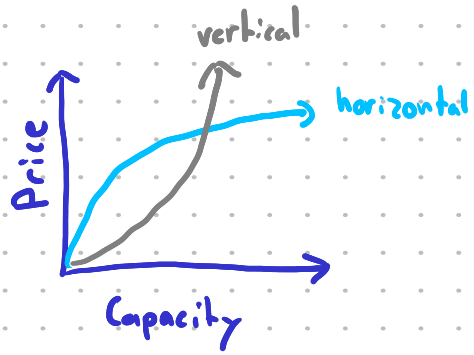
Dystributed System 25

1) Scaling



Vertical Scaling: more memory, faster CPUs

- \+ lower cost at massive scale
- \+ easier fault tolerance
- \+ higher availability
- \- adoption of code needed
- \- more complex → more components

Horizontal Scaling: more machines

- \+ current trend
- \+ lower cost at smaller scale
- \+ no SW adaption needed
- \- HW limits for scaling
- \- More HW → bigger risk of failure

2) Location

Move the server closer to the user → eg: CDN

Latency: RTT from src to dst and back.

Bound to speed of light → there is always latency

Fault Tolerance → how a (computer) system maintains proper operation in case of errors/faults

All Systems will fail eventually! Not if but when.

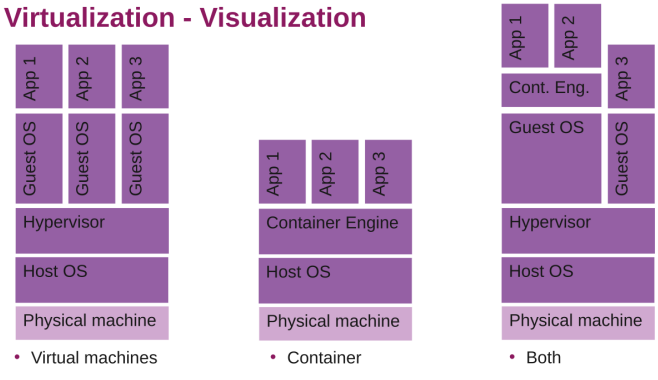
Bitflips can happen and will cause wrong results

↳ eg. from solar rays → happens on earth too, rover on Mars ∅ 280 bflips/day

↳ error correcting memory exists

Distributed Systems provide a solution by adding redundancy, but add complexity.

3) Containers / Virtual Machines



(Docker) container

- \+ small size
- \+ quick spinning up
- +/- shared memory
- +/- process based isolation (same CPU)

VMs

- \+ app can access all OS resources
- \+ live migrations
- +/- Full isolation

Monorepo / Multirepo → skipped

4) Docker & Docker Debugging.

→ build your own container

Dockerfile 499 B

```
##
## Build
##

FROM golang:1.24-alpine AS build

ARG VERSION

WORKDIR /app

COPY go.mod go.sum ./
RUN go mod download

COPY . ./
RUN CGO_ENABLED=0 go build \
        -ldflags "-s -w -X 'ost.ch/educdn/cmd.version=${VERSION}'" \
        -trimpath \
        -o ./bin/educdn \
        main.go

RUN apk add --no-cache ca-certificates

##
## Prepare Container
##

FROM scratch

COPY --from=build /app/bin/educdn /educdn
COPY --from=build /etc/ssl/certs /etc/ssl/certs

EXPOSE 8080
ENTRYPOINT ["/educdn"]
```

- Every Line of the dockerfile is one Layer
- a Docker image contains all layers, they can be unpacked individually
- a layer is cached and not rebuilt unless changed (using docker build)

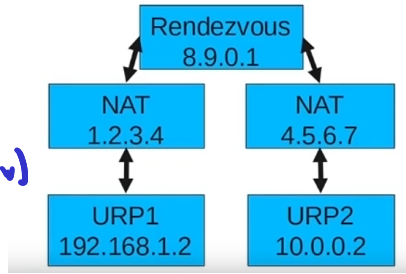
Overlay FS does the management of files across docker layers. Data is kept in lower layers but only marked as deleted.

Cgroups is a linux tool to limit CPU usage/memory/disk/... can be used in docker run with flags eg: docker run --cpuset-cpus=0 --cpu-shares=20
↳ kernel 0 ↳ max 20% usage

Network namespaces: provides network isolation for docker containers. Use NAT to communicate out of the network
↳ abstrahiert von Docker

Hole punching: Method of connecting two unreachable peers via Nat and rendezvous point.

- Entrig in NAT tables make connection work



Docker compose: deploy multiple servers (front-end, back-end usw)

Docker Security:
- keep images small and up to date
- use tiny images → alpine
- don't expose Docker daemon socket
- don't use root
- limit resources, a garbage collector might not be agressive enough, so when process gets to big it is killed by docker
  ↳ limit resources to make garbage collector docker aware
- Filesystem read only if possible

Suitable log level:
NOTICE → production
INFO:
DEBUG:
TRACE:
WARN:
ERROR
FATAL: → doomsday

log events into correct levels

5) Load Balancing

Distribution of workloads across multiple computing resources (horizontal scaling)

+ high availability
+ add / remove containers

Hardware LB: F5 / Cisco → only used if you have your Datacenter $$$

Software LB: L2/L3: Seesaw

L4: Envoy, trino, Nginx, Traefik, ...

L7: Neutrino, traefik, caddy (use for next project)

Cloud based:
- pay for use
- all big internet companies have a variation of this
- useful if one has no control over datacenter

Types of load balancing:

- DNS load balancing:
  - Multiple IP adresses returned, client can choose what IP to use
  - Depending on location, different IPs returned
  - split horizon DNS (?)
- Anycast loadbalancing: same IP for multiple servers, AS needed, bgp returns fastest path

Load balancing algorithms:

- Round robin
- weighted round robin → some servers prioritized
- least connections
- random (Traefik)

Health checks → load balancer realizes when a server is down

L7 load balancing more resource intense as TLS / HTTPS is terminated

Stickyness: can be achieved with cookies, session is connected to same container

Normally session lb is done. Packet LB is tricky

6) Web architecture

Server side rendering:
- html/css/js rendered by server and sent to client
- rendered by server side code in Java, GO, ...

+ Search engine optimization as crawler gets the whole page

Single Page application:
- interaction with single web page
- site updates dynamically with js
- data received via API communication

clicks caught by js and site reloaded → site not loaded again

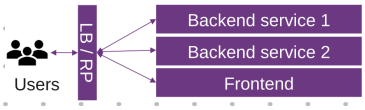
Vue, React, ...

CORS: Cross Origin Resource Sharing

For security reasons, browsers restrict access from different origins

→ use reverse proxy

problematic



good

State of the art: Initial HMTL with SSR, then dynamically updated with APIs/js

7) Categorization What is a distributed system?

A group of computers that appear to be working as one to the user

Tightly coupled: access to a common memory
loosely coupled: no access to a common memory

homogeneous: all processors are the same type (cluster of identical servers) OS Rechenleistung
heterogeneous: contains processors of different types

small scale: web app + DB
large scale: >2 machines

Decentralized → not owned by one actor → chains

CAP theorem: → can't be all three at the same time, either CP or AP
Consistency → every node has the same state
Availability → all nodes respond
Partition tolerant → system continues to be consistent even when the network partitions

More ways of classifying Distributed systems:

Architecture: client-server, peer to peer, ...

Communication model: synchronous or asynchronous (pub/sub events)

! No single universally applicable categorization of distributed system !

Distributed System	Decentralized System
• soo many	• Bittorrent / Blockchain
• 1 responsible org	• N responsible orgs
Secure environment	Hostile environment
Consistent Hashing	Consistent Hashing (distributed Hash table)
Master node	Flooding Broadcasting
No NAT issues as network under control	NAT issues → peer to peer conection difficile
Conistence managed by leader	Weak consistency → proof of work / stake ↳ largest chain wins

Transparency in Dsys: user should not be aware of / care about →

- location
- access → resource a single/uniform way
- replication
- concurrent → don't be aware of other users
- Failure → <u>not</u> be aware of recovery mechanisms

8 Fallacies (wrong assumptions) programmers like to make

1) network is reliable
2) Latency is zero
3) Bandwith is infinite
4) network is secure
5) topology doesn't change (request can take different route than reply)
6) There is one admin
7) Transport cost is zero
8) network is homogeneous

8) Authentication

Important concepts of authentication:

- CIA triad: confidentiality → message can't be eavesdropped → use encryption
  Integrity → Message hasn't been changed → Hash functions, digital signatures
  Availability
- Non-repudiation: So both parties involved can't claim a ** (e.g. transaction) hasn't happened
- Identification: claiming to be wroma
- Authentication: using pw or other factors to prove you are
- Authorisation: Permission applied to an account after succesfull login

Single Factor: Password only

Multi Factor: something you have, something you own, something you are

TOTP: time-based one-time password

Basic Auth

- Basic auth can happen at the load balancer or in the server itself.
- Basic auth should only be used with https
  ↳ base64 is no encryption!

Format username:password → can be encoded as a link: https://username:pw@url.ch

Suitable for simple, intern apps but if you can don't use it!

Digest Auth

Fixes pw in plain text weakness in basic auth (among others)

- Uses Nonce and Hash to prevent replay attacks
- Pw encrypted using MD5 (or Sha256)

Browser Lode and Feel bad and quite outdated too

Certificate based auth

→ CA must be trusted!

In the lecture, Thomas created a CA and created certs for client and user. After some configuration, he was able to authenticate between client and loadbalancer. Not really used in practice. Used more often to use TLS between LB and backend/frontend.

Let's encrypt automated cert / https renewal process.

- Domain ownership proven by http challenge to ./well-known/acme-challenge
- if site not public → DNS challenge using TXT record
- cert invalid after 90 days → more automatic renewals → can be automated (implemented in Caddy)

Stateful authentication Session based

Login with upn + pw → server creates and stores the session

Session is also returned to client and can be used to authenticate

If server changes → login not possible. Use sticky sessions on LB to always connect to the same server

Stateless sessionbased authentication used by OAuth

Better suited for distributed systems, as every server can verify the sesh

Instead of only the session, the auth server replies with JWT Token.

- JSON-based access tokens
  - Header: {"alg" : "HS256"}
  - Payload: {"sub" : "tom", "role" : "admin", "exp" : 1422779638} (more fields can be added if needed)

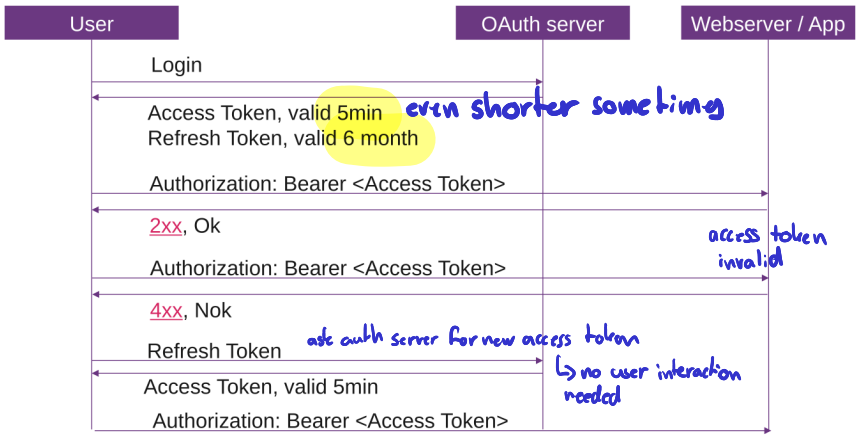
JWT contains header, payload and signature all base64 encoded

https://jwt.io/introduction

The signature is created with a public/private key pair or a secret (using HMAC) (not entirely sure how this works)

a webserver can now verify the JWT token by using header, payload and public key. If the signature is equal → verification ✓

Access Token / Refresh Token



- Refresh token only used against IdP
- Longer lifetime

- Access token short lifetime
- usable against all kinds of systems

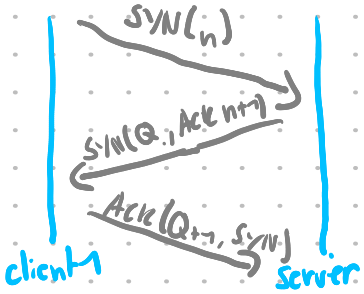
This enables SSO into "all" apps

If account disabled → Refresh token disabled and no new access token granted
But access token still valid until timestamp abgelaufen
↳ can't be changed as it is in header.

3) Protocols TCP & Quic

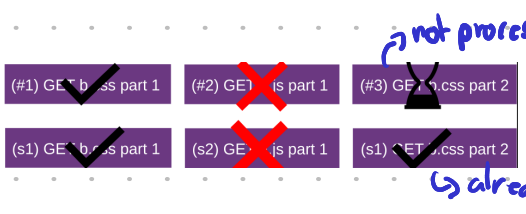
Goes into OSI model, but skipped here

TCP handshake explained
random numbers used.



Sequence number to order packages and detect missing ones.

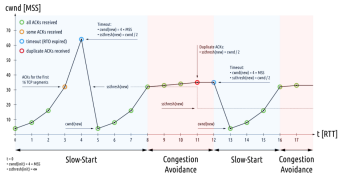
TCP does head-of-line blocking.
one stream does not affect the other



Flow control: Recipient tells the sender how many packages he can handle
↳ sliding window, adjusted after each packet

congestion control: sender tests the limit of how many packets can be sent before the traffic overflows the network

Starts exponentially (increasing) → then linearly → repeat

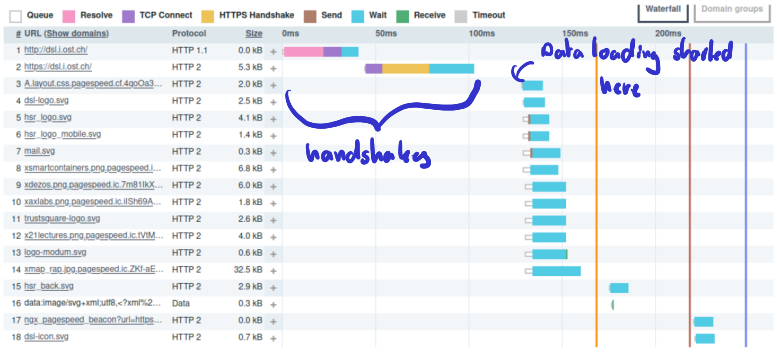


Due to latency, it can take up to a second until data can be sent. DNS query, TCP Handshake + Security = 3 Round Trips → newer protocols http3
↳ Key Exchange

Normally TLS is also used (basis for https). TLS < 1.3 requires two round trips, with TLS 1.3 it is only one.

While developing locally, RTT is almost 0 it can have a big impact on distributed systems. Thus it is important to take this into consideration.

- Resolve → DNS, TCP Connect → TCP Handshake, HTTPS Handshake → TLS/SSL



From LA to dsl.i.ost.ch took almost 1 sec for initial connection, loading the content was quicker 0.7 sec

QUIC / HTTP3

Known connections → 0 RTT ✓, UDP based

New connection: 1 RTT for connection and security

Security built in by default

read blog post about Quic, interesting and probs relevant for exam

10) Application protocol

In this lecture it was difficult to figure out what is relevant.

HTTP Hypertext Transfer Protocol

Version 1 introduced in 1989, currently using HTTP2 or HTTP3 (with Quic)

stateless → server does not keep track of sessions

HTTP stores data in header, most important Status code 200 OK 404 not found 403 Forbidden ...

Uses GET, PUT, PATCH, DELETE, ...

Websockets

bi-directional

Websockets are a two way communication channel between client/server

HTTP handshake first, then elevated to websocket

Server side Events (SSE)

One way communication channel from server to client

Implemented in challenge Task :)

automatic reconnection if site is reloaded, uses standard HTTP connections

JSON / REST API

De facto standard, used in many web applications

Human readable json used to transmit data

This has overhead and json parsing is slower compared to other methods

↳ This is why one might develop its own layer 7 protocol.

Custom protocol

+ control every aspect of it
- more time developing and troubleshooting

Avro IDL (interface description language)
↳ define the protocol / interface and can be converted into many different languages

You can use binary encoding, which is usually faster

Examples are: · Avro
· Protobuf → smaller and faster than xml, used by pretty much all of Google internal communication.

Application Part II

(Binary) Encodings: binary encoding and there are A LOT of options
↳ Eg: Bencoding (used in BitTorrent), UBJSON, Cap'n Proto
· Which encoding to choose depends on the implementation → do Benchmarks

WebRTC

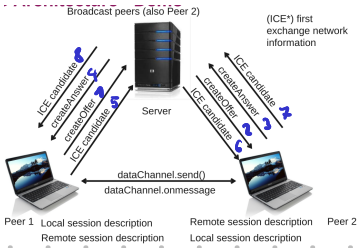
RTC = Real time communication

A protocol mostly use to enable video calls across browsers. Peer to peer for the most part.
↳ Eliminates Plugins and Flash Applications as supported by all browsers.
· First cross browser call 2013 → standardized in 2021.

The only real downside is that browsers get bloated

WebRTC can have Problems with NAT. As a developer there is no need to worry about it, as there are protocols implemented to deal with NAT.

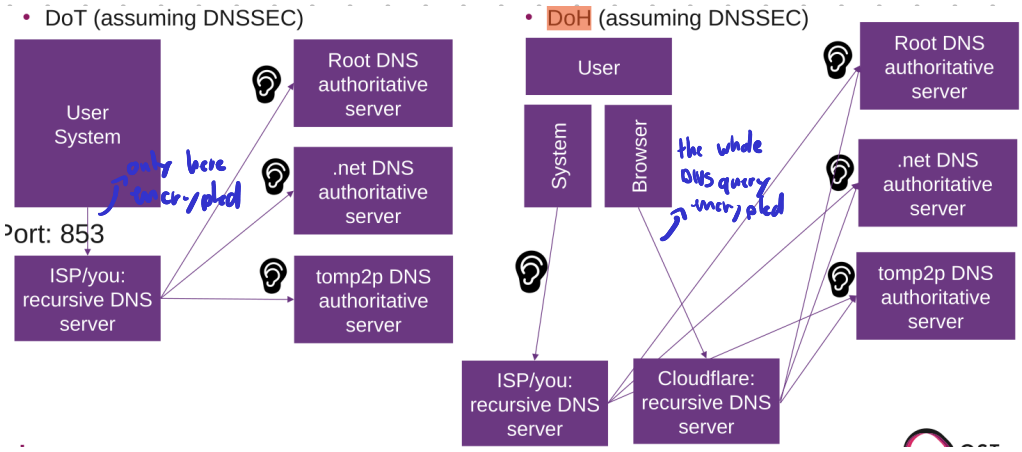
- STUN: help a client find its public IP
- TURN (Travels using relays around NAT): When P2P isn't possible, a relay server is used where both clients connect to.
- ICE (Internet connectivity establishment): Test multiple ways of establishing a WebRTC connection (p2p preffered) and helps find an optimal way that works.



DNS

Goes into details that I'll skip, because it is like the 4th time looking at this

DoH = DNS over HTTPS, DoT = DNS over TLS



Fallback: DNS via port 53

11) Exam prep → Todo

12) Deployment

Rolling Deployment: new version gradually deployed to replace old system
the whole system is not taken down at once

Blue-Green Deployment: one live, develop / test the other then switch

Canary Release: new features to a few people, if all goes well extend

Feature toggle: finer canary release, enable features for a group of users

Big Bang: deploy everything at once

Technical Deployment: Dockerization as the basis

- Ansible → push based via ssh, ssh device list needed
- Plain docker / podman: as in seProj → docker content
- Docker Swarm → has lost the battle against Kubernetes
- Kubernetes - Container orchestrierung → industry standard
  - define goal (how many container...) in Yaml and Kubernetes makes sure to deploy it

12) Performance

Idea is to get a feeling how long different operations take.

Network RTT remained stable across the last 20 years, Hardware performance got faster

Branch prediction CPU → example showed that JS code runs faster if the if-clause is predictable. As in always true because array is sorted. → no real use in real life

Latency: time per operation } often one of is quick. eg. key value store has quick latency for one object, but longer to respond to a lot of them.
Throughput: amount of operations }

more compression leads to less data to be sent

Best practice → enable compression, use the default value (lossless compression)

Image compression → lossful compression

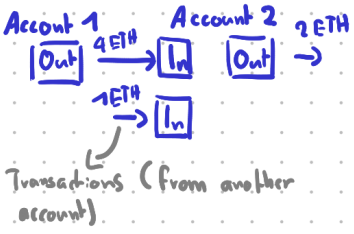
13) Blockchain

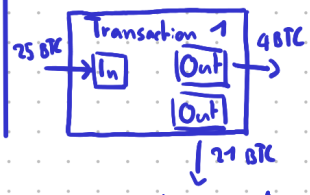
No central entity. Block validation by proof of work

Wallet has private and public key

Transaction broadcasted to all peers and placed in block → verified in around 10 min

Account-based (Ethereum)	UTXO-based (Bitcoin) (unspent transaction output)
• Global state has a list with balance on each account • Transaction is valid if sender has enough	• Every referenced input must be valid and not yet spent • All outputs are always spent





• into another Transaction or into an account

rest sent back to yourself

Mining: Block SHA256 hashed → miners try to find hashes starting with 0000... by changing Nonce
Each block has a pointer to the previous one → blockchain
• 51% attack can take over chain