**System Design**

Contents

[ **Distributed System Fundamentals** 1](#_Toc217924751)

[ **System Design** 5](#_Toc217924752)

[ **References** 40](#_Toc217924753)

Start **broad** and go deeper in a few areas.

The **system design interview** is an **open-ended conversation**.

Keep in mind that everything is a **trade-off**.

* **Steps to learn System Design**

1. Distributed System Fundamentals
2. The architecture of large-scale web applications
3. Designing distributed systems.

* **Distributed System Fundamentals**

**Distributed systems** scale our applications at exponential rates. Is a collection of computers that work together to form a single computer for the end user. All of the computers in the collection share the same state and operate concurrently. These machines can also fail independently without affecting the entire system.

Data Structures and Durability

Replication

Partitioning and Sharding

Consistent hashing

Distributed Transactions

Stateless and stateful systems

* **Benefits**

Scaling: **scale horizontally** to account for more traffic.

Modular growth

**Fault tolerance**: are more fault-tolerant than single machines.

Cost-effective

Low latency: You can have a node in multiple locations, so traffic will hit the closest node.

Efficiency: break complex data into smaller pieces.

Parallelism: multiple processors divide up a complex problem into smaller chunks.

<https://www.educative.io/blog/distributed-systems-considerations-tradeoffs>

<https://www.educative.io/blog/distributed-system-design-patterns>

* **Distributed system failures**
* System failure: occur because of software or hardware failures.
* Communication medium failure: occurs as a result of communication link failures or the shifting of nodes.
* Secondary storage failure: occurs when the information on the secondary storage device is inaccessible. It can be the result of many different things, including node crashing, dirt on the medium, and parity errors.
* Methods failure: Method failures usually halt the distributed system and make it unable to perform any executions at all. A system may enter a deadlock state or do protection violations during method failures.
* **MapReduce**

Is a framework developed by Google to handle large amounts of data in an efficient manner.

* **Stateless and Stateful systems**
* Stateless system: maintains **no state** of past events. It executes based on the inputs we provide to it.
* Stateful system: maintains and mutates a state.
* **Raft**

Establishes the concept of a replicated state machine and the associated replicated log of commands.

* **Distributed System Design patterns**

Categories:

* **Object communication**: describes messaging protocols and permissions for different components of the system to communicate.
* **Security**: handles confidentiality, integrity, and availability concerns to ensure the system is secure from unauthorized access.
* **Event-driven**: describes the production, detection, consumption, and response to system events.

Top Distributes System Design Patterns:

* Command and Query Responsibility Segregation (**CQRS**)
* Two-Phase Commit (2PC)
* Saga
* **Replicated Load-Balanced Services** (RLBS)
* Sharded Services

<https://www.educative.io/blog/distributed-system-design-patterns>

* **The architecture of large-scale web applications**

Most large-scale applications are web applications.

* N-tier applications: or distributed applications that have more than three components involved.

Those components can be:

Caches

Message queues

Load balancers

Search servers

Components involved in processing large amounts of data.

Components running heterogeneous tech, commonly known as web services.

Examples: Instagram, Facebook, Uber.

* HTTP and REST APIs
* DNS (Domain Name System): mapping simple domain names to IP addresses.
* Caching: is hardware or software that you use to temporarily store data so it can be accessed quickly.

Used by cache clients: web browsers, CPUs, operating systems, DNS servers.

* Cache invalidation
* **Cache eviction**: If a cache has space, data will be easily inserted. If a cache is full, some data will be evicted. Eviction policies:

First in first out (FIFO): the cache evicts the first block accessed first without any regard to how often or how many times it was accessed before.

Last in first out (LIFO): the cache evicts the block accessed most recently first without any regard to how often or how many times it was accessed before.

**Least recently used (LRU): the cache evicts the least recently used items first.**

Most recently used (MRU): the cache evicts the most recently used items first.

Least frequently used (LFU): the cache counts how often an item is needed. The items that are used least frequently are evicted first.

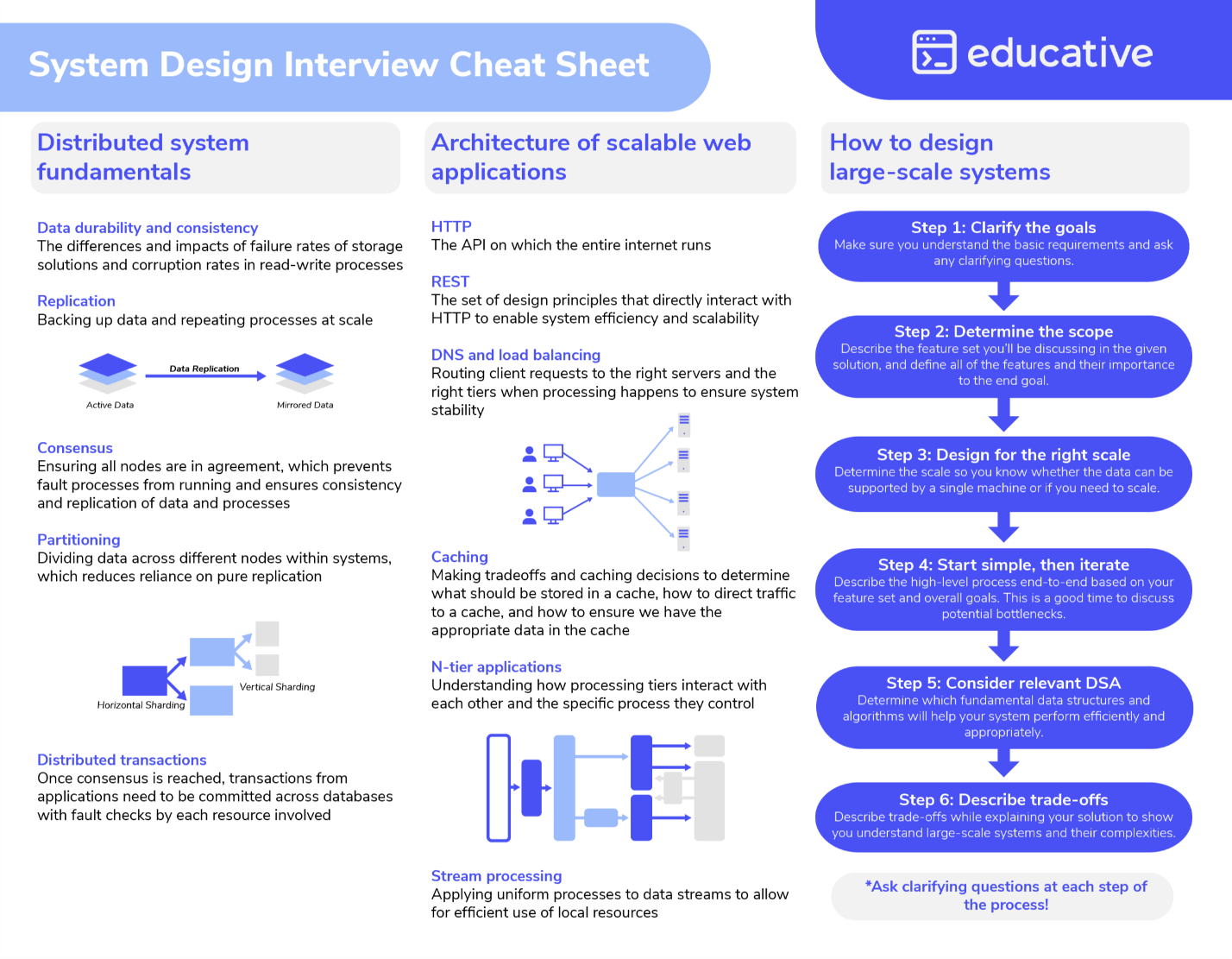
Random replacement (RR): the cache randomly selects a candidate and evicts it.

* Load balancing: scale effectively with increases in traffic and stay highly available.
* Microservices
* Key-value storage
* Stream Processing: focus on the real-time processing of continuous streams of data.

Example: Kafta, Storm, Flink.

<https://www.heavy.ai/technical-glossary/stream-processing>

* **Designing distributed systems**



Example of functional and non-functional requirements.

* **Building Blocks**

**Domain Name Systems**: maps domain names to IP addresses.

**Load Balancers**: distributes client requests among servers.

A diagram of a computer system

Description automatically generated

Client send a request, Load Balancer send request to first Server, Server response is available.

**Databases**: stores, retrieves, modifiers & deletes data.

**Key-Value Store**: stores data as key-value pairs.

**Content Delivery Network**: distributes in-demand content to end users.

**Sequencer**: generates unique ID for events & database entries.

**Service Monitoring**: analyzes system for failures & send alerts.

**Distributed Caching**: stores frequently accessed data.

**Distributed Messaging Queue**: decouples messaging producers from consumers.

**Publish-Subscribe System**: supports asynchronous service-to-service communication.

**Rate Limiter**: Throttles incoming request for services.

**Blob Store**: stores unstructured data.

**Distributed Search**: returns relevant content for user queries.

**Distributed Logging**: enables services to log events.

**Distributed Task Scheduling**: allocates resources to tasks.

**Sharded Counters**: counts concurrent read/write requests.

* **System Design**
* **Principles of scalable system design**

A screenshot of a computer screen

Description automatically generated

**System Design** defines all the elements in a **distributed system** (modules, components, **architecture**, and interfaces) based on the specific **needs** of an organization (optimizing user experience, minimizing cost).

* **Stages**

**Requirements** gathering

**Analysis**

**Architecture** design

**Component** design

**Interface** design

**Testing**

**Deployment**

* **System Design Templates**

[**https://leetcode.com/discuss/career/229177/My-System-Design-Template**](https://leetcode.com/discuss/career/229177/My-System-Design-Template)

* **The RESHADED Approach for System Design**

A diagram of a process

Description automatically generated

1. **Requirements**: what the service is, how it works, what its main features are. Gather Functional and Non-functional requirements.

Consider:

System goals

Key features

System constraints

User expectations

* **Functional requirements**: are requirements the system has to deliver. Include business rules, authentication, administrative functions, authorization levels.
* **Non-Functional requirements**: restrict the system design through different qualities. They need to be analyzed, and if they are not fulfilled, they can harm the business plan or goals. Include: performance, security, reliability, scalability, maintainability, availability.

Example: Twitter

Functional requirements:

Users should be able to post new tweets.

Users should be able to follow another user.

Users should be able to mark tweets as favorite.

Non-functional requirements:

High availability

Consistency

A latency of around 200ms for timeline generation

1. **Estimation**: help us understand the scale of the system we’ll design. Estimate **hardware** & infrastructure needed to implement at scale.

Consider requirements for:

Numbers of servers

Daily storage

Network

Example: Twitter

Keep in mind parameters like storage, bandwidth estimation, total tweet-views, etc. Let’s say we have around 200 million daily active users, a hundred million new tweets, and each user follows about 200 people.

Example:

We’ll decide what type of database to use for storing our data, which data structure will give optimal performance.

Examples of questions:

How many servers will we require to provide smooth services to 500 million Daily Active Users (DAU).

How much storage do we need if we have to store 125 million tweets per day, and 20% of tweets contain media?

1. **Storage schema (optional)**: articulate data model, which tables we need and what type of fields are part of each table.

Define:

Structure of data

Tables to use

Type of fields in tables

Relationship between tables (optional)

Relevant when you:

Expect highly normalized data.

Will store different parts of data in various formats.

Face performance & efficiency concerns around storage.

1. **High-level design**: identifying the main components and building **blocks** we’ll use to design our desired system, from our functional and non-functional requirements.

For each, identify:

**How** they work.

**Why** they’re needed.

**How** they integrate.

This layered visual shows dependencies between building blocks. **Blocks in lower layers support those above**.

A diagram of a building block

Description automatically generated with medium confidence

1. **API design**: build interface (API calls) for our service. Translate functional requirements into API calls.

Example:

Requirement: Users should be able to access all items.

API call: GET /items

1. **Detailed design**: limitations of the high-level design, workflow of our design and its usage of different technologies.

Improve high-level design.

Consider all non-functional requirements & complete design.

Examples:

How much data do we need to cache to speed up the response time?

Where should we need to use load-balancer?

Do we need to partition data to distribute to multiple databases?

1. **Evaluation**: measure the effectiveness of our solution.

Evaluate design against requirements.

Explain tradeoffs & pros/cons of different solutions.

Address overlooked design problems.

Identify and resolve bottlenecks.

Examples:

Is there a single point of failure in this system? How do we remove it?

Do you have enough data replicas to serve the user in case you lose a few servers?

Do we have enough copies of our services to prevent shutdown?

1. **Distinctive component/feature**: identify a unique aspect for each design problem.

Discuss a distinctive feature that meets requirements.

Example:

**Concurrency control** in high-traffic apps.

The Uber design problem has payment service and fraud detection as its unique feature. In contrast, **Google Docs** has concurrency control, which is required when different users want to edit the same section of a document simultaneously.

<https://www.educative.io/courses/grokking-modern-system-design-interview-for-engineers-managers/the-reshaded-approach-for-system-design>

* **Design methods**
* **Architectural** **design**: views, model, behavior, **infrastructure** of a system.
* **Logical** **design**: data flow and **inputs/outputs** of a system.
* **Physical** **design**: how users can add information, how a system represents **information** to users, and how data is **modeled/stored**.
* **System Design patterns**
  + **Bloom filters**

Are probabilistic data structures designed to determine whether an item does not exist in a set or if an item might exist in a set.

* + **Consistent hashing**

Maps data to physical nodes and ensures that only a small set of keys move when servers are added or removed.

<https://www.educative.io/answers/what-is-consistent-hashing>

* + **Quorum**

Is the minimum number of servers on which a distributed operation needs to be performed successfully before declaring the operation’s overall success.

* + **Checksum**
  + **Merkle trees**

Is a binary tree of hashes, in which each internal node is the hash of its two children, and each leaf node is a hash of a portion of the original data.

* + **Leader election**

Is the process of designating a single process as the organizer of tasks distributed across several computers.

<https://aws.amazon.com/builders-library/leader-election-in-distributed-systems/>

* **Scalability**

Application’s ability to **handle and withstand an increased workload without sacrificing latency**.

* + **Patterns**

<https://horicky.blogspot.com/2010/10/scalable-system-design-patterns.html>

Ways to scale an application:

1. **Horizontal scaling (scaling out):**

Adding more **hardware (machines or nodes)** to distribute the load, increases the computational power of the system as a whole.

Advantages:

Scalability

Fault Tolerance: if one node fails, others can continue to operate.

Flexibility

Cost-Effective

Disadvantages:

Complexity: requires **load balancing**, **distributed storage**, and **data consistency mechanisms**.

Latency

Use Cases:

**Web applications** with high user traffic: **social media** platforms.

**Distributed** databases: NoSQL databases like Casandra or MongoDB.

**Microservices** architectures

A diagram of a computer

Description automatically generated

Examples:

**Web Servers**:

Scenario: A popular **e-commerce** **website** experiences a significant increase in **traffic** during holiday sales.

Solution: Add more web servers to the **server pool**. A **load balancer** distributes incoming requests across the multiple servers to **handle** the increased traffic.

**NoSQL Databases**:

Scenario: A social media platform needs to store and manage a rapidly growing amount of user data.

Solution: Implement a **NoSQL database** like **Cassandra** or **MongoDB** that can horizontally scale by adding more nodes to the **database cluster**. Each **node** **handles** a **portion** of the **data**, allowing for efficient data distribution and access.

**Microservices Architecture**:

Scenario: A large online service uses **microservices** to handle different parts of its functionality (user authentication, payment processing, content delivery).

Solution: Each microservice runs on its own set of **instances**, and more instances can be added as needed to handle increased load. This allows for scaling out individual services independently.

1. **Vertical scaling (scaling up):**

Adding more power to your server, increases the power of the hardware running the application.

Involves **increasing** the **capacity** of an **existing machine or node** by adding more resources (**CPU**, **RAM**, **storage**).

Upgrades the hardware of a single node **to handle more load**.

Advantages:

Simplicity: involves a single system, **avoiding** the complexities of **distributed systems**.

Performance

Low Latency: resources are local to the **single machine**.

Disadvantages:

Cost: very expensive.

Limits: how much a single machine can be upgraded.

Risk: if the machine goes down, the entire system is affected.

Use Cases:

**Legacy** applications

Databases: for **transactional operations** (like relational databases).

Workloads: **large-scale data analysis**.

Examples:

CPU:

cores, L2 Cache

Disk:

for hard drives: PATA, SATA, SAS

RAID

RAM

**Relational Databases**:

Scenario: A financial institution needs a highly reliable and powerful database to handle transactions.

Solution: Upgrade the existing **database server** by adding more **CPU**, **memory**, and **storage**. This increases the database’s capacity to process more transactions and store more data without changing the application architecture.

**Application Server**:

Scenario: A medium-sized business runs an enterprise application that becomes slow due to growing data and user demands.

Solution: Increase the hardware specifications of the application server, such as adding more **RAM** and faster processors, to improve performance and handle more simultaneous users.

**In-Memory Databases**:

Scenario: A **real-time analytics** application needs to process large amounts of **data quickly**.

Solution: Use an **in-memory** database like **Redis** and vertically scale it by adding more **memory** and **CPU** to the server running Redis. This allows the application to **store** and process more **data in memory**, resulting in faster performance.

1. **Hybrid:**

Use a combination of both horizontal and vertical scaling to leverage the performance and capacity needs efficiently:

Example:

**Cloud Services**

Scenario: A software-as-a-service (**SaaS**) company hosts its application on a **cloud platform**.

Solution: Initially, the company might scale vertically by upgrading virtual machines (VMs) with more resources. As the user base grows, they might scale horizontally by adding more VMs and using auto-scaling groups to automatically manage the number of instances based on demand.

* **Horizontal vs Vertical Scaling:**

|  |  |  |
| --- | --- | --- |
| **Considerations** | **Horizontal Scaling** | **Vertical Scaling** |
| Workload Type | Better for **distributed** workloads | Suitable for intensive, **single**-machine tasks. |
| Budget | Can be more cost-effective with commodity.  More cost-effective in the long run. | Might require significant investment in high-end hardware.  Cheaper in the short term. |
| System Architecture | Distributed manner for application or data storage. | Single system for application or data storage. |
| Future Growth | If you anticipate continuous growth, it offers more flexibility to scale out as needed. |  |

A white rectangular boxes with blue stripes

AI-generated content may be incorrect.

A diagram of a computer system

Description automatically generated with medium confidence

A diagram of a server

Description automatically generated with medium confidence

* + **Scalability Trade-offs**

**Scalability, Availability, Stability Patterns**

<https://www.slideshare.net/slideshow/scalability-availability-stability-patterns/4062682#11>

1. **Performance vs Scalability:**

A service is **scalable** if it results in increased **performance** in a manner proportional to resources added.

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**.

1. **Latency vs Throughput:**

**Latency**: **time** to perform some action or to produce some result.

**Throughput**: **number of actions** or results per unit of time.

You should aim for **maximal throughput** with **acceptable latency.**

e.g.: An assembly line is manufacturing cars. It takes eight hours to manufacture a car and the factory produces 120 cars per day.

Latency: 8 hours

Throughput: 120 cars/day or 5 cars/hour.

1. **Availability vs Consistency:**
   1. **CAP theorem (by Eric Brewer):**

It states that a distributed system can only provide **2** of 3 properties simultaneously (asynchronous network) across of write/read storage.

Formalizes the **trade-off** between **consistency** and **availability** in the presence of **partitions**.

You can either be **Consistent or Available**, but **no both**, e.g.: we can’t build a database that both responds to every request and returns the results that you would expect every time.

**C**onsistency: a read is guaranteed to return the most recent write or an error.

**A**vailability: every request receives a response within a reasonable amount of time (no error or timeout), without guarantee that it contains the most recent version of the information.

**P**artition Tolerance: the system continues to operate despite arbitrary partitioning due to network failures.

A diagram of a variety of colors

Description automatically generated

**Networks aren’t reliable**, so you’ll need to support **partition tolerance**. You’ll need to make a **software** **tradeoff** between **consistency** and **availability**.

**Centralized system (RDBMS, etc): don’t have network partitions (P in CAP). You get both: Availability and Consistency.**

**Distributed system: will have network partitions (P in CAP). You get only pick one: Availability or Consistency.**

**CP: Consistency and partition tolerance:**

Waiting for a response from the partitioned node might result in a timeout error.

CP (choose Consistency over Availability) is a good choice if your business needs require **atomic reads** and **writes**.

**AP: Availability and partition tolerance:**

Availability over Consistency

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

Good choice if the business needs to allow for **eventual consistency** or when the system needs to continue working despite external errors (shopping carts, etc).

The decision between **Consistency** and **Availability** is a **software** **trade off**.

System design fundamentals: What is the CAP theorem?

<https://www.educative.io/blog/what-is-cap-theorem#whatiscaptheorem>

L16: The CAP Theorem

[**https://www.youtube.com/watch?v=k-Yaq8AHlFA**](https://www.youtube.com/watch?v=k-Yaq8AHlFA)

ATM example

* + **Consistency patterns**

Use case: read after write.

1. Weak consistency
2. Eventual consistency
3. String consistency

<https://snarfed.org/transactions_across_datacenters_io.html>

* + **Techniques and tradeoffs**

Master/Slave replication (M/S):

Usually asynchronous: good for throughout, latency

Most RDBMSes: MySQL binary logs

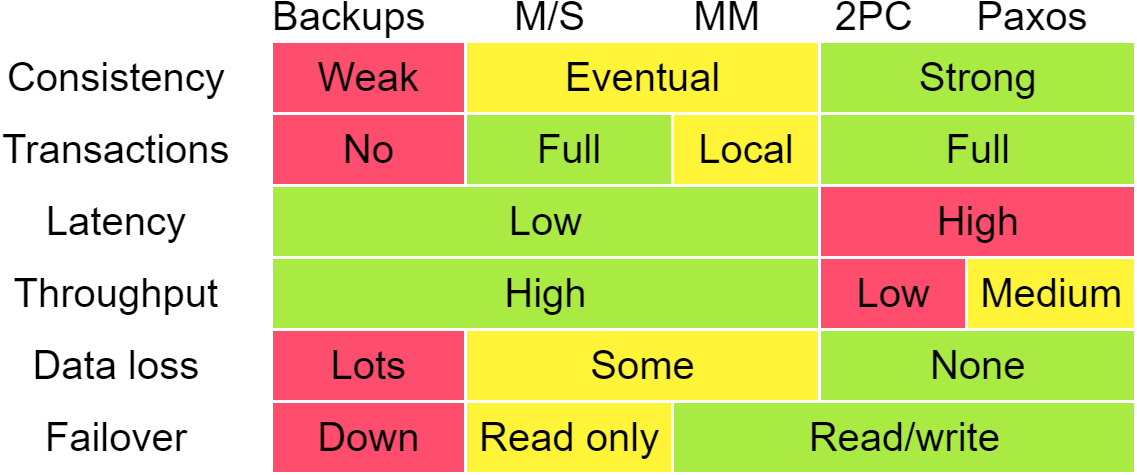
Weak/eventual consistency

Multi-master replication (MM):

Asynchronous, eventual consistency

No global transactions

Two Phase Commit (2PC)



* + **Availability patterns**

1. Fail-over

Active-passive

Active-active

1. Replication

Master-slave and master-master

Tree replication

Buddy Replication

* + **Stability Patterns**

Timeouts

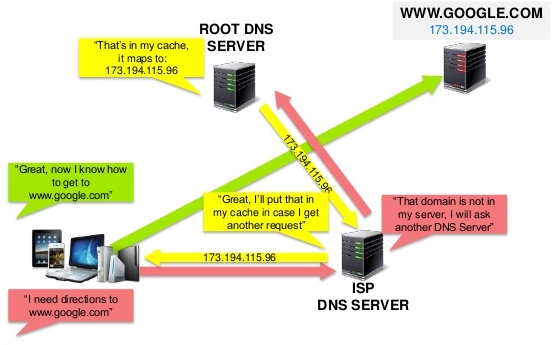
Circuit Breaker

Fail fast: avoid slow responses

Bulkheads

Throttling: maintain a steady pace

* **Domain Name System (DNS)**



**DNS** translates a **domain name** to an **IP address**.

**ISP** or **router** provides information about which **DNS servers** to contact when lookup.

**DNS** results can be **cached** by your browser or OS for a certain period of time determined by the TTL.

**Time To Live (TTL)**: limits the lifespan or lifetime of data in a computer or network.

* Who controls the DNS servers?

<https://superuser.com/questions/472695/who-controls-the-dns-servers/472729>

Level 1: DNS Root Servers: run the internet, cache, maintained by organizations, are 13 servers.

Level 2: Secondary DNS Servers: cache, maintained by governments, ISP’s, private companies (Google, OpenDNS).

* + Methods to route traffic used by DNS services

1. Weighted Round Robin

Prevent traffic from going to servers under maintenance.

Balance between varying cluster sizes.

A/B testing

1. Latency-based

<https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/routing-policy-latency.html>

1. Geolocation-based

Based on the geographic location of your users.

<https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/routing-policy-geo.html>

* + Disadvantages

Accessing a DNS server introduces a delay, although mitigated by caching.

DNS server could be complex, generally managed by governments, ISPs, and large companies.

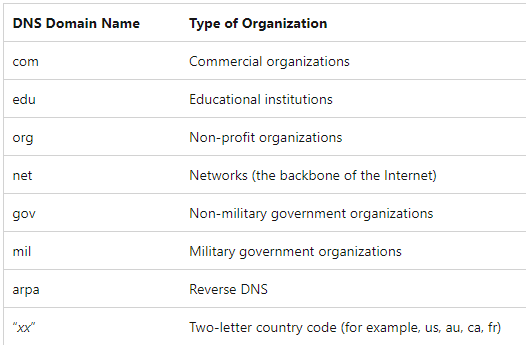
Recently come under **DDoS attack**.

* + DNS Architecture

<https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2008-R2-and-2008/dd197427(v=ws.10)?redirectedfrom=MSDN>

DNS domain namespace

DNS top-level domain names (TLDs)



DNS articles

<https://support.dnsimple.com/categories/dns/>

* **Content Delivery Network (CDN)**

Is a globally distributed network of **proxy servers**, serving content (static file: HTML, CSS, JSS, photos, videos, dynamic content: Amazon’s CloudFront) from locations closer to the user.



* Advantages:

Users receive content from data centers close to them.

Your servers do not have to serve requests that the CDN fulfills.

* Disadvantages:

Cost depending on traffic.

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.

* Operations:

Push CDNs: receive content when changes occur on your server.

Pull CDNs: grab content from server when the first user requests the content.

* **Load Balancer**

Can be implemented with **hardware** (expensive) or with **software** (HAProxy).

To protect against failures must set up multiple load balancers in active-passive or active-active mode.

A diagram of a worker

Description automatically generated

* Advantages:

Preventing requests from going to unhealthy servers.

Preventing overloading resources.

Helping to eliminate a single point of failure.

SSL termination

Session persistence

* Disadvantages:

A single point of failure, configuring multiple load balancers increase complexity.

* Horizontal scaling:

improve **performance** and **availability**.

Disadvantages:

Server should be **stateless**: **not** contain **user-related data** (**sessions**, **profile**).

**Sessions** can be stored in a centralized data stores (database SQL, NoSQL) or a persistent cache (**Redis, Memcached**).

* Metrics:

Random

Least loaded

Session/cookies

Round Robin or Weighted Round Robin

Layer 4: transport layer. Involve source, destination IP addresses, ports in the header, not content. Network Address Translation (NAT).

Layer 7: application layer. Involve header contents, message, cookies.

* + **Load Balancing Algorithms**

<https://www.jscape.com/blog/load-balancing-algorithms>

* Round Robin:

Most widely used algorithm

Manage requests in a cyclical fashion.

A computer network with a computer and a load balancer

Description automatically generated

No suitable for servers with disproportionate capacities (specs).

* Weighted Round Robin:

Manage requests considering weights (capacity, spec) of servers.

A computer network diagram with many lines

Description automatically generated with medium confidence

* Least Connections:

Manage requests considering instances/connections (# clients currently connected) to the servers.

A computer network with many lines

Description automatically generated with medium confidence

Clients 1 and 3 are already disconnected, while 2, 4, 5, 6 are still connected.

* Weighted Least Connections:

Manage requests considering weights/capacities of each server and instances/connections (# clients currently connected) to the servers.

* Random:

Manage requests randomly.

Suitable for servers (nodes) with similar configurations (CPU, RAM, etc).

Elastic Load Balancing (Amazon)

<https://docs.aws.amazon.com/elasticloadbalancing/latest/classic/elb-listener-config.html>

* **Reverse proxy (web server)**

A diagram of a red oval with black lines and arrows

Description automatically generated

Centralizes internal services and unified interfaces to the public.

Load balancer vs reverse proxy

Advantages:

Increased security

Increased scalability and flexibility

SSL termination

Caching

Disadvantages:

Single point of failure

Complexity

* **Application Layer**

A diagram of a web server

Description automatically generated

Microservices

Service Discovery:

Consul, Etcd, Zookeeper, help services find each other by keeping track of registered names, addresses, ports.

Heath checks

Service Oriented Architecture (SOA)

* **Microservices**

Is an **architectural style** that **structures an application using loosely coupled services**. It divides a large application into a collection of separate, **modular services**. The modules can be independently developed, deployed, and maintained.

A diagram of a microarch system

Description automatically generated

Microservices architecture tutorial: All you need to get started

<https://www.educative.io/blog/microservices-architecture-tutorial-all-you-need-to-get-started>

* **Proxy servers (Forward Proxy)**

**Act as a channel between a user and the internet**. It separates the end-user from the website they’re browsing.

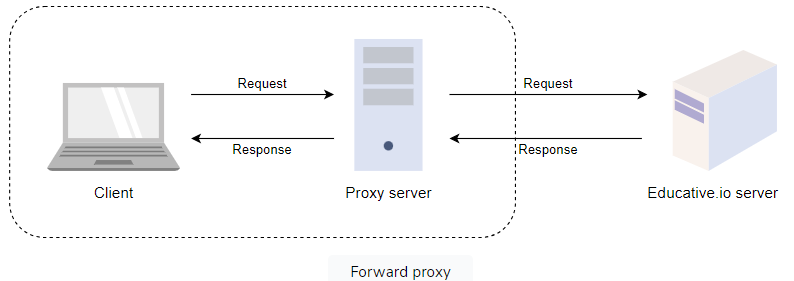
Improved security

Improved privacy

**Access to blocked resources**

Control of the internet usage of employees and children

Cache data to speed up requests



* **Redundancy and replication**
* **Redundancy**: process of **duplicating critical components of a system** with the intention of increasing a system’s reliability or overall performance. It usually comes in the form of a backup or fail-safe.
* **Replication**: process to **sharing information to ensure consistency between redundant resources**. You can replicate software or hardware components to improve reliability, fault-tolerance, or accessibility.

Used in database managements systems (**DBMS**) with a **primary-replica relationship** between the original and its copies.

A black text on a white background

Description automatically generated

* + **Storage**

Techniques:

* **Block storage**: data is broken down into blocks of equal sizes, and each individual block is given a unique identifier for easy accessibility.
* **File storage**: is a hierarchical storage methodology, the data is stored in files, the files are stored in folders, which are then stored in directories. Is only good for a limited amount of data, primarily structured data.
* **Object storage**: to handle large amounts of **unstructured** **data**, offers dynamic scalability.

A diagram of a cylinder

Description automatically generated

<https://www.educative.io/answers/what-is-object-storage>

* Redundant Array of Inexpensive Disk (RAID)

Use multiple disks in concert to build a faster, bigger, and more reliable disk system.

* **Message queues**

Routes messages from a source to a destination (from the sender to the receiver).

It follows the **FIFO** (first in first out) policy.

Facilitate asynchronous behavior, allows modules to communicate with each other in the background.

* Apache Kafka: distributed event streaming platform. Is capable of handling trillions of records per day.

Kafka, RabbitMQ, Kinesis, Flink

* **File systems**

Operations:

File naming

Storage management

Directories

Folders

Access rules

* Google File System (GFS)
* Hadoop Distributed File System (HDFS)
* **Databases**

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

<https://highscalability.com/an-unorthodox-approach-to-database-design-the-coming-of-the/>

* + **Relational databases**

SQL (Structured Query Language) databases, structured, have predefined schemas, stores data in rows and columns, tables.

Relational Database Management System (RDBMS):

MySQL

Oracle

MS SQL Server

Microsoft Access

SQLite

PostgreSQL

MariaDB

* **ACID properties:**

Maintain integrity of a database for **transactions**.

**Transactions**: unit of work, any change in a database, isolation.

**A**tomicity: each transaction is all or nothing

**C**onsistency: any transaction will bring the database from one valid state to another.

**I**solation: not affect other transactions. Transactions concurrently same results as transactions serially.

**D**urability: transaction committed must remain (persistent storage).

<https://www.educative.io/answers/what-are-acid-properties-in-a-database>

* Why transactions?

Correctness

Consistency

Enforce invariants

ACID

* Techniques to scale:

1. **Replication**

Disadvantages:

Potential for loss of data if master fails before any newly data can be replicated to other nodes.

More slaves, then more replicas, it increases replication lag.

Some cases write in parallel to master, whereas read support single thread.

Adds more hardware and complexity.

* 1. master-slave replication

A diagram of a person

Description automatically generated

If master goes offline, the system operates in read-only mode until a slave is promoted to a master or a new master is provisioned.

Disadvantages:

Add logic to promote slave to master.

* 1. **master-master replication**

A diagram of a diagram

Description automatically generated

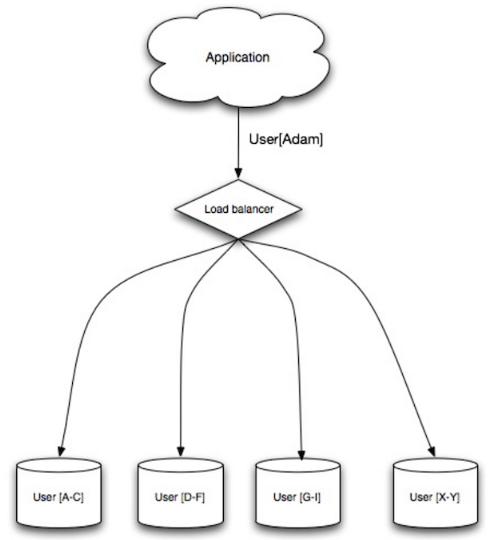
Disadvantages:

Need Load Balancer to determine where to write.

Increase latency.

1. **Federation (Functional partitioning)**
2. **Sharding**

Is a horizontal **partition**, data is divided into various smaller, distinct chunks called shards, stores in separate servers.



How to shard:

shard a table of users can be through the **user’s last name initial** or the **user’s geographic location**.

Advantages:

More write bandwidth: allow write in parallel with increases throughput.

Improve performance, scalability, availability, data distribution

Availability: if one shard goes down, the other shards are still operational, you can add replication to avoid data loss.

Faster queries: smaller amounts of data in each user group.

Less read/write traffic

Less replication

More cache hits

Easy to manage, fast

Disadvantages:

**Rebalancing** adds complexity: rebalancing happens when a shard outgrows its storage and needs to move to a different shard. A sharding function (**Consistency hashing**) can reduce the amount of transferred data.

Complex SQL queries

Joining data from multiple shards: complex SQL queries, to improve use **caching**.

Add more hardware and complexity.

Implementing shards is not well supported: implement by your own approach.

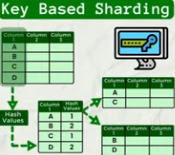
Types:

Horizontal/Vertical partitions

A diagram of a family

Description automatically generated

Key based sharding



Range based sharding

A diagram of a product

Description automatically generated

Directory based sharding

A screenshot of a diagram

Description automatically generated

**Partitioning**:

breaks up a big database into smaller parts across multiple machines to improve our application’s performance, availability, load balancing, and manageability.

<https://www.educative.io/answers/what-is-database-sharding>

Methods:

Horizontal: splits by rows into tables

Vertical (normalization): splits by cols into tables

Consistency Hashing

<https://www.paperplanes.de/2011/12/9/the-magic-of-consistent-hashing.html>

1. **Denormalization**

Redundant copies of the data are written in multiple tables to avoid expensive joins, increase **performance**. RDBMS use materialized views.

Disadvantages:

Data is duplicated

1. **SQL tuning**

<https://mode.com/sql-tutorial/sql-performance-tuning>

<https://www.geeksforgeeks.org/sql-performance-tuning/>

Database indexing:

Make faster and easier to search through your tables.

**Indexes** **speed** up data **retrieval**, but **slow down** data **insertion** and **updates** because of their **size**.

Database schemas:

Are abstract designs that **represent the storage** of data in a database.

<https://www.educative.io/blog/what-are-database-schemas-examples>

<https://www.educative.io/blog/database-design-tutorial>

* + **Non-relational databases**

No-SQL (Not Only SQL) databases, unstructured, have a dynamic schema.

Data is denormalized, joins are done in app code.

**Lack ACID transactions, favor Eventual Consistency.**

* **BASE properties:**

Choose availability over consistency

**BA**sically available

**S**oft state: the state of the system may change over time, even without input.

**E**ventual consistency: system become consistent over a period of time even when don’t receive input.

* **Types:**

**Key-value** store:

Abstraction: Hash table

read/write O(1)

Backed by memory or SSD

Hish performance used in in-memory cache layer.

e.g.: **Redis**, **DynamoDB** (Amazon), **Memcached**

Memcached architecture: <https://adayinthelifeof.nl/2011/02/06/memcache-internals/>

**Document** store:

Abstraction: key-value store with documents stored as values.

Documents (store objects): XML, JSON, YAML, BSON, binary, etc., are organized by collections, tags, metadata, directories.

**Used with changing data**.

e.g.: **MongoDB**, CouchDB, **DynamoDB** (key-values and documents), Elasticsearch

MongoDB: <https://www.mongodb.com/resources/products/fundamentals/mongodb-architecture>

CouchDB: <https://blog.couchdb.org/2016/08/01/couchdb-2-0-architecture/>

DynamoDB: <https://www.read.seas.harvard.edu/~kohler/class/cs239-w08/decandia07dynamo.pdf>

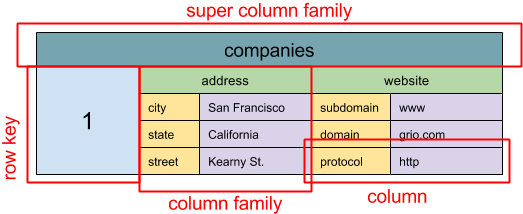
Elasticsearch: <https://www.elastic.co/blog/found-elasticsearch-from-the-bottom-up>

Document-oriented database

<https://en.wikipedia.org/wiki/Document-oriented_database>

**Wide-column** store:

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

****

High availability and scalability.

Allow efficient retrieval of selective key ranges.

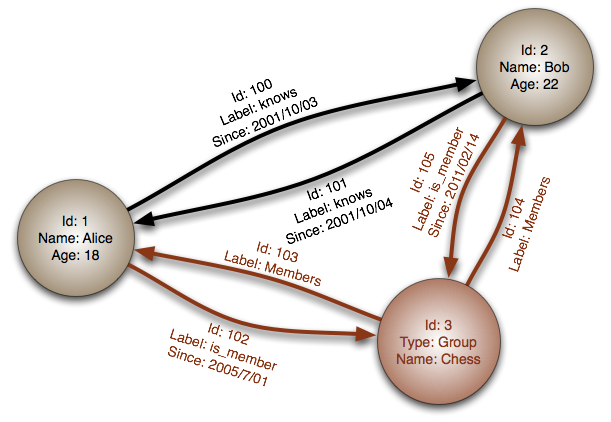
Used for very large data sets.

e.g.: **Bigtable** (Google), **HBase** (Yahoo open-source, used in Hadoop ecosystem), **Cassandra** (Facebook).

SQL & NoSQL, a brief history

<https://blog.grio.com/2015/11/sql-nosql-a-brief-history.html>

**Graph** database:



Each **node** is a record and each **arc** is a relationship between two nodes.

Represent complex relationships with many foreign keys or many-to-many relationships.

Many graphs can only be accessed with REST APIs.

e.g.: **Neo4J**, InfiniteGraph, FlockDB

**NoSQL databases a survey and decision guidance**

<https://medium.baqend.com/nosql-databases-a-survey-and-decision-guidance-ea7823a822d#.wskogqenq>

A diagram of a computer

Description automatically generated with medium confidence

A diagram of storage systems

Description automatically generated

A chart with text on it

Description automatically generated with medium confidence

A diagram of a computer flowchart

Description automatically generated

A table of information with text

Description automatically generated with medium confidence

NoSQL patterns

<https://horicky.blogspot.com/2009/11/nosql-patterns.html>

* + **How to choose a database**

Factor in speed, reliability and accuracy:

ACID

BASE

SQL joins

Normalization

Persistence

* SQL or NoSQL:

A close-up of a diagram

Description automatically generated

|  |  |
| --- | --- |
| **SQL** | **NoSQL** |
| Structured data | Semi-structured data |
| Strict schema | Dynamic or flexible schema |
| Relational data | Non-relational data |
| Need for complex joins | No need for complex joins |
| Transactions | Store many TB (or PB) of data |
| Clear patterns for scaling | Very data intensive workload |
| More established: developers, community, code, tools, etc. | Very high throughput for IOPS (Input/Output operations per Second) |
| Lookups by index are very fast |  |

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

<https://www.sitepoint.com/sql-vs-nosql-differences/>

* **Cache**

A diagram of a worker

Description automatically generated

Dispatcher = Load balancer

<https://www.slideshare.net/slideshow/from-cache-to-in-memory-data-grid-introduction-to-hazelcast/34802471>

* Advantages:

Improves page load times and can reduce the load on your servers and databases.

* Disadvantages:

Need to maintain consistency between caches and the source of truth (database) through cache invalidation.

Cache invalidation is complex.

Need to make application changes such as adding Redis or Memcached.

* Common Cache Attributes:

Maximum size, e.g.: quantity of entries.

Cache algorithm used for invalidation/eviction:

Least Recently Used (LRU)

Least Frequently Used (LFU)

FIFO

Eviction percentage

Expiration:

Time-To-Live (TTL)

Absolute/relative time-based expiration

* + **Types**
* Client caching

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

* CDN (Content Delivery Network) caching
* Web server caching

Reverse proxies and caches (Varnish) can serve static and dynamic directly.

* Database caching

Database includes caching in a default configuration.

* Application caching

In-memory caches (Memcached, Redis) are key-value stores between application and data storage.

Cache is in **RAM**, faster than databases (where data is stored on disk). **RAM is more limited than disk**, so for that is used **cache invalidation** algorithms (**LRU** = Least Recently Used).

Redis features:

Persistence option

Built-in data structures such as sorted sets and lists.

Categories of caching: **database queries** and **objects**

Row level

Query-level

Fully-formed serializable objects

Fully-rendered HTML

Note: Avoid using file-based caching because it makes cloning and auto-scaling more difficult.

* + **Cache Access Patterns: When to update the cache**
* 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

A diagram of a person with a blue rectangle

Description automatically generated

e.g.: Memcached

- 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

A diagram of a cloud

Description automatically generated

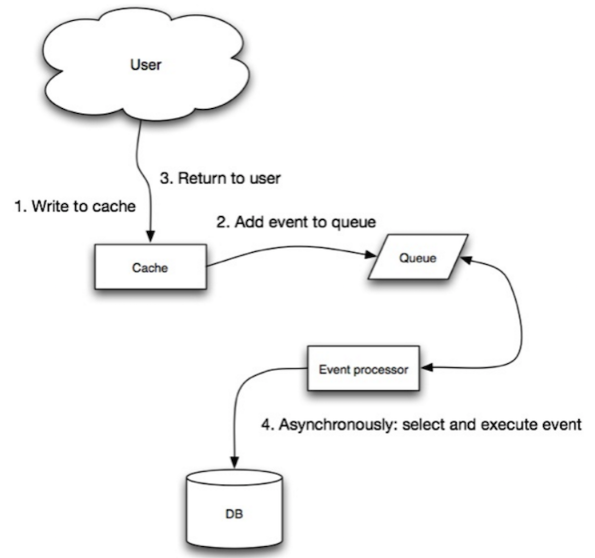
Data in the cache in not stale.

* Write-behind (write-back):

The application does the following:

Add/update entry in cache

Asynchronously write entry to the data store, improving write performance.



* Refresh-ahead:

You can configure the cache to automatically refresh any recently accessed cache prior to its expiration.

A grey hexagon with black text

Description automatically generated

* **Asynchronism**

A diagram of a server

Description automatically generated

* Message queues

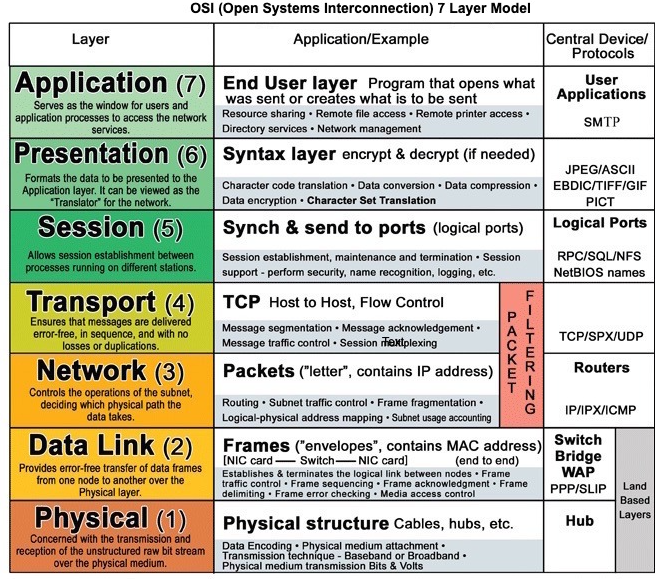
Receive, hold and deliver messages.

e.g.: Redis (Message Broker), RabbitMQ, Amazon SQS

- Task queues

Receive tasks and their related data, runs them, then delivers their results. e.g.: Celery

* Back pressure
* **Communication**



A diagram of a computer network

Description automatically generated

<http://www.escotal.com/osilayer.html>

* HTTP (Hypertext Transfer Protocol):

Is a method for encoding and transporting data between a client and a server.

It is a request/response protocol.

A basic HTTP request consists of a verb (method) and a resource (endpoint).

HTTP verbs: GET, POST, PUT, PATCH, DELETE

HTTP is an application layer protocol relying on lower-level protocols (**TCP** and **UDP**).

* + **Transmission Control Protocol (TCP)**

A close-up of a message

Description automatically generated

Is a connection-oriented protocol over an IP network. Connection is established and terminated using a handshake.

Implements flow control and congestion control.

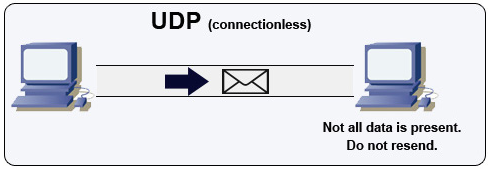
* When use TCP over UDP:

For applications that require **high reliability** but are **less time critical**. e.g.: **web servers**, **database** info, **SMTP**, **FTP**, **SSH**.

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)**



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.

Not support congestion control.

* When use UDP over TCP:

UDP is less reliable but works well in **real time use cases** (VoIP, video chat, streaming, real-time multiplayer games).

You need the lowest latency.

Late data is worse than loss of data

You want to implement your own error correction

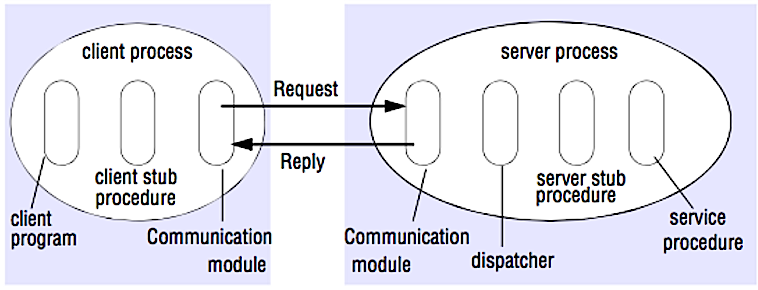
* + **TCP vs UDP**

<https://www.cyberciti.biz/faq/key-differences-between-tcp-and-udp-protocols/>

|  |  |  |
| --- | --- | --- |
| **Factor** | **TCP** | **UDP** |
| Reliability | Is connection-oriented protocol. No corruption while transferring. | Is connectionless protocol. May be corruption while transferring. |
| Ordered | If you send two messages out, you know the first message will get first. | If you send two messages out, you don’t know what order they’ll arrive in. |
| Weight | Heavyweight: because resend requests when exists wrong corruption. | Lightweight: No ordering of messages, no tracking connections. |
| Packets | Streaming: There may be multiple packets per read call. | Datagrams: One packet per one read call. |
| Examples | World Wide Web (Apache TCP port 80), e-mail (SMTP TCP port 25), File Transfer Protocol (FTP port 21), Secure Shell (OpenSSH port 22). | Domain Name System (DNS UDP port 53), streaming media applications (IPTV, movies), Voice over IP (VoIP), Trivial File Transfer Protocol (TFTP), Online Multiplayer games. |

* + **Remote Procedure Call (RPC)**

A client causes a procedure to execute on a different address space, usually a remote server.



* + **Representational State Transfer (REST)**
* 4 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.

**HATEOAS** (HTML interface for HTTP): your web service should be **fully accessible in a browser**.

* + **RPC vs REST**

<https://apihandyman.io/do-you-really-know-why-you-prefer-rest-over-rpc/>

* **Security**

API security checklist

<https://github.com/shieldfy/API-Security-Checklist>

* **References**
* SSD (Solid State Drive): faster than mechanical drives, have no moving parts, cost more money, are smaller
* Round Robin
* **URI vs URL:**

Both are used to identify resources on the internet.

URI (Uniform Resource Identifier): used to identify a resource by location, name or both.

e.g.:

<http://example.com/resource>

urn:isbn:0451450523

URL (Uniform Resource Locator): is a specific type of URI that provides the means to locate a resource on a network. Are a subset of URIs.

e.g.:

<http://example.com/resource>

<ftp://ftp.example.com/file>

* System Design

Basic of System Design

[**https://www.educative.io/blog/system-design-primer**](https://www.educative.io/blog/system-design-primer)

[**https://www.educative.io/blog/complete-guide-to-system-design**](https://www.educative.io/blog/complete-guide-to-system-design)

How to Design a Web Application: Software Architecture 101

<https://www.educative.io/blog/how-to-design-a-web-application-software-architecture-101>

How machine learning gives you an edge in System Design

<https://www.educative.io/blog/machine-learning-edge-system-design>

**The System Design Primer**

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

System Design Interview questions

Object-Oriented Design interview questions

* Containerization and System Design

**Containerization** is the packaging of software code with its dependencies to create a “container” that can run on any infrastructure.

**Docker** is a **containerization platform**, **Kubernetes** is a **containerization software** that allows us to control and manage **containers** and **VMs**. With Kubernetes, you can run Docker containers and manage your containerized applications. Containers are grouped into **pods**, and those pods can be scaled and managed however you want.

[**https://www.educative.io/blog/docker-kubernetes-beginners-guide**](https://www.educative.io/blog/docker-kubernetes-beginners-guide)

* The Cloud and System Design

Cloud computing allows access to services like storage or development platforms on-demand via internet-connected offsite data center.

Cloud system architectures:

Multi-cloud

Hybrid cloud

Single cloud

Public cloud

Private cloud

* Web Hosts

Bluehost

DreamHost

Go Daddy

Host Gator

Pair Networks

* VPSes: Virtual Private Server, is our server

DreamHost

Go Daddy

Host Gator

Linode

Pair Networks

Slicehost

VPSLAND