Distributed Systems | DSy

Zusammenfassung

CONTENTS

1. What are Distributed Systems? . . . 2
1.1. Categorization . . . 2
1.2. Transparency in distributed systems . . . 3
1.3. Fallacies of distributed computing . . . 3

2. Advantages of Distributed Systems . . . 4
2.1. Scaling . . . 4
2.2. Location . . . 5
2.3. Fault Tolerance . . . 7

3. Containers and VMs . . . 8
3.1. Virtual Machine . . . 8
3.2. Container . . . 8
3.3. Comparison . . . 9

4. Docker . . . 10
4.1. Underlying Components . . . 10
4.2. Docker Concepts . . . 11
4.3. Security . . . 12
4.4. Debugging . . . 12

5. Monorepos / Polyrepos . . . 13

6. Load Balancing . . . 14
6.1. Software based load balancing . . . 14
6.2. Traefik . . . 15

7. Web Architecture . . . 15
7.1. Server-side rendering (SSR) . . . 15
7.2. Static site generation (SSG) . . . 15
7.3. Single page application (SPA) . . . 16
7.4. Hydration . . . 16
7.5. Web rendering overview . . . 16
7.6. CORS . . . 17

8. Authentication . . . 17
8.1. Key Concepts . . . 17
8.2. Basic Auth . . . 18
8.3. Digest Auth . . . 18
8.4. Public / Private Key Auth . . . 19
8.5. Let's Encrypt . . . 19
8.6. Session-based Authentication . . . 19
8.7. JSON Web Token (JWT) . . . 19
8.8. OAuth . . . 20

9. Protocols . . . 20
9.1. Layer abstraction . . . 20
9.2. TCP . . . 21
9.3. Transport Layer Security (TLS) . . . 22
9.4. QUIC and HTTP/3 . . . 22
9.5. UDP . . . 23
9.6. SCTP . . . 23
9.7. DDoS Amplification Attack . . . 23
9.8. Protocol Overview . . . 23

10. Application Protocols . . . 24
10.1. Different IDLs . . . 24
10.2. RPC . . . 24
10.3. HTTP . . . 25
10.4. WebSockets . . . 25
10.5. Server-Sent Events (SSE) . . . 25
10.6. Serialization Formats . . . 26
10.7. WebRTC . . . 26
10.8. DNS . . . 27

11. Deployment . . . 29
11.1. Deployment Strategies . . . 29
11.2. Practical Deployment . . . 29
11.3. Ideal Deployment Strategy . . . 31

12. Performance . . . 31
12.1. Throughput vs. Latency . . . 32
12.2. Compression Ratios . . . 32
12.3. Benchmarking . . . 32

13. Bitcoin / Blockchain . . . 33
13.1. Key characteristics . . . 33
13.2. Numbers . . . 33
13.3. Mechanism . . . 34
13.4. Blockchain . . . 35
13.5. Mining . . . 35
13.6. 51% Attack . . . 36
13.7. Coins and Mechanisms . . . 36
13.8. Discussion . . . 36

1. WHAT ARE DISTRIBUTED SYSTEMS?

A distributed system in its simplest definition is a ***group of computers*** working together as to ***appear as a single computer*** to the user.

1.1. CATEGORIZATION

- ***Tightly coupled*** *(The processing elements have access to a common memory)* vs. ***loosely coupled*** *(No shared memory)*
- ***Homogeneous*** *(all processors are of the same type)* vs. ***heterogeneous*** *(contains processors of different types, more common)*
- ***Small-scale system*** *(Web app + database on the same system)* vs. ***large-scale system*** *(with more than 2 machines)*
- ***Decentralized*** *(e.g. Blockchain, distributed in the technical sense, but not owned by one actor)* vs. ***distributed*** *(owned by one actor)*

1.1.1. CAP theorem

States that a distributed data store ***cannot simultaneously be consistent, available and partition tolerant***.

- ***Consistency:*** Every node has the same consistent state & data
- ***Availability:*** Every non-failing node always returns a response
- ***Partition tolerant:*** The system continues to be consistent even when network partitions *(not all nodes reachable due to network outages)*

You can only have two. Since ***network partition*** needs to be a given with distributed system, you need to choose between availability and consistency. Is a system ***AP*** or ***CP***?

- ***Eventual Consistency:*** Consistency is eventually restored when the partition ends *(e.g. DNS server)*
  + Explicitly allows data partitions on different nodes
  + Allows high availability and high performance, because nodes can confirm operations locally and don't need to wait for synchronization with other nodes.
  – Developers need to account for partitions

- ***Strong Consistency:*** Every node is consistent all the time, everything gets run as if it was only executed on one machine.
  + Easiest mental model
  – Expensive in terms of performance and latency
  – Can hinder availability during network partitions

- ***Casual Consistency:*** Mix of eventual and strong consistency. Guarantees that operations that depend on each other *(reading and incrementing a value)* will be seen in the correct order, but independent operations can be seen in different orders.

The CAP theorem is often criticized for being ***too simplistic***, as most of the time there isn't an "either-or" between consistency and availability, there are many levels in between. But it offers a ***good conceptual framing***.

1.1.2. Other Categorizations

There are more classifications.

- ***Classifications based on architecture:*** Client-server, peer-to-peer *(P2P)*, hybrid and more
- ***Classifications based on software architecture:*** Layered, object-based, microservices, event-based, data-centric
- ***Classifications based on communication model:*** Synchronous or asynchronous
- ***Other Classification:*** Degree of transparency *(The degree to which the distribution is concealed from the user)*, fault tolerance, scalability, degree of consistency, data replication *(how and where data copies are stored)*, data partitioning, heterogeneity.

There is ***no universally applicable categorization*** of distributed systems.

"Controlled" distributed systems	***"Fully" decentralized systems***
1 responsible organization	N responsible organizations
low churn *(low fluctuation of the participating nodes, planned)*	high churn *(high fluctuation)*
Examples: Amazon DynamoDB, Client/server	**Examples:** BitTorrent, Blockchain, E-Mail
Secure environment *(operator has control)*	Hostile environment *(some users are probably malicious)*
High availability *(predictable)*	Unpredictable availability *(depends on the behavior of many)*
Can be homogeneous or heterogeneous	Is heterogeneous
Mechanisms that work well: Consistent hashing, master nodes, central coordinator	**Mechanisms that work well:** Consistent hashing with distributed hash tables, flooding / broadcasting
Network is under control or client/server – no NAT issues	NAT and direct connectivity is a huge problem because of different network environments
Consistency: Leader election *(leader is responsible for the sequence of state changes and must therefore always be available.)*	**Consistency:** Weak consistency *(DHTs have little consistency over order)*, Nakamoto consensus *(proof of work)*, proof of stake – leader election, PBFT protocols *(Practical Byzantine Fault Tolerance)*
Replication principles: More replicas: higher availability, higher reliability, higher performance, better scalability, but: requires maintaining consistency in replicas	
Transparency principles: apply	

1.2. TRANSPARENCY IN DISTRIBUTED SYSTEMS

Distributed systems should ***hide*** their distributed nature.

- ***Location transparency:*** Users should not be aware of physical location
- ***Access transparency:*** Users should access resources in a single, uniform way
- ***Migration/relocation transparency:*** Users should not be aware that resources have moved
- ***Replication transparency:*** Users should not be aware about replicas, it should appear as a single resource
- ***Concurrent transparency:*** Users should not be aware of other users *(For concurrent access, users should not accidentally modify data of another user – it should seem as if everyone has exclusive access. Exceptions: Online Status on Messengers)*
- ***Failure transparency:*** Users should not be aware of recovery mechanisms
- ***Security transparency:*** Users should be minimally aware of security mechanisms *(only e.g. login at the beginning)*

1.3. FALLACIES OF DISTRIBUTED COMPUTING

- ***The network is reliable:*** Submarine cables break a lot
- ***Latency is zero:*** Ping to Australia is ~ 300ms
- ***Bandwidth is infinite:*** 1 GBit/s means that 1TB needs 2h and 16minutes to transfer
- ***The network is secure:*** Assume someone is listening.
- ***Topology doesn't change:*** Request can take different route than reply
- ***There is one administrator:*** Sometimes your route goes from one company to another rival company *(UPC, Init7)*
- ***Transport cost is zero:*** Someone builds and maintains the network
- ***The network is homogeneous:*** From fiber to WiFi cable, server, desktop, mobile – extremely heterogeneous

2. ADVANTAGES OF DISTRIBUTED SYSTEMS

Distribution ***adds complexity*** which should generally be ***avoided***. But sometimes, it is necessary.

1. ***Scaling*** *(if one machine is not enough)*
2. ***Location*** *(to move closer to the user)*
3. ***Fault-tolerance*** *(Hardware will fail eventually)*

2.1. SCALING

There are two different ways of scaling:

2.1.1. Vertical Scaling: Better Hardware

Vertical scaling is the process of ***increasing the power of a single machine*** by buying better hardware *(Scale one machine up)*.

Advantages	*Disadvantages*
+ Lower cost with small scale + No adaption of software required + Less complexity + Faster machine for the same price as the machine from some years ago	– Higher cost with massive scale *(Slightly better hardware costs a lot more in the upper tiers)* – Hardware limits for scaling – Risk of hardware failure causing outage – More difficult to add fault-tolerance

Vertical Scaling Performance

Moore's Law is the observation that the number of transistors in an integrated circuit (IC) doubles about every two years *(Other predictions: doubling chip performance every 18 months)*.

Moore's law is based on observation of historical trends which cannot continue forever, because we are bound by the size of atoms. It might be dead already in 2025.

Nielsen's Law states that high end Users' bandwidth grows by 50% per year. In other words, the amount of available bandwidth to individual users roughly doubles every 1.5 years.

The bandwidth grows slower than computer power because telecoms companies are conservative and most users are reluctant to spend much money on bandwidth because current connections are fast enough for them. Therefore, you should ***optimize for bandwidth, not for CPU***.

Kryder's Law states that the density of information on hard drives doubles every 13 months or in other words increases by a factor of 1,000 every 10.5 years.

User behavior changed in the time of SSDs, ***speed is more important*** than raw storage size. Hardware today is already very fast. For small and simple applications, vertical scaling is the way to go. There is also less demand for local storage due to cloud storage.

Wirth's Law states that software is getting slower more rapidly than hardware is becoming faster.

Software ***complexity and resource demands*** grow faster than the hardware speed. As a result, software often ***feels slower*** despite running on much faster hardware. ***Example:*** Modern webpages often utilize more resources than the original DOOM from 1993.

Example: Let's Encrypt

Let's Encrypt, the service providing free TLS certificate for HTTPS connections ***runs all requests against a single MariaDB to minimize complexity***. Although some read operations are delegated to mirrors, write operations are executed by the main database. After ***upgrading their main server***, it runs at 25% CPU utilization compared to the previous 90%.

2.1.2. Horizontal Scaling: More machines

Horizontal Scaling is the process of *adding more machines* to a resource pool in a system to distribute the workload *(Scale out to multiple machines)*.

Advantages	*Disadvantages*
+ Lower cost with massive scale + Easier to add fault-tolerance *(e.g. no server failure)* + Higher availability	– Higher initial cost *(due to initial synchronization effort)* – Adaption of software required *(state synchronization)* – More complex system, more components involved

Machine Learning at the moment uses *horizontal scaling*, because vertical scaling is not yet feasible due to *hardware limitations* *(Mainboards don't support the amount of RAM needed in a single machine)*.

2.2. LOCATION

Latency is the time a signal needs to travel from source to destination and back, the *round-trip-time* *(RTT)*. Everything gets faster, but latency stays because nothing is faster than the speed of light *($\sim 300'000$ km/s)*.

In a *perfect*, direct vacuum light tube to Sydney, the RTT would be:

$$\frac{\overbrace{16'540 \text{ km}}^{\text{distance}}}{\underbrace{300'000 \text{ km/s}}_{\text{speed of light}}} \cdot \overbrace{2}^{\text{both ways}} = 0.110\text{s} = 110\text{ms}$$

In practice, it is more like ~ 298 ms.

With *Starlink*, under perfect conditions, optimal location, no processing delay, no handoff between satellites, the theoretical latency is around 7.3ms. In practice, it is around $20 - 60$ms.

There are two main factors that reduce the speed from the calculated perfect time.

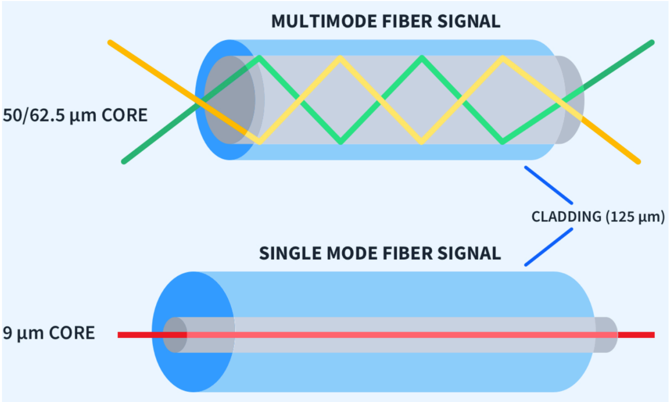
- *No direct path:* The cables are not laid out in a straight line. The distance to Sydney via cable is around $24'000$km.
- *Signal is slower than speed of light:* In fiber glass, the signals travel about $2/3$ of the speed of light ($200'000$km/s).
- *Other reasons:* Non-optimal routing, queuing delays, routing delays, traffic inspection, signal repeating and protocol overhead.

$$\frac{\overbrace{24'000 \text{ km}}^{\text{distance}}}{\underbrace{200'000 \text{ km/s}}_{\text{speed of signal}}} \cdot \overbrace{2}^{\text{both ways}} = 0.240\text{s} = 240\text{ms} \underbrace{+ \sim 50 - 60\text{ms}}_{\text{other reasons for delay}} = 298\text{ms}$$

2.2.1. Transmission Modes

Multimode fibers transmit *multiple rays of light* simultaneously. They are used to transmit data over *short to medium ranges* and have a high bandwidth for short distances. Multimode fibers are *cheap and easy to install* but *perform badly over long distances*.

Singlemode fibers provide lower latency than multimode fibers because they have a *far smaller core size*. This *reduces signal loss* by reducing bouncing of the signal. They are *more expensive* than multimode fibers, but have a much *larger transmission distance* *(up to 100km without signal regeneration)*. They are often used in undersea and data center communication and other *critical infrastructure*.



Hollow-Core fibers combine ***low latency***, ***high bandwidth*** and ***low loss***. Light travels 50% faster than in solid core fibers because it travels through air instead of solid glass and is guided and restricted by a reflective cladding.

Copper propagates ***faster than fiber***, but not much. It is also ***not usable*** over long distances.

Satellites have ***direct connection*** and signal travels ***almost at speed of light***. But the distance is often bigger and ***weather conditions*** affect the signal strength. There is also ***protocol overhead***, ***network processing***, ***encoding/decoding*** and ***queuing*** which also add to the latency. Geostationary satellites with a higher altitude than Starlink uses have a latency of about 477ms.

Wi-Fi could ***in theory have the lowest latency*** due to the same advantages of satellites. But it also has a ***lot of delays***: CSMA/CA *(collision avoidance)*, wait times before transmission, acknowledgement packages, retransmissions, signal processing at receiver and transmitter, MAC layer procession, protocol stack traversal, DCF backoff, channel busy waiting… In the end, it adds up to ***additional*** 5ms ***of latency***.

Media	***Velocity Factor***	***Description***
Vacuum	100%	Vacuum or free space
Thick coaxial cable	77%	Originally used for ethernet, referred to as “thicknet”
Optical fiber	67%	Silica waveguide used to transport optical energy
Thin coaxial cable	65%	Referred to as ethernet “thinnet” or “cheapernet”
Unshielded twisted pair	59%	Multipaired copper cabling used for LAN and telecom applications
Microstrip	57%	PCB trace on FR4 dielectric, $\mu r = 3.046$ *(PCB = Printed Circuit Board)*
Stripline	47%	PCB trace in FR4 dielectric, $\mu r = 4.6$

2.2.2. Importance of Latency

If a website is slower, less people visit. Google for example measured a 20% drop in traffic after the latency was 500ms higher. In gaming, if a game lags 300ms, it is unplayable. Even a 50ms latency reduces the performance noticeably. Human reaction time is around 200ms. Depending on the device, the time from the key press to display can be $70\text{ms} - 150\text{ms}$. Keyboards have a delay of $15 - 60\text{ms}$ due to key travel time and USB polling adding $\sim 8\text{ms}$ *(although gaming keyboards can reach* 1ms *polling)*. A standard monitor with 60hz display rate also has a delay of 8ms. The input lag for tablet pens is around $20 - 80\text{ms}$. To reduce these factors, competitive gamers often use a low latency keyboard and mouse and a monitor with 240hz.

2.2.3. Reducing Latency

With a distributed system, the latency can be reduced by ***placing services closer to the users***. This also ***increases the bandwidth*** *(local networks are often better developed)* and ***improves reliability*** and ***availability*** *(less hops)*. The downside is that data replication and caching are ***more complicated*** to coordinate.

An example for this is ***CDN*** *(Content delivery network)*. Places your images, sites and scripts close to your users.

2.2.4. Bandwidth

Another important factor is bandwidth, or ***how much data can be transmitted per second over a certain medium***. The current world record belongs to an experimental fiber optic cable that can transmit $\sim 23\text{Pb/s}$. NASA also experimented with transmitting data via laser into space and reached $\sim 200\text{Gb/s}$, while Starlink can exchange $\sim 100\text{Gb/s}$ between its satellites with multiroute.

2.3. FAULT TOLERANCE

Any hardware will crash eventually. Failures are not a question of "if", but "when".

2.3.1. Random Bit Flips

There are ***random bit flips*** due to bad pin connections, incorrect RAM timings, clock issues, design flaws or cosmic rays *(A bit flip added 4096 votes to a candidate in a Belgian election. The candidate had more votes than were possible)*.

Influencing Factors are the ***sensitivity*** of each transistor, ***number*** of transistors on the microchip, ***altitude***, … *(The Cassini mars rover has 280 bitflips daily, during a solar proton storm even up to 890 on its 300MB TMR RAM)*

Error-correcting code memory *(ECC)* uses ***TMR*** *(Triple modular redundancy - every memory operation is triple validated)* or ***Hamming Code*** to detect or additionally correct bitflips. Hamming Code can only correct one bitflip and detect two. If there are more, the error correction fails. Bitflips are more likely than you may think: The Jaguar supercomputer with 360TB ECC RAM had a double bitflip every 24 hours *(can be detected, but not corrected)*. Uncorrected bitflips can lead to `SEGFAULT`s or even OS crashes.
ECC is currently used mostly ***for servers, not for customer products***, although the current DDR5 RAM standard has ECC built in.

HDDs can break and SSDs wear out with time. An SSD consists of ***NAND cells*** *(Flash memory that can store data without using power)* with a ***limited lifetime***. It has ***spare*** NAND that are used when cells break.

There are different kinds of NAND-Types:

	SLC **(single level cell)**	***MLC*** **(multi level cell)**	***TLC*** **(triple level cell)**	***QLC*** **(quad level cell)**
Bits per cell	1	2	3	4
P/E-Cycles *(Program-Erase Cycles)*	100′000	10′000	3′000	1′000

SLC has the highest lifetime, but limited capacity and costs the most. It is therefore often used for ***caching*** files / cells that are frequently accessed. Rarely used data should be stored on MLC/TLC/QLC drives for cost efficiency.

Wear Leveling

Distribute write and erase operations across all memory cells. This technique is used to prolong the life of memory.

Bitsquatting

Specific type of ***DNS Hijacking.*** If a user wants to visit `example.com` and a bitflip occurs, he might land on `uxample.com`. Some malicious user could register a domain with a single bit error and hope that somebody lands on it by accident.

2.3.2. Network Outages

There is also always the possibility of network outages *(i.e. damaged undersea cables or a ISP messing up BGP again)*. If your system is distributed, the likelihood of this affecting you strongly is much smaller.

2.3.3. Conclusion

It is not a question of "if" hardware will fail, but "when". Multiple machines provide ***redundancy***. When one machine fails, others can take over. The ***load*** can be ***redistributed*** among the remaining machines and systems can ***continue*** to function despite individual failures.

But distributed systems also ***add complexity***, this strategy is therefore a ***tradeoff***. Use only when benefits outweigh the added complexity.

3. CONTAINERS AND VMS

3.1. VIRTUAL MACHINE

Virtualization is the act of creating a virtual machine that ***acts like a real computer*** with an operating system.

- ***Host Machine:*** The machine the virtualization software runs on
- ***Guest Machine:*** The virtual machine itself

Hypervisor

Software that runs the virtual machine. There are two types:
- ***Type 1:*** bare-metal, hypervisor acts as the operating system running on the machine *(e.g. Xen, Proxmox)*
- ***Type 2:*** Hosted, the hypervisor runs as a regular program on an operating system *(e.g. VirtualBox)*

On newer processors, unmodified OS can run as guests with virtualization extensions like Intel VT-X or AMD-V enabled. If these are not present or disabled, a guest OS can be modified to communicate with the hypervisor instead of the hardware directly; it runs ***paravirtualized***. But ***VMs should not access memory directly***, only through the hypervisor. Otherwise this would be considered a sandbox escape, because the guest could modify unrelated files on the host.

The guest machine needs to have the ***same architecture*** as the host, else an ***emulator*** is needed *(i.e. game consoles or desktop OS for ARM/RISC-V processors)*. There are hardware-specific *(Snes9x, Dolphin, PCSX2)* and generic emulators *(QEMU)*.

Virtual Machines can be ***expensive***. New providers tend to be cheap at first, but become more expensive over time. ***Switching*** to a different provider can be very ***complicated***. This is why it is important to ***compare*** all providers carefully before making a decision.

MicroVM

Alternative to normal VMs, purpose-built for ***serverless workloads***. Since everything else is removed, this results in a ***lightweight*** virtual machine with near-instant startup times and low resource consumption. Examples are Firecracker or microvm.

Virtual Desktop Infrastructure (VDI)

VDI ***provides virtual machines over the network***. Users can access them from any endpoint and all processing is done on the server hosting the VDI. The client only ***sends input commands*** and receives the image of the virtual machine, not unlike the terminal systems in the 1970ies.

3.1.1. Pricing types on cloud providers

- ***On-Demand:*** Pay per machine, used network traffic and IP address. Billing is done per started hour of usage.
- ***Reserved:*** Pay to use VM for a certain time frame *(i.e. one year)*
- ***Spot Instances:*** Runs on unused instances of the cloud provider. Cheaper than on-demand, but can be taken back by the provider if other customers need the capacity

Comparing different cloud providers is ***difficult***, as they offer slightly different services. It also may be difficult to move vendors if you depend heavily on the specific features a provider offers.

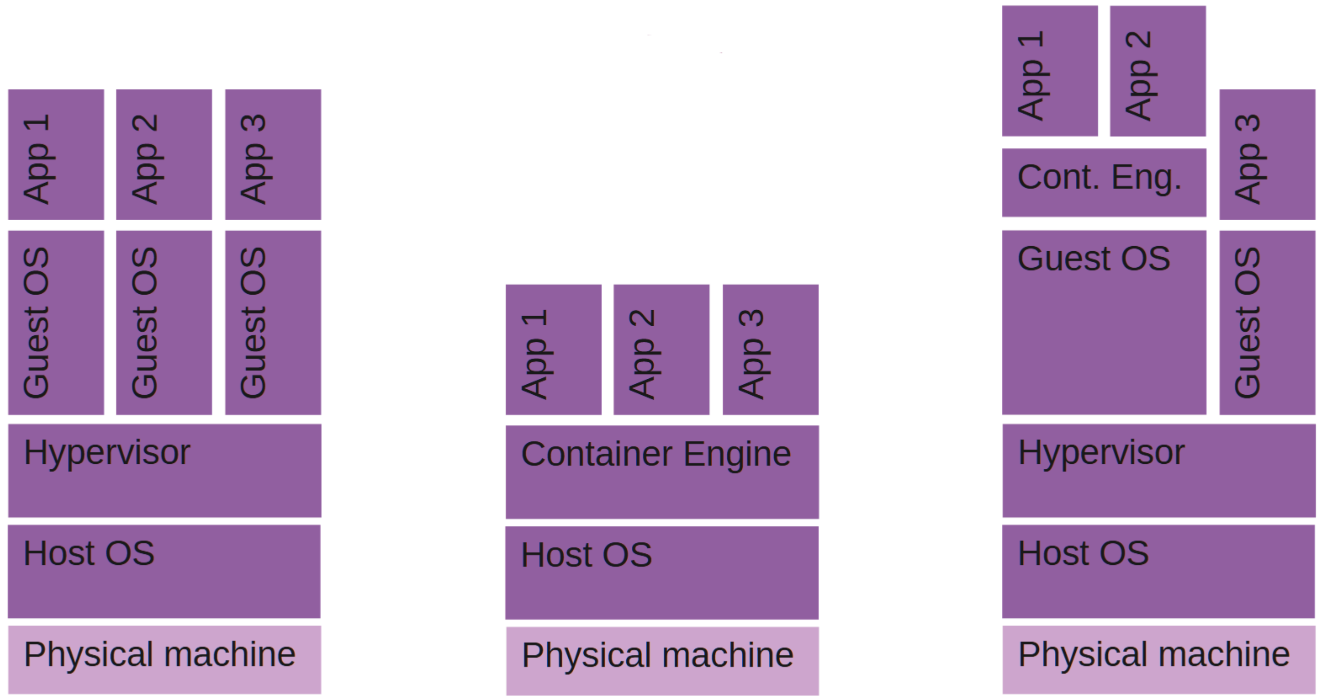
3.2. CONTAINER

Containers are ***isolated*** user-space instances. The OS needs to provide support for them, because they share the kernel of the host provides resources – there is no guest OS.

There are a lot of different container softwares. The most well known is Docker, but there are also others:
- ***LXC:*** Lower abstraction level and direct use of Linux kernel features
- ***systemd-nspawn:*** Part of the systemd project, minimalist container manager
- ***Solaris Zones:*** Oracle/Sun-specific container technology
- ***Linux-VServer:*** Kernel patch for Linux, older virtualization technology
- ***OpenVz:*** Operating system-level virtualization, popular hosting tool
- ***rkt:*** CoreOS developed alternative to Docker, focus on security
- ***Podman:*** Has a daemon-less architecture and rootless containers. Since the aim of podman is to be a replacement for Docker, it is compatible with Docker compose files and uses similar workflows and commands.

There are also ***application-level containers*** that sandbox on the application level. These are often more geared towards graphical applications instead of the CLI/Service applications of container. The most widely used is ***Bubblewrap***, powering sandboxed ***Flatpak*** applications on Linux. Others are syd, Firejail, GVisor and minijail.

3.3. COMPARISON



- Virtual machines
- Container
- Both

Container	***Virtual Machine***
+ Reduced size of snapshots *(2MB vs 45MB)* + Quicker spinning up of apps + Available memory is shared *(efficient resource usage)* + Process-based isolation *(share same kernel, reducing overhead)*	+ App can access all OS resources + Live migrations *(move VM to another host without interruption)* + Pre-allocates memory *(offers planning security)* + Full isolation *(better security)*
– Available memory is shared *(Containers can affect others with high RAM usage)* – Process-based isolation *(less secure isolation than VMs)*	– Pre-allocates memory *(inefficient resource usage)* – Full isolation *(more complex setup for certain scenarios)* – Relatively long startup time
Use case: complex application setup, with container less complex configuration	**Use case:** better hardware utilization / resource sharing

Using both at the same time: Complete isolation in VMs, but still profit from the efficiency of containers.

4. DOCKER

Docker is a ***containerization platform*** which packages software into containers. These ***"images"*** can be viewed on Docker Hub. The containers are ***isolated*** from each other and communicate over well-defined channels.

The apps running inside the containers need to be compatible with the host OS.

Docker Commands

- *`docker run <image-name>`* fetches the specified image from Docker Hub *(container image repository)*. If no version is provided, it uses the latest available version.
- *`docker save <image-name> -o image.tar`:* Save image as a `tar` file
- *`tar xf image.tar`:* Extract `tar`
- *`docker images`:* See installed images
- *`docker rmi <image-name-or-id>`:* Delete image
- *`docker ps -a`:* List all running and exited docker Containers
- *`docker rm <container-name-or-id>`:* Delete container
- *`docker build . -t <image-name>`:* Create and build image on basis of `./dockerfile`
- *`docker system prune -a`:* Remove everything
- *`docker exec -it <container-name-or-id> sh`:* Run shell in container

Docker can also be used over a GUI like Docker Desktop.

4.1. UNDERLYING COMPONENTS

Docker itself is built on preexisting Linux components, this is why Docker itself can be implemented in just around 100 lines of Bash, as shown in the "Bocker" project.
This chapter explains these concepts, but ***Docker handles the configuration*** of these tools itself, so you don't need to manually fiddle with them.

4.1.1. OverlayFS

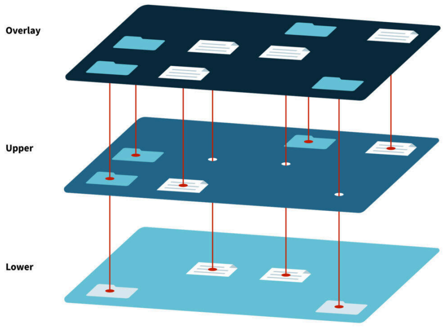
OverlayFS is a virtual file system that combines multiple Linux mount points into one and creates a ***"Union Mount Point"*** to make it seem like contents from different file systems are actually in one directory. It does this by ***layering*** the file systems on top of each other. OverlayFS is the basis for Dockers ***container isolation***.

When working with it, the first layer is usually referred to as ***"upper"*** and normally ***writable***. The next layer, referred to as ***"lower"*** is ***read-only*** most of the time *(or another overlay)*. The ***workdir*** is used to prepare files as they are switched between the layers.

File access	***upper*** **(writable)**	***lower*** **(read-only)**
Read file	Accessed first	Accessed if file doesn't exist in upper
Write file	As normal	Copy is created in the overlay and modified there
Delete file	As normal	Marked as deleted in overlay, still exists in lower

Example: The ***lower directory*** can be ***read-only*** or could be an overlay itself. The ***upper directory*** is normally ***writable***. The ***workdir*** is used to prepare files as they are switched between the layers.

```
cd /tmp
mkdir lower upper workdir overlay
sudo mount -t overlay -o \
lowerdir=/tmp/lower, upperdir=/tmp/upper, \
workdir=/tmp/workdir/ none /tmp/overlay
# The source device is "none", because the FS doesn't
# originate from a block device like /dev/sda
```



4.1.2. Cgroups

Cgroups are ***control groups***. They define limits, isolation, prioritization of CPU, memory, disk I/0, network. They can be used for processes or for docker containers. Docker uses cgroups behind the scenes. Windows and MacOS don't support cgroups, but Docker just uses a lightweight Linux VM to run its containers on these OS.

Example: Create a group to limit CPU usage to 20%.

```
# cgroups with regular processes
sudo cgcreate -g cpu:red # Create group for CPU controller
echo -n "20" > /sys/fs/cgroup/red/cpu.weight
cgexec -g cpu:red bash # Start bash with group
# In newly created bash shell
taskset -0 sha256sum /dev/urandom # Runs on core 0
```

```
# cgroups with Docker
docker run \
  --name="low prio" \
  --cpu-shares="20" \
  --cpuset-cpus="0" \
  alpine sha256sum /dev/urandom
```

If another group with 80% would be created and ran on the same core, the usage will be around the specified limit.

4.1.3. Linux Network Namespaces

Provides ***isolation*** on the system resources associated with networking. Docker uses it so each container can run its own network ***without conflicts***. Management works via the regular OS networking tools. Network namespaces can ***create virtual Ethernet connections***, ***configure their networks*** and ***run programs in isolated networks***. Per default, Docker containers on the same network ***can communicate*** with each other. For communication ***outside*** of the internal network, further configuration is required by setting up routes in the hosts routing table and enabling NAT forwarding to the virtual address.

```
ip netns add testnet # Create new namespace
ip link add veth0 type veth peer name veth1 netns testnet # Create new adapter in NS & link them
ip addr add 10.1.1.1/24 dev veth0 # Set IP on host adapter
ip netns exec testnet ip addr add 10.1.1.2/24 dev veth1 # Set IP on guest adapter
ip link set dev veth0 up # Start host adapter
ip netns exec testnet ip link set dev veth1 up # Start guest adapter

echo 1 > /proc/sys/net/ipv4/ip_forward # Enable IPv4 forwarding on the host
ip netns exec testnet ip route add default via 10.1.1.1 dev veth1 # Create new route
iptables -t nat -A POSTROUTING -s 10.1.1.0/24 -o <real-network-dev> -j MASQUERADE # Setup NAT
ip netns exec testnet nc -l -p 8000 # Run netcat inside the namespace
```

Hole punching

Hole punching is a technique for ***establishing a direct connection*** between two parties in which one or both are ***behind firewalls*** or routers with ***NAT***.

This is done by connecting to an ***unrestricted rendezvous server*** which temporarily stores external and internal address and port information for each client and relays that data to each of the clients, so they can establish a direct connection. All further traffic can then be sent without the rendezvous server.

4.2. DOCKER CONCEPTS

4.2.1. Layers

A ***new layer*** is added for ***each instruction*** in the Dockerfile. The layers are ***cached*** and will not be recreated unless the input changes. Usually when you create a new Docker image, you will start from a ***base layer*** that provides the OS for your container, like the `alpine` image and start creating new layers from there.

4.2.2. Dockerfile

Build your binary or create your own image with a ***dockerfile***.
The most ***basic workflow*** consists of…

- picking a base image with ***FROM***,
- setting the working directory with ***WORKDIR***,
- using ***COPY*** to copying your application and configurations to that working directory
- setting the ***ENTRYPOINT*** to start that application.

```
FROM alpine
WORKDIR /scripts
COPY hello.sh .
ENTRYPOINT ["sh", "hello.sh"]
```

hello.sh

```
#!/bin/sh
echo "Hello"
```

You can also create ***additional containers*** just for building the application you want to dockerize. The Docker image can then be built with `docker build` and ran with `docker run`. The images should be kept ***small***. You can use multi-stage builds *(use `FROM` to split your dockerfile into different stages)* and then copy the required files from one stage to the next. This fixes problems with Dockers aggressive caching.

4.2.3. Docker Compose

With a Docker Compose file, you can more easily ***deploy multiple containers*** *(i.e. your service, load balancer, DB etc.)*. You can specify the dockerfile(s) to be built and started and configure additional things like networking or specify dependencies between containers *(i.e. start backend only after DB is running)*. It is also possible to override settings from the dockerfiles. The compose file is written in `yml` syntax and serves as ***lightweight*** orchestration for Docker.

```
# docker-compose.yml
services:
  server1: # The server container
    build: . # Dockerfile for server is in build directory
    ports:
      - "80:25565"

  client: # The client container
    image: alpine
    entrypoint: sh -c "echo hallo"
```

```
# Dockerfile - compile server
FROM golang:alpine AS builder
WORKDIR /build
COPY server.go .
RUN go build server.go
# Run the compiled server
FROM alpine
WORKDIR /app
COPY --from=builder /build/server .
ENTRYPOINT ["./server"]
```

For ***services***, it is often necessary to ***allow certain ports to connect*** to your Docker container. The allowed ports can be ***different*** from ***inside*** and ***outside*** the container. In the example above, the host can make a request to port 80 and the program in the container will receive it on port 25565. Specifying ports like this is only necessary to either ***avoid port conflicts*** on the host or for ***outside connections***: Containers in the same network can communicate among themselves over all ports by default *(The `client` container could directly send a request to port 25565 without this rule, but the host could not)*.

4.3. SECURITY

- Keep images ***small*** and ***up to date***. This reduces attack surface.
- Use ***tiny runtimes*** like alpine or even go distroless. Avoid big images like Ubuntu / Arch / Debian.
- ***Check*** your image for ***vulnerabilities*** *(e.g. with `snyk` or `trivy`)*
- Do ***not expose*** the Docker daemon socket even to the containers, as it allows wide reaching permissions
- ***Set a user***, do not run as `root` *(Reduces risk in case container is compromised)*
- ***Limit capabilities*** *(grant only specific capabilities needed by a container)*
- ***Disable inter-container communication*** if not specifically required in production
- ***Limit resources*** *(memory, CPU, file descriptors, processes, restarts)*
- Set filesystem and volumes to ***read-only*** *(Limits malicious or accidental changes to critical files)*
- ***Lint the Dockerfile*** at build time *(i.e. with an IDE to check for insecure configurations)*
- Run Docker in ***root-less mode*** *(Podman is better at this)*

4.3.1. Logging

- Set the logging level to ***at least INFO*** and set the log level per environment. *(There is TRACE – only use during development, DEBUG – logs about anything that happens in the program, INFO – log all actions that are user-driven or system specific, NOTICE – should be used in production, WARN – logs everything that could become an error, ERROR – logs errors, FATAL – doomsday)*
- Use ***logging libraries***
- Use ***meaningful*** messages
- Log in ***JSON*** – structured logging. Keep the log structure consistent. *(allows automated analysis of logs)*
- Avoid logging ***sensitive information***
- ***Secret Management:*** Use SaaS solutions like GitHub Secrets or AWS Secrets.

4.4. DEBUGGING

- ***What is my base image?*** Has consequences on resource usage and configuration. Larger base images are usually more convenient to configure, but have a larger attack surface
- ***What is my C STD Library?*** Tools compiled against `glibc` don't work on systems with only `musl` installed

- Alpine has ***tools for debugging*** like the `busybox` or `toybox` tool suites
- ***Remove everything*** and do a clean start with `docker system prune -a`
- ***What is going on in my container?*** Connect to it with `docker exec -it <id> sh` or use `nc` to connect to ports or `wget` to request HTTP resources if your container serves those
- Try to ***ping*** another container with `ping containername`
- ***Is my service bound to `localhost`?*** Then it cannot be accessed outside Docker *(check with `netstat -lnt` on the container)*
- Check ***logfiles*** and `docker stats` *(shows resource usage)*
- Is Docker application ***Docker aware***? *(Docker memory limits are not hard limits, if over limit, the application restarts)*
- To ***reduce performance impacts***, use multi-stage dockerfiles and `.dockerignore` to exclude files not needed inside the container *(i.e. `node_modules`)*

4.4.1. Pitfalls / Supply chain attacks

Docker uses ***aggressive caching***, i.e. files downloaded with `wget` in Dockerfiles are cached. Use links with ***version numbers instead of `latest`*** *(ie. https://example.com/package-1.2.0.zip)*. Even better: use the `ADD` keyword in your dockerfile to download resources instead of `wget`.

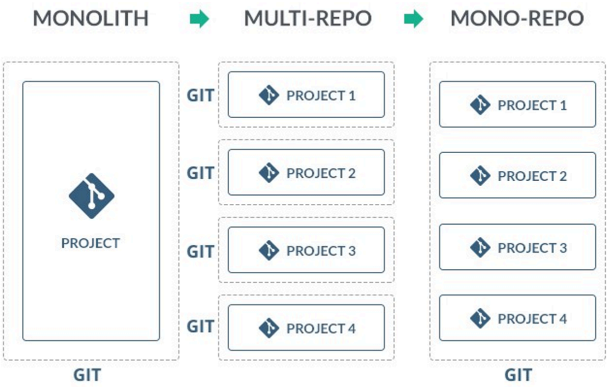
Supply chain attacks happen often. One of your dependencies could be modified to include malicious code. Because of dependency hell, you could be affected without using a unsafe library directly yourself *(nested dependencies)*. ***Mitigation Strategies*** are dependency scanning tools, or not executing dependencies on your own machine but in a container.

5. MONOREPOS / POLYREPOS

General Rules about Project Setup:

1. Be consistent
2. Other projects always prefer other structures
3. A perfect structure does not exist
4. Try to follow best practices for your project, but deviate where it makes sense.

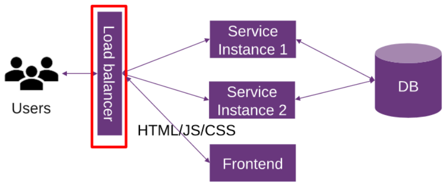
Monolith	***Polyrepo***	***Monorepo***
Only one code base which is managed by only one Git repo. + Easy to handle – Bad usability in big projects	Multiple repositories for a project. Sync via git submodules or via bash script. Also called manyrepo or multirepo. Creates loose coupling of projects. + Flexible, scalable, reliable, loose coupling, fine grained access control + Encourages code sharing across organizations – Hard to share resources – Complex to work with	One repository to rule them all. 1 sub-directory with frontend, 1 sub-directory with backend. Creates tight coupling of projects. + easy to share resources, common dependencies only need to be installed once, independent and flexible + Encourages code sharing within organization – slow to pull, tight coupling, hard to grant limited access



In bigger projects, ***split up*** backend and frontend into separate repositories. In smaller projects, use a monorepo.

6. LOAD BALANCING

A load balancer ***distributes workloads*** *(requests)* across ***multiple computing resources*** *(machines)*. This is an example of ***horizontal scaling***. It ensures high ***availability*** and ***reliability*** by sending requests only to servers that are online. It also ***provides the flexibility*** to add or subtract servers as demand dictates.



There are three types of load balancers:

- ***Hardware load balancer:*** A metal box with specialized processors and often uses proprietary software. This is less generic, but has better performance. Only usable if you control your own datacenter *(Examples: loadbalancer.org, Cisco)*.
- ***Software load balancer:*** There are different load balancers that function on different levels of the OSI model. Seesaw runs on Layer 2/3 *(data link / network layer)* , LoadMaster, HAProxy, ZEVENET, Neutrino, Nginx, Gobetween and Traefik run on Layer 4 *(transport layer)* and Layer 7 *(application layer)*.
- ***Cloud-based load balancers:*** Pay for use. They often also provide predictive analytics and depending on the product also operate on different OSI layers *(Examples are AWS, Google Cloud, Cloudflare, DigitalOcean and Azure)*

A layer 7 load balancer is ***more resource intensive*** than a layer 4 because they need to terminate HTTP and TLS, analyze the packets and then create new connections to the destination based on the packets data.

6.1. SOFTWARE BASED LOAD BALANCING

6.1.1. Load Balancing Algorithms

- ***Round robin:*** Distribute all available server addresses equally by looping sequentially. This is a very simple algorithm, often the default. It might drop requests on congested nodes *(use only with stateless services)*.
- ***Weighted round robin:*** More powerful machines get more work. But high variance in server load may drop requests.
- ***Least connections:*** The machine with the fewest requests gets the new request. Needs to keep track of outstanding requests. Not the best for latency.
- ***Peak exponentially weighted moving average:*** Considers latency, complexity increases.
- ***Others:*** ip_hash, least_time, random, uri_hash, cookie

Stateless and Stateful

Don't store anything in the service that can only be accessed on that server – this is a stateful service. If you do, you need a ***sticky session*** that always sends a user to the same machine. Makes fault tolerance more complex and it makes scaling near impossible.

If you have to store something about the user *(i.e. contents of shopping basket)*, either store it in a database accessible from every service or store the information on the user-side via cookie or similar. The user then has to provide that cookie with every request. This service is ***stateless*** because the service itself does not store that information and the user can request the same information from every other instance of that service.

Health Check

Tell your load balancer if you are running low on resources. There are two methods to do this:

- ***Active Health Check:*** Sending ***active probes*** *(e.g. request status every 3s)*. This method is also called ***inline*** or ***out-of-band*** (OOB) because it uses a different API to verify the status. Can be used to get more information about a service *(i.e. connection to DB may be okay, but the state isn't)*.

- ***Passive Health Check:*** Each sent request gets examined if a server is still alive. The disadvantages of passive checks are that there are only checks when a request happens and there is no option to get more information about a service like with active checks.
  There are two subtypes of passive health checks:
  - The server ***caches the request*** and ***sends it to another server*** if it fails *(default of Nginx)*
  - The server ***doesn't cache*** the request and ***sends an error*** to the user *(default of Caddy)*

6.1.2. DNS Load balancing

DNS Load balancing runs on Layer 7. Has different modes of balancing:

- ***Round Robin:*** On the DNS, a domain name is mapped to multiple IP addresses. On a request, the DNS returns all of these and the client can pick one of them. Basic & simple load balancing, but IPs are static, client can choose suboptimal address, and updating IPs can be slow due to caching.
- ***Split Horizon DNS:*** Provide different sets of DNS information selected by the source address of the DNS request *(i.e. a European server for Swiss IP address)*.

Very easy to setup. Caching with no fast changes. Requires stateless services. Offers reduced downtime, scalability and redundancy, but has a negative caching impact.

6.1.3. Layer 3 Load balancing

Via Anycast, returns IP address with the ***lowest latency***. You need your own AS *(Autonomous System)* for that, this is ***difficult*** and time consuming. Usually only used on the ISP or ***big datacenter level***.

6.2. TRAEFIK

Traefik is a layer 4 & 7 load balancer. It works by defining ***entrypoints*** *(i.e. all incoming traffic on a specific port)* that correspond to ***routers*** that can filter traffic based on path prefixes *(i.e. a router that handles all requests to `/api`)*. The routers then forward traffic to ***services*** defined inside Traefik *(i.e. frontend, backend, DB…)* which then distribute it to the machines registered for that service.

Configuration can be done via TOML files for Traefik or directly inside a Docker compose with labels *(i.e. `labels: - traefik.http.routers.dashboard.rule=PathPrefix('/dashboard')`)*

	Traefik	*Caddy*	*Nginx*
Description	Open source, software based load balancer with dashboard	Open source, software based reverse proxy with dynamic configuration	Has a free and a commercial version, very versatile
Initial setup effort	easy	difficult	medium
Proxy setup effort	easy	medium-difficult	difficult
Overall simplicity	easy	difficult	difficult

7. WEB ARCHITECTURE

7.1. SERVER-SIDE RENDERING (SSR)

"Classic" approach: the server generates HTML/JS/CSS ***dynamically*** and sends the assets in real-time to the browser. Works with these steps:

1. ***User request:*** Browser sends a request to the web server *(server-side routing)*.
2. ***Server processing:*** Server processes request by running server-side code like PHP, C# or Java, fetch required data from a database or other resources and generates HTML *(i.e. via a HTML template engine)*.
3. ***Response:*** Generate the appropriate HTML, CSS and JS for the requested page. The browser receives the response and renders the page.

Advantage: Search engine web crawlers receive the full page, which leads to better SEO, but needs server rendering for every request.

7.2. STATIC SITE GENERATION (SSG)

HTML/CSS/JS is ***pre-rendered*** since it is the same for every user. Done only once, respectively only if the content changes. Can also include ***DB access***.

Example: Page content is written in Markdown and gets converted to HTML when the Markdown file changes.

7.3. SINGLE PAGE APPLICATION (SPA)

Also called ***client site rendering*** *(CSR)*. Interactions occur within a single web page, there is no visible "page change" like on regular websites. Client page ***dynamically updates*** as the user interacts with it, providing a smooth, ***app-like experience***. Relies on ***JavaScript*** to update the UI.

Steps to render a page:

1. ***Initial request:*** Browser sends a request to receive initial HTML/CSS/JS
2. ***Initial response:*** server returns a single *(placeholder)* HTML file with CSS and JS. JS files contain the applications logic.
3. ***Browser rendering:*** Shows HTML file, typically a spinner *(loading animation)*, then executes JavaScript.
4. ***User Interactions:*** JS manages the UI updates. Application does not require full page reloads. When you click a link in an SPA, instead of making a traditional HTTP request:
    1. The JS intercepts the click event
    2. Prevents the default browser navigation
    3. Updates the URL using the History API
    4. Renders new content without requesting new HTML documents
5. ***API communication:*** When the SPA needs to send or fetch data, it does this by communicates through APIs.

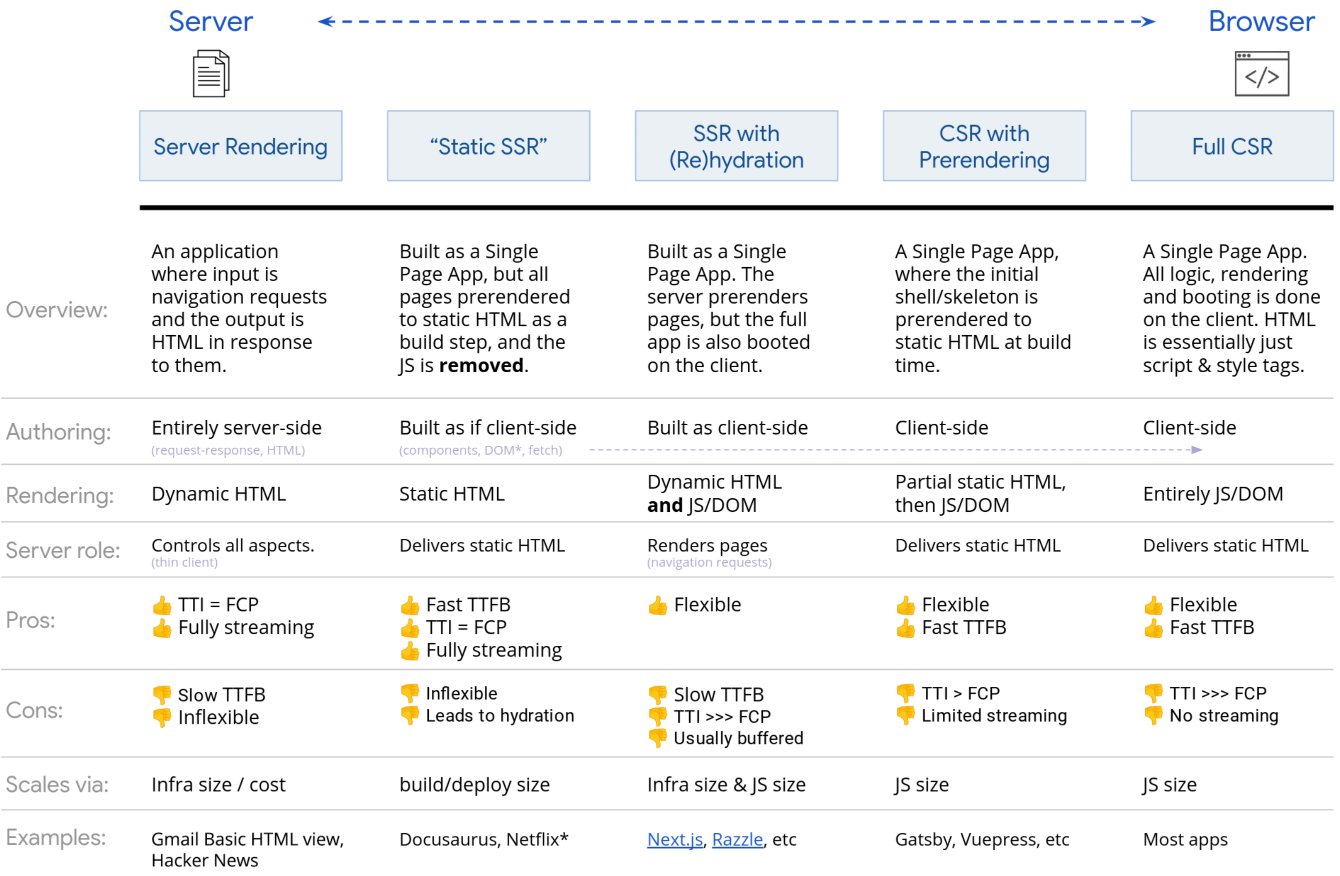
This can be done by using a framework like React, Angular or Vue. ***SEO*** only works if JS is executed at the search engine's crawler. Has ***good separation:*** UI in HTML/CSS/JS, backend in `/api`. The navigation is done via ***client-side-routing*** without intervention of the server. Requests directly to the server usually get redirected to the default `index.html` that contains the client-side-routing information.

7.4. HYDRATION

"State-of-the-art". Initial HTML not with a "spinner", but already has the first content in HTML like with SSR, and further access with API like with SPA. Essentially a ***combination of SSR & SPA***.

This is ***best of both worlds***, but ***adds complexity*** and needs JavaScript in the backend.

7.5. WEB RENDERING OVERVIEW



Server
Browser

	Server Rendering	"Static SSR"	SSR with (Re)hydration	CSR with Prerendering	Full CSR
Overview:	An application where input is navigation requests and the output is HTML in response to them.	Built as a Single Page App, but all pages prerendered to static HTML as a build step, and the JS is **removed**.	Built as a Single Page App. The server prerenders pages, but the full app is also booted on the client.	A Single Page App, where the initial shell/skeleton is prerendered to static HTML at build time.	A Single Page App. All logic, rendering and booting is done on the client. HTML is essentially just script & style tags.
Authoring:	Entirely server-side (request-response, HTML)	Built as if client-side (components, DOM*, fetch)	Built as client-side	Client-side	Client-side
Rendering:	Dynamic HTML	Static HTML	Dynamic HTML **and** JS/DOM	Partial static HTML, then JS/DOM	Entirely JS/DOM
Server role:	Controls all aspects. (thin client)	Delivers static HTML	Renders pages (navigation requests)	Delivers static HTML	Delivers static HTML
Pros:	👍 TTI = FCP 👍 Fully streaming	👍 Fast TTFB 👍 TTI = FCP 👍 Fully streaming	👍 Flexible	👍 Flexible 👍 Fast TTFB	👍 Flexible 👍 Fast TTFB
Cons:	👎 Slow TTFB 👎 Inflexible	👎 Inflexible 👎 Leads to hydration	👎 Slow TTFB 👎 TTI >>> FCP 👎 Usually buffered	👎 TTI > FCP 👎 Limited streaming	👎 TTI >>> FCP 👎 No streaming
Scales via:	Infra size / cost	build/deploy size	Infra size & JS size	JS size	JS size
Examples:	Gmail Basic HTML view, Hacker News	Docusaurus, Netflix*	Next.js, Razzle, etc	Gatsby, Vuepress, etc	Most apps

7.6. CORS

CORS stands for ***Cross-Origin Resource Sharing***. For security reasons, browsers ***restrict*** cross-origin HTTP requests *(Requests to resources not on the same domain, port or scheme)* initiated from scripts *(i.e. JS)*. CORS is a HTTP header based mechanism to indicate browsers from which other origins the site is allowed to load assets from.

Solution

Use ***reverse proxy*** with built-in webserver, e.g. nginx or user reverse proxy with external webserver. The client only sees the same origin for the API and the frontend assets. Alternatively, CORS header can be set to specify which domains can be fetched from

```
w.Header().Set("Access-Control-Allow-Origin", "https://lost.university");
w.Header().Set("Access-Control-Allow-Origin", "*"); // use only for dev
```

CORS can also cause problems with ***Preflight Requests:*** When connecting to a new domain, browsers often perform a preflight request with a HTTP `OPTION` request with headers that indicate what kind of request it is about to make to verify if the server will respond successfully. Single Page Applications need to be correctly configured to handle preflight requests to work correctly. A reverse proxy avoids the need to do this, as preflight requests will only be sent on new domains.

8. AUTHENTICATION

8.1. KEY CONCEPTS

CIA Triad

The CIA triad is a trade-off between three concepts. You can't achieve all three simultaneously:

- ***Confidentiality:*** Confidentiality protects transmitted data against eavesdroppers *(Through encryption)*
- ***Integrity:*** Provides protection against the modification of a message *(Through hashing or digital signatures)*
- ***Availability:*** Data needs to be available when needed *(Through redundancy, load distribution)*

Goals of access control

- ***Non-Repudiation:*** Provides that neither the sender nor the receiver can deny that a communication has taken place
- ***Identification:*** E.g. with a username "alice", you are claiming to be Alice
- ***Authentication:*** Verifying a claim of identity. E.g. Alice shows passport, so another person can authenticate her Different authentication types are:
  - ***Something you know:*** Things such as a PIN or a password
  - ***Something you have:*** A key or a swipe card
  - ***Something you are:*** Biometrics like palm or fingerprint
- ***Authorization:*** Determines which resources an authenticated user is permitted to access

8.1.1. Authentication Basics

Can be split into two different categories:

- ***Single-factor*** *(only uses one type of authentication, usually password. Simple to implement, but limited protection)*
- ***Multi-factor (2FA)*** *(uses multiple factors, usually password & one-time-pin. Token via SMS are considered insecure)*

When choosing a password, it should ***not contain the following***: Names of persons *(or pets)* close to you, anniversary or birth dates, name of a favorite holiday, favorite sports team or the word password. But it is better anyway if you use a ***password manager*** and generate a password that way, which also solves the problem of password reuse. In general, the length of the password matters most: The ***longer*** it is, the longer it takes to crack it. You also should not enter login credentials on unencrypted websites *(HTTP-only)*.

Passwords on a web service should be ***stored in encrypted form***. To prevent the creation of ***rainbow tables*** *(pre-calculated tables that map hashes to the plaintext passwords, because identical passwords lead to identical hashes)*, a ***salt*** should be applied to the password. A salt is a ***random value*** for each user that gets added to the password before it is hashed.

There are multiple hash algorithms that are recommended for passwords:

- ***Argon2id***: Most current algorithm, recommended, prevents brute-force attacks with CPU and time-based side channel attacks.
- ***scrypt:*** Memory intense, designed to slow down hardware attacks by utilizing big amounts of memory
- ***bcrypt:*** Older, but still considered secure. Relatively slow, which slows down brute force attacks. Scales good with better hardware
- ***PBKDF2:*** Older, but still considered secure. Less protections against hardware attacks

Argon2id, scrypt and bcrypt utilize ***salts*** by default. Slow algorithms are better, because attackers cannot test as many passwords in a short time frame. With the newer algorithms, the time it takes to hash a password can be set. Even if it is just a few milliseconds, it will slow down attackers massively.

8.1.2. Software Token: Time-based One-time password (TOTP)

Often used as a second factor, based on hash-based message authentication code *(HMAC)*. Authentication should happen either in a service, e.g. in your HTTP server *(Fine-grained per-service access, but with a lot of redundancy)* or in the load balancer, e.g. traefik *(central authentication, easier)*.

$$\text{TOTP}(K, T) = \overbrace{\text{Truncate}}^{\text{first 6 chars}}(\text{HMAC-SHA-256}(K, T))$$

K: shared secret
T: current Unix timestamp / 30 sec

The server ***generates a key*** that is usually presented to the user as a ***QR code or a link*** *(which can contain additional metadata)*. The user then ***inputs the key*** into their authenticator and now both the server and authenticator can ***generate the same token at the same time***. After entering the password and the current TOTP token, the server also generates the same token and compares the two. If they match, ***access is granted***. Because of ***timing deviations***, a server may also accept a token from the next or previous time window.

8.2. BASIC AUTH

Basic auth transmits the credentials in the `Authorization` HTTP header with base64 encoding.

- On an unauthenticated request, the ***server will reply with the header:*** `WWW-Authenticate: Basic realm="restricted area"`. The browser will display a authentication dialog together with the `realm` information to indicate what area the user is logging into.
- The ***client will provide the authorization*** by setting the `Authorization: Basic <base64>` HTTP header. `<base64>` contains `username:password` in base64 encoding. The credentials could also be ***encoded in URL*** *(e.g. https://username:password@dsl.i.ost.ch)*. This can be a security risk if the URL gets saved or passed on.

Does ***not provide encryption for credentials***, only formatting for HTTP Header. Should ***only be used with HTTPS***, otherwise login credentials are transmitted in the open. State would need to be managed by the services *(i.e. with a list of users)*. Very easy to implement in a load balancer.

Suboptimal when logging off: There is no standardization, the client must enter incorrect login data in order to receive a new authentication request, resulting in inconsistent behavior.

8.3. DIGEST AUTH

A little better than basic auth. Has a hash and a nonce which helps against replay attacks.

1. Client sends ***request***
2. Server sends a 401 with HTTP header `WWW-Authenticate: Digest` that contains a ***nonce***
3. Client ***sends response:*** `Authorization: Digest username="Alice", realm="testrealm@host.com", nonce="...", uri="../index.html", ... , response="..."`. The response is a hash which is calculated from the password amongst other things.
4. Server ***calculates hash*** as well
5. If the hashes match, the user gets ***access***

Advantages:
Password is ***not sent in clear text*** and gets ***only used once*** for the calculation of the response. Multiple encryption algorithms are available, with ***sess variants*** that support session keys, where the ***nonce changes*** with every new session. If a hash gets compromised, the attacker can only use it while the session is valid.

Disadvantages:
Uses the ***UX of the browser*** and ***cannot use scrypt or bcrypt*** to store passwords.

8.4. PUBLIC / PRIVATE KEY AUTH

The ***client provides a certificate*** signed by the server to authenticate. Usually used between load balancer and services. Better than Digest Auth.

1. ***Generate a SSL CA*** *(Certificate Authority)* on the server and configure proxy or service to require a client certificate
2. ***Create CSR*** *(certificate signing request)*, sign with CA and install certificate on client in browser
3. The browser sends the signed certificate to the server when prompted

8.5. LET'S ENCRYPT

Free, automated, open certification authority. Has automated certificate creation and renewal. Certificates are valid for 90 days. It is integrated in modern web servers.
There are two main ways to get a certificate from Let's Encrypt:

HTTP challenge verification	***DNS challenge verification***
– Server must be ***publicly accessible*** *(no VPN or internal-only)* – A ***token*** generated by Let's Encrypt is placed under `/.well-known/acme-challenge/` – Let's Encrypt verifies the token ***via HTTP request***	– Server does ***not have to be publicly accessible*** – Enables ***wildcard certificates*** *(*.ost.ch)* – Token is placed as a ***TXT entry in DNS*** – Let's Encrypt ***checks the DNS entry***

8.6. SESSION-BASED AUTHENTICATION

After a ***successful login***, the server creates a session and sends the user a ***session ID*** to authenticate with. Usually stored in a Cookie. ***Stateful, needs sticky session*** because authentication is done in the service which then has an authenticated session for the client – meaning when accessing another service *(or the load balancer selects another server)*, ***re-authentication*** is required. The load balancer avoids this with a ***sticky session***: Sending the same user to the same service every time. Example for this is spring-boot.

8.7. JSON WEB TOKEN (JWT)

Stateless, all server instances know a secret token / public key. When the user ***logs in***, the server sends back a token. The User Token gets saved in local storage. The client then sends the `Authorization: Bearer <token>` header with every request to authenticate.

A JWT consists of ***three parts***:

- ***Header:*** Contains token type *(`typ`)*, encryption algorithm *(`alg`)*
- ***Payload:*** Usual fields are subject *(`sub`)*, user role *(`role`)*, issued at in Unix time *(`iat`)*, token expiry date in Unix time *(`exp`)*. Can also contain custom fields
- ***Signature:*** generated from the header and payload together with a secret

To generate a JWT, these three parts are ***individually encoded with Base64Url*** *(a modification of Base64 that replaces all special URL characters)* and concatenated with dots, so they can be easily processed. JWTs can also contain ***sensitive/private information***, due to only the server holding the decryption keys *(Server can detect user modification of the JWT)*. Can be useful to avoid looking up user related information in the database.

```
{ // Header
  "alg": "HS256",
  "typ": "JWT"
}
{ // Payload
  "sub": "133769420",
  "name": "Nina",
  "role": "admin",
  "iat": 1750239022,
  "exp": 1800000000
}
```

Secret for the Signature:

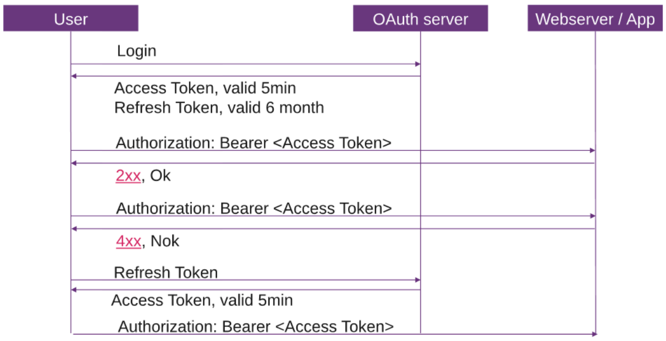
```
a-string-secret-at-least-256-bits-long
```

Generated JWT Token:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMzM3Njk0MjAiLCJuYW1lIjoiTmluYSIsInJvbGUiOiJ
hZG1pbiIsImlhdCI6MTUxNjIzOTAyMiwiZXhwIjoxODAwMDAwMDAwfQ.
nSL0D5Dm4j5wTYYA7wq19kxX7rNKKUzHM9v981u68qg
```

8.8. OAUTH

Protocol for authorization of 3rd party integration. Grant access on other websites without giving them the passwords. Often implemented with JWT.


User
OAuth server
Webserver / App
Login
Access Token, valid 5min
Refresh Token, valid 6 month
Authorization: Bearer <Access Token>
2xx, Ok
Authorization: Bearer <Access Token>
4xx, Nok
Refresh Token
Access Token, valid 5min
Authorization: Bearer <Access Token>

Access Tokens should only have a ***short lifetime*** *(e.g. 5 minutes)*. If the public key / secret is known, the content in the token can be trusted.

Refresh Token can have a ***longer lifetime*** *(e.g. 6 months)*. A refresh token is used to get a new access token. IAM service *(Identity & Access management)* or a Auth server creates access tokens.

Why are both access and refresh token needed?

- ***Only access token:*** If a user credential is revoked, how can every service be informed?
- ***Only refresh token:*** Tightly coupled service & auth: For every request to the service auth needs to be involved
- ***Access + Refresh token:*** If a user credential is revoked, user has a maximum of 10 minutes remaining to access the service. The Auth is only involved if access token is expired – a pragmatic middle way.

9. PROTOCOLS

9.1. LAYER ABSTRACTION

Internet Model	***OSI Model***	***Description***	***Header***
Application	Application	Provides the ***interface between applications*** used to communicate and the underlying network.	`Data`
	Presentation	Formats, compresses and encrypts data	`Data`
	Session	Handles the ***exchange of information***, restarts sessions	`Data`
Transport	Transport	Logical ***end-to-end connection***. *Ports*	`TCP Header, Data`

Internet	Network	Responsible for ***delivering the IP packets*** from source to destination. *IP addresses*	`IP Header,`
			`TCP Header, Data`

Link	Data link	Responsible for ***delivering data link frames*** on the same network. Error detection mechanisms. *MAC Addresses*	`Ethernet Header,`
			`IP Header,`
			`TCP Header, Data`

–	Physical	The medium the data travels through *(copper, fibre, air…)*	–

Historically, every vendor had its own non-cross-compatible networking solution. The goal of the OSI layer definition is ***interoperability***. There are a lot of different designations for the layers, everyone calls stuff differently.

Important to know: Each layer takes on a ***specific part*** of the communication task and communicates with the ***corresponding layer*** on the other side. This means that the layers are modular and protocols can be replaced.

Protocols enable an entity/instance to interact with an entity/instance at the same layer in another host. Layers don't need to worry about the ones below it, as that has already been taken care of. ***Service definitions:*** provide functionality to an (N)-layer by an $(N-1)$-layer *(every layer offers a specific service to the layer above it)*.

Each ***Protocol Data Unit*** *(PDU)* contains a protocol header and a payload, the ***Service Data Unit*** *(SDU)*.

Advantage of the layer model

Individual layers can be ***exchanged*** without changing other layers.

9.2. TCP

TCP *(Transmission Control Protocol)* allows applications to send data ***reliably*** without worrying about network layer issues.

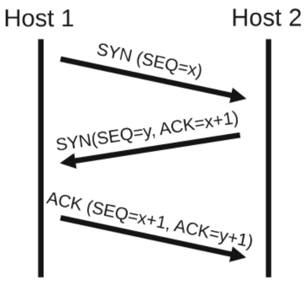
- Reliable
- Packets arrive in the order they have been sent
- The window is the capacity of receiver
  (transmitter adapts to sender)
- Checksum: 16 bits
- TCP Overhead: 20 bytes

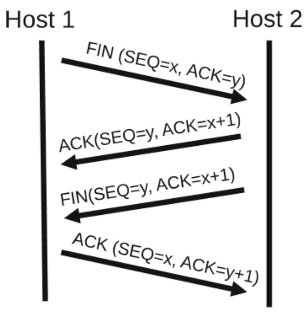
TCP tries to ***correct errors***: If a packet is missing or corrupt, it initializes a retransmission of that packet.

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The connection establishment is done over a ***three-way handshake***. Two things need to be exchanged: The initial sequence numbers (ISN) and exchange parameters. TCP needs 3 handshakes to establish the connection.

- The client sends a ***`SYN`*** message with a sequence number
- The server replies with an ***`SYN/ACK`*** message. The SYN contains a differing sequence number than the client.
- The client responds with an ***`ACK`*** message





- **Initialization of TCP session: *`SYN, SYN-ACK, ACK`*** *(left image)*
  The initial sequence number is more or less random to prevent TCP Sequence Prediction Attacks.
- **Connection Termination *`FIN, ACK + FIN, ACK`*** *(right image)*
  3 or 4-way. Prevents semi-open connections where one side thinks the connection is still open and keeps transmitting data.

Sequences and ACKs
Identification of each byte of data. The order of bytes can be reconstructed. There are two main ways to ***detect lost data***:

- ***RTO*** *(Retransmission Timeout)*: Normally set to twice the measured roundtrip time. If a packet has not been acknowledged by then, it is resent
- ***DupACK*** *(Duplicate ACK)*: A duplicate acknowledgment is sent when a receiver receives out-of-order packets. After 3 ACKs, the sender retransmits the missing packages

9.2.1. TCP window size adjustment and flow control

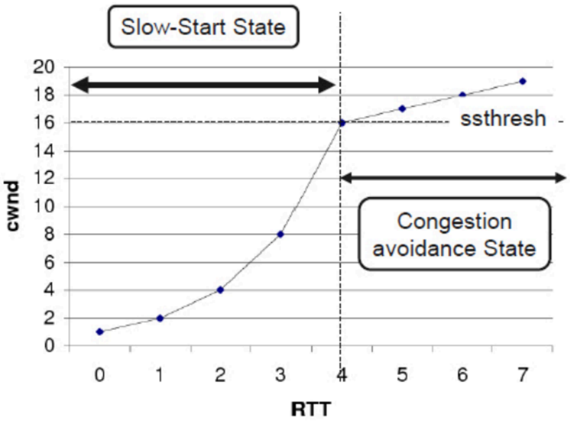
When the server receives data from the client, it places it into the ***buffer***. The server must then do two things with this data:

1. ***Acknowledgment:*** The server must send an ACK back to the client to indicate that the data was received.
2. ***Transfer:*** The server must process the data, transferring it to the destination application process.

In the basic ***sliding window system***, data is acknowledged when received, but ***not necessarily immediately transferred out of the buffer***. It is possible for the buffer ***to fill up with received data faster than the receiving TCP can empty it***. When this occurs, the receiving device may need to ***adjust the window size*** to prevent the buffer from being overloaded. A device that reduces its receive window to zero is said to have ***closed the window***.

9.2.2. TCP Congestion handling and congestion avoidance algorithms

When ***congestion increases*** on the network, ***segments would be delayed*** or dropped, which would cause them to be retransmitted. This would ***increase the amount of traffic*** on the network between client and server.



This can lead to a ***vicious circle***, resulting in a condition called ***congestion collapse***. Because of this, the TCP slow start and congestion avoidance states are needed.

- ***Slow Start:*** Slow start state is entered at the beginning of the connection or after timeout. Congestion window *(cwnd)* starts with initial `cwnd` size and increases by one or more MSS *(Maximum segment size)* for each ACK received.
- ***Congestion avoidance:*** Congestion avoidance state is entered when `cwnd` reaches slow start threshold *(ssthresh)*. `cwnd` increases by one MSS for each RTT *(if no duplicate ACK arrives)*

There are ***different algorithms*** for congestion control.

Difference Flow Control/Congestion Control:
Flow Control manages the capacity of a single client, Congestion Control of the entire network.

9.2.3. TCP Considerations

The TCP handshake is ***not*** flexible: You need 1 roundtrip for the TCP handshake. It only ensures that both sides are ready to transmit data and nothing else. There is ***no mechanism to cannot exchange public / private keys via TCP***, you need another security layer for the exchange, but this ***adds at least another roundtrip***. Additional roundtrips for DNS or old security protocols may be necessary. There is a ***big overhead***.

9.3. TRANSPORT LAYER SECURITY (TLS)

Security and encryption for most applications *(i.e. HTTPS)* is provided by TLS. It runs after the initial TCP handshake.

TLS session establishment

1. ***"client hello"*** lists cryptographic information, TLS version and supported ciphers/keys by the client
2. ***"server hello"*** sends chosen cipher, session ID, random bytes, digital certificate *(gets verified by client)* and optionally a "client certificate request" *(Client is required to send their own certificate for authentication)*
3. ***key exchange*** using random bytes sent by server, now server and client can calculate the secret key
4. ***"finished" message***, encrypted with the secret key

This process needs 3 RTTs to send first byte *(1x TCP handshake, 2x TLS handshake)* and 4RTTs to receive the first data byte. When connecting to Australia, 3 RTTs mean a latency of ~900ms.

To decrease the amounts of RTTs, ***TLS 1.3*** was released in 2018. It ***decreases*** the TLS RTTs from 2 to 1.

1. Together with the "Client Hello", a ***"key share"*** is sent. It assumes the cipher the server is going to pick.
2. "***Server Hello"***, server ***key share***, certificate verification and the ***"finish"*** are also condensed into one message.

On reconnect, even ***0 RTTs*** are possible if the cryptographic data is already known.

9.4. QUIC AND HTTP/3

QUIC *(Quick UDP Internet Connections)* was developed by Google and was adopted as the basis of the HTTP/3 standard. QUIC ***only requires one handshake*** for the ***connection*** and ***security setup***. Known connections can also be resumed with ***0 RTTs***. QUIC is based on ***UDP*** and implements ***TCP-like reliability mechanisms*** itself, but remains ***more flexible & efficient*** than those. QUIC has encryption already built-in, there is ***no un-encrypted*** variant of it *(unlike HTTP/2)*.

Some devices that perform NAT can have ***problems with assessing the status of HTTP/3 connections***. Because the connection handshake packets containing `SYN`, `ACK` & `FIN` are already encrypted, these devices don't know when a connection ends, leading to the need to ***keep these entries in the routing table for longer*** and thus ***higher resource usage***. HTTP/3 also offers ***HTTP header compression*** and can reference previously transmitted headers, leading to ***greater complexity*** in implementations.

9.4.1. Multiplexing & Head-of-line blocking

Multiplexing was already introduced in HTTP/2. It allows to ***fetch multiple resources*** from a server with only ***one TCP connection*** instead of creating a new connection for every resource *(reduces number of required handshakes)*.

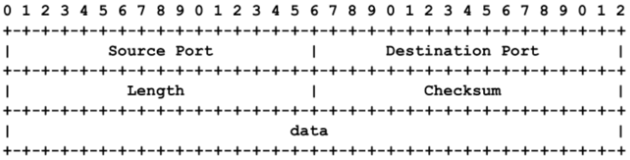
But in HTTP/2, multiplexing has a problem called ***head-of-line blocking:*** TCP guarantees the order of all packages. If a packet gets dropped, it can impact all other resources in that connection.

1. Client ***requests resources*** A and B in a ***multiplexed connection***.
2. Server sends A and B, but one packet containing part of resource B ***gets lost***
3. Due to the ***ordering guarantee***, the TCP stack on the client ***can't deliver resource*** A, even though it has been fully transmitted. Only when the missing packet from B has been received, both resources can be processed further.

QUIC solves this by introducing ***streams***. Every requests gets its own stream. A ***packet loss in one stream doesn't impact all other streams*** in the same connection.

9.5. UDP

UDP *(User Datagram Protocol)* is a ***simple connection-less communication model*** without any guarantees regarding ***delivery*** *(you don't know if package has been received)*, ***ordering*** *(packets may appear in any order)* or ***duplicate protection*** *(client needs to handle receiving the same packet twice)*. Its main use is for ***DNS*** and ***audio/video streaming***.



```
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port           |        Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Length              |           Checksum             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              data                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

9.6. SCTP

SCTP *(Stream Control Transmission Protocol)* is ***message-based*** *(unlike TCP & UDP, which are byte-based)*, which allows data to be ***divided into multiple streams***. It uses a ***four-way handshake*** with a signed "SYN-cookie" and it supports ***multihoming*** by allowing devices to have multiple IP addresses. It is ***not widely supported***, the router firmware OpenWRT doesn't enable it by default and the UFW firewall doesn't support it at all. Despite this, it is ***used by WebRTC*** *(see chapter "WebRTC" (Page 26))*, but it gets ***tunneled over UDP***.

9.7. DDOS AMPLIFICATION ATTACK

Badly designed UDP implementations can be used to perform a DDoS amplification attack, in which the client makes a ***small request*** *(i.e. 10 bytes)*, to which the server delivers a ***much larger response*** *(i.e. 100 bytes)*. If the client performs ***IP spoofing to redirect the response towards a victim***, for every byte the attacker sends, the victim receives ***10 times the amount of traffic.***

Most modern programming languages don't allow changing the sender address in UDP packages, but lower level tools like `hping3` can do that. To prevent amplification attacks, set up a ***network filtering system*** that monitors for spoofed packages *(egress filtering)*.

9.8. PROTOCOL OVERVIEW

	UDP	*TCP*	*SCTP*	*QUIC*
Layer	Transport layer			
Connection	Connection-less	Connection-oriented		
Transfer safety	Unreliable	Reliable		
Data transmission	Messages	Streams	Messages	Multistream
Ordering	Unordered	Guaranteed	User can choose	Guaranteed
Usage	DNS, HTTP/3, other	HTTP/1 & 2, other	WebRTC	HTTP/3
Overhead	Light	Heavy		
Fault tolerance	Error checking, no recovery	Error checking & recovery		Error & Integrity check, recovery
Flow & congestion control	✗	✓	✓	✓

10. APPLICATION PROTOCOLS

For some applications, it can be worthwhile to develop a ***custom protocol***. They can be ***more efficent*** *(space/performance)*, but ***more time and effort*** is needed to develop and test the new protocol. Alternatively, there are ***protocol generators*** to do the heavy lifting for you *(i.e. Thrift, Avro, Protobuf)*. These produce a standardized ***IDL*** *(interface definition language)* that can ***generate code*** in various programming languages. They can ***ease interoperability*** by delegating (de-)serialization and communication to the generated code. But using these generators comes at the cost of ***higher overhead***.

```
// Avro IDL
@namespace("ch.ost.i.dsl")
protocol MyProtocol {
  record AMessage {
    string request;
    int code;
  }
  record BMessage {
    string reply;
  }
  BMessage GetMessage(AMessage msg);
}
```

Implementing custom encoding/decoding
If you implement your own custom en- and decoding, you ***control every aspect of it***. But that also means you have to spend more time ***testing*** it.

An important aspect is the ***byte order***/***endianness*** of your protocol.
Networking protocols often use ***Big-endian*** *(higher-order byte is stored first – `0A 0B 0C 0D`$_h$ gets stored as `0A 0B 0C 0D`$_h$)* while most processor architectures use ***Little-endian*** *(lower-order byte is stored first – `0A 0B 0C 0D`$_h$ gets stored as `0D 0C 0B 0A`$_h$)*. When ***transferring data*** between these two types, you have to make sure that the bytes are ***interpreted correctly***. Explicitly specify the endianness and provide converting functions.

10.1. DIFFERENT IDLS

We want to encode a message containing the string `"Anybody there?"` and the code `5` *(15 bytes total)* as efficiently as possible. Encoding it in XML would look like this: `<code>5</code><message>Anybody there?</message>` *(48 bytes)*.

10.1.1. ASN.1

ASN.1 *(Abstract Syntax Notation One)* is a IDL released in 1984 used for serializing data in X.509 certificates *(the certificates behind TLS/SSL)*. It encodes data in ***binary***. The example would be encoded in ***21 bytes***.

10.1.2. Avro

Avro is a data serialization system that can perform ***remote procedure calls*** *(RPC)*. It is used inside Apache Hadoop, a big-data-framework. It doesn't require any code generation, all schemas are defined in JSON. The example message would be ***16 bytes*** long in Avro.

10.1.3. Protobuf

Developed by Google with the goal of being ***more compact than XML***. Google uses it for nearly all inter-machine communication internally. The example message would have ***18 bytes***.

10.1.4. Thrift

RPC framework from Facebook. Example message would have ***49 bytes*** with Thrift's RPC overhead.

10.2. RPC

Remote Procedure Call *(RPC)* lets programs ***execute code on remote machines as if it were a call on a local machine***. It ***abstracts*** the networking part and ***eases the development*** for distributed systems. The most widely used implementation is ***gRPC*** which adds RPC functionality around Protobuf. Its features are ***authentication***, ***bidirectional streaming***/***flow control***, ***blocking***/***non-blocking bindings***, ***cancellation*** and ***timeouts***.

With the RPC overhead, the example from "Different IDLs" (Page 24) requires 171 bytes in the request and 124 bytes in the reply.

Is transmitting JSON to a REST API also RPC?
While similar, there are differences:
- ***REST*** is ideally ***stateless***, while RPC can be stateful
- ***REST*** responses ***contain all metadata***, while RPC depends on interface definitions
- ***JSON*** is ***usually slower***, because it needs to be parsed

10.3. HTTP

The Hypertext Transfer Protocol *(HTTP)* is the ***foundation for data communication*** in the World Wide Web. It was started in 1989 by Tim Berners-Lee. ***HTTP/1.1***, released in 1997, added ***reusable connections***. ***HTTP/2*** from 2015 added ***header compression***, ***multiplexing*** and was in general more ***efficient***. For ***HTTP/3***, see chapter “QUIC and HTTP/3” (Page 22).

HTTP is a ***text based protocol***, HTTP communication can be directly ***observed***. It is also considered a ***stateless protocol*** because a web server doesn’t directly maintain state. If state is required, the application needs to implement it themselves.

HTTP works through the ***request/response-principle*** by requesting resources from a server, which then responds with the corresponding resource. Every resource is identified with a ***URL*** *(Uniform Resource Locator)*.

```
http://tbocek:password@dsl.i.ost.ch:443/lect/fs25?id=1234&lang=de#topj
|___|  |_____________| |__________| |_||________| |_____________||___|
  |           |              |        |     |             |         |
Scheme    User info        Host    Port   Path          Query    Fragment
```

Every HTTP request contains a ***request method*** *(also called “verb”)* that indicates to the server what it wants to to. The interpretation of a verb is mostly up to the server *(but it should conform to expectations)*. When a browser makes a request, it usually adds various HTTP headers to it. A HTTP response contains a ***status code*** and the requested content, if any.

HTTP request methods	***HTTP status codes***
`GET` *(Fetch data)*, `HEAD` *(Only headers from GET)*, `POST` *(Send data to server)*, `PUT` *(Create or update resource)*, `DELETE` *(Delete resource)*, `TRACE` *(Sends back the request, useful for debugging)*, `OPTIONS` *(lists supported methods)*, `CONNECT` *(establish TLS connection via a proxy)*, `PATCH` *(Partially replace resource)*	`1xx` *(Informational)*, `2xx` *(Success)*, `3xx` *(Redirection)*, `4xx` *(Client Error)*, `5xx` *(Server Error)* **Most common:** `200` *(OK)*, `403` *(Forbidden)*, `404` *(Not found)*

10.4. WEBSOCKETS

The REST model only offers ***one-directional communication*** *(request, then response)*. How can the server notify the browser *(client)* when it wants to send updates?

10.4.1. Polling

Classified into two types: ***Short polling***, where the client sends a request every few seconds to ask the server for updates, and ***Long polling***, that opens a `keep-alive` connection that is open until the server responds or a timeout occurs. Both of these methods are ***simple***, but ***inefficient***.

WebSockets offer ***full-duplex*** *(bi-directional)* communication over TCP. A client can create a WebSocket with a compatible server by setting the two headers: ***`Connection: Upgrade`*** and ***`Upgrade: websocket`*** to notify a protocol change from HTTP to WebSocket. The server then responds with a HTTP `101 Switching Protocol`. After successful connection, binary and text data can be transmitted from/to both sides. The URL schema changes to `ws://` *(unencrypted)* or `wss://` *(TLS-encrypted)*.

Some proxies and load balancers may need ***special configuration*** for WebSockets due to the protocol switch.

10.5. SERVER-SENT EVENTS (SSE)

Server-Sent Events are a ***one-way communication from server to browsers***. They are ***more limited*** compared to WebSockets, but ***easier*** to set up.

They use standard HTTP communication: The client sends a HTTP request with the `Accept: text/event-stream` header. The server keeps the connection open and can now push data to the client. When setting up SSEs in JavaScript, the `EventSource` API can be used to simplify SSE handling. Every message is terminated through a double new line.

SSEs offer different “channels” to group messages by. If the connection gets lost, SSEs will ***automatically try to reconnect***. There is a maximum number of concurrent connections to the same domain *(6 for most browsers)*.

Use cases are ***news feeds***, ***status updates*** and ***real-time notifications***.

10.6. SERIALIZATION FORMATS

Bencode is a serialization format that is most commonly used in the ***BitTorrent protocol***. It encodes integers *(`i42e`)* and strings together with their length *(`4:test`)*. Lists are similarly delimited like integers *(`l4:testi42ee`)*, as are maps/ dictionaries *(`d4:name4:nina5:hobby8:sleeping`)*

Other serialization formats include ***UBJSON*** *(binary formatted JSON)*, ***Cap'n Proto***, Flatbuffers *(both not serialization formats, they just copy data in little-endian)* and ***Apache Arrow*** *(fast data serialization for data in tables)*.

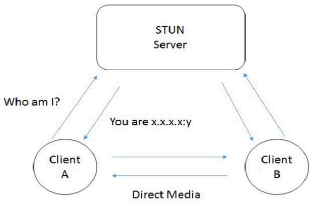
10.7. WEBRTC

Web Real-Time Communication *(WebRTC)* is a protocol released in 2011 by Google that offers ***browser-to-browser communication***. Its main usage is ***audio/video calls*** inside the browser. It communicates mostly over Peer-to-Peer. Before WebRTC, users had to either install plugins like Adobe Flash or install a native program; now most calls inside the browser *(WhatsApp Web, Facebook Messenger)* use WebRTC.

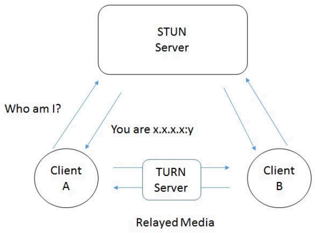
Until the WebRTC specification was ***completely standardized in 2021***, there were a lot of implementation differences between browsers. Now, WebRTC works well across most browsers. To guarantee compatibility, all WebRTC implementations ***must support the VP8 and H264 codecs***.

10.7.1. WebRTC and NAT

An advantage of WebRTC is that developers ***don't need to care about NAT***. It implements ***STUN*** *(Session Traversal Utilities for NAT)* that detects what kind of NAT the client is using by connecting to STUN servers. It is not a NAT traversal solution by itself, but a building block.



If STUN doesn't suffice, WebRTC uses ***TURN*** *(Traversal using relays around NAT)*. It routes the traffic over a ***relay server*** to make connections in network environments where peer-to-peer would not work correctly *(i.e. symmetrical NAT)*. The communication with the TURN servers happens over ***UDP***, but can also be sent over TCP, in case firewalls block UDP traffic. ***TURN works guaranteed***, but it is ***less efficient*** and the communication ***no longer peer-to-peer***.



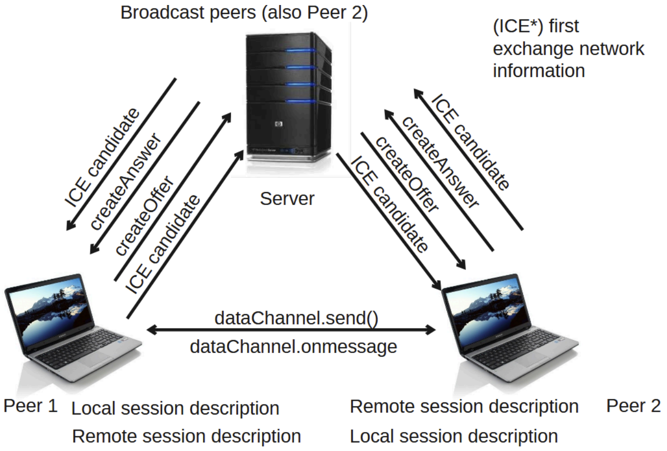
In theory, TURN could be avoided by implementing ***NAT-PMP*** *(NAT Port Mapping Protocol)* that automatically opens ports inside the NAT, similar to UPnP. But implementation of it into browsers has been stalled for a long time.

ICE *(Interactive Connectivity Establishment)* is a different protocol for NAT traversal. It works by exchanging multiple IPs/ Ports that are tested for peer-to-peer connectivity – first checking for direct connection and then falling back on STUN and TURN if necessary.

10.7.2. Architecture

WebRTC uses the ***triangle architecture***. To establish a connection, the clients must first exchange connection information *(Session description, ICE candidates)*. They exchange that information through a ***signaling server***. The communication to it is not specified, but often, ***WebSockets*** are used. When a direct connection is possible, they then connect directly, otherwise a STUN/TURN server is used.

Since the signalling itself can be implemented freely, there even exists a protocol to exchange that information ***over sound waves*** captured by microphones on the devices.


Broadcast peers (also Peer 2)
(ICE*) first exchange network information
ICE candidate
createAnswer
createOffer
ICE candidate
Server
ICE candidate
createAnswer
createOffer
ICE candidate
dataChannel.send()
dataChannel.onmessage
Peer 1
Local session description
Remote session description
Remote session description
Local session description
Peer 2

Detailed connection establishment

1. ***Peer 1 sends a `createOffer`*** with its local session description to the signalling server, server relays it to Peer 2
2. ***Peer 2 sets the local session description*** from Peer 1 as its remote session description
3. ***Peer 2 sends a `createAnswer`*** to Peer 1 with its local session description over signalling server
4. ***Peer 1 sets the local session description*** from Peer 2 as its remote session description
5. ***Peers exchange their ICE candidates*** and a direct/STUN/TURN connection is established based on the results
6. ***Peers can send messages*** with `dataChannel.send()` and get notified about incoming messages with `dataChannel.onmessage`

10.7.3. Criticism

There are a few points of ***criticism*** about WebRTC:

- Its inclusion into browsers was criticized as ***bloating them up*** and hogging even more resources
- The ***complexity*** of the WebRTC API was also called into question, many developers wished for a more simplified API *(PeerJS is a simplified wrapper library)*. The protocol itself is also quite complex with ***SCTP over DTLS over UDP***; necessary to guarantee security, but hard to debug.
- Early versions also had ***security concerns***: it was possible to leak your public and private IP address to others, even behind a VPN or the TOR network – this has been fixed. On the flip side, WebRTC does not allow unencrypted connections and even encrypts the media itself *(with SRTP & DTLS)*.

10.8. DNS

DNS translates human readable domain names to IP adresses. It works hierarchical, with the authority over subdomains transferred to the owners of these: The root servers have authority over TLDs *(.com, .ch, etc.)*, TLDs over sub-level domains. DNS uses ***UDP port 53*** for communication. It was designed in 1983 with no encryption or verification mechanism. Before DNS, the `host.txt` containing all IP-domain mappings needed to be distributed – not scalable!

While domain names only allow ASCII characters, ***Punycode*** can be used to “translate” non-ASCII characters *(i.e. bücher.ch → xn--bcher-kva.ch)*

10.8.1. DNS Setup

Typically, there are ***primary*** and ***secondary*** DNS servers for redundancy in case of failure. The secondary servers get its data from the primary through ***zone transfer*** that copies the entries to the secondary.

To make DNS queries more efficient, different DNS server types exist:

- ***Caching/Forwarding DNS:*** Cache DNS queries, so name lookups don’t need to be repeated. Usually on routers in the LAN.
- ***Recursive servers:*** Execute the actual name resolution for the client. Ask the authoritative servers and return the result. Usually on the ISP.
- ***Authoritative server:*** Provide the definitive answer to a name resolution. At the root and each TLD and SLD.

In essence: Authoritative servers allow others to find your domain, recursive servers allow you to find other domains.

10.8.2. Root servers

The root servers were originally limited to 13 due to the maximum DNS packet limit of 512 bytes, but with ***Anycast*** *(One address represents multiple servers)*, these 13 servers now ***correspond to over a 1000 servers worldwide***. For example, `l.root-servers.net` has one IP, but is mirrored in 138 locations.

The control over these root zones lies with the ***United States Departement of Commerce*** and they are operated by ***ICANN*** *(Internet Corporation for Assigned Names and Numbers)*.

In 2015, a ***DDoS attack*** against DNS root servers with 5 million requests per second was committed. The root servers as a whole could ***withstand the attack***, but some mirrors did not.

10.8.3. DNS resource records

A DNS resource record can have various types:

- ***SOA:*** Start of Authority, serial number *(version number of this DNS record)* and the default caching times
- ***NS:*** Name Server Record, sets the authoritative/primary server for this zone. If there are multiple, round robin is used, but split horizon can also be configured
- ***MX:*** Name and relative preference of mail servers
- ***A/AAAA:*** IPv4/IPv6 address record
- ***TXT:*** arbitrary and unformatted text *(used by SPF, TKIM, Let's Encrypt)*
- ***PTR:*** opposite of address record *(reverse DNS lookup)*

TTL defines the duration in seconds that the record may be cached by any resolver. "0" means no cache.

```
$TTL 3D
$ORIGIN tomp2p.net.
@ SOA ns.nope.ch. root.nope.ch.
(2018030404 8H 2H 4W 3H)
          NS         ns.nope.ch.
          NS         ns.jos.li.
          MX    10   mail.nope.ch.
          A          188.40.119.115
          TXT        "v=spf1 mx-all"
www       A          188.40.119.115
bootstrap A          188.40.119.115
$INCLUDE "/etc/opendkim/keys/
mail.txt“
$INCLUDE "/etc/bind/dmarc.txt
```

10.8.4. DynDNS

Dynamic DNS is a method of ***automatically updating a name server in the DNS***, without having complete access to a DNS server. It verifies access through TSIG *(Transaction Signatures)* that works through a shared secret and cryptographic hashing. The two main use cases are ***allowing customers of a hosting service to change DNS entries*** without giving them complete access to the DNS server and ***setting up a domain on a home network without a static IP***.

10.8.5. DNSSEC

DNSSEC is a ***security extension*** for DNS that ***guarantees authentication*** and ***data integrity***, but ***not confidentiality*** *(no data encryption)*. It is the basis for other security measures like certificates, SSH fingerprints and IPSec public keys.

DNSSEC uses two types of keys: ***Zone Signing Keys (ZSK)*** to sign records and ***Key Signing Keys (KSK)*** to sign ZSKs. ZSKs are rotated frequently, unlike KSKs.

It also introduces new record types:

- ***RRSIG:*** Resource Record Signature, contains the signature of a resource record that was signed by a ZSK
- ***DNSKEY:*** Contains the public key for the DNSSEC validation, signed by the KSK
- ***DS:*** Delegation Signer, points to the KSK in the parent zone

With DNSSEC, all DNS queries are now verified and cannot be modified, but they are still unencrypted.

10.8.6. DNS over TLS/HTTPS

DoT *(DNS over TLS)* and ***DoH*** *(DNS over HTTPS)* are two different methods to ***encrypt DNS queries*** and thus provide confidentiality of lookups in transit. In both cases, only the connection from the user to the recursive server is encrypted, the connections from the recursive to the authoritative servers aren't.

	DoH (DNS over HTTPS)	***DoT (DNS over TLS)***
Functionality	DoH-capable applications send queries over the HTTPS port 443 to a DoH capable resolver *(i.e. Cloudflare)*. This hides DNS traffic, which makes it harder to filter.	Sends DNS queries over TLS with port 853, meaning this traffic can be easily blocked or prioritized.
Support	Trivially deployed, DNS response are served like simple web pages	Widely supported by DNS servers and public resolvers
Performance	TCP + TLS handshake: 2/3 RTTs *(but Cloudflare is close to you)*	TCP + TLS handshake: 2/3 RTTs *(but ISP is close to you)*
Installation	Needs to be set up per application and the application needs to support DoH. All other programs on the system use regular DNS.	System wide: Clients can test if their resolver supports it by checking port 853, otherwise fall back to regular DNS

11. DEPLOYMENT

Back in the old days, there were two main strategies for deployment:

- ***OTS:*** "Off-the-shelf software" – ready-made Packages which could be easily installed with package managers.
- ***Custom Software:*** More complicated, you had to upload Java Web Archive files *(.war)* to the application server, which in turn extracted and installed the software.

Problems: "It runs on my machine", Who installs Java in the right version?, What happens on crashes?, Scaling, Hardware Defects, Misconfiguration *(Security risks if configuration wrong)*

VMs / Containers help a lot to combat these problems. They don't give the programs access to the complete PC, can scale, can more easily be moved to another machine and can pre-install the right Java version.

11.1. DEPLOYMENT STRATEGIES

There are a lot of different strategies to deploy containers.

- **Rolling Deployment / Ramped Deployment:** The new version is ***gradually deployed*** to replace the old version - ***without taking the entire system down*** at once.
  - + Minimal downtime, low risk
  - – Complexity, longer deployment times
- **Blue-Green Deployment:** ***2 environments***, current prod *(blue)* and current prod with new release *(green)*. ***Test, then switch***.
  - + Instant rollback, Zero downtime
  - – 2 prod environments, keeping data in sync is trickier
- **Canary Releases:** Release new version to a ***small group of users or servers first***, if all goes well, release to more users.
  - + Risk reduction, user feedback
  - – Complexity, inconsistencies
- **Feature Toggle:** Fine grained canary, ***set feature for specific users***.
  - + More risk reduction, specific user feedback
  - – Increase complexity of codebase, config management necessary
- **Big Bang:** Deploy ***everything at once***.
  - + Simple
  - – High risk, limited rollback

There are other strategies:

- ***Shadow Deployment:*** New version is deployed alongside existing one, a copy of the incoming request is sent to the shadow version for testing
- ***A/B testing:*** Version B is released to a subset of users under specific conditions.

Which strategy to choose ***depends on the environment and the product***.

11.2. PRACTICAL DEPLOYMENT

The base for the deployment is the ***containerization***. There are a lot of ***different tools*** for deployment. Most of these are ***Orchestration tools***, which coordinate automated tasks over multiple machines in overarching workflows.

11.2.1. Ansible

Ansible is a configuration tool which connects ***via SSH*** to the server. Not originally built to work with containers, but can still be used with them. Can also be used with other tools like Progress Chef or Puppet. The Ansible workflow is designed around ***playbooks*** that define the state a server is meant to be in. In a list of SSH hosts, the server it should administer are entered. Ansible then connects to them and executes the playbook. The host should run the same OS as the Ansible host to avoid conflicts.

Ansible has ***no agents running*** unlike Progress Chef or puppet. Very good for ***structured deployment***.

Run it with `ansible-playbook playbook.yml`

11.2.2. Docker

Simplest solution, no orchestrating tools necessary. Docker can set up a ***context*** with which Docker commands can be executed on remote machines. You can deploy as easily as this:

```
export DOCKER_HOST="ssh//xyz@192.168.1.2"
docker compose up -d --build
```

The build is done on the remote server. Copying files into the image works, as Docker sends this file from the local machine to the remote. This does ***not work with volumes***, however.

With the policy `restart: unless-stopped` in the Docker Compose file you can ensure that the container will automatically restart if something goes wrong, unless it was explicitly stopped.

+ very simple
– no scaling, no logging, no resource monitoring

11.2.3. Podman

Podman is daemonless. Instead it ***directly interacts*** with image registry, container, and image storage through the ***runC container runtime process***.

With tools like ***Quadlet*** *(Docker Compose-like system)* you can…

- run container under `systemd` in a declarative way
- use another container config file to create a `systemd` config file which gets installed in the OS as service / daemon
- use another project to create a container config from a podman command
- run ***seamless*** image upgrades

+ Simpler
– Deployment needs more work

11.2.4. Docker Swarm

Docker Swarm is an ***integrated orchestrating tool for Docker*** which is built into Docker and can be directly used within Docker Compose files. The most important Swarm commands are:

- ***docker swarm:*** Manage swarm *(Group of servers)*
- ***docker stack:*** Manage deployments *(Deploy programs from docker compose into the swarm)*
- ***docker node:*** Manage nodes *(server in swarm)*

The ***scheduler*** is responsible for placement of containers to nodes, it automatically deploys them based on criteria you set. Setup is easy *(but you need to do it yourself)*. The big cloud providers don't offer automatic set up, but it can be configured manually.

Docker Swarm has already ***lost the battle against Kubernetes*** for supremacy in the container orchestration space because Kubernetes can do more than Docker Swarm and supports more complex requirements.

11.2.5. Kubernetes (K8s)

Kubernetes ***automates deployment, scaling, and management of containerized applications.*** Started by Google in 2014, now it's in the hands of the CNCF *(Cloud Native Compute Foundation)*. Has become the industry standard. All big cloud providers support Kubernetes. But they often have a difficult pricing scheme.

Kubernetes simplifies ***application deployment*** and management by automatically determining how and where to deploy applications. It ensures ***high availability*** and ***fault tolerance and supports*** auto-scaling ***based on demand.*** Kubernetes facilitates ***rolling updates and rollbacks***, which is nice because rollbacks are hard, especially with state. It provides a powerful ecosystem of tools and services.

Design Principles

- ***Configuration is declarative:*** Declare the state with YAML/JSON. Specify the goal, not the way to get there.
- ***Immutable Containers:*** A container is created once and then stays the same. If changes need to be made, a new container is created and the existing one is replaced. ***Don't store state in a container.*** If a health check fails, Kubernetes removes the container and starts a new one. ***Rollback*** is done by replacing the new container with the old one. If there were changes made on the schema, this needs to be rolled back as well.

Architecture

Master Node	***Worker Node***
Controls the overall state of the cluster. – ***API Server*** manages communication within the cluster – ***etcd*** stores configuration data for the cluster – ***Controller manager*** ensures the desired state of the cluster *(Health checks)* – ***Scheduler*** assigns workloads to worker nodes.	Runs application containers. – ***Kubelet*** communicates with the master node and manages containers – ***kube-proxy*** handles network routing and load balancing – ***Container runtime*** executes containers *(containerd, CRI-0, etc.)*

Key Concepts

- ***Pod:*** Smallest deployable unit, contains one or more containers which run on the same server. They are ephemeral, meaning they are created anew each time the server starts.
- ***Service:*** Stable network endpoint to expose a set of Pods *(Makes Pods visible to the outside network)*
- ***Deployment:*** Manages the desired state of an application, defines scale, hardware limits, updates.
- ***ConfigMap:*** Stores non-sensitive configuration data for an application *(e.g. environment variables)*
- ***Secret:*** Stores sensitive configuration data like passwords and API keys.
- ***Volume:*** Persistent storage for data generated by a container. Kubernetes supports multiple types of volumes.
- ***Namespaces:*** run multiple projects on one cluster, separate them with namespaces.

11.3. IDEAL DEPLOYMENT STRATEGY

Ideally, you should only have to ***change a single line*** to deploy your application with ***another provider***. This is often not the case, however.

Terraform is a Infrastructure as a Service Tool to describe your infrastructure in terraform files. Terraform can then recreate the infrastructure on another cloud provider, which should result in better platform independence.

11.3.1. Best practices

- ***Automate the deployment as much as possible:*** The more automated something is, the less likely it is to go wrong.
- ***Infrastructure as code:*** With terraform or other tools. Leads to independence of specific providers.
- ***Immutable Infrastructure:*** Always deploy new version with changes, don't update the current version.
- ***Health Checks / Monitoring***
- ***Rollbacks:*** Always plan for rollbacks if you deploy.

12. PERFORMANCE

While hardware has made great performance leaps, network ***latency*** stays the same *(as seen in chapter "Location" (Page 5))*.

Processors can ***predict the outcomes of branches*** *(if-statements)*. If you loop over an array of numbers and have a `if (number < n)`, a sorted array will be processed much faster than a unsorted one, because the branch prediction can predict the `true`'s when `n` has not been reached and vice-versa. When a prediction is wrong, it needs to be thrown out – a ***branch miss***.

A ***L1 cache reference*** needs around 1ns, a branch miss around 3ns.

	NVMe	***SATA SSD***	***HDD***
Description	Non Volatile Memory Express achieves ultra-low latency by using parallelism and Queues, PCIe Interface and direct CPU communication. Used for DBs.	Solid State Drives use integrated circuits to store data persistently. They have no moving parts which makes them faster than HDDs.	Hard Disk Drives are electro-mechanical storages with rotating platters coated with magnetic material.
Read latency	$\sim 20\mu s$	$\sim 100\mu s$	$2 - 5ms$
Write latency	$\sim 30\mu s$	$\sim 200\mu s$	$5 - 10ms$

12.1. THROUGHPUT VS. LATENCY

Latency is the delay in network communication and shows the time that data takes to transfer across the network or ***how long it takes to open a file***. ***Throughput*** refers to the average volume of data that can pass through the network over a specific time or ***how many files can be processed in a time frame***.

The two metrics are often ***contradictory***, a system with low latency often has low throughput and vice versa. Both metrics need to be considered.

Cost is also a factor that needs to be considered, faster is more expensive.

12.2. COMPRESSION RATIOS

There is a ***trade-off to be made between CPU usage and network usage:*** Compression requires CPU resources, but reduces the amount of data transferred.

The compression ratio for ***HTML*** is around ***2–3x***, and for ***source code*** it is around ***2–4x***. ***Text*** can be compressed relatively well. If you have ***more read than write accesses***, then compression is very useful.

HTTP Compression Algorithms

- ***gzip:*** General compression tool. Most widely used option for web content.
- ***Brotli:*** optimized for web content because of included dictionary for web keywords *(HTML tags, HTTP headers, CSS properties, JavaScript keywords, URL fragments etc.)*
- ***zstd:*** General purpose, better than gzip, because it is faster and has a better compression ratio

Best practice: Enable compression with default settings, only tweak if necessary
Image compression: Choose the right tool for the job: JPEG *(try different encoders)*, JPEG XL *(GOATed format)*, avif, WebP, PNG, GIF… Use SVG for vector graphics, can be very efficient for simple graphics like logos.

12.3. BENCHMARKING

- ***Benchmark via Tool like Artillery:*** The client's CPU can become the bottleneck
- ***Benchmark via WiFi:*** Tests the WiFi bottleneck
- ***Benchmark via new connection:*** Tests the TCP slow start
- ***Benchmark via login page:*** Tests the computational complexity of used password hash algorithm

Conclusion: When benchmarking, it is always important to understand ***what the limiting factor is***, because this is what is being tested.

Benchmark example with `ab`

`ab` is an Apache HTTP server benchmarking tool. The following command tests the site http://localhost/info with 5000 requests, 50 requests at a time, with keep-alive and with a custom header.

```
ab -n 5000 -c 50 -k -H "Connection: keep-alive" http://localhost/info
```

12.3.1. Best practice

- Make it ***work***, then make it ***correct***, then make it ***fast***
- ***Premature optimization*** is the root of all evil *(Only optimize after performance testing where performance is bad)*
- Only measure and optimize ***your use-case*** *(Benchmarks may not reflect your use case)*.

13. BITCOIN / BLOCKCHAIN

Bitcoin is an experimental ***digital currency***. It is fully peer-2-peer and therefore has no central entity. The first bitcoin was issued on January 3, 2009. The smallest possible unit is 0.00000001 BTC, it is called 1 satoshi.

13.1. KEY CHARACTERISTICS

There is a ***maximum of around 21 million BTCs*** because the initial subsidy has 33 bits. The reward for Bitcoin mining halves with every 210'000 blocks mined. This is called a halving and takes place around every 4 years. After the 33rd halving, the reward is practically 0, which means that no more new bitcoins can be generated. This moment will occur at around the year 2140.

Every transaction broadcasts to all peers. Every peer knows all transactions, their total size is around 660 GB as of today.

Validation is done by ***proof-of-work*** by calculating partial hash collisions. This is difficult to fake and makes sure that there is ***no double-spending*** *(using the same Bitcoins twice)*.

The initiator is unknown so far. We know only the pseudonym Satoshi Nakamoto. There are rumors, but so far nobody has found out who the person or organization is and why the bitcoins which Satoshi has *(around 1mio BTC)* were never moved.

The first Bitcoin boom *(and crash!)* was in 2013 and another one in 2014. The trend is upwards. At the moment, the price of 1 BTC is around 104'000 USD. After every halving, the price tends to go up.

Bitcoins have evolved in the public interest from an obscure currency to an ***integral part of the financial infrastructure***.

Bitcoin does not rely on trust *(like a currency with a central entity would)*, but on ***strong cryptography***. Because there is no central bank, Bitcoin is "governed" by its development community. If there are disagreements, forks of Bitcoin can be created.

There is only ***weak anonymity*** *(pseudonymity)* because all peers know all transactions. This complete transparency makes it possible to trace the ***history of every single Bitcoin***. Techniques such as ***clustering*** can be used to determine which addresses presumably belong to the same wallet, which further minimizes pseudonymity.

BIPs *(Bitcoin Improvement Proposals)* are a standardized way to propose changes and features to Bitcoin. All BIPs are published on GitHub where they can be discussed by the community.

Bitcoins can be ***exchanged for "real" currencies***. US and Switzerland are considered Bitcoin friendly, China not that much *(mainly because of energy consumption)*.

13.2. NUMBERS

At the moment there are a total of 20 Million BTC mined, which results in a market capitalization of 2 Trillion USD.

13.2.1. Fake Volumes

There is evidence that some markets report fake trading volumes to appear more attractive than they actually are. If a volume has a ***high spread*** *(Difference between buying and selling price, should be around 0.01USD)* it is an indicator that it is fake. ***Regular trading patterns***, ***sudden spikes in volume*** and ***unnatural distribution of trading sizes*** also indicate that the reported figures do not reflect actual market activity.

13.2.2. Statistics

- There are around ***3-11 transactions per second***, which is a relatively small number compared to e.g. Visa.
- ***Transaction fees*** can be quite high, depending on the network utilization *(currently 80 million USD)*.
- The ***blockchain*** is around ***660GB*** at the moment.
- The ***Network Hashrate*** is a measure of the combined computing power of all miners in the Bitcoin network. It is currently around 1000 Exahashes per second *(1000 with 18 zeros)*. One single hash corresponds to around 12'700 floating point operations per second. So approximately 12.7 YottaFLOPS of computing power are needed in 2025.
- The ***difficulty of the calculations is adjusted*** circa every 14 days to keep the mining time at around ***10 minutes***.

13.3. MECHANISM

A wallet has public-private keys.

- The ***public key*** is created using ECDS 256bit *(Elliptic Curve Digital Signature Algorithm)* and is used for the Bitcoin Address that can receive Bitcoins.
- The ***address*** is created using the ***SHA256*** function, followed by ***RIPEMD160*** and then encoded in ***Base58***. This results in an address looking like this: `1GCeaKuhDYnNLNR6LGmBtKhPqEJD4KeEtF`.

$$\text{base58}(\text{RIPEMD160}(\text{Sha256}(\text{ecdsa public key})))$$

- The ***private key*** is used for signing transactions. If you lose your key, you lose your bitcoins.

13.3.1. Transaction

1. Peer A wants to ***send BTC*** to peer B. To do this, he creates a ***transaction message***
2. Transaction message contains ***inputs and outputs*** *(where the BTC came from and where it goes)*
3. Peer A ***broadcasts*** the transaction to all the peers in the network
4. Transactions are ***stored in blocks***. A new block is ***created / verified*** every 10 minutes, this process is called ***mining***.

The system is designed to be both ***secure*** and ***transparent***, as all transactions are stored ***publicly*** in the blockchain. The blockchain technology ensures that ***every transaction is validated by the entire network*** before it is considered valid, creating a ***high level of security*** and trust without the need for centralized authority.

13.3.2. Key Bitcoin Operations

Private key authorizes the transaction. If keys are stolen, the thief may use your coins. If the keys are lost, the coins are lost. There is no way to restore or reset them.

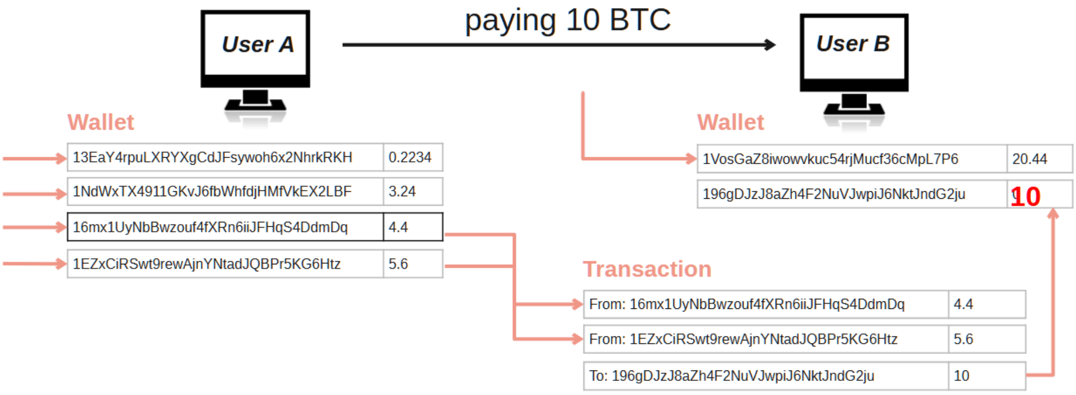
In Bitcoin, the ***UTXO system*** *(unspent transaction output)* is used. A UTXO is an output of a past transaction that has not yet been spent – Bitcoins that are currently in your wallet.

Example:
User A wants to ***send 10 BTC*** to User B. In his wallet, User A has multiple UTXOs, including one with ***4.4*** and one with ***5.6*** Bitcoins. Together, this ***results in 10 Bitcoins.*** The UTXOs ***cannot be split***. If you cannot pay the exact amount with your UTXOs, the output of a transaction consists of the amount you want to send and the "change", which you have to send back to yourself.

User B ***generates a new address*** in his wallet to which the Bitcoins are to be sent. User A then ***creates a transaction*** that refers to these two UTXOs and specifies that 10 Bitcoins are to be sent to this address of User B. The transaction is ***signed with A's private key***.

This signed transaction is then ***distributed*** in the network. The miners ***verify*** that A actually has access to the UTXOs and the transaction is then ***confirmed by inclusion in a block***. After confirmation, user B can ***access the 10 bitcoins*** by using the corresponding private key for the address. This means that he makes a transaction and ***proves with his signature*** that he can access this key.

The UTXOs in the wallet of user A are then ***changed to STXO*** *(spent transaction output)* and cannot be spent anymore. This mechanism makes sure that ***every bitcoin can only be spent once***.



13EaY4rpuLXRYXgCdJFsywoh6x2NhrkRKH	0.2234
1NdWxTX4911GKvJ6fbWhfdjHMfVkEX2LBF	3.24
16mx1UyNbBwzouf4fXRn6iiJFHqS4DdmDq	4.4
1EZxCiRSwt9rewAjnYNtadJQBPr5KG6Htz	5.6

1VosGaZ8iwowvkuc54rjMucf36cMpL7P6	20.44
196gDJzJ8aZh4F2NuVJwpiJ6NktJndG2ju	10

From: 16mx1UyNbBwzouf4fXRn6iiJFHqS4DdmDq	4.4
From: 1EZxCiRSwt9rewAjnYNtadJQBPr5KG6Htz	5.6
To: 196gDJzJ8aZh4F2NuVJwpiJ6NktJndG2ju	10

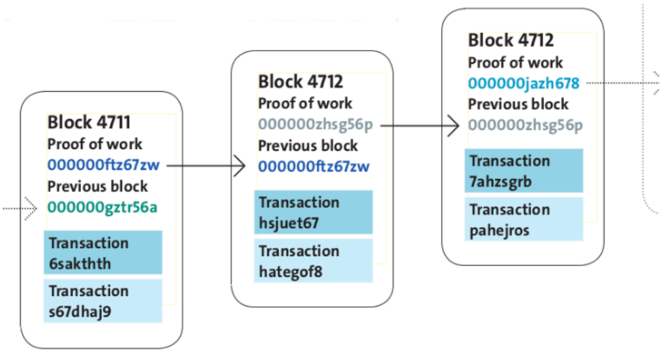
13.4. BLOCKCHAIN

Transactions are ***collected*** in blocks. A new block is created approximately every 10 minutes.

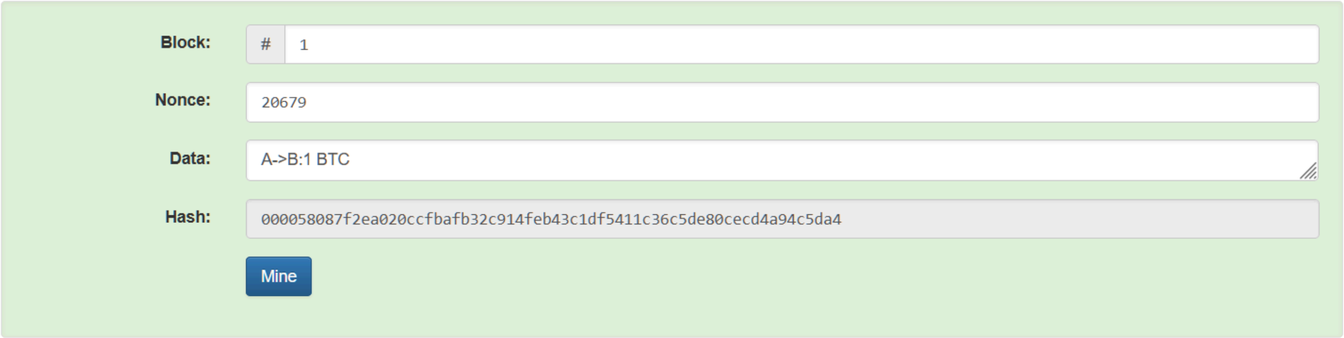
Every block contains solved crypto puzzles in the form of ***partial hash collisions***.

A block has 3 elements: ***the previous block hash value*** *(pointer to previous block)*, the ***transaction data*** and the ***nonce value***.

The nonce value is the ***solution to the puzzle***. You need to find a nonce value that, with the other elements, generates a hash value that starts with ***multiple zeroes***. Finding this nonce value is called ***mining***. It uses a lot of computing power.



Example: Nonce gets incremented until the hash value starts with 4 zeroes.


Block: # 1

Nonce: 20679

Data: A->B:1 BTC

Hash: 000058087f2ea020ccfbafb32c914feb43c1df5411c36c5de80cecd4a94c5da4

If you ***change*** something ***in the chain***, all the ***pointers*** behind this block in the chain are ***no longer correct*** and would also have to be ***recalculated***, which is very computationally intensive.

If you solve the puzzle, you get a ***reward in form of Bitcoins***. At the moment, miners get 3.125 BTC per creation. After the next halving, the reward will be at 1.5625 BTC. The ***difficulty*** gets ***adapted*** every ***10 minutes***.

Today, miners need ***specific hardware*** like ***Application Specific Integrated Circuits*** *(ASIC)*, which are only optimized for calculation of SHA256 hashes.

13.5. MINING

If you have a ***small miner at home***, you end up ***paying much more for the electricity*** you use ***than you earn in rewards***. You would need to join a ***mining pool***.

This problematic leads to the ***formation of large mining farms*** that are located in regions with ***favorable electricity prices*** or can ***negotiate favorable conditions***. This raises questions about ***decentralization*** because the farms are concentrated in certain places.

The ***longest chain***, so the chain with the highest cumulative difficulty is considered the ***valid chain***.

There are ***different levels of confirmation***, normally around ***3-6 blocks*** confirmation is considered secure *(Probability of a reversal of the transaction is negligible)*

13.6. 51% ATTACK

A 51% attack is a potential threat to blockchain networks where a ***group of miners may control more than 50% of the networks mining hash rate*** which allows them to ***prevent new transactions***, ***halt payments*** and even ***reverse transactions***.

Larger networks are less likely to fall victim to a 51% attack, ***smaller networks are more vulnerable***.

Example

An online shop for clothes accepts payments in Bitcoins. What are the 4 steps of a 51% attack on this online shop?

1. Order clothes with a ***regular transaction***. Simultaneously ***start a new chain*** with ***more than 50% computing power*** that does not include that transaction.
2. The clothes shop ***confirms the transaction*** and ***despatches the ordered clothes***.
3. ***Publish the secret, longer chain.*** Because the chain is now the longest in the Bitcoin network, it is now considered the correct chain and the chain with the payment made to the online shop is ***discarded***. The Bitcoins were therefore ***never spent***.
4. ***Profit***.

13.7. COINS AND MECHANISMS

All electronic coins are backed by scarce resources to avoid double spending.

- ***Bitcoin:*** SHA256 partial hash collision, time, ASIC, electricity *(proof of work)*
- ***Ethereum:*** Opcodes in Bitcoin, smart contracts – miners need to deposit Ethereum Coins to be able to mine; the more, the more likely one can become a validator *(proof of stake)*. This is more energy efficient.
- ***Litecoin:*** scrypt partial hash collision: Time, GPU, memory, electricity
- ***Ripple XRP:*** Unique node list, web of trust *(Participants need to trust each other)*

13.8. DISCUSSION

Disadvantages	***Advantages***
– Power consumption *(ecological footprint is terrible)*	+ Low *(fixed)* transaction fees, around 30 cents
– Not scalable *(Poor throughput of transactions)*	+ Scalable *(Hardware / storage gets faster)*
– Anonymity *(can be used for illegal activities)*	+ Anonymity *(preserving privacy)*
– Volatile exchange rate	+ No major “crashes” *(of the tech itself)*
– Central elements like core developers	+ Decentralized
	+ There are many other blockchain use cases