



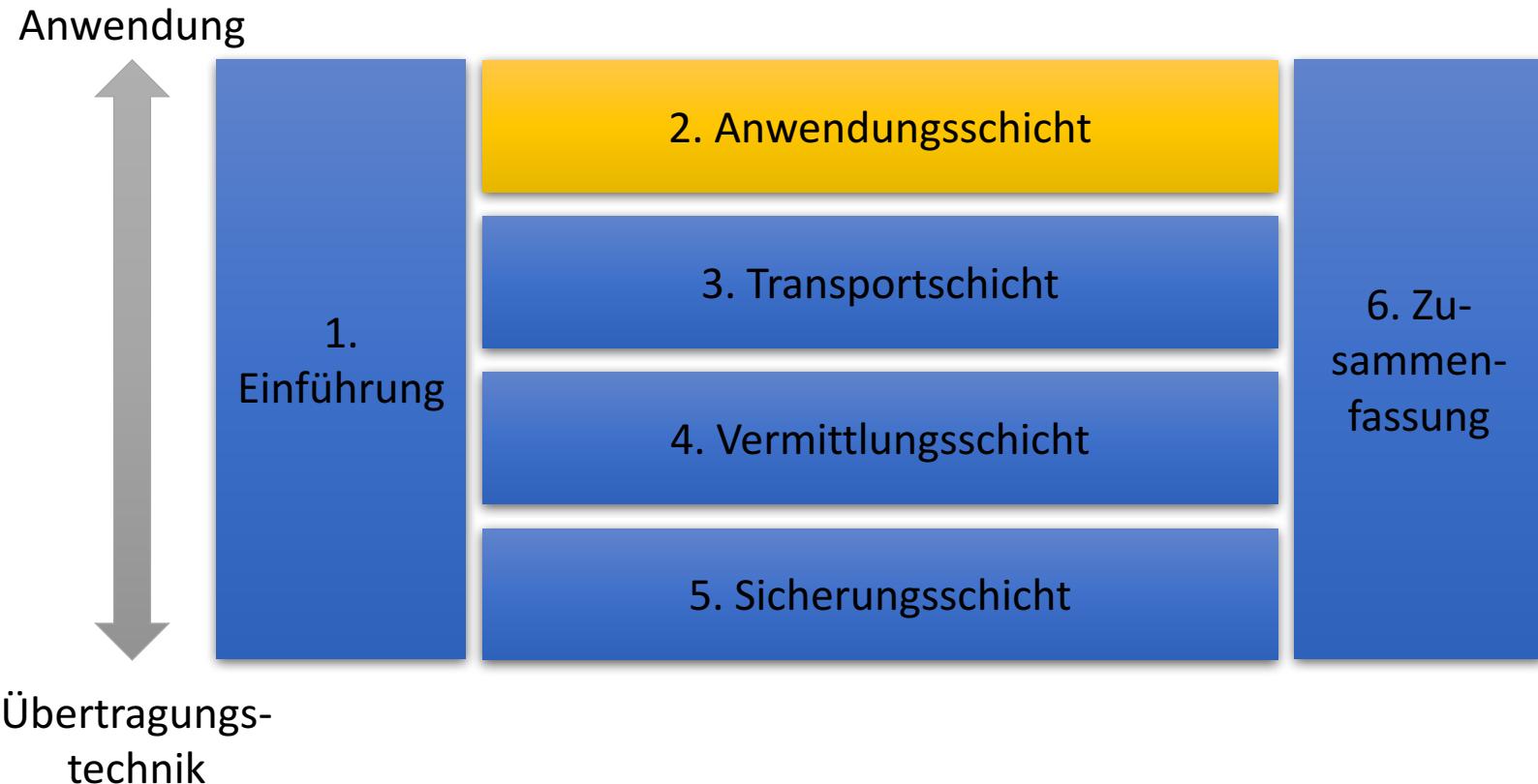
Kapitel 2  
**Die Anwendungsschicht**

Vorlesung Kommunikationsnetze 1  
Wintersemester 2017/18

Oliver P. Waldhorst

(Basierend auf Materialien von J. Kurose und K. Ross © 1996-2017)

# Gliederung der Vorlesung



# Ziele dieses Kapitels

Kennenlernen der Anwendungsschicht aus Konzept- und Implementierungssicht

- Dienstmodelle der Transportschicht
- Client-Server-Anwendungen
- Peer-to-Peer-Anwendungen
- Socket-Programmierung

Kennenlernen von Anwendungsschichtprotokollen anhand von Beispielen:

- HTTP
- SMTP/POP3/IMAP
- DNS

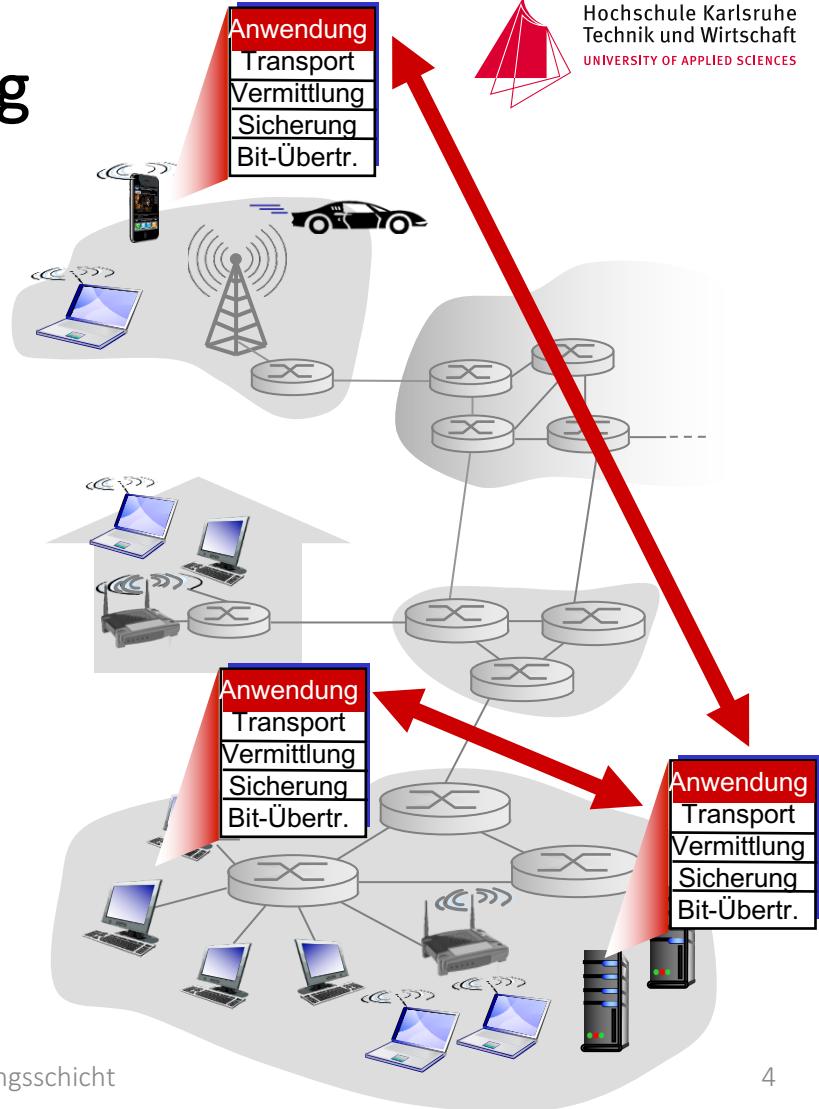
# Erstellen einer Netzanwendung

Schreiben eines Programm das

- auf verschiedenen Endsystemen läuft
- über das Netz (Internet) kommuniziert
- Beispiel: Browser Software kommuniziert mit Web Server Software

Es ist nicht notwendig, Software für Geräte im Netzinneren zu schreiben

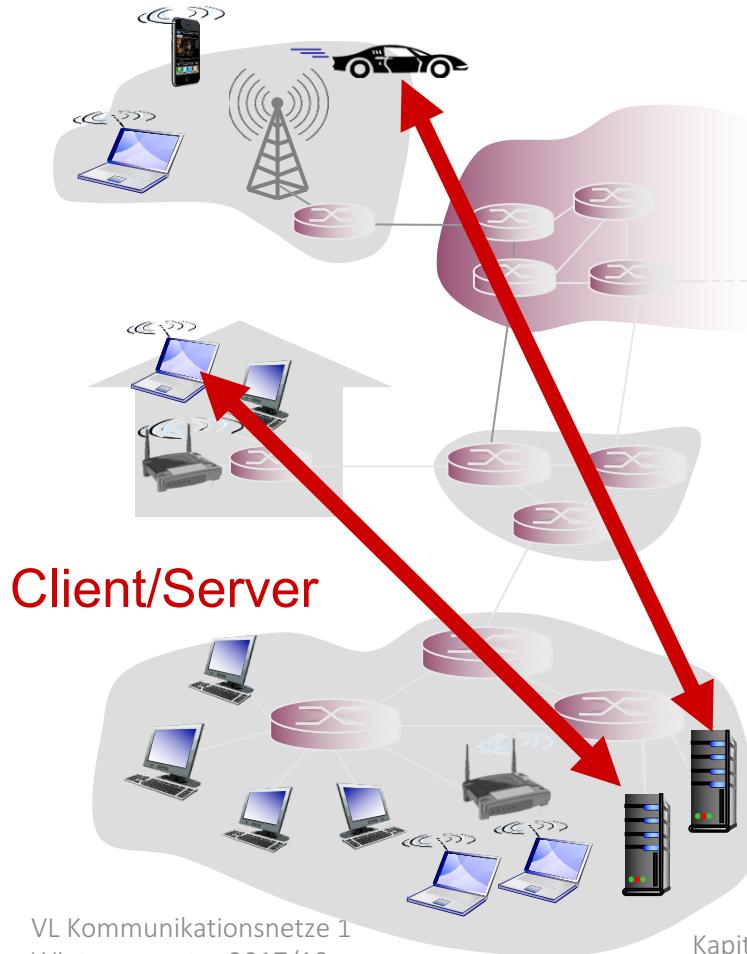
- Diese müssen keine Anwendungsspezifische Funktionalität bereitstellen
- Erleichtert Entwicklung und Ausbringen von neuen Anwendungen



# Typische Anwendungsarchitekturen

- Client-Server
- Peer-to-Peer (P2P)

# Client-Server-Architekturen



## Server:

- Ständig verfügbar
- Feste IP-Adresse
- Ggf. in Rechenzentrum „Cloud“

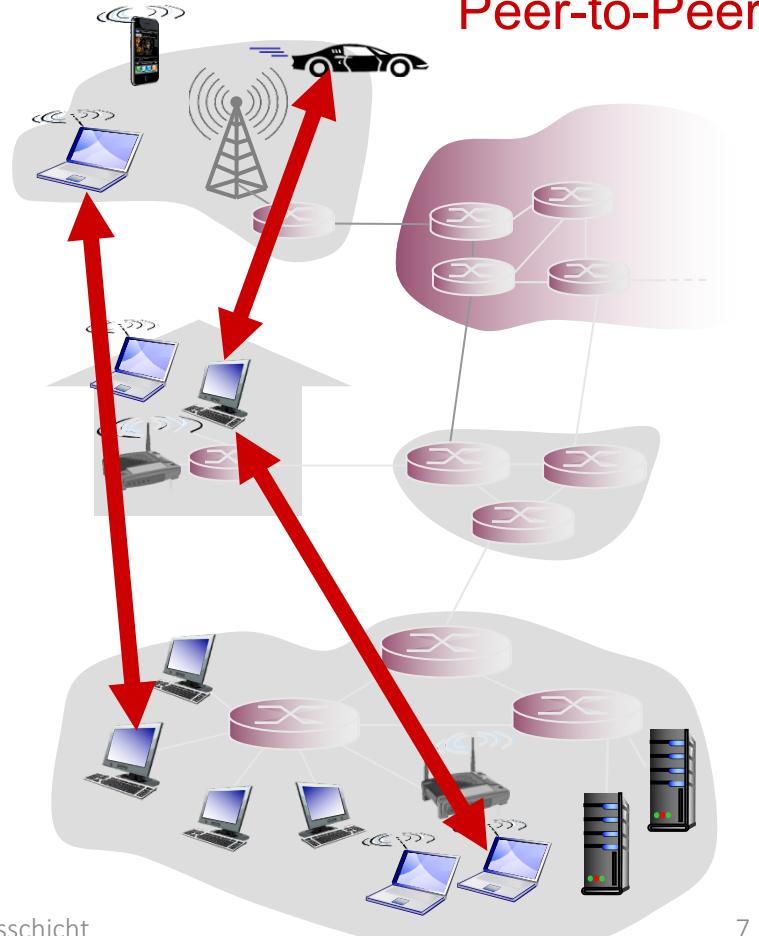
## Clients:

- Kommunizieren mit dem Server
- Verbindungen zum Netz können getrennt werden („*intermittently connected*“)
- Kommunizieren nicht direkt miteinander

# Peer-to-Peer (P2P) Architektur

Kein ständig verfügbarer Server

- Direkte Kommunikation zwischen Vielzahl von Endgeräten („Peers“)
- Peers nutzen Dienste von anderen Peers, stellen selber Dienste bereit
  - Selbstskalierend – neue Peers bringen sowohl Kapazität als auch Bedarf mit
- Peers sind nicht ständig verbunden und können IP-Adressen wechseln



# Kommunikation zwischen Prozessen

**Prozess:** Ein Programm auf einem Endgerät

- Zwei Prozesse auf dem selben Endgerät kommunizieren über **Interprozesskommunikation** (vgl. Vorlesung Betriebssysteme)
- Prozesse auf verschiedenen Endgeräten kommunizieren durch Austausch von **Nachrichten** über das Netz

Unterscheidung Client- & Server-Prozesse

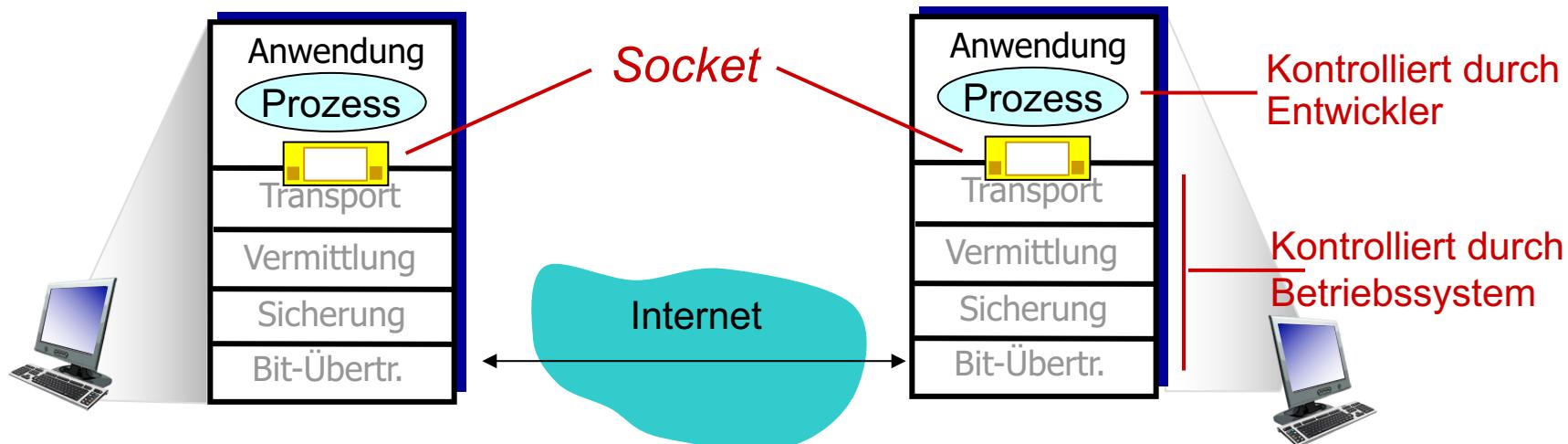
- **Client-Prozess:** Initiiert die Kommunikation
- **Server-Prozess:** Wartet darauf, kontaktiert zu werden

*Anmerkung: Auch in Anwendungen mit Peer-to-Peer-Architektur gibt es Client- und Server-Prozesse*

# Sockets

Prozesse senden und Empfangen Nachrichten über **Sockets**

- Analogie: Socket ist der „Briefkasten“ der Prozesse
  - Sender Prozess „wirft Nachricht in den Briefkasten“
  - Sender Prozess verlässt sich auf Transportinfrastruktur der Post (d.h. Dienste der Transportschicht), die Nachricht zum empfangenen Prozess bringt und dort in den Briefkasten wirft



# Was erwarten Anwendungen von der Transportschicht?

## Zuverlässigkeit

- Manche Anwendungen (z.B. Dateiübertragung, Bank-Transaktionen) erwarten 100% zuverlässigen Transport
- Andere Anwendungen (z.B. Audio) können einige Datenverluste tolerieren

## Timing

- Manche Anwendungen (z.B. Internet Telefonie, Spiele) erwarten niedrige Verzögerungen, um nutzbar zu sein

## Durchsatz

- Manche Anwendungen (e.g., Multimedia) brauchen einen Mindestdurchsatz, um nutzbar zu sein
- Andere Anwendungen („elastische“) nutzen jeden Durchsatz, den sie bekommen können

## Sicherheit

- Verschlüsselung, Datenintegrität, ...

# ! Übung: Definieren Sie typische Anforderungen an die Transportschicht!

Anwendung	Kein Datenverlust	Hoher Durchsatz	Geringe Verzögerung
Dateiübertragung			
Email			
Web Dokumente			
Echtzeit Audio/Video			
Gesp. Audio/Video			
Interaktive Spiele			
Textnachrichten			

# Dienste der Transportprotokolle im Internet (mehr dazu in Kapitel 3)

## TCP Dienst

- Zuverlässiger Transport
- Flusskontrolle: Sender „überschwemmt“ Empfänger nicht mit Nachrichten
- Staukontrolle: Bei Überlastung des Netz wird Sender gedrosselt
- Verbindungsorientiert: Verbindungsauf- und -abbau benötigt
- TCP bietet nicht: Timing, Durchsatzgarantien, Sicherheit

## UDP Dienst

- Unzuverlässiger Transport
- UDP bietet nicht: Zuverlässigkeit, Fluss- und Staukontrolle, Timing, Durchsatzgarantien, Verbindungen, Sicherheit

Frage: Warum brauchen wir überhaupt UDP?

# Anwendungen und Transportprotokolle

<b>Anwendung</b>	<b>Anwendungs- protokoll</b>	<b>Verwendetes Transportprotokoll</b>
E-Mail	SMTP [RFC 2821]	TCP
Remote Terminal Access	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
Dateiübertragung	FTP [RFC 959]	TCP
Multimediasreaming	HTTP (e.g., YouTube), RTP [RFC 1889]	TCP oder UDP
Internet Telefonie	SIP, RTP, proprietär (e.g., Skype)	TCP oder UDP

# Sicherer Transport

TCP & UDP bieten keine Verschlüsselung

- Z.B. Passworte im Klartext werden im Klartext über Internet übertragen

Abhilfe schafft **Secure Socket Layer (SSL)** → Vgl. VL IT-Sicherheit

- Stellt verschlüsselte TCP-Verbindungen bereit
- Sorgt für Datenintegrität
- Identifiziert Verbindungsendpunkte

SSL liegt in der Anwendungsschicht

- Anwendungen nutzen SSL-Bibliothek um mit TCP zu kommunizieren
- So werden z.B. Passworte im Klartext an SSL gesendet und verschlüsselt über TCP übertragen

# Adressierung von Prozessen

Um eine Nachricht erhalten zu können, benötigt Prozess eine **Kennung (Identifier)**

- Endsysteme haben **32-bit IP-Adressen** (vgl. Kapitel 4)
- Frage: Reicht die IP-Adresse zum Zustellen aus?
  - Antwort: Nein, es können viele Prozesse auf dem selben Endsystem laufen!
- Kennung auf Anwendungsschicht besteht aus IP-Adresse und **Port-Nummer**, z.B.
  - HTTP Server: 80
  - E-Mail Server: 25
  - Für HTTP-Anfrage an [www.hs-karlsruhe.de](http://www.hs-karlsruhe.de):  
IP-Adresse: **193.196.64.99**      Port: **80**

# Durch Anwendungsprotokoll definiert

**Typen** von ausgetauschten Nachrichten

- Z.B. Anfrage, Antwort, ...

**Nachrichten-Syntax**

- Felder in der Nachricht und deren Darstellung

**Nachrichten-Semantik**

- Bedeutung der Informationen in den Feldern

**Regeln** wie und wann Nachrichten zu senden und zu beantworten sind

Beispiele für Protokolle

- Offen (definiert in RFCs): HTTP, SMTP, ...
- Proprietär: Skype, ...

# World Wide Web und HTTP

## Grundsätzliches

- Webseite besteht aus mehreren Objekten
  - HTML-Datei, JPEG Bild, Java Applet, Audiodatei, ...
- Zusammenhang: Basis ist HTML-Datei, die viele Objekte referenziert
- Jedes Objekt kann über einen Uniform Resource Locator ([URL](#), [RFC 3986]) adressiert werden:

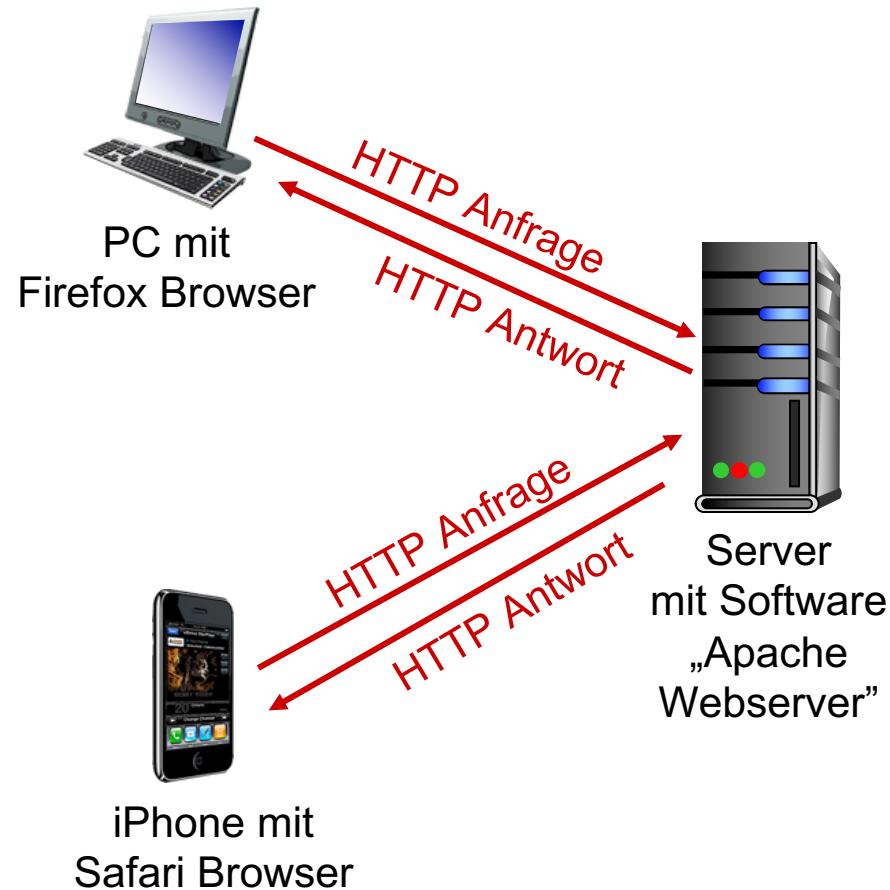
[www.hs-karlsruhe.de/fk-iwi.html](http://www.hs-karlsruhe.de/fk-iwi.html)

Host-Name

Pfad

# Hypertext Transfer Protokoll (HTTP) – Überblick

- HTTP ist das Anwendungsprotokoll des World Wide Web
- Client-Server-Modell
  - Client: Browser, der Web-Objekte anfragt und empfängt (mittels HTTP) sowie anzeigt
  - Server: Web Server sendet auf Anfrage Objekte an Clients (mittels HTTP)



# Hypertext Transfer Protokoll (HTTP) – Überblick

HTTP verwendet TCP

- Client initiiert TCP-Verbindung (öffnet Socket) zum Server, Port 80
- Server akzeptiert die TCP Verbindung
- HTTP-Nachrichten (Nachrichten auf Anwendungsschicht) werden zwischen Browser (HTTP Client) und Web Server (HTTP Server) ausgetauscht
- TCP-Verbindung wird geschlossen

HTTP ist **zustandslos**

- Server speichert keine Informationen über vorausgegangene Verbindungen
- Reduziert Komplexität drastisch!

# HTTP Verbindungen

## Nicht-persistentes HTTP

- Über eine TCP-Verbindung wird maximal ein Objekt übertragen
  - Danach wird die Verbindung geschlossen
- Übertragung von mehreren Objekten erfordert mehrere Verbindungen!
- War Standardverhalten bei HTTP/1.0

## Persistentes HTTP

- Über eine TCP-Verbindung können viele Objekte übertragen werden
  - **Pipelining** erlaubt: Client darf viele Anfrage unmittelbar hintereinander senden, ohne Antwort abzuwarten
- Standardverhalten ab HTTP/1.1

Beispiel: Anfrage [www.hs-karlsruhe.de/fk-iwi.html](http://www.hs-karlsruhe.de/fk-iwi.html)  
(enthält 10 JPEG Objekte, nicht-persistentes HTTP)

### Ia. HTTP Client initiiert TCP

Verbindung zu HTTP  
Server (Prozess) unter  
[www.hs-karlsruhe.de](http://www.hs-karlsruhe.de),  
Port 80

### 2. HTTP Client sendet

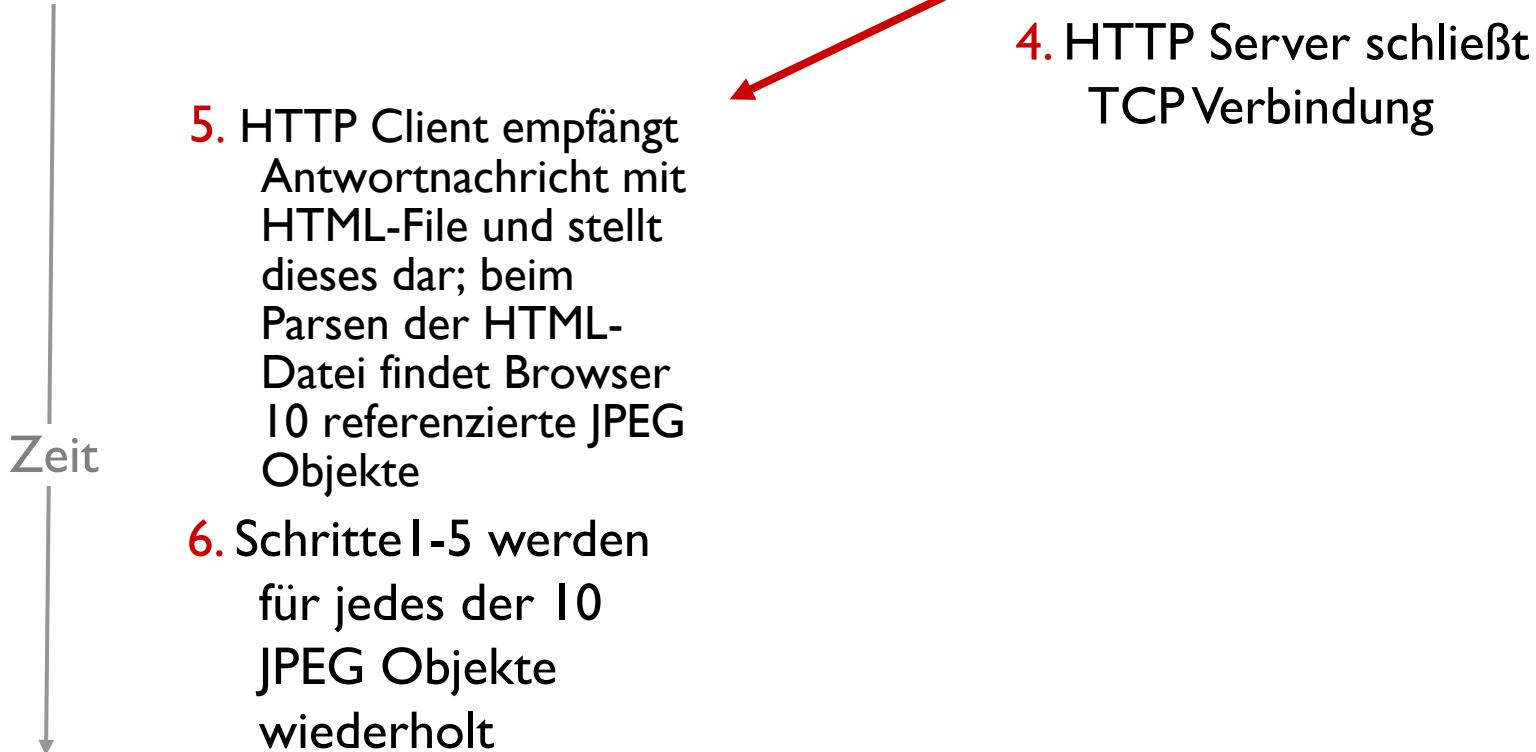
**HTTP-Anfragenachricht**  
(enthält URL) über TCP-  
Verbindungs-Socket;  
Nachricht besagt, dass  
der Client [fk-iwi.html](http://www.hs-karlsruhe.de/fk-iwi.html)  
beziehen möchte

Ib. HTTP Server auf Host  
[www.hs-karlsruhe.de](http://www.hs-karlsruhe.de) wartet  
auf TCP Verbindungen auf Port  
80; „Akzeptiert“ Verbindung;  
benachrichtigt Client

3. HTTP Server empfängt  
Anfragenachricht, erzeugt  
**HTTP-Antwortnachricht**, die  
gewünschtes Objekt enthält  
und sendet diese in das Socket

Zeit

Beispiel: Anfrage [www.hs-karlsruhe.de/fk-iwi.html](http://www.hs-karlsruhe.de/fk-iwi.html)  
(enthält 10 JPEG Objekte, nicht-persistentes HTTP)

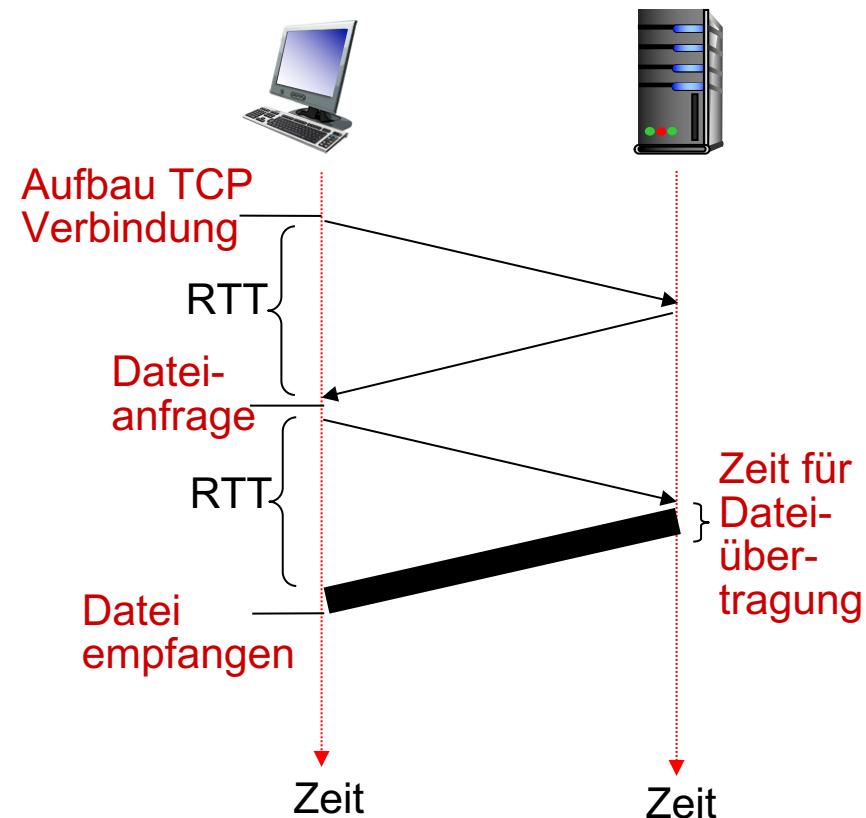


# Nicht-persistentes HTTP: Antwortzeiten

**Round Trip Time (RTT)**: Zeit, die ein kleines Paket benötigt, um von Client zum Server und zurück zu gelangen

**HTTP Antwortzeit**:

- Eine RTT für Aufbau der TCP Verbindung (vgl. Kap. 3)
- Eine RTT für HTTP Anfrage und Übertragung der ersten Bytes der HTTP Antwort
- Übertragungszeit der Dateien
- Antwortzeit für nicht-persistentes HTTP = **2RTT + Übertragungszeit**



# Vergleich persistentes vs. nicht-persistentes HTTP

## Probleme nicht-persistentes HTTP

- Braucht 2 RTT pro Objekt
- Zusätzlich Overhead des Betriebssystems für jede TCP Verbindung
- Häufig öffnen Browser parallele TCP-Verbindungen um referenzierte Objekte anzufordern

## Persistentes HTTP

- Server hält Verbindung nach Senden der Antwort offen
- Nachfolgende HTTP-Nachrichten zwischen Client und Server werden über dieselbe Verbindung übertragen
- Client sendet HTTP-Anfrage sobald er referenziertes Objekt findet
  - Lediglich eine RTT für alle referenzierten Objekte

# ! Übung: Persistentes HTTP mit Pipelining

Wie würde der Ablauf von Folie 23 aussehen, wenn

- die Web-Seite zwei eingebettete JPG-Objekte enthält und
- Browser und Server persistentes HTTP mit Pipelining unterstützen (und auch verwenden)?

Wie lange dauert dann die gesamte Übertragung?

# HTTP-Anfragenachricht

- Zwei Typen von HTTP-Nachrichten: Anfrage, Antwort
- HTTP-Anfrage-Nachricht
  - ASCII (lesbar)

Anfragezeile  
(GET, POST,  
HEAD Kommandos)

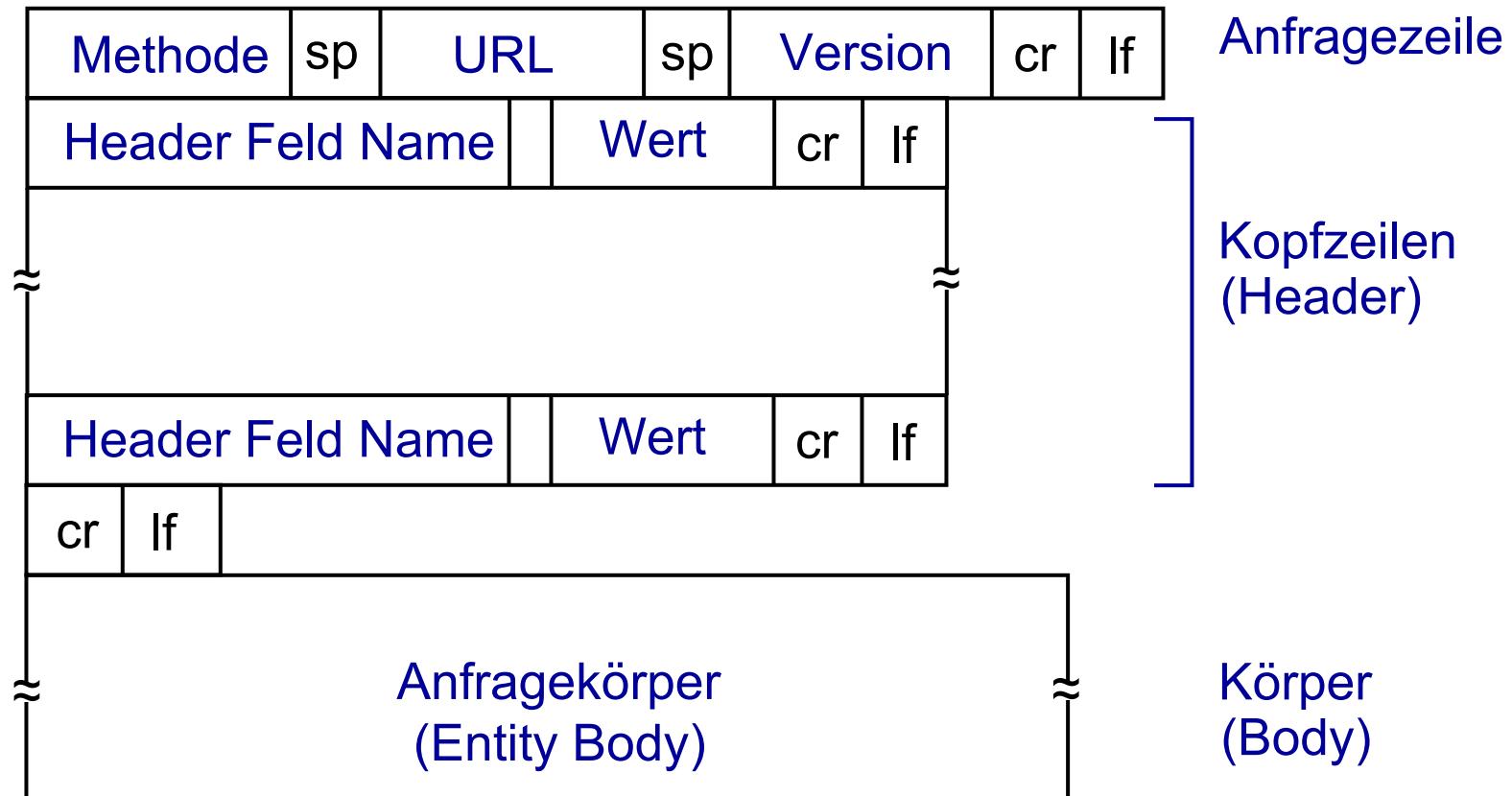
Kopfzeilen  
(Header)

Wagenrücklauf,  
Zeilenvorschub  
am Zeilenanfang  
zeigt Header-Ende

```
GET /fk-iwi.html HTTP/1.1\r\n
Host: www.hs-karlsruhe.de\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
```

Wagenrücklauf  
Zeilenvorschub

# HTTP-Anfrage-Nachricht: Generelles Format



# Methodentypen

## HTTP/1.0

- **GET**
- **POST**
  - Übertragung von Daten im Body, z.B. Formulareingabe
- **HEAD**
  - Überträgt alles außer dem Angefragten Objekt

## HTTP/1.1

- **GET, POST, HEAD**
- **PUT**
  - Hochladen einer Datei im Body in den in der URL spezifizierten Pfad
- **DELETE**
  - Löschen der durch die URL spezifizierten Datei



# Übertragung von Benutzereingaben

## POST-Methode

- Web-Seiten enthalten oft Formulare für Benutzereingaben
- Eingaben werden im HTML Body zum Server übertragen

## URL-Methode

- Benutzung der GET-Methode
- Eingabe wird in der URL übertragen:

**www.somesite.com/animalsearch?monkeys&banana**

# HTTP-Antwortnachricht

Statuszeile  
(Protokoll  
Status-Code  
Status-Meldung)

Kopfzeilen  
(Header)

Daten, z.B.,  
angefragte  
HTML Datei

```
HTTP/1.1 200 OK\r\nDate: Mon, 20 Mar 2017 10:11:42 GMT\r\nServer: Apache/2.4.7 (Ubuntu)\r\nX-Powered-By: PHP/5.5.9-1ubuntu4.21\r\nVary: Accept-Encoding\r\nContent-Encoding: gzip\r\nContent-Type: text/html; charset=utf-8\r\nKeep-Alive: timeout=5, max=500\r\nConnection: Keep-Alive\r\nTransfer-Encoding: chunked\r\n\r\ndata data data data data ...
```

# HTTP-Antwort Status Codes

Statuscode erscheint in der ersten Zeile in der Antwortnachricht

- 1xx – Information
- 2xx – Erfolgreiche Operation
- 3xx – Umleitung
- 4xx – Client-Fehler
- 5xx – Server-Fehler

Beispiele:

- **102 Processing:** Vermeidung von Timeout bei zeitintensiver Anfrage
- **200 OK:** Anfrage erfolgreich, angefragtes Objekt folgt in dieser Nachricht
- **301 Moved Permanently:** Angefragtes Objekt wurde verschoben, neuer Ort folgt in dieser Nachricht („Location:“-Header)
- **400 Bad Request:** Server hat Anfragenachricht nicht verstanden
- **404 Not Found:** Angefragtes Dokument konnte auf dem Server nicht gefunden werden
- **505 HTTP Version Not Supported**

# ! HTTP zum Ausprobieren

## I. Verbinden Sie sich per Telnet zum Web-Sever:

```
telnet www.hs-karlsruhe.de 80
```

Öffnet TCP Verbindung zu Port 80  
(Standard-Port für HTTP Server)  
von **www.hs-karlsruhe.de**  
Jede Eingabe in Telnet wird gesendet  
an Port 80 von **www.hs-karlsruhe.de**

## 2. Geben Sie eine GET HTTP Anfrage ein:

```
GET /fk-iwi.html HTTP/1.1  
Host: www.hs-karlsruhe.de
```

Durch diese Eingabe (inkl. 2x Return),  
senden Sie diese minimale (aber  
vollständige)  
GET Anfrage an den HTTP-Server

## 3. Schauen Sie sich die Antwort-Nachricht des HTTP-Servers an!



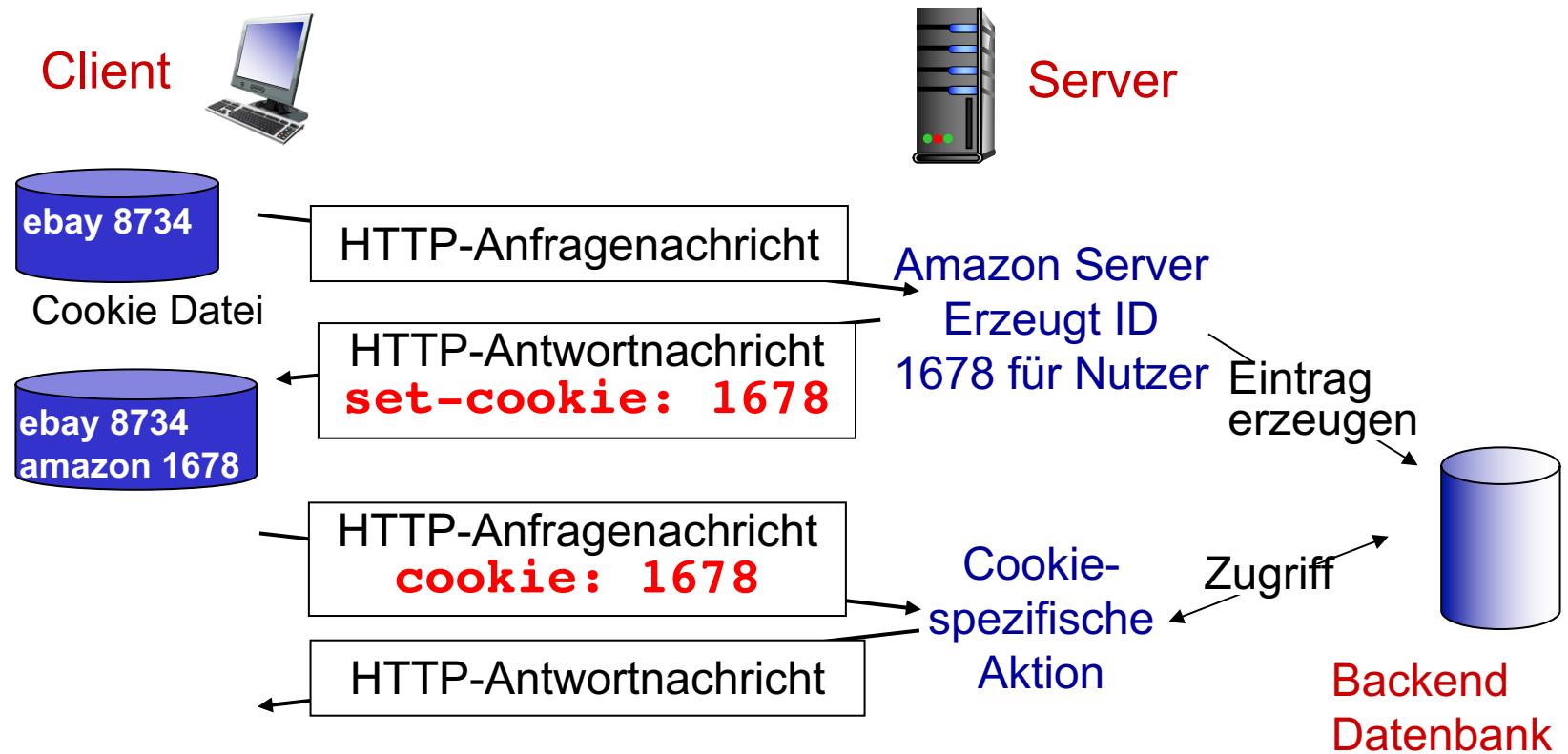
# Benutzer-Status auf dem Server: Cookies

Viele Web-Seiten nutzen **Cookies**

Vier Bausteine:

1. „**Set-cookie:**“ Header in der HTTP Antwortnachricht
2. „**Cookie:**“ Header in der nächsten HTTP Anfragenachricht
3. Cookie-Datei auf dem Endsystem den Benutzers, verwaltet vom Web-Browser
4. Backend Datenbank auf Webseite

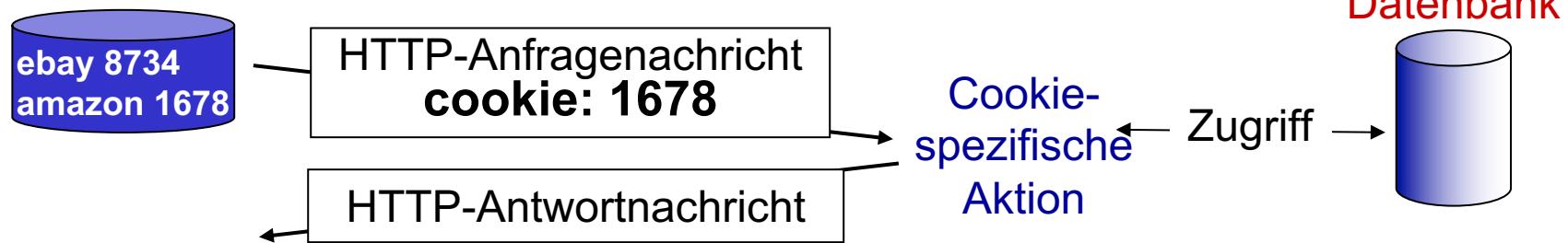
# Zustandshaltung mit Cookies





# Zustandshaltung mit Cookies

Eine Woche später...





# Bemerkungen zu Cookies

Wofür Cookies gebraucht werden können

- Autorisierung
- Einkaufswagen
- Empfehlungen
- Zustand einer Nutzersitzung (z.B. Web-Email)

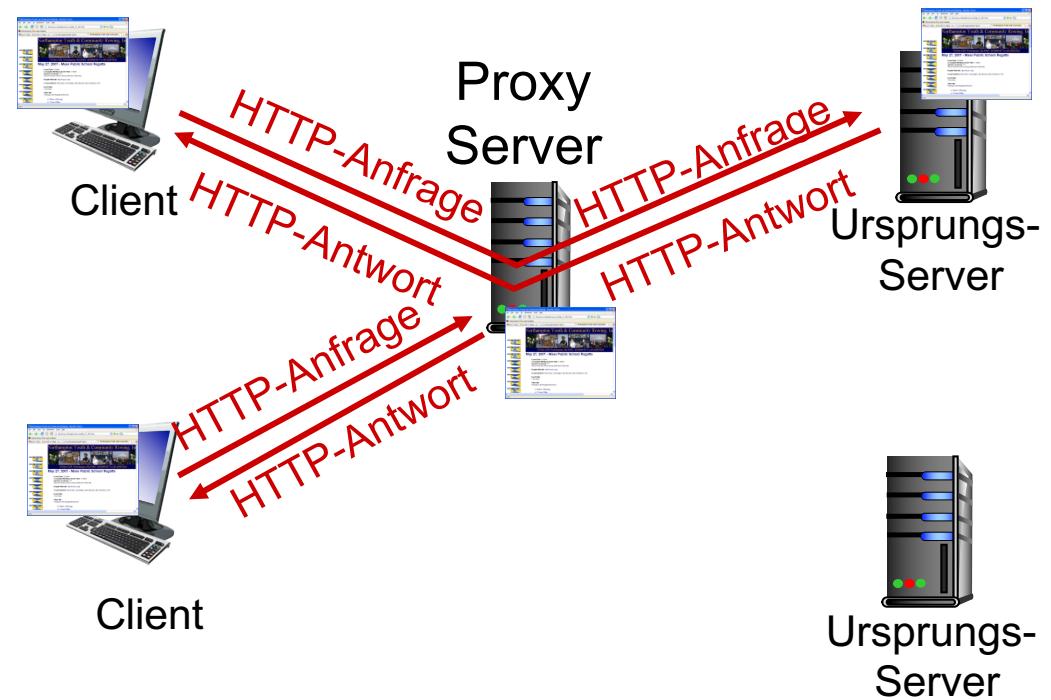
Unterschiedliche Arten der Zustandshaltung!

- Protokollendpunkte: Halten Zustand bei Sender/Empfänger über mehrere Transaktionen
- Cookies: Zustand wird in HTTP-Nachrichten übertragen

# Web-Cache (Proxy Server)

Ziel: Web-Anfragen ohne Kontakt zum **Ursprungs-Server (Origin Server)** eines Web-Objekts beantworten

- Browser-Konfiguration: Alle Zugriffe über Web Cache
- Browser sendet alle HTTP-Anfragen an den Web-Cache
  - Objekt im Cache: Cache liefert Objekt zurück
  - Ansonsten bezieht Cache Objekt vom Ursprungs-Server, speichert Kopie, liefert an Browser aus



# Bedingtes (Conditional) GET

Ziel: Objekt nicht übertragen, wenn Cache eine aktuelle Version hat

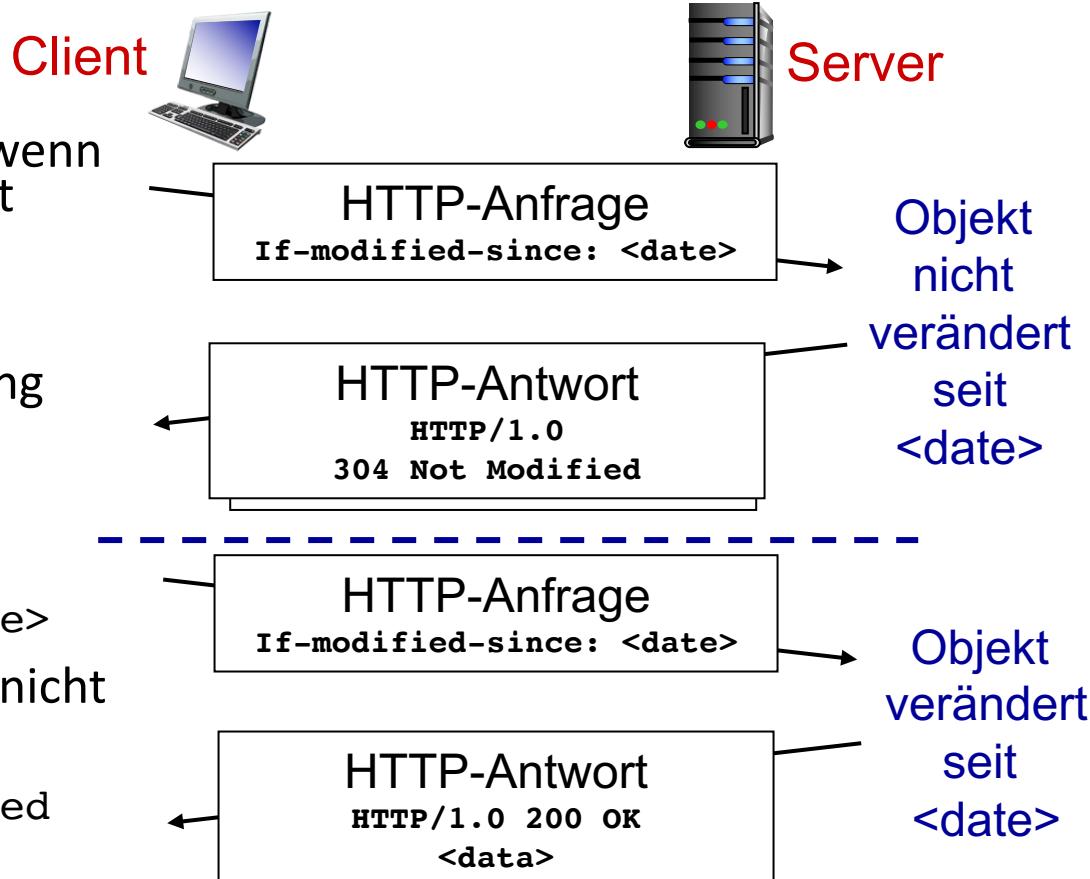
- Reduziert Verzögerung durch Objektübertragung
- Geringere Bandbreitennutzung

Cache gibt Zeitstempel der gespeicherten Kopie in HTTP-Anfrage an:

If-modified-since: <date>

Server liefert kein Objekt, falls nicht modifiziert

HTTP/1.0 304 Not Modified



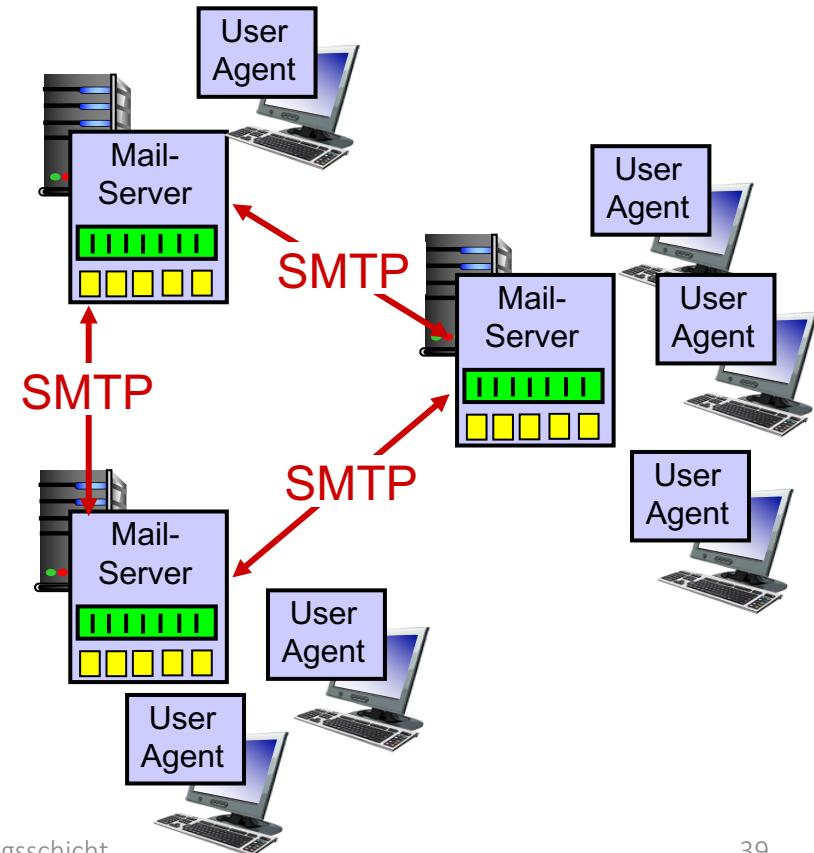
# Electronic Mail (Email)

## Drei Komponenten:

- User Agent (UA)
- Mail-Server
- Simple Mail Transfer Protocol (SMTP)

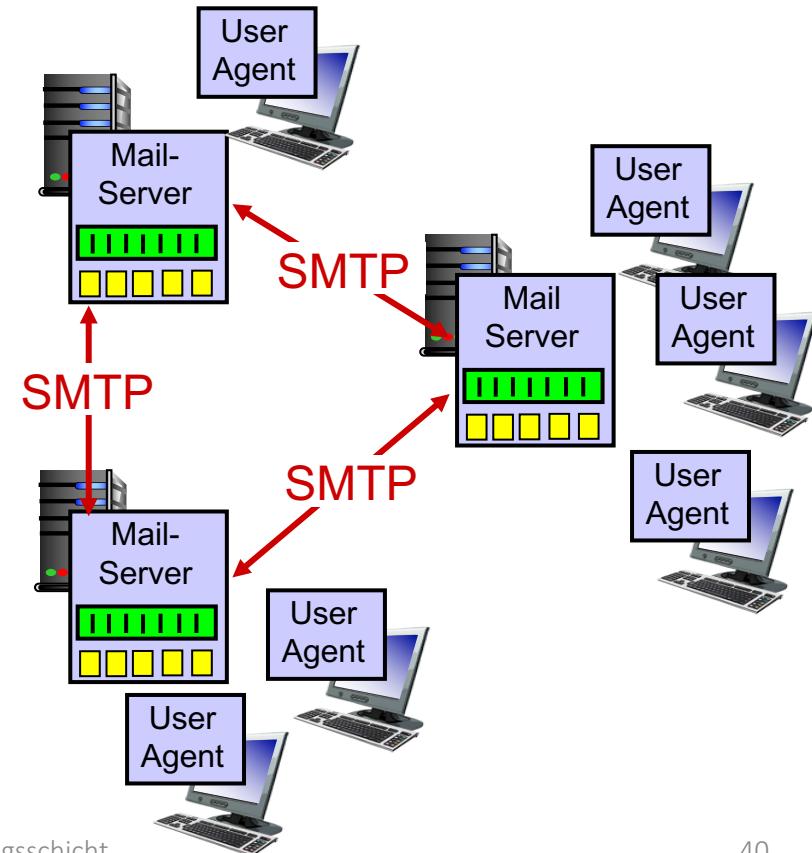
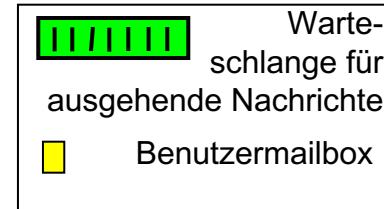
## User Agent

- Auch „Email-Programm“ (Outlook, Thunderbird, iPhone-Client...)
- Erstellen und Lesen von Email-Nachrichten
- Eingehende und ausgehende Nachrichten werden auf Mail-Server gespeichert



# Email: Mail-Server

- **Mailbox** enthält eingehende Nachrichten pro Benutzer
- **Nachrichten-Warteschlange (Message Queue)** mit ausgehenden (zu sendenden) Nachrichten
- SMTP Protokoll zur Nachrichtenübertragung zwischen Mail-Servern
  - Client: Sender Mail-Server
  - Server: Empfangender Mail-Server

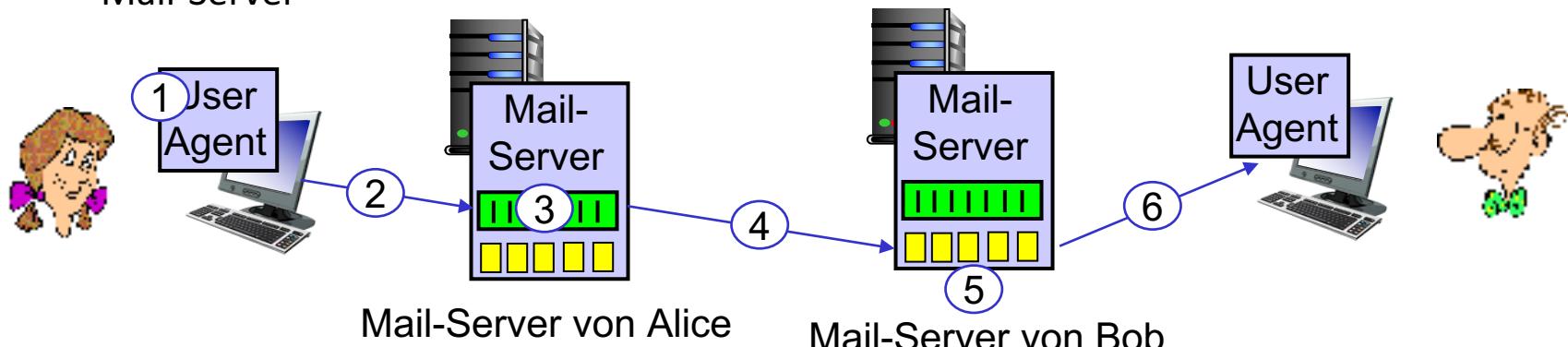


# Email: SMTP [RFC 5321]

- Benutzt TCP für zuverlässige Email-Übertragung zwischen Client und Server, Port 25
- Direkter Transfer: Sender Server zu empfangendem Server
- Drei Phasen
  - Handshake (Begrüßung)
  - Nachrichtenübertragung
  - Schließen der Verbindung
- Dialog aus Kommandos und Antworten (vergl. HTTP)
  - Kommandos: ASCII Text
  - Antworten: Status-Code und Status-Phrase
- Nachrichten sind 7-bit ASCII

# Szenario: Alice sendet Email an Bob

1. Alice erstellt Nachricht an („to“) bob@someschool.edu mit UA
2. UA von Alice sendet Nachricht an ihren Mail-Server; Nachricht wird in Nachrichten-Warteschlange gespeichert
3. SMTP-Client auf Alices Mail-Server öffnet TCP-Verbindung mit Bobs Mail-Server
4. SMTP-Client von Alices Mail-Server sendet Nachricht über die TCP-Verbindung
5. Bobs Mail Server legt Nachricht in Bobs Mailbox ab
6. Bobs nutzt seinen UA um die Nachricht zu lesen



# Beispiel-Ablauf SMTP

```
S: 220 hamburger.edu
C: HELO crepes.fr
S: 250 Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C: How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
```

# ! SMTP zum Ausprobieren

- **telnet servername 25**
- Wenn alles gut geht, antwortet Server mit 220
- Geben Sie HELO, MAIL FROM, RCPT TO, DATA, QUIT Kommandos ein
- ... so senden Sie Emails ohne Mail-Client!

# Anmerkungen zu SMTP

- SMTP nutzt persistente Verbindungen
- SMTP erfordert Nachrichten (Header & Body) in 7-bit ASCII
- SMTP kennzeichnet Nachrichten-Ende mit CRLF . CRLF

## Vergleich mit HTTP

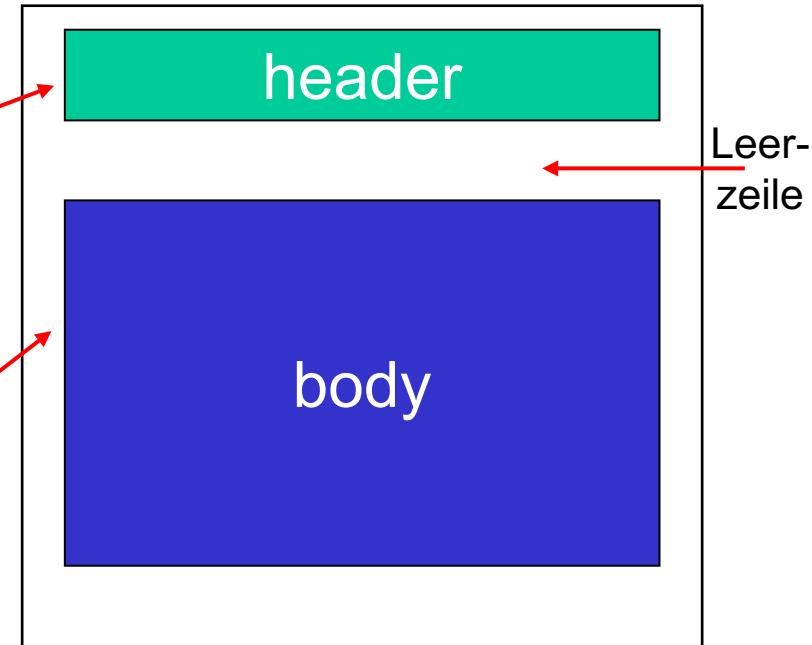
- HTTP: Pull  
SMTP: Push
- Beide Klartext, Kommando / Antwort, Status Codes
- HTTP: Jedes Objekt in eigener Antwort-Nachricht
- SMTP: Viele Objekte in einer einzigen (Multipart) Nachricht

# Email Nachrichtenformat

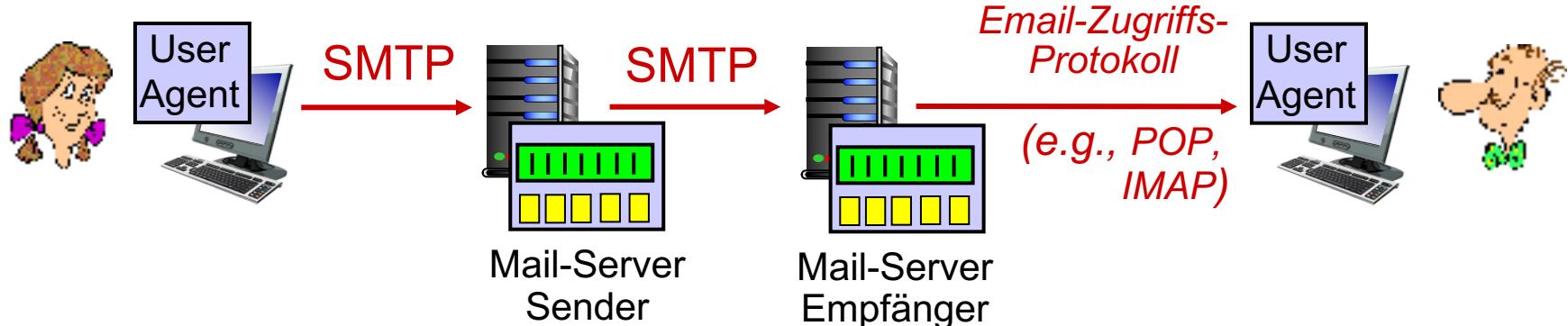
SMTP / RFC 2821: Protokoll für den Austausch von Nachrichten

RFC 822: Format für Textnachrichten

- Kopfzeilen (Header), z.B.
  - To:
  - From:
  - Subject:
- Achtung: Anders als SMTP MAIL  
FROM:, RCPT TO:  
Kommandos!
  - Körper (Body): Die eigentliche Nachricht
  - Nur ASCII Buchstaben



# Email-Zugriffsprotokolle



- SMTP: Auslieferung an dem Mail-Server des Empfängers
- **Email-Zugriffsprotokolle:** Abholen der Nachricht vom Server
  - POP3: Post Office Protocol Version 3 [RFC 1939]: Autorisierung, Download
  - IMAP: Internet Mail Access Protocol [RFC 1730]: Mehr Features, inkl. Bearbeitung der auf Server gespeicherten Nachrichten
  - HTTP: Gmail, Hotmail, GMX, ...

# POP3 Protokoll

## Autorisierungsphase

- Client-Kommandos
  - user: Benutzername angeben
  - Pass: Passwort angeben
- Server-Antworten
  - +OK
  - -ERR

## Transaktionsphase (Client)

- list: Nachrichtennummern auflisten
- retr: Nachricht empfangen anhand von Nummer
- dele: Nachricht löschen
- quit: Verbindung beenden

!

Rufen Sie per Telnet Ihre Emails ab!  
(telnet pop.hs-karlsruhe.de 110)

```
S: +OK POP3 server ready
C: user bob
S: +OK
C: pass hungry
S: +OK user successfully logged on
```

```
C: list
S: 1 498
S: 2 912
S: .
C: retr 1
S: <message 1 contents>
S: .
C: dele 1
C: retr 2
S: <message 2 contents>
S: .
C: dele 2
C: quit
S: +OK POP3 server signing off
```

# Vergleich POP3 und IMAP

## POP3

- Im Beispiel „Download“-Mode
  - Bob kann die Nachricht von anderem Rechner nicht mehr lesen
- POP3 „Download-and-Keep“: Nachrichten bleiben auf dem Server, Kopien auf verschiedenen Clients
- POP3 speichert keinen Zustand zwischen Sitzungen

## IMAP

- Alle Nachrichten an einem Platz: Mail-Server
- Erlaubt die Anordnung in Ordnern
- Speicher Zustand zwischen Sitzungen
  - Ordnernamen, Zuordnung von Nachrichten-IDs zu Ordnern

# Das Domain Name System (DNS)

## [RFC 1034; RFC 1035]

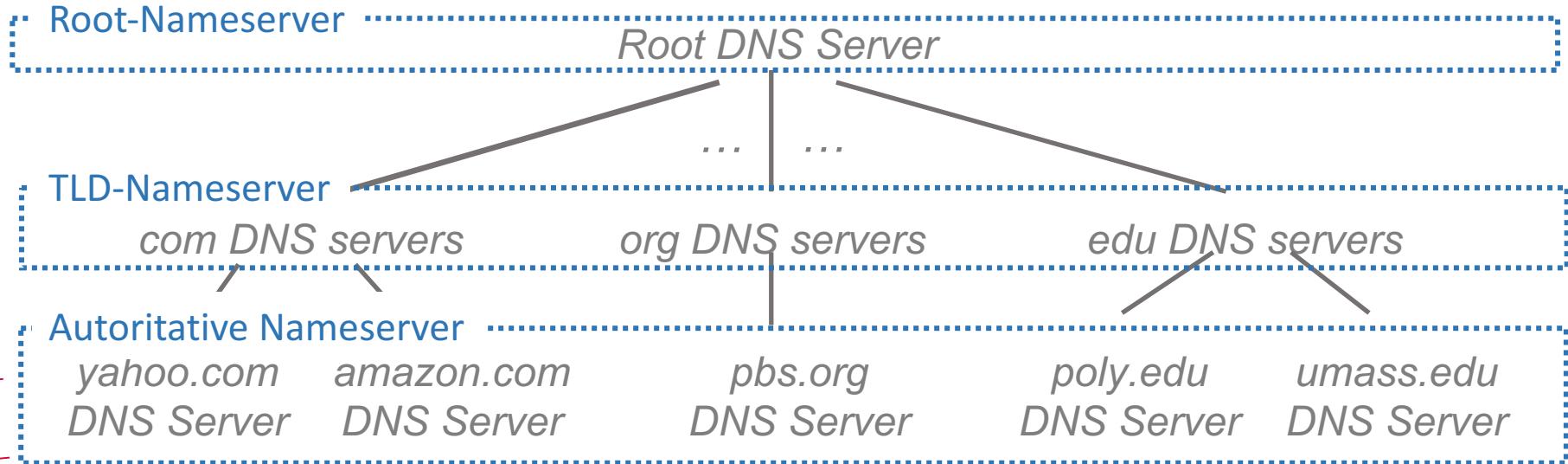
Die Aufgabe: Auflösen von Rechner-Namen (Hostname) auf IP-Adressen

- Verteilte Datenbank implementiert als Hierarchie von vielen **Nameservern**
- Anwendungsschicht-Protokoll: Endsysteme und Nameserver kommunizieren zur Namensauflösung
  - Kernfunktionalität des Internets auf Anwendungsschicht realisiert!
  - Komplexität am Netzrand!

Von DNS bereitgestellte Dienste:

- Übersetzung Host-Name auf IP-Adressen
- Bereitstellung Alias für Host (Host Aliassing)
  - Kanonische und Alias-Namen
- Lastverteilung
  - Replizierte Web Server: Viele IP-Adressen gehören zu einem Hostname

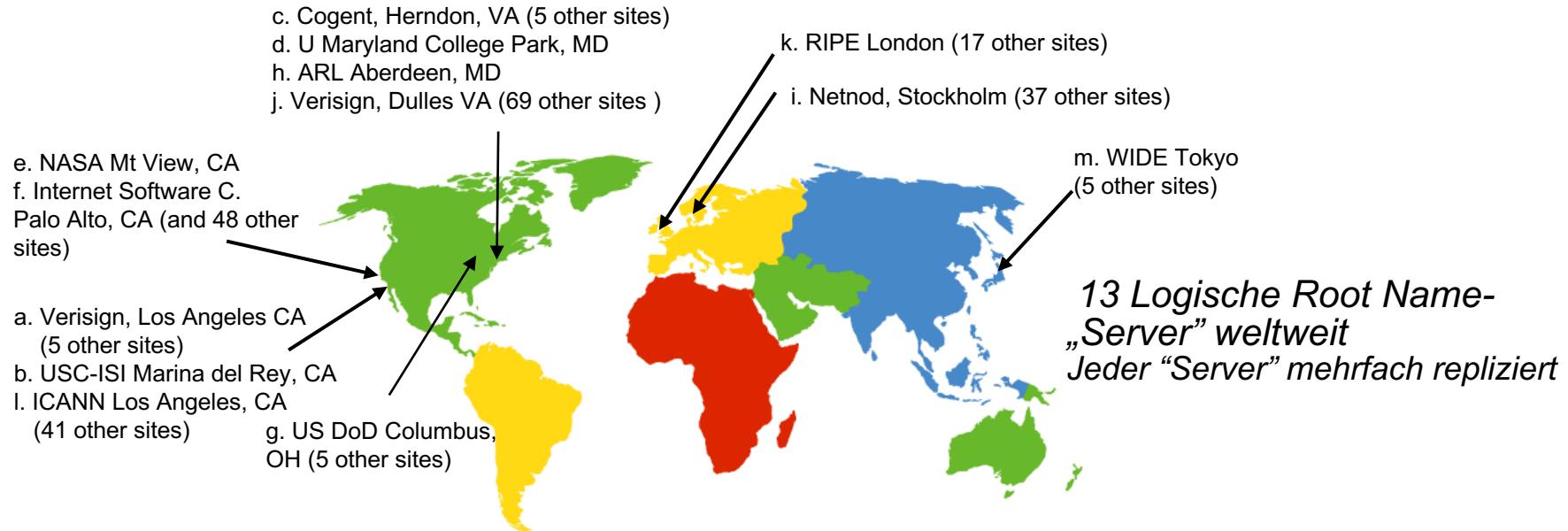
# DNS: Verteilte, hierarchische Datenbank



Client benötigt Adresse für [www.amazon.com](http://www.amazon.com) (1. Näherung)

- Client fragt Wurzel-Server (Root Server) nach “com” DNS-Server
- Client fragt „com“ DNS-Server nach „amazon . com“ DNS-Server
- Client fragt „amazon . com“ DNS-Server nach IP-Adresse von „www . amazon . com“

# DNS: Root Nameserver



Root Nameserver werden von lokalen Nameservern kontaktiert, wenn diese Namen nicht auflösen können

- Kontaktiert **autoritative Nameserver** wenn Namen-IP-Abbildung unbekannt
- Erhält die Auflösung
- Gibt die Auflösung an lokalen Name-Sever weiter

# TLD & Autoritative Nameserver

## Top-Level-Domain (TLD) Nameserver

- Zuständig jeweils für com, org, edu, aero, jobs, museums, ... Und alle Länder-spezifischen Top-Level Domains (z.B. de, uk, fr, ca, ...)
- *DENIC* betreibt Server für .de TLD
- *Network Solutions* für .com TLD
- *Educause* für .edu TLD

## Autoritative Nameserver

- Eigener DNS-Server einer Organisation (ggf. mehrere), stellen autoritative Abbildungen von Host-Namen auf IP-Adressen für alle Hosts einer Organisation bereit
- Können von Organisation selber oder von Service-Provider betrieben werden

# Lokale DNS Nameserver

- Gehört nicht streng zur DNS-Server-Hierarchie
- Jeder ISP (lokaler ISP, Firma, Hochschule) betreibt einen (oder mehrere)
  - Häufig als „Default Nameserver“ bezeichnet
- Wenn Host eine DNS-Anfrage stellt, wird diese zum lokalen DNS-Server gesendet
  - Dieser sucht zunächst in Cache nach aktueller Name-zu-Adresse-Abbildung (die aber veraltet sein kann)
  - Fungiert als Proxy und sendet Anfragen in die DNS-Hierarchie

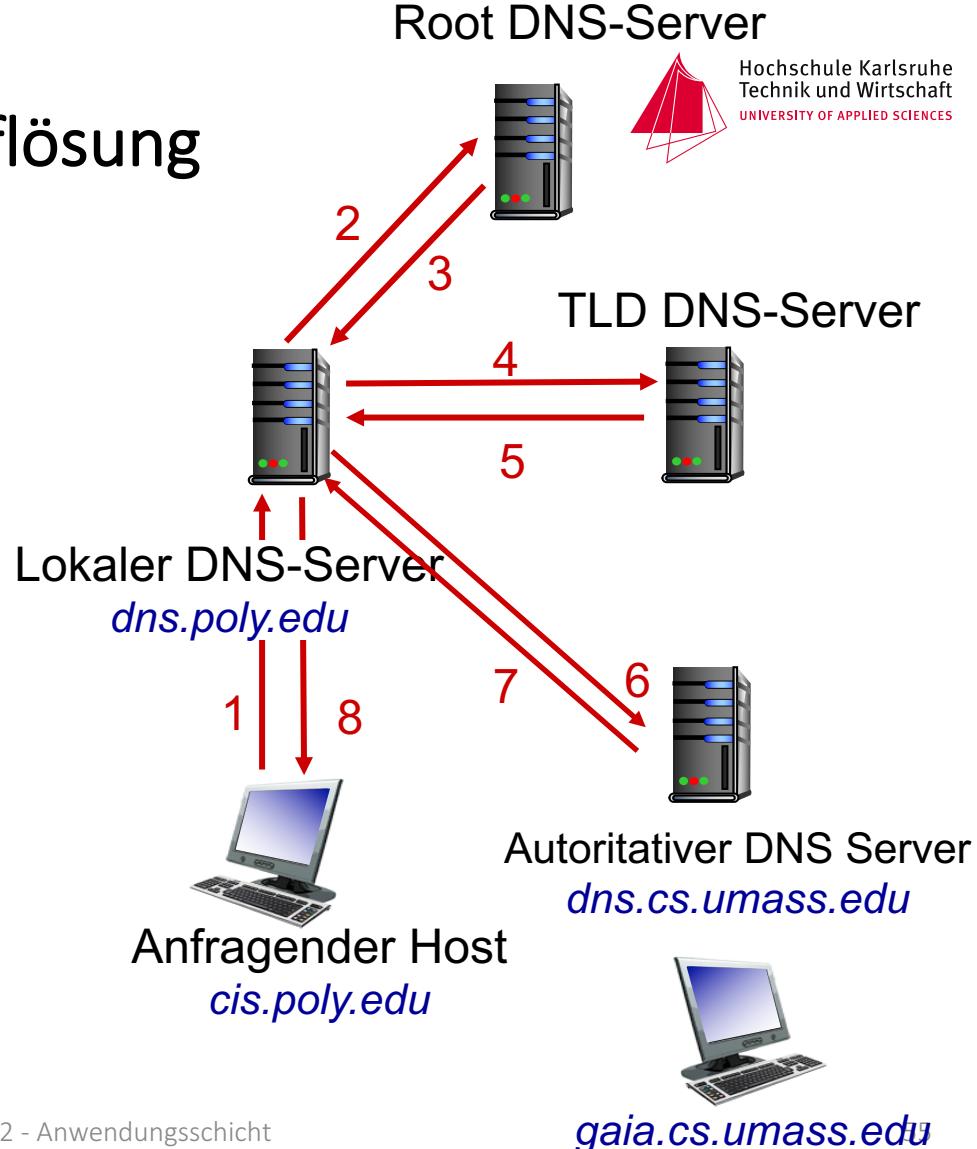


# Beispiel: DNS Namensauflösung

Host at **cis.poly.edu**  
braucht IP-Adresse von  
**gaia.cs.umass.edu**

## Iterative Anfrage:

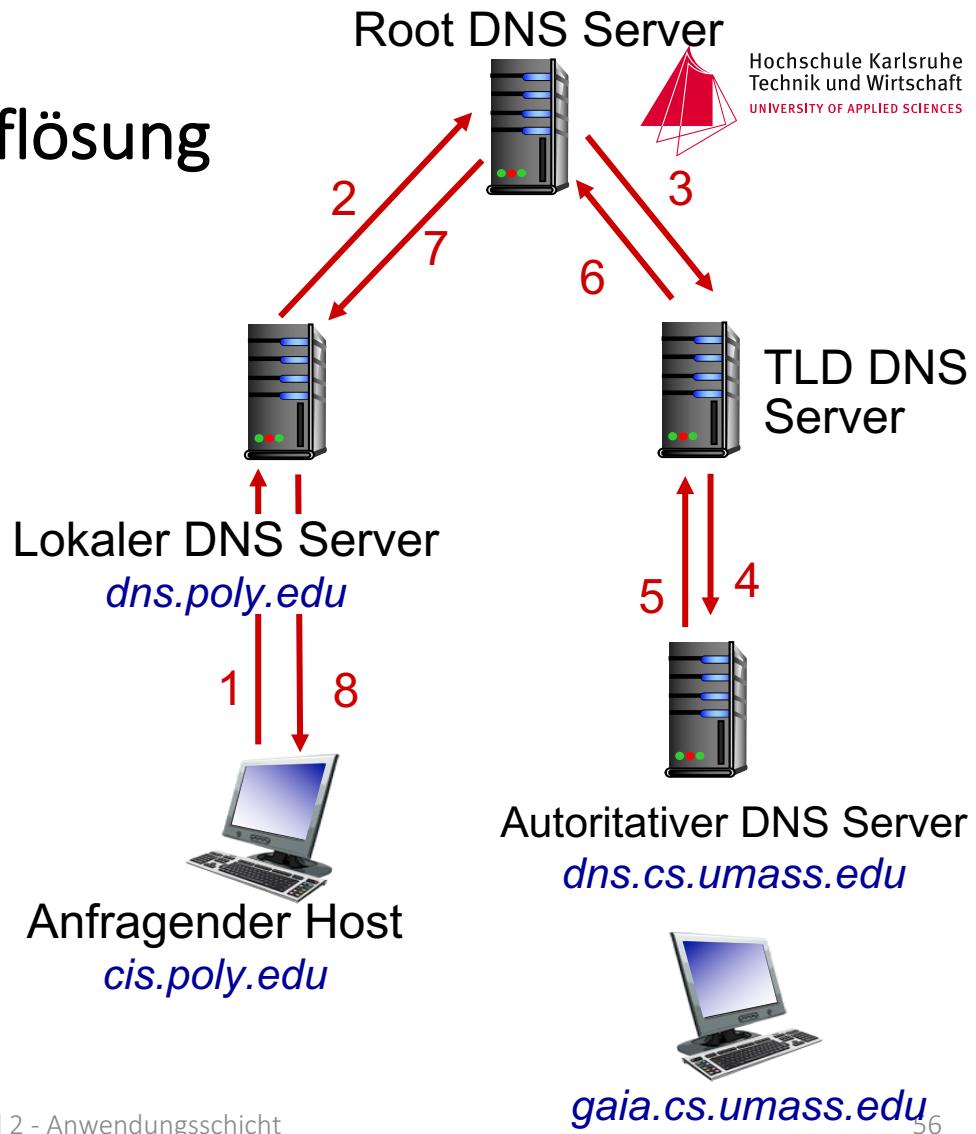
- Der Angefragte Server antwortet mit dem als nächstes zu kontaktierenden Server
- „Ich kenne die Antwort nicht, aber frage diesen Server“



# Beispiel: DNS Namensauflösung

## Rekursive Anfrage:

- Kontaktierter Nameserver muss komplett Namensauflösung übernehmen
- Höhere Last auf oberen Ebenen der Hierarchie?



# ! Übung: DNS Namensauflösung

Wie würde der Ablauf aussehen, wenn der Root DNS-Server nur iterative, alle anderen Name-Server aber rekursive Anfragen unterstützen / verwenden?

# DNS: Caching und Aktualisierung

Immer wenn (irgendein) Nameserver Abbildung lernt, wird diese in Cache gespeichert

- Der Cache-Eintrag hat begrenzte **Lebenszeit (Time-to-Live, TTL)** und wird dann aus dem Cache gelöscht
- TLD-Server sind normalerweise im Cache des lokalen Nameserver gespeichert
  - Daher werden Root-Server nur selten kontaktiert
- Cache-Einträge können veraltet sein - Namensauflösung nach Bestreben (**best-effort**)
  - Wenn ein Host seine IP-Adresse wechselt, kann die neue Adresse im Internet unbekannt sein, bis alte Name-Adress-Abbildung abläuft
- Aktualisierungs-/Benachrichtigungsmechanismus ist IETF Standard [RFC 2136]

# DNS-Einträge

DNS: Verteilte Datenbank, die **Ressource-Einträge (Resource-Records, RR)** speichert:

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

## **type=A**

- name ist ein Host-Name
- value ist eine IP-Adresse

## **type=CNAME**

- name ist Alias für den „richtigen“ kanonischen (canonical) Namen

## **type=NS**

- name ist eine Domain (z.B., foo.com)
- value ist Host-name des autoritativen Nameservers für Domain

## **type=MX**

- value ist der Name des mit name assoziierten Mail-Servers

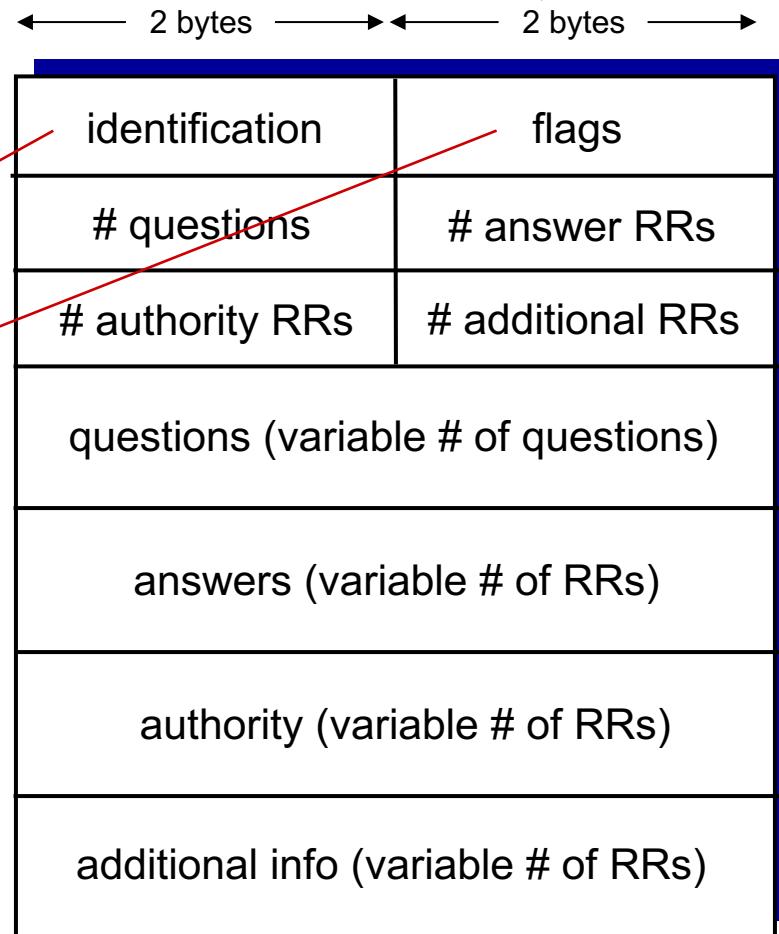
# DNS Protokollnachrichten

Anfrage und Antwort nutzen das selbe Nachrichtenformat

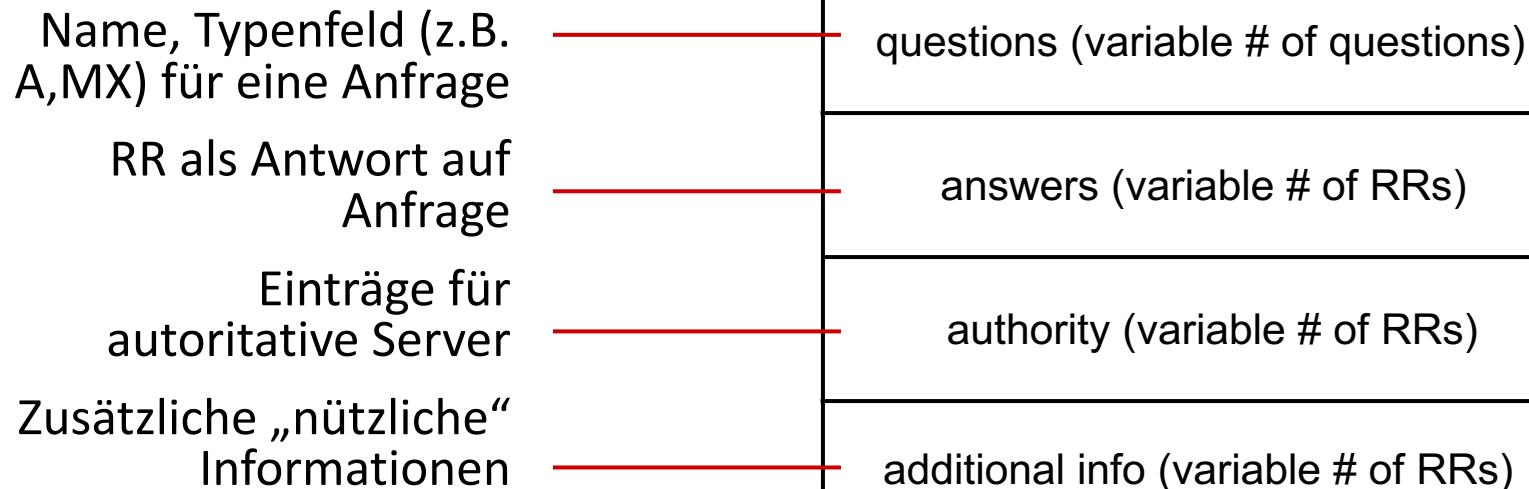
- Übertragung per UDP oder TCP, jeweils Port 53

Kopfzeilen (Header)

- Identifikation: 16 Bit Zahl für Anfrage, Antworten enthalten die selbe Zahl
- Flags
  - Anfrage oder Antwort
  - Rekursion angestrebt
  - Rekursion verfügbar
  - Antwort ist autoritativ



# DNS Protokollnachrichten



# ! DNS zum Ausprobieren

Versenden von DNS-Nachrichten ist möglich mit Programm  
`nslookup`

- `<domainname>` sendet Anfrage (z.B. `www.hs-karlsruhe.de`)
- `server <server-ip>` setzt DNS-Server
- `set type=<type>` setzt Anfrage-Typ auf  
`type ∈ {A, NS, MX, ...}`

Wie heißen die Name-Server für die Domain `hs-karlsruhe.de`?

Wie heißt der Mail-Server für die Domain `hs-karlsruhe.de`?

# Einfügen von DNS-Einträgen

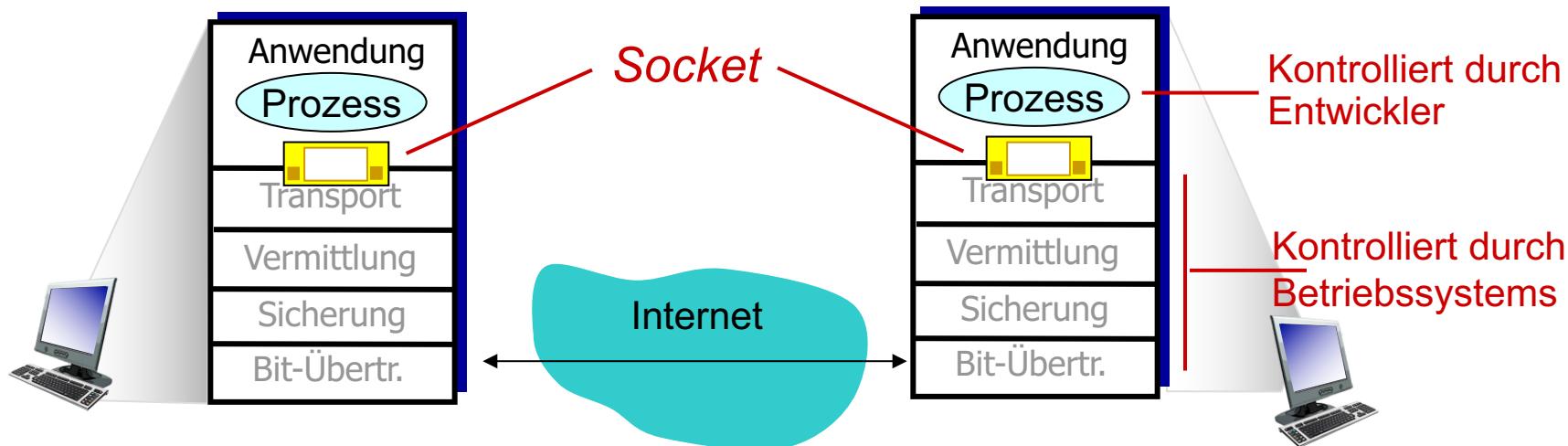
- Beispiel: Neues Startup „Network Utopia“
- Registrierung von neuem Namen **networkutopia.com** bei DNS Registrar (meist Internetdienstleister)
  - Bereitstellen von Name und IP-Adressen von autoritativen Nameservern (Primär und Sekundär)
  - Registrar trägt pro Nameserver zwei RRs in .com TLD Server ein:  
(**networkutopia.com**, **dns1.networkutopia.com**, NS)  
(**dns1.networkutopia.com**, **212.212.212.1**, A)
  - Anlegen von Type A Einträgen auf autoritativen Nameservern z.B. für **www.networkutopia.com**; Typ MX Eintrag für **networkutopia.com**

# Socket-Programmierung

Ziel: Kennenlernen des Aufbaus einer Client-/Server-Anwendung, die über Sockets kommuniziert

- Hier betrachtet: TCP-Sockets!

Socket: „Briefkasten“ zwischen Anwendungsprozessen und Ende-zu-Ende-Transportprotokoll



# Kommunikation zwischen Prozessen

Zur Erinnerung: Kommunikation zwischen Prozessen (Interprozesskommunikation) folgt in der Regel einem **Client-Server-Modell**

**Client** und **Server** sind Rollen der Prozesse

- Server-Prozess bietet Dienst an
- Client-Prozess verbindet sich mit Server-Prozess um Dienst in Anspruch zu nehmen



# Analogie: Telefonanruf

Eine (Client-)Person ruft andere (Server-)Person per Telefon an



- Client muss Server kennen (insbesondere Adresse)
- Server braucht vor Verbindungsaufbau Client nicht zu kennen
- Nach Verbindungsaufbau können sowohl Client als auch Server Informationen senden und Empfangen

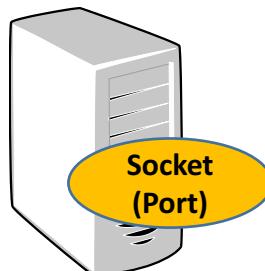
# Grundlegendes Konzept: Socket

**Socket** ist Endpunkt einer bidirektionalen Kommunikationsverbindung zwischen zwei Prozessen

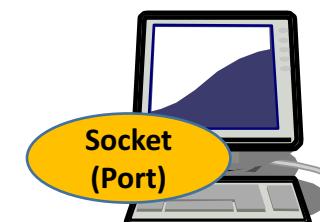
- Annahme: Client- und Server-Prozess auf verschiedenen Rechnern im Internet

Socket ist an **Port**-Nummer gebunden, damit Transportschicht (TCP) Daten für Socket identifizieren kann → vgl. Kapitel 3

Sowohl Client als auch Server besitzen eigenes Socket



Server IP-Adresse



Client IP-Adresse

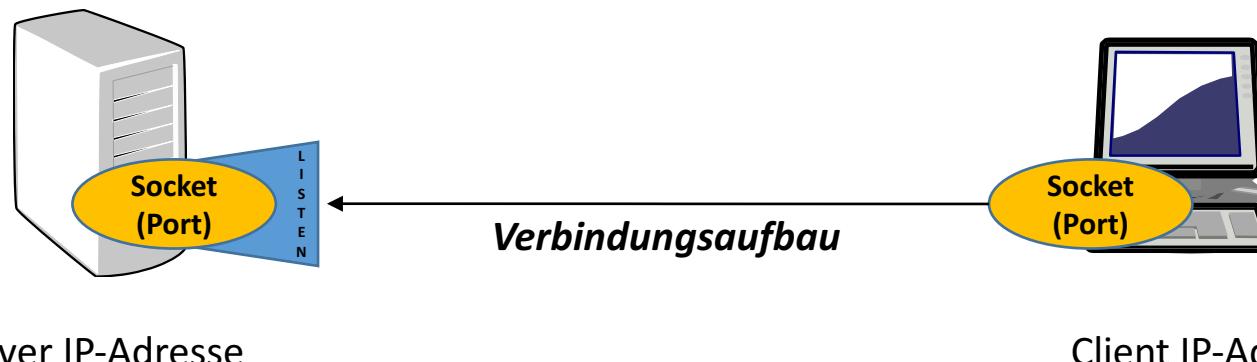
# Verbindungsauftbau

Server wartet auf Verbindungsauftbau an **Server-Socket**

Client nutzt Server-IP Adresse und Port des Server-Sockets für Verbindungsauftbau

- Z.B. IP-Adresse 193.196.64.99 (www.hs-karlsruhe.de), Port 80

Client identifiziert sich durch Client-IP-Adresse und lokalen Port (in der Regel vom Betriebssystem zugewiesen)



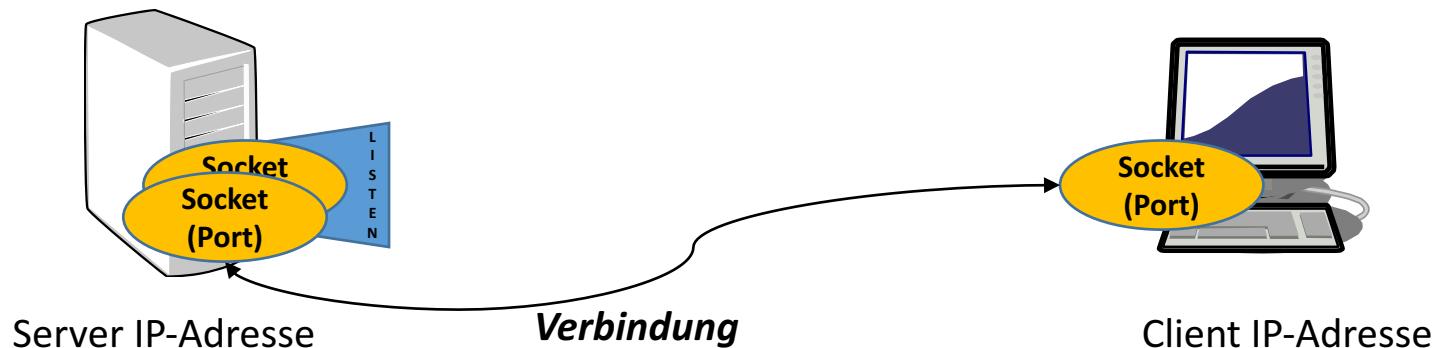
Server IP-Adresse

Client IP-Adresse

# Verbindung

Wenn Server Verbindung **akzeptiert**, wird neues Socket erzeugt

- Notwendig, um mit einem Client zu kommunizieren und gleichzeitig auf Verbindungen von anderen Clients zu hören



# Notwendige Schritte

## Server-Seite

1. Server-Socket erzeugen und an Port binden
2. Auf eingehende Verbindungen warten und neues Socket erzeugen
3. Eingabe-/Ausgabe-Ströme für das neue Socket öffnen
4. Daten gemäß Server-Protokoll senden und Empfangen
5. Ströme und Sockets schließen

## Client-Seite

1. Socket erzeugen und verbinden
2. Eingabe-/Ausgabe-Ströme für das Socket öffnen
3. Daten gemäß Server-Protokoll senden und empfangen
4. Ströme und Sockets schließen

# Beispiel: EchoServer und EchoClient

## Prozess EchoServer

- Hört auf Verbindungen auf spezifischem Port
- Liest Daten von akzeptierter Verbindung und sendet diese über dieselbe Verbindung zurück
- Ende: Eingabeende signalisiert durch Client

## Prozess EchoClient

- Baut Verbindung zu EchoServer mit spezifischer IP-Adresse / spezifischem Port auf
- Liest Text von Konsole und sendet diesen über aufgebaute Verbindung
- Liest Text von aufgebauter Verbindung und gibt diesen auf Konsole aus
- Ende: Eingabeende-Zeichen von Konsole

# Analyse des Client Codes

```
import java.io.*;
import java.net.*;

public class EchoClient {
    public static void main(String[] args) throws IOException {
        if (args.length != 2) {
            System.err.println(
                "Usage: java EchoClient <host name> <port number>");
            System.exit(1);
        }

        String hostName = args[0];
        int portNumber = Integer.parseInt(args[1]);
        ...
        Socket echoSocket = new Socket(hostName, portNumber);
        PrintWriter out =
            new PrintWriter(echoSocket.getOutputStream(), true);
        BufferedReader in =
            new BufferedReader(
                new InputStreamReader(echoSocket.getInputStream()));
        BufferedReader stdIn =
            new BufferedReader(
                new InputStreamReader(System.in))
        ...
        String userInput;
        while ((userInput = stdIn.readLine()) != null) {
            out.println(userInput);
            System.out.println("echo: " + in.readLine());
        }
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host " + hostName);
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for the connection to " +
                hostName);
            System.exit(1);
        }
    }
}
```

Einlesen der Kommandozeilen-Attribute

Socket erzeugen und verbinden

Ein- und Ausgabeströme verbinden

Daten senden und Empfangen

Fehlerbehandlung

# Client: Socket erzeugen und verbinden

Neues Socket Objekt echoSocket wird angelegt

- Konstruktor erwartet Server-Namen `hostName` und Port `portNumber`  
(kommen hier von Kommandozeile)

```
[...]  
Socket echoSocket = new Socket(hostName, portNumber);  
[...]
```

# Client: Ein-/Ausgabeströme öffnen

PrintWriter **out** wird auf Ausgabestrom des Sockets geöffnet  
BufferedReader **in** wird auf Eingabestrom des Sockets geöffnet  
Zusätzlich wird für Konsoleneingabe BufferedReader **stdIn** geöffnet

```
[...]  
  
PrintWriter out =  
        new PrintWriter(echoSocket.getOutputStream() , true);  
  
BufferedReader in =  
        new BufferedReader(  
            new InputStreamReader(echoSocket.getInputStream()));  
  
BufferedReader stdIn =  
        new BufferedReader(  
            new InputStreamReader(System.in));  
[...]
```

# Client: Daten senden und Empfangen

Client wartet auf Benutzereingabe über Objekt `stdIn`  
Benutzereingabe wird über Objekt `out` an Server gesendet  
Serverantwort wird über Objekt `in` von Server gelesen  
• Ausgabe auf Konsole  
while-Schleife terminiert bei Eingabeende (Ctrl+D)

```
[...]  
  
String userInput;  
  
while ((userInput = stdIn.readLine()) != null) {  
    out.println(userInput);  
    System.out.println("echo: " + in.readLine());  
}  
  
[...]
```

# Analyse des Server Codes

```

import java.io.*;
import java.net.*;
import java.io.*;

public class EchoServer {
    public static void main(String[] args) throws IOException {
        if (args.length != 1) {
            System.err.println("Usage: java EchoServer <port number>");
            System.exit(1);
        }

        int portNumber = Integer.parseInt(args[0]);
        try {
            ServerSocket serverSocket =
                new ServerSocket(portNumber);

            Socket clientSocket = serverSocket.accept();

            PrintWriter out =
                new PrintWriter(clientSocket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(
                new InputStreamReader(clientSocket.getInputStream()));

            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                out.println(inputLine);
            }
        } catch (IOException e) {
            System.out.println("Exception caught when trying to listen on port "
                + portNumber + " or listening for a connection");
            System.out.println(e.getMessage());
        }
    }
}

```

Einlesen der Kommandozeilen-Attribute

Server-Socket erzeugen und verbinden

Auf eingehende Verbindungen warten und neues Socket erzeugen

Ein- und Ausgabeströme verbinden

Daten senden und Empfangen

Fehlerbehandlung

# Server: Socket erzeugen und verbinden

Neues ServerSocket Objekt serverSocket wird angelegt

- Konstruktor erwartet Port portNumber (kommt hier von Kommandozeile)

```
[...]  
ServerSocket serverSocket =  
    new ServerSocket(portNumber);  
[...]
```

# Server: Eingehende Verbindungen annehmen und neues Socket erzeugen

Server wartet mittels Methode `accept()` auf eingehende Verbindungen

- Ausführung blockiert, bis Verbindung eingeht

Für eingehende Verbindung wird neues Socket Objekt `clientSocket` erzeugt

```
[...]  
Socket clientSocket = serverSocket.accept();  
[...]
```

# Server: Eingehende Verbindungen akzeptieren und neues Socket erzeugen

Für das neue Socket `clientSocket` werden Ein-/Ausgabeströme verbunden

- ... mit BufferedReader `in` bzw. PrintWriter `out`, vgl. Client-Implementierung

```
[...]  
  
PrintWriter out =  
        new PrintWriter(clientSocket.getOutputStream(), true);  
  
BufferedReader in = new BufferedReader(  
        new InputStreamReader(clientSocket.getInputStream()));  
[...]
```

# Server: Daten Senden und Empfangen

- Server wartet auf eingehende Daten über Objekt `out`
- Eingehende Daten werden über Objekt `in` versendet
- Ende wenn Eingabeende von Client empfangen wird

```
[...]  
  
String inputLine;  
  
while ((inputLine = in.readLine()) != null) {  
    out.println(inputLine);  
  
[...]
```

# Server für mehrere Clients

Vorgestellter Server bearbeitet nur einen einzelnen Client und  
Terminiert dann

In der Praxis meist nutzlos, denn l.d.R. sollen Server viele  
Client-Anfragen parallel beantworten können

Lösung:

- Nach `accept()` neuen Thread erzeugen
  - Neues (Client-)Socket zur Abarbeitung an Thread übergeben
  - Kommunikation in Thread bearbeiten und bei Kommunikationsende Thread terminieren
- Hauptprogramm kann wieder bei `accept()` fortgesetzt werden

# Zusammenfassung Kapitel 2

Wir habe Wissen über Anwendungsprotokolle erworben

- Typischerweise Dialog aus Anfrage-/Antwortnachrichten
  - Client fordert Informationen oder Dienste an
  - Server antwortet mit Daten, Status-Codes
- Nachrichtenformate
  - Header: Beschreibung der Daten
  - Daten: Informationen, Nutzlast (Payload), die übertragen wird

Grundlegende Fragestellungen

- Architektur: Client-Server vs. Peer-to-Peer
- Anforderung an Transportschicht: Zuverlässiger oder unzuverlässiger Transport?
- Komplexität am Netzrand!