Distributed Systems

Table of Contents

1 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. They do not share a common memory or clock. Each computer in the system is typically responsible for its own processing and storage, and the computers may be geographically dispersed and connected through a network.

Erhöhen die Komplexität, diese gilt es jedoch zu vermeiden

Nicht vergessen, dass heutige Hardware sehr schnell ist, eine Website mit wenigen DB Calls braucht keine starke Hardware oder sogar horizontale Skalierung, jedoch sind Dinge wie Machine Learning oder Gaming Hardware-intensive

Werden gebraucht für:

- **Skalierung**
  - **Vertikal / Scale up**: mehr CPU / Memory Ressourcen hinzufügen
  - Vorteile:
    - tiefere Kosten bei kleinen Systemen
    - Software muss nicht angepasst werden
    - niedrige Komplexität
  - Nachteile:
    - Hardware-Limit bei Skalierung
    - Risiko von Ausfall wegen Hardware-Fehlern
    - Fault Tolerance ist schwieriger
  - **Horizontal / Scale out**: mehr Maschinen, aktueller Trend, einzige Option bei Machine Learning
  - Vorteile:
    - tiefere Kosten bei riesigen Systemen
    - Fault Tolerance kann einfach hinzugefügt werden
    - höhere Verfügbarkeit
  - Nachteile:
    - Software muss angepasst werden
    - komplexere Software
    - mehr Komponenten
- **Location**: näher zum User rücken, alles wird schneller, ausser die Latenz, welche durch die Lichtgeschwindigkeit limitiert ist
  - perfekte Glasfaser von Rapperswil nach Sydney hat Round Trip Time von 110ms

    - Nur wenig Optimierungspotenzial: z.B. ist aber Kupfer etwas schneller als Fiber und die Netzwerkkomponenten wie Repeater, Switches oder Router können durch Low-Latency Modelle ersetzt werden
    - Latenz hat oft einen direkten Einfluss auf das Geschäft des Unternehmens
    - Services näher beim User reduzieren Latency, erhöhen die Bandbreite, den Durchsatz, die Zuverlässigkeit und die Verfügbarkeit, jedoch muss man sich um die Datenreplizierung und Caching kümmern
    - Content delivery network (CDN): verteilte DBs, Edge Computing, Bilder, Seiten und Scripts sind näher beim User
- **Fault-tolerance**: HW kann ausfallen
    - Bit flips: schlechte Pin Verbindungen, falsche RAM Timings, Clock Issues, RAM Design flaws, CPU/RAM/Motherboard Logic Defects, Cosmic Rays
    - Error correcting code (ECC) Memory: Triple modular redundancy (TMR) oder Hamming Code kann 1 Bitflip korrigieren und 2 Bitflips erkennen, wird auf Servern aber selten auf Clients eingesetzt
    - Bitsquatting: DNS Hijacking ohne Exploiting, Domains mit nur einem Bit unterschiedlich von der originalen Domain kaufen, darauf z.B. Login abfangen
    - HDDs können kaputtgehen
    - SSDs nutzen sich ab: NAND Zellen haben begrenzte Lebenszeit, Ersatz-NANDs sind vorhanden, kann z.B. mit smartctl auf Linux kontrolliert werden, Wear Leveling verteilt Write/Erase Operationen auf alle Memory Zellen
        - SLC: 10'000 – 100'000 Write/Erase Cycles
        - MLC: 10'000 Write/Erase Cycles
        - TLC: 1'000 Write/Erase Cycles
        - QLC: weniger als 1'000 Write/Erase Cycles
    - Unterseekabel werden oft beschädigt

Moore's Law: Number of transistors doubles every 2 years

Nielsen's Law: a high-end user's connection speed grows by 50% each year
Bandbreite wächst langsamer als Compute power, weil Telekom-Firmen zurückhaltend sind, Benutzer nicht gerne viel Geld für Bandbreite ausgeben und die Anzahl User steigt

		Annualized Growth Rate	Compound Growth Over 10 Years
Nielsen's law	Internet bandwidth	50%	57×
Moore's law	Computer power	60%	100×

Kryder's Law: disk density doubles every 13 months

Die $/GB sinken, Speed ist sehr wichtig (SSD) und viel Kapazität weniger (wegen Streaming)

2 Categorization

Kategorien:

- **tightly coupled**: Nodes haben Zugriff auf gemeinsamen Memory
- **loosely coupled**: Nodes haben keinen gemeinsamen Memory
- **homogen**: alle Prozessoren haben den gleichen Typ
- **heterogen**: Prozessoren haben verschiedene Typen
- **small scale**: z.B. Web App + Datenbank
- **large scale**: mehr als 2 Maschinen
- **decentralized**: technisch distributed, aber nicht nur ein Besitzer

CAP Theorem:

Consistency (jedes Node hat den gleichen konsistenten Zustand), *Availability* (jedes non-failing Nodes gibt immer eine Antwort), *Partition Tolerance* (System ist immer noch konsistent, auch wenn sich das Netzwerk partitioniert), man muss sich für zwei entscheiden, ist zu simpel gehalten

Architektur Kategorien:

- **Client-Server**: Client sendet Input an Server, welcher mit Output antwortet
- **Peer-to-Peer**: Nodes sind gleichzeitig Client und Server
- **Hybrid**: Mischung von Client-Server und P2P
- **Software Architektur**: Layers, bspw. OSI Modell

Kommunikationsmodell:

- **Synchron**
- **Asynchron**

Weitere Classification: Transparenzgrad, Fault Tolerance, Skalierbarkeit, Konsistenz (stark, eventuell, kausal), Daten-Replikation, Daten-Partitioning, Heterogenität

"Controlled" distributed systems

- 1 verantwortliche Organisation
- Niedrige Abwanderungsrate
- Beispiele: Amazon DynamoDB, Client-Server
- Sichere Umgebung
- High Availability
- Homogen oder heterogen
- Gut funktionierende Mechanismen: Konsistentes Hashing (DynamoDB, Cassandra), Master Nodes, zentraler Koordinator
- Netzwerk unter Kontrolle oder Client-Server, keine NAT Issues

"Fully" decentralized systems

- N verantwortliche Organisationen
- Hohe Abwanderungsrate
- Beispiele: BitTorrent, Blockchain
- "Feindselige" Umgebung
- Nicht vorhersagbare Availability
- Heterogen
- Gut funktionierende Mechanismen: Konsistentes Hashing (DHTs), Flooding / Broadcasting (Bitcoin)
- NAT und direkte Verbindungen sind ein Problem

- Konsistenz mit Leader Election: Zookeeper / Zab, Paxos, Raft
- Mehr Replicas bedeuten höhere Verfügbarkeit, bessere Performance, bessere Skalierbarkeit, aber Konsistenz muss gewahrt werden
- Transparency Prinzipien anwenden

- Weak consistency mit DHTs, Nakamoto consensus, Proof of stake, PBFT Protokolle
- Gleiche Replication Prinzipien
- Transparency Prinzipien anwenden

2.1 Transparency Prinzipien

- **Location transparency**: User sollten physischen Ort nicht merken
- **Access transparency**: User sollten das System nur über einen einzigen einheitlichen Weg aufrufen können
- **Migration und relocation transparency**: User sollten nicht merken, dass Ressourcen verschoben wurden
- **Replication transparency**: User sollten Replicas nicht merken, es soll wie eine einzige Ressource erscheinen
- **Concurrent transparency**: User sollten andere User nicht bemerken
- **Failure transparency**: User sollten recovery Mechanismen nicht merken
- **Security transparency**: User sollten security Mechanismen so wenig wie möglich merken

2.2 Falsche Annahmen

- Netzwerk ist zuverlässig → Unterseekabel können ausfallen
- Latenz ist null → Ping bis Australien sind 300ms
- Bandbreite ist unlimitiert → 1 GBit/s heisst 1 TB braucht 2h und 16 Minuten zum Übertragen
- Netzwerk ist sicher → immer davon ausgehen dass jemand zuhört und verschlüsseln
- Topologie ändert sich nicht → beim Ping nach Australien kann die Antwortroute anders wie die Anfrageroute sein
- Es gibt einen Administrator → Route kann über Konkurrenzfirmen gehen (UPC, Init7)
- Transportkosten sind null → Jemand hat das Netzwerk gebaut und betreibt es
- Netzwerk ist homogen → es gibt Fiber, WiFi, Kupfer, Desktops, Server, Handys, ...

3 Load Balancing

- Workloads (Requests) werden effizient auf verschiedene computing resources (machines) verteilt
- nötig bei horizontaler Skalierung
- ermöglicht High Availability und Reliability indem Requests nur an aktive Server gesendet werden
- ermöglicht Flexibilität, da einfach Server hinzugefügt oder entfernt werden können je nach Nachfrage

3.1 Typen

- **Hardware**: geht nur mit eigenem Datacenter, z.B. loadbalancer.org, F5, Cisco
  - Proprietäre Software für spezialisierte Prozessoren
  - Gibt aber auch Open Source Varianten wie HAProxy
  - Nicht generic, sehr performant
- **Cloud-based**: pay for use
  - AWS: Application Load Balancer (ALB, L7), Network Load Balancer (L4)
  - Google Cloud: L3, L4, L7
  - Cloudflare: L4, L7
  - DigitalOcean: L4
  - Azure: L4, L7
- **Software**
  - L2/L3: Seesaw
  - L4: LoadMaster, HAProxy, ZEVENET, Neutrino, Balance, Nginx, Gobetween, Traefik
  - L7: alle L4 ausser Balance, Envoy, Eureka

3.2 Software based DNS Load Balancing

- Round-robin DNS: mehrere IPs für den gleichen Namen, es kommt eine zufällige IP als Antwort für den Client
- Split horizon DNS: mehrere Zone Files pro Domain, Client bekommt andere Antwort je nach geografischem Ort
- Anycast: eigene BGP AS benötigt, sehr schwierig und aufwändig, IP mit tiefster Latenz wird returned, es gibt Anycast as a service oder AWS Global Accelerator

3.3 Algorithmen

- Round robin: sequentiell an Server senden, am einfachsten
- Weighted round robin: ein paar Server haben mehr Leistung und erhalten mehr Anfragen
- Least connections: Verbindung geht an Sever mit den wenigsten Verbindungen zu Clients
- Least time: Kombination der schnellsten Antwortzeit und wenigsten aktiven Verbindungen
- Least pending requests: kleinste Anzahl aktiver Sessions
- Agent-based: Server reporten Load mit Agent
- Hash: Ein definierter Key (z.B. Source-IP) wird hashed und dann der Server aufgrund des Hashs ausgewählt, hat sticky Session
- Random

Stateless: nichts sollte auf dem Service gespeichert werden, wenn nicht stateless werden sticky sessions benötigt

Sticky session: gleicher User geht immer zu gleichem Service, wird z.B. mit Cookies realisiert, welche im Browser abgelegt werden und dann immer mitgeschickt werden, so kann der LB den Service entscheiden

Healthcheck: Load Balancer merkt, wenn ein Service stark ausgelastet ist

- Inline/Inband mit Service
- Out of band (OOB): API für Healthcheck beim Service, z.B. ist der inline check ok, aber die DB Connection nicht

3.4 Reverse Proxy

Transparent Reverse Proxy: bei Caddy bedeutet es, dass der HTTP Host header nicht entfernt wird

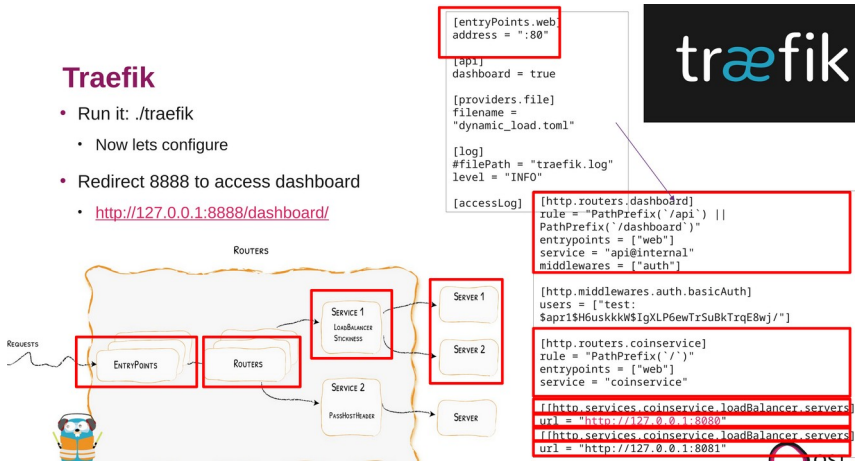
L4: Client macht TCP Connection zu LB (Source-IP ist die vom Client), LB macht TCP Connection zu Server (Source-IP ist die vom LB), Routing Entscheidungen nur aufgrund von Src-IPs, Dst-IPs und Ports, kein smartes Load Balancing, keine streaming/kepp-alive Connections, keine TLS Termination, weniger ressourcen-intensiv

L7: Client macht TCP Connection zu LB (Source-IP ist die vom Client), LB macht TCP Connection zu Server (Source-IP ist die vom LB), Authentication möglich, Smartes Routing bspw. aufgrund von HTTP Paths, TLS Termination, ressourcen-intensiver

Proxy: gesetzt im OS oder Applikation, wird für Internetzugriffe verwendet, jeder Request wird vom Proxy Reverse Proxy gesendet, welcher Requests splittet und zu Services sendet

3.5 Produkte

- **Traefik**: Golang single binary, open source, L4/L7, Authentication, experimental HTTP/3 support, Dashboard, Health Checks, Middlewares, diverse Algorithmen



- **Caddy**: Golang, open source, L7, dynamische Konfiguration, statische Konfiguration mit Caddyfile, One-liners für static file server und reverse proxy, automatisches HTTPS, HTTP/1.1, HTTP/2 und experimentelles HTTP/3
- **Nginx**: C, free und commercial Version, L4/L7, schneller Webserver mit 35% Marktanteil, gekauft von F5 networks in 2019, HTTP/Mail/Reverse Proxy, Load Balancer, keine aktiven Health checks in der free Version, nur passive, keine sticky sessions, Algorithmen: least_conn, ip_hash, cookie (nur commercial), weighted balancing (weight=1)
- **HAProxy**: C, open source mit commercial support, L4/L7, Sticky sessions mit appsession, inband healthchecks, Algorithmen: roundrobin, leastconn, source, Primnary/secondary für Services

3.6 CORS

Cross Origin Resource Sharing

Aus Sicherheitsgründen blockieren Browser cross-origin HTTP Requests ausgeführt von Scripts, um das freizuschalten, braucht man den HTTP Header Access-Control-Allow-Origin beim Backend, darin muss https://frontend.origin.ch stehen

4 Container

4.1 VM

VM: verhält sich wie ein echter Computer mit Betriebssystem, auf Host machine läuft Virtualisierungs-Software, Guest machine ist die VM, OS wird mit Intel VT-x/AMD-V oder paravirtualized ausgeführt

Hypervisor: Type 1 (bare-metal, läuft direkt auf Hardware, z.B. Xen) & Type 2 (hosted, läuft auf normalem Betriebssystem, z.B. VirtualBox)

Emulation: wird gebraucht wenn Architektur von Host / Guest nicht gleich ist, z.B. QEMU

VDI: Virtual Desktop Infrastructure, mit VM über Netzwerk interagieren

Use Case: Bessere Hardwareauslastung und Resource sharing

Cloud Provider: AWS EC2, Azure Virtual Machines, Google Compute Engine, DigitalOcean Droplets

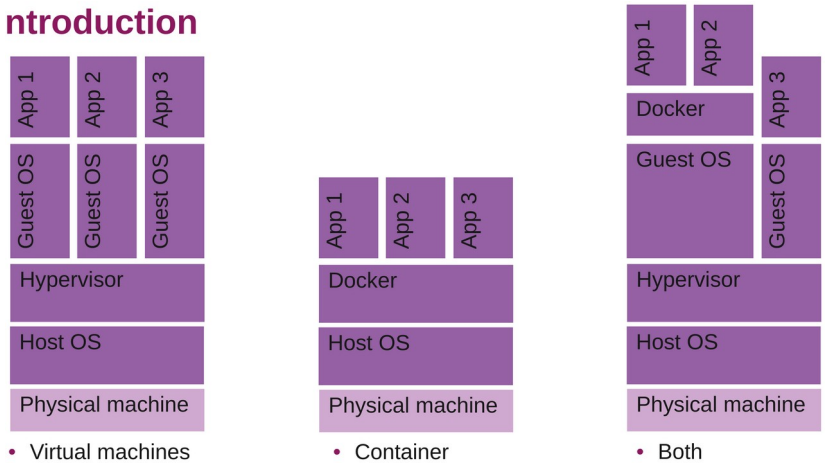
4.2 Container

OS-supported isolierte user-space Instanzen

Use case: Komplexe Application Setups, mit Containern weniger komplexe Konfiguration

Cloud Provider: AWS ECS, Google Kubernetes Engine, Docker / Kubernetes on Azure

4.3 VM vs. Container



VM
+ App kann alle OS Ressourcen verwenden
+ Live migrationen
+ / - Memory pre allocated
+ / - Volle Isolation

Container
+ Kleinere Snapshots
+ Schnellere spin-ups
+ Weniger IT Management Ressourcen
+ viele Container auf einer Maschinen
+ weniger und einfachere Security Updates
+ / - Verfügbarer Memory ist shared
+ / - Prozessbasierte Isolation (gleicher Kernel)

4.4 Docker

Produkt von Firma Docker Inc., Containerization platform, Software delivery framework, Software wird in Container verpackt (Images), existierende Images sind z.B. auf dem Docker Hub, OS-level Virtualisierung, Container sind isoliert voneinander und kommunizieren über well-defined channels

Basis für: Kubernetes, OpenShift, docker-compose, AWS Fargate, Google Cloud Run, Digital Ocean App Platform, Nomad

Commands:

- Image ausführen, bei Bedarf von Registry pullen: docker run <image-name>
- Image in Tar speichern: docker save <image-name> -o image.tar
- Tar entpacken: tar xf image.tar
- Cached Images auflisten: docker images
- Image löschen: docker rmi <image-name-oder-id>
- Alle laufenden/exited Container auflisten: docker ps -a
- Container löschen: docker rm <container-name-or-id>
- Image erstellen aufgrund von ./Dockerfile: docker build . -t <image-name>
- Alles entfernen: docker system prune -a
- Shell ausführen in Container: docker exec -it <container-name-or-id> sh

Bocker: Docker in ca. 100 Lines Bash implementiert

OverlayFS: union filesystem, kombiniert verschiedene Mount Points in einen, *lowerdir* kann read-only oder selber ein overlay sein, *upperdir* ist normalerweise schreibbar, im *workdir* werden Files vorbereitet, wenn sie zwischen den Layers verschoben werden

Layers: pro Instruktion im Dockerfile wird ein neues Layer hinzugefügt, die Layers werden cached und nicht neu erstellt, wenn der Input sich nicht ändert

Volumes können beim Builden nicht mounted werden, um bspw. Dependencies zu cachen
Lösung: Container nicht neu builden, sondern Code-Folder mounten und Hot Reloading des Frameworks nutzen

Cgroups: limitiert / isoliert / priorisiert CPU, Memory, Disk I/O und Netzwerk, kann für Prozesse oder Docker Container verwendet werden

Network Namespaces: isoliert System Ressourcen, welche mit Networking zu tun haben, es können virtuelle ethernet Verbindungen erstellt werden, können durch iptables Regeln von aussen erreichbar gemacht werden

Docker compose: YAML um mehrere Container zu deployen, Services werden konfiguriert und man kann etwas orchestrieren mit Dependencies

Kompose: docker-compose zu Kubernetes Manifests konvertieren

Dockerfile: Eigenes Image mit einem Dockerfile erstellen, Images sollten möglichst klein gehalten werden, Multi Stage builds möglich (Build in einem temporären Container durchführen und Binary in richtiges Image kopieren)

Security:

- Image möglichst klein mit wenigen Dependencies halten, so wird die Angriffsfläche verkleinert
- aktuelle Versionen verwenden
- Kleine Runtimes wie Alpine Linux (musl) oder Distroless (libc) verwenden, keine grossen Images wie Ubuntu/Arch/Debian/Fedora verwenden, Alpine hat viele nützliche Debugging Tools dabei, für glibc Busybox
- Images nach Vulnerabilities scannen mit bspw. snyk oder clair
- Docker Socket nicht in die Container exposen
- User im Container setzen, Hauptprozess nicht als root laufen lassen
- Capabilities des Containers aufs Nötigste limitieren
- Inter-container Kommunikation deaktivieren
- Ressourcen (CPU, RAM, File Descriptors, Prozesse, Restarts) limitieren
- Filesysteme und Volumes read-only setzen
- Dockerfile linten
- Docker im rootless Modus laufen lassen oder Podman verwenden
- Logging Level auf geeignetes Level für das Environment setzen (TRACE (local env) / DEBUG (dev env) / INFO (prod env) / NOTICE / WARN / ERROR / FATAL) setzen
- Ans Zielpublikum denken
- Logging Libraries verwenden
- In JSON loggen für die Struktur
- Keine sensitiven Daten loggen
- Secrets Manager verwenden, z.B. Github secrets, HashiCorp Vault, AWS secrets manager, GCP secret manager, git secret

Debugging: Anderen Container pingen (Docker compose hat eigenen DNS), Service nur an localhost bound?, Logs prüfen, docker stats,

Alternative: Podman, etwas andere Architektur, hat keinen Daemon, zu dem man verbindet, docker-compose wird unterstützt

5 Repositories

5.1 Monorepo

Ein Repository für alle Projekte, 1 Ordner für frontend, 1 Ordner für backend, usw.

Auch genannt: onerepo, unirepo

Benötigt evtl. Tools wie lerna (mehrere JS Packages aus einem Repo)

5.2 Polyrepos

Mehrere Repositories für ein Projekt, frontend in anderem Repo als backend

Auch genannt: manyrepo, multirepo

Sync mit git submodules (git repo als subdirectory eines anderen git repos) oder bash script

5.3 Vergleich

Monorepo

- Projekte sind eng gekoppelt (z.B. kann openapi.yml im backend generiert werden und ins frontend kopiert werden)
- alle sehen allen code / commits
- code sharing in der Organisation wird gefördert
- grosse repos, spezielles Tooling

Polyrepo

- Projekte sind schwach gekoppelt (zur openapi.yml Generierung braucht man Zugriff vom Backend Repo aufs Frontend z.B. mit curl + token)
- granulares Access control
- code sharing über Organisationen hinweg wird gefördert
- viele projects, spezielle Koordination

Meinung Thomas: Polyrepo für kleine Teams und eher unabhängige Projekte, Monorepo für Projekte mit enger Kopplung

6 Service Worker

Ein Service Worker ist ein Computerprogramm, das im Hintergrund eines Webbrowsers läuft und als Vermittler zwischen der Webanwendung, dem Browser und dem Rechnernetz dient. Es handelt sich um eine Technik im Bereich der Progressive Web Apps (PWAs) und ermöglicht es Webanwendungen beispielsweise, bestimmte Funktionen offline verfügbar zu machen und Push-Benachrichtigungen zu senden. Er kann Netzwerk-Requests intercepten und verarbeiten und verwaltet einen Response-Cache.

6.1 Einsatzzwecke

Offline-Nutzung von Websites: löst den Application Cache ab, beim Initialisieren des Workers ist eine Internetverbindung erforderlich und alle benötigten Ressourcen werden heruntergeladen und gespeichert

Push Benachrichtigungen: vom Server versandte Benachrichtigungen, die den Benutzer auch dann erreichen sollen, wenn dieser die entsprechende Seite nicht geöffnet hat (Web push)

Firebase Cloud Messaging (FCM, früher Google Cloud Messaging GCM): integriert in Android, iOS und alle modernen Browser, alternativ mit MicroG als FCM Android-Client oder UnifiedPush als Server, Browser Push Notifications bei geschlossenem Browser funktionieren nur, wenn Browser und OS ausreichend integriert sind wie bspw. bei Android, nicht aber bei Firefox auf Windows

Background Sync: Requests zwischenspeichern und bei Verbindung senden

7 Authentication

7.1 Access control Konzepte und Begriffe

- **Confidentiality**: übertragene Daten vor Eavesdroppern schützen
- **Integrity**: Schutz vor Abänderung der Nachrichten
- **Availability**: Daten müssen verfübar sein, wenn sie gebraucht werden
- **Non-repudiation**: Sender und Empfänger können Kommunikation nicht abstreiten
- **Identification**: man behauptet eine Identität, z.B. mit einem Username
- **Authentication**: man bestätigt die behauptete Identität (am besten mit 2 Faktoren)
  - Something you know: Passwort oder PIN
  - Something you have: Key oder Karte
  - Something you are: Biometrie wie Fingerabdruck, Gesicht, Handvenen
  - Software Token: Time-based one-time password (TOTP), oft als 2. Faktor verwendet, basiert auf keyed-hash message authentication code, Truncate(HMAC-SHA-256(shared secret, (current unix time / default 30s)))
- **Authorization**: prüfen auf welche Ressourcen der authenticated User Zugriff hat

7.2 Basic Auth

kann vom Load Balancer gemacht werden, sollte nur mit HTTPS gemacht werden

Client sendet Header: Authorization: Basic <base64(username:password)>

Server antwortet mit Header: WWW-Authenticate: Basic realm="restricted area"
User sieht Information "restricted area"

Kann in der URL encodet werden: https://username:password@dsl.hsr.ch

7.3 Digest Auth

1. Client macht Request
2. Client bekommt von Server eine Nonce und einen 401 Authentication Request
3. Client sendet folgendes Response-Array:
   (username, realm, md5(nonce, username, realm, URI, password_from_browser))
4. Server nimmt username und realm, er weiss auch die URI und berechnet den Hash selbst md5(nonce, username, realm, URI, password_from_database)
5. Wenn die Hashes matchen, wird der Zugriff gewährt

Nonce schützt vor Replay Attacken, verfügbar in Traefik, Passwort wird nicht im Klartext übertragen, ist also auch ohne HTTPS sicher, Passwort kann nicht verschlüsselt auf Server gespeichert werden

7.4 Public / private key auth

1. CA auf Server generieren und Proxy konfigurieren, dass er Client-Zertifikat braucht
2. CSR erstellen, mit CA signieren und Zertifikat auf Client im Browser installieren
3. Client muss Zertifikat an Proxy sendet, sonst bekommt er keinen Zugriff

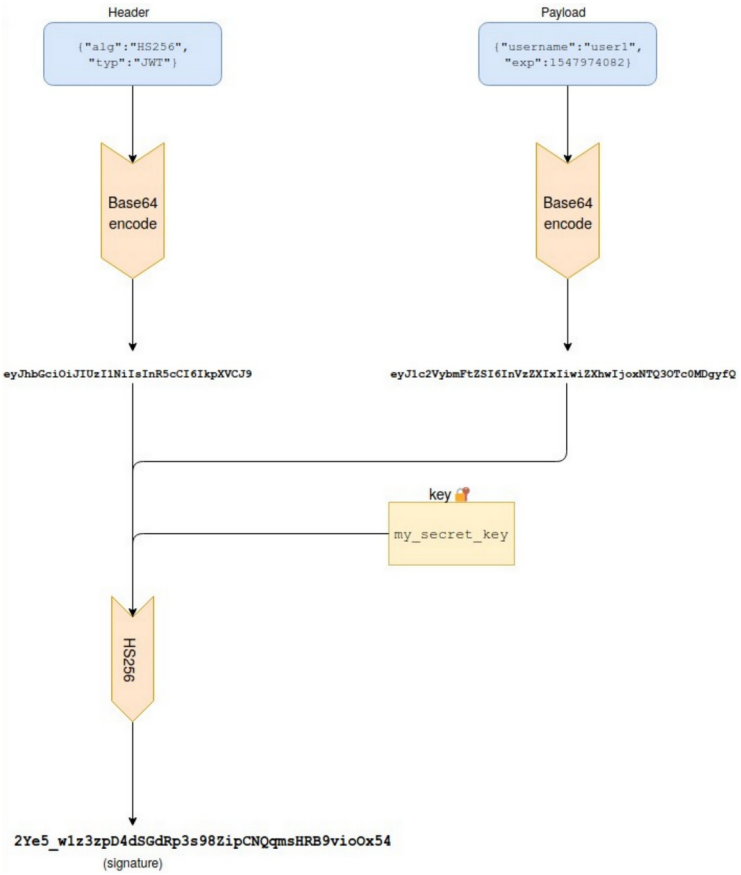
7.5 Session based auth

Stateful, benötigt sticky session, weil Auth beim Service geschieht und dieser dann eine authenticated session für den Client hat

7.6 JSON Web Token

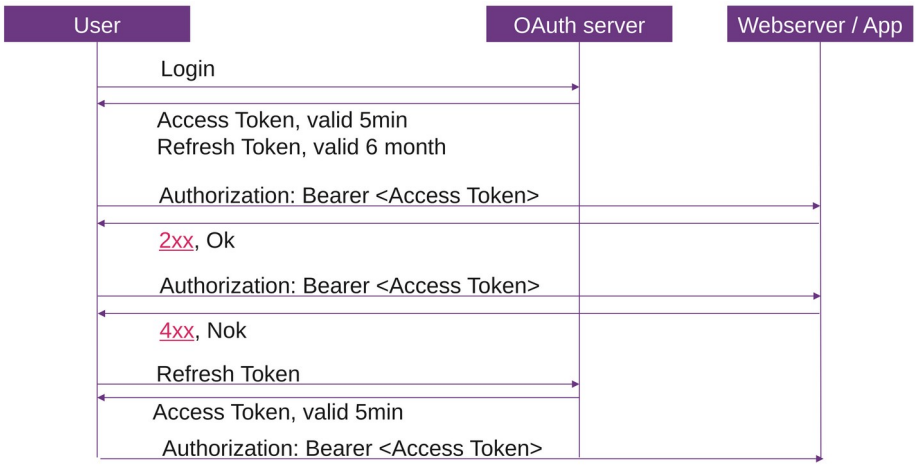
Stateless, alle Services kennen einen secret token und einen public key, wenn ein User sich einloggt, sendet ihm der Server einen mit dem secret token signierten user_token, diesen sendet der Client dann bei jedem Request im Header mit: "Authorization: Bearer <token>"

user_token speichern: Local storage



7.7 OAuth

Zugriff auf andere Sites geben ohne ihnen das Passwort zu geben



Wenn der public key / das secret bekannt sind, kann dem Inhalt des Access Tokens im Service vertraut werden. Er kann auch userId, Role, usw. enthalten. Der Refresh Token hat eine längere Lebensdauer und wird gebraucht, um einen neuen Access Token zu holen beim IAM / Auth Server. Wenn ein User revoked wurde, funktioniert der Access Token bei diesem Beispiel für max. 5 Minuten, da der Auth Server für den Refresh Token keinen neuen Access Token ausstellt.

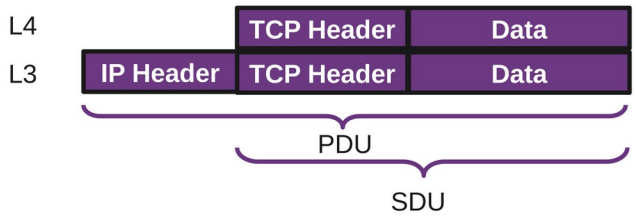
Refresh und Access Tokens können als HTTPS-only Cookie oder im Local Storage gespeichert werden im Browser

8 Protokolle

Das OSI und das Internet Modell stellen funktionierendes Networking zwischen verschiedenen Herstellern sicher

OSI model	"Internet model"
Application	Application
Presentation	
Session	
Transport	Transport
Network	Internet
Data link	Link
Pysical	

Protokolle erlauben Interaktion zwischen Entities auf dem gleichen Layer, N-1 Layer stellt Funktionalität für N Layer bereit, jede protocol data unit (PDU) enthält einen Header und Payload (=service data unit SDU)



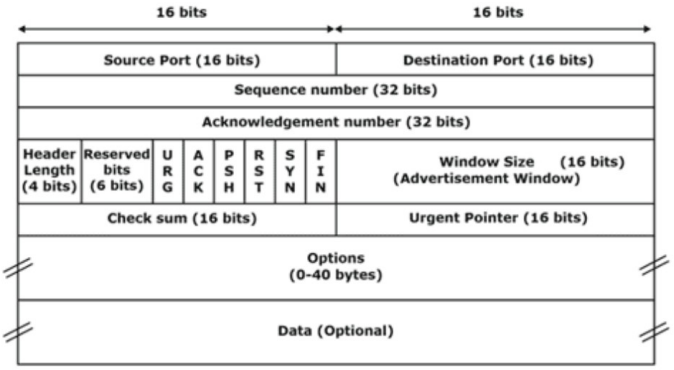
L2: Ethernet Frame, 18 Bytes Overhead

L3: IP Paket, 20 Bytes Overhead

8.1 L4: Transport, TCP

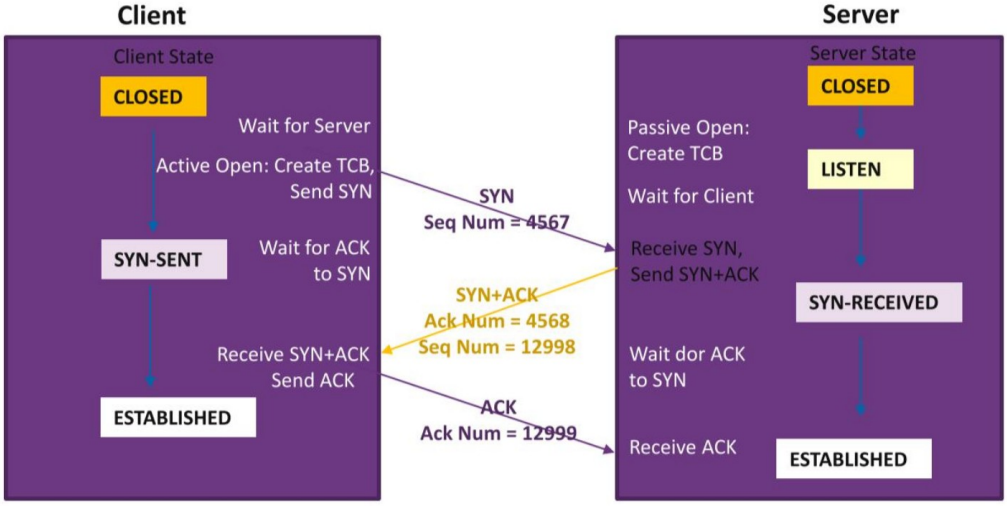
Verbindungsorientiert, bidirektional (Partner können senden, egal wer die die Verbindung erstellt hat), mehrere Verbindungen zwischen den gleichen Partnern möglich, zuverlässig, mit ACKs, Daten können nicht verloren gehen, nimmt einen Block von Daten und teilt ihn in TCP-Segmente, damit Messages für IP entstehen

4.4.1 Header



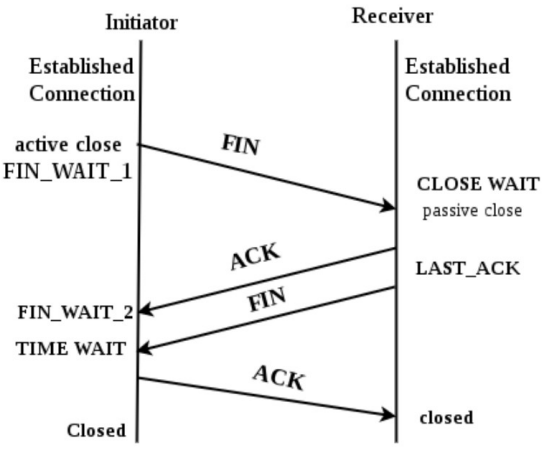
- Src/Dst Port
- Sequence Number: hilft bei Sortierung der Segmente, da diese in falscher Reihenfolge ankommen könnten
- Ack Number: Sequence Number, die der Sender als nächstes erwartet
- Header Length
- Reserved Bits: für Zukunft, alles 0
- URG: 1 = Urgent
- ACK: 1 = aktiviert Auswertung Ack number
- PSH: 1 = Push, nicht buffern und Sliding Window deaktiviert, z.B. bei Telnet
- RST: 1 = Reset, Verbindung abbrechen nach technischen Problemen
- SYN: 1 = Sync, Verbindung initiieren
- FIN: 1 = Finish, Verbindung beenden
- Advertisement Window
- Checksum: Checksumme über Header + Daten
- Urgent Pointer: Gibt Position des ersten Bytes des Datenstroms an
- Options: optionale weitere Verbindungsdaten

4.4.2 Connection Establishment



- TCB = Transmission Control Block
- 4567 & 12998: Initial Sequence Numbers (ISN) von Client bzw. Server, zufällig vom OS generiert

4.4.3 Connection Termination



In der Grafik links sieht man eine saubere Termination. Es kann jedoch manchmal zu Problemen kommen, beispielswiese halb-offene Connections oder unerwartete Messages. Dann sendet das Gerät, welches ein Problem erkannt hat, ein Segment mit der RST Flag gesetzt.

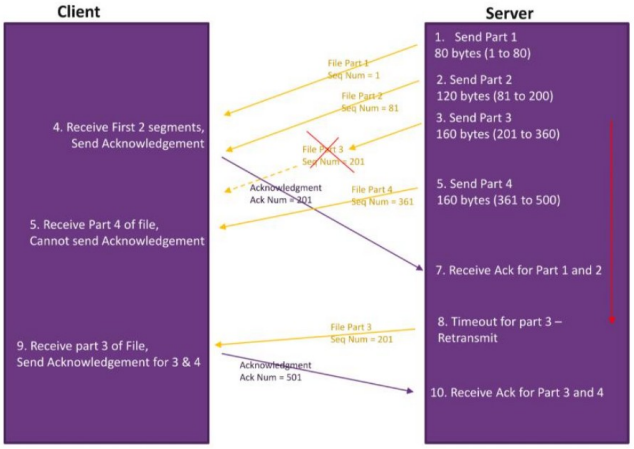
4.4.4 Maximum Segement Size (MSS)

Gewisse schwache Geräte haben einen limitierten Buffer und möchten die Segment-Grösse limitieren. Kompromiss zwischen:

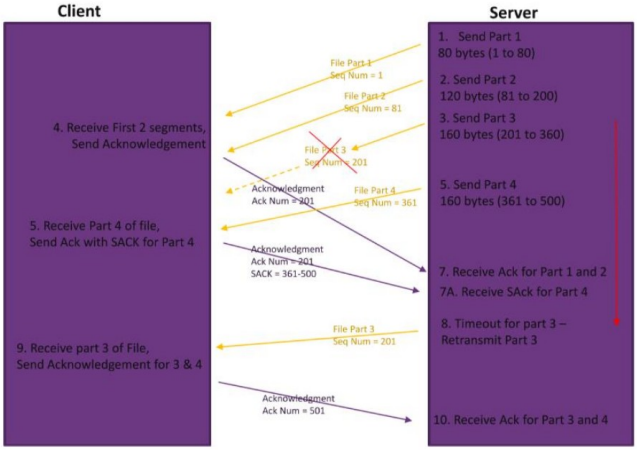
- **Overhead**: MSS=40B, IP-Header=20B, TCP-Header=20B, also 50% Header, 50% Header-Overhead, je grösser desto effizienter
- **Fragmentation**: Wenn MSS > MTU – 40B IP/TCP-Header, dann werden Segmente fragmentiert, zu gross = Fragmentation

Die Default MSS ist 536B. Der Grund ist, dass die minimale MTU 576B ist, 40B für TCP/IP-Header abgezogen sind 536.

4.4.5 Acknowledgement & Retransimission



1. Von jedem gesendeten Segment wird eine Kopie in die Retransimission Queue platziert
2. Wenn ein ACK vor dem Retransmission Timeout ankommt, wird das Segment aus der Retransmission Queue entfernt
3. Wenn kein ACK ankommt, wird nach dem Retransmission Timeout automatisch retransmitted



Wenn jedes Segment Acknowledget warden muss, generiert das sehr viel Overhead, daher gibt es Selective Acknowledgement (SACK).

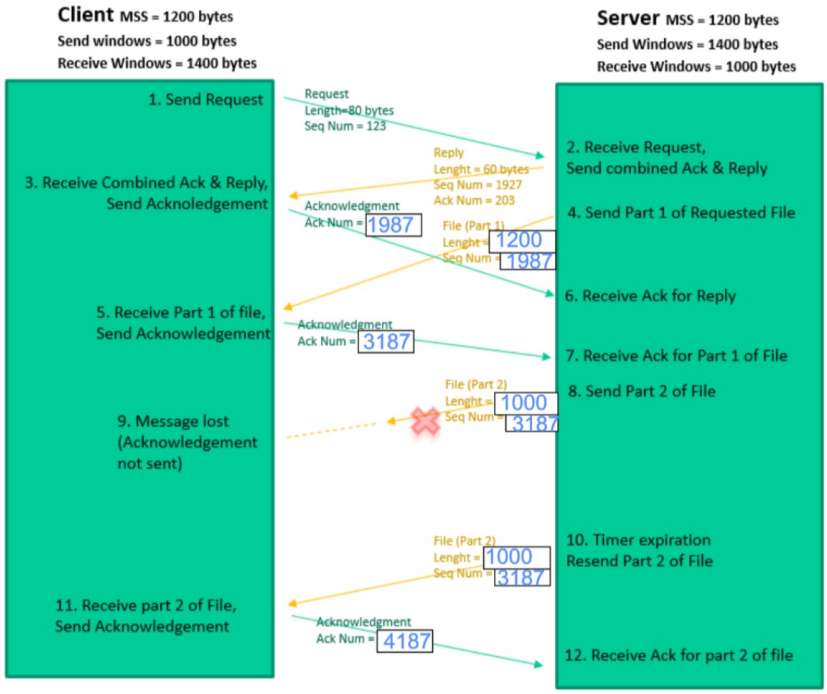
4.4.6 Sliding Window

Es gibt vier Kategorien von Bytes:

- 1: Bytes gesendet und Acknowledged
- 2: Bytes gesendet und noch nicht Acknowleged
- 3: Bytes noch nicht gesendet, für die der Empfänger bereit ist
- 4: Bytes noch nicht gesendet, für die der Empfänger noch nicht bereit ist

Send window: Kategorien 2 + 3
Usable window: Kategorie 3

Wenn ein Gerät alle Bytes im usable window gesendet hat, wurden alle Bytes von Kategorie 3 zu 2 verschoben. Die Grösse des Send window ist also immer gleich und die Grösse des usable window kann 0 sein. Für jedes acknowledgte Byte (Verschiebung von 2 zu 1) kommt dann ein Byte aus 4 zu 3.

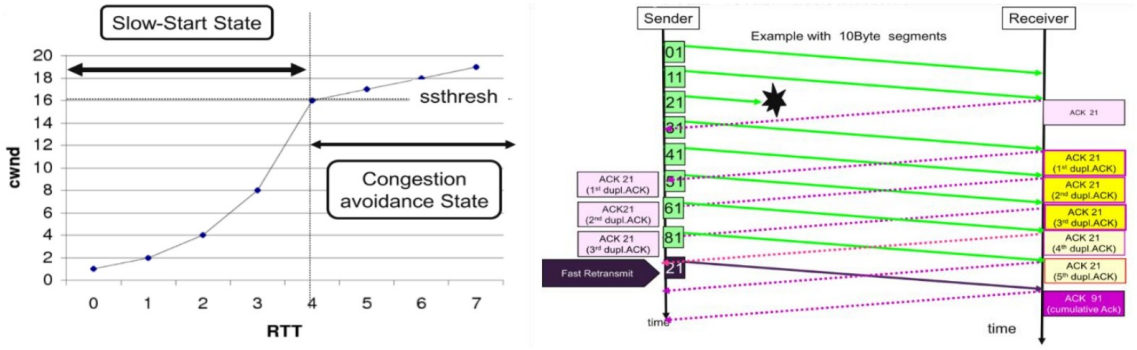


4.4.7 Slow Start, Duplicate ACK, Fast Retransmit, Fast Recovery

Das Ziel von Slow Start ist, das congestion window (cwnd = send window) so gross wie möglich zu machen, ohne das Netzwerk zu überlasten. Für jedes angekommene ACK wird ein oder mehr MSS addiert. Immer wenn ein Packet Loss (kein ACK) auftritt, wir das cwnd halbiert.

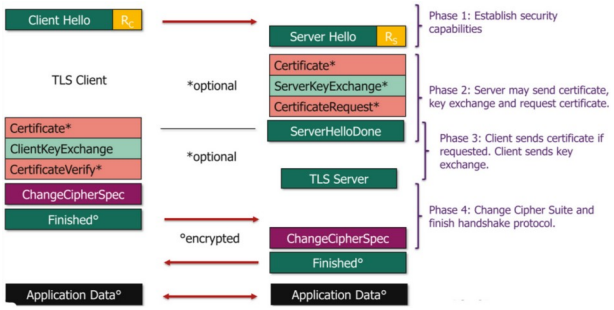
Sobald der slow start threshold (ssthresh) erreicht ist, wird für jede Round Trip Time (RTT) ein MSS addiert. Halbiert wird nicht mehr.

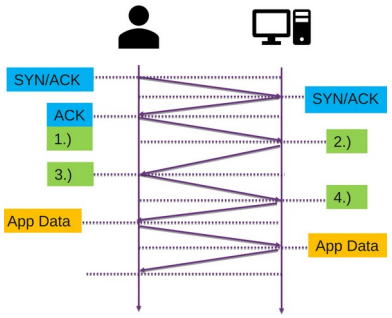
Wenn Fast Retransmit verendet wird, um ein verlorenes Segment nochmals zu senden, verwendet das Gerät Congestion Avoidance, benutzt aber vorher nicht Slow Start. So wird die Performance verbessert.



TCP + Security braucht mind. 2 Roundtrips, DNS braucht 3 Roundtrips, ältere Security Protokolle brauchen 4 Round Trips, Worst case: Starlink/Australien/DNS/TCP/Altes Security Protokoll: 1.4s bevor Daten gesendet werden können

8.2 Transport Layer Security



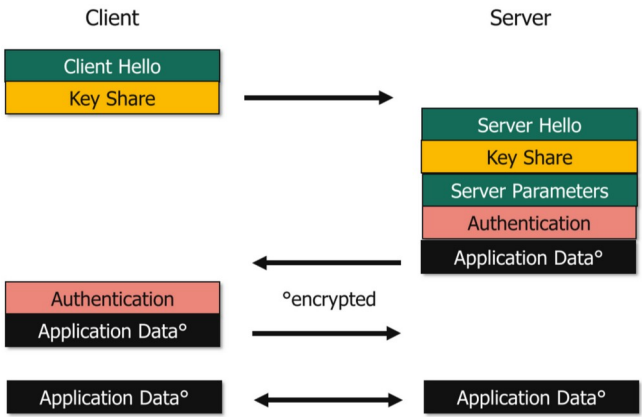


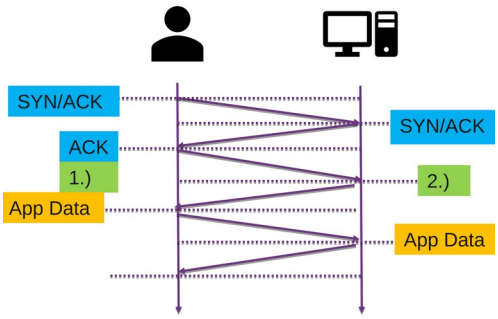
3 Roundtrips bis erstes Byte gesendet wird, 4 Roundtrips bis erstes Byte erhalten wurde

1 = Client Hello, 2 = Server Hello, 3 = Key Exchange, 4 = finished

TLS 1.3: nur 1 Roundtrip oder sogar 0-RTT, dann aber ohne Perfect Forward Secrecy

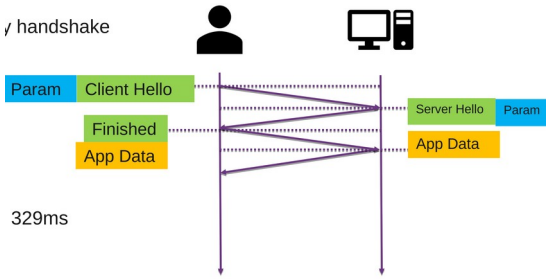
1 = Client Hello & Key Share, 2 = Server Hello & Key Share & Zertifikatsverifikation & Finished





8.3 QUIC & HTTP/3

QUIC: 1 RTT Verbindung + TLS Handshake, für bekannte Connections sogar 0 RTT



HTTP/2: Multiplexing ermöglicht es, verschiedene HTTP Requests über die gleiche TCP Verbindung zu senden, das hat aber auch einen Nachteil, das head-of-line blocking, wenn es nur bei einem Request packet loss gibt, sind alle Requests über diese TCP Verbindung betroffen, weil bei TCP die Ordnung eingehalten werden muss

QUIC bzw. HTTP/3 behebt dieses Problem, die Streams beeinflussen sich nicht gegenseitig

Allerdings verstehen NAT Router QUIC noch nicht, sie tracken die TCP Verbindungen (syn / ack / fin) in der NAT Tabelle, mit QUIC wissen sie nicht, wann eine Verbindung endet, somit gibt es timeouts und viele veraltete NAT Tabellen Entries. In HTTP/2 gab es die header compression HPACK, dabei müssen Sender und Empfänger ihre dynamischen Header Tables synchronisieren, das geht aber wegen der fehlenden Synchronisation in QUIC nicht. Es gibt ein neues header compression Schema QPACK für QUIC, dabei gibt es einen zusätzlichen QUIC Stream für die Header Table Updates. Andere TCP Optimierungen funktionieren auch nicht.

8.4 User Datagram Protocol UDP

gebraucht für DNS, Audio- und Video-Streaming, verbindungslos und simpel, keine Garantien für erfolgreiche Zustellung, Ordnung und keine Duplikate

8.5 Stream Control Transmission Protocol

Message-basiert, Daten können in mehrere Streams aufgeteilt werden, Syn Cookies (4 way handshake mit signed cookie), multi-homing mehrerer IP Adressen von Endpoints, selten verwendet und teilweise unsupported, WebRTC verwendet es tunneled in UDP

8.6 Vergleich

TCP *	UDP *	SCTP *	(QUIC) *
Transport layer	Transport layer	Transport layer	Transport layer*
Connection oriented	Connection less	Connection oriented	Connection oriented
Reliable transfer	Unreliable transfer	Reliable transfer	Reliable transfer
Streams	Messages	Messages	Multistream
Guaranteed order	Unordered	User can choose	Guaranteed order
Widely used – HTTP/1, HTTP/2	Widely used – DNS, HTTP/3	WebRTC	HTTP/3
Flow and congestion control	No flow, congestion	Flow and congestion control	Flow and congestion
Heavyweight	Lightweight	Heavyweight	Heavyweight*
Error checking and recovery	Error checking, no recovery	Error checking and recovery	Integrity check

8.7 DDoS Amplification Angriff

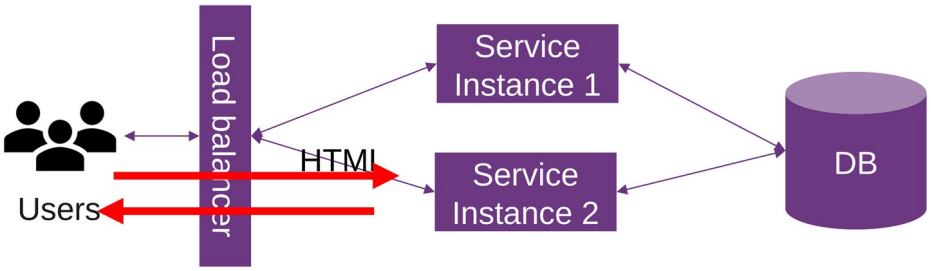
Dabei sendet der Angreifer z.B. mit dem Pentesting Tool hping3 einen UDP Request mit einem spoofed Absender. Der schlecht designte Server antwortet dann mit einer Antwort, die grösser als der Request ist, an den spoofed Absender.

9 Web Architektur

9.1 Server Side Rendering SSR

Server generiert HTML / JS / CSS dynamisch und sendet die Assets in Echtzeit zum Browser, der Vorteil davon ist SEO, der Nachteil davon ist, dass der Server für jeden Request gebraucht wird (kein Caching)

1. User sendet Request an Web Server (Server Side Routing)
2. Server führt Server Side Code aus
    1. Benötigte Daten werden von DB oder anderen Quellen fetched
    2. HTML wird von Template Engine (z.B. Handlebars) generiert (Reusability)
3. Server antwortet mit passendem HTML, CSS und JS für den Request



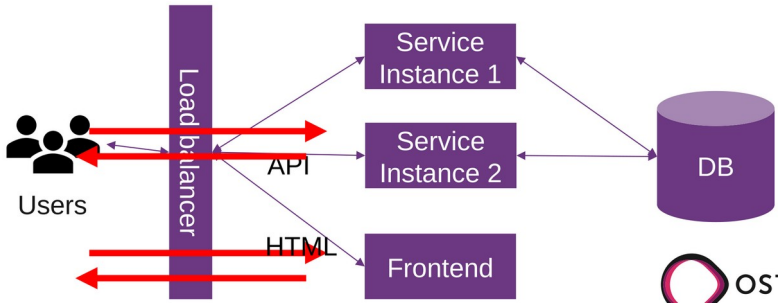
9.2 Static Site Generation SSG

HTML / CSS / JS ist pre-rendered und gleich für jeden User, das Rendering wird nur einmal bzw. bei Änderungen ausgeführt, kann auch DB Zugriff enthalten

9.3 Single Page Application SPA / Client Side Rendering CSR

Interaktionen passieren innerhalb einer einzigen Web Page, die Client Site updated sich dynamisch, wenn der User damit interagiert, das Benutzererlebnis ist flüssig und wie eine App, Umsetzung mit einem Framework wie React, Angular oder Vue, das Backend bearbeitet nur API Requests, SEO funktioniert nur, wenn die Search Engine JavaScript ausführt, man braucht CORS, wenn die API nicht das gleiche Protokoll, Domain oder Port hat

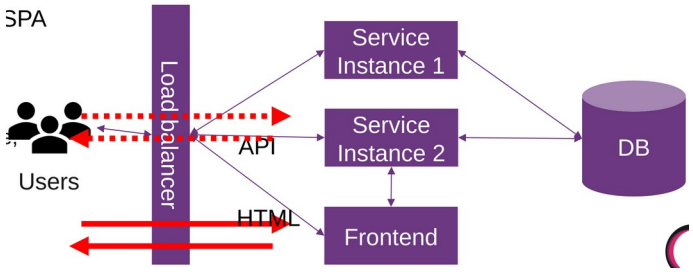
1. Browser sendet Request an Web Server für HTML / CSS / JS
2. Server antwortet mit einzigem HTML mit CSS & JavaScript, die JavaScript Files enthalten die ganze Applikationslogik
3. Browser zeigt HTML an (typischerweise ein Ladezeichen) und führt dann JavaScript aus
4. JavaScript macht die UI Updates, die Applikation braucht keine Page Reloads
5. Wenn die SPA Daten fetchen oder senden möchte, geschieht das über APIs
6. Das Routing auf andere Pfade wird ebenfalls auf dem Client gemacht



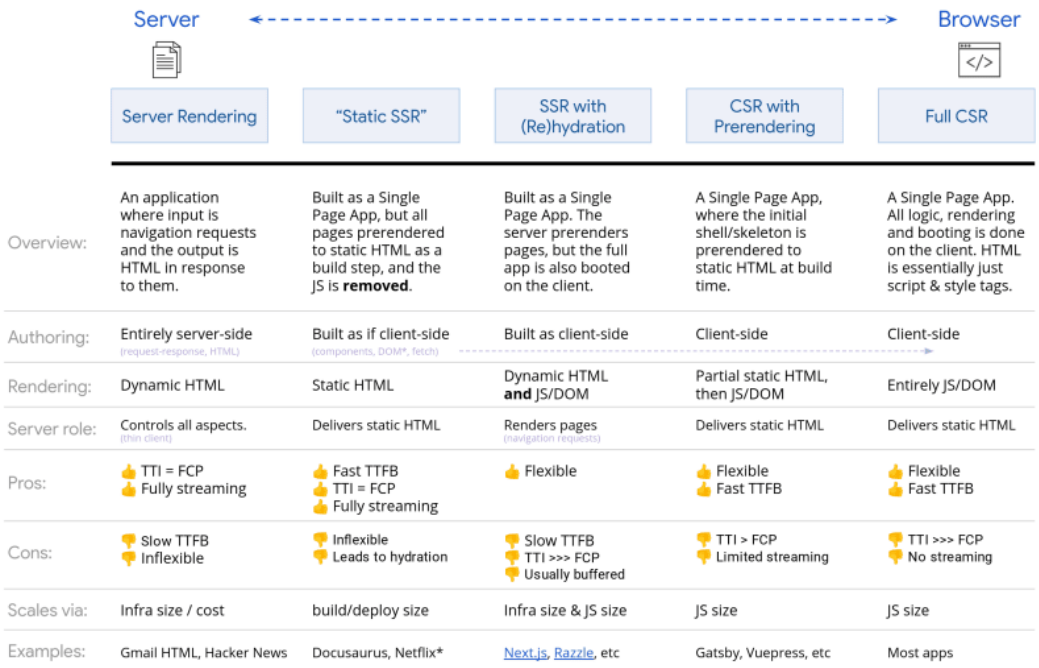
9.4 Hydration

State of the art, Kombination von SSR & SPA, intiales HTML nicht mit Ladezeichen, sondern erstem Inhalt im HTML wie beim SSR (pre-hydration), z.B. mit next.js, weiterer Zugriff auf APIs wie bei SPA

React Empfehlung: Framework wie next.js verwenden, traditioneller Weg mit Bundlers wie Vite oder CRA nicht mehr empohlen



9.5 Vergleich



	Server Rendering	"Static SSR"	SSR with (Re)hydration	CSR with Prerendering	Full CSR
	Server ← → Browser				
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 HTML, Hacker News	Docusaurus, Netflix*	Next.js, Razzle, etc	Gatsby, Vuepress, etc	Most apps

10 Deployment

10.1 Vergangenheit

Pakete wurde mit Paketmanager auf Linux installiert (apt, yum, pacman), für Java wurde ein Script unter /etc/init.d abgelegt

Probleme:

- "It works on my machine", wer installiert Java in der richtigen Version?
- Was passiert bei Crashes?
- Skalierung?
- Hardware defekt?
- Bei Misskonfiguration Zugriff auf ganzen Server?

10.2 Neuer Weg: Container

Isolation gegenüber Server, Java in der richtigen Version im Container, einfache Skalierung und Migration auf anderen Server, automatische Restarts

10.3 Strategien

- **Rolling Deployment**: Neue Version wird schrittweise deployed ohne das ganze System zu stoppen
  - + Minimale Downtime, niedriges Risiko
  - - Komplexität, lange Deployment Zeiten
- **Blue-Green Deployment**: 2 Environments, aktuelle Prod (blue), aktuelle Prod mit neuem Release (green), green testen und danach darauf umschalten
  - + Sofortiger Rollback, 0 downtime
  - - 2 Prod Environments, man muss sich um den Datensync kümmern
- **Canary Releases**: Neue Version für kleine Gruppe von Usern oder Servern, wenn alles gut läuft, restliche User oder Server freigeben
  - + Risikoreduzierung, User Feedback
  - - Komplexität, Inkonsistent
- **Feature Toggle**: Detailliertes Canary, Feature für bestimmte User-Gruppe freigeben
  - + Mehr Risikoreduzierung, spezifisches User-Feedback
  - - Codebase wird komplexer, Config Management benötigt
- **Big Bang**: alles auf einmal deployen
  - + Simpel

    - - Hohes Risiko, limitierter Rollback

10.4 Deployment in der Praxis

- **Ansible**: Playbook gegen SSH Hostliste ausführen, Hosts sollten gleiches OS haben, keine Agents auf den Hosts, Push-based
- **Chef / Puppet**: ähnlich wie Ansible
- **Docker Swarm**: funktioniert mit docker-compose.yml, simples Deployment auf mehrere Docker Hosts, built-in in Docker, Scheduler platziert Container auf den Nodes, ist aber deprecated, weil es von Kubernetes abgelöst wurde
- **Kubernetes**
- **Plain docker / podman**: Simpel, docker CLI kann auch zu remote Docker Hosts mit –context oder DOCKER_HOST Umgebungsvariable verbinden, podman ist etwas aufwändiger fürs Deployment, weil Systemd anstatt dem Daemon verwendet werden muss, alle Container und docker-compose funktionieren aber

10.5 Kubernetes

weit verbreitete Container Orchestrierung, automatisiertes Deployment, Skalierung und Management von containerisierten Applikationen, ursprünglich von Google, gehört aber nun der Cloud Native Computing Foundation, als PaaS auf Google, AWS, Azure, DigitalOcean, usw. verfügbar

Gründe für Kubernetes:

- Erleichtert Applikations-Deployment und Management
- Stellt high availability und fault tolerance sicher
- Unterstüzt auto-scaling basierend auf der Nachfrage
- Macht Rolling Updates und Rollbacks einfach
- grosses Ökosystem mit Tools und Services

Design Prinzipien:

- Deklarative Konfiguration mit YAML oder JSON
- Container sind immutable, keine State darin speichern, Kubernetes startet neuen Container wenn Healthcheck failed, für ältere Version muss Schema geändert werden

Architektur:

- **Master Node**: kontrolliert den Status des Clusters
    - API Server: Kommunikation mit dem Cluster
    - Etcd: speichert Konfigurationsdaten des Clusters
    - Controller manager: stellt Soll-Zustand des Clusters sicher
    - Scheduler: weist Workload einem passenden Worker Node zu

- **Worker Node**: darauf laufen die Applikations-Container
  - kubelet: kommuniziert mit den Master Nodes und verwaltet Container auf dem Node
  - kube-proxy: verwaltet Netzwerk-Routing und Load-Balancing
  - Container runtime: führt Container aus (containerd / cri-o)

Wichtigste Konzepte:

- **Pod**: kleinste deploybare Einheit, besteht aus einem oder mehreren Containern
- **Service**: stabiler Netzwerk-Endpoint, der mehrere Pods zusammenfasst
- **Deployment**: setzt den Soll-Zustand einer Applikation mit z.B. Skalierung, Hardware-Limits, Liveness Probe und Readiness Probe fest
- **ConfigMap**: enthält non-sensitive Konfigurationsdaten einer Applikation
- **Secret**: enthält sensitive Konfigurationsdaten einer Applikation, wie Passwörter
- **Volume**: Persistenter Storage für Daten eines Containers
- **Namespace**: Namespaces unterteilen einen Cluster in verschiedene Projekte

Tools:

- **Minikube**: Single-Node Cluster lokal ausführen
- **K3s**: Abgespeckte Kubernetes Distribution für Single- und Multi-Node Cluster inkl. HA
- **kubectl**: CLI Tool um Cluster zu verwalten
- **Dashboard**: grafisches User Interface, um einen Cluster zu verwalten

11 Bitcoin

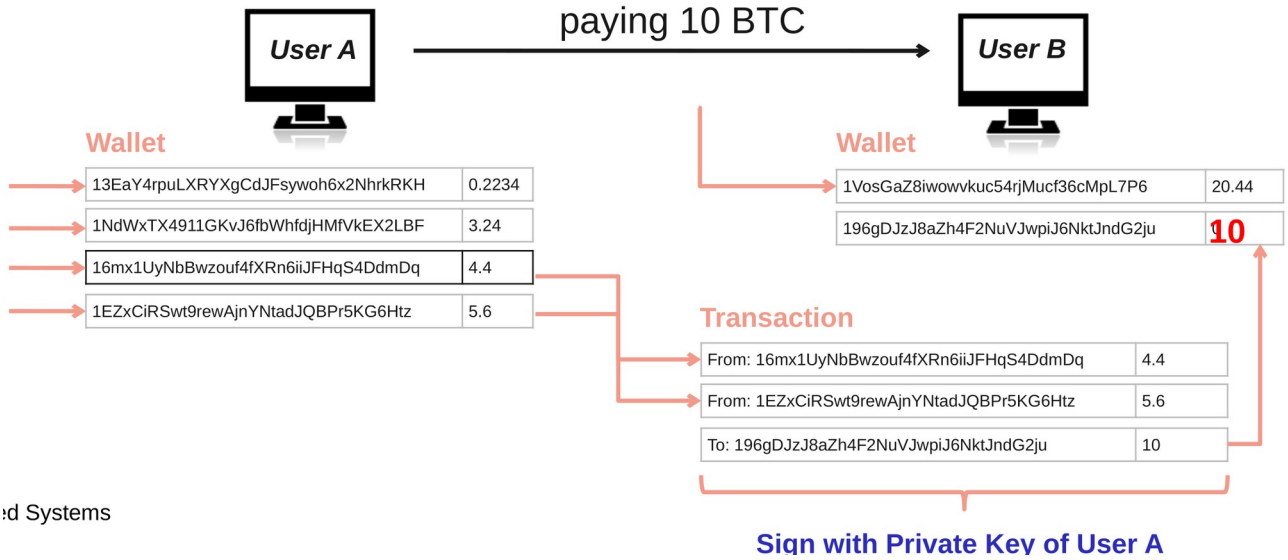
- Ist komplett Peer 2 Peer (keine zentrale Entität, Development community)
- Erster Bitcoin wurde am 3. Januar 2009 ausgestellt
- Erfinder ist unbekannt, Satoshi Nakamoto ist nur ein Pseudonym, er war der erste Miner und hat ca. 1 Million BTC, ist also reich
- Die kleinste Einheit ist 1 Satoshi (0.00000001 BTC)
- Es kann maximal ca. 21 Millionen BTC geben
- Jede Transaktion wird an alle Peers broadcastet, somit kennen alle Peers alle Transaktionen (ca. 570GB)
- Validierung mit proof-of-work (partielle Hash Kollision), schwierig zu fälschen, kein Double-Spending
- Boom hat in 2013 gestartet, wurde als gefährlich und tot beschrieben, heute sind auch grosse Banken, Visa, Mastercard und Nationalbanken dabei
- Man verlässt sich nicht auf Vertrauen, sondern starke Kryptografie
- Schwache Anonymität (Pseudonymität): alle Peers kennen alle Transaktionen, Clustering: wenn eine Transaktion mehrere Input-Adressen hat, nimmt man an dass diese Adressen zum gleichen Wallet gehören
- Es gibt Forks wie Bitcoin Cash oder SV
- BIP: Bitcoin Improvement Proposals
- Viele Firmen bieten den Umtausch von Bitcoin in echte Währungen wie EUR oder CHF an
- USA & CH sind Bitcoin-freundlich, China nicht
- 1 BTC entspricht momentan 58’749.50 USD
- Es wurden ca. 20 Mio. BTC mined
- Marktkapitalisierung ist 1.15 Billionen USD
- Rekord sind 450'000 Transaktionen pro Tag, 3 – 11 Transaktionen pro Sekunde
- Maximale Transaktionsgebühren waren 81M USD an einem Tag, als Bitcoin NFT lanciert wurde
- Netzwerk startete mit 155 PetaFLOPS (10^{15}) Hashrate in 2012 und hat heute 9.1 YottaFLOPS (10^{24}), zum Vergleich hat der schnellste Supercomputer 1700 PetaFLOPS
- Zum Teil Spread beim Wechselkurs

11.1 Mechanismen

- **Wallet** besteht aus public-private Keys (wallet.dat)

- **Public Key** ist 256 bit ECDSA, daraus kann die Bitcoin Adresse berechnet werden, wo man Bitcoins hinschicken kann: base58(RIPEM160(Sha256(ecdsa public key)))
- **Private Key** wird zum Signieren der Transaktion verwendet
- Wenn die Keys gestohlen werden, kann sie der Dieb die Coins verwenden
- Wenn die Keys verloren sind, sind die Coins verloren

- **Transaktion**
  - Peer A möchte BTC an Peer B senden, dazu erstellt er eine Transaction message
  - Transaction message enthält input (wo die BTC herkommen) / output (wo die BTC hingehen)
  - Peer A braodcastet die Transaktion an alle Peers im Netzwerk
  - Transaktionen werden in Blocks gespeichert, der Block wird erstellt und verifiziert, das dauert ca. 10 Minuten



- Double Spending verhindern
  - Transaktionen in Blocks sind bestätigt.
  - Nonce raten, der zusammen mit der Block-Nr und den Block-Daten einen Hash mit einer gewissen Anzahl 0-Bits (Difficulty) ergibt, das wird Crypto Puzzle genannt
  - Verkettete Proofs of work
- Generierung von Coins
  - Pro mined / erstelltem Block gibt es 3.125BTC
  - Irgendwann in 2028 werden es dann noch 1.5635 BTC sein

11.2 BIP39

- Ziel: Sicherheit von Wallets verbessern und einfaches Backup ermöglichen
- Schlüsselkomponente: Mnemonic phrase (human-readable seed), Hierarchical Deterministic (HD) wallets, mit phrase können Wallets generiert und recovered werden

1. Generate random entropy (128-256 bits)
2. Calculate SHA256 checksum (4, 8 bits)
3. Concatenate entropy & checksum (128bit random + 4 bit (msb of SHA256))
4. Divide into groups of 11 bits
5. Match each group with a predefined word from the BIP39 wordlist (2048 words)

11.3 Blockchain

Transaktionen werden in Blocks gesammelt, es gibt ca. alle 10 Minuten einen Block, Blöcke enthalten gelöste Crypto Puzzles in Form von partiellen SHA256 Hash Kollisionen, ein Block hat einen Pointer auf den letzten Block, neue Blocks erstellen heisst Mining

Es gibt verschiedene Levels von Bestätigungen, 3 – 6 Block Confirmations werden als sicher gesehen, wenn jemand mehr als 50% der Rechenleistung hat, kann er Transaktionen excluden und die Ordnung ändern

11.4 Mining Evolution

1. CPUs
2. GPUs
3. Field Programmable Gate Array (FPGA)
4. Application Specific Integrated Circuits (ASIC) Farms

11.5 Coins mit ähnlichen Verfahren

- Bitcoin: SHA256 partial hash collision: time, ASIC, electricity
- Litecoin: scrypt partial hash collision: time, GPU, memory, electricity
- Ethereum: Opcodes in Bitcoin, smart contracts in Ethereum
- Ripple XRP: Unique node list (trusted validators, 1000): web of trust
- Tezos, Ethereum: proof of stake: Holding / staking von 1% generiert z.B. 1% der Coins, energieeffizient

11.6 Nachteile

- Energieverbrauch wie Polen
- Skalierbarkeit: Bitcoin kann 7 Transaktionen pro Sekunde abwickeln, Visa 57'000
- Ist anonym und kann somit für illegale Aktivitäten benutzt werden
- Exchange rate ist volatil
- Es gibt zentrale Elemente wie Core Entwickler
- 51% Attacke: die Chain mit der meisten Compute power wächst am schnellsten und überholt die anderen Chains, wenn jemand mehr als die Hälfte der Compute power hat, kann

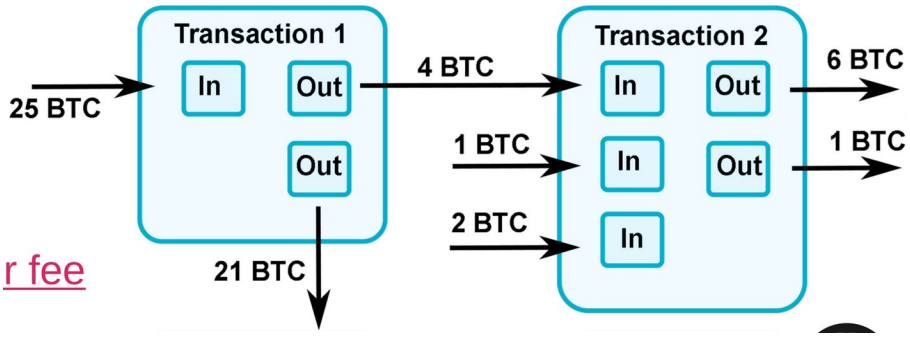
er seine falsche Chain validieren, so eine Attacke würde 5 – 20 Mrd. USD kosten, Double spending und Transaction-Roolbacks wären möglich

11.7 Vorteile

- Tiefe, fixe Transaktionskosten
- Skalierbar, weil Hardware und Storage schneller wird
- Ist anonym und schützt so die Privatsphäre
- Keine grossen Crashes
- Dezentralisiert: Protokoll ist offen und Forks sind möglich
- Weitere Blockchain Usecases wie Smart contracts

11.8 UTXO-based

- unspent transaction output
- Jeder referenzierte Input muss valid und noch nicht ausgegeben sein
- Summe der Inputs muss gleich wie die Summe der Outputs sein
- Es wird immer der ganze Output ausgegeben



12 Ethereum

- 1 ETH ist 3090 USD wert
- Ist eine generalisierte Blockchain mit Loops, Arithmetik, usw.
- White Paper wurde im Dezember 2013 veröffentlicht
- Protokolle wurden from scratch entworfen
- NPO Ethereum foundation ist in Zug ansässig
- Alle 12s wird ein Block erstellt, die Mining Belohnung ist 3%
- Vitalik Buterin ist Gründer
- Aktueller Release ist Dencun, Homestead war erster Stable Release in 2016
- Marktkapitalisierung ist 350 Mrd. USD
- 196 Mio. Transaktionen pro Tag, durchschnittlich 23 pro Sekunde
- 7500 Nodes
- 90 – 270KB Blocksize
- 270 Mio. Accounts

12.1 Vergleich mit Bitcoin

- Bitcoin implementiert neue Features sehr langsam, daher gibt es viele Hard Forks
- Bitcoin Script ist limitiert
- Weder Bitcoin noch Ethereum sind in allem besser als der andere

12.2 Mechanismen

- Smart Contracts sind turing complete, jede Instruktion muss mit Gas bezahlt werden, die Anzahl ist höher oder tiefer je nach Komplexität der Instruktion
- Der Gas price wird der Transaktion mitgegeben, Miner entscheiden, ab welchem Gas price sie Transaktionen durchführt, somit werden Transaktionen mit höherem price schneller im Block aufgenommen und durchgeführt

12.3 Einheiten

1 ether =	
1000000000000000000	wei
1000000000000000	Kwei
1000000000000	Mwei
1000000000	Gwei
1000000	szabo
1000	finney
1	ether
0.001	Kether
0.000001	Mether
0.000000001	Gether
0.000000000001	Tether

12.4 Ethereum Virtual Machine (EVM)

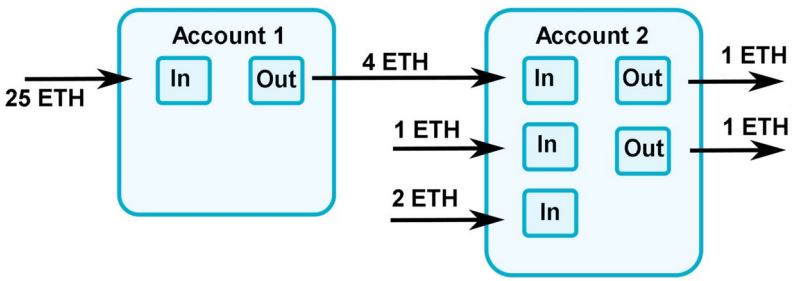
- Die EVM ist grundsätzlich ein globaler Computer, der immer läuft und immer korrekt ist
- Jeder Contract wird auf jedem vollwertigen Ethereum Node berechnet, die Nodes vergleichen dann ihre Resultate, daher sind die Transaktionsgebühren eher hoch

12.5 Accounts

- **Externally controlled accounts**: kann Ether haben und senden, kontrolliert mit Private Keys
- **Contract accounts**: kann Ether haben und senden, kontrolliert von Code, alle Aktionen werden von externally controlled accounts ausgelöst

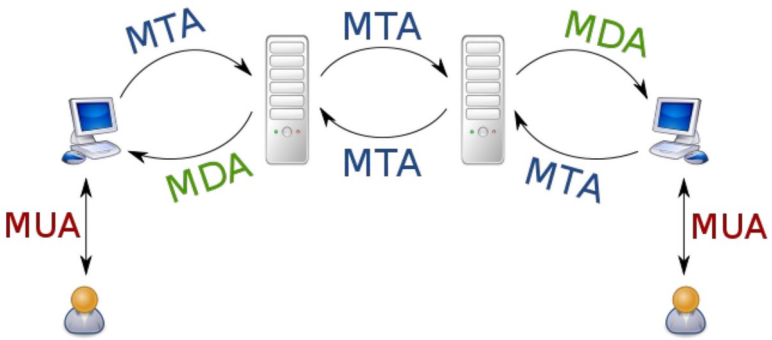
12.6 Account-based

- Globaler Status speichert eine Liste von Accounts mit Account-Balances und/oder Code
- Transaktion ist valid, wenn der sendende Account eine genug grosse Balance hat
- Balance des Senders wird subtrahiert



13 Mail

- Mail User Agent MUA: Email Client wie Thunderbird
- Mail Transfer Agent MTA: Mail Server mit SMTP
- Mail Delivery Agent MDA: erhält Mails vom MTA und liefert sie in die Mailbox des Empfängers (IMAP / POP3), wenn er zuständig für die Domain ist
- SMTP / IMAP / POP3 haben an Popularität eingebüsst, weil nun oft Webmail verwendet wird
- Im DNS braucht es einen MX Record, um den MTA der Domain zu setzen



13.1 Simple Mail Transfer Protocol SMTP

Standard Protokoll, um Mails vom Mailprogramm zu versenden und zur Kommunikation zwischen MTAs

Ports:

- 25 für unverschlüsselte Kommunikation (typischerweise zwischen MTAs)
- 587 für verschlüsseltes Versenden von Mails (typischerweise zwischen Client und MTA)
- 465 für SMTP over SSL (SMTPS), deprecated

STARTTLS:

- STARTTLS auf Port 25 & 587 kann eine unverschlüsselte Verbindung auf eine verschlüsselte Verbindung upgraden auf dem gleichen Port
- Man startet eine normale SMTP Session und macth das Upgrade nur, wenn beide Parteien TLS unterstützen
- Vorteile: rückwärtskompatibel, compliant mit vielen Standards, generell empfohlen
- Nachteile: potentiall anfällig für Downgrade Attacken
- Let's Encrypt kann verwendet werden

SMTPS:

- Enkapsuliert ganze SMTP Session in TLS von Anfang an
- Vorteile: Einfachheit (Verbindung ist immer verschlüsselt), garantierte Verschlüsselung
- Nachteile: Legacy, Port wurde revived wegen verbreitetem inoffizieller Verwendung

- Let's Encrypt kann verwendet werden

13.2 JSON Meta Application Protocol JMAP

Kombiniert IMAP (Mails abrufen), CardDav (Kontakte) und CalDav (Kalender)

13.3 Internet Message Access Protocol IMAP

Stellt ein Netzwerkdateisystem für Mails bereit. Benutzer können ihre Mails, Ordnerstrukturen und Einstellungen auf den (Mail-)Servern speichern und belassen.

13.4 Post Office Protocol POP3

Über dieses Protokoll kann ein E-Mail Client Emails von einem Email-Server abholen. Die Funktionalität ist sehr beschränkt und erlaubt nur das Auflisten, Abholen und Löschen von E-Mails am E-Mail-Server.

13.5 Spam Prevention

- **Greylisting**: Mails von neuen IP-Adressen oder Absenderadressen werden inital rejected mit "try again later", legitime Mailserver probieren es nach einem Delay nochmals, Spammer probieren es nicht nochmals, nach dem Retry wird der Absender von der Greylist entfernt und Mails werden sofort akzeptiert
  Vorteile: reduziert Spam effektiv, niedriger Ressourcenverbraucht, einfache Umsetzung
  Nachteile: legitime Mails werden delayed, Spammer können ihre Server adaptieren
- **SURBL Filter**: URLs in Emails werden gescannt und gegen Echtzeit-Blacklists abgegelichen
- **DNSBL**: MTA vergleicht IPs, welche Mails an ihn senden möchten, mit Echtzeit-Blacklists mit bekannten IPs, die Spam senden
- **Bayesian Analysis**: Machine Learning klassifiziert Mails aufgrund ihres Inhalts, die KI wurde mit vielen Spam und Nicht-Spam Mails trainiert, Wörter wie discount und offer haben höhere Wahrscheinlichkeit, Spam zu sein
- **Sender Policy Framework SPF**: Domain Owner spezifiert IPs, welche Mails von dieser Domain senden dürfen, in Form des SPF DNS Eintrags, der Empfänger kann dann den Eintrag prüfen, wenn er ein Mail von dieser Domain erhält
- **Domain Keys Identified Mail DKIM**: Mails werden mit Private Key siginiert, Public Key ist im DKIM DNS Record und Empfänger können die Signatur verifizieren
- **Domain-based Message Authentication Reporting and Conformance DMARC**: basiert auf SPF und DKIM, fügt Reporting Funktion hinzu, mit welcher Empfänger Feedback an den Sender senden können, Sender kann definieren, wie Mails behandelt werden sollen, die SPF/DKIM failen (reject, quarantine, allow)
- **Brand Indicators for Message Identification BIMI**: standardisiertes Logo im SVG Format im TXT Format im DNS publizieren, braucht verified mark certificate (VMC)