Technische Hochschule Ingolstadt
Fakultät Informatik

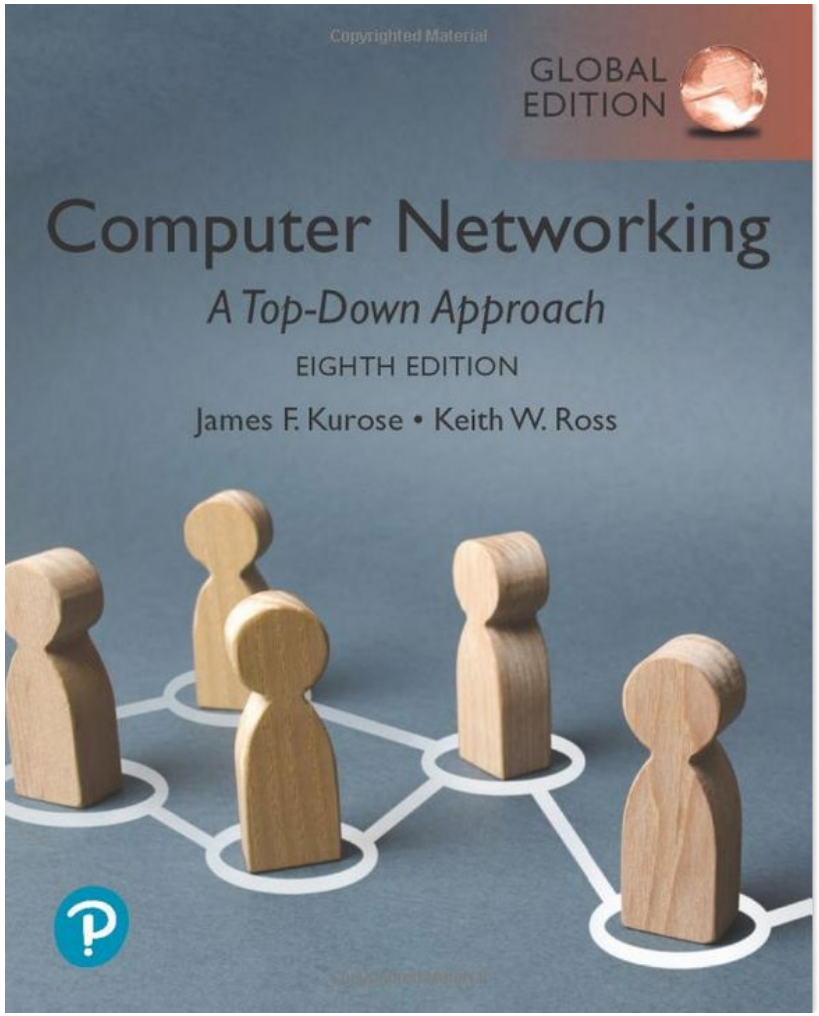
Kapitel 2: Applikationsschicht

FFI_NW WS 2024

Vorlesung „Netzwerke“

18.09.2024

Der Inhalt des Foliensatzes basiert auf bzw. ist adaptiert aus:



Computer Networking: A Top-Down Approach

8th edition [Global Edition]
Jim Kurose, Keith Ross
Pearson, 2021

ISBN-10 : 1292405465
ISBN-13 : 978-1292405469

Mehrere Ausgaben (auch deutsche Editionen) in der Bibliothek verfügbar

Applikationsschicht: Übersicht

- **Prinzipien vernetzter Anwendungen**
- Web und HTTP
- Das Domain Name System DNS
- P2P Applikationen
- Video Streaming und Content Distribution Networks
- Socket Programmierung mit UDP und TCP

Unsere Ziele:

- Konzeptuelle- und Implementations-Aspekte von Applikationsschicht-Protokollen
- Client-Server Paradigma
- Peer-to-Peer Paradigma
- Wichtige Applikationsschichtprotokolle und Infrastruktur
  - HTTP
  - DNS
  - Video Streaming Systeme, CDNs
- Programmieren von Netzanwendungen
  - Socket API

Einige Netzanwendungen

- Soziale Netze
- Web
- Textnachrichten
- E-Mail
- Multiplayer Games
- Streaming (YouTube, Netflix, Amazon Prime)
- P2P Dateiaustausch
- Voice over IP (z.B. Skype)
- Echtzeitvideokonferenzen (z.B. Zoom)
- Internet Suche
- Remote Login
- …

Frage: Was sind Ihre Favoriten?

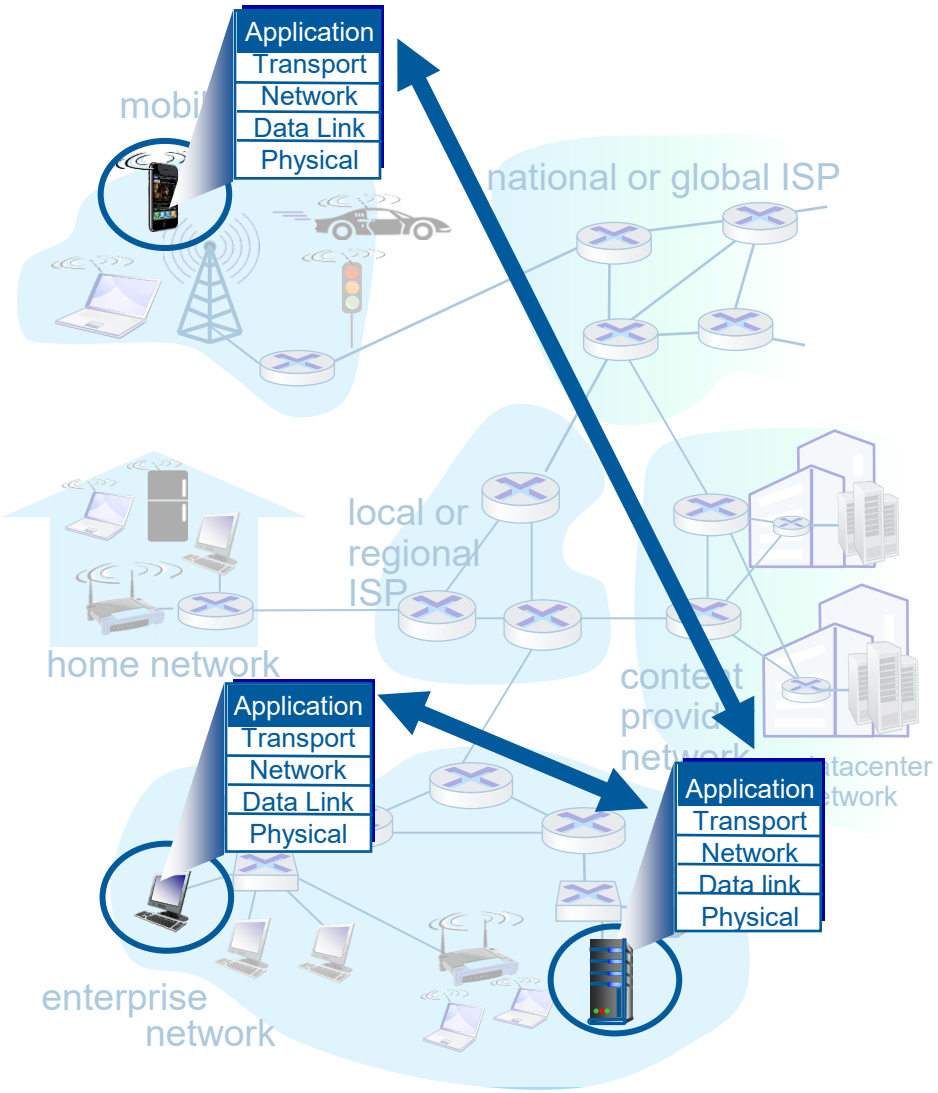
Erstellen einer Netzanwendung

Schreiben von Programmen, die:

- auf (verschiedenen) Endsystemen laufen
- über das Netz kommunizieren
- z.B. Webserver Software, die mit Browser Software kommuniziert

Keine Notwendigkeit Code für Zwischenknoten zu schreiben

- Nutzeranwendungen laufen nicht auf Geräten im Kernnetz
- Applikationen auf Endsystemen können schnell entwickelt und verbreitet werden



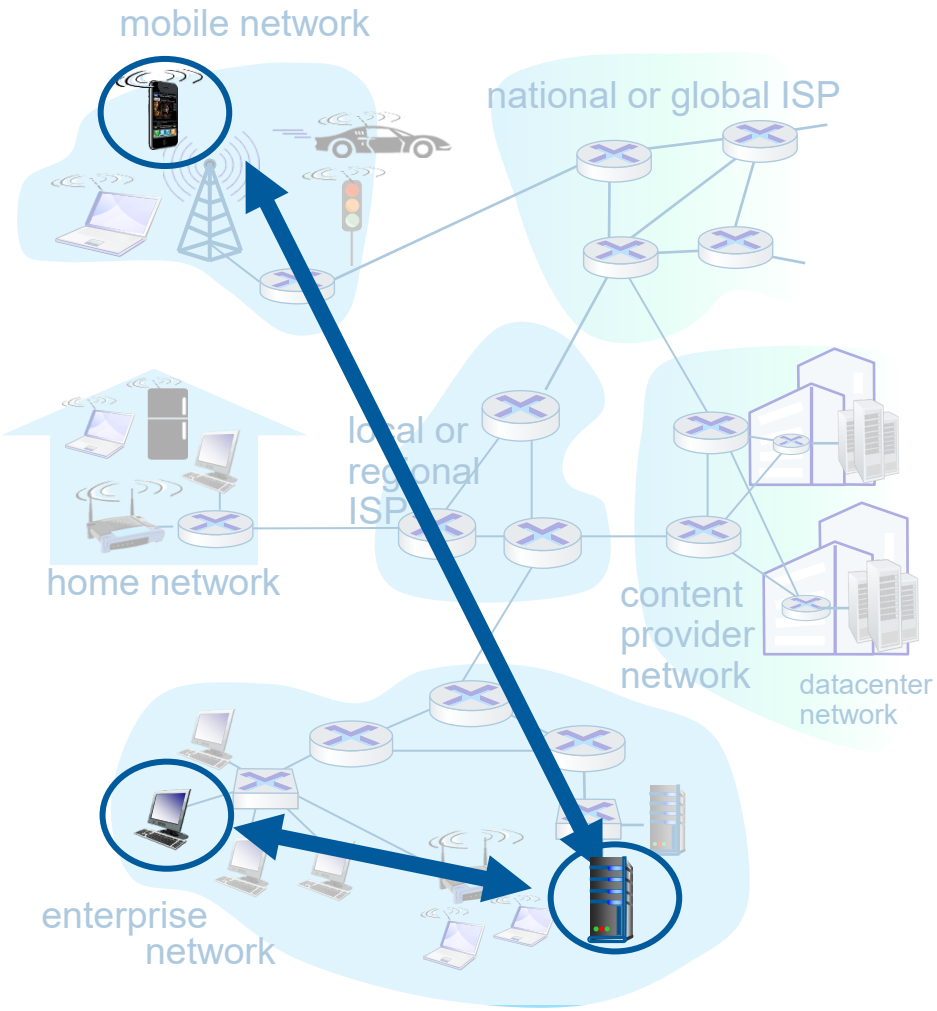
Client-Server Paradigma

Server:

- Immer aktiver Host
- Permanente IP-Adresse
- Oft aus Skalierungsgründen in Datenzentren

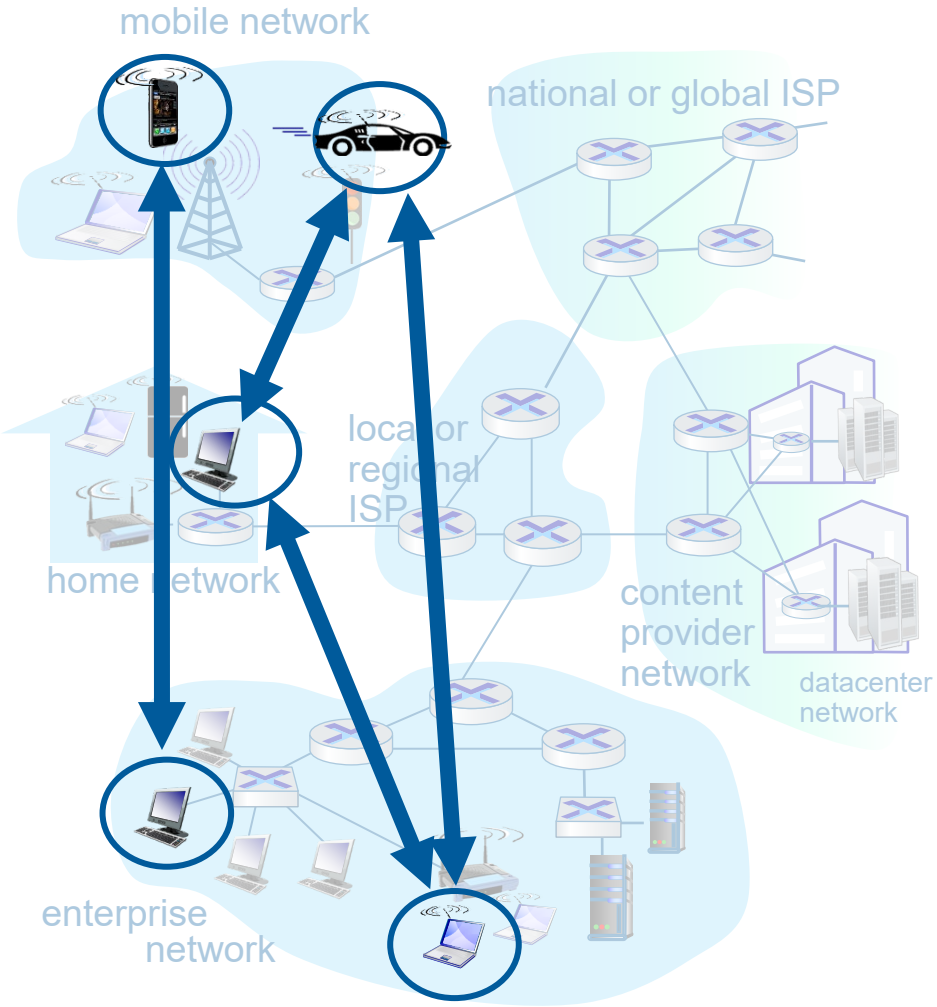
Clients:

- Kontaktieren, kommunizieren mit Server
- Können mit Unterbrechungen Verbunden sein
- Können dynamische IP-Adressen haben
- Kommunizieren *nicht* direkt miteinander
- Beispiele: HTTP, IMAP, FTP



Peer 2 Peer Architektur

- Kein immer aktiver Server
- Beliebige Endsystem kommunizieren direkt
- Peer fragen Dienst bei anderen Peers an und bieten als Gegenleistung ebenfalls einen Dienst an
  - ***Skaliert von selbst* – neue Peers bringen sowohl neue Dienstkapazität als auch höhere Nachfrage**
- Peers sind mit Unterbrechungen verbunden und können IP-Adressen ändern
  - komplexes Management
- Beispiel: P2P Dateiaustausch (BitTorrent), Skype (früher)



Prozess: ein Programm, das auf einem Host läuft

- Innerhalb desselben Hosts kommunizieren zwei Prozesse mittels **Inter-Prozess Kommunikation** (definiert durch das Betriebssystem)
- Prozesse in verschiedenen Hosts kommunizieren über den Austausch von **Nachrichten**

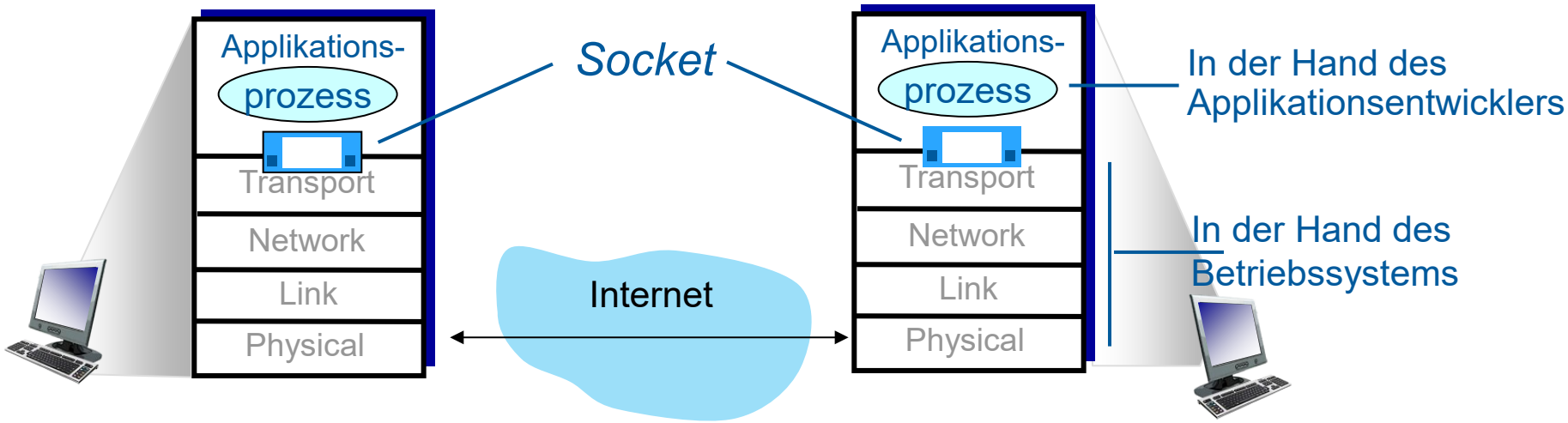
Clients, Server

Client-Prozess: Prozess der Kommunikation initiiert

Server-Prozess: Prozess der auf Kontaktanfrage wartet

- Hinweis: Anwendungen mit P2P-Architekturen haben Client-Prozesse und Server-Prozesse

- Prozess sendet/empfängt Nachrichten an/von seinem **Socket**
- Socket-Analogie: Tür
  - Sendeprozess schiebt Nachricht vor die Tür
  - Der Sendeprozess ist auf die Transportinfrastruktur auf der anderen Seite der Tür angewiesen, um die Nachricht beim Empfangsprozess an den Socket zu übermitteln
  - Zwei Sockets: einer auf jeder Seite



Adressieren von Prozessen

- Um Nachrichten zu empfangen, muss der Prozess über eine **Kennung** verfügen.
- Das Host-Gerät verfügt über eine eindeutige 32-Bit-IP-Adresse
- **Frage:** Reicht die IP-Adresse des Hosts, auf dem der Prozess ausgeführt wird, zur Identifizierung des Prozesses aus?
  - **Antwort:** Nein, viele Prozesse können auf demselben Host ausgeführt werden

- Die **Kennung** enthält sowohl die **IP-Adresse** als auch die **Portnummern**, die dem Prozess auf dem Host zugeordnet sind.
- Beispiele für Portnummern:
  - HTTP-Server: 80
  - Mailserver: 25
- So senden Sie eine HTTP-Nachricht an gaia.cs.umass.edu Webserver:
  - **IP-Adresse:** 128.119.245.12
  - **Portnummer**: 80
- mehr in Kürze...

Ein Applikationsschichtprotokoll definiert:

- **Die Art der ausgetauschten Nachrichten**
  - z.B. Anfrage, Antwort
- **Nachrichtensyntax:**
  - Welche Felder es in Nachrichten gibt und wie diese beschrieben werden
- **Nachrichtensemantik**
  - Bedeutung der Informationen in den Feldern
- **Regeln** für das Wann und Wie Prozesse Nachrichten senden & empfangen

Offene Protokolle:

- definiert in RFCs, jeder hat Zugang zur Protokolldefinition
- ermöglicht Interoperabilität
- z.B. HTTP, SMTP

Propietäre Protokolle:

- z.B. Skype, Zoom

Welchen Transportdienst benötigt eine Anwendung?

Datenintegrität

- Einige Apps (z. B. Dateiübertragung, Webtransaktionen) erfordern eine 100% zuverlässige Datenübertragung
- Andere Apps (z. B. Audio) können einen gewissen Verlust tolerieren

Durchsatz

- Einige Apps (z. B. Multimedia) erfordern einen Mindestdurchsatz, um "effektiv" zu sein
- Andere Apps ("elastische Apps") nutzen den Durchsatz, den sie erhalten

Timing

- Einige Apps (z. B. Internettelefonie, interaktive Spiele) erfordern eine geringe Verzögerung, um "effektiv" zu sein

Sicherheit

- Verschlüsselung, Datenintegrität, ...

Transportdienstanforderungen: Verbreitete Applikationen

Applikation	Datenverlust	Durchsatz	Latenz-empfindlich?
Dateitransfer/Download	kein Verlust	variabel	nein
E-mail	kein Verlust	variabel	nein
Webdokumente	kein Verlust	variabel	nein
Echtzeit-Audio/Video	Verlust-tolerant	Audio: 5Kbps-1Mbps Video:10Kbps-5Mbps	ja, 10x ms
Streaming-Audio/Video	Verlust-tolerant	wie oben	ja, wenige Sek.
Interaktive Spiele	Verlust-tolerant	Kbps+	ja, 10x ms
Textnachrichten	kein Verlust	variabel	ja und nein

Dienste der Internet-Transportprotokolle

TCP-Dienst:

- **Zuverlässiger Transport** zwischen Sende- und Empfangsprozess
- **Flusskontrolle:** Der Absender überfordert den Empfänger nicht
- **Überlastungskontrolle:** Drosselung des Senders bei Überlastung des Netzwerks
- **verbindungsorientiert:** Verbindungsaufbau zwischen Client- und Serverprozessen erforderlich
- **Bietet nicht:** Timing, Mindestdurchsatzgarantie, Sicherheit

UDP-Dienst:

- **Unzuverlässige Datenübertragung** zwischen Sende- und Empfangsprozess
- **Bietet nicht:** Zuverlässigkeit, Flusskontrolle, Überlastungskontrolle, Timing, Durchsatzgarantie, Sicherheit oder Verbindungseinrichtung.

Frage:

Warum sich die Mühe machen?
Warum gibt es UDP?

Applikation	Applikationsschichtprotokoll	Transport Protokoll
Dateitransfer/Download	FTP [RFC 959]	TCP
E-mail	SMTP [RFC 5321]	TCP
Webdokumente	HTTP 1.1 [RFC 7320]	TCP
Internet-Telefonie	SIP [RFC 3261], RTP [RFC 3550], oder proprietär	TCP oder UDP
Streaming-Audio/Video	HTTP [RFC 7320], DASH	TCP
Interaktive Spiele	WOW, FPS (proprietär)	UDP oder TCP

Applikationsschicht: Übersicht

- Prinzipien vernetzter Anwendungen
- **Web und HTTP**
- Das Domain Name System DNS
- P2P Applikationen
- Video Streaming und Content Distribution Networks
- Socket Programmierung mit UDP und TCP

- Eine Webseite besteht aus **Objekten**, die jeweils auf anderen Webservern liegen können
- Ein Objekt kann eine HTML-Seite sein, ein JPEG Bild, ein Java-Applet, eine Audiodatei,…
- Die Webseite besteht aus einer **Basis HTML-Datei**, die **mehrere Referenzobjekte** referenziert, wovon jede durch eine **URL** adressierbar ist, z.B.,

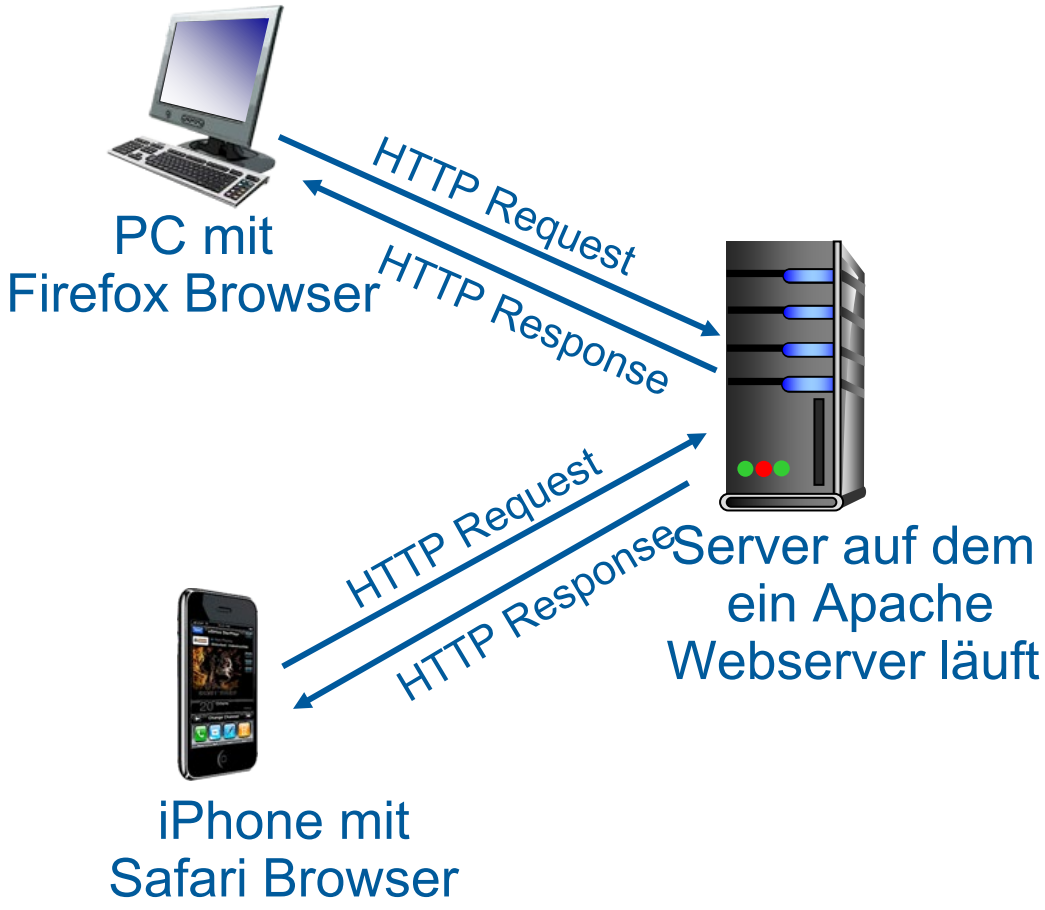
`www.thi.de/typo3conf/ext/in2template/Resources/Public/Images/Site/thi_logo_wb_RGB.svg`

Host Name

Pfad Name

HTTP: Hypertext Transfer Protokoll

- Applikationsschichtprotokoll des Webs
- Client/Server Modell:
  - **Client:** Browser der Webobjekte (mittels HTTP-Protokoll) anfragt, empfängt und “darstellt”
  - **Server:** Webserver sendet Objekte (mit HTTP-Protokoll) auf Anfrage



HTTP-Überblick (Fortsetzung)

HTTP nutzt TCP:

- Client initiiert TCP-Verbindung zum Server auf Port 80 (erstellt Socket)
- Server akzeptiert TCP-Verbindung vom Client
- HTTP-Nachrichten (Applikationsschicht-Protokoll Nachrichten) werden zwischen Browser (HTTP-Client) und Webserver (HTTP-Server) ausgetauscht
- TCP-Verbindung geschlossen

HTTP ist “zustandslos”

- Server hält *keine* Informationen über frühere Clientanfragen

Am Rande

Protokolle die “Zustand” halten sind komplex! → siehe TCP

- Vergangenheit (Zustand) muss behalten werden
- wenn Server/Client abstürzen, kann ihr “Zustand” inkonsistent sein und muss angeglichen werden

HTTP-Verbindungen: Zwei Arten

Nicht-persistentes HTTP

1. TCP-Verbindung geöffnet
2. Maximal ein Objekt wird über die TCP-Verbindung geschickt
3. TCP-Verbindung geschlossen

Herunterladen mehrerer Objekte erfordert mehrere Verbindungen

Persistentes HTTP

- TCP-Verbindung zu einem Server geöffnet
- Mehrere Objekte können über eine Verbindung zwischen Client und diesem Server übertragen werden
- TCP-Verbindung geschlossen

Nicht-persistentes HTTP: Beispiel

Nutzer gibt URL ein: **https://www.thi.de/fakultaet-informatik/index.html**
(beeinhaltet Text, Referenzen zu 10 JPEG Bildern)

1a. HTTP-Client initiiert TCP-Verbindung zu HTTP-Server (Prozess) bei https://www.thi.de/ auf Port 80

1b. HTTP-Server auf Host https://www.thi.de/ wartet auf TCP-Verbindung auf Port 80, "akzeptiert" Verbindung, benachrichtigt Client

2. HTTP-Client sendet HTTP **Request Nachricht** (mit URL) in den TCP Verbindungs-Socket. Die Nachricht gibt an, dass Client das Objekt "fakultaet-informatik/index.html" will

3. HTTP-Server empfängt Request, erstellt **Response Nachricht** die das angefragte Objekt beinhaltet und sendet sie in den Socket

Zeit

Nicht-persistentes HTTP: Beispiel (Fortsetzung)

Nutzer gibt URL ein: **https://www.thi.de/fakultaet-informatik/index.html**
(beeinhaltet Text, Referenzen zu 10 JPEG Bildern)

4. HTTP-Server schließt TCP-Verbindung.

5. HTTP-Client empfängt Response Nachricht mit HTML-Datei, zeigt HTML. Beim Parsen der HTML-Datei, werden 10 referenzierte JPEG-Objekte gefunden

6. Schritte 1-5 werden für jedes der 10 JPEG-Objekte wiederholt

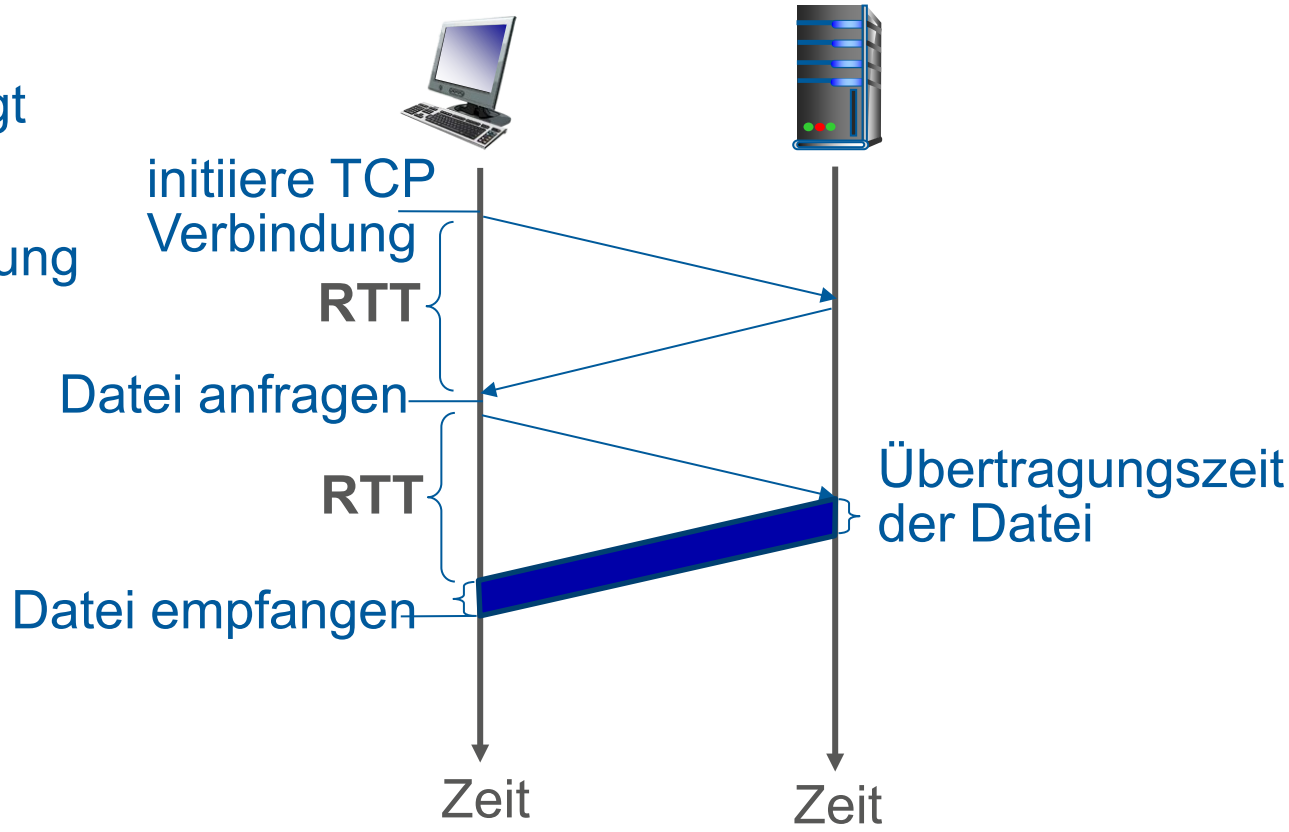
Zeit

Nicht-persistentes HTTP : Antwortzeit

RTT (Definition): Zeit, die ein kleines Paket vom Client zum Server und zurück benötigt

HTTP Antwortzeit (pro Objekt):

- eine RTT zum Initiieren der TCP-Verbindung
- eine RTT für HTTP Request und bis die ersten paar Byte der HTTP-Response ankommen
- Objekt/Dateiübertragungszeit



Nicht-persistentes HTTP Antwortzeit = 2RTT+ Dateiübertragungszeit

Persistentes HTTP (HTTP 1.1)

Probleme von Nicht-persistentem HTTP:

- benötigt 2 RTTs pro Objekt
- OS-Overhead für *jede* TCP-Verbindung
- Browser öffnen oft mehrere parallele TCP-Verbindungen, um referenzierte Objekte parallel abzuholen

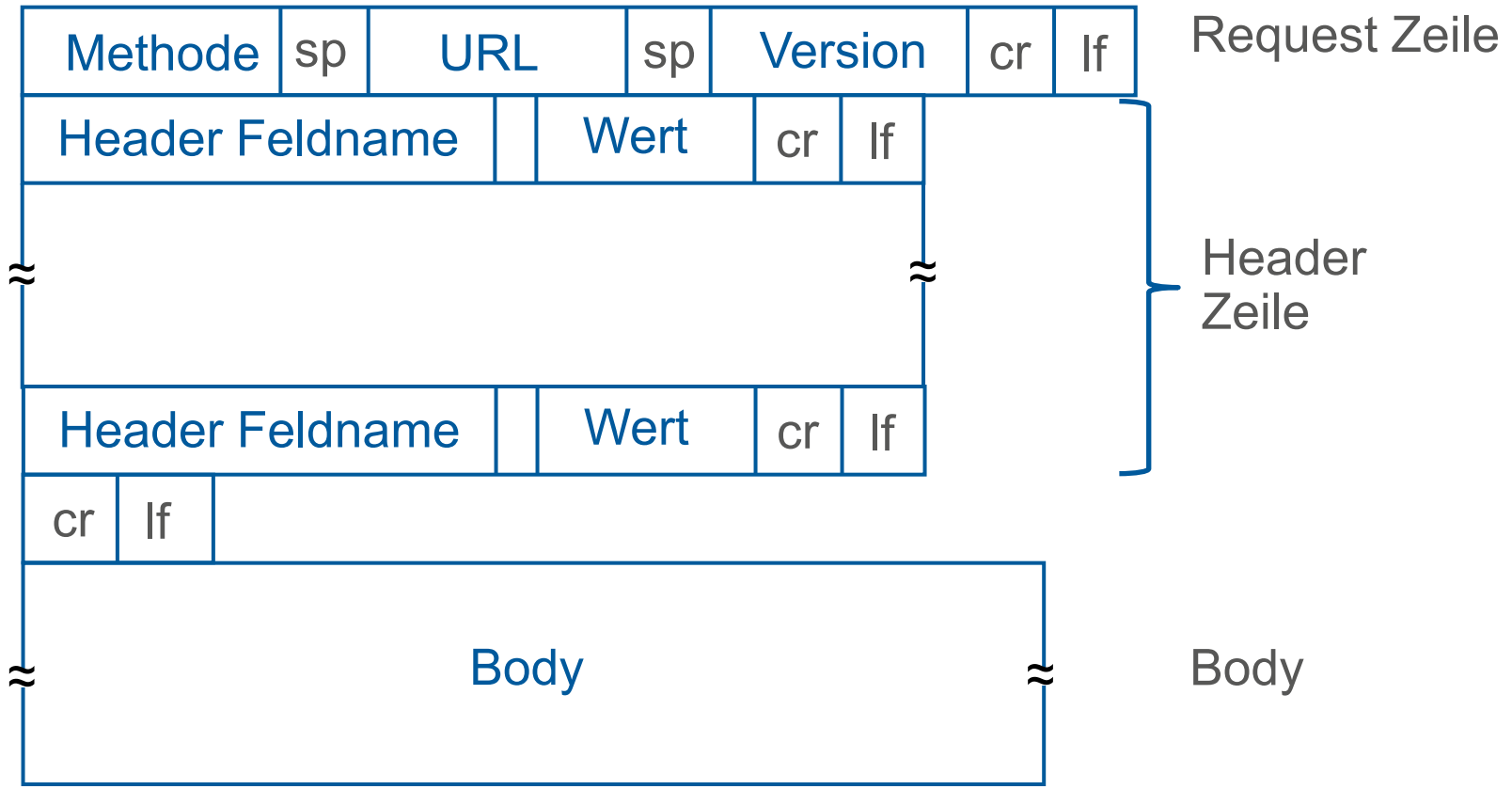
Persistentes HTTP (HTTP1.1):

- Der Server lässt die Verbindung offen, nachdem er die Antwort gesendet hat
- Nachfolgende HTTP-Nachrichten zwischen demselben Client/Server nutzen die offene Verbindung
- Client sendet Anfragen sobald ihm ein referenziertes Objekt begegnet
- nur eine RTT für alle referenzierten Objekte (halbieren der Antwortzeit)

HTTP Request Nachricht

- Zwei Arten von HTTP-Nachrichten: **Request, Response**
- **HTTP Request Nachricht:**
  - ASCII (Menschen-lesbares Format)

HTTP Request Nachricht: Allgemeines Format



Weitere HTTP Request Nachrichten

POST-Methode:

- Webseite enthält oft Formulareingabefelder
- Nutzereingabe wird vom Client zum Server im Body einer HTTP POST Request Nachricht

GET-Methode (zum Senden von Daten an Server):

- Einbauen von Nutzerdaten in das URL-Feld einer HTTP GET Request Nachricht (hinter einem '?'):

https://moodle.thi.de/course/view.php?id=8432

HEAD-Methode:

- fragt (nur) Header an, die geliefert würden *falls* eine spezifische URL mit der HTTP GET Methode angefragt worden wäre.

PUT-Methode:

- Lädt neue Datei (Objekt) auf den Server
- Ersetzt die Datei, die bei der spezifizierten URL liegt vollständig mit dem Inhalt des Bodys des POST HTTP Request

HTTP Response Nachricht

Status Zeile (Protokoll Status Code Status Phrase)

Header Zeilen

Daten, z.B. angeforderte HTML-Datei

```
HTTP/1.1 200 OK
Date: Tue, 08 Sep 2020 00:53:20 GMT
Server: Apache/2.4.6 (CentOS) OpenSSL/1.0.2k-fips
    PHP/7.4.9 mod_perl/2.0.11 Perl/v5.16.3
Last-Modified: Tue, 01 Mar 2016 18:57:50 GMT
ETag: "a5b-52d015789ee9e"
Accept-Ranges: bytes
Content-Length: 2651
Content-Type: text/html; charset=UTF-8
\r\n
data data data data data ...
```

HTTP Response Status Codes

- Status Code erscheint in der 1ten Zeile der Server-an-Client Response Nachricht.
- einige ausgewählte Codes:

  200 OK
  - Request erfolgreich, angefordertes Objekt später in dieser Nachricht

  301 Moved Permanently
  - angefragtes Objekt verschoben, neuer Ort später in dieser Nachricht

  400 Bad Request
  - Request Nachricht vom Server nicht verstanden

  404 Not Found
  - angefragtes Dokument konnte nicht auf dem Server gefunden werden

  505 HTTP Version Not Supported

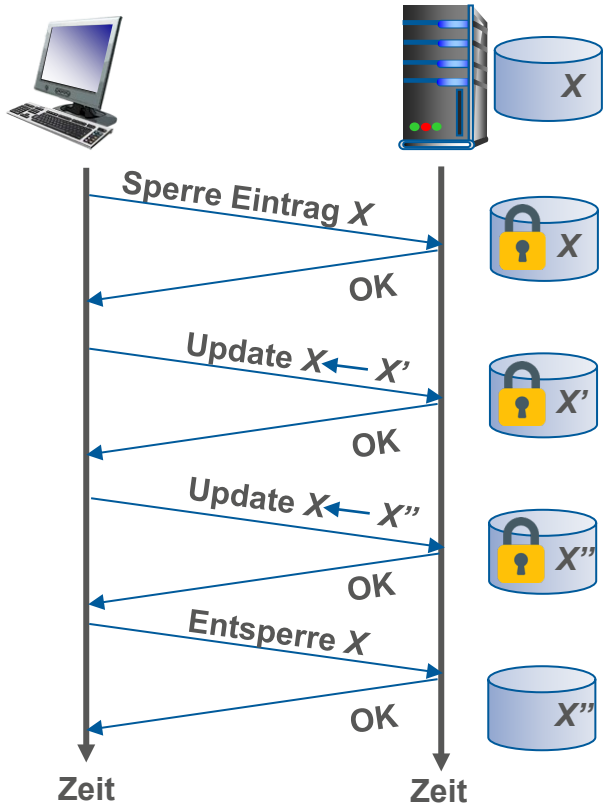
- Vollständige Liste: https://http-status-code.de/

Halten von Nutzer/Server Zustand: Cookies

HTTP GET/Response Interaktion ist **zustandslos**

- kein Konzept von mehr-stufigem Austausch von HTTP Nachrichten zum abschließen einer Web-“Transaktion”
  - kein Bedarf für Client/Server den “Zustand” eines mehrstufigen Austauschs zu verfolgen
  - alle HTTP Requests sind voneinander unabhängig
  - kein Bedarf für Client/Server sich um unerwartet beendete Verbindungen zu kümmern.

Ein zustandsbehaftetes Protokoll: Client macht zwei Änderungen an X oder keine



Halten von Nutzer/Server Zustand: Cookies

Webseiten und Client Browser nutzen **Cookies** um Zustand zwischen Transaktionen zu halten

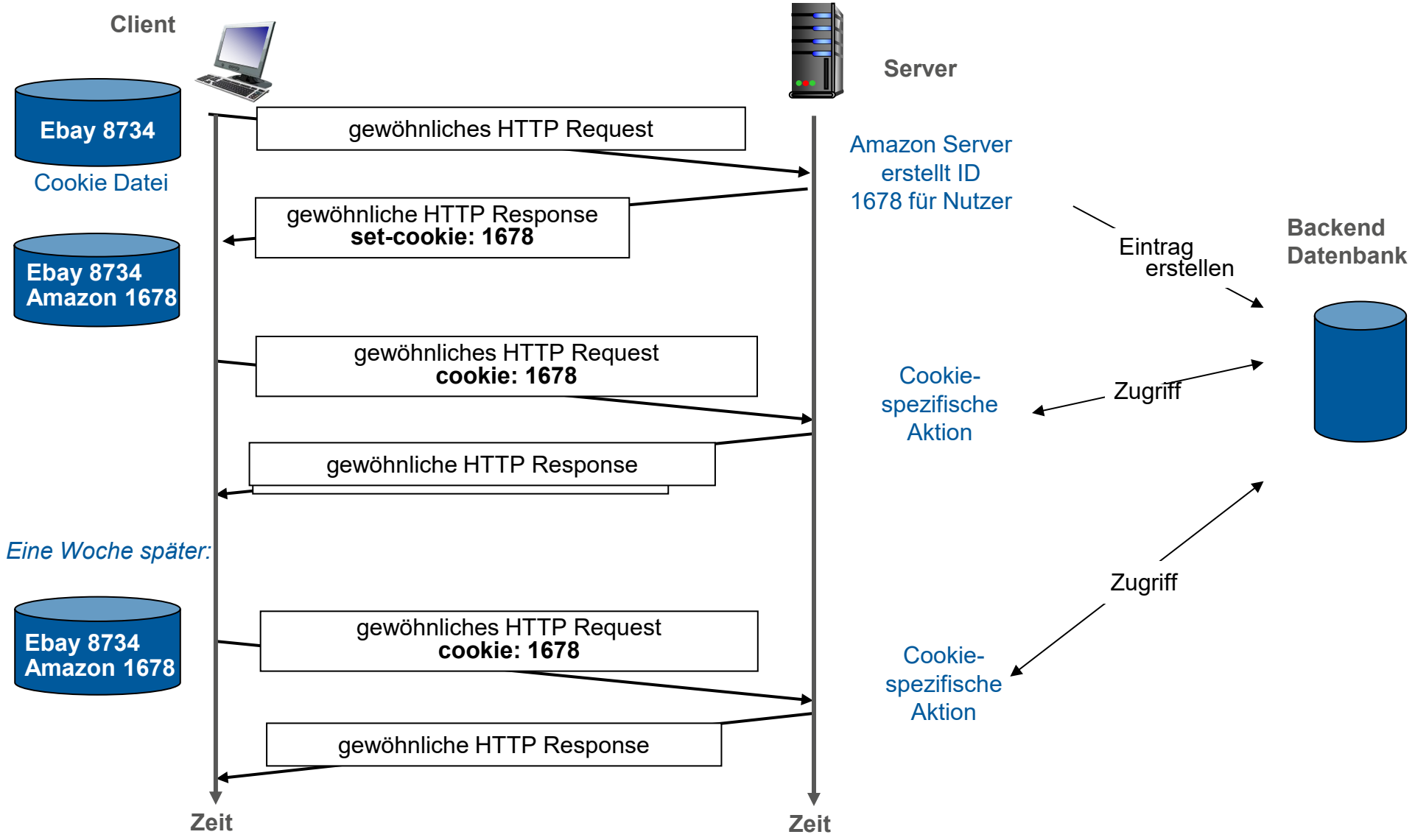
Vier Komponenten:

1) Cookie Header Zeile der HTTP *Response* Nachricht
2) Cookie Header Zeile in der nächsten HTTP *Request* Nachricht
3) Cookie Datei wird auf dem Host des Nutzers gespeichert und vom Browser verwaltet
4) Backend Datenbank der Webseite

Beispiel:

- Susanne nutzt den Browser auf ihrem Laptop und besucht eine spezifische Shop-Seite zum ersten Mal
- Wenn der initiale HTTP Request bei der Seite ankommt, erstellt die Seite:
  - eindeutige ID (aka “Cookie”)
  - Eintrag in der Backend Datenbank für die ID
- nachfolgende HTTP Requests von Susanne an diese Seite, werden den Cookie ID Wert beinhalten und damit der Seite erlauben Susanne zu “identifizieren”

Halten von Nutzer/Server Zustand: Cookies



Wofür Cookies verwendet werden können:

- Autorisierung
- Einkaufswägen
- Empfehlungen
- Zustand einer Nutzersitzung (Webmail)

Herausforderung: Wie Zustand halten?

- **in Protokoll-Endpunkten:** halten des Zustands beim Sender/Empfänger über mehrere Transaktionen
- **in Nachrichten:** Cookies in HTTP-Nachrichten tragen Zustand

Am Rande

Cookies und Privatsphäre:

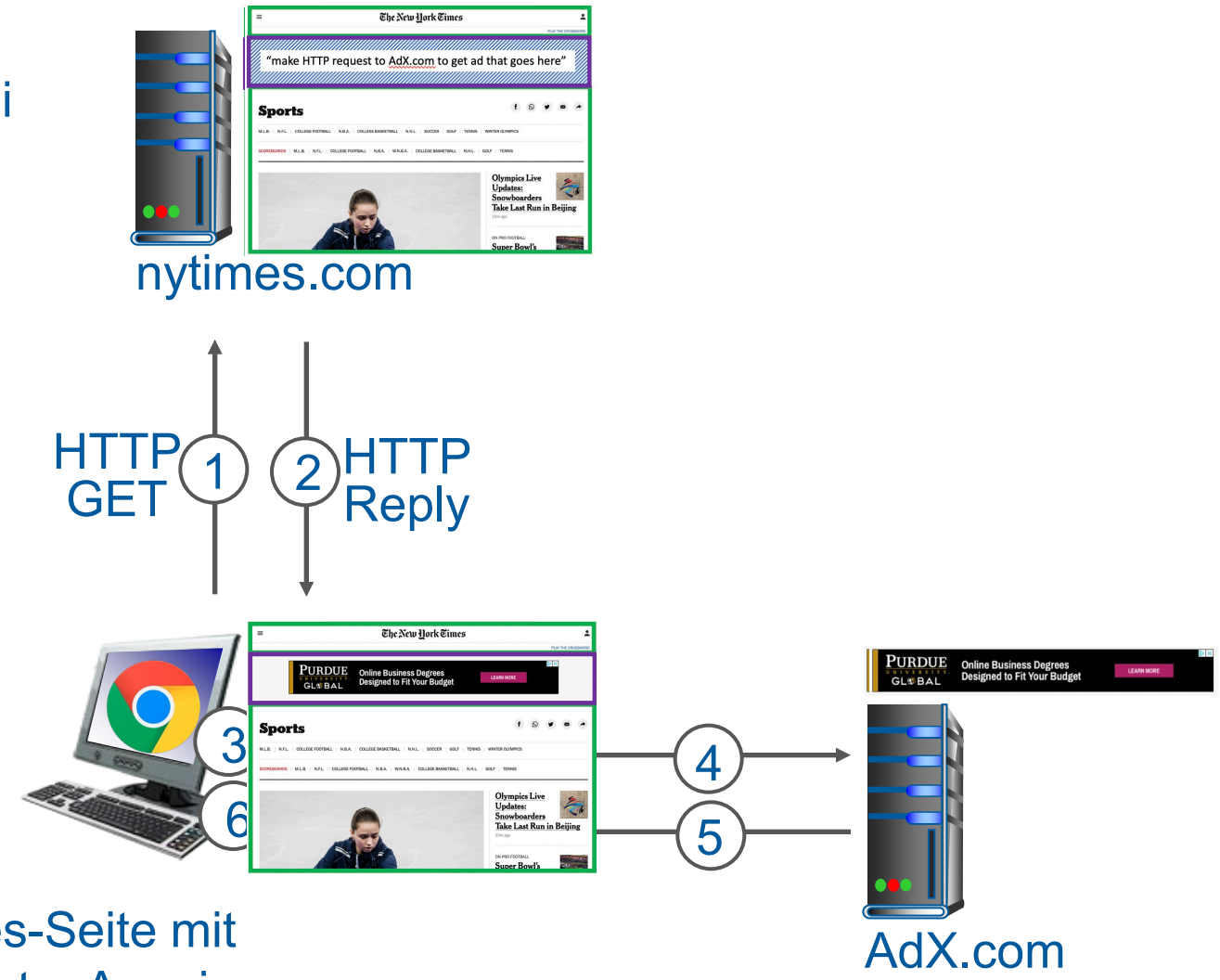
- Cookies erlauben Seiten viel über sie zu *lernen*
- Persistente Cookies dritter Parteien (Tracking Cookies) erlauben eine “Common Identity” (Cookie Wert) über mehrere Webseiten hinweg zu verfolgen

Beispiel: Darstellung der NY Times Webseite

1 2 GET Basis-HTML-Datei von nytimes.com

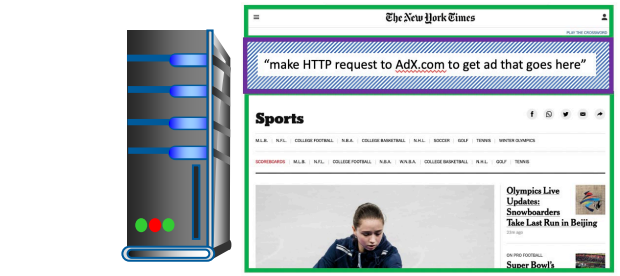
4 5 Anzeige von AdX.com abrufen

7 Zusammengesetzte Seite anzeigen



NY Times-Seite mit eingebetteter Anzeige

Cookies: Verfolgung des Surfverhaltens eines Nutzers



nytimes.com (Sport)

1634: Sport, 15.02.22

"First-Party"-Cookie – von der Website, die Sie besucht haben (stellt die Basis-HTML-Datei bereit)

HTTP GET

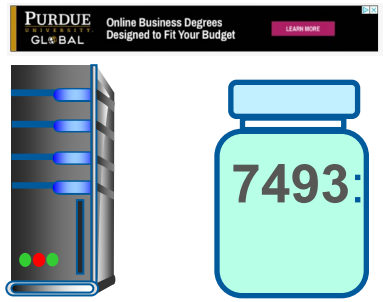
HTTP Reply
Set cookie: 1634

HTTP GET
Referrer: NY Times Sport

4

5

HTTP Reply
Set cookie: **7493**



AdX.com

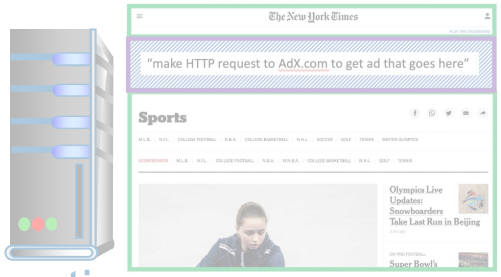
7493: NY Times Sport, 15.02.2022

NY Times: 1634
AdX: **7493**

"Drittanbieter"-Cookie – von der Website, die Sie **nicht** besucht haben

Cookies: Verfolgung des Surfverhaltens eines Nutzers

socks.com



nytimes.com

1634: Sport, 15.02.2022

AdX:

- **verfolgt mein Surfen im Internet** über Websites mit AdX-Anzeigen
- Kann gezielte Werbung basierend auf dem Browserverlauf zurückgeben

HTTP Reply 2

HTTP GET 1

HTTP GET
Referrer: socks.com, Cookie: **7493** 4

5 HTTP Reply
Set cookie: **7493**

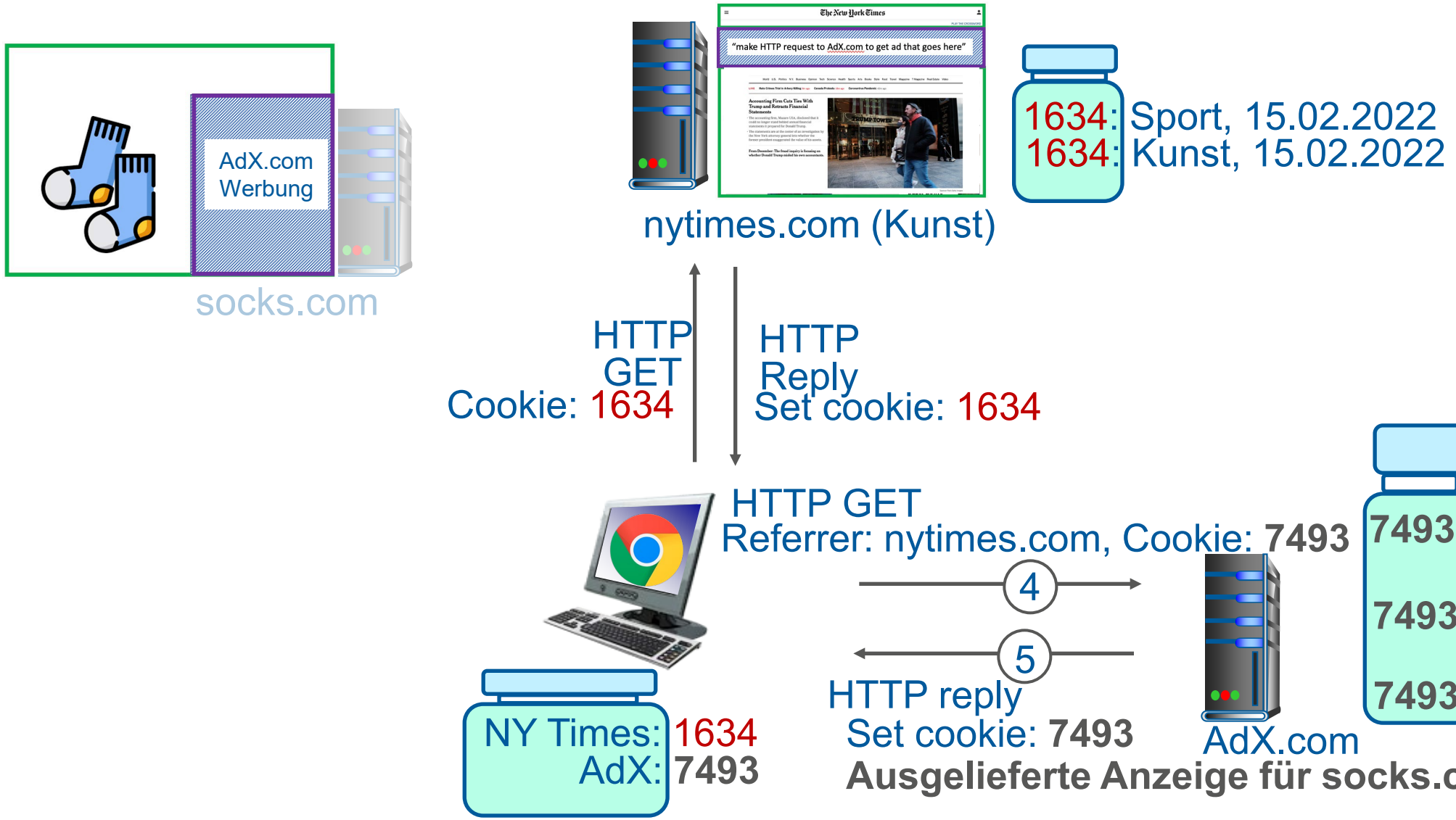
NY Times: 1634
AdX: **7493**

AdX.com

7493: NY Times Sport, 15.02.2022

7493: socks.com, 16.02.2022

Cookies: Verfolgung des Surfverhaltens eines Nutzers (Ein Tag später)



Cookies: Verfolgung des Surfverhaltens eines Nutzers

Cookies können verwendet werden, um:

- Verfolgung des Benutzerverhaltens auf einer bestimmten Website (**First-Party-Cookies**)
- Verfolgen Sie das Benutzerverhalten über mehrere Websites hinweg (**Cookies von Drittanbietern**), ohne dass der Benutzer jemals die Tracker-Website besucht (!)
- Die Sendungsverfolgung kann für den Benutzer unsichtbar sein:
  - Anstatt der angezeigten Anzeige, die HTTP GET zum Tracker auslöst, könnte es sich um einen unsichtbaren Link handeln

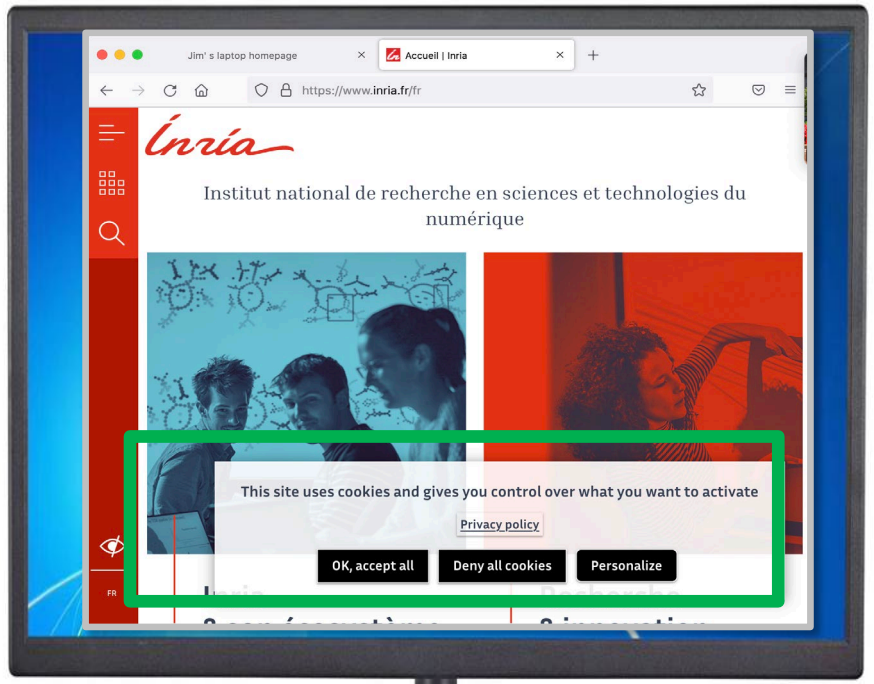
Tracking von Drittanbietern über Cookies:

- in Firefox- und Safari-Browsern standardmäßig deaktiviert
- wird im Chrome-Browser im Jahr 2023 deaktiviert

GDPR/DS-GVO und Cookies

"Natural persons may be associated with online identifiers […] such as internet protocol addresses, cookie identifiers or other identifiers […].
This may leave traces which, in particular when combined with unique identifiers and other information received by the servers, may be used to create profiles of the natural persons and identify them."

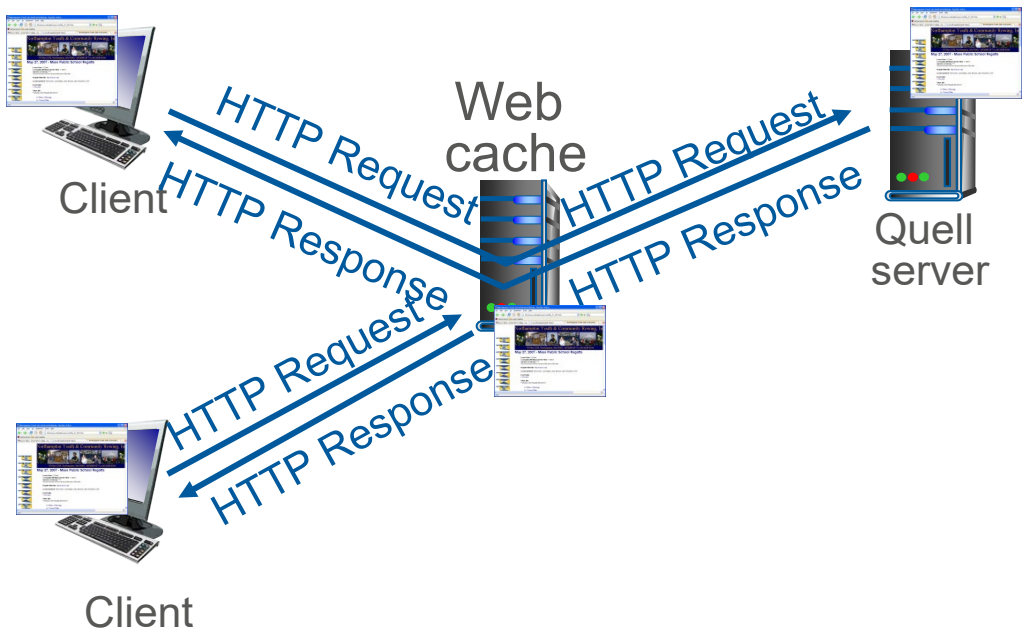
Wenn Cookies eine Person identifizieren können, gelten Cookies als personenbezogene Daten, die den DSGVO-Vorschriften für personenbezogene Daten unterliegen



Der Benutzer hat die ausdrückliche Kontrolle darüber, ob Cookies zugelassen werden oder nicht

Ziel: Erfüllen von Client Requests ohne den Quellserver zu involvieren

- Benutzer konfigurieren Browser, dass er auf einen (lokalen) **Web Cache** verweist
- Browser sendet alle HTTP Requests zum Cache
  - **falls** Objekt im Cache: Cache liefert Objekt an Client
  - **ansonsten** fragt der Cache das Objekt beim Quellserver an, speichert das empfangene Objekt und liefert es an den Client aus



Web Caches (aka Proxy Server)

- Web Cache handelt sowohl als Client als auch Server
  - Server aus Sicht des anfragenden Client
  - Client aus Sicht des Quellserver
- Server sagt Cache wie lange ein Objekt gespeichert werden darf im Response Header:

```
Cache-Control: max-age=<seconds>
```

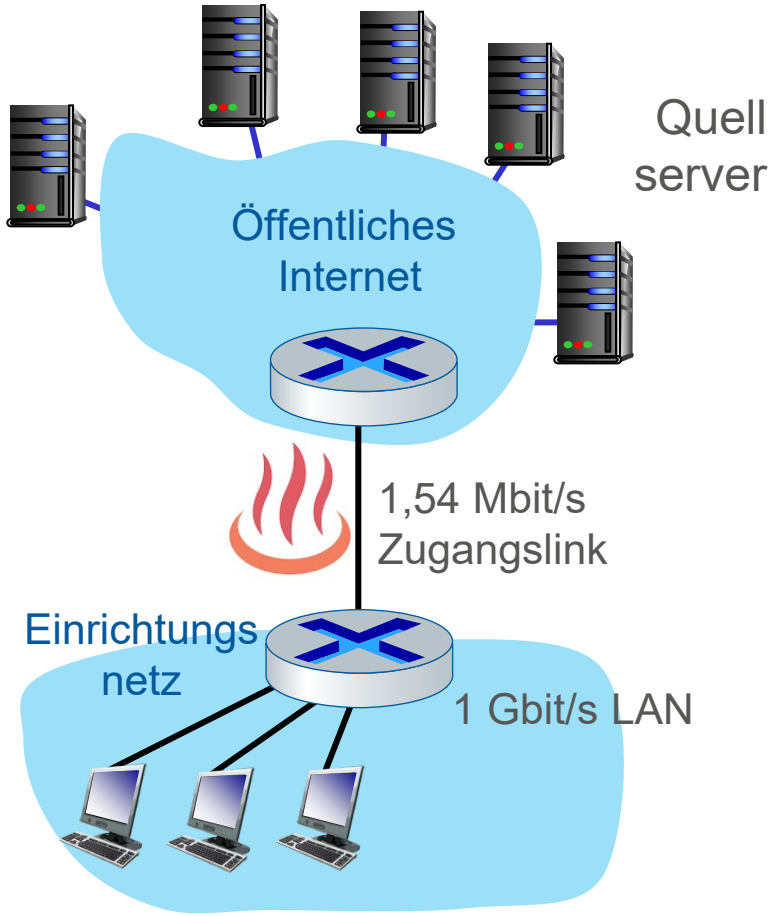
```
Cache-Control: no-cache
```

- **Warum** Web Caching?
- Reduzieren der Antwortzeit für ein Client Request
  - Cache ist näher am Client
- Reduzieren von Verkehr auf dem Zugangslink einer Einrichtung
- Internet ist voll von Caches
  - ermöglicht “armen” Content Providern ihren Inhalt effektiver auszuliefern

Caching Beispiel

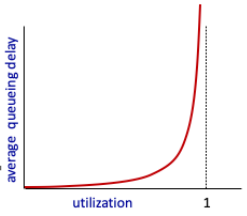
Szenario:

- Zugangslink-Rate: 1,54 Mbit/s
- RTT vom Einrichtungs-Router zum Server: 2 s
- Web-Objektgröße: 100 Kbit
- Durchschnittliche Anfragerate von Browsern an Quellserver: 15/s
  - Durchschnittliche Datenrate an Browser: 1,50 Mbit/s



Leistung:

- Zugangslinkauslastung= 0,97
- LAN-Auslastung: 0,0015
- Ende-Ende Latenz = Internet Latenz + Zugangslink-Latenz + LAN Latenz
  = 2 s + Minuten + µs



Problem: große Warteschlangen-verzögerung bei hoher Auslastung!

Option 1: Schnelleren Zugangslink kaufen

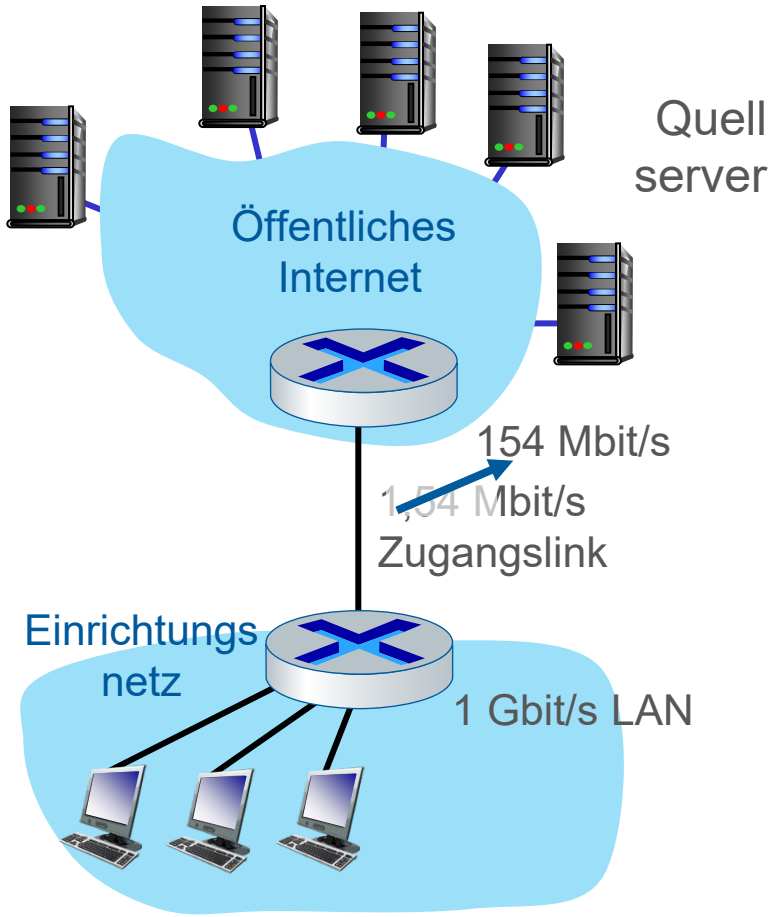
Szenario:

- Zugangslink-Rate: 1,54 Mbit/s → 154 Mbit/s
- RTT vom Einrichtungs-Router zum Server: 2 s
- Web-Objektgröße: 100 Kbit
- Durchschnittliche Anfragerate von Browsern an Quellserver: 15/s
  - Durchschnittliche Datenrate an Browser: 1,50 Mbit/s

Leistung:

- Zugangslinkauslastung= 0,97 → 0,0097
- LAN-Auslastung: 0,0015
- Ende-Ende Latenz = Internet Latenz + Zugangslink-Latenz + LAN Latenz
  = 2 s + Minuten + µs → ms

Kosten: schneller Zugangslink (teuer!)



Option 2: Web Cache installieren

Szenario:

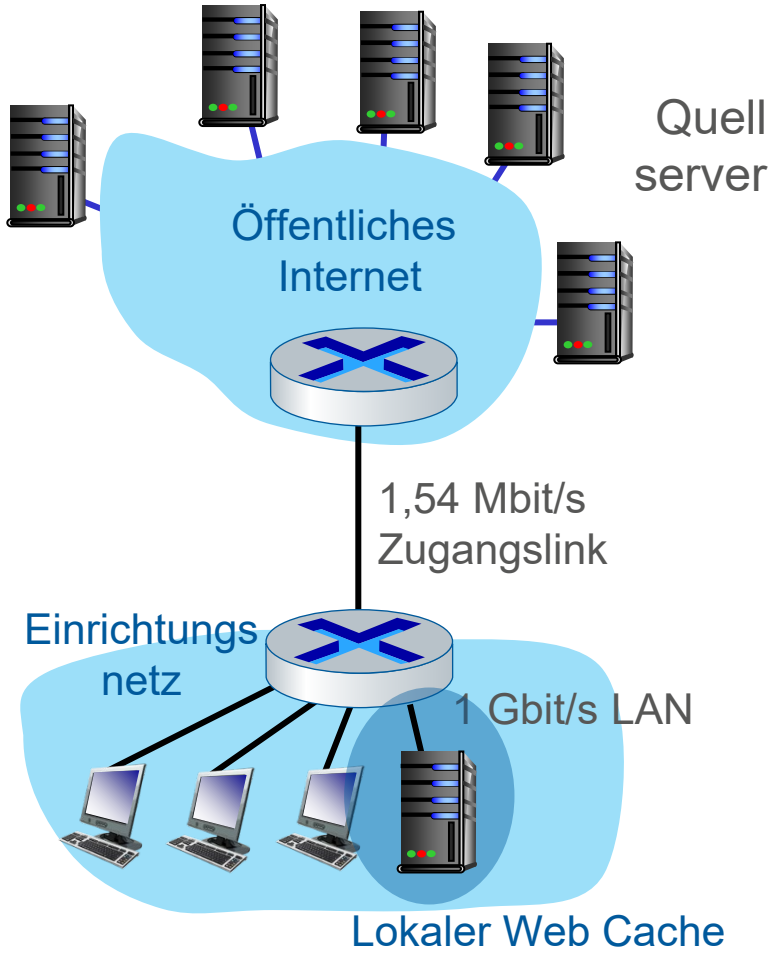
- Zugangslink-Rate: 1,54 Mbit/s
- RTT vom Einrichtungs-Router zum Server: 2 s
- Web-Objektgröße: 100 Kbit
- Durchschnittliche Anfragerate von Browsern an Quellserver: 15/s
  - Durchschnittliche Datenrate an Browser: 1,50 Mbit/s

Kosten: Web Cache (billig!)

Leistung:

- LAN-Auslastung: ?
- Zugangslinkauslastung = ?
- Durchschnittliche Ende-Ende Latenz = ?

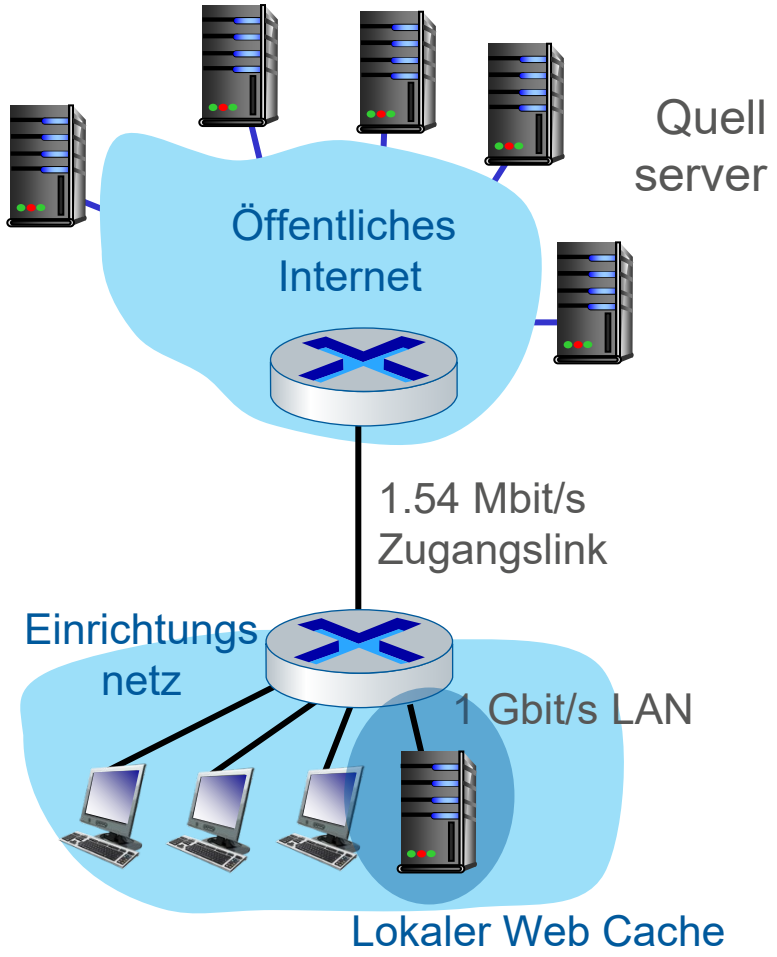
Wie wird die Linkauslastung/ Latenz berechnet?



Berechnung von Zugangslinkauslastung und Ende-Ende Latenz mit Cache:

Angenommen der Cache hat eine Trefferrate von 0,4:

- 40% der Requests vom Cache mit geringer Latenz (ms) bedient
- 60% der Requests werden von der Quelle bearbeitet
  - Rate an Browser über Zugangslink
    = 0,6 * 1,50 Mbit/s = 0,9 Mbit/s
  - Zugangslinkauslastung = 0,9/1,54 = 0,58 bedeutet geringe (ms) Warteschlangenverzögerung am Zugangslink
- Durchschnittliche Ende-Ende Latenz:
  = **0,6** * (Verzögerung von Quellservern)
  + **0,4** * (Verzögerung zum Cache)
  = 0,6 (2,01) + 0,4 (~ms) = ~ 1,2 s



Geringere durchschnittliche Ende-Ende Latenz als mit dem 154 Mbit/s Link (und billiger!)

Konditionales GET

Ziel: Objekt nicht senden, falls der Cache eine aktuelle Version gespeichert hat

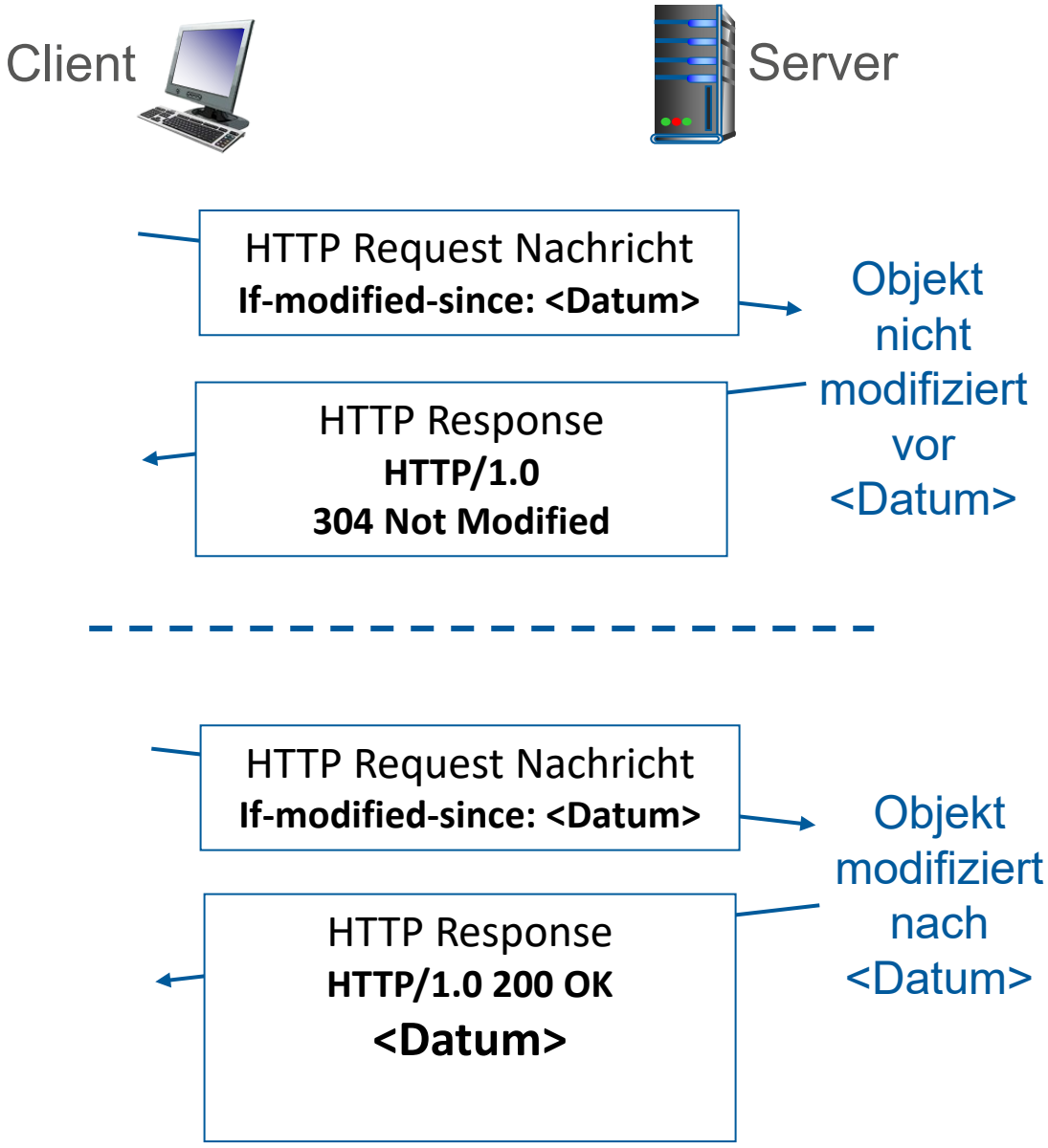
- Keine Übertragungsverzögerung (oder Nutzen von Netzressourcen)

- **Client:** spezifizieren des Datums der gespeicherten Kopie im HTTP Request

  If-modified-since: <Datum>

- **Server:** Antwort enthält kein Objekt, falls die gespeicherte Kopie aktuell ist:

  HTTP/1.0 304 Not Modified



Wichtigstes Ziel: Geringere Verzögerung für Multi-Objekt-HTTP-Requests

HTTP1.1: führte **mehrere, aufeinanderfolgende GETs** über eine einzige TCP Verbindung ein

- Server antwortet *in Reihenfolge* (FCFS: first-come-first-served) auf GET Requests
- mit FCFS, müssen kleine Objekte hinter großen auf ihre Übertragung warten (**Head-of-Line (HOL) Blocking**)
- Wiederholungsübertragung verlorener TCP Segmente blockiert die Objektübertragung

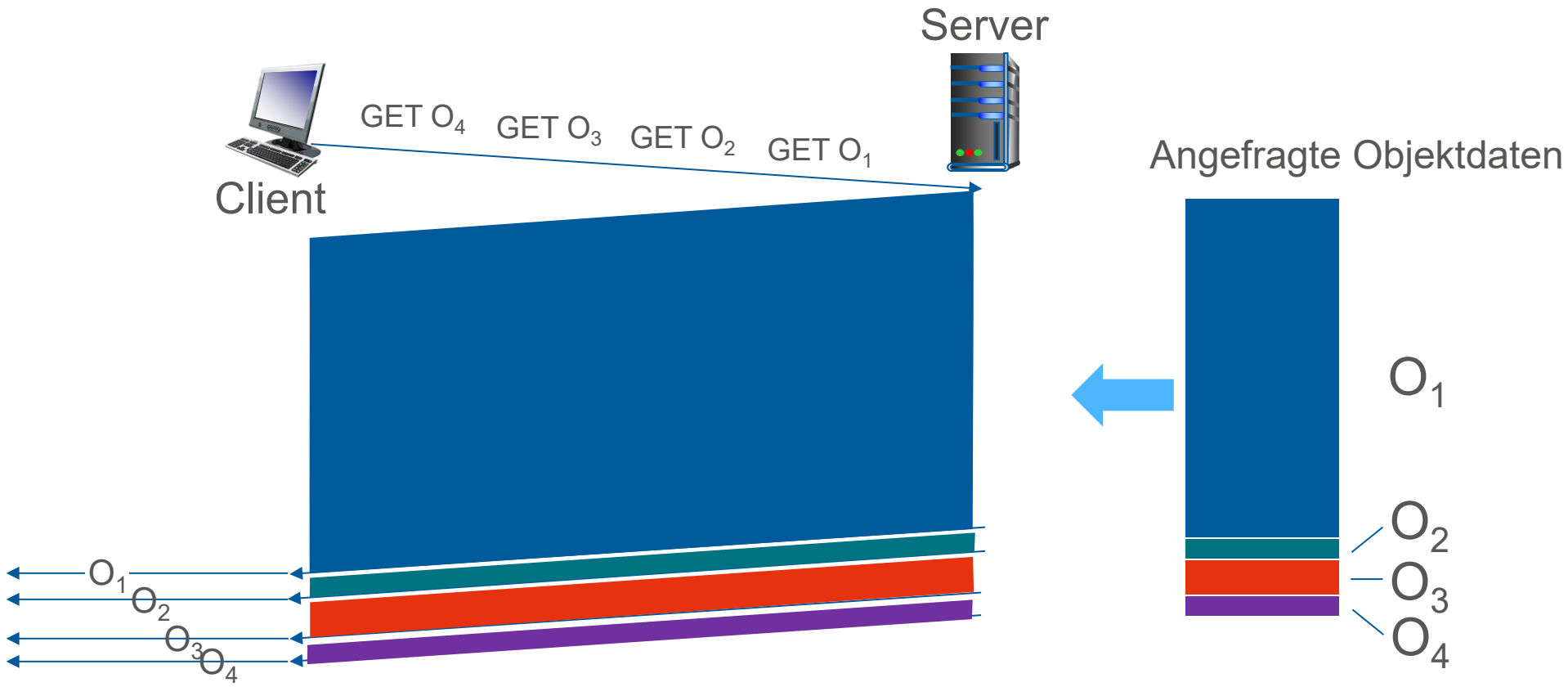
Wichtigstes Ziel: Geringere Verzögerung für Multi-Objekt-HTTP-Requests

HTTP/2: [RFC 7540, 2015] erhöhte die Flexibilität im *Server* beim Senden von Objekten zum Client:

- Methoden, Status Codes, die meisten Header Felder bleiben unverändert von HTTP 1.1
- Übertragungsreihenfolge angefragter Objekte basiert auf einer vom Client spezifizierten Objektpriorisierung (nicht unbedingt FCFS)
- Übertragen (push) von nicht angefragten Objekten zum Client

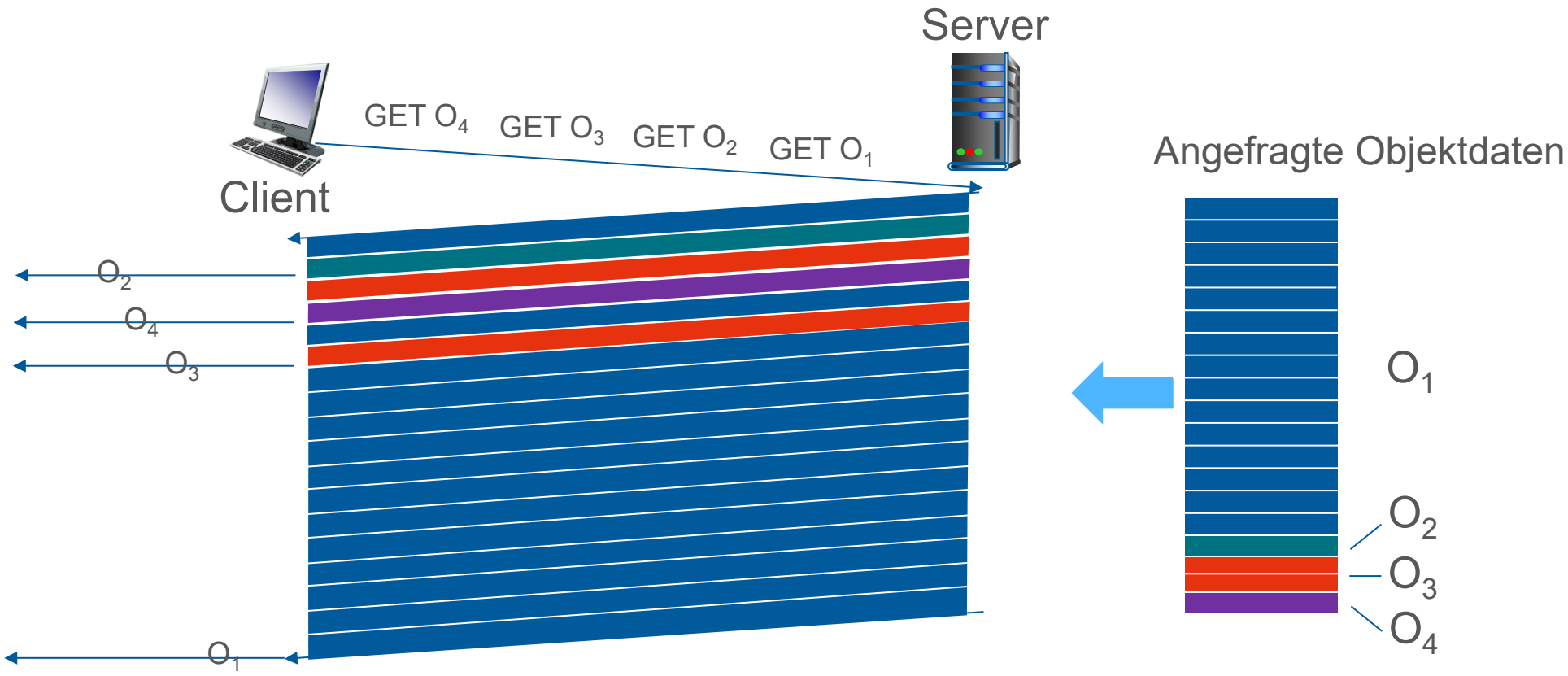
HTTP/2: Vermeiden von HOL Blocking

HTTP 1.1: Client fragt 1 großes Objekt (z.B. Videodetail) und 3 kleinere Objekte an



Objekte werden in angefragter Reihenfolge übertragen: O_2, O_3, O_4 warten hinter O_1

HTTP/2: Objekte in Rahmen unterteilt, abwechselnde Rahmenübertragung



O_2, O_3, O_4 schnell geliefert, O_1 leicht verzögert

HTTP/2 über eine einzige TCP-Verbindung bedeutet:

- Paketverlust führt weiterhin zur Verzögerung aller Objektübertragungen
  - wie schon in HTTP 1.1, haben Browser einen Anreiz mehrere parallele TCP-Verbindungen zu öffnen, um Verzögerungen zu reduzieren und den Durchsatz zu erhöhen
- keine Sicherheit über einer gewöhnlichen TCP-Verbindung
- **HTTP/3:** fügt Sicherheit & pro Objekt Fehler- und Überlastkontrolle hinzu; läuft über UDP

Applikationsschicht: Übersicht

- Prinzipien vernetzter Anwendungen
- Web und HTTP
- **Das Domain Name System DNS**
- P2P Applikationen
- Video Streaming und Content Distribution Networks
- Socket Programmierung mit UDP und TCP

DNS: Domain Name System

Menschen:

- Steuer ID, Name, Passnummer

Internet Hosts, Router:

- IP-Adresse (32 Bit)
- “Name”, z.B. thi.de – verwendet von Menschen

Frage: Wie übersetzt man zwischen IP-Adressen und Namen?

Domain Name System (DNS):

- **Verteilte Datenbank,** implementiert als eine Hierarchie von vielen *Name Servern*
- **Applikationsschichtprotokoll:** Hosts & DNS Server kommunizieren, um Namen **aufzulösen**
  - *Hinweis:* **Internet Kernfunktion, implementiert als Applikationsschichtprotokoll**
  - Komplexität am Netzrand (“Edge”)

DNS Dienste:

- Hostname-zu-IP-Adresse Übersetzung
- Host Aliasing
  - kanonische, alternative Namen
- Mail Server Aliasing
- Lastverteilung
  - Webserver Replikas: viele IP Adressen entsprechen einem Namen

Frage: Warum DNS nicht zentralisieren?

- Single Point of Failure
- Verkehrsaufkommen
- entfernte, zentrale Datenbank
- Wartung

Antwort: Skaliert nicht!

- Google DNS Server alleine: 1,2 Billionen DNS Anfragen/Tag
- Akamai DNS Server alleine: 2,2 Billionen DNS Anfragen/Tag

Riesige, verteilte Datenbank

- ~ Milliarden einfache Einträge

Bearbeitet viele *Billionen* Anfragen/Tag:

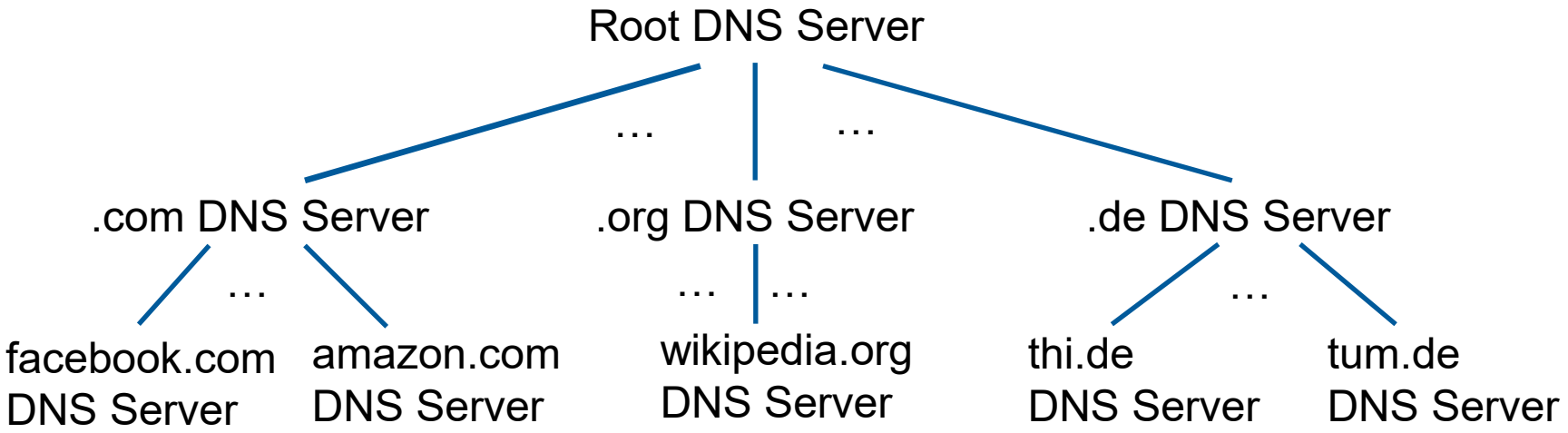
- *viel* mehr Lese- als Schreibzugriffe
- *Leistung zählt: fast jede Internet Transaktion interagiert mit DNS* – Millisekunden zählen!

Organisatorisch, physisch dezentralisiert

- Millionen verschiedener Organisationen sind für ihre Einträge verantwortlich

Ausfallsicherheit: Zuverlässigkeit, Sicherheit

DNS: Eine verteilte, hierarchische Datenbank



Root

Top Level Domain (TLD)

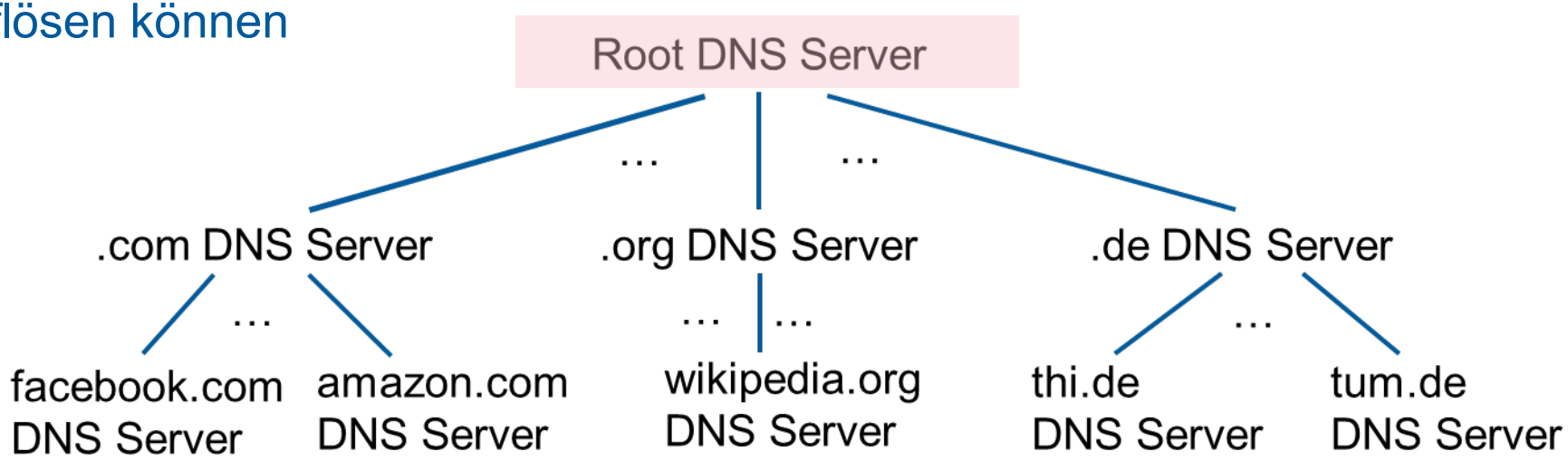
Authoritative

Client sucht IP Adresse für www.amazon.com:

- Client fragt Root Server um .com DNS Server zu finden
- Client fragt .com DNS Server um amazon.com DNS Server zu finden
- Client fragt amazon.com DNS Server um die IP Adresse für www.amazon.com zu erhalten

DNS: Root Nameserver

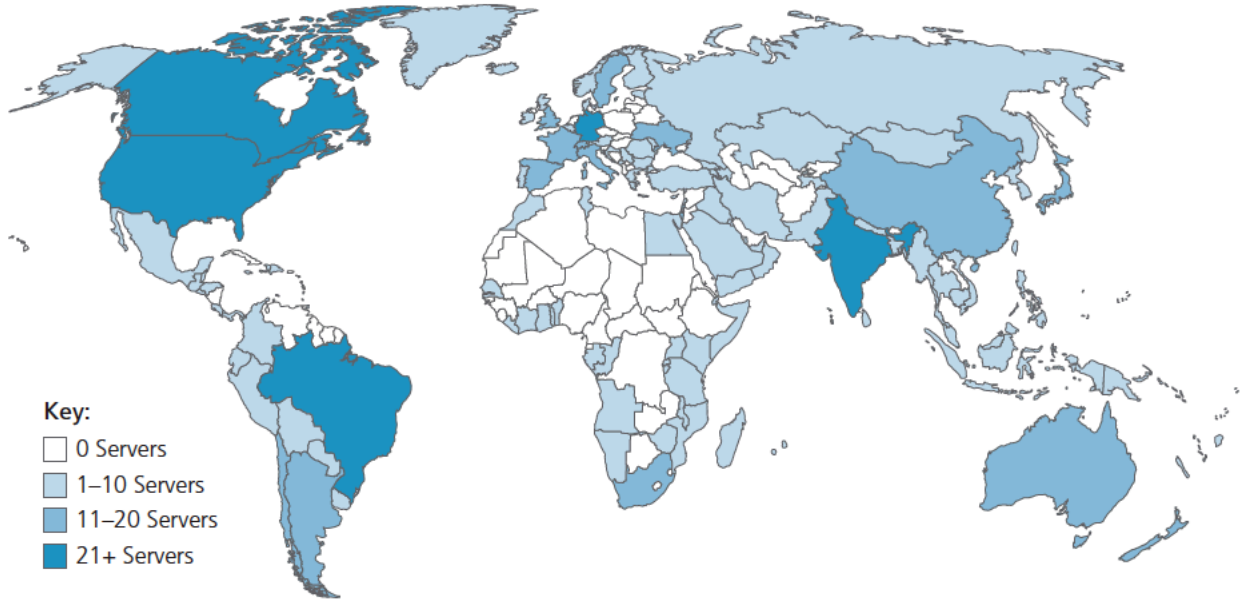
- offizieller, letzter Ausweg für Server die einen Namen nicht auflösen können



DNS: Root Nameserver

- offizieller, letzter Ausweg für Server die einen Namen nicht auflösen können
- **Unglaublich wichtige** Internet Funktion
  - Das Internet würde ohne sie nicht funktionieren!
  - DNSSEC – stellt Sicherheit bereit (Authentifikation, Nachrichten Integrität)
- ICANN (Internet Corporation for Assigned Names and Numbers) verwaltet die Root DNS Domäne

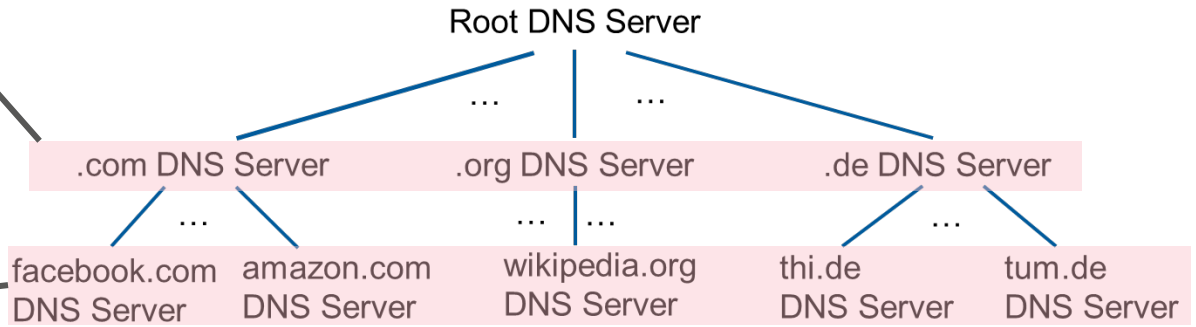
13 logische Root Name “Server” weltweit
jeder “Server” ist oft repliziert
(~200 Servers in den USA)



Top-Level Domain und autorisierende Server

Top-Level Domain (TLD) Server:

- Verantwortlich für .com, .org, .net, .edu, .aero, .jobs, .museums und alle Top-Level Länderdomänen, z.B.: .cn, .de, .fr, .ca, .jp



Autorisierende DNS Server

- Der eigene DNS Server einer Organisation, stellt autorisierende Hostnamen-zu-IP Adress Übersetzungen für die mit Namen versehenden Hosts einer Organisation bereit
- Kann von der Organisation oder einem Dienstleister betrieben werden

Lokale DNS-Nameserver

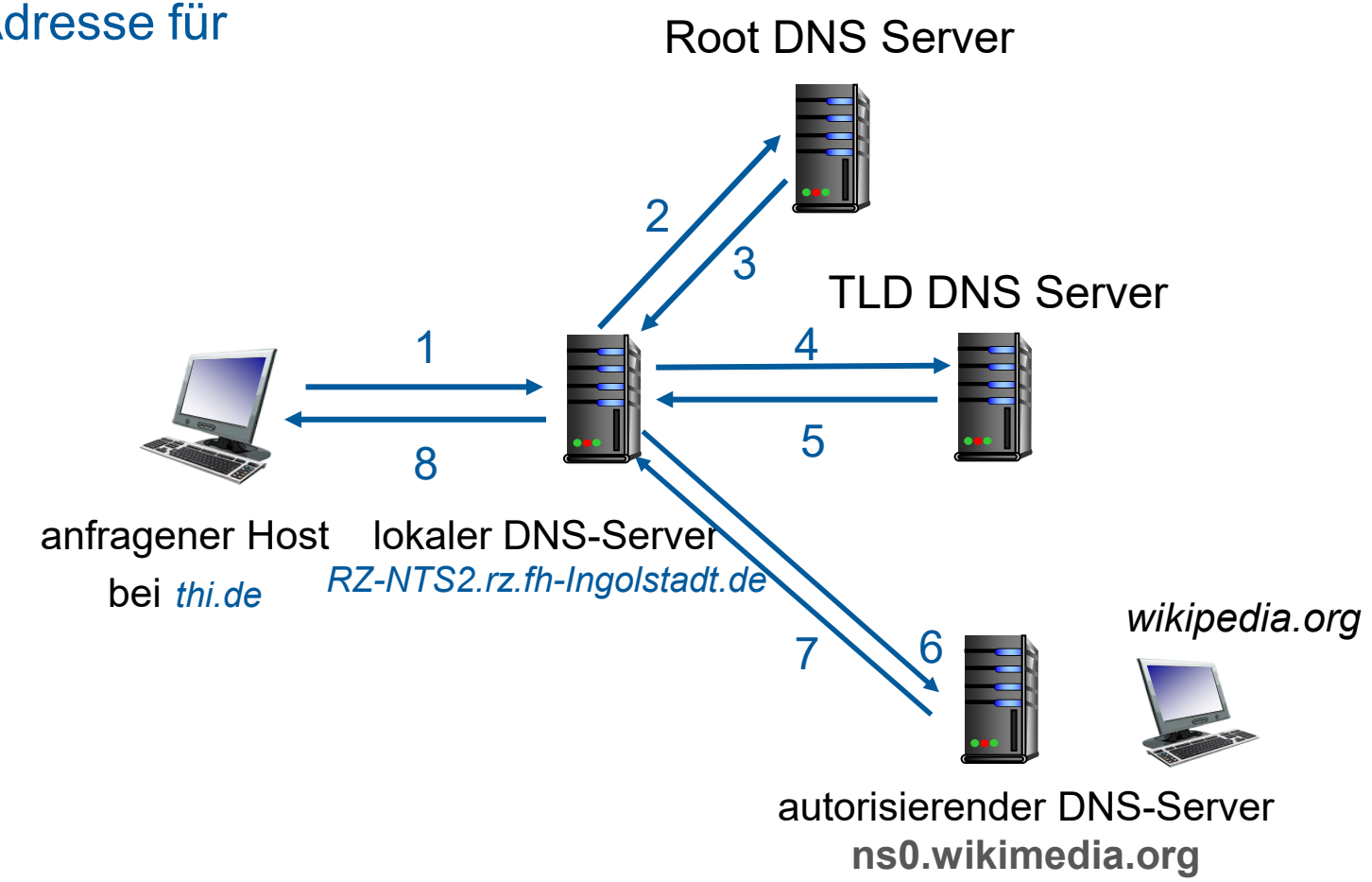
- wenn ein Host eine DNS-Anfrage stellt, sendet er sie an seinen *lokalen* DNS-Server
  - Lokaler DNS-Server antwortet:
    - aus seinem lokalen Cache von kürzlich übersetzten Namen/Adress-Paaren (möglicherweise veraltet!)
    - durch Weiterleiten der Anfrage in die DNS-Hierarchie zur Auflösung
  - jeder Anbieter hat seinen lokalen DNS Name Server; um ihren zu finden:
    - MacOS: `% scutil –dns`
    - Windows: `>ipconfig /all`
- lokale DNS-Server gehören streng genommen nicht zur Hierarchie

DNS-Namensauflösung: Iterierte Anfrage

Beispiel: Host aus thi.de sucht IP-Adresse für wikipedia.org

Iterierte Anfrage:

- Kontaktierter Server antwortet mit dem Namen des anzufragenden Servers
- “Ich kenne den Namen nicht, aber frage diesen Server”

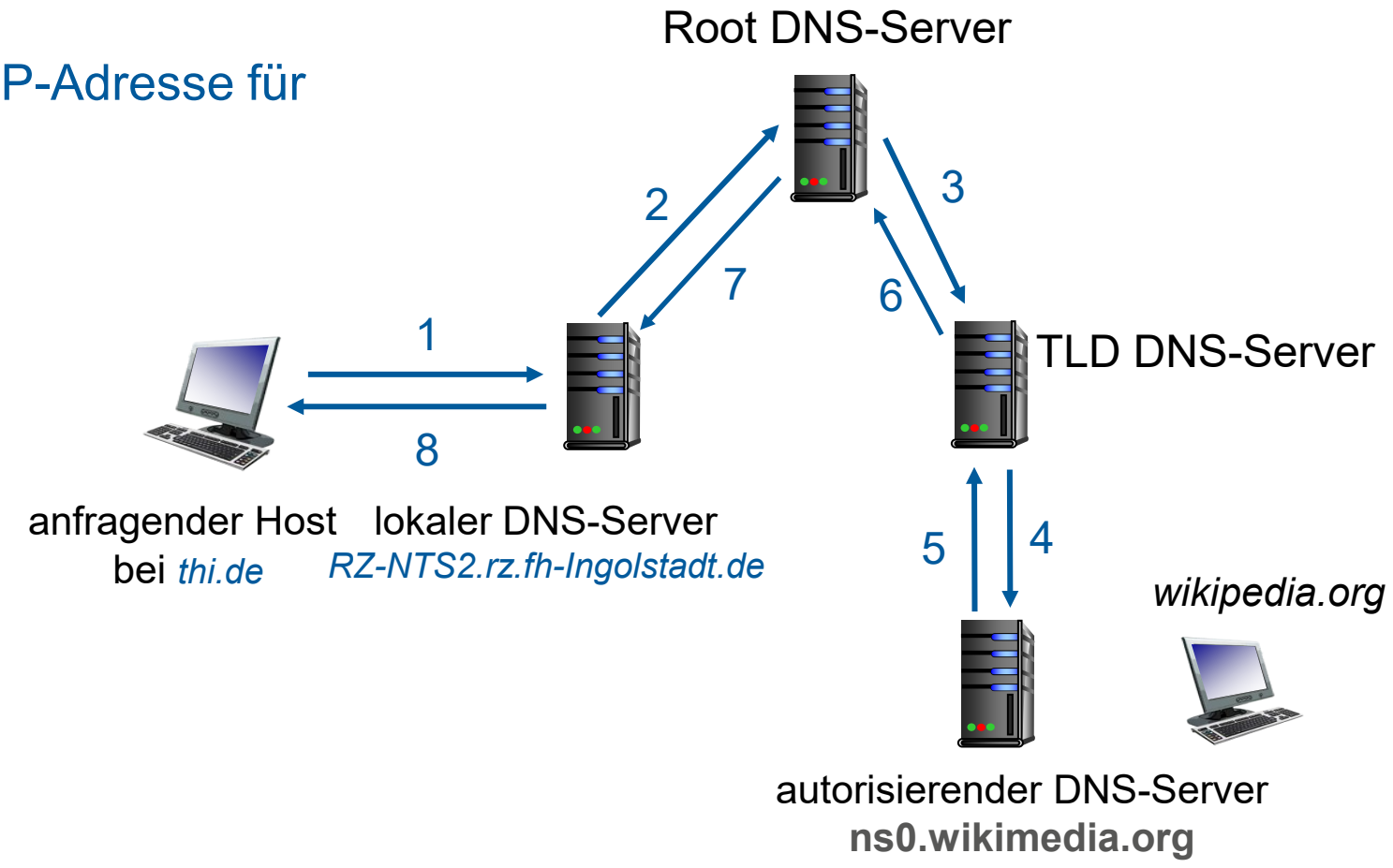


DNS-Namensauflösung: Rekursive Anfrage

Beispiel: Host aus thi.de sucht IP-Adresse für wikipedia.org

Rekursive Anfrage:

- Bürdet dem kontaktierten Nameserver die Aufgabe der Namensauflösung auf
- Hohe Last in den höheren Ebenen der Hierarchie?



DNS Informationscache

- sobald ein Nameserver eine Namen-Adress Assoziation lernt, **speichert** er die Assoziation und gibt **sofort** den gespeicherten Eintrag als Antwort auf eine Anfrage zurück
  - Speichern im Cache verbessert die Antwortzeit
  - Cache Einträge verschwinden nach einiger Zeit (TTL)
  - TLD Server sind typisscherweise in lokalen Nameservern gecached
- Gespeicherte Einträge können **veraltet sein**
  - Wenn ein Host seine IP ändert, ist das vllt. erst Internet-weit bekannt, wenn eine TTLs abgelaufen sind
  - **Best-Effort Name-zu-Adressübersetzung!**

DNS: verteilte Datenbank speichert Einträge **(R**esource **R**ecords**)**

RR Format: (`name, value, type, ttl`)

Type=A

- Name ist Hostname
- Wert ist eine IP Adresse

Type=NS

- Name ist eine Domäne (z.B. thi.de)
- Wert ist der Hostname des autorisierenden Nameserver für diese Domäne

Type=CNAME

- Name ist Alias Name für einen “kanonischen” (echten) Namen
- www.ibm.com ist in Wirklichkeit servereast.backup2.ibm.com
- Wert ist der kanonische Name

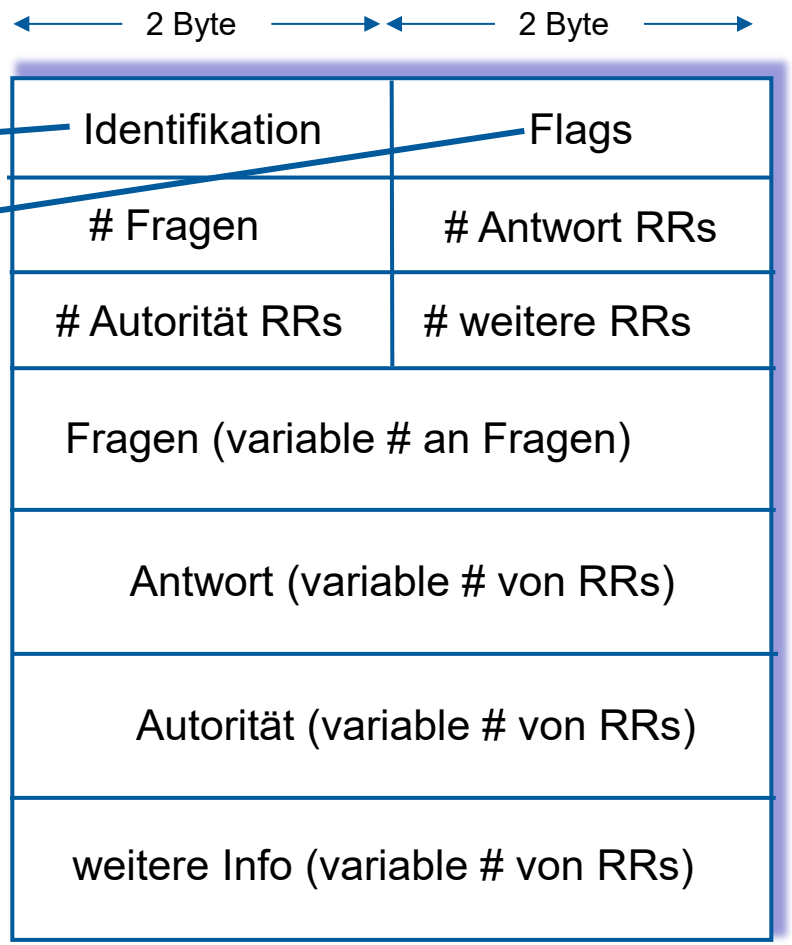
Type=MX

- `value` ist der Name des SMTP Mailservers assoziiert mit `name`

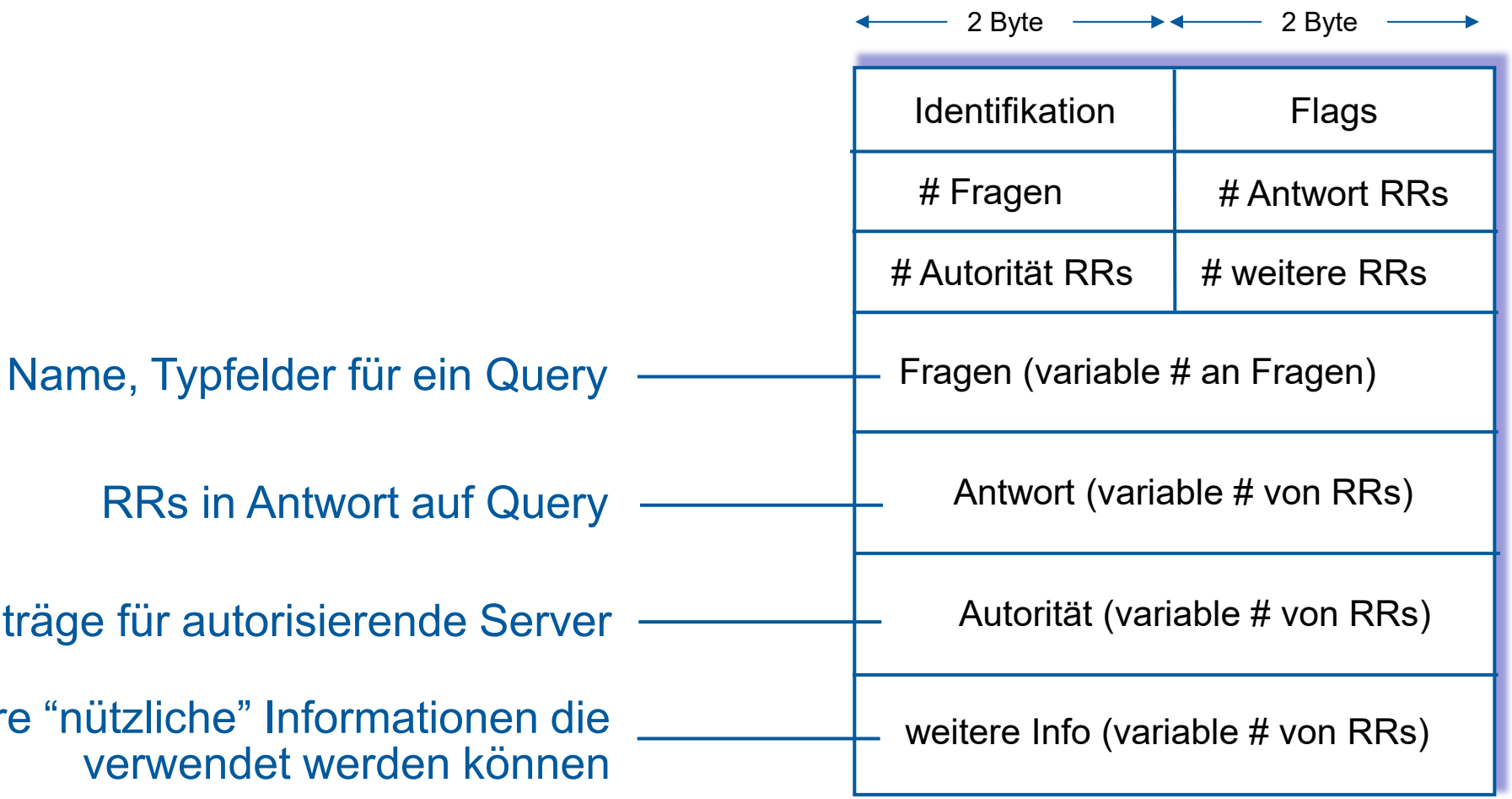
DNS **Query** und **Reply** Nachrichten, haben beide das selbe **Format:**

Nachrichtenheader

- **Identifikation:** 16 Bit # für Query, Reply nutzt dieselbe #
- **Flags:**
  - Query oder Reply
  - Rekursion gewünscht
  - Rekursion verfügbar
  - Antwort ist autorisierend



DNS **Query** und **Reply** Nachrichten, haben beide das selbe **Format:**



DNS Eintrag erstellen

- Beispiel: neues Startup “Schanzer Solutions”
- Name schanzer-solutions.de bei **DNS Registrar** (z.B. DENIC) anmelden
  - Zur Verfügung stellen von Namen, IP Adressen von autoritativen Nameservern (primär und sekundär)
  - Registrar fügt NS, A RRs in den .de TLD Server ein:
  - (schanzer-solutions.de, dns1.schanzer-solutions.de, NS)
  - (dns1.schanzer-solutions.de, 212.212.212.1, A)
- Erstellen eines lokalen autoritativen Servers mit IP Adresse 212.212.212.1
  - Typ A Eintrag für www. schanzer-solutions.de
  - Typ MX Eintrag für schanzer-solutions.de

DDoS Angriffe

- Bombardieren der Root-Server mit Verkehr
  - Bisher nicht erfolgreich
  - Verkehrs-Filter
  - lokale DNS Server cachen IPs von TLD Servern, ermöglichen Umgehen der Root Server
- Bombardieren der TLD Server
  - Möglicherweise gefährlicher

Spoofing Angriffe

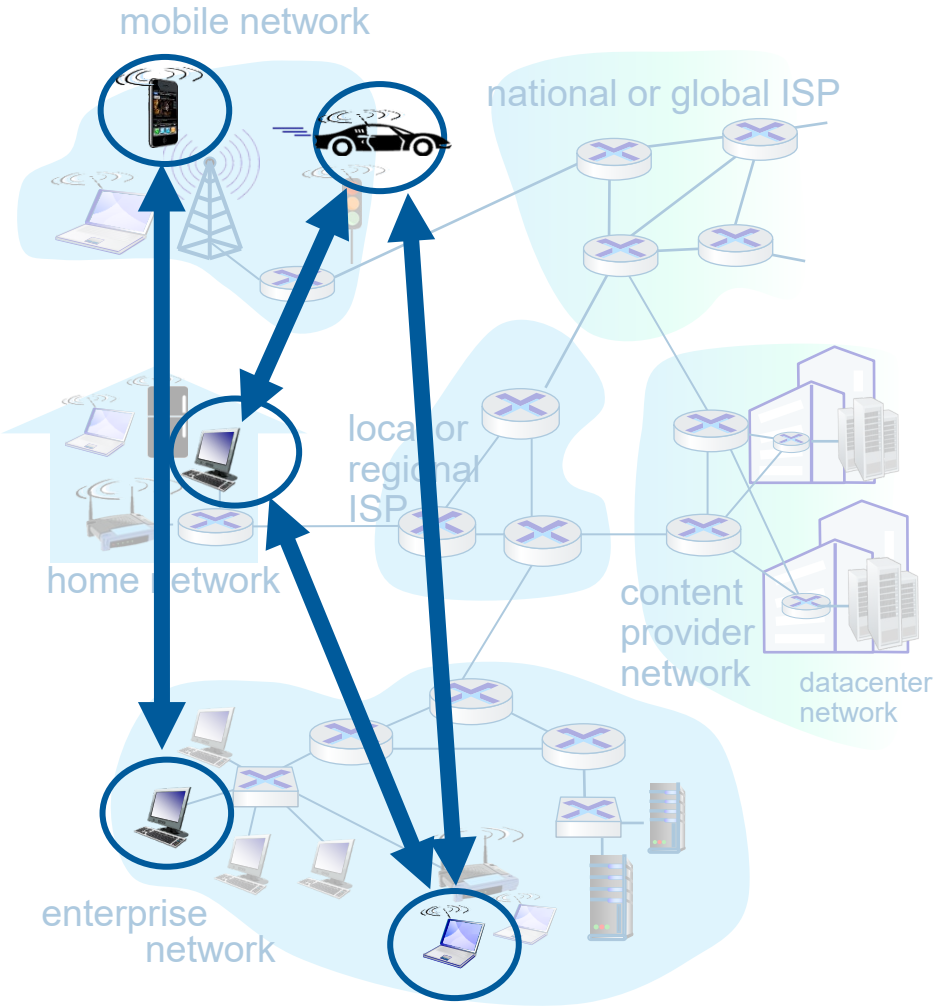
- Abfangen von DNS Queries, antworten mit falschen Replies
  - DNS Cache Poisoning
  - RFC 4033: DNSSEC Authentifikationsdienste

Applikationsschicht: Übersicht

- Prinzipien vernetzter Anwendungen
- Web und HTTP
- Das Domain Name System DNS
- **P2P Applikationen**
- Video Streaming und Content Distribution Networks
- Socket Programmierung mit UDP und TCP

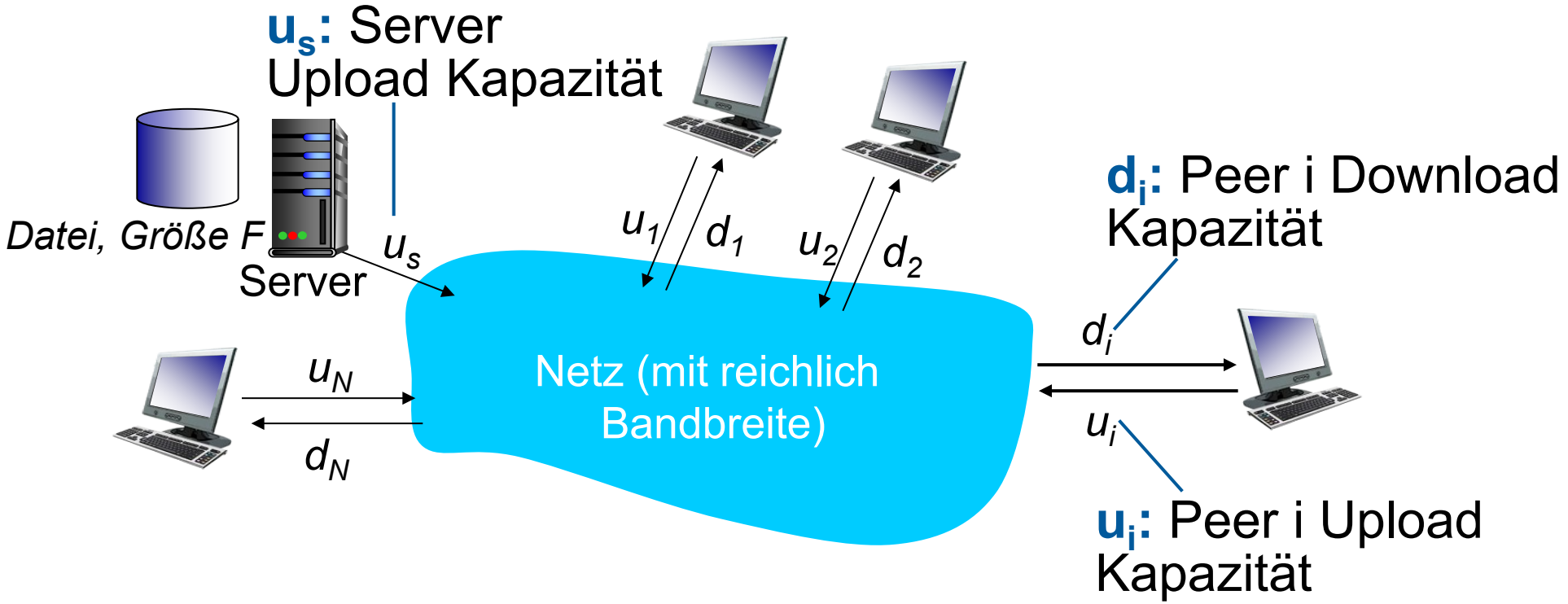
Peer 2 Peer Architektur

- Kein immer aktiver Server
- Beliebige Endsystem kommunizieren direkt
- Peer fragen Dienst bei anderen Peers an und bieten als Gegenleistung ebenfalls einen Dienst an
  - ***Skaliert von selbst* – neue Peers bringen sowohl neue Dienstkapazität als auch höhere Nachfrage**
- Peers sind mit Unterbrechungen verbunden und können IP Adressen ändern
  - komplexes Management
- Beispiel: P2P Dateiaustausch (BitTorrent), Skype (früher)



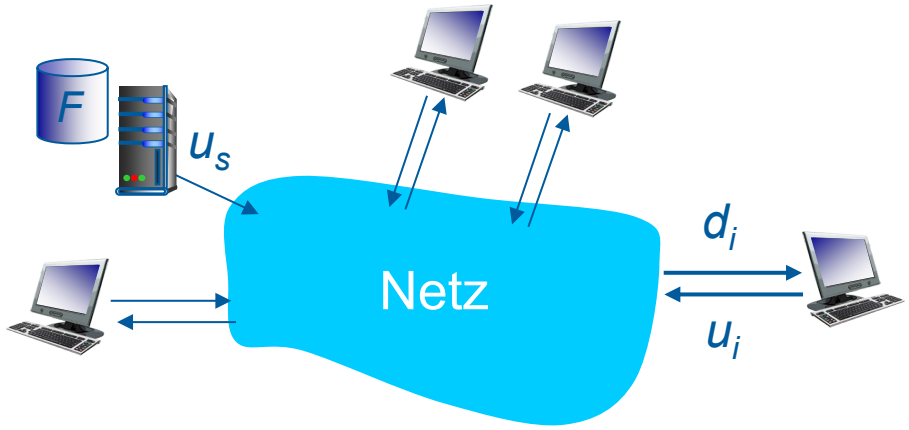
Frage: Wie lange dauert es eine Datei (Größe F) von einem Server zu N Peers zu verteilen?

- Peer Upload/Download Kapazität ist die beschränkte Ressource



Dauer der Datei-Verteilung: Client-Server

- **Übertragung vom Server:** muss *N* Kopien einer Datei hochladen:
  - Sendedauer einer Kopie: F/u_s
  - Sendedauer von N Kopien: NF/u_s

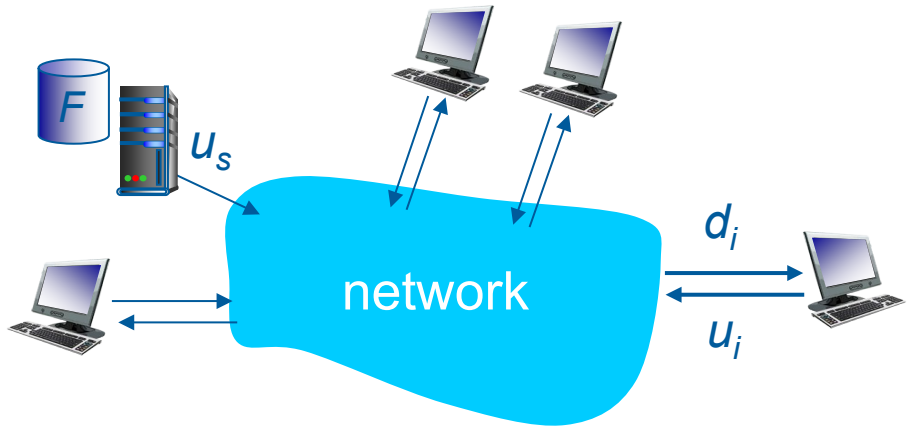


- **Client:** Jeder Client muss eine Kopie der Datei herunterladen
  - d_{min} = min. Client Download Rate
  - Min. Client Download Dauer: F/d_{min}

Zeitdauer zum Verteilen von F an N Clients mit dem Client-Server Ansatz	$D_{c\text{-}s} \geq max\{NF/u_s, F/d_{min}\}$

Dauer der Datei-Verteilung: P2P

- **Übertragung vom Server :** muss mindestens eine Kopie hochladen
  - Sendedauer einer Kopie: F/u_s
- **Client:** jeder Client muss eine Kopie herunterladen
  - Min. Client Download Dauer: F/d_{min}
- **Clients:** müssen zusammen NF Bits herunterladen
  - Max. Upload Rate (beschränkt die max. Download Rate) ist $u_s + Su_i$

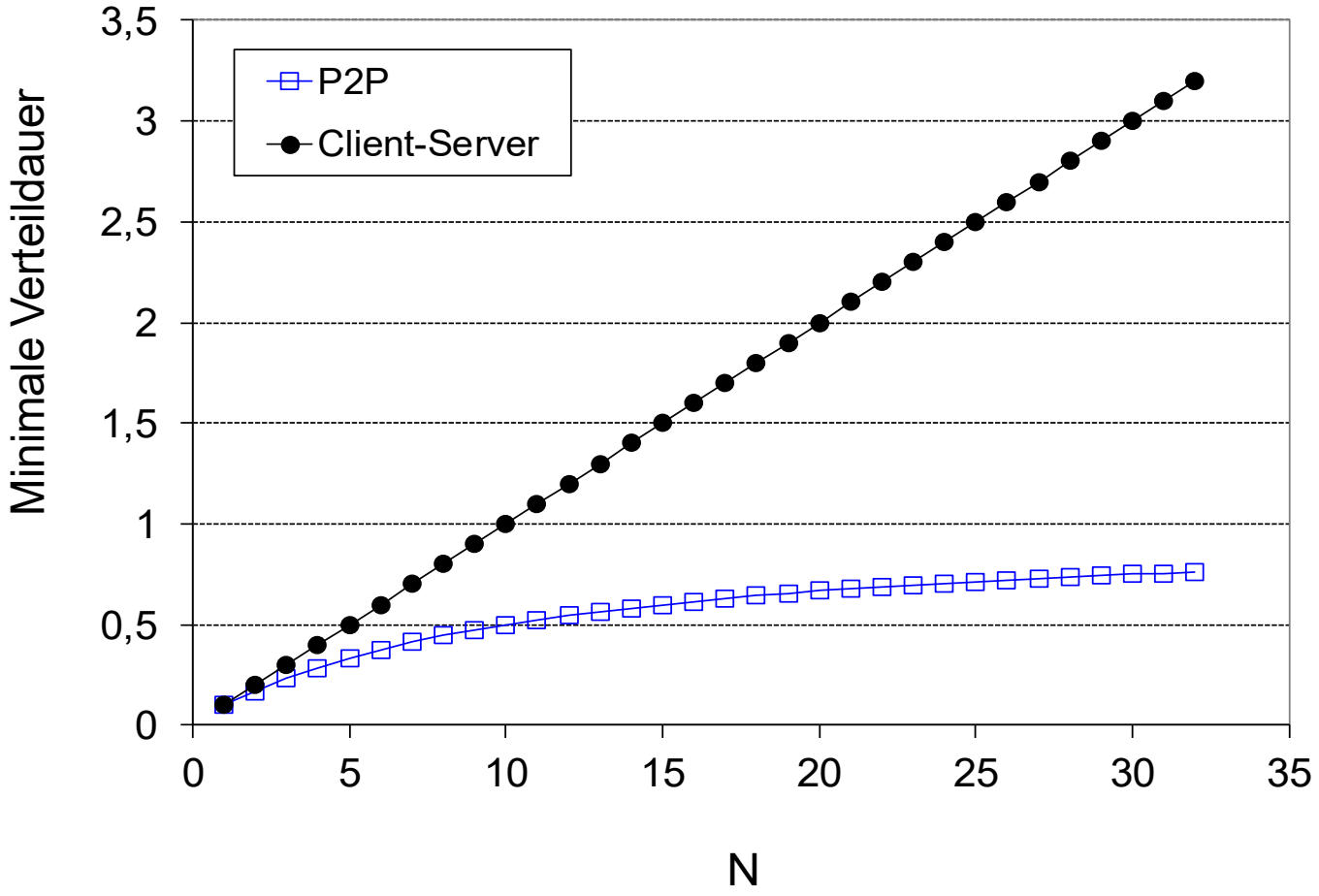


Zeitdauer zum Verteilen von F an N Clients mit dem P2P Ansatz

$$D_{P2P} \geq max\{F/u_s, F/d_{min}, NF/(u_s + \Sigma u_i)\}$$

Client-Server vs. P2P: Beispiel

Client Upload Rate = u, F/u = 1 Stunde, $u_s = 10u$, $d_{min} \geq u_s$

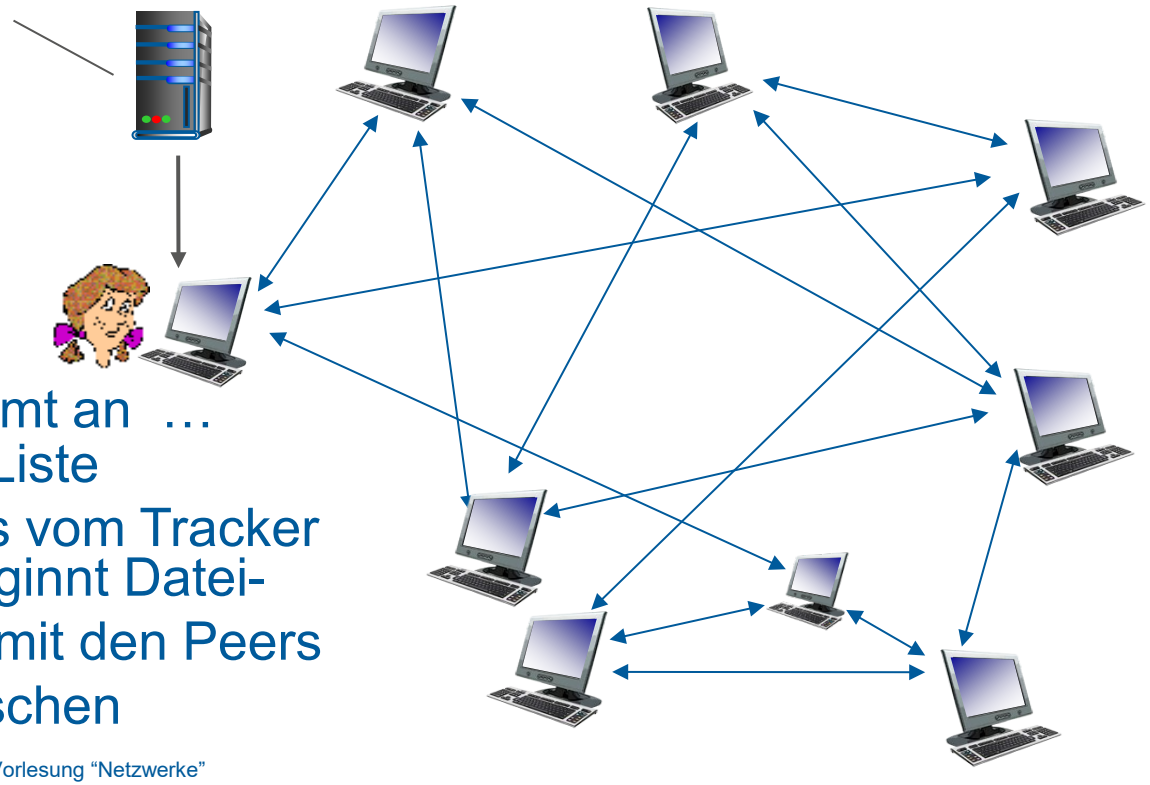


P2P Dateiverteilung: BitTorrent

- Datei wird in 256Kb “Chunks” unterteilt
- Peers senden/empfangen Datei-”Chunks”

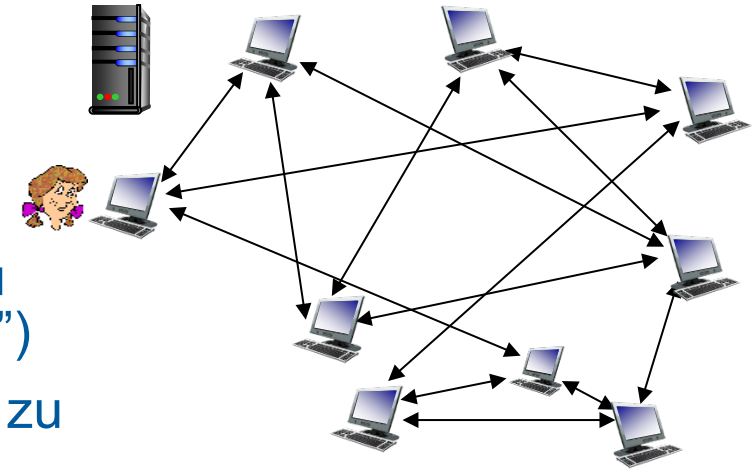
Tracker: erfasst teilnehmende Peers

Torrent: Gruppe von Peers, die “Chunks” einer Datei austauschen



Alice kommt an …
… erhält Liste von Peers vom Tracker
… und beginnt Datei-”Chunks” mit den Peers auszutauschen

P2P Dateiverteilung: BitTorrent



- Ankommender Peer:
  - hat keine Chunks, aber wird diese mit der Zeit von anderen Peers ansammeln
  - Meldet sich am Tracker an, um die Liste von Peers zu erhalten, verbindet sich mit einigen Peers (“Nachbarn”)
- Während des Downloads lädt der Peer gleichzeit Chunks zu anderen Peers hoch
- Peer kann die Peers wechseln mit denen Chunks ausgetauscht werden
- **Churn (Fluktuation):** Peers können kommen und gehen
- Sobald der Peer die gesamte Datei besitzt, kann er (egoistisch) gehen oder (uneigennützig) bleiben und Datei weiter teilen

Anfragen von Chunks:

- Zu jeder Zeit besitzen verschiedene Peers, verschiede Chunks der Datei
- Alice fragt jeden Peer periodisch nach der Liste von Chunks, die diese besitzen
- Alice fragt fehlende Chunks von Peers an – die seltensten zuerst

Senden von Chunks: Tit-for-Tat

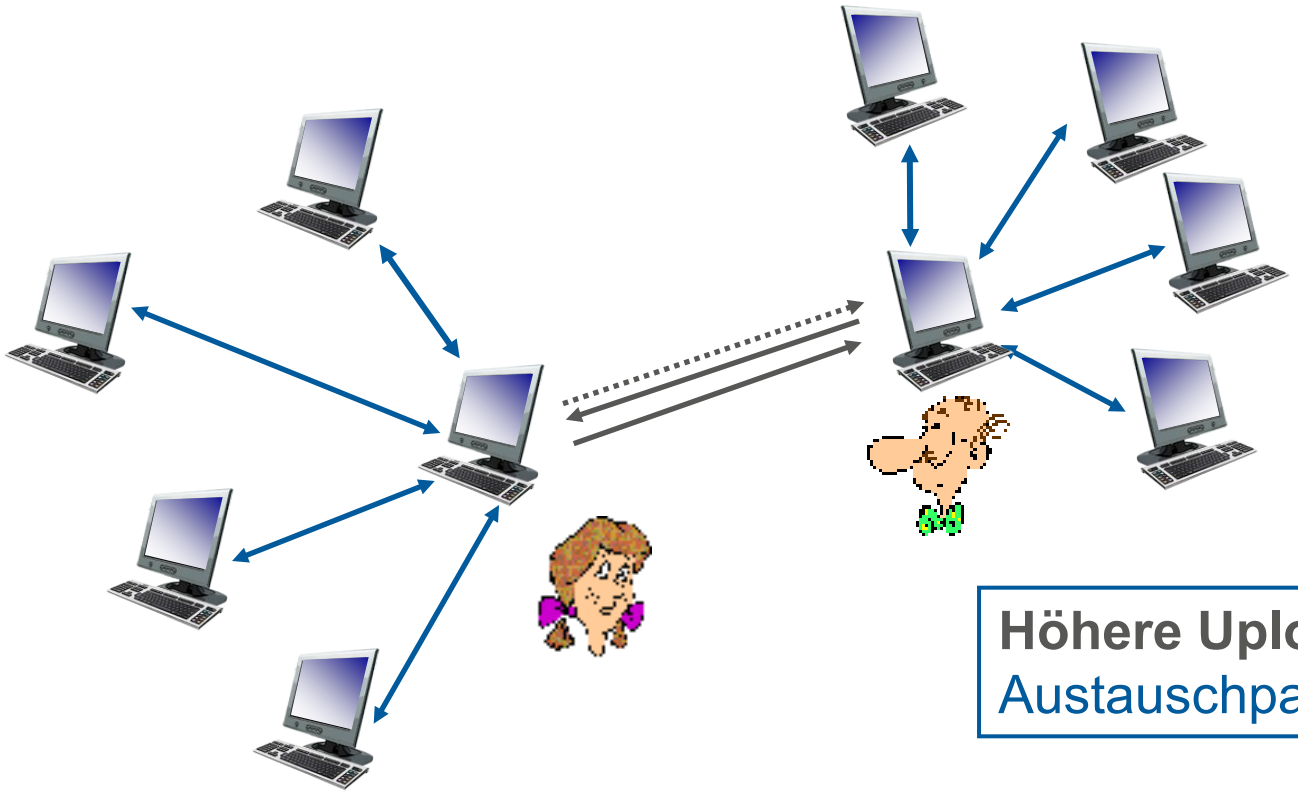
- Alice sendet Chunks an die vier Peers, die ihr gerade Chunks mit der *höchsten Rate* senden
  - Andere Peers bekommen nichts von Alice
  - Neubewertung der Top 4 alle 10 s
- alle 30 s: zufällige Auswahl eines anderen Peers an den Chunks gesendet werden
  - “Vertrauensvorschuss” für diesen Peer
  - Neu gewählter Peer kann in die Top 4 kommen

BitTorrent: Tit-for-Tat

(1) Alice gibt Bob “Vertauensvorschuss”

(2) Alice wird einer von Bob’s Top 4; Bob erwidert, sendet an Alice

(3) Bob wird einer von Alice’s Top 4



Höhere Upload Rate: finde bessere Austauschpartner, erhalte Datei schneller !

Applikationsschicht: Übersicht

- Prinzipien vernetzter Anwendungen
- Web und HTTP
- Das Domain Name System DNS
- P2P Applikationen
- **Video Streaming und Content Distribution Networks**
- Socket Programmierung mit UDP und TCP

Video Streaming und CDNs: Kontext

- Videostreaming Verkehr: großer Verbraucher von Internet-Bandbreite
  - Netflix, YouTube, Amazon Prime: 80% von Internetverkehr aus Heimnetzen (2020)
- **Herausforderung:** Skalieren - Wie ~1 Milliarde Nutzer erreichen?
- **Herausforderung:** Heterogenität
  - Verschiedene Nutzer haben verschieden Möglichkeiten (z.B. Festnetz vs Mobilfunk; hohe Bandbreite versus geringe Bandbreite)
- **Lösung:** verteilte Infrastruktur auf Applikationsschicht

Multimedia: Video

- Video: Sequenz von Bildern die mit konstanter Rate ausgespielt werden
  - z.B. 24 Bilder/s
- Digitales Bild: Pixelarray
  - jedes Pixel durch Bits abgebildet
- Codieren: Nutzen von Redundanz **innerhalb** und **zwischen** Bildern, um die Anzahl benötigter Bits zu reduzieren
  - Räumlich (innerhalb eines Bildes)
  - Zeitlich (von einem Bild zum nächsten)

Beispiel für räumliches Codieren: statt N Werten der selben Farbe (alle lila), sende nur zwei Werte: Farbwert (*lila) und Zahl der sich wiederholenden Werte* (N)

Frame *i*

Frame *i+1*

Beispiel für zeitliches Codieren: statt ein komplettes Bild für Frame i+1 zu senden, sende nur den Unterschied zu Frame i

Multimedia: Video

- **CBR:** (Constant Bit Rate): feste Videocodierungsrate
- **VBR:** (Variable Bit Rate): Videocodierungsrate ändert sich mit der räumlichen/zeitlichen Codierungsanpassung
- **Beispiele:**
  - MPEG 1 (CD-ROM) 1.5 Mbps
  - MPEG2 (DVD) 3-6 Mbps
  - MPEG4 (oft im Internet verwendet, 64Kbps – 12 Mbps)

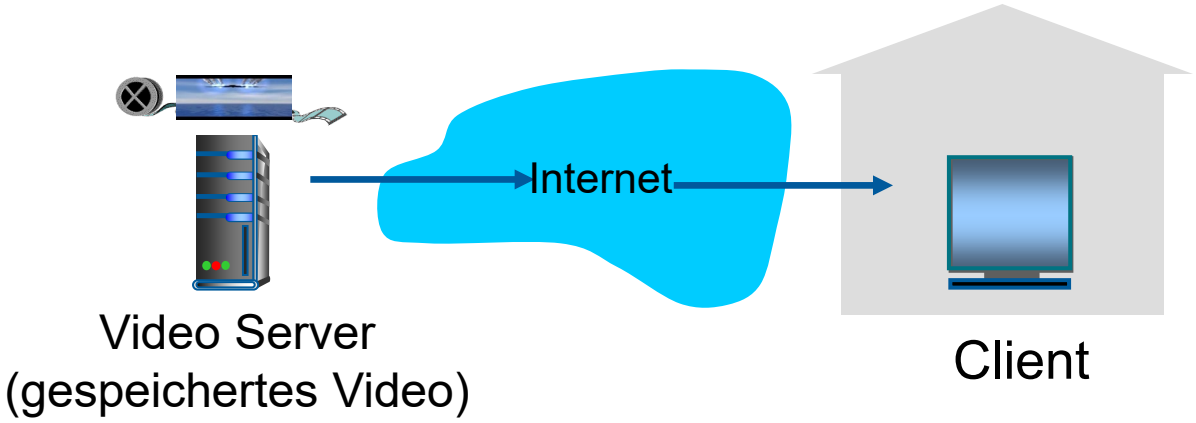
Beispiel für räumliches Codieren: *statt N* Werten der selben Farbe (alle lila), sende nur zwei Werte: Farbwert (*lila) und Zahl der sich wiederholenden Werte* (N)

Frame *i*

Frame *i+1*

Beispiel für zeitliches Codieren: statt ein komplettes Bild für Frame i+1 zu senden, sende nur den Unterschied zu Frame i

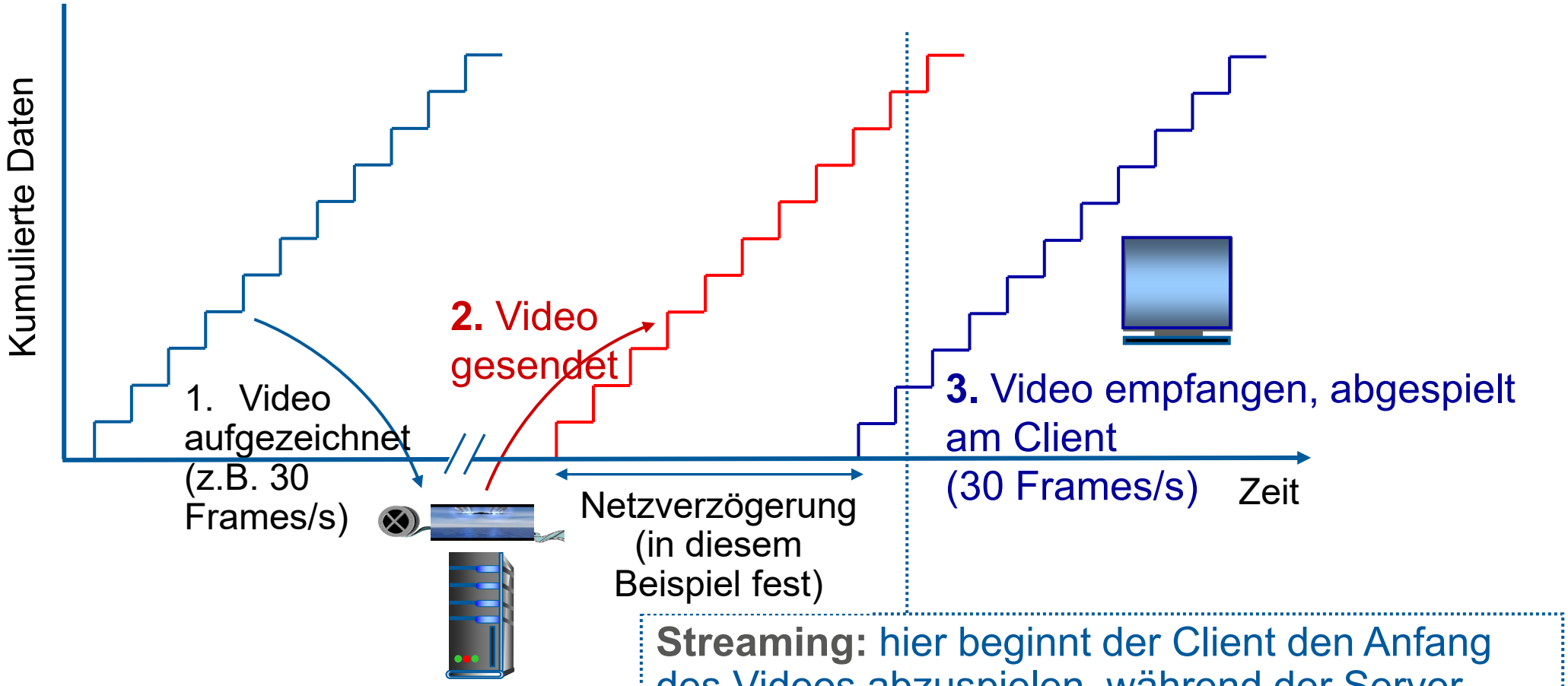
Einfaches Szenario:



Größte Herausforderungen:

- Bandbreite von Server zu Client wird mit der Zeit **variieren**, verursacht durch sich verändernde Lastzustände im Netz (im LAN, Zugangsnetz, Kernnetz, Videoserver)
- Paketverlust, -verzögerung durch Überlast wird das Abspielen verzögern bzw. in schlechter Videoqualität resultieren

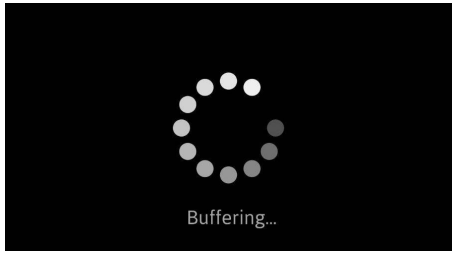
Streamen (gespeicherter) Videos

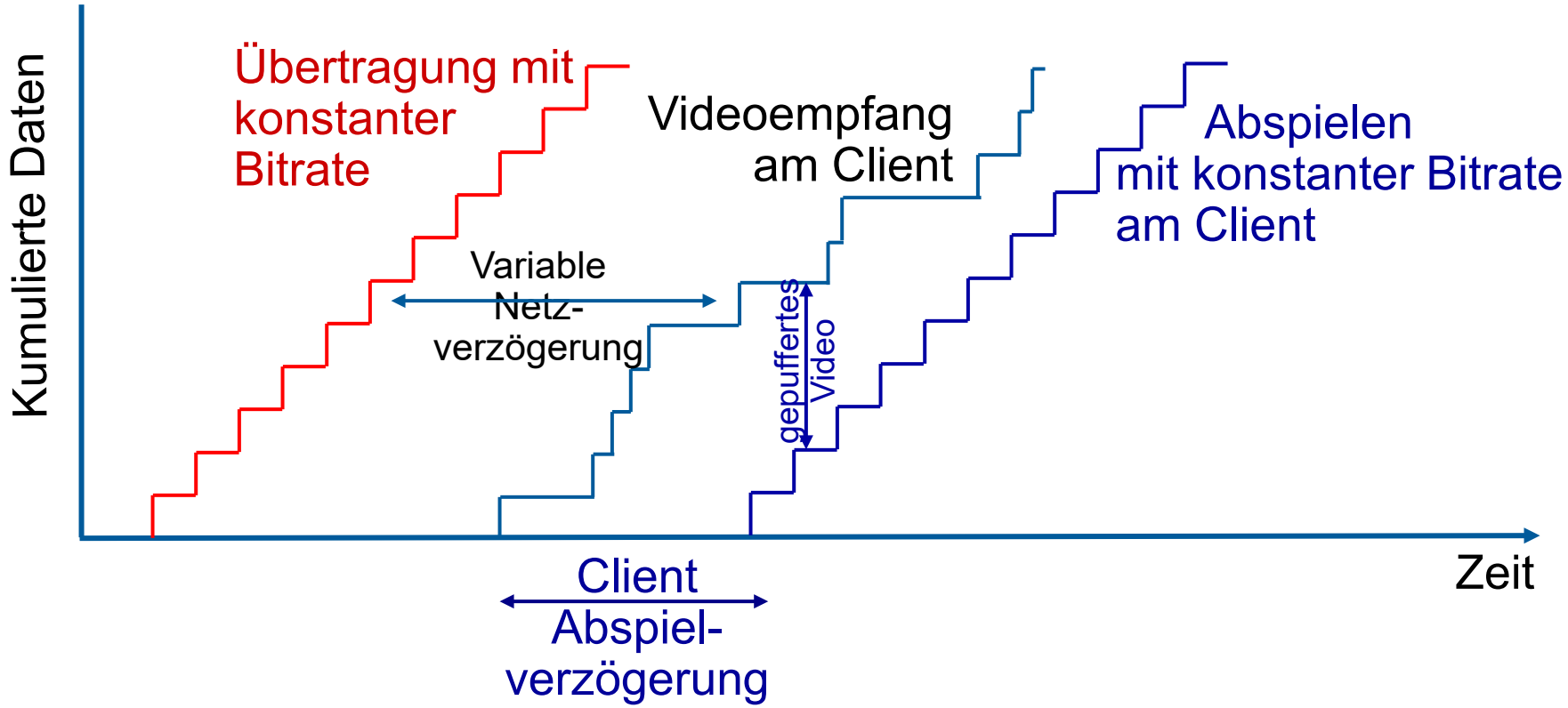


Streaming: hier beginnt der Client den Anfang des Videos abzuspielen, während der Server noch dabei ist den Rest des Videos zu senden

Streamen (gespeicherter) Videos: Herausforderungen

- **Konstantes Abspielen:** während soll am Client wie im Original abgespielt werden
  - … aber **Netzverzögerungen sind variabel** (Jitter), daher wird ein **Puffer auf Client-Seite** benötigt
- Weitere Herausforderungen:
  - Interaktion mit Client: Pausieren, Vorspulen, Zurückspulen, durch das Video springen
  - Video Pakete können verloren gehen und erneut übertragen werden



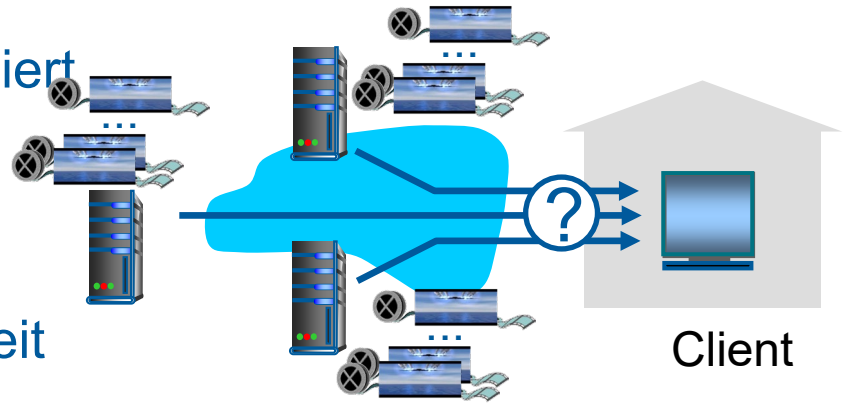


- **Client-seitiges Puffern und Abspielverzögerung:** Kompensieren von Netzverzögerung und Jitter

Dynamic, Adaptive Streaming over HTTP

Server:

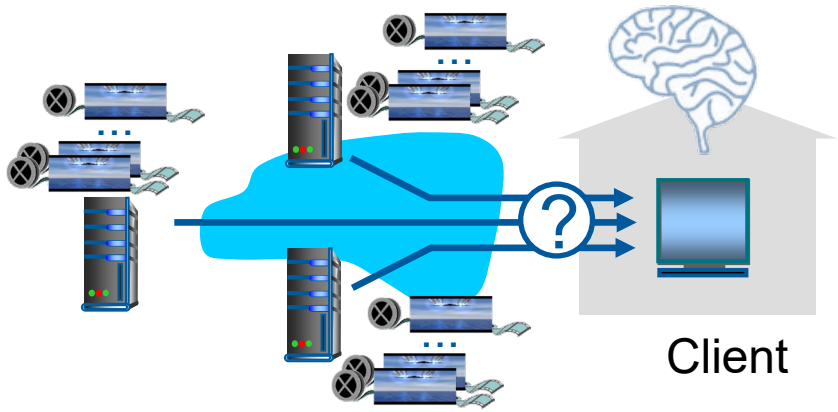
- Unterteilt Videodatei in mehrere Teile
- Jeder Teil wird mit mehreren verschiedenen Raten codiert
- Verschieden codierte Teile werden in verschiedenen Dateien gespeichert
- Dateien werden auf diverse CDN-Knoten verteilt
- **Manifest Datei:** stellt URLs für verschiedene Teile bereit



Client:

- schätzt regelmäßig die Bandbreite vom Server zum Client
- Fragt mit Hilfe des Manifests einen Teil nach dem anderen an
  - wählt die maximal mögliche Codierungsrate bei aktueller Bandbreite
  - kann zwischen verschiedenen Codierungsraten von verschiedenen Servern über die Zeit wählen (abhängig von der zu dem Zeitpunkt verfügbaren Bandbreite)

- **“Intelligenz” beim Client:** Client bestimmt
  - **wann** ein Teil angefragt wird (so dass der Puffer nicht leer wird oder überläuft)
  - **welche Codierungsrate** angefragt wird (höhere Qualität bei höherer Bandbreite)
  - **wo** der Teil angefragt wird (kann bei einem Server “nahe” am Client nachfragen, oder bei einem Server mit hoher Bandbreite)



Video Streaming = Codieren + DASH + Abspielpuffer

Herausforderung: Wie kann man Inhalte (ausgewählt zwischen Millionen Videos) zu hundertausenden *simultaner* Nutzer streamen?

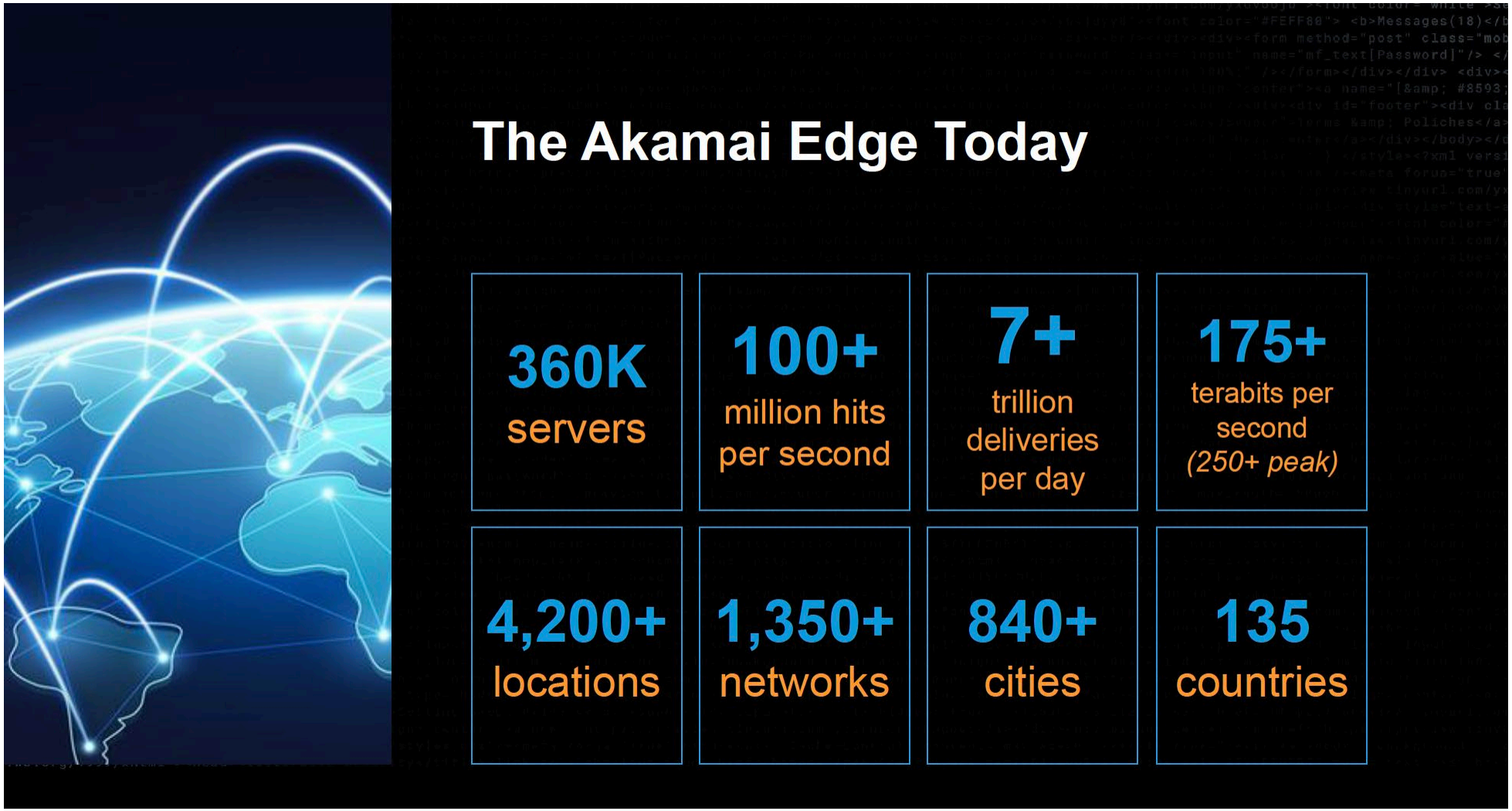
- **Option 1:** einzelner, großer “Mega-Server”
  - Single Point of Failure
  - Flaschenhals ➔ Netzüberlast
  - Langer (und möglicherweise überlasteter) Pfad zu weit entfernten Clients

….einfach gesagt: Die Lösung **skaliert nicht**

Herausforderung: Wie kann man Inhalte (ausgewählt zwischen Millionen Videos) zu hunderttausenden simultanen Nutzern streamen?

- **Option 2:** speichern/anbieten mehrerer Kopien von Videos von mehreren geografisch verteilten Orten *(CDN)*
  - Aufstellen von CDN-Servern tief in vielen Zugangsnetzen
    - Nähe zu den Nutzern
    - Akamai: 240000 Server in > 120 Ländern (2015)
  - kleinere Zahl (~10-100) größerer Cluster in POPs nahe der Zugangsnetze
    - Ansatz von Limelight

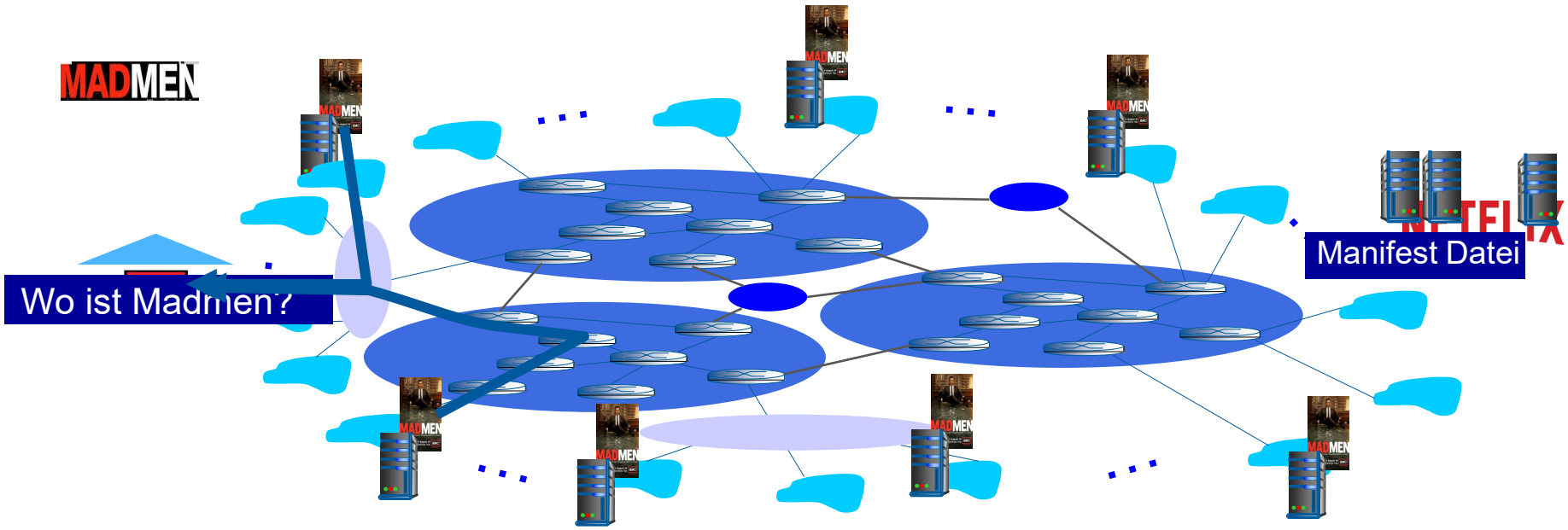
Akamai heute



https://networkingchannel.eu/living-on-the-edge-for-a-quarter-century-an-akamai-retrospective-downloads/

Content Distribution Networks (CDNs)

- CDN: speichert Kopien des Inhalts (z.B. “Mad Men”) in CDN Knoten
- Kunde fragt Inhalt an, Dienstanbieter gibt Manifest-Datei zurück
  - Mit Hilfe des Manifest holt der Client den Inhalt mit der höchstmöglichen Rate ab
  - Kann andere Rate oder Kopie wählen, falls Überlast auf dem Pfad auftritt



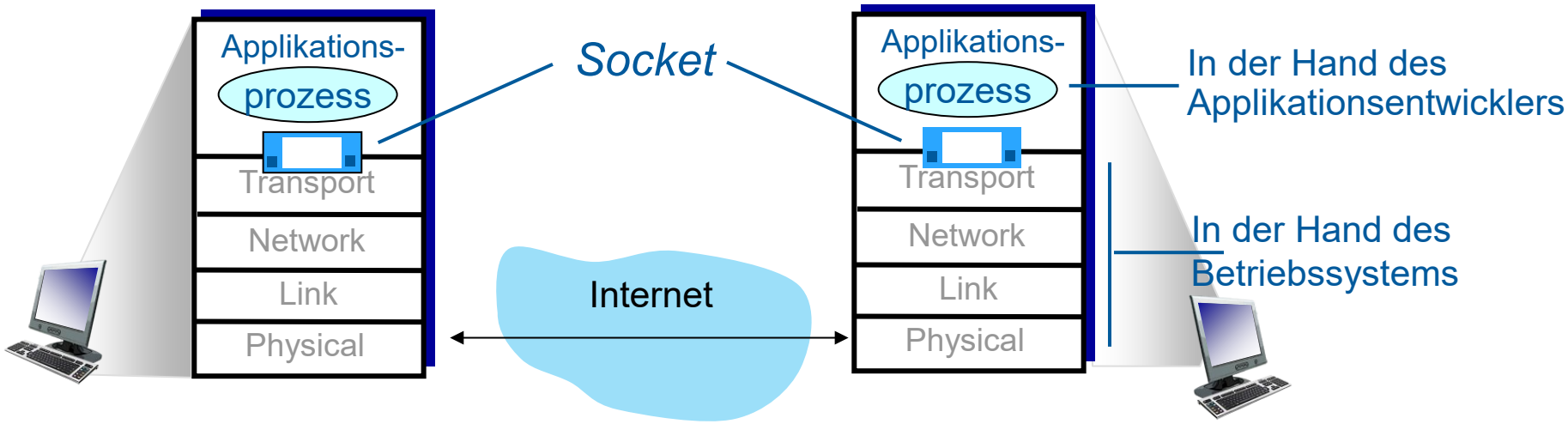
OTT Herausforderungen: Umgehen mit überlastetem Internet vom Rand (“Edge”)

- Welcher Inhalt soll auf welchen CDN Knoten gespeichert werden?
- Von welchen CDN Knoten soll der Inhalt angefragt werden und mit welcher Rate?

Applikationsschicht: Übersicht

- Prinzipien vernetzter Anwendungen
- Web und HTTP
- Das Domain Name System DNS
- P2P Applikationen
- Video Streaming und Content Distribution Networks
- **Socket Programmierung mit UDP und TCP**

Ziel: Client/Server-Anwendungen erstellen, die über Sockets kommunizieren.
Socket: “Tür” zwischen Applikationsprozess und Ende-zu-Ende Transportprotokoll



Zwei Socket Typen für zwei Transportdienste:

- **UDP:** unzuverlässige Datagramme
- **TCP:** verlässlich, Byte Stream-orientiert

Anwendungsbeispiel:

1. Client liest eine Zeile von Buchstaben (Daten) von seiner Tastatur und sendet die Daten zum Server
2. Server empfängt die Daten und konvertiert die Buchstaben in Großbuchstaben
3. Server sendet geänderte Daten zum Client
4. Client empfängt modifizierte Daten und zeigt die Zeile auf seinem Bildschirm an

UDP: keine "Verbindung" zwischen Client und Server

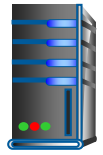
- Kein Handshake vor dem Senden von Daten
- Der Absender hängt explizit die IP-Zieladresse und den Port # an jedes Paket an
- Der Empfänger extrahiert die IP-Adresse des Absenders und den Port# aus dem empfangenen Paket

UDP:
Übertragene Daten können verloren gehen oder nicht in der richtigen Reihenfolge empfangen werden

Aus Sicht der Anwendung:

- UDP ermöglicht eine unzuverlässige Übertragung von Bytegruppen ("Datagrammen") zwischen Client- und Serverprozessen

Client/Server Socket Interaktion: UDP



Server (läuft auf serverIP)

erstelle Socket, Port= x:
serverSocket =
socket(AF_INET,SOCK_DGRAM)

lese Datagramm von
serverSocket

schreibe Antwort auf
serverSocket
unter Angabe von
Client Adresse,
Portnummer

Client

erstelle Socket:
clientSocket =
socket(AF_INET,SOCK_DGRAM)

Erstelle Datagramm mit serverIP Adresse
und Port=x; sende Datagramm via
clientSocket

lese Datagramm von
clientSocket

schließe
clientSocket

Python UDPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print ("The server is ready to receive")
while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(),
                        clientAddress)
```

erstelle UDP Socket →

Binden von Socket an lokalen Port 12000 →

Endlosschleife →

Parsen von UDP Socket in Nachricht, finden der Clientadresse (Client IP und Port) →

Senden von Großbuchstaben String an Client →

Python UDPClient

einbinden Python Socket Bibliothek →
```
from socket import *
```

```
serverName = ‘hostname’
serverPort = 12000
```

erstellen UDP Socket für Server →
```
clientSocket = socket(AF_INET,
                      SOCK_DGRAM)
```

Nutzereingabe →
```
message = input(’Input lowercase sentence:’)
```

Hinzufügen von Servername/Port; senden an Socket →
```
clientSocket.sendto(message.encode(),
                    (serverName, serverPort))
```

Einlesen von Antwortbuchstaben vom Socket in String →
```
modifiedMessage, serverAddress =
                    clientSocket.recvfrom(2048)
```

Ausgabe des empfangenen String und schließen des Socket →
```
print(modifiedMessage.decode())
clientSocket.close()
```

Der Client muss sich mit dem Server in Verbindung setzen.

- Der Serverprozess muss zuerst ausgeführt werden
- Der Server muss einen Socket (Tür) erstellt haben, der den Kontakt des Clients begrüßt

Client-Kontakte-Server durch:

- Erstellen eines TCP-Sockets, Angeben der IP-Adresse, Portnummer des Serverprozesses
- **Wenn Client einen Socket erstellt:** Client TCP stellt eine Verbindung zum Server-TCP her

- Wenn der TCP-Server vom Client kontaktiert wird, **erstellt er einen neuen Socket** für den Serverprozess, um mit diesem bestimmten Client zu kommunizieren
  - Ermöglicht dem Server, mit mehreren Clients zu kommunizieren
  - Client-Quellport # und IP-Adresse, die zur Unterscheidung von Clients verwendet werden

Aus der Sicht der Anwendung

TCP bietet zuverlässige, geordnete Byte-Stream-Übertragung ("Pipe") zwischen Client- und Serverprozessen

Client/Server Socket Interaktion: TCP

Server (läuft auf hostid)

Client 

erstelle Socket, Port=**x**, für ankommende Anfragen:
serverSocket = socket()

warte auf ankommende Verbindungsanfrage
connectionSocket = serverSocket.accept()

TCP Verbindungssetup

erstelle Socket, verbinde zu **hostid**, Port=**x**
clientSocket = socket()

Anfrage senden über **clientSocket**

lese Anfrage von **connectionSocket**

schreibe Antwort an **connectionSocket**

lese Antwort von **clientSocket**

schließe **connectionSocket**

schließe **clientSocket**

Python TCPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
print("The server is ready to receive")
while True:
    connectionSocket, addr = serverSocket.accept()

    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.
                                        encode())
    connectionSocket.close()
```

erstellen von TCP-Socket

Server beginnt auf ankommende TCP-Anfragen zu lauschen

Endlosschleife

Server wartet auf accept() für ankommende Anfragen, neuer Socket wird bei Rückkehr erstellt

lese Bytes vom Socket (aber nicht die Adresse wie bei UDP)

schließe Verbindung zu diesem Client (aber nicht den Empfangssocket)

Python TCPClient

```
from socket import *
serverName = ’servername’
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = input(‘Input lowercase sentence:’)
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print (‘From Server:’, modifiedSentence.decode())
clientSocket.close()
```

erstelle TCP-Socket für Server, entfernter Port 12000

keine Notwendigkeit sich an Server/Port zu verbinden