:30

Guide: https://github.com/donnemartin/system-design-primer?tab=readme-ov-file#system-design-topics-start-here

Topics covered:

- Vertical scaling
- Horizontal scaling
- Caching
- Load balancing
- Database replication
- Database partitioning

1. VPS:

VPS stands for Virtual Private Server. It's a virtualized server that acts as an independent physical server but is actually hosted on a larger physical server along with multiple other virtual servers. Each VPS runs its own operating system (OS) instance and has dedicated resources such as CPU, RAM, storage (Disk), and network connectivity.
Overall, VPS hosting provides a flexible and scalable solution for hosting websites, applications, and other online services, offering the benefits of dedicated server resources without the associated cost and complexity.

Types of resources that are allocated:

CPU:

- Cores, L2 cache,…

Disk:

- PATA, SATA, SAS (faster rpms "disks literally spin", which allow for faster read/write operations),…
- SSD (have no disks which allow for even faster read/write operations, can't handle as much storage as disks nowadays, at a higher cost)
- RAID

RAM
…

Here's a breakdown of some key concepts related to VPS:

Vendor: Company that hosts the virtualization services.

Virtualization: VPS is made possible through virtualization technology, which allows multiple virtual servers to coexist on the same physical hardware. Virtualization software, such as VMware, KVM, or Hyper-Visor, creates and manages these virtualized environments.

Isolation: Each VPS is isolated from other VPS instances on the same physical server. This means that activities and configurations on one VPS do not affect others, providing a level of security and privacy similar to that of a physical server. (You can have privacy from everyone but the company who hosts the VPS itself, the only way to get full privacy is to operate your own servers.).

Resource allocation: Resources such as CPU, RAM, and storage are allocated to each VPS based on predefined specifications. This allows for customizable configurations tailored to the needs of different applications and workloads. As a server can have multiple CPUs which at the same time can have multiple 'cores' (cores are like brains so more than one core means we can have multiple processes running in parallel at the same time), vendors allocate these resources also based on the need of every server at every specific time, allowing an efficient split of the resources of the physical server into different VPSs.

Cost-effectiveness: VPS hosting is often more cost-effective than dedicated server hosting because multiple VPS instances can share the cost of the underlying physical hardware. This makes it an attractive option for individuals and businesses looking for affordable hosting solutions.
Scalability: VPS hosting offers scalability by allowing users to easily adjust resource allocations as their needs change. This flexibility makes it suitable for both small-scale applications and large-scale deployments.

Control and management: Users have administrative access to their VPS instances, giving them full control over the operating system, software installations, and configurations. They can also manage their VPS through a web-based control panel provided by the hosting provider.

2. Vertical Scaling:

So imagine the demand of our services blows overnight, what can we do now?

There is the option of using Vertical Scaling.

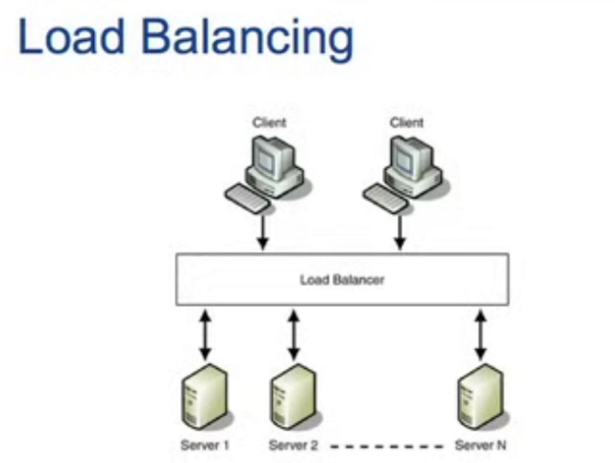
Vertical scaling, also known as scaling up, involves increasing the capacity of a single server by adding more resources such as CPU, memory (RAM), or storage (Disk). It typically involves upgrading the hardware components of the server to handle increased workload or performance requirements.
There is a ceiling though, there is a certain point you can reach with vertical scaling, either you are gona exhaust your finance resources on building a better server or your gonna reach the state of the art level at the market.

Constraints: Scale limitations based on the limit on how much you can improve a physical server.

3. Horizontal Scaling:

Horizontal scaling, also known as scaling out, involves increasing the capacity of a system by adding more machines or instances in a distributed manner. Instead of upgrading a single server, horizontal scaling distributes the workload across multiple servers, allowing the system to handle increased traffic or workload by adding more computational power in parallel. Allows to scale using cheaper servers.

To distribute the traffic into multiple servers, a Load Balancer is used:



3.1 Load Balancer:

A load balancer distributes incoming network traffic across multiple servers or computing resources in a balanced manner, ensuring optimal utilization of resources and preventing any single server from being overwhelmed.
In addition to distributing network traffic, a load balancer often acts as a single entry point for clients and manages IP addresses by providing a consistent IP address or domain name for clients to access the distributed resources. This simplifies the client-side interaction and allows the load balancer to dynamically route traffic to healthy servers, even if their IP addresses change or if new servers are added to the pool. It also allows security that clients cannot know the Private IP addresses fo those servers so they cannot be accessed directly.

LOAD BALANCING TECHNIQUES:

- An efficient technique to not waste resources is to have dedicated servers. For example if we have a web site service we can have a dedicated server for the images the website contains. We can do that by just uisng different URLs like www.images.com, www.gifs.com,.. Etc. Host names or URLs are translated into IPs using a DNS server. So one way could be to consider a Load Balancer as a fancy DNS server.

- Another way a Load Balancer can balance load, is by pointing for the first server to the first request it gets, the second request would go to the 2 server and so one, till the pool of servers end and it starts with the first server again. This technique is called **Round Robin**. The constraints of this techniques is that one server out of bad luck could get all the hard requests while the rest receive simple ones, overloading such server. Caching could also contribute to servers being overloaded. Therefore, there is a need of taking server load into account when distributing the requests.

- Other techniques could be a combination of previous mentioned ones with the addition of taking server load into account.

HW Load balancers are overly expensive so usually SW solutions are prefeared.

STICKY SESSIONS:

If the load balancer distributes the requests for the same User towards multiple servers, this rises the need to store Sessions.
Sessions are a way for web servers to remember user interactions. They use a unique session ID to track users. Session data can be stored in-memory, in a database, as files, or in external storage systems like Redis (persistant cache) or Memcached, so Cache is also a way of storing sessions. If the Session is not stored, this could cause, for example, the user having to log in again everytime their request goes to a different server. If we want to store Sessions in the load balancer, even though we would solve the issue of Shared States, we would sacrifice some redundancy (reability and fault tolerance) as we would be introducing a new problem, what if that server dies? All session info would be lost. To solve this redundancy flaw inside the same server RAID could be used.

Another solution to handle Sessions, would be to have **Shared Storage** across all servers. Shared storage can come in a bunch of different ways.

- FC (Fire channel) is a type of file server that can provide fast shared storage but its expensive.
- iSCSI uses IP protocols which use ethernet cables to exchange data between servers. It is a cheaper way usually used with single servers.
- MySQL database that could be used to store Cookies and metadata.

This solutions still require a server that cannot fail, so usually Shared data is also replicated across different servers to improve redundancy.

Cookies are also used for this purpose. We cannot store everything on cookies because of privacy violations (every user for the laptop could check what you are planning to buy, etc). Cookies also need to be small. Instead they store a big "random" number that the **Load balancer** will use to remember to which server their requests should go. This way we avoid revealing servers IPs and can maintain privacy.

3.2 Caching:

Caching is the process of storing frequently accessed data or computed results in a temporary storage location, known as a cache, to expedite subsequent requests for the same data. Caches act as a buffering layer between applications and data storage, which is usually stored in RAM.
When a request is made, the system first checks the cache to see if the data is already available. If it is, the data can be retrieved from the cache quickly, without needing to perform expensive computations or accessing slower storage systems like databases or remote servers. This helps improve performance, reduce latency, and alleviate load on backend systems. Caching is commonly used in various computing systems, including web servers, databases, and applications, to enhance overall performance and user experience. Examples of objects suitable for caching: Examples include user sessions, fully rendered blog articles, activity streams, and user-friend relationships.

DNS caching specifically refers to the temporary storage of resolved domain name information by DNS servers. When a DNS server receives a request to resolve a domain name (e.g., www.example.com) to an IP address, it typically performs a lookup to find the corresponding IP address. Once the IP address is obtained, it is stored in the server's cache for a certain period of time, known as the Time to Live (TTL).
This means that the same user traffic will be redirected to the same server, reason why load balancing techniques like Round Robin could cause one server to be overloaded if the requests of their recurrent users are too big.

DNS servers usually asign a TTL (time to live) to their answers stored in cache, but it can still mean that for the same day same user will be accessing the same server.

- **MySQL query Cache** is used to store cache in a mySQL database server. This is called **file-based caching**. This is done with using indexing so we can find data by their index. There are different storage engines used for caching which basically automatically compress cached data to be able to be stored using less space of a db.

- **Memcached**: Is a **in-memory cache** that can be placed in any server that allows to store memory in RAM (Random Access Memory). This solution is faster than using a database for retreiving cached data. To remove data, cached data has a TTL (time to live) and if more space is needed, data is removed using a FIFO (First in first out) approach. This could be used for example by facebook to avoid regenerating your profile by queries to their databases if it is not expected to change very often.

- **Redis**: Redis is an open-source, **in-memory** data structure store/ cache, that can be used as a database, cache, and message broker. It is known for its high performance, scalability, and versatility. Redis stores data primarily in RAM, which allows for extremely fast data access and retrieval. It supports various data structures such as strings, hashes, lists, sets, and sorted sets, making it suitable for a wide range of use cases including caching, session storage, real-time analytics, and message queuing. Is usually prefeared over memcached.

Caching Patterns:

- ***Cached Database Queries:*** This common pattern involves storing query results in the cache, using a hashed version of the query as the cache key. However, it presents challenges with expiration and cache invalidation.

- ***Cached Objects:*** This recommended pattern involves treating data as objects, assembling datasets from databases, and storing complete instances of classes or assembled datasets in the cache. This approach facilitates easy cache invalidation and enables asynchronous processing.

3.3 RAID

RAID, which stands for Redundant Array of Independent Disks, is a technology used to combine multiple physical disk drives into a single logical unit for improved performance, reliability, or both. RAID accomplishes this by distributing or replicating data across the drives in different ways, known as RAID levels. RAID configurations are commonly used in servers, storage systems, and enterprise environments to **improve data reliability, availability, and performance inside a Single server**. The choice of RAID level depends on factors such as performance requirements, fault tolerance, and cost considerations.

Some common RAID levels include:

- **RAID 0**: Striping - Data is evenly distributed across multiple drives to enhance performance. However, there is no redundancy, so if one drive fails, all data is lost. Good for performance since not all Read operations will go to the same drive but is bad for no redundancy, because data is not duplicated.

- **RAID 1**: Mirroring - Data is duplicated across multiple drives, providing redundancy. If one drive fails, data can still be accessed from the mirrored drive. This option is pricey because we are sacrificing one half of the total storage for keeping a replica of the data since data is not distributed across multiple drives, therefore out of 4 drives of 1Tb each, 2Tb are being used for redundancy.

- **RAID 5**: Striping with Parity - Data is striped across multiple drives, and parity information is distributed across all drives. This allows for both improved performance and redundancy. If one drive fails, data can be reconstructed using parity information stored on the remaining drives. This is more storage efficient, because if we have 4 drives, of 1Tb each, only 1Tb is used for redundancy so we still have 3Tb for storage and if one dies, all data can still be restored.

- **RAID 6**: Improves RAID 5 redundancy, with an extra drive it can provide fault tolerance of 2 drives having to die in order to lose all data.

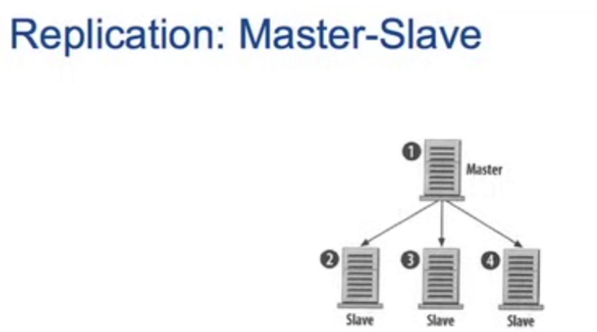
- **RAID 10 (RAID 1+0)**: Mirrored Stripes - Data is both striped across multiple drives for performance and mirrored for redundancy. This combines the benefits of RAID 0 and RAID 1, providing high performance and fault tolerance. But usually costs more money due to the need of more Drives.

3.4 Database Replication

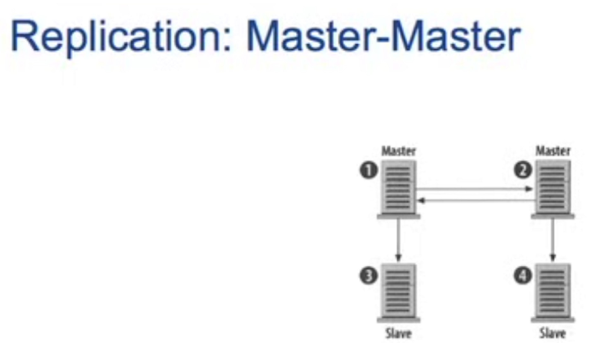
Replication is the process of creating and maintaining copies of data or resources in multiple locations, typically for the purpose of improving availability, fault tolerance, or performance. In the context of databases, replication involves copying data from one database to another in near real-time, ensuring that changes made to the primary database are propagated to all replica databases. Replication helps distribute workload, reduce latency, and provide redundancy in case of failures. It's commonly used in distributed systems, high availability setups, and disaster recovery scenarios.

REPLICATION TOPOLOGIES:

If Master is down, we can promote a slave to master to get some redundancy. Even though this will mean having some downtime while this promotion takes place.



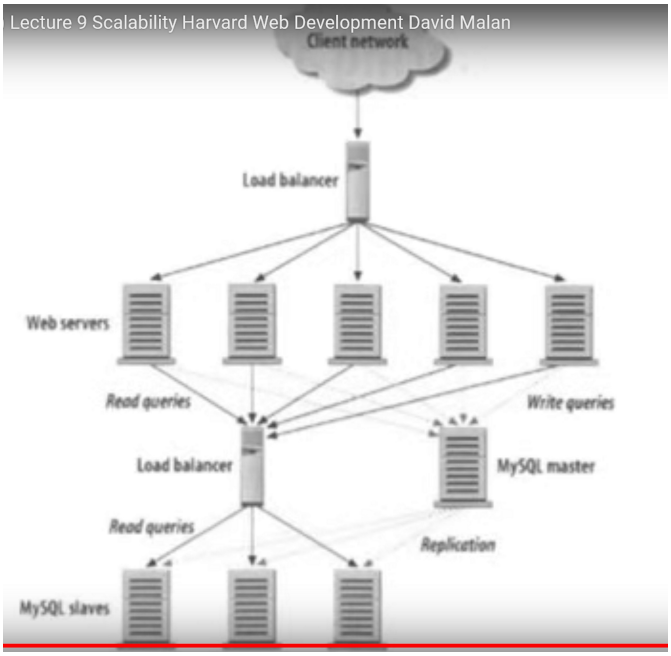
Queries that reach server 1 are replicated in server 2 and viceversa, so if one goes down, the other one can take the full master role.



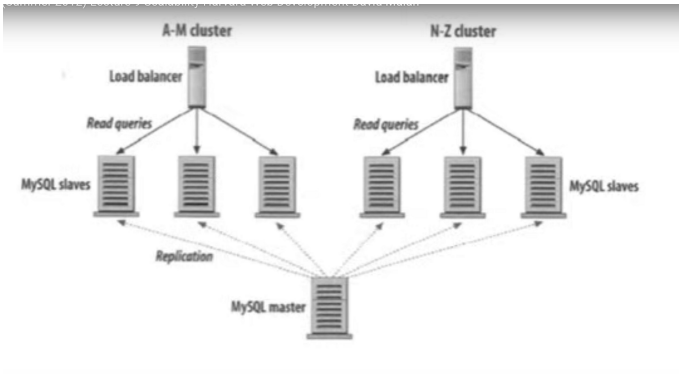
This is a multi tear architecture, web servers usually operate with this kind of architectures. Single point of failures in this architecture would be Load balancers and the mySQL server master.

Usually we have more than one load balancer that are in constant communication to check the status of the other one. If one fails the other one takes charge. This is done by an Active-Active mode if both sends and receive from the other. Or Active-Pasive, so if the pasive one stops receiving it automatically asumes the Active role taking over the other load balancer IP adress so all traffic gets redirected to it instead.

LOAD BALANCING + REPLICATION:

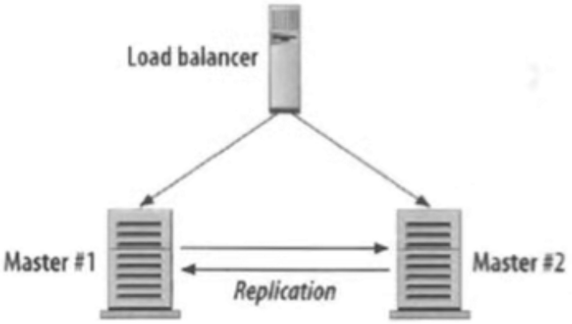


This approach would improve the load balancer failure issue.



Same concept can be applied to servers to avoid having just 1 that could fail, we can replicate the data to another master server and keep the comunication between them either Active-Active or Active-Passive in case one fails and the other one needs to take charge.

High availability:



3.5 Clones

So what about deployment ? How can we propagate changes into the application SW that runs inside our servers? This is done by using server clones once deploying our replicas.
When we talk about a server clone, it means creating a copy of a server setup that includes all its settings, software, and configurations. It's as if you're making a photocopy of a server so that you can easily reproduce it elsewhere. This helps maintain consistency and efficiency, especially in situations where you need multiple servers to work together seamlessly, like in a web service where users might interact with different servers at different times.

To create a server clone we first have to create an snapshot of a server with the updated changes in our data or application.
An AMI, or Amazon Machine Image, is essentially a snapshot of a virtual machine in the Amazon Web Services (AWS) environment. It's like taking a picture of a server setup, including the operating system, applications, and data. So, when we refer to creating a server clone using an AMI, it means capturing all the configurations and software of a server instance and using it as a blueprint to replicate identical server instances. In short, an AMI is a crucial part of creating and deploying server clones in AWS.

3.6 Database partitioning

Database partitioning divides large tables or indexes into smaller, more manageable parts. Each partition holds a subset of the data, improving performance, manageability, and scalability by focusing queries on relevant data, simplifying data management, and enabling horizontal scalability across multiple storage devices or servers.

3.7 Challanges of scaling databases

The scalability of your servers can lead to issues with database performance, especially with relational databases like MySQL. Relational databases are usually scaled vertically since using horizontal scaling which implies distributing the data across multiple servers, which becomes complex for structured data that relies on JOIN operations to retrieve data from multiple tables. As the dataset grows, performing joins can become computationally expensive and slow down query performance. Relational databases adhere to ACID (Atomicity, Consistency, Isolation, Durability) properties, which ensure data integrity and consistency. However, maintaining ACID compliance can impact scalability, especially in distributed environments.

In contrast, NoSQL databases like MongoDB or CouchDB are designed to address some of these scalability challenges. They offer flexible schema designs, horizontal scalability through sharding, and optimized data distribution models, making them better suited for handling large and growing datasets in distributed environments.

Two paths are presented to address the database scalability issues:

- Path #1 involves sticking with MySQL, implementing master-slave replication, and upgrading the master server with more RAM (scale the master vertically). However, this approach may become increasingly complex and costly over time.

- Path #2 suggests denormalizing the database from the beginning, avoiding joins in queries, and possibly switching to a NoSQL database like MongoDB or CouchDB. Joins would then be handled in the application code. This approach aims for scalability and simplicity but may also require implementing caching.

3.8 Asynchronism

Asynchronism is used to avoid the unwanted delays that users could experience in web services or web apps.
In general, there are two ways / paradigms asynchronism can be done.

First paradigm:

Let's stay in the former bakery picture. The first way of async processing is the **"bake the breads at night and sell them in the morning**" way. No waiting time at the cash register and a happy customer. Referring to a web app this means doing the time-consuming work in advance and serving the finished work with a low request time.

Very often this paradigm is used to turn dynamic content into static content. Pages of a website, maybe built with a massive framework or CMS, are pre-rendered and locally stored as static HTML files on every change. Often these computing tasks are done on a regular basis, maybe by a script which is called every hour by a cronjob. This pre-computing of overall general data can extremely improve websites and web apps and makes them very scalable and performant. Just imagine the scalability of your website if the script would upload these pre-rendered HTML pages to AWS S3 or Cloudfront or another Content Delivery Network! Your website would be super responsive and could handle millions of visitors per hour!

Second paradigm:

Back to the bakery. Unfortunately, sometimes customers has **special requests** like a birthday cake with "Happy Birthday, Steve!" on it. **The bakery can not foresee these kind of customer wishes, so it must start the task when the customer is in the bakery and tell him to come back at the next day**. Refering to a web service that means to handle tasks asynchronously.

Handling tasks asynchronously in response to user requests, such as computing-intensive tasks that take several minutes to finish. This involves sending tasks to a job queue, processing them with workers, and notifying users once the task is complete.

The workflow in asynchronous task handling goes as follows:

- Users initiate computing-intensive tasks on the website frontend, which are sent to a job queue.
- Workers constantly check the job queue for new tasks and process them and process them asynchronously.
- Once a task is completed, the frontend is notified, and users are informed about the task's completion.

Asynchronous processing is a programming technique where tasks are executed independently of the main program flow, allowing the program to continue running while the tasks are being completed in the background. In asynchronous processing, tasks are initiated and processed asynchronously, meaning that the program does not wait for each task to complete before moving on to the next one. This allows for improved efficiency and responsiveness, especially for tasks that may take a long time to complete, such as network requests, file I/O operations, or complex computations. Asynchronous processing is commonly used in web development, distributed systems, and concurrent programming to optimize resource utilization and improve overall performance.

2 - Trade-offs

viernes, 26 de abril de 2024 18:20

Performance vs Scalability:

A service is **scalable** if it results in increased **performance** in a manner proportional to resources added. Generally, increasing performance means serving more units of work, but it can also be to handle larger units of work, such as when datasets grow.

Another way to look at performance vs scalability:

- If you have a **performance** problem, your system is slow for a single user.
- If you have a **scalability** problem, your system is fast for a single user but slow under heavy load.

A service is said to be scalable if when we increase the resources in a system, it results in increased performance in a manner proportional to resources added. Increasing performance in general means serving more units of work, but it can also be to handle larger units of work, such as when datasets grow.
In distributed systems there are other reasons for adding resources to a system; for example to improve the reliability of the offered service. Introducing redundancy is an important first line of defense against failures. *An always-on service is said to be scalable if adding resources to facilitate redundancy does not result in a loss of performance*.

Performance: Performance focuses on the efficiency and speed of a system under specific conditions. It measures how well a system performs tasks and responds to requests with existing resources. A high-performance system can handle a given workload efficiently without excessive delay or resource consumption. In essence, performance optimization aims to maximize the speed and responsiveness of a system's operations.

Scalability: Scalability, on the other hand, addresses the system's ability to handle increased workload or data volume by adding resources in a proportional manner. It measures how well a system can grow and adapt to meet evolving demands. A scalable system can accommodate growing user bases, increased data volumes, or higher request rates without sacrificing performance. Scalability optimization focuses on designing systems that can scale seamlessly to support larger workloads without compromising performance.

Comparison: While performance optimization aims to enhance the efficiency of a system under current conditions, scalability optimization focuses on ensuring the system can grow and handle future demands effectively. Performance improvements aim to maximize the throughput and minimize response times for existing workloads, while scalability enhancements prepare the system to handle increased workloads without performance degradation. In essence, performance optimization seeks to improve the current state of the system, while scalability optimization aims to future-proof the system against growth and expansion.

Latency vs throughput

Latency is the time required to perform some action or to produce some result. Latency is measured in units of time -- hours, minutes, seconds, nanoseconds or clock periods.

Throughput is the number of such actions executed or results produced per unit of time. This is measured in units of whatever is being produced (cars, motorcycles, I/O samples, memory words, iterations) per unit of time. The term "memory bandwidth" is sometimes used to specify the throughput of memory systems.

Generally, you should aim for **maximal throughput** with **acceptable latency**.

A simple example
The following manufacturing example should clarify these two concepts:
An assembly line is manufacturing cars. It takes eight hours to manufacture a car and that the factory produces one hundred and twenty cars per day.
The latency is: 8 hours.
The throughput is: 120 cars / day or 5 cars / hour.

A design example
Now that these two concepts are clear, let us apply these concepts to a problem "closer to home."
A designer is given the task to create hardware for a communications device that has the following characteristics:

Clock frequency:	100MHz
Time available to perform the computation:	1000ns

Throughput of the device: 640 Mbits / second
Word width of each output: 64 bits

Let us translate these requirements into latency and throughput measurements that are more meaningful from the point of view of the hardware designer.

Latency: 1000 ns = 1000 ns * (1 s / 10^9 ns) * (100 * 10^6 clock periods/ 1s) = 10^11/10^9 = 100 clock periods.

Throughput = 640 Mbits / s = (640 * 10^6 bits/s) * (1 word / 64 bits) * (1 s / 100 * 10^6 clock periods) = 640 * 10^6 / 64 * 100 * 10^6 = 10 * 10 / 100 = 1 / 10 = 0.1 words / clock period.

The throughput could be read more conveniently as follows: "one word every 10 clock periods"
Latency expressed in clock periods, and throughput expressed in number of available clock cycles between words, are parameters that a designer can use to create the desired hardware according to the performance specifications.

Availability vs consistency

CAP theorem:

The CAP theorem, also known as Brewer's theorem, states that in a distributed system, it's impossible to simultaneously achieve all three of the following guarantees:

Consistency: Every read receives the most recent write or an error.

Availability: Every request receives a response, without guaranteeing that it contains the most recent write.

Partition tolerance: The system continues to operate despite network partitions (communication failures) that might cause messages to be lost or delayed between nodes.

In essence, the CAP theorem highlights the trade-offs that must be made when designing distributed systems. While it's possible to prioritize any two out of the three guarantees, achieving all three simultaneously is not feasible. Therefore, designers must carefully consider their system's requirements and choose the appropriate trade-offs to meet those requirements effectively.

Networks aren't reliable, so you'll need to support partition tolerance if the system is distributed and not centralized. You'll need to make a software tradeoff between consistency and availability.

A centralized system is one where control, decision-making, and data processing are concentrated in a single location or entity/server, resulting in a single point of control, data storage, and potential bottlenecks. In this case we would get A and C but not P.

CP - consistency and partition tolerance

Waiting for a response from the partitioned node might result in a timeout error. CP is a good choice if your business needs require atomic reads and writes.
In scenarios where data consistency is paramount, such as financial transactions or healthcare records, CP (Consistency and Partition tolerance) might be preferred over AP (Availability and Partition tolerance) to ensure data integrity even during network partitions.

AP - availability and partition tolerance

Responses return the most readily available version of the data available on any node, which might not be the latest. Writes might take some time to propagate when the partition is resolved.

AP is a good choice if the business needs to allow for eventual consistency or when the system needs to continue working despite external errors. Eventual consistency means that after a write, reads will eventually see it (typically within milliseconds). Data is replicated asynchronously.
This approach is seen in systems such as DNS and email. Eventual consistency works well in highly available systems.

Consistency patterns

With multiple copies of the same data, we are faced with options on how to synchronize them so clients have a consistent view of the data.

Weak consistency: After a write, reads may or may not see it. A best effort approach is taken. This approach is seen in systems such as memcached. Weak consistency works well in real time use cases such as VoIP, video chat, and realtime multiplayer games. For example, if you are on a phone call and lose reception for a few seconds, when you regain connection you do not hear what was spoken during connection loss.

Eventual consistency: After a write, reads will eventually see it (typically within milliseconds). Data is replicated asynchronously.
This approach is seen in systems such as DNS and email. Eventual consistency works well in highly available systems.

Strong consistency: After a write, reads will see it. Data is replicated synchronously. This approach is seen in file systems and RDBMSes (relational databases). Strong consistency works well in systems that need transactions. Transactions adhere to ACID. Usually centralized systems.

ACID for data transactions in relational databases:

ACID stands for Atomicity, Consistency, Isolation, and Durability. These are the four key properties that guarantee the reliability and consistency of transactions in a database system:

> **Atomicity**: Atomicity ensures that a transaction is treated as a single unit of work, either all of its operations are executed successfully, or none of them are. In other words, a transaction is "all or nothing".
>
> **Consistency**: Consistency ensures that a transaction transitions the database from one valid state to another valid state. It prevents transactions from leaving the database in an inconsistent state.
>
> **Isolation**: Isolation ensures that the execution of transactions concurrently does not result in interference or inconsistency. Each transaction appears to execute independently of other transactions, even if they are executed concurrently.
>
> **Durability**: Durability ensures that once a transaction is committed, its effects are permanent and survive system failures. The changes made by a committed transaction are stored in non-volatile memory (usually disk) and are not lost, even in the event of a system crash or power failure.

Avialability patterns

There are two complementary patterns to support high availability: **fail-over** and **replication**.

Fail-over

Active-passive : With active-passive fail-over, heartbeats are sent between the active and the passive server on standby. If the heartbeat is interrupted, the passive server takes over the active's IP address and resumes service. **Pasive server does not manage traffic while in stand-by.**

The length of downtime is determined by whether the passive server is already running in 'hot' standby or whether it needs to start up from 'cold' standby. Only the active server handles traffic.
Active-passive failover can also be referred to as master-slave failover.

Active-active: In active-active, **both servers are managing traffic,** spreading the load between them.
If the servers are public-facing, the DNS would need to know about the public IPs of both servers. If the servers are internal-facing, application logic would need to know about both servers.
Active-active failover can also be referred to as master-master failover.

Disadvantage(s): failover

- Fail-over adds more hardware and additional complexity.
- There is a potential for loss of data if the active system fails before any newly written data can be replicated to the passive.

Replication

Master-slave and master-master

This topic is further discussed in the Database section:

- Master-slave replication
- Master-master replication

Availability in numbers

Availability is often quantified by uptime (or downtime) as a percentage of time the service is available. Availability is generally measured in number of 9s--a service with 99.99% availability is described as having four 9s.

99.9% availability - three 9s

Duration	Acceptable downtime
Downtime per year	8h 45min 57s
Downtime per month	43m 49.7s
Downtime per week	10m 4.8s
Downtime per day	1m 26.4s

99.99% availability - four 9s

Duration	Acceptable downtime
Downtime per year	52min 35.7s
Downtime per month	4m 23s
Downtime per week	1m 5s
Downtime per day	8.6s

Availability in parallel vs in sequence

If a service consists of multiple components prone to failure, the service's overall availability depends on whether the components are in sequence or in parallel.

In sequence : Overall availability decreases when two components with availability < 100% are in sequence:

Availability (Total) = Availability (Foo) * Availability (Bar)

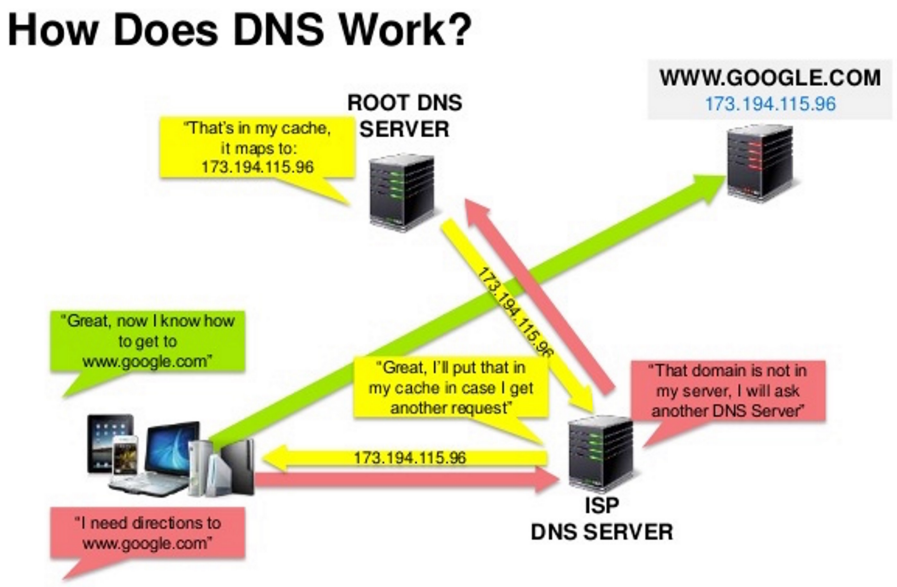
If both Foo and Bar each had 99.9% availability, their total availability in sequence would be 99.8%.

In parallel: Overall availability increases when two components with availability < 100% are in parallel:
Availability (Total) = 1 - (1 - Availability (Foo)) * (1 - Availability (Bar))
If both Foo and Bar each had 99.9% availability, their total availability in parallel would be 99.9999%.

3 - DNS

viernes, 26 de abril de 2024 18:56

A Domain Name System (DNS) translates a domain name such as www.example.com to an IP address.



DNS is hierarchical, with a few authoritative servers at the top level. Your router or ISP provides information about which DNS server(s) to contact when doing a lookup. Lower level DNS servers cache mappings, which could become stale due to DNS propagation delays. DNS results can also be cached by your browser or OS for a certain period of time, determined by the time to live (TTL).

- **NS record (name server)** - Specifies the DNS servers for your domain/subdomain.
- **MX record (mail exchange)** - Specifies the mail servers for accepting messages.
- **A record (address)** - Points a name to an IP address.
- **CNAME (canonical)** - Points a name to another name or CNAME (example.com to www.example.com) or to an A record.

Services such as CloudFlare and Amazon Route 53 provide managed DNS services. Some DNS services can route traffic through various methods:

- Weighted round robin
  - Prevent traffic from going to servers under maintenance
  - Balance between varying cluster sizes
  - A/B testing
- **Simple routing policy**
  - Directs traffic to a single resource serving a specific function for your domain.
  - Ideal for web servers hosting content for your website.
  - Configurable within a private hosted zone.
- **Geolocation routing policy**
  - Routes traffic based on the geographical location of users.
  - Directs users to the closest or most appropriate resource.
  - Configurable within a private hosted zone.
- **Latency routing policy**
  - Directs traffic to the region with the lowest latency.
  - Optimizes user experience by selecting the best-performing region.
  - Configurable within a private hosted zone.
- **Multivalue answer routing policy**
  - Responds to DNS queries with multiple healthy records selected randomly.
  - Enhances fault tolerance and availability by distributing load across resources.
  - Configurable within a private hosted zone.
- **Weighted routing policy**
  - Distributes traffic across multiple resources based on specified proportions.
  - Allows fine-grained control over resource utilization.
  - Configurable within a private hosted zone.

  In DNS, a hosted zone is a container that holds information about how you want to route traffic for a specific domain, including DNS records such as A, AAAA, CNAME, and others. A private hosted zone is a hosted zone that is associated with a specific Amazon Virtual Private Cloud (VPC) or a private network, and it's not publicly accessible from the internet.

4 - CDN

sábado, 27 de abril de 2024 19:47

A content delivery network (CDN) is a globally distributed network of proxy servers, serving content from locations closer to the user. Generally, static files such as HTML/CSS/JS, photos, and videos are served from CDN, although some CDNs such as Amazon's CloudFront support dynamic content. The site's DNS resolution will tell clients which server to contact.

Serving content from CDNs can significantly improve performance in two ways:

- Users receive content from data centers close to them
- Your servers do not have to serve requests that the CDN fulfills

Push CDNs

Push CDNs receive new content whenever changes occur on your server. You take full responsibility for providing content, uploading directly to the CDN and rewriting URLs to point to the CDN. You can configure when content expires and when it is updated. Content is uploaded only when it is new or changed, minimizing traffic, but maximizing storage.
Sites with a small amount of traffic or sites with content that isn't often updated work well with push CDNs. Content is placed on the CDNs once, instead of being re-pulled at regular intervals.

Pull CDNs

Pull CDNs grab new content from your server when the first user requests the content. You leave the content on your server and rewrite URLs to point to the CDN. This results in a slower request until the content is cached on the CDN.
A time-to-live (TTL) determines how long content is cached. Pull CDNs minimize storage space on the CDN, but can create redundant traffic if files expire and are pulled before they have actually changed.
Sites with heavy traffic work well with pull CDNs, as traffic is spread out more evenly with only recently-requested content remaining on the CDN.

Disadvantages of CDN

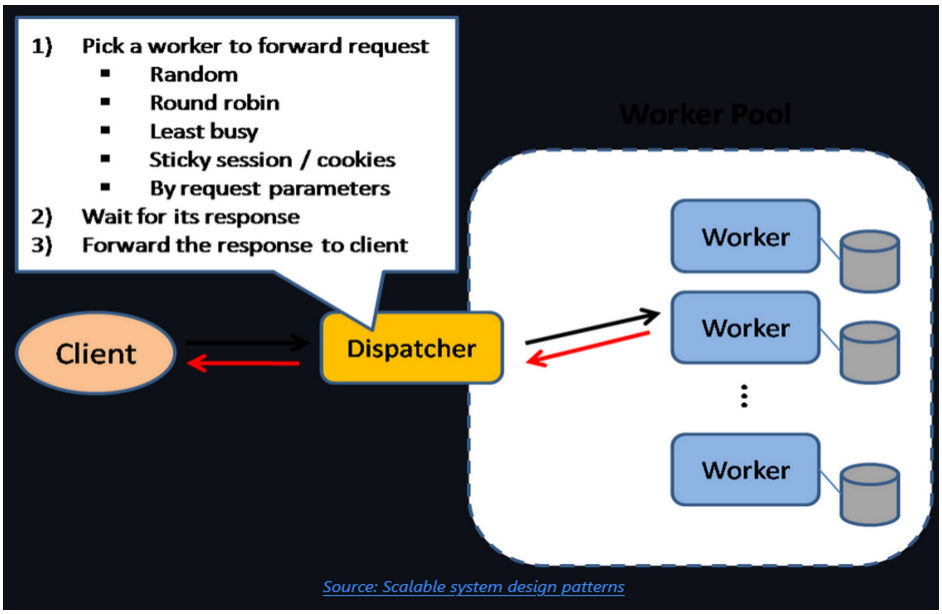
- CDN costs could be significant depending on traffic, although this should be weighed with additional costs you would incur not using a CDN.
- Content might be stale if it is updated before the TTL expires it.

CDNs require changing URLs for static content to point to the CDN.

5 - Load balancer

sábado, 27 de abril de 2024 21:28

Load balancers



Load balancers distribute incoming client requests to computing resources such as application servers and databases. In each case, the load balancer returns the response from the computing resource to the appropriate client. Load balancers are effective at:

- Preventing requests from going to unhealthy servers
- Preventing overloading resources
- Helping to eliminate a single point of failure

Load balancers can be implemented with hardware (expensive) or with software such as HAProxy.

Additional benefits include:

- **SSL termination** - Decrypt incoming requests and encrypt server responses so backend servers do not have to perform these potentially expensive operations
  - Removes the need to install X.509 certificates on each server (a security certificate). All trafic below the load balancer will be decrypted and all traffic on top of it (towards the internet) will be encrypted.
- **Session persistence** - Issue cookies and route a specific client's requests to same instance if the web apps do not keep track of sessions.

To protect against failures, it's common to set up multiple load balancers, either in active-passive or active-active mode.

Load balancers can route traffic based on various metrics, including:

- **Random**: Routes traffic randomly to available servers without considering their current load or capacity.
- **Least loaded**: Routes traffic to the server with the least current load or fewest active connections, aiming to evenly distribute the workload.
- **Session/cookies**: Routes traffic based on session information or cookies associated with client requests, ensuring that subsequent requests from the same client are directed to the same server. When using cookies for session persistence and encountering a loaded server, the load balancer must balance between maintaining session continuity and ensuring optimal performance by potentially redirecting the request to a less loaded server. Sessions typically allow for the restoration of user activities done in the previously used server which will introduce a delay when switching to a new server. Therefore there is a trade-off with performance, reason why users should be pointed to the same server when possible.
- **Round robin**: Routes traffic sequentially to each server in a rotation, ensuring that each server receives an equal share of requests over time.
- **Weighted round robin**: Similar to round robin, but assigns weights to servers based on their capacities or loads, distributing traffic proportionally to these weights.
- **Layer 4**:

Layer 4 load balancing operates at the network transport layer (Layer 4) of the OSI model, and typically does not involve the use of cookies or sessions. It routes traffic based on TCP or UDP protocol information, such as source and destination IP addresses and port numbers. This type of load balancing typically uses techniques like Network Address Translation (NAT) for traffic interception and connection persistence for maintaining session state. NAT is a method that rewrites the destination IP address of incoming packets to the IP address of a selected server in the backend pool, allowing the load balancer to route traffic to the appropriate server without the client being aware of the backend server's IP address. Layer 4 load balancers perform health checks on backend servers and use algorithms like round-robin or least connections for traffic distribution. Overall, Layer 4 load balancing provides scalability, fault tolerance, and high availability for distributed applications.

- **Layer 7**: Layer 7 load balancing, operating at the application layer of the OSI model, offers advanced capabilities for routing and distributing traffic based on application-specific criteria. Unlike lower-layer load balancing, which focuses on network-level information such as IP addresses and port numbers, Layer 7 load balancing allows load balancers to analyze and make routing decisions based on the actual content and context of application requests.
  For example, with Layer 7 load balancing, a load balancer can examine HTTP headers, URLs, cookies, or even the content of requests to determine how to route traffic. This enables load balancers to direct requests to specific servers based on factors such as content type, user session information, geographic location, or even specific application features.
  By understanding the application layer protocols and payloads, Layer 7 load balancers can optimize traffic distribution to improve performance, enhance user experience, and ensure efficient resource utilization across backend servers. Additionally, Layer 7 load balancers can support more sophisticated load balancing algorithms and provide features like SSL termination, content caching, and application-layer security capabilities.

Horizontal scaling

Load balancers can also help with horizontal scaling, improving performance and availability. Scaling out using commodity machines is more cost efficient and results in higher availability than scaling up a single server on more expensive hardware, called **Vertical Scaling**. It is also easier to hire for talent working on commodity hardware than it is for specialized enterprise systems.

Disadvantage(s): horizontal scaling

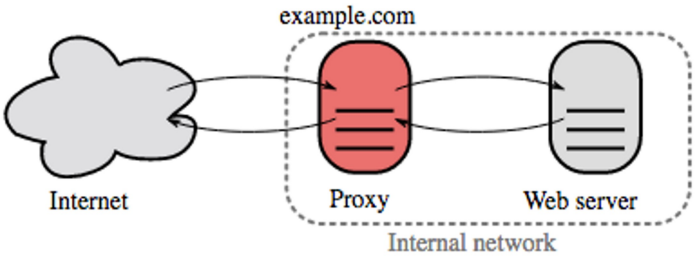
- Scaling horizontally introduces complexity and involves cloning servers
  - Servers should be stateless: they should not contain any user-related data like sessions or profile pictures
  - Sessions can be stored in a centralized data store such as a database (SQL, NoSQL) or a persistent cache (Redis, Memcached)
- Downstream servers such as caches and databases need to handle more simultaneous connections as upstream servers scale out

Disadvantage(s): load balancer

- The load balancer can become a performance bottleneck if it does not have enough resources or if it is not configured properly.
- Introducing a second load balancer to help eliminate a single point of failure results in increased complexity.
- A single load balancer is a single point of failure, configuring multiple load balancers further increases complexity.

6 - Reverse proxy (web server)

domingo, 28 de abril de 2024 12:07



A reverse proxy is a web server that centralizes internal services and provides unified interfaces to the public. Requests from clients are forwarded to a server that can fulfill it before the reverse proxy returns the server's response to the client. Additional benefits include:

- **Increased security** - Hide information about backend servers, blacklist IPs, limit number of connections per client
- **Increased scalability and flexibility** - Clients only see the reverse proxy's IP, allowing you to scale servers or change their configuration
- **SSL termination** - Decrypt incoming requests and encrypt server responses so backend servers do not have to perform these potentially expensive operations
    - Removes the need to install X.509 certificates on each server
- **Compression** - Compress server responses
- **Caching** - Return the response for cached requests
- **Static content** - Serve static content directly
    - HTML/CSS/JS
    - Photos
    - Videos
    - Etc

Load balancer vs reverse proxy

- Deploying a load balancer is useful when you have multiple servers. Often, load balancers route traffic to a set of servers serving the same function.
- Reverse proxies can be useful even with just one web server or application server, opening up the benefits described in the previous section.
- Solutions such as NGINX and HAProxy can support both layer 7 reverse proxying and load balancing.

In summary, while reverse proxies and load balancers can overlap in functionality, they serve different primary purposes. Reverse proxies focus on client-server communication and content delivery optimization, while load balancers prioritize resource distribution and high availability across backend servers. Depending on the specific requirements of an application or network architecture, organizations may choose to deploy one or both of these technologies to achieve their goals.

Reverse proxies would usually point towards a load balancer, especially in large-scale deployments which usually utilize both.

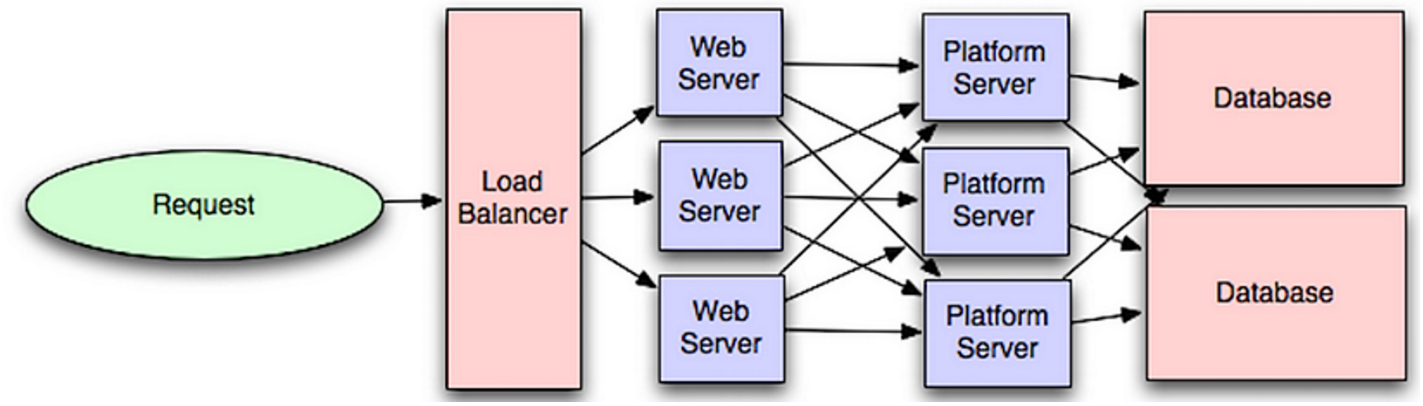
Disadvantage(s): reverse proxy

- Introducing a reverse proxy results in increased complexity.
- A single reverse proxy is a single point of failure, configuring multiple reverse proxies (ie a failover) further increases

complexity.

7 - Application Layer

domingo, 28 de abril de 2024 12:31



Separation of Web and Application Layers

- **Scalability and Configuration**: Separating the web layer from the application layer (also known as Platform layer) allows for independent scaling and configuration of both layers. This means that adding a new API may only require adding application servers without the need for additional web servers.
- **Single Responsibility Principle**: The single responsibility principle advocates for small and autonomous services that work together. Small teams with small services can plan more aggressively for rapid growth.
- **Asynchronism with Workers**: Workers in the application layer enable asynchronous task handling, improving performance and responsiveness by offloading time-consuming tasks to background processes. This approach enhances scalability, resource utilization, and fault tolerance.

Microservices Architecture

Modular Services: Microservices architecture consists of independently deployable, small, modular services. Each service runs a unique process and communicates through well-defined, lightweight mechanisms to serve specific business goals. For example, Pinterest could have microservices for user profile, follower, feed, search, photo upload, etc.

Service Discovery

Systems such as **Consul, Etcd, and Zookeeper** can help services find each other by keeping track of registered names, addresses, and ports. Health checks help verify service integrity and are often done using an HTTP endpoint. Both Consul and Etcd have a built in key-value store that can be useful for storing config values and other shared data.

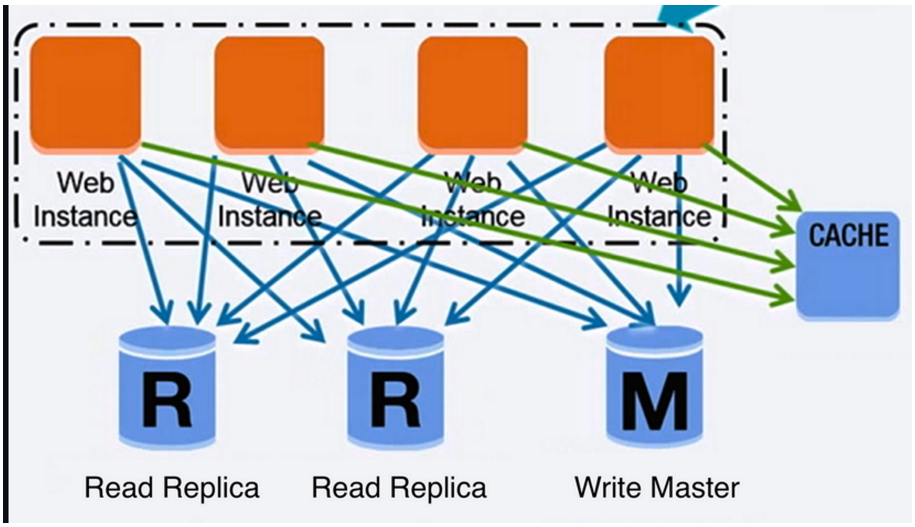
Disadvantages of the Application Layer

- **Architectural Complexity (vs Monolithic Systems):** Transitioning to an application layer with loosely coupled services requires a different approach architecturally, operationally, and in terms of processes, compared to a monolithic system. In a monolithic system, all components are tightly integrated into a single codebase and deployed as a single unit. This can lead to challenges in scalability, maintainability, and agility as the system grows in size and complexity.
- **Deployment and Operations Complexity**: Microservices can add complexity in terms of deployments and operations, necessitating robust processes and tools for monitoring,

managing and maintaining the system.

8 - Database

domingo, 28 de abril de 2024 12:44



Relational database management system (RDBMS)

A relational database like SQL is a collection of data items organized in tables.
ACID is a set of properties of relational database transactions.

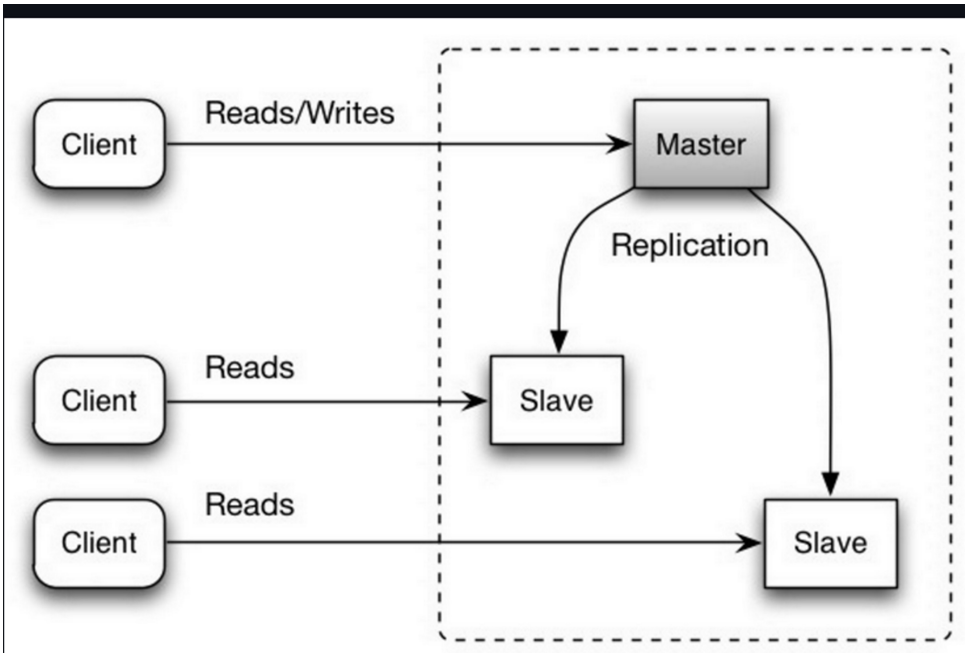
- **Atomicity** - Each transaction is all or nothing
- **Consistency** - Any transaction will bring the database from one valid state to another
- **Isolation** - Executing transactions concurrently has the same results as if the transactions were executed serially
- **Durability** - Once a transaction has been committed, it will remain so

There are many techniques to scale a relational database: **master-slave replication**, **master-master replication**, **federation**, **sharding**, **denormalization**, and **SQL tuning**.

Replication

Master-slave replication

The master serves reads and writes, replicating writes to one or more slaves, which serve only reads. Slaves can also replicate to additional slaves in a tree-like fashion. If the master goes offline, the system can continue to operate in read-only mode until a slave is promoted to a master or a new master is provisioned.

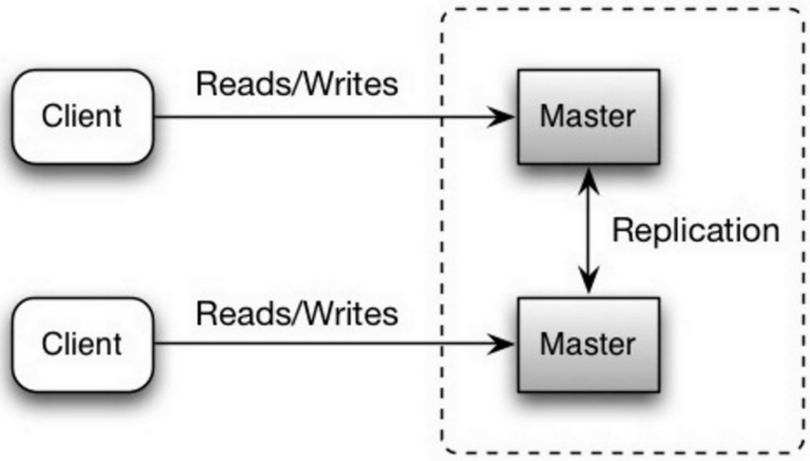


Disadvantage(s): master-slave replication

- Additional logic is needed to promote a slave to a master.

Master-master replication

Both masters serve reads and writes and coordinate with each other on writes. If either master goes down, the system can continue to operate with both reads and writes.



Disadvantage(s): master-master replication

- You'll need a load balancer or you'll need to make changes to your application logic to determine where to write.
- Most master-master systems are either loosely consistent (violating ACID) or have increased write latency due to synchronization.
- Conflict resolution comes more into play as more write nodes are added and as latency increases.

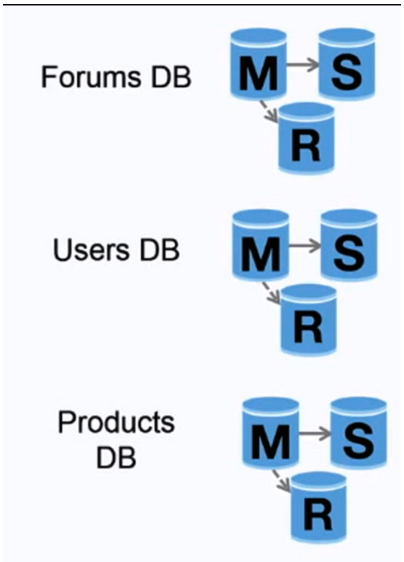
Disadvantage(s): replication

- There is a potential for loss of data if the master fails before any newly written data can be replicated to other nodes.
- Writes are replayed to the read replicas. If there are a lot of writes, the read replicas can get bogged down with replaying writes and can't do as many reads.
- The more read slaves, the more you have to replicate, which leads to greater replication lag.
- On some systems, writing to the master can spawn multiple threads to write in parallel, whereas read replicas only

support writing sequentially with a single thread.

- Replication adds more hardware and additional complexity.

Federation

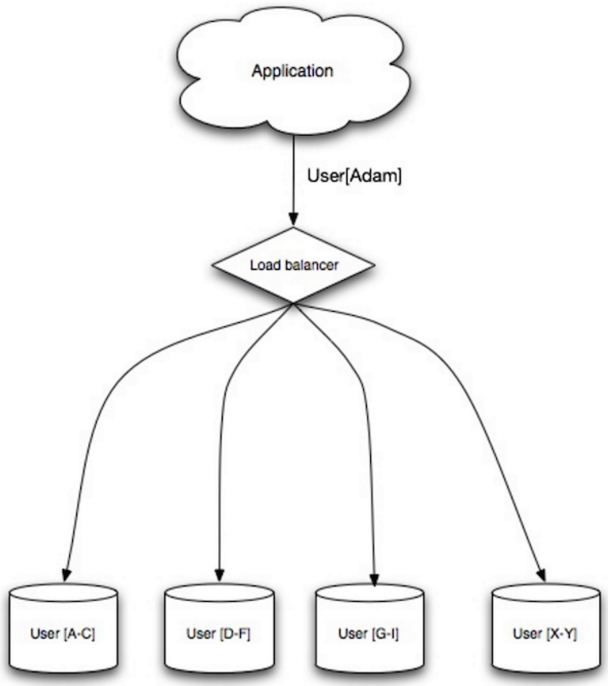


Federation (or functional partitioning) splits up databases by function. For example, instead of a single, monolithic database, you could have three databases: **forums**, **users**, and **products**, resulting in less read and write traffic to each database and therefore less replication lag. Smaller databases result in more data that can fit in memory, which in turn results in more cache hits due to improved cache locality. With no single central master serializing writes you can write in parallel, increasing throughput.

Disadvantage(s): federation

- Federation is not effective if your schema requires huge functions or tables.
- You'll need to update your application logic to determine which database to read and write.
- Joining data from two databases is more complex with a server link.
- Federation adds more hardware and additional complexity.

Sharding



Sharding distributes data across different databases such that each database can only manage a subset of the data. Taking a users database as an example, as the number of users increases, more shards are added to the cluster.

Similar to the advantages of federation, sharding results in less read and write traffic, less replication, and more cache hits. Index size is also reduced, which generally improves performance with faster queries. If one shard goes down, the other shards are still operational, although you'll want to add some form of replication to avoid data loss. Like federation, there is no single central master serializing writes, allowing you to write in parallel with increased throughput.

Common ways to shard a table of users is either through the user's last name initial or the user's geographic location.

Disadvantage(s): sharding

- You'll need to update your application logic to work with shards, which could result in complex SQL queries.
- Data distribution can become lopsided in a shard. For example, a set of power users on a shard could result in increased load to that shard compared to others.
    - Rebalancing adds additional complexity. A sharding function based on consistent hashing can reduce the amount of transferred data during rebalancing by ensuring that only a portion of the data needs to be moved when adding or removing shards.
- Joining data from multiple shards is more complex, requiring coordination and potentially impacting application performance.
- Sharding adds more hardware resources and introduces additional operational complexity.

Denormalization

Denormalization attempts to improve read performance at the expense of some write performance. Redundant copies of the data are written in multiple tables to avoid expensive joins (some data is duplicated). Some RDBMS such as PostgreSQL and Oracle support materialized views which handle the work of storing redundant information and keeping redundant copies consistent. Materialized views are database objects that contain the results of a precomputed query. Unlike regular views, which are virtual and execute the underlying query each time they are accessed, materialized views store the query results physically on disk. This allows for faster data retrieval and improved query performance, especially for complex and frequently accessed queries.

Once data becomes distributed with techniques such as federation and sharding, managing joins across data centers further increases complexity. Denormalization might circumvent the need for such complex joins.
In most systems, reads can heavily outnumber writes 100:1 or even 1000:1. A read resulting in a complex database join can be very expensive, spending a significant amount of time on disk operations.

Suitable for db with way more reads than writes.

Disadvantage(s): denormalization

- **Data Duplication**: Denormalization leads to the duplication of data across multiple tables, increasing storage requirements and potentially leading to inconsistencies if not properly managed.
- **Complexity of Synchronization**: Constraints are required to ensure that redundant copies of information stay synchronized. Managing these constraints adds complexity to the database design and increases the risk of data inconsistencies.
  Constraints in the context of databases refer to rules or conditions applied to data to ensure its integrity and consistency.
- **Performance Impact under Heavy Write Load**: In scenarios with heavy write operations, a denormalized database may perform worse than its normalized counterpart. The additional redundant data and synchronization constraints can lead to increased overhead and slower write performance.

SQL tuning

SQL tuning encompasses various techniques to optimize the performance of SQL queries and database operations. Here are some key considerations and optimizations:

Benchmarking and Profiling

- **Benchmarking**: Simulate high-load scenarios using tools like Apache Benchmark (ab) to identify performance bottlenecks.

- **Profiling**: Enable tools such as the slow query log to track and analyze performance issues.

Schema Optimization

- **Tighten Schema**: Optimize data storage and access efficiency.
  - MySQL dumps data to disk in contiguous blocks for fast access.
  - Use CHAR instead of VARCHAR for fixed-length fields to enable faster random access.
  - Use TEXT for large text blocks like blog posts, allowing boolean searches.
  - Utilize appropriate data types like INT for larger numbers and DECIMAL for currency to avoid representation errors.
  - Avoid storing large BLOBs directly; store pointers to their locations.
  - Set NOT NULL constraints where applicable to improve search performance.

Indexing

- **Use Good Indices**: Create indices on columns frequently used in queries (e.g., SELECT, GROUP BY, JOIN) for faster data retrieval.
  - Indices are typically represented as self-balancing B-trees, facilitating efficient data access.
  - Be aware that maintaining indices can impact write performance and consume additional memory.

Avoiding Expensive Joins

- **Denormalization**: Consider denormalizing data where performance requirements justify it, reducing the need for complex joins.

Partitioning Tables

- **Partitioning**: Split large tables into smaller partitions, particularly separating hot spots into separate tables to optimize memory usage and access times.

Query Cache Tuning

- **Optimize Query Cache**: Evaluate and fine-tune the query cache configuration, as it may sometimes lead to performance issues if not configured optimally.

SQL tuning involves a combination of these techniques tailored to the specific requirements and characteristics of the database and workload. Regular monitoring, benchmarking, and profiling are essential for identifying and addressing performance bottlenecks.

NoSQL (non-relational databases):

NoSQL is a collection of data items represented in a **key-value store**, **document store**, **wide column store**, or a **graph database**. Data is denormalized, and joins are generally done in the application code. Most NoSQL stores lack true ACID transactions and favor eventual consistency. It's the option to be used for unstructured data.

BASE is often used to describe the properties of NoSQL databases. In comparison with the CAP Theorem, BASE chooses availability over consistency.

- **Basically available** - the system guarantees availability.
- **Soft state** - the state of the system may change over time, even without input.
- **Eventual consistency** - the system will become consistent over a period of time, given that the system doesn't receive input during that period.

In addition to choosing between SQL or NoSQL, it is helpful to understand which type of NoSQL database best fits your use case(s). We'll review **key-value stores**, **document stores**, **wide column stores**, and **graph databases** in the next section.

Key-value store

Abstraction: hash table

A key-value store generally allows for O(1) reads and writes and is often backed by memory or SSD. Data stores can maintain keys in lexicographic order, allowing efficient retrieval of key ranges. Key-value stores can allow for storing of metadata with a value.
Key-value stores provide high performance and are often **used for simple data models or for rapidly-changing data, such as an in-memory cache layer**. Since they offer only a limited set of operations, complexity is shifted to the application layer if additional operations are needed.
A key-value store is the basis for more complex systems such as a document store, and in some cases, a graph database.

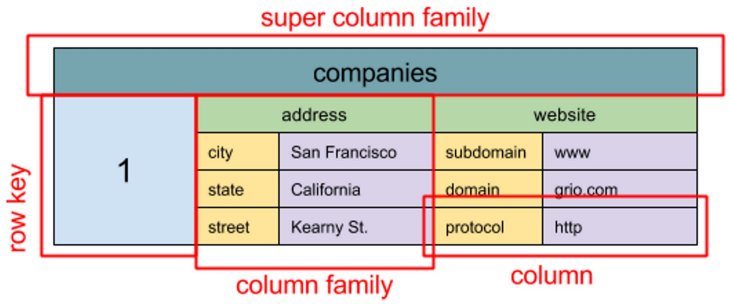
Document store

Abstraction: key-value store with documents stored as values

A document store is centered around documents (XML, JSON, binary, etc), where a document stores all information for a given object. Document stores provide APIs or a query language to query based on the internal structure of the document itself. *Note, many key-value stores include features for working with a value's metadata, blurring the lines between these two storage types.*
Based on the underlying implementation, documents are organized by collections, tags, metadata, or directories. Although documents can be organized or grouped together, documents may have fields that are completely different from each other.
Some document stores like MongoDB and CouchDB also provide a SQL-like language to perform complex queries. DynamoDB supports both key-values and documents.
Document stores provide high flexibility and are often **used for working with occasionally changing data**.

Wide column store

Abstraction: nested map ColumnFamily<RowKey, Columns<ColKey, Value, Timestamp>>



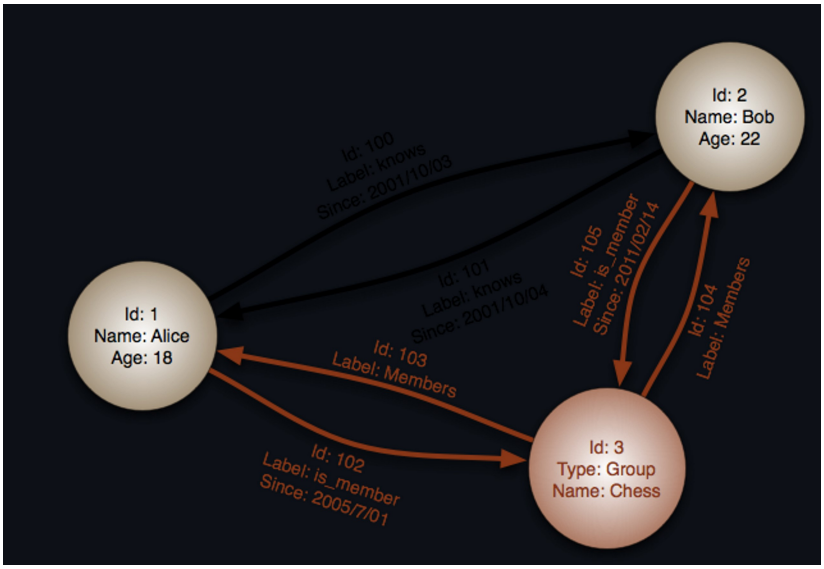
A wide column store's basic unit of data is a column (name/value pair). A column can be grouped in column families (analogous to a SQL table). Super column families further group column families. You can access each column independently with a row key, and columns with the same row key form a row. Each value contains a timestamp for versioning and for conflict resolution.

Google introduced Bigtable as the first wide column store, which influenced the open-source HBase often-used in the Hadoop ecosystem, and Cassandra from Facebook. Stores such as BigTable, HBase, and Cassandra maintain keys in lexicographic order, allowing efficient retrieval of selective key ranges.

Wide column stores offer high availability and high scalability. They are often used for very large data sets.

Graph database

Abstraction: graph



In a graph database, each node is a record and each arc is a relationship between two nodes. Graph databases are optimized to represent complex relationships with many foreign keys or many-to-many relationships.
Graphs databases offer high performance for data models with complex relationships, such as a social network. They are relatively new and are not yet widely-used; it might be more difficult to find development tools and resources. Many graphs can only be accessed with REST APIs.

Which database should we chose?

- **Document Database (e.g., MongoDB)**:
  - Choose for flexible schema and varied data structures.
  - Ideal for fast, iterative development and frequent schema changes.
  - Suitable for content management systems, blogging platforms, and e-commerce applications.
- **Key-Value Database (e.g., Redis)**:
  - Opt for simple data models with key-value pairs.
  - Effective for caching, session management, and real-time analytics.
  - Suitable for scenarios demanding high performance, scalability, and low-latency data access.
- **Graph Database (e.g., Neo4j)**:
  - Choose for complex relationships and network analysis.
  - Suitable for social networks, recommendation engines, and fraud detection systems.
  - Provides efficient traversal of interconnected data and supports graph algorithms for pattern detection.
- **Column-Family Database (e.g., Apache Cassandra)**:
  - Opt for large volumes of structured and semi-structured data.
  - Ideal for time-series data, IoT applications, and distributed analytics.
  - Offers high availability, fault tolerance, and linear scalability.

When should we choose SQL or NoSQL ?

- Reasons for **SQL**:
  - Structured data
  - Strict schema
  - Relational data
  - Need for complex joins
  - Transactions
  - Clear patterns for scaling
  - More established: developers, community, code, tools, etc
  - Lookups by index are very fast

- Reasons for **NoSQL**:
  - Semi-structured data

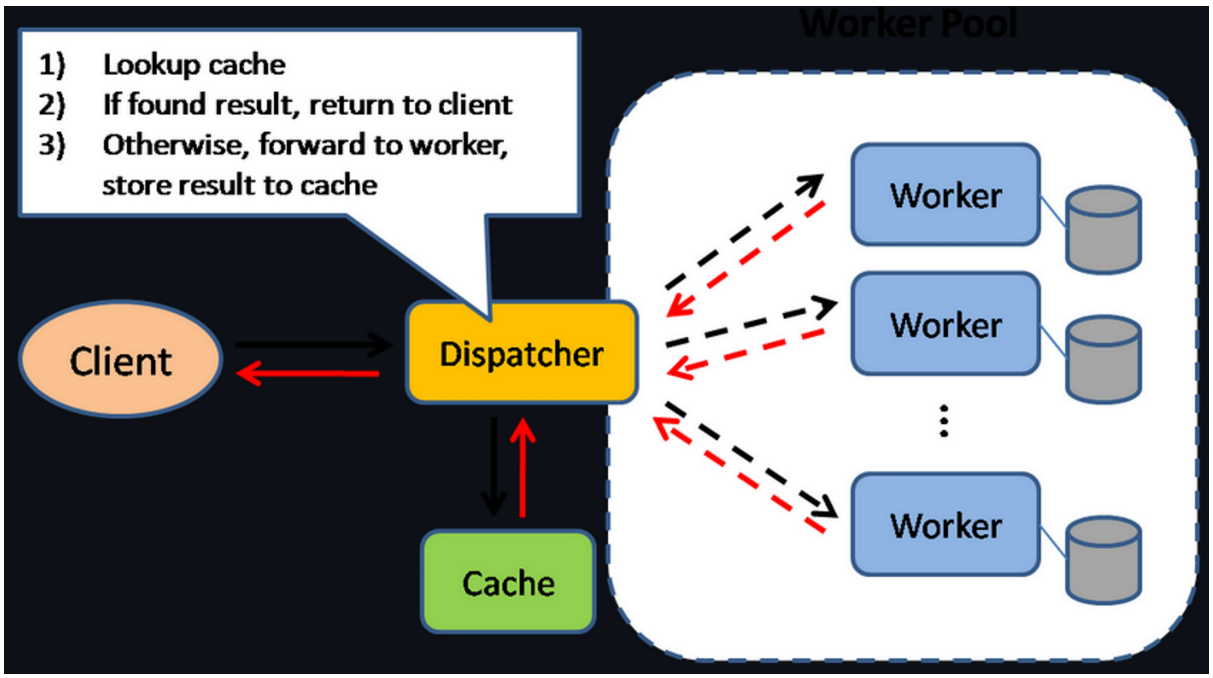
- Dynamic or flexible schema
- Non-relational data
- No need for complex joins
- Store many TB (or PB) of data
- Very data intensive workload
- Very high throughput for IOPS (Input/Output Operations Per Second)

Sample data well-suited for NoSQL:

- Rapid ingest of clickstream and log data
- Leaderboard or scoring data
- Temporary data, such as a shopping cart
- Frequently accessed ('hot') tables
- Metadata/lookup tables

9 - Cache

domingo, 28 de abril de 2024 16:56



Caching improves page load times and can reduce the load on your servers and databases. In this model, the dispatcher will first lookup if the request has been made before and try to find the previous result to return, in order to save the actual execution.

Databases often benefit from a uniform distribution of reads and writes across its partitions. Popular items can skew the distribution, causing bottlenecks. Putting a cache in front of a database can help absorb uneven loads and spikes in traffic.

Caches can be located on the client side (OS or browser), server side, or in a distinct cache layer.

Types of caching:

CDN caching

CDNs are considered a type of cache.

Web server caching

Reverse proxies and caches such as Varnish can serve static and dynamic content directly. Web servers can also cache requests, returning responses without having to contact application servers.

Database caching

Your database usually includes some level of caching in a default configuration, optimized for a generic use case. Tweaking these settings for specific usage patterns can further boost performance.

Application caching

In-memory caches such as Memcached and Redis are key-value stores between your application and your data storage. Since the data is held in RAM, it is much faster than typical databases where data is stored on disk. RAM is more limited than disk, so cache invalidation algorithms such as least recently used (LRU) can help invalidate 'cold' entries and keep 'hot' data in RAM.

Redis has the following additional features:

- Persistence option
- Built-in data structures such as sorted sets and lists

There are multiple levels you can cache that fall into two general categories: **database queries** and **objects**:

- Row level
- Query-level
- Fully-formed serializable objects
- Fully-rendered HTML

Generally, you should try to avoid file-based caching, as it makes cloning and auto-scaling more difficult.

Caching at the database query level

Whenever you query the database, hash the query as a key and store the result to the cache. This approach suffers from expiration issues:

- Hard to delete a cached result with complex queries
- If one piece of data changes such as a table cell, you need to delete all cached queries that might include the changed cell

Caching at the object level

See your data as an object, similar to what you do with your application code. Have your application assemble the dataset from the database into a class instance or a data structure(s):

- Remove the object from cache if its underlying data has changed
- Allows for asynchronous processing: workers assemble objects by consuming the latest cached object

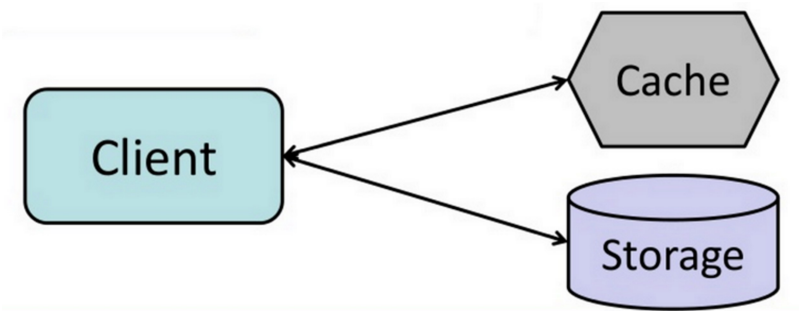
Suggestions of what to cache:

- User sessions
- Fully rendered web pages
- Activity streams
- User graph data

When to update the cache

Since you can only store a limited amount of data in cache, you'll need to determine which cache update strategy works best for your use case.

Cache-aside



The application is responsible for reading and writing from storage. The cache does not interact with storage directly. The application does the following:

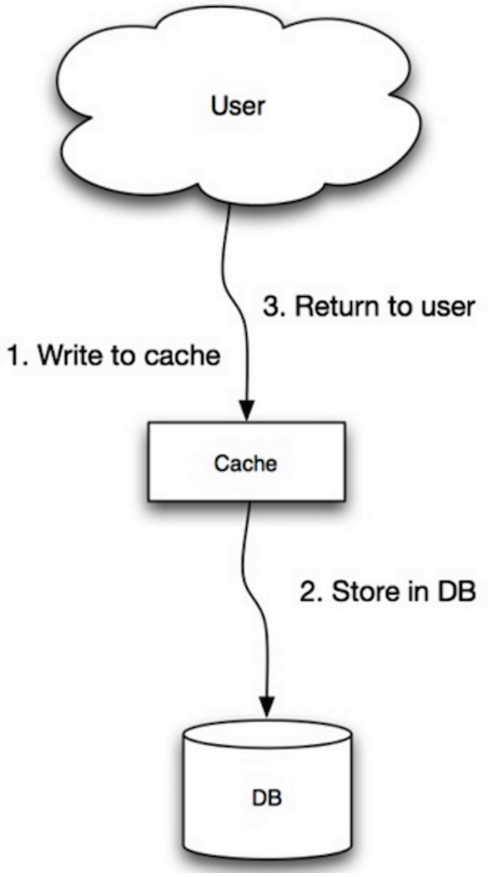
- Look for entry in cache, resulting in a cache miss
- Load entry from the database
- Add entry to cache
- Return entry

Memcached is generally used in this manner.
Subsequent reads of data added to cache are fast. Cache-aside is also referred to as lazy loading. Only requested data is cached, which avoids filling up the cache with data that isn't requested.

Disadvantage(s): cache-aside

- Each cache miss results in three trips, which can cause a noticeable delay.
- Data can become stale if it is updated in the database. This issue is mitigated by setting a time-to-live (TTL) which forces an update of the cache entry, or by using write-through.
- When a node fails, it is replaced by a new, empty node, increasing latency.

Write-through



The application uses the cache as the main data store, reading and writing data to it, while the cache is responsible for reading and writing to the database:

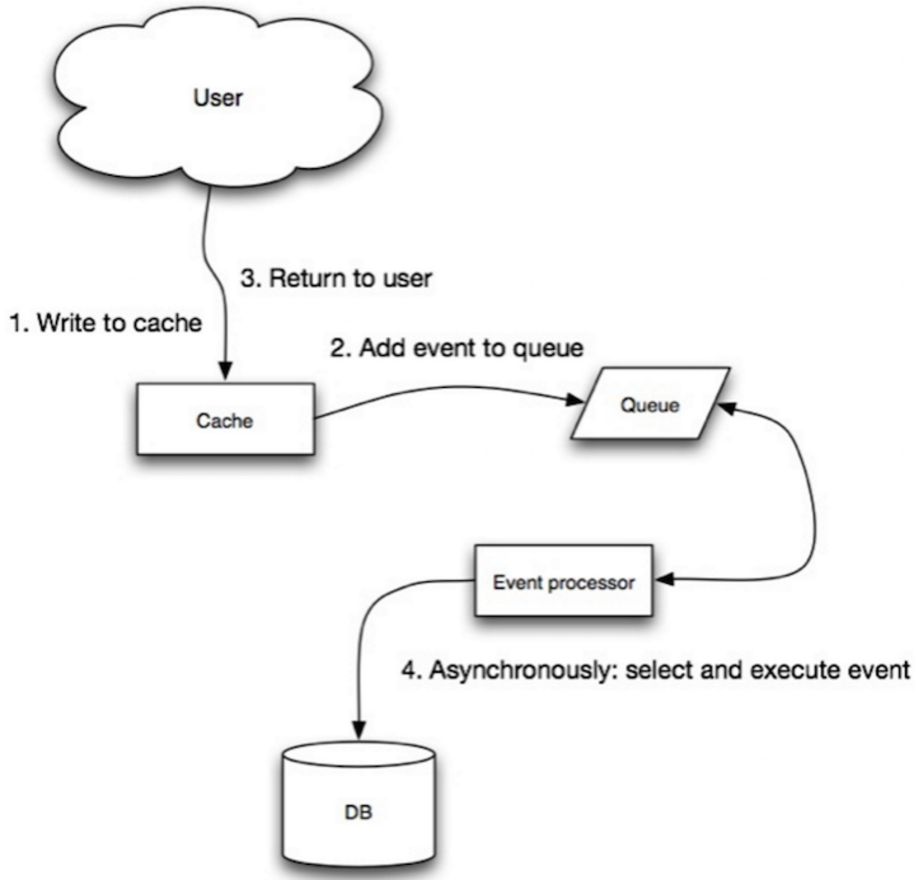
- Application adds/updates entry in cache
- Cache synchronously writes entry to data store
- Return

Write-through is a slow overall operation due to the write operation, but subsequent reads of just written data are fast. Users are generally more tolerant of latency when updating data than reading data. Data in the cache is not stale.

Disadvantage(s): write through

- When a new node is created due to failure or scaling, the new node will not cache entries until the entry is updated in the database. Cache-aside in conjunction with write through can mitigate this issue.
- Most data written might never be read, which can be minimized with a TTL.

Write-behind (write-back)



In write-behind, the application does the following:

- Add/update entry in cache
- Asynchronously write entry to the data store, improving write performance

Disadvantage(s): write-behind

- There could be data loss if the cache goes down prior to its contents hitting the data store.
- It is more complex to implement write-behind than it is to implement cache-aside or write-through.

Refresh-ahead

You can configure the cache to automatically refresh any recently accessed cache entry prior to its expiration.
Refresh-ahead can result in reduced latency vs read-through if the cache can accurately predict which items are likely to be needed in the future.

Disadvantage(s): refresh-ahead

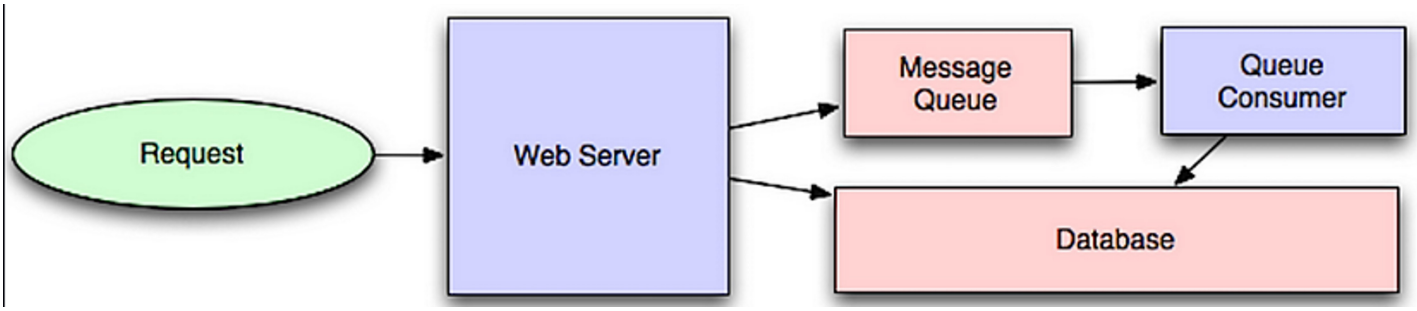
- Not accurately predicting which items are likely to be needed in the future can result in reduced performance than without refresh-ahead.

Disadvantage(s): cache

- Need to maintain consistency between caches and the source of truth such as the database through cache invalidation.
- Cache invalidation is a difficult problem, there is additional complexity associated with when to update the cache.
- Need to make application changes such as adding Redis or memcached.

10 - Asynchronism

domingo, 28 de abril de 2024 18:26



Asynchronous workflows help reduce request times for expensive operations that would otherwise be performed in-line. They can also help by doing time-consuming work in advance, such as periodic aggregation of data.

Message queues

Message queues receive, hold, and deliver messages. If an operation is too slow to perform inline, you can use a message queue with the following workflow:

- An application publishes a job to the queue, then notifies the user of job status
- A worker picks up the job from the queue, processes it, then signals the job is complete

The user is not blocked and the job is processed in the background. During this time, the client might optionally do a small amount of processing to make it seem like the task has completed. For example, if posting a tweet, the tweet could be instantly posted to your timeline, but it could take some time before your tweet is actually delivered to all of your followers.

Redis is useful as a simple message broker but messages can be lost.
RabbitMQ is popular but requires you to adapt to the 'AMQP' protocol and manage your own nodes.
Amazon SQS is hosted but can have high latency and has the possibility of messages being delivered twice.

Task queues

Tasks queues receive tasks and their related data, runs them, then delivers their results. They can support scheduling and can be used to run computationally-intensive jobs in the background.
Celery has support for scheduling and primarily has python support.

Back pressure

If queues start to grow significantly, the queue size can become larger than memory, resulting in cache misses, disk reads, and even slower performance. Back pressure can help by limiting the queue size, thereby maintaining a high throughput rate and good response times for jobs already in the queue. Once the queue fills up, clients get a server busy or HTTP 503 status code to try again later. Clients can retry the request at a later time, perhaps with exponential backoff.

Disadvantage(s): asynchronism

- Use cases such as inexpensive calculations and realtime workflows might be better suited for synchronous operations, as introducing queues can add delays and complexity.

11 - Communication

domingo, 28 de abril de 2024 18:38

OSI (Open Systems Interconnection) 7 Layer Model

Layer	Application/Example		Central Device/ Protocols	
Application (7) Serves as the window for users and application processes to access the network services.	**End User layer** Program that opens what was sent or creates what is to be sent Resource sharing • Remote file access • Remote printer access • Directory services • Network management		**User Applications** SMTP	
Presentation (6) Formats the data to be presented to the Application layer. It can be viewed as the "Translator" for the network.	**Syntax layer** encrypt & decrypt (if needed) Character code translation • Data conversion • Data compression • Data encryption • **Character Set Translation**		JPEG/ASCII EBDIC/TIFF/GIF PICT	
Session (5) Allows session establishment between processes running on different stations.	**Synch & send to ports** (logical ports) Session establishment, maintenance and termination • Session support - perform security, name recognition, logging, etc.		**Logical Ports** RPC/SQL/NFS NetBIOS names	
Transport (4) Ensures that messages are delivered error-free, in sequence, and with no losses or duplications.	**TCP** Host to Host, Flow Control Message segmentation • Message acknowledgement • Message traffic control • Session multiplexing	PACKET FILTERING	TCP/SPX/UDP	
Network (3) Controls the operations of the subnet, deciding which physical path the data takes.	**Packets** ("letter", contains IP address) Routing • Subnet traffic control • Frame fragmentation • Logical-physical address mapping • Subnet usage accounting		**Routers** IP/IPX/ICMP	
Data Link (2) Provides error-free transfer of data frames from one node to another over the Physical layer.	**Frames** ("envelopes", contains MAC address) [NIC card — Switch — NIC card] (end to end) Establishes & terminates the logical link between nodes • Frame traffic control • Frame sequencing • Frame acknowledgment • Frame delimiting • Frame error checking • Media access control		**Switch Bridge WAP** PPP/SLIP	Land Based Layers
Physical (1) Concerned with the transmission and reception of the unstructured raw bit stream over the physical medium.	**Physical structure** Cables, hubs, etc. Data Encoding • Physical medium attachment • Transmission technique - Baseband or Broadband • Physical medium transmission Bits & Volts		**Hub**	

Hypertext transfer protocol (HTTP)

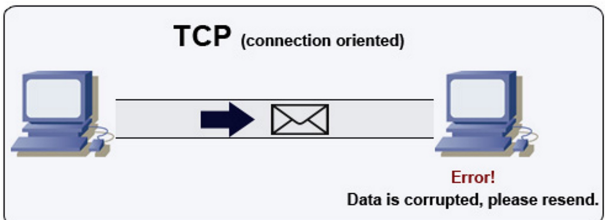
HTTP is a method for encoding and transporting data between a client and a server. It is a request/response protocol: clients issue requests and servers issue responses with relevant content and completion status info about the request. HTTP is self-contained, allowing requests and responses to flow through many intermediate routers and servers that perform load balancing, caching, encryption, and compression.

A basic HTTP request consists of a verb (method) and a resource (endpoint). Below are common HTTP verbs:

Verb	Description	Idempotent*	Safe	Cacheable
GET	Reads a resource	Yes	Yes	Yes
POST	Creates a resource or trigger a process that handles data	No	No	Yes if response contains freshness info
PUT	Creates or replace a resource	Yes	No	No
PATCH	Partially updates a resource	No	No	Yes if response contains freshness info
DELETE	Deletes a resource	Yes	No	No

*Can be called many times without different outcomes.
HTTP is an application layer protocol relying on lower-level protocols such as **TCP** and **UDP**.

Transmission control protocol (TCP)



TCP is a connection-oriented protocol over an IP network. Connection is established and terminated using a handshake. All packets sent are guaranteed to reach the destination in the original order and without corruption through:

- Sequence numbers and checksum fields for each packet
- Acknowledgement packets and automatic retransmission

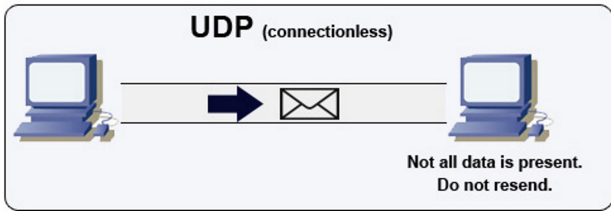
If the sender does not receive a correct response, it will resend the packets. If there are multiple timeouts, the connection is dropped. TCP also implements flow control and congestion control. These guarantees cause delays and generally result in less efficient transmission than UDP.

To ensure high throughput, web servers can keep a large number of TCP connections open, resulting in high memory usage. It can be expensive to have a large number of open connections between web server threads and say, a memcached server. Connection pooling can help in addition to switching to UDP where applicable.
TCP is useful for applications that require high reliability but are less time critical. Some examples include web servers, database info, SMTP, FTP, and SSH.

Use TCP over UDP when:

- You need all of the data to arrive intact
- You want to automatically make a best estimate use of the network throughput

User datagram protocol (UDP)



UDP is connectionless. Datagrams (analogous to packets) are guaranteed only at the datagram level. Datagrams might reach their destination out of order or not at all. UDP does not support congestion control. Without the guarantees that TCP support, UDP is generally more efficient.

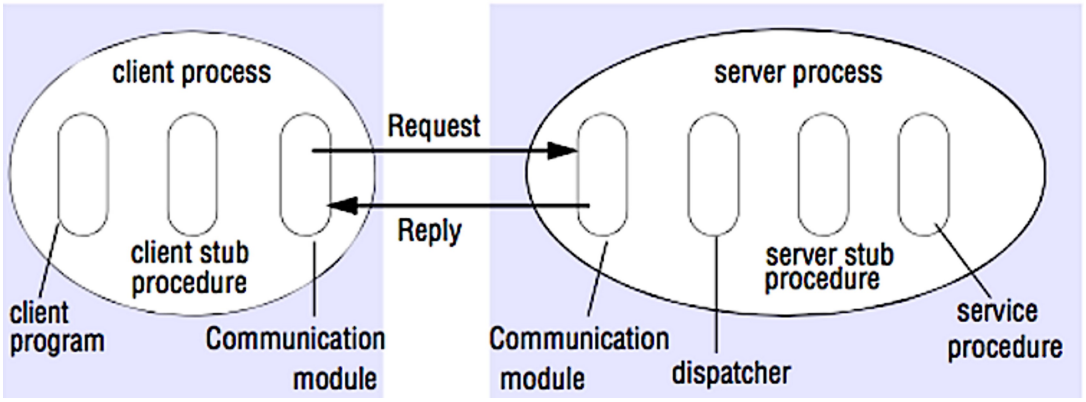
UDP can broadcast, sending datagrams to all devices on the subnet. This is useful with DHCP because the client has not yet received an IP address, thus preventing a way for TCP to stream without the IP address.

UDP is less reliable but works well in real time use cases such as VoIP, video chat, streaming, and realtime multiplayer games.

Use UDP over TCP when:

- You need the lowest latency
- Late data is worse than loss of data
- You want to implement your own error correction

Remote procedure call (RPC)



In an RPC, a client causes a procedure to execute on a different address space, usually a remote server. The procedure is coded as if it were a local procedure call, abstracting away the details of how to communicate with the server from the client program. Remote calls are usually slower and less reliable than local calls so it is helpful to distinguish RPC calls from local calls. Popular RPC frameworks include Protobuf, Thrift, and Avro.

RPC is a request-response protocol:

- **Client program** - Calls the client stub procedure. The parameters are pushed onto the stack like a local procedure call.

- **Client stub procedure** - Marshals (packs) procedure id and arguments into a request message.
- **Client communication module** - OS sends the message from the client to the server.
- **Server communication module** - OS passes the incoming packets to the server stub procedure.
- **Server stub procedure** - Unmarshalls the results, calls the server procedure matching the procedure id and passes the given arguments.
- The server response repeats the steps above in reverse order.

Sample RPC calls:

```
GET /someoperation?data=anId
POST /anotheroperation
{
  "data":"anId";
  "anotherdata": "another value"
}
```

RPC is focused on exposing behaviors. RPCs are often used for performance reasons with internal communications, as you can hand-craft native calls to better fit your use cases.

Choose a native library (aka SDK) when:

- You know your target platform.
- You want to control how your "logic" is accessed.
- You want to control how error control happens off your library.
- Performance and end user experience is your primary concern.

HTTP APIs following **REST** tend to be used more often for public APIs.

Disadvantage(s): RPC

- RPC clients become tightly coupled to the service implementation.
- A new API must be defined for every new operation or use case.
- It can be difficult to debug RPC.
- You might not be able to leverage existing technologies out of the box. For example, it might require additional effort to ensure RPC calls are properly cached on caching servers such as Squid.

Representational state transfer (REST)

REST is an architectural style enforcing a client/server model where the client acts on a set of resources managed by the server. The server provides a representation of resources and actions that can either manipulate or get a new representation of resources. All communication must be stateless and cacheable.

There are four qualities of a RESTful interface:

- **Identify resources (URI in HTTP)** - use the same URI regardless of any operation.
- **Change with representations (Verbs in HTTP)** - use verbs, headers, and body.
- **Self-descriptive error message (status response in HTTP)** - Use status codes, don't reinvent the wheel.
- **HATEOAS (HTML interface for HTTP)** - your web service should be fully accessible in a browser.

Sample REST calls:

```
GET /someresources/anId

PUT /someresources/anId
{"anotherdata": "another value"}
```

REST is focused on exposing data. It minimizes the coupling between client/server and is often used for public HTTP APIs. REST uses a more generic and uniform method of exposing resources through URIs, representation through headers, and actions through verbs such as GET, POST, PUT, DELETE, and PATCH. Being stateless, REST is great for horizontal scaling and partitioning.

A stateless communication protocol, such as HTTP in the context of REST, means that each request from a client to a server contains all the information needed for the server to fulfill that request. The server does not maintain any knowledge of past interactions with the client. In other words, each request is independent and self-contained, and the server does not store any information about the client's previous requests or state.

Disadvantage(s): REST

- With REST being focused on exposing data, it might not be a good fit if resources are not naturally organized or accessed in a simple hierarchy. For example, returning all updated records from the past hour matching a particular set of events is not easily expressed as a path. With REST, it is likely to be implemented with a combination of URI path, query parameters, and possibly the request body.
- REST typically relies on a few verbs (GET, POST, PUT, DELETE, and PATCH) which sometimes doesn't fit your use case. For example, moving expired documents to the archive folder might not cleanly fit within these verbs.
- Fetching complicated resources with nested hierarchies requires multiple round trips between the client and server to render single views, e.g. fetching content of a blog entry and the comments on that entry. For mobile applications operating in variable network conditions, these multiple roundtrips are highly undesirable.
- Over time, more fields might be added to an API response and older clients will receive all new data fields, even those that they do not need, as a result, it bloats the payload size and leads to larger latencies.

RPC and REST calls comparison

Operation	RPC	REST
Signup	**POST** /signup	**POST** /persons

Resign	**POST** /resign
	{
	"personid": "1234"
	}
		DELETE /persons/1234

Read a person	**GET** /readPerson?personid=1234	**GET** /persons/1234
Read a person's items list	**GET** /readUsersItemsList?personid=1234	**GET** /persons/1234/items

Add an item to a person's items	**POST** /addItemToUsersItemsList
	{
	"personid": "1234";
	"itemid": "456"
	}
		POST /persons/1234/items
		{
		"itemid": "456"
		}

Update an item	**POST** /modifyItem
	{
	"itemid": "456";
	"key": "value"
	}
		PUT /items/456
		{
		"key": "value"
		}

Delete an item	**POST** /removeItem
	{
	"itemid": "456"
	}
		DELETE /items/456

12 - Security

domingo, 28 de abril de 2024 18:41

- **Encrypt in Transit and at Rest**: Ensure that sensitive data is encrypted both when it is being transmitted over networks (in transit) and when it is stored in databases or storage systems (at rest). Encrypting data in transit prevents unauthorized interception or eavesdropping during transmission, while encrypting data at rest protects it from unauthorized access if the storage medium is compromised.

- **Sanitize All User Inputs**: Validate and sanitize all user inputs or any input parameters exposed to users to prevent Cross-Site Scripting (XSS) and SQL injection attacks. XSS attacks occur when malicious scripts are injected into web pages and executed in the context of a user's browser, potentially compromising user data or hijacking sessions. SQL injection attacks involve injecting malicious SQL queries into input fields to manipulate or access unauthorized data in databases.

- **Use Parameterized Queries**: Utilize parameterized queries or prepared statements in database interactions to prevent SQL injection attacks. Parameterized queries separate SQL code from user input data, ensuring that user input is treated as data rather than executable code. This approach mitigates the risk of SQL injection vulnerabilities by preventing malicious input from altering the structure or behavior of SQL queries.

- **Principle of Least Privilege**: Follow the principle of least privilege, which states that users should be granted the minimum level of access or permissions necessary to perform their tasks. Restrict access to sensitive resources and limit privileges to only those required for legitimate operations. By minimizing access rights, organizations can reduce the potential impact of security breaches and limit the scope of unauthorized activities.

By adhering to these security best practices, organizations can enhance the security posture of their systems and applications, mitigating the risk of common security threats and vulnerabilities.