Netzwerktechnologien
3 VO

Univ.-Prof. Dr. Helmut Hlavacs
helmut.hlavacs@univie.ac.at

Dr. Ivan Gojmerac
gojmerac@ftw.at

Bachelorstudium Medieninformatik
SS 2012

Kapitel 2 – Anwendungsschicht

2.1 Grundlagen der Netzwerkanwendungen
2.2 Das Web und HTTP
2.3 Dateitransfer: FTP
2.4 E-Mail im Internet
2.5 DNS – der Verzeichnisdienst des Internets
2.6 Peer-to-Peer-Anwendungen
2.7 Socket-Programmierung mit TCP
2.8 Socket-Programmierung mit UDP

2.1 Grundlagen der Netzwerkanwendungen

Entwicklung von Netzanwendungen

Programme, die

- auf mehreren (verschiedenen) Endsystemen laufen
- über das Netzwerk kommunizieren
  - Beispiel: Die Software eines Webservers kommuniziert mit dem Browser

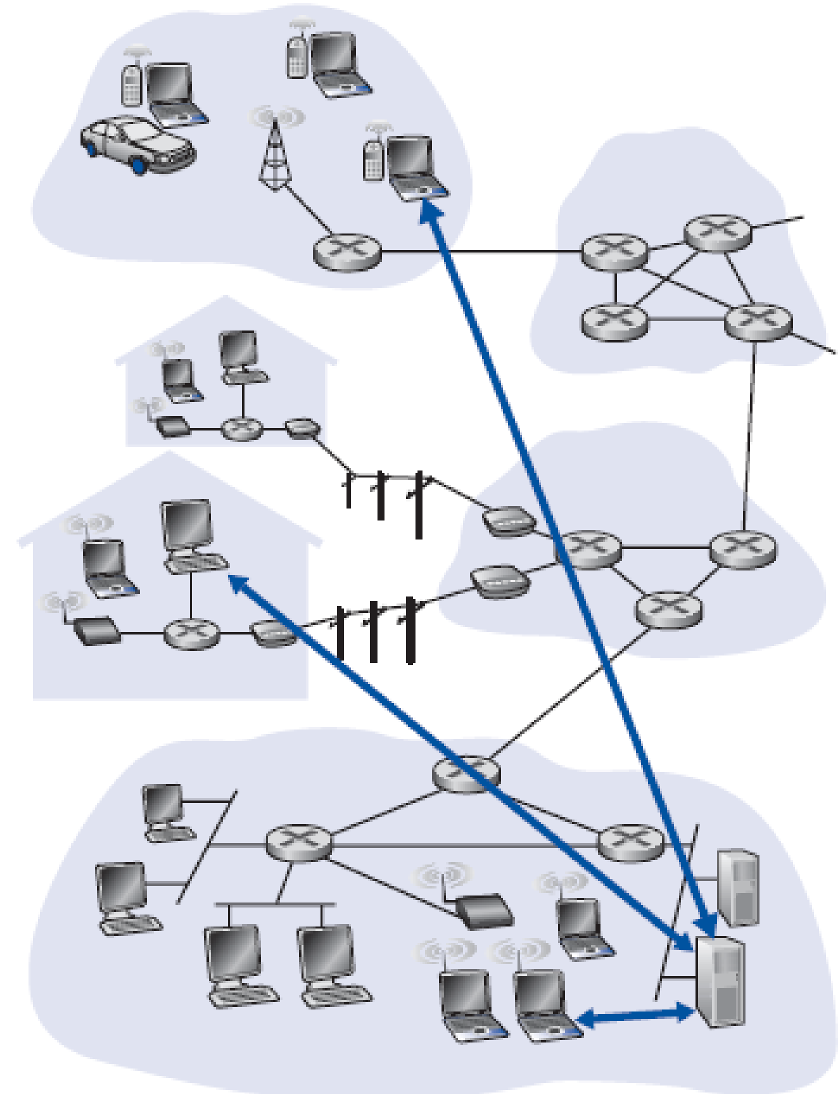
Kaum Software für das Innere des Netzwerkes

- Im Inneren des Netzwerkes werden keine Anwendungen ausgeführt
- Die Konzentration auf Endsysteme erlaubt eine schnelle Entwicklung und Verbreitung der Software

Beispiele für Netzanwendungen:

- E-Mail
- Web
- Instant Messaging
- Terminalfernzugriff
- P2P-Filesharing
- Netzwerkspiele
- Streaming von Videoclips
- Voice over IP (VoIP)
- Videokonferenzen
- Grid Computing

2.1.1 Architektur von Netzwerkanwendungen



Client-Server-Architektur

Server

- Bearbeitet Anfragen von Clients
- Immer eingeschaltet
- Feste IP-Adresse
- Serverfarmen, um zu skalieren

Clients

- Kommunizieren mit Servern
- Permanent oder nur manchmal online
- Können dynamische IP-Adressen haben
- Kommunizieren nicht direkt miteinander

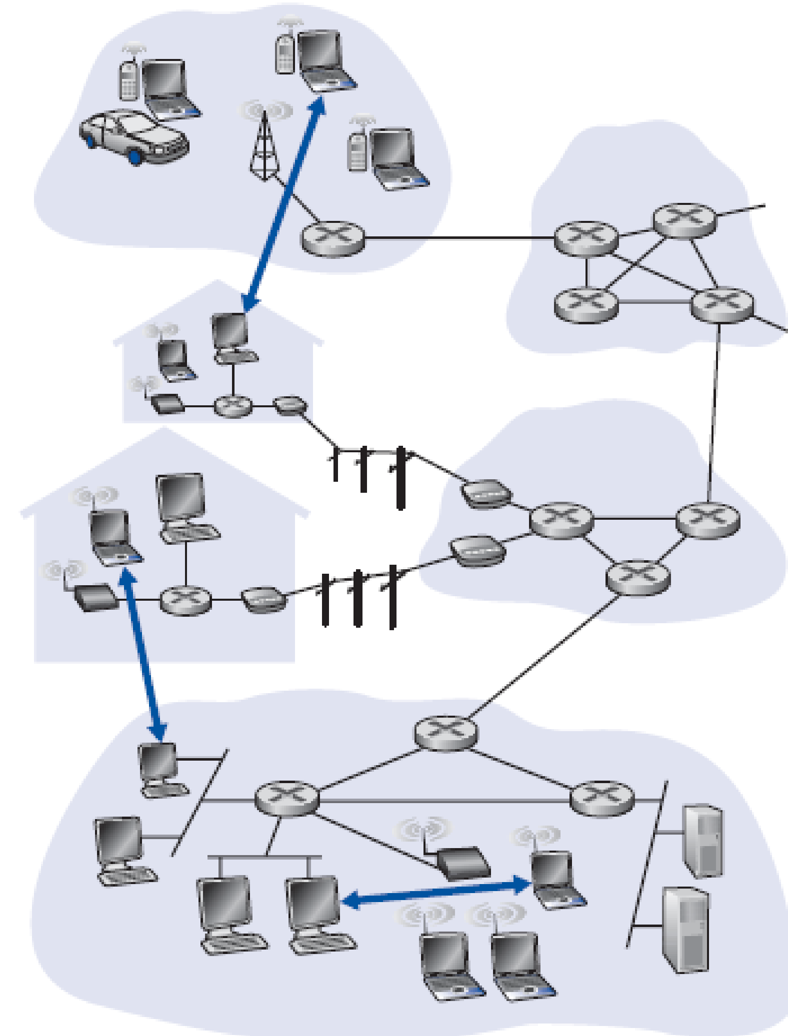
Beispiele für bekannte Anwendungen mit Client-Server-Architektur:

- Das Web
- FTP
- Telnet
- E-Mail

2.1.1 Architektur von Netzwerkanwendungen

Reine Peer-to-Peer-Architektur (P2P)

- **Keine** Server (kostengünstig)
- Beliebige Endsysteme kommunizieren direkt miteinander
- Peers sind nur sporadisch angeschlossen und wechseln ihre IP-Adresse
- Selbstskalierbarkeit (!), aber schwer zu warten und zu kontrollieren!



Beispiele für bekannte Anwendungen mit P2P-Architektur:

- File-Distribution (z.B. BitTorrent)
- Filesharing (z.B. eMule, LimeWire)
- IPTV (z.B. PPLive)

2.1.1 Architektur von Netzwerkanwendungen

Kombination von Client-Server und P2P

Skype - Voice-over-IP Anwendung

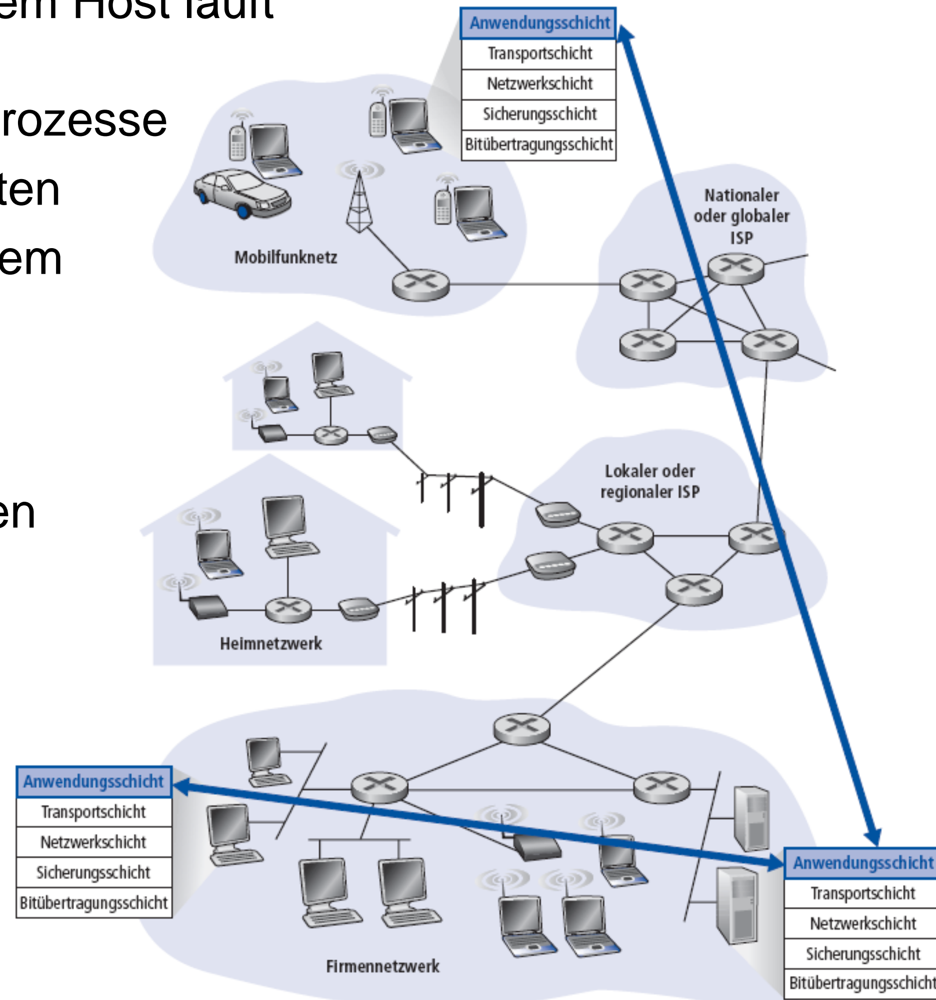
- Zentraler Server: Adresse des Kommunikationspartners finden
- Verbindung zwischen Clients: direkt (P2P)

Instant Messaging

- Chat zwischen zwei Benutzern: P2P
- Zentralisierte Dienste: Erkennen von Anwesenheit, Zustand, Aufenthaltsort eines Anwenders
- Benutzer registriert seine IP-Adresse beim Server, sobald er sich mit dem Netz verbindet
- Benutzer fragt beim Server nach Informationen über seine Freunde und Bekannten

2.1.2 Kommunikation zwischen Prozessen

- Prozess: Programm, welches auf einem Host läuft
- Innerhalb eines Hosts können zwei Prozesse mit Inter-Prozess-Kommunikation Daten austauschen (durch das Betriebssystem unterstützt)
- Prozesse auf verschiedenen Hosts kommunizieren, indem sie Nachrichten über ein Netzwerk austauschen



Die Abbildung veranschaulicht, wie Prozesse miteinander mittels der Anwendungsschicht des fünfschichtigen Internet-Protokollstapels kommunizieren.

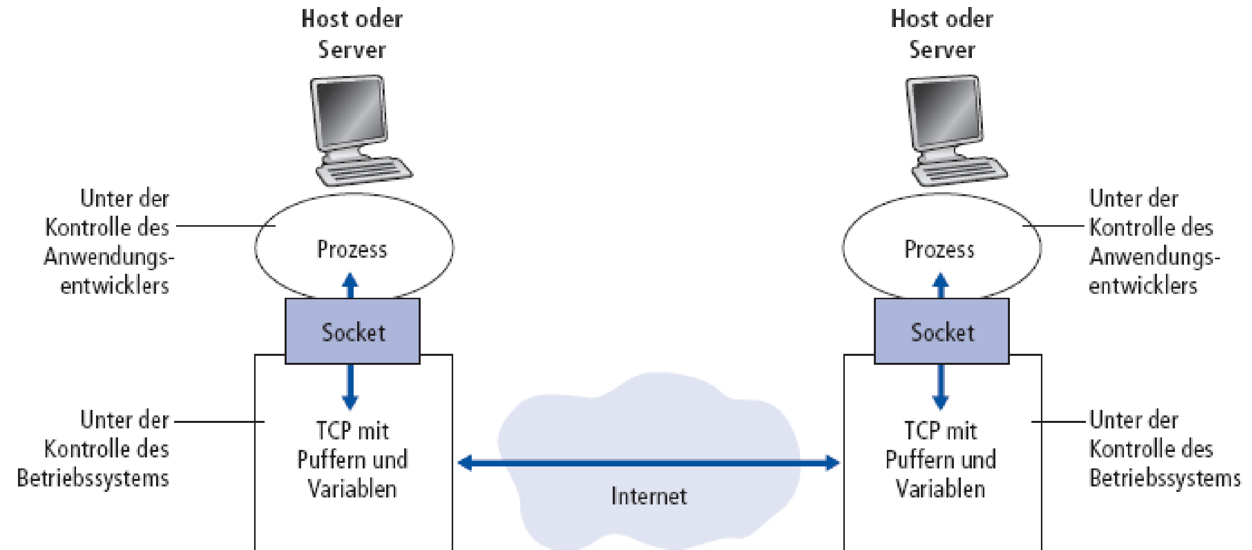
2.1.2 Client- und Server-Prozesse

Eine Netzanwendung besteht aus Prozesspaaren, die einander Nachrichten über ein Netzwerk zusenden:

- Client-Prozess: definiert als Prozess, der die Kommunikation beginnt
- Server-Prozess: definiert als Prozess, der darauf wartet, kontaktiert zu werden

- P2P: Rollen wechseln

2.1.2 Sockets



- Prozesse senden/empfangen Nachrichten über einen **Socket**
- **Schnittstelle** zwischen der Anwendungsschicht und der Transportschicht = Anwendungsprogrammierschnittstelle (API)
- Analogie Tür zu einem Haus (entspricht Socket zu einem Prozess):
  - Der sendende Prozess schiebt die Nachrichten durch seine Tür raus
  - Der sendende Prozess verlässt sich auf die Transportinfrastruktur auf der anderen Seite der Tür, um die Nachricht zum Socket des empfangenden Prozesses zu bringen
  - Sobald die Nachricht beim Zielhost ankommt, tritt sie durch die Tür des empfangenden Prozesses, der danach auf die Nachricht reagiert

2.1.2 Addressierung von Prozessen

- Um eine Nachricht empfangen zu können, muss ein Prozess identifiziert werden können
- Prozesse werden durch die 32Bit lange IP-Adresse des Hosts auf dem sie laufen UND eine Portnummer identifiziert
  - Beispiel-Portnummern:
    - HTTP-Server: 80
    - E-Mail-Server: 25

2.1.2 Anwendungsschicht

Anwendungsprotokolle bestimmen:

- Arten von Nachrichten, Syntax der Nachrichten, Semantik der Nachrichten, Regeln für das Senden von und Antworten auf Nachrichten

Öffentlich verfügbare Protokolle:

- Definiert in RFCs
- Ermöglichen Interoperabilität
- z.B. HTTP, SMTP

Proprietäre Protokolle:

- z.B. Skype

- Brauchen: **Transportprotokoll**

2.1.3 Transportdienste für Anwendungen

Wahl des Transportdienstes nach 3 Kriterien:

- Datenverlust
  - Einige Anwendungen können Datenverlust tolerieren (z.B. Audioübertragungen)
  - Andere Anwendungen benötigen einen absolut zuverlässigen Datentransfer (z.B. Dateitransfer)
- Bandbreite
  - Einige Anwendungen (z.B. Multimedia-Streaming) brauchen eine Mindestbandbreite, um zu funktionieren (in-elastisch, BB-empfindlich)
  - Andere Anwendungen verwenden einfach die verfügbare Bandbreite (bandbreitenelastische Anwendungen)
- Zeitanforderungen
  - Einige Anwendungen (z.B. Internettelefonie oder Netzwerkspiele) tolerieren nur eine sehr geringe Verzögerung
- Sicherheit

2.1.3 Beispiele für Anforderungen von Anwendungen

Anwendung	Datenverlust	Bandbreite	Echtzeit
Dateitransfer	Kein Verlust	Elastisch	Nein
E-Mail	Kein Verlust	Elastisch	Nein
Web	Kein Verlust	Elastisch (wenige Kbps)	Nein
Internettelefonie/ Bildkonferenz	Toleriert Verluste	Audio: wenige Kbps bis 1 Mbps Video: 10 Kbps bis 5 Mbps	Ja: einige Hundert ms
Gespeichertes Audio/Video	Toleriert Verluste	Wie oben	Ja: wenige Sekunden
Interaktive Spiele	Toleriert Verluste	Wenige Kbps bis 10 Kbps	Ja: einige Hundert ms
Instant Messaging	Kein Verlust	Elastisch	Ja und nein

2.1.4 Dienste der Transportprotokolle

TCP-Dienst:

- Verbindungsorientierung: Herstellen einer Verbindung zwischen Client und Server
- Zuverlässiger Transport zwischen sendendem und empfangendem Prozess
- Überlastkontrolle: Bremsen des Senders, wenn das Netzwerk überlastet ist

- Nicht: Zeit- und Bandbreitengarantien, Verschlüsselung

UDP-Dienst:

- Unzuverlässiger Transport von Daten zwischen Sender und Empfänger

- Nicht: Verbindungsorientierung, Zuverlässigkeit, Überlastkontrolle, Zeit- oder Bandbreitengarantien, Verschlüsselung

2.1.4 Beispiele für Anwendungsschicht- und Transportprotokolle im Internet

Anwendung	Anwendungsschichtprotokoll	Zugrunde liegendes Transportprotokoll
E-Mail-Dienst	SMTP [RFC 2821]	TCP
Remote-Terminalzugang	Telnet [RFC 854]	TCP
World Wide Web	HTTP [RFC 2616]	TCP
Dateitransfer	FTP [RFC 959]	TCP
Multimedia-Streaming	HTTP (z. B. YouTube), RTP	TCP oder UDP
Internettelefonie	SIP, RTP oder proprietär (z. B. Skype)	Normalerweise UDP

2.2 Web und HTTP

- Eine Webseite besteht aus Objekten
  - Objekte können sein: HTML-Dateien, JPEG-Bilder, Java-Applets, Audiodateien …
- Eine Webseite hat eine Basis-HTML-Datei, die mehrere referenzierte Objekte beinhalten kann
- Jedes Objekt kann durch eine URL (Uniform Resource Locator) adressiert werden
  - Beispiel für eine URL:

```
www.someschool.edu/someDept/pic.gif
```

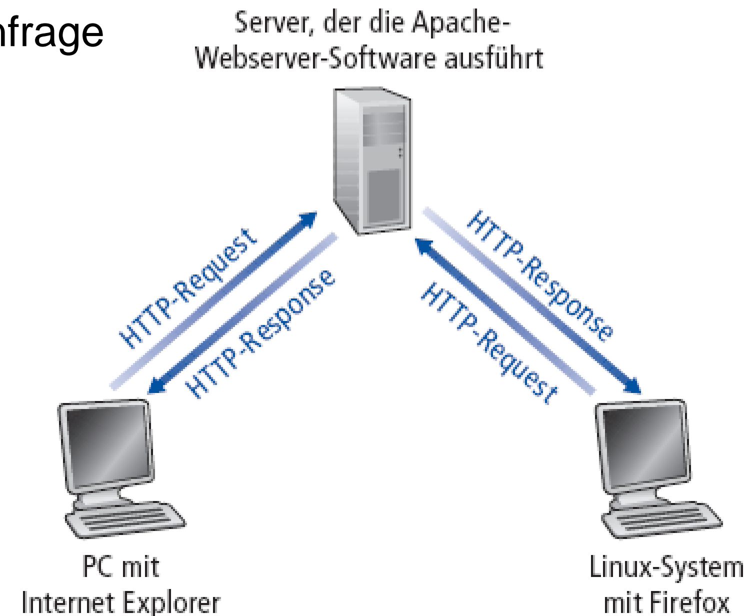
Hostname: www.someschool.edu

Pfad: /someDept/pic.gif

2.2.1 Überblick über HTTP

HTTP (= HyperText Transfer Protocol)

- **Das** Anwendungsprotokoll des Web
- Client/Server-Modell
  - Client: Browser, der Objekte anfragt, erhält und anzeigt
  - Server: Webserver verschickt Objekte auf Anfrage
- Definiert in
  - HTTP 1.0: RFC 1945
  - HTTP 1.1: RFC 2068

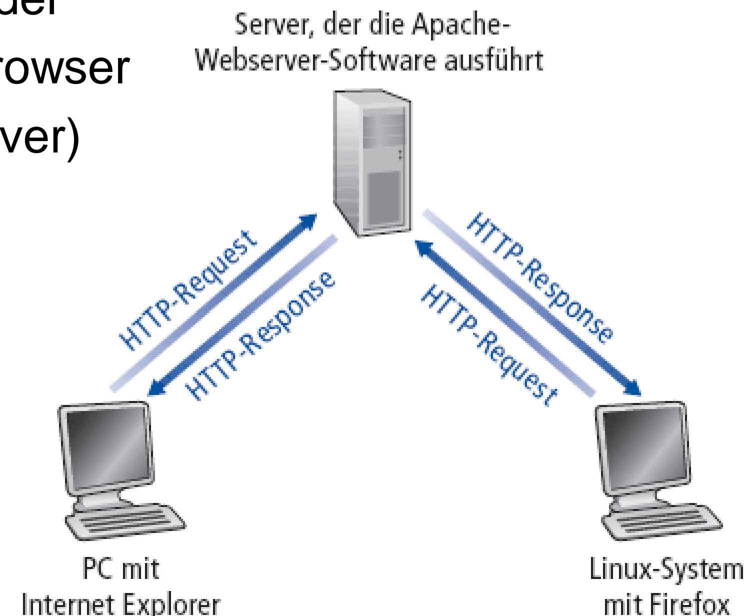


2.2.1 Überblick über HTTP

Verwendet TCP:

1. Client baut mit der Socket-API eine TCP-Verbindung zum Server auf
2. Server wartet auf Port 80
3. Server nimmt die TCP-Verbindung des Clients an
4. HTTP-Nachrichten (Protokollnachrichten der Anwendungsschicht) werden zwischen Browser (HTTP-Client) und Webserver (HTTP-Server) ausgetauscht
5. Die TCP-Verbindung wird geschlossen

→ HTTP ist "zustandslos"
- Server merkt sich keine Informationen über frühere Anfragen von Clients



2.2.2 Nichtpersistente und persistente Verbindungen

Nichtpersistentes HTTP

- Maximal ein Objekt wird über eine TCP-Verbindung übertragen

→ HTTP/1.0 verwendet nichtpersistentes HTTP

Persistentes HTTP

- Mehrere Objekte können über eine TCP-Verbindung übertragen werden

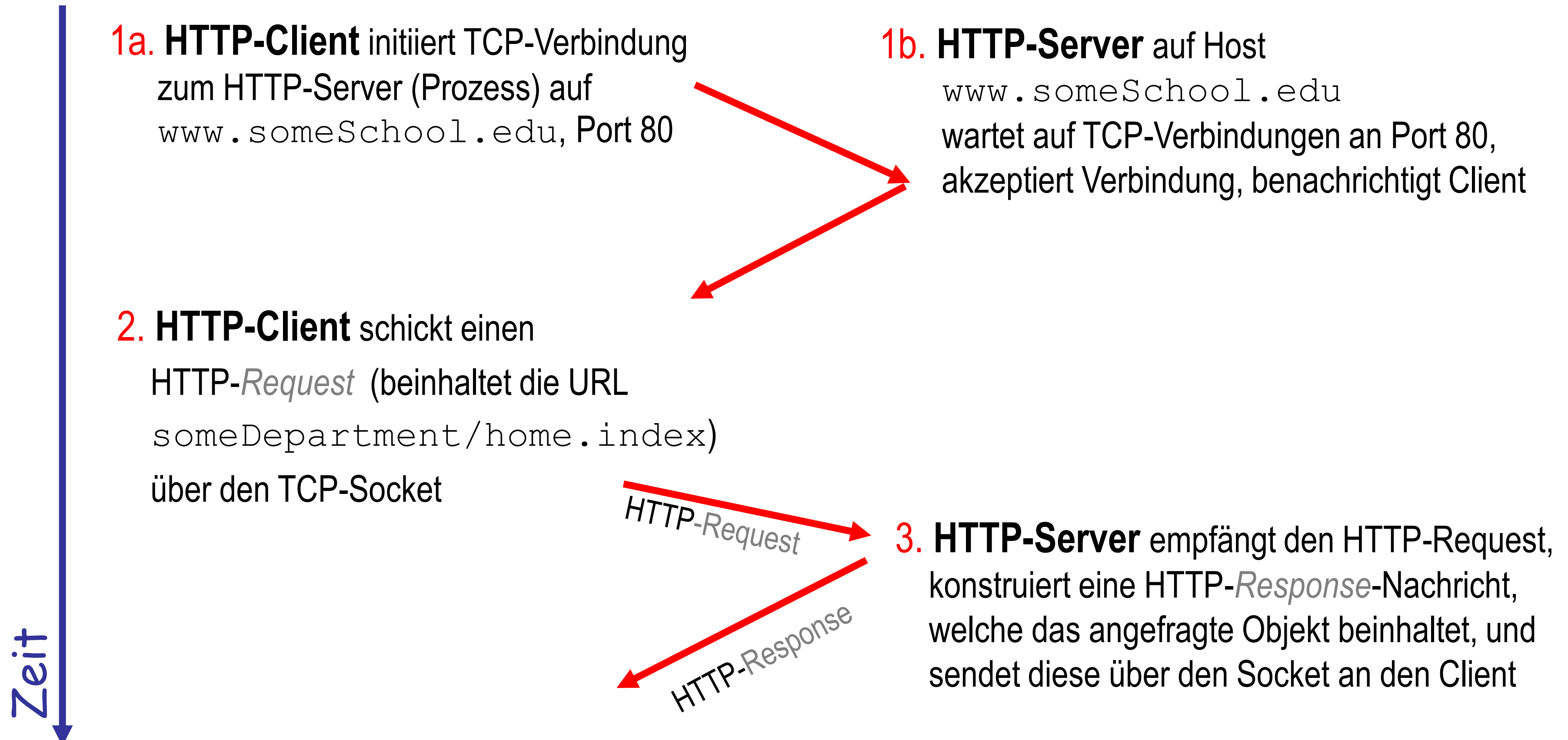
→ HTTP/1.1 verwendet standardmäßig persistentes HTTP

2.2.2 Bsp: Nichtpersistentes HTTP

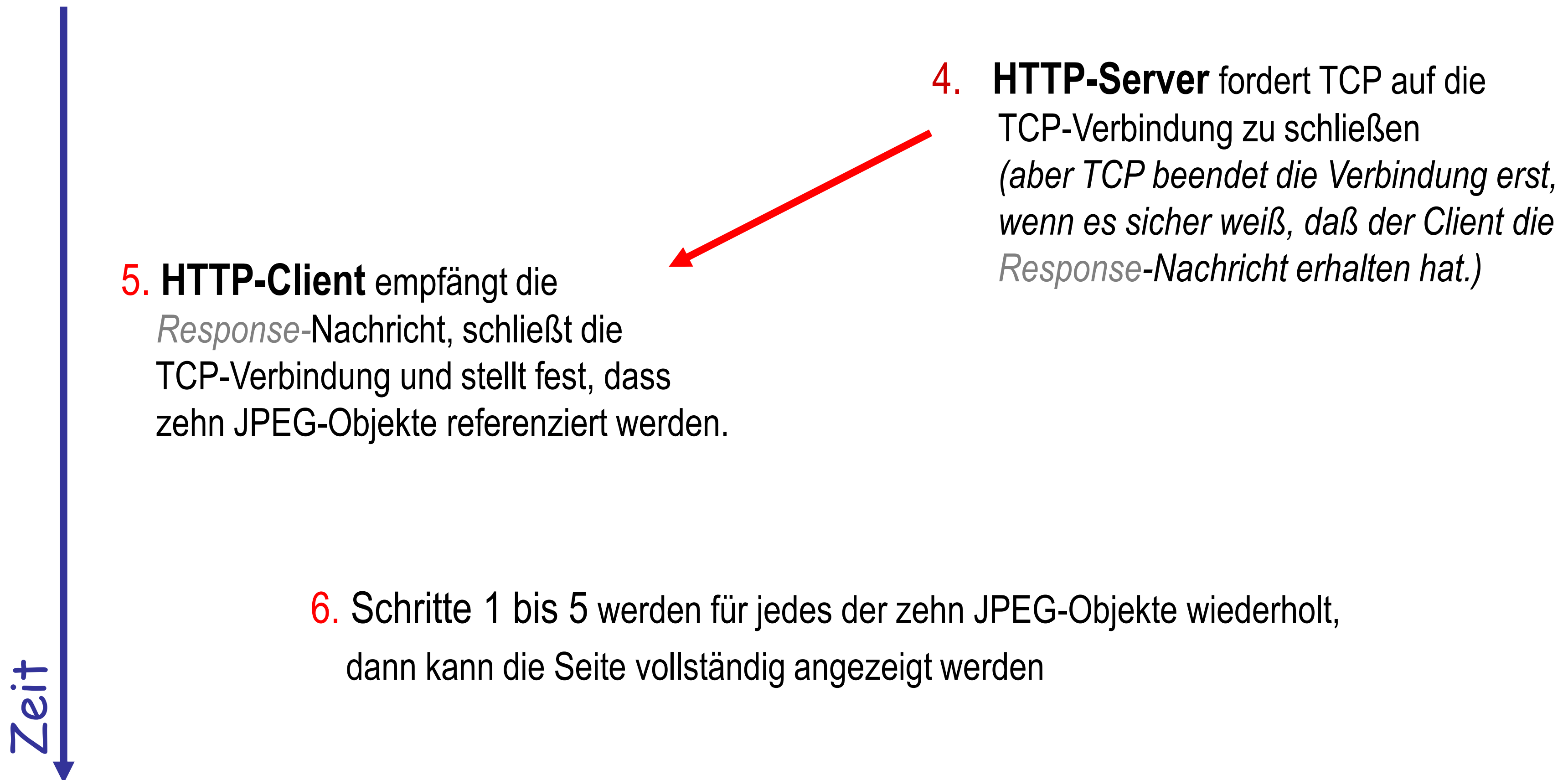
(Die URL beinhaltet Text und Referenzen auf 10 JPEG-Bilder)

Es soll folgende URL geladen werden:

`www.someSchool.edu/someDepartment/home.index`



2.2.2 Nichtpersistentes HTTP

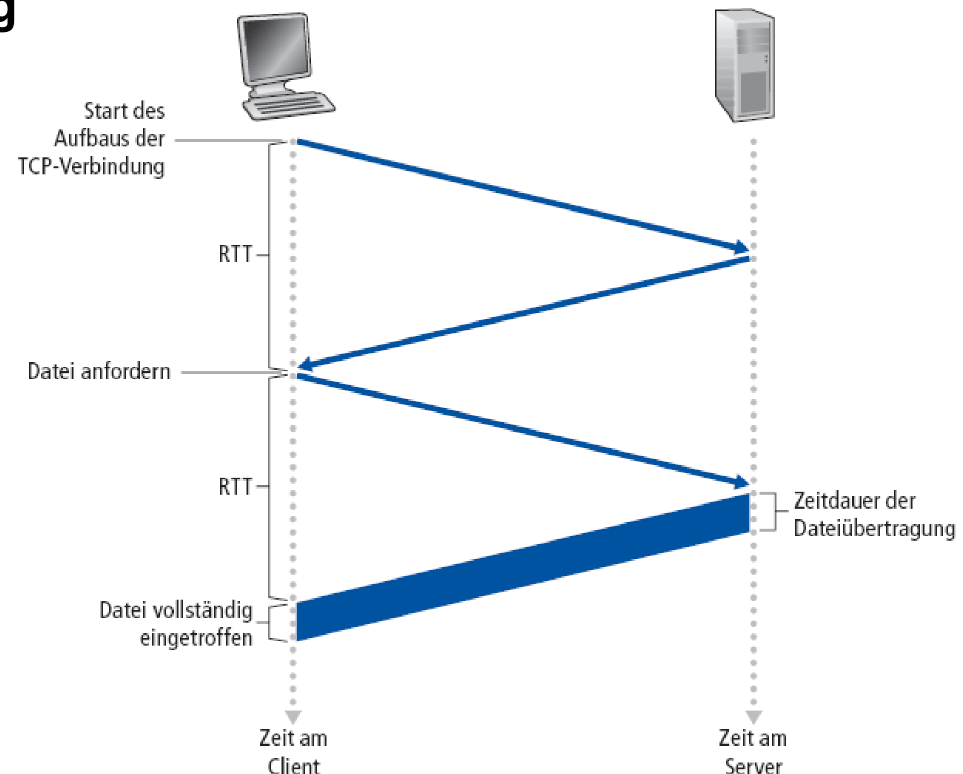


2.2.2 Nichtpersistentes HTTP

Verzögerung

1 RTT	für den TCP-Verbindungsaufbau
+1 RTT	für den HTTP-Request, bis das erste Byte der HTTP-Response beim Client ist
+ Zeit	für das Übertragen der Daten auf der Leitung

= 2 RTT + Übertragungsverzögerung



→ Definition **RTT** (Round Trip Time):
Zeit, um ein kleines Paket vom Client zum Server und zurück zu schicken

2.2.2 Vorteile von persistentem HTTP

Probleme mit nichtpersistentem HTTP:

- 2 RTTs pro Objekt
- Aufwand im Betriebssystem für jede TCP-Verbindung
- Browser öffnen oft mehrere parallele TCP-Verbindungen, um die referenzierten Objekte zu laden

Persistentes HTTP

- Server lässt die Verbindung nach dem Senden der Antwort offen
- Nachfolgende HTTP-Nachrichten können über dieselbe Verbindung übertragen werden

2.2.2 Persistentes HTTP

Persistent ohne Pipelining:

- Client schickt neuen Request erst, nachdem die Antwort auf den vorangegangenen Request empfangen wurde
- 1 RTT für jedes referenzierte Objekt (ca.½ Dauer von nichtpersistentem HTTP)

Persistent mit Pipelining:

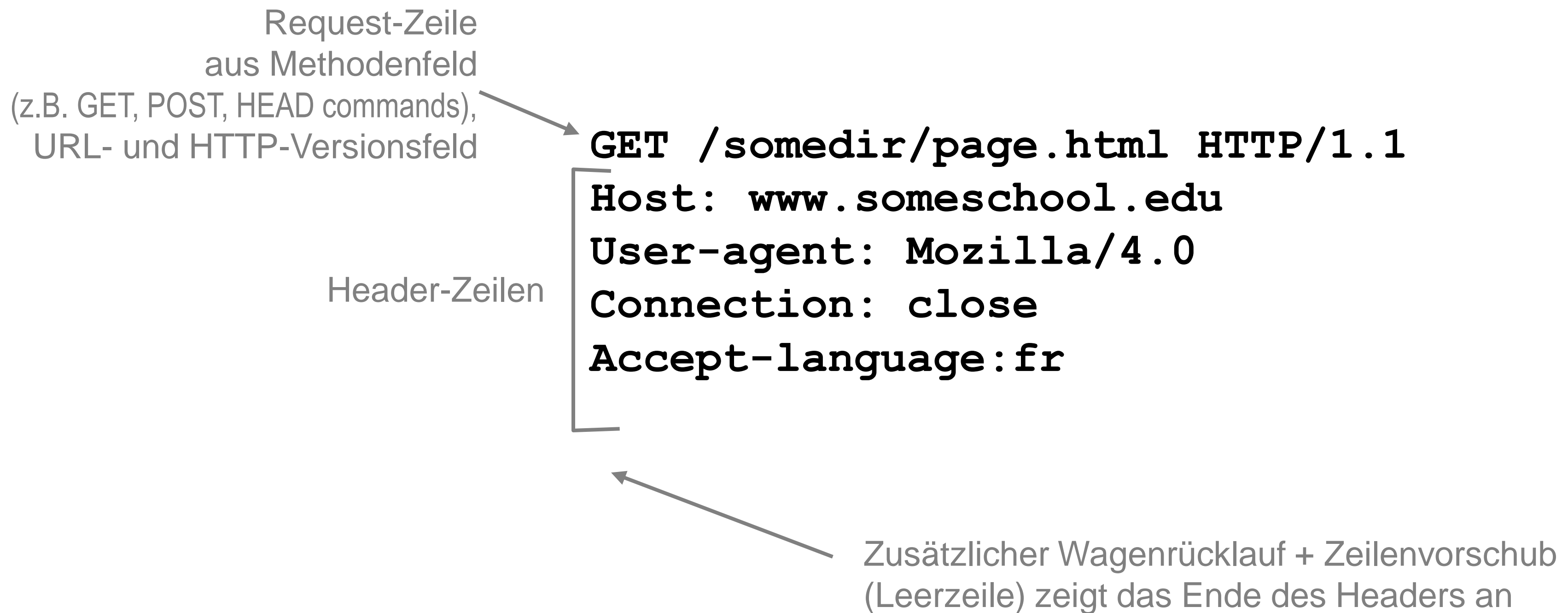
- Standard in HTTP/1.1
- Client schickt Requests, sobald er die Referenz zu einem Objekt findet
- Idealerweise wird nur wenig mehr als 1 RTT für das Laden **aller** referenzierten Objekte benötigt

2.2.3 HTTP-Nachrichtenformat

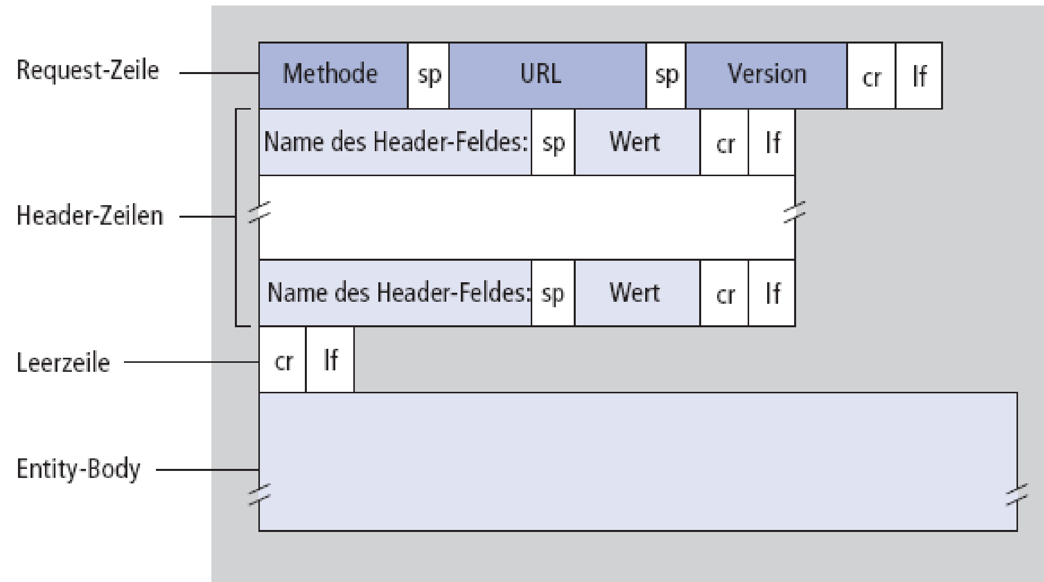
Es gibt zwei Arten von HTTP-Nachrichten: *Request* und *Response*

HTTP-Request-Nachricht

- In ASCII (vom Menschen leicht zu lesen)



2.2.3 Request-Nachricht Format

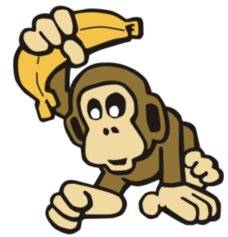


Der Entity-Body wird bei der GET-Methode nicht verwendet (voriges Beispiel), aber bei der POST-Methode um Daten zu versenden.

→ Bsp. Wenn ein Benutzer Suchbegriffe an eine Suchmaschine sendet.

Manche HTML-Formulare verwenden allerdings auch die GET-Methode um eingegebene Daten zu übermitteln indem sie diese in die angeforderte URL schreiben:

→ `www.somesite.com/animalsearch?monkey&banana`



2.2.3 Verfügbare Anweisungen bei HTTP Versionen

HTTP/1.0

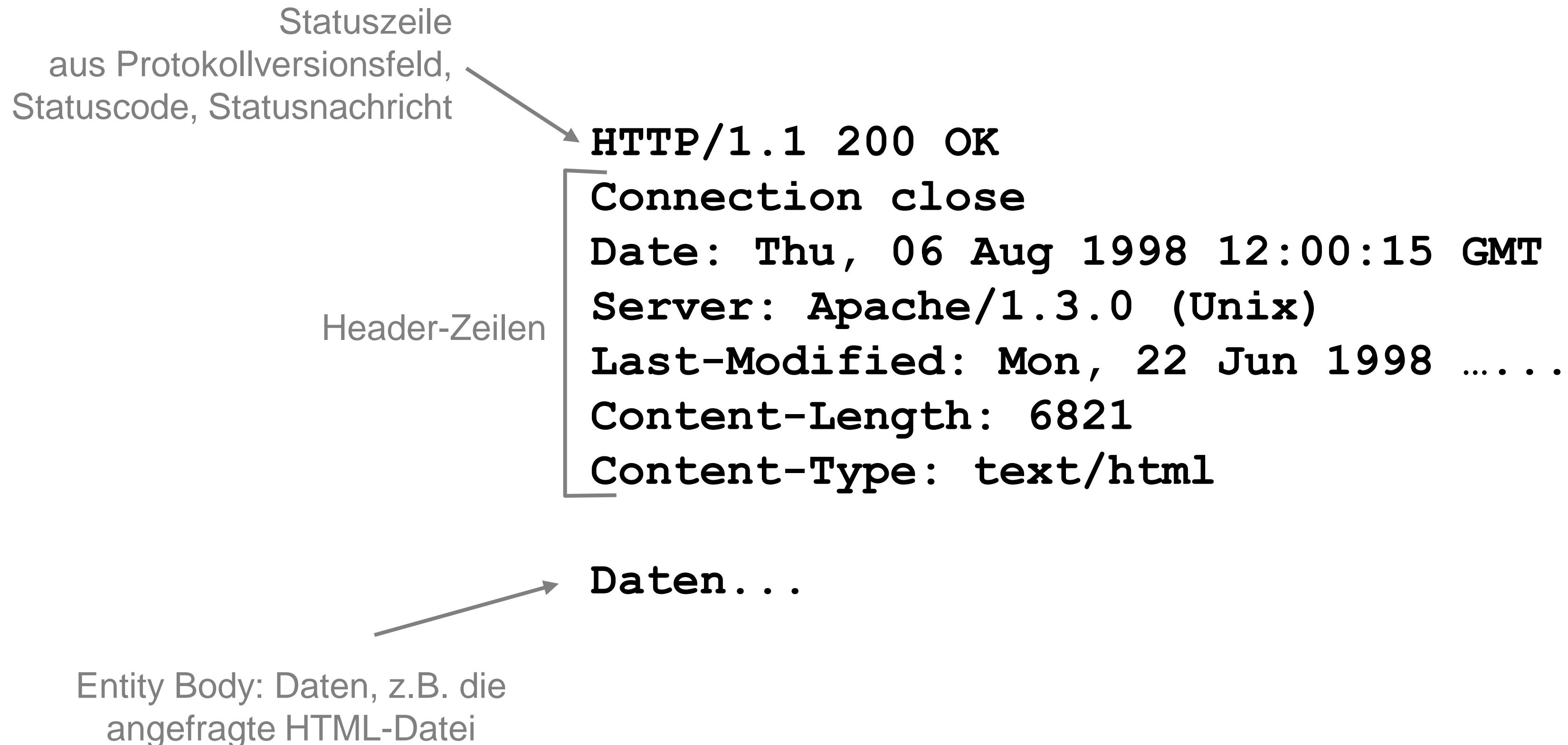
- GET
- POST
- HEAD
  Bittet den Server, nur die Kopfzeilen der Antwort (und nicht das Objekt) zu übertragen

HTTP/1.1

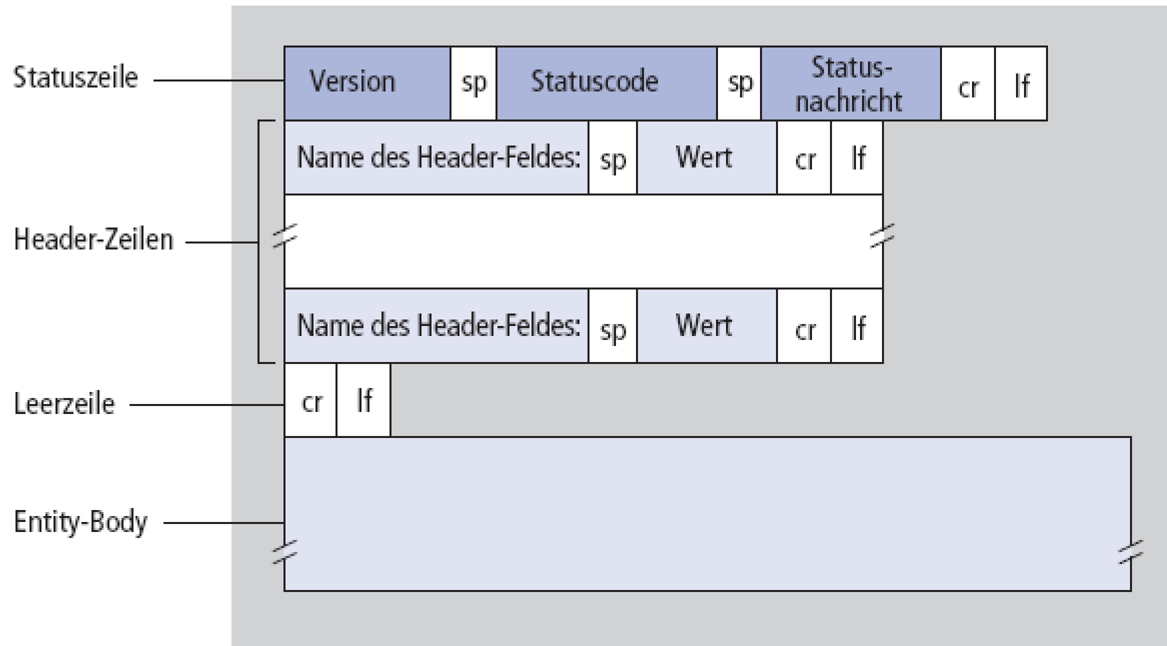
- GET
- POST
- HEAD
- PUT
  Lädt die im Datenteil enthaltene Datei an die durch eine URL bezeichnete Position hoch
- DELETE
  Löscht die durch eine URL angegebene Datei auf dem Server

2.2.3 HTTP-Nachrichtenformat

HTTP-Response-Nachricht



2.2.3 Response-Nachricht Format



Der Entity-Body ist das wichtigste Element der Nachricht - er enthält das (auf der vorherigen Folie als „Daten...“ symbolisierte) angeforderte Objekt.

2.2.3 Response-Nachricht Statuscodes

`200 OK`
- Request war erfolgreich, gewünschtes Objekt ist in der Antwort enthalten

`301 Moved Permanently`
- Gewünschtes Objekt wurde verschoben, neue URL ist in der Antwort enthalten

`400 Bad Request`
- Request-Nachricht wurde vom Server nicht verstanden

`404 Not Found`
- Gewünschtes Objekt wurde nicht gefunden

`505 HTTP Version Not Supported`

2.2.4 Benutzer-Server-Interaktion: Cookies

HTTP ist “zustandslos”

- Server merkt sich keine Informationen über frühere Anfragen von Clients

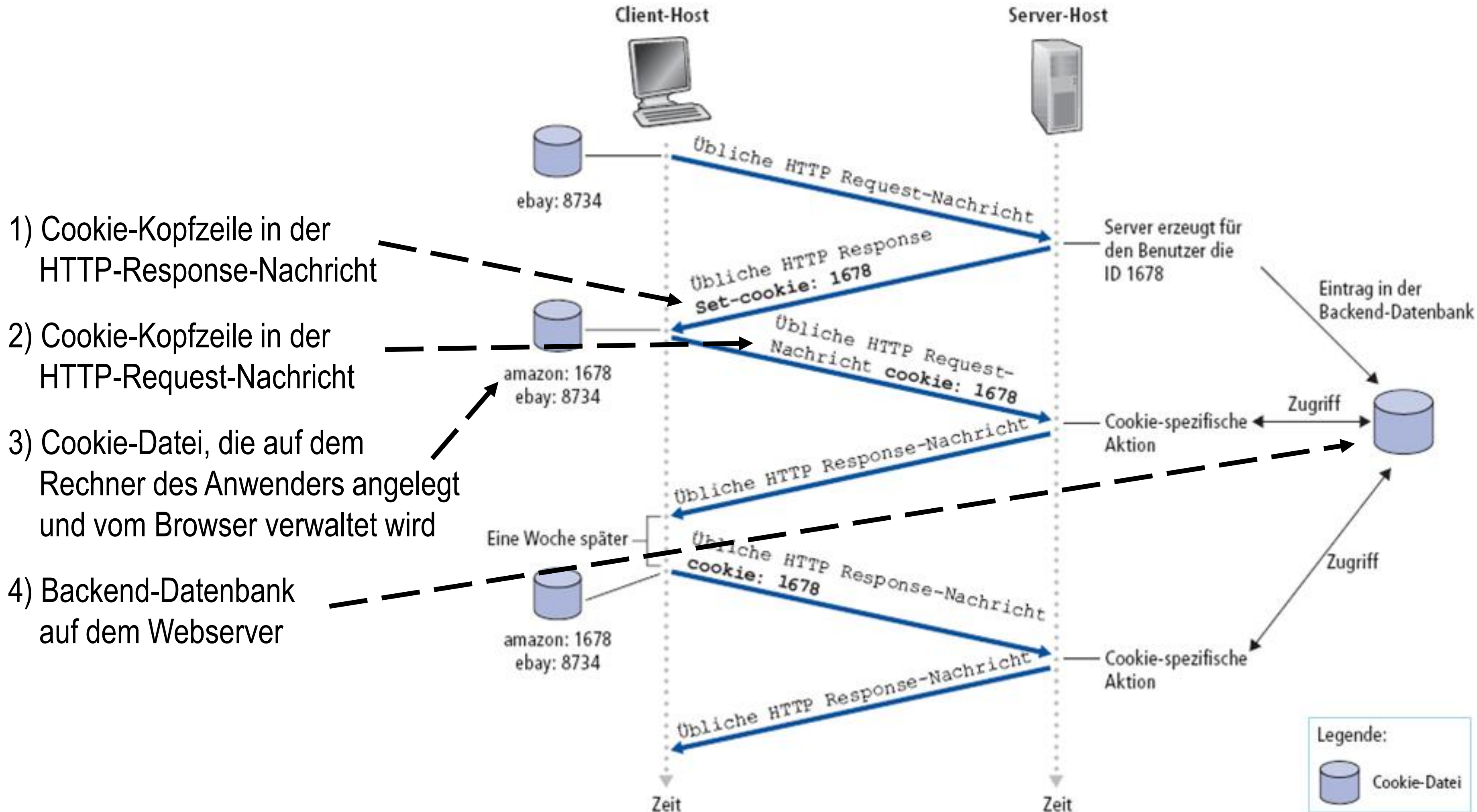
Zum *Merken* benötigt man Cookies:

- Definiert in RFC 2965
- Werden clientseitig gespeichert
- Ermöglichen es Websites Benutzer wiederzuerkennen

Einsatz von Cookies z.B. für:

- Autorisierung
- Einkaufswagen
- Empfehlungen
- Sitzungszustand (z.B. bei Web-E-Mail)

2.2.4 Benutzer-Server-Interaktion: Cookies



2.2.4 Benutzer-Server-Interaktion: Cookies

Cookies und Privatsphäre

Cookies ermöglichen es Websites, viel über den Anwender zu lernen:

- Formulareingaben (Name, E-Mail-Adresse)
- Besuchte Seiten

Alternativen um Zustand zu *merken*

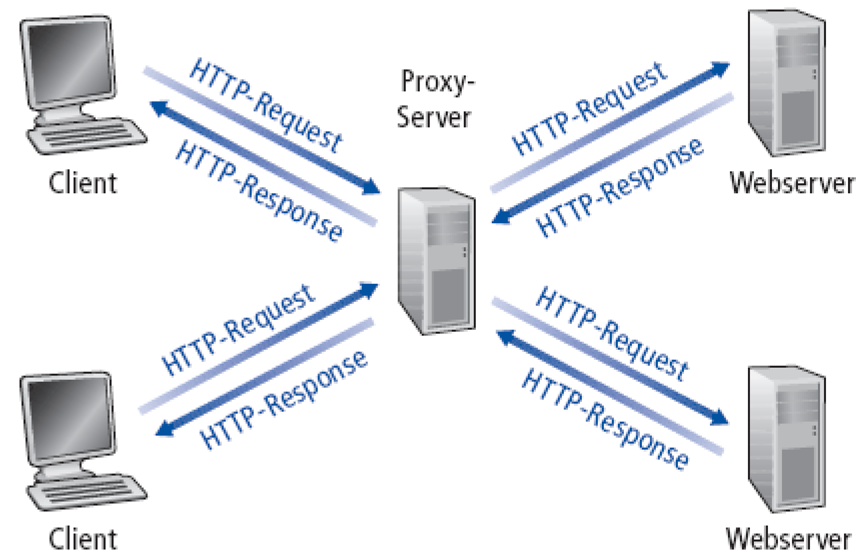
- In den Endsystemen: Zustand wird im Protokoll auf dem Client oder Server gespeichert und für mehrere Transaktionen verwendet
- Cookies: HTTP-Nachrichten beinhalten den Zustand

2.2.5 Webcaching

Webcache (auch Proxyserver genannt)

- Netzwerkentität, die im Namen des eigentlichen Webservers HTTP-Requests beantwortet
- Hat seinen eigenen Plattenspeicher in dem er Kopien der vor kurzem angeforderten Objekte aufbewahrt

1. Benutzer konfiguriert Browser so, daß HTTP-Requests zuerst an den Webcache gerichtet werden
2. Browser stellt eine TCP-Verbindung zum Webcache her und sendet einen HTTP-Request für das gewünschte Objekt
3. Webcache überprüft ob Objekt im Cache:
    - *Falls vorhanden*: Cache gibt Objekt in einer HTTP-Response-Nachricht an Client-Browser zurück
    - *Sonst*: Webcache öffnet TCP-Verbindung zum eigentlichen Server, fragt das Objekt an und leitet es dann zum Client weiter



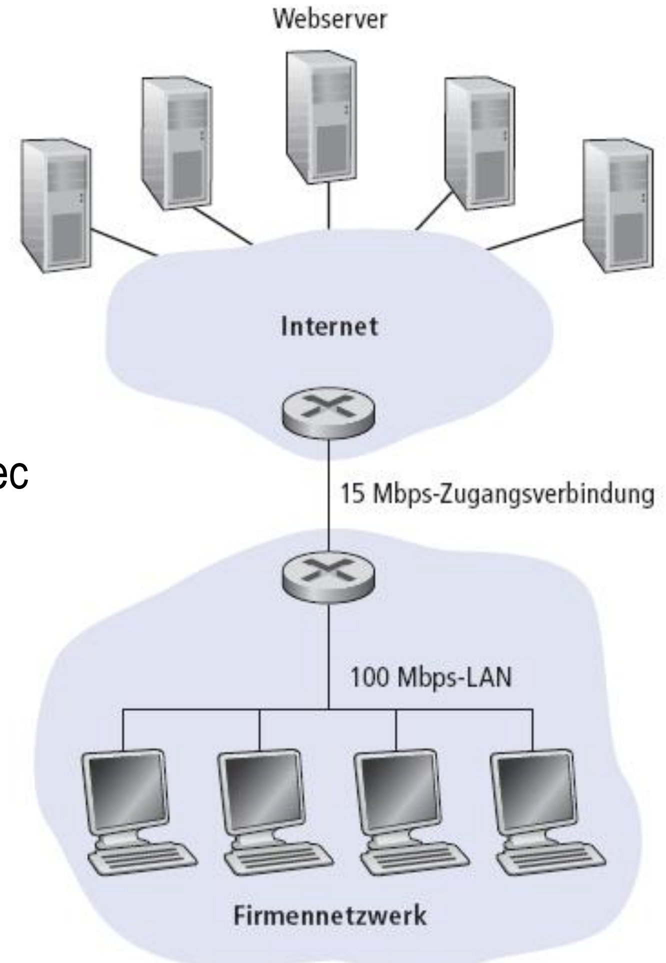
2.2.5 Webcaching

- Cache arbeitet als Client UND als Server
- Explizit oder transparent
- Üblicherweise ist der Cache beim ISP installiert
  - z.B. Universität, Firma oder ISP für Privathaushalte

Vorteile von Webcaching

- Verringert die Antwortzeit
  - Oft höhere Bandbreite zwischen Client und Cache (der bei ISP läuft) verfügbar als zwischen Client und Webserver
- Verringert den Datenverkehr auf der Zugangsleitung eines Firmennetzwerkes
- Kostengünstig
  - Inhalts-anbieter können durch die Nutzung vieler Caches ihre Inhalte gut verbreiten (Ähnliches kann durch P2P-Filesharing erreicht werden)

2.2.5 Beispiel für Webcaching



Annahmen

Bandbreite der Zugangsleitung = 15 Mbps

Bandbreite des LAN = 100 Mbps

Ø Größe eines Objektes = 100.000 Bit

Ø Rate von Anfragen aller Webbrowser der Firma = 150/s

Verzögerung v. Router d. Firma zum Server und zurück = 2 sec

Resultat

Auslastung des LAN = 15%

Auslastung der Zugangsleitung = 100%

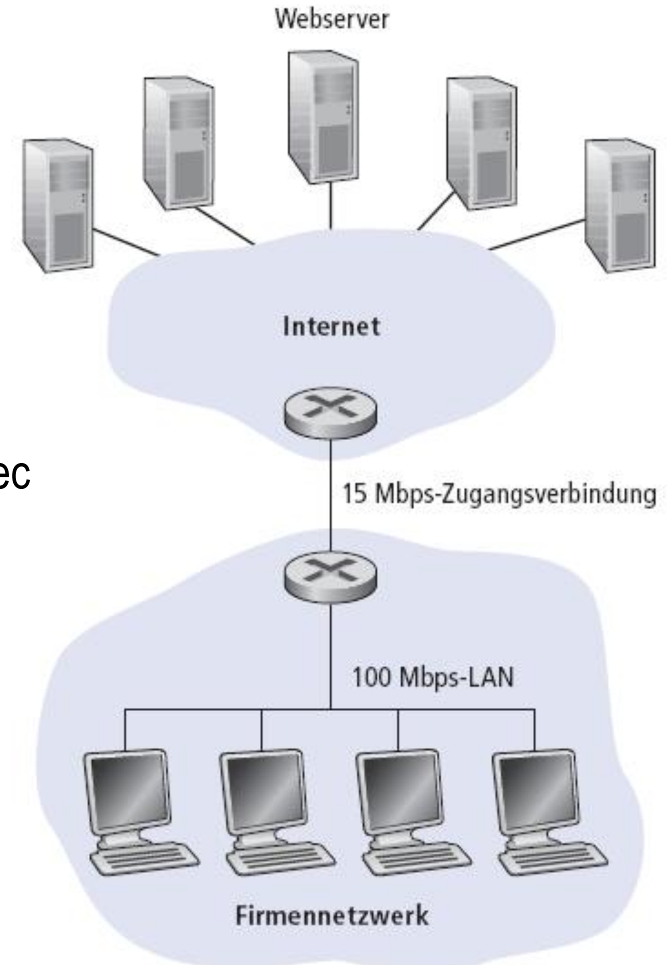
Verzögerung von Internet + Zugangsleitung + LAN =

2s + Minuten (!) + Millisekunden

→ **Wartezeit untragbar!**

Offensichtlicher Lösungsansatz:
Bandbreite der Zugangsleitung erhöhen

2.2.5 Beispiel für Webcaching



Annahmen

Bandbreite der Zugangsleitung jetzt = **100 Mbps**

Bandbreite des LAN = 100 Mbps

Ø Größe eines Objektes = 100.000 Bit

Ø Rate von Anfragen aller Webbrowser der Firma = 150/s

Verzögerung v. Router d. Firma zum Server und zurück = 2 sec

Resultat

Auslastung des LAN = 15%

Auslastung der Zugangsleitung = **15%**

Verzögerung von Internet + Zugangsleitung + LAN =

2s + **Millisekunden** + Millisekunden

ABER: Bandbreite der Zugangsleitung erhöhen ist oft sehr teuer!

→ Anderer Lösungsansatz: Webcaching

2.2.5 Beispiel für Webcaching

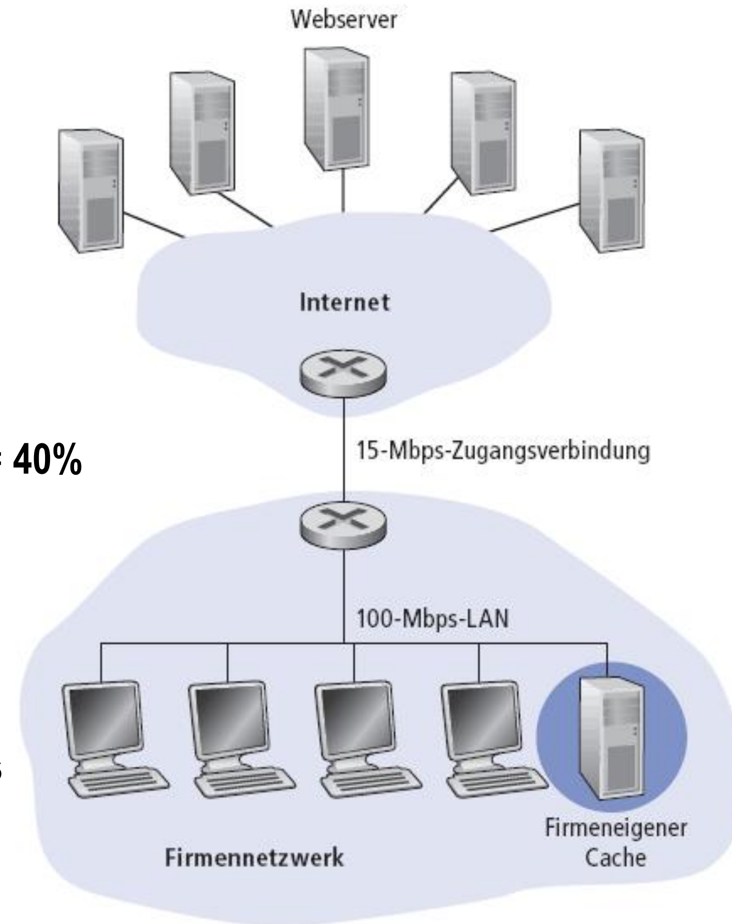
Annahmen

Annahmen bleiben gleich

Bandbreite der Zugangsleitung wieder nur = 15 Mbps

Diesmal Lösungsansatz **Web-Cache**

Angenommene Cache-Trefferrate = 0,4



Resultat

Anfragen die **nahezu sofort** aus dem **Cache** beantwortet werden = **40%**

Anfragen die weiterhin von Webservern beantwortet werden = 60%

Auslastung der Zugangsleitung nur noch = **60%**

Verzögerungen auf der Zugangsleitung **verringert** (~ bei 10 msec)

Verzögerung von Internet + Zugangsleitung + LAN =

0,6 * 2s + 0,4 * **Millisekunden** + Millisekunden **< 1,4s**

0,6 da 60% von Webservern beantwortet

0,4 da 40% Im Cache gefunden

→ Erhebliche Verbesserung

→ Kostengünstiger als Erhöhung der Bandbreite

2.2.6 Bedingtes GET

Conditional GET (Bedingtes GET)

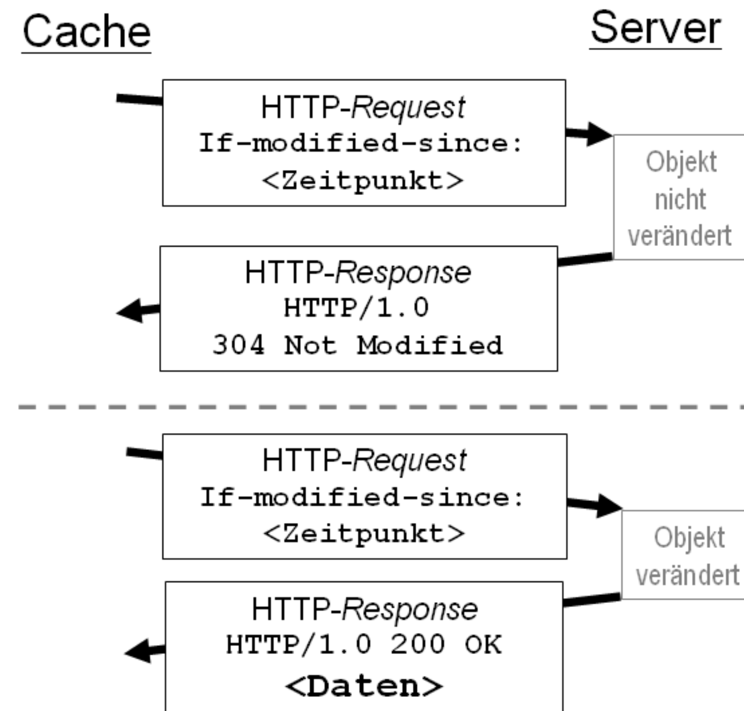
Ein Mechanismus von HTTP mit dem ein Cache beim Server sicherstellen kann, daß die Kopien in seinem Speicher nicht veraltet sind.

Eine HTTP-Request-Nachricht ist eine Conditional-GET-Nachricht, wenn sie enthält:

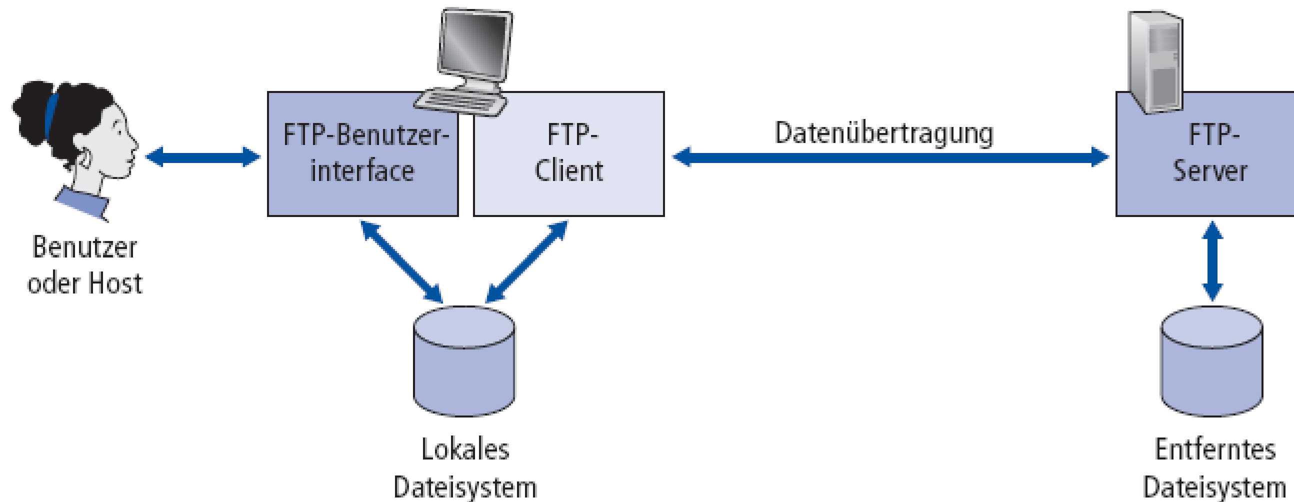
- Die GET-Methode und
- Die Header-Zeile `If-Modified-Since:`

- **Cache:** gibt Änderungsdatum der gespeicherten Version im HTTP-Request an
- **Server:** HTTP-Response enthält kein Objekt, wenn die Version im Cache aktuell ist:

  Code: **`HTTP/1.0 304 Not Modified`**

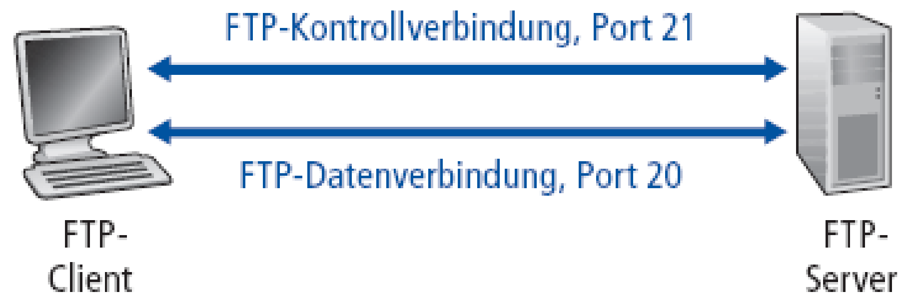


2.3 Dateitransfer: FTP



- Protokoll zum Übertragen einer Datei von/zu einem entfernten Rechner.
- Client/Server-Modell
  - *Client:* Seite, die den Transfer initiiert (vom oder zum entfernten Rechner)
  - *Server:* entfernter Rechner
- Definiert über RFC 959
- FTP-Server verwenden TCP Port 21

2.3 Dateitransfer: FTP



1. FTP-Client kontaktiert FTP-Server auf Port 21, verwendet TCP als Transportprotokoll
2. Client autorisiert sich über die Kontrollverbindung
3. Client betrachtet das entfernte Verzeichnis indem er Kommandos über die Kontrollverbindung schickt
4. Jedes Mal wenn der Server ein Kommando für eine Dateiübertragung empfängt öffnet er eine neue TCP-Datenverbindung zum Client
5. Nach der Übertragung einer Datei schließt der Server die Verbindung

2.3 Dateitransfer: FTP

Kontrollverbindung seperat zum Datenkanal:
→ Nennt man **Out-of-Band** Übermittlung der Steuerinformationen

FTP-Server speichern Informationen zu jedem Benutzer (Gegensatz zu HTTP):
- Zugehörige Kontrollverbindungen
- Aktuelles Verzeichnis auf dem entfernten Host in dem der Benutzer navigiert

→ Gesamtanzahl von Sitzungen, die gleichzeitig verwaltet werden können dadurch stark eingeschränkt

2.3.1 FTP-Befehle und -Antworten

Kommandos:

Werden als ASCII-Text über die Kontrollverbindung übermittelt

- `USER` *`username`*
- `PASS` *`password`*
- `LIST` - gibt eine Liste der Dateien im aktuellen Verzeichnis zurück
- `RETR filename` - lädt eine entfernte Datei auf den lokalen Rechner
- `STOR filename` - überträgt eine lokale Datei auf den entfernten Rechner

Antworten:

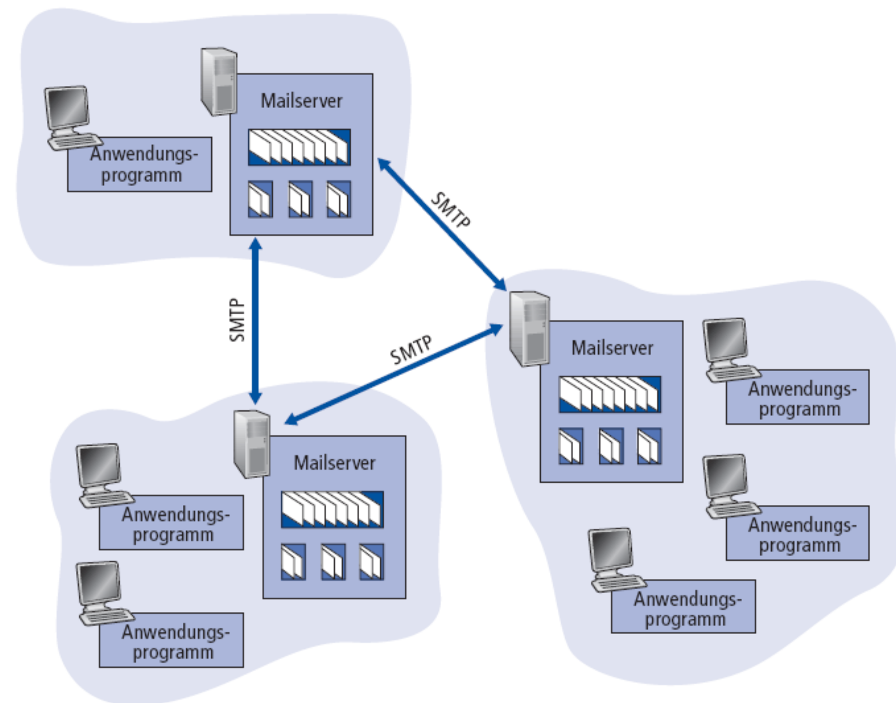
Statuscodes und Erläuterungen (wie bei HTTP)

- `331 Username OK, password required`
- `125 data connection already open; transfer starting`
- `425 Can't open data connection`
- `452 Error writing file`

2.4 E-Mail im Internet

E-Mail besteht aus drei Hauptbestandteilen:

1. Anwendungsprogramm
2. Mailserver
3. Übertragungprotokoll: SMTP



Legende:

2.4 E-Mail im Internet

1. Anwendungsprogramm ("Mail Reader"):

- Funktion: Erstellen, Editieren, Lesen von E-Mail-Nachrichten
  - z.B. Eudora, Outlook, elm, Mozilla Thunderbird

→ Eingehende und ausgehende Nachrichten werden auf dem Server gespeichert

2. Mailserver:

- Die Mailbox enthält die eingehenden Nachrichten eines Benutzers
- Die Warteschlange für ausgehende Nachrichten enthält die noch zu sendenden E-Mail-Nachrichten

3. Übertragungsprotokoll SMTP (Simple Mail Transfer Protocol):

- Wird verwendet, um Nachrichten zwischen Mailservern auszutauschen
  - Client: sendender Mailserver
  - Server: empfangender Mailserver

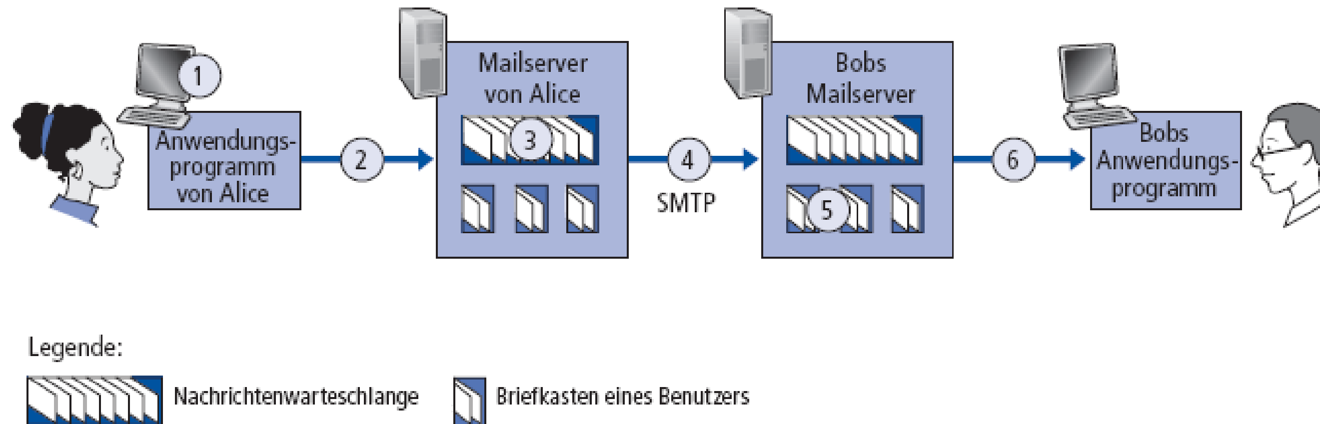
2.4.1 SMTP

- Definiert in RFC 2821
- Zuverlässiger Transport: E-Mail-Nachrichten werden vom Client zum Server mit TCP (Port 25) übermittelt
- Direkter Transport der Nachrichten: von den Mailservern der Absender zu den Mailservern der Empfänger ohne Zwischenlagerung
- Verwendet persistente Verbindungen: bei mehreren Nachrichten mit gleichem Sender und Empfänger können alle über dieselbe TCP-Verbindung übertragen werden
- Nachrichten (sowohl Header als auch Daten) müssen in 7-Bit-ASCII kodiert sein
  → Veraltete Beschränkung; Früher wegen knapper Übertragungskapazität

2.4.1 SMTP

- Drei Phasen des Mail-Versands: Analog zu einer Unterhaltung
  - Handshaking (Begrüßung)
  - Transfer of Messages (Austausch von Informationen)
  - Closure (Verabschiedung)
- Interaktion basiert auf dem Austausch von Befehlen (*Commands*) und Antworten (*Responses*)
  - *Command*: ASCII-Text
  - *Response*: Statuscode und Bezeichnung
- Ein SMTP-Server verwendet `CRLF.CRLF` (`CR` für Wagenrücklauf /carriage return, `LF` für Zeilenvorschub / line feed) um das Ende einer Nachricht zu signalisieren

2.4.1 SMTP Beispiel



1. Alice verwendet ihr Anwendungsprogramm, um eine Nachricht an `bob@someschool.edu` zu erstellen
2. Alices Anwendungsprogramm versendet die Nachricht an ihren Mail-Server; Nachricht wird in der Warteschlange gespeichert
3. Alices Mailserver öffnet als Client eine TCP-Verbindung zu Bobs Mailserver
4. SMTP-Client versendet die Nachricht von Alice über die TCP-Verbindung
5. Bobs Mailserver empfängt die Nachricht von Alices Mailserver und speichert diese in Bobs Mailbox
6. Bob verwendet (irgendwann) sein Anwendungsprogramm und liest die Nachricht

2.4.1 Beispiel für eine SMTP Sitzung

```
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
```

← eigentlich `CRLF.CRLF` um Nachrichtenende zu signalisieren

2.4.2 Vergleich SMTP und HTTP

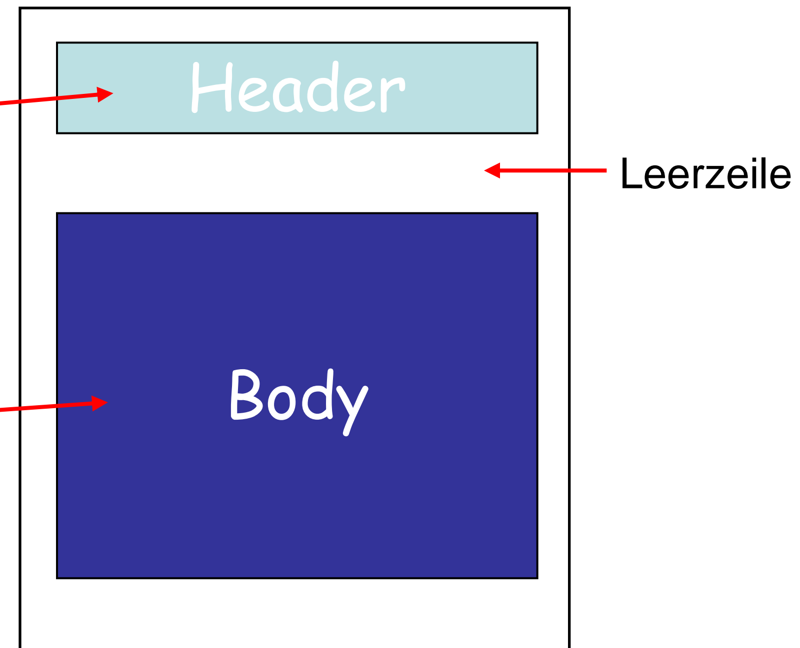
- Protokolltyp:
  - HTTP: Pull-Protokoll (Protokoll zum Herunterladen)
    → TCP-Verbindung wird von dem Host aufgebaut, der die Datei erhalten will
  - SMTP: Push-Protokoll (Protokoll zum Senden von Daten)
    → TCP-Verbindung wird von dem Mailserver aufgebaut, der die Datei senden will
- Beide interagieren mittels ASCII-Befehl/Antwort-Paaren sowie Statuscodes
- Kodierung:
  - SMTP: Überträgt Header und Daten in 7Bit-ASCII-Format. Sonderzeichen und Binärdaten (z.B. eine Bilddatei) müssen extra in 7Bit-ASCII codiert werden
  - HTTP: kennt diese Einschränkung nicht
- Umgang mit Dokumenten mit Medienobjekten:
  - HTTP: Jedes Objekt ist in einer eigenen Antwortnachricht gekapselt
  - SMTP: Mehrere Objekte können in einer Mail-Nachricht (multipart msg) versendet werden

2.4.3 Mail-Nachrichtenformate

RFC 822: Standard für Textnachrichten

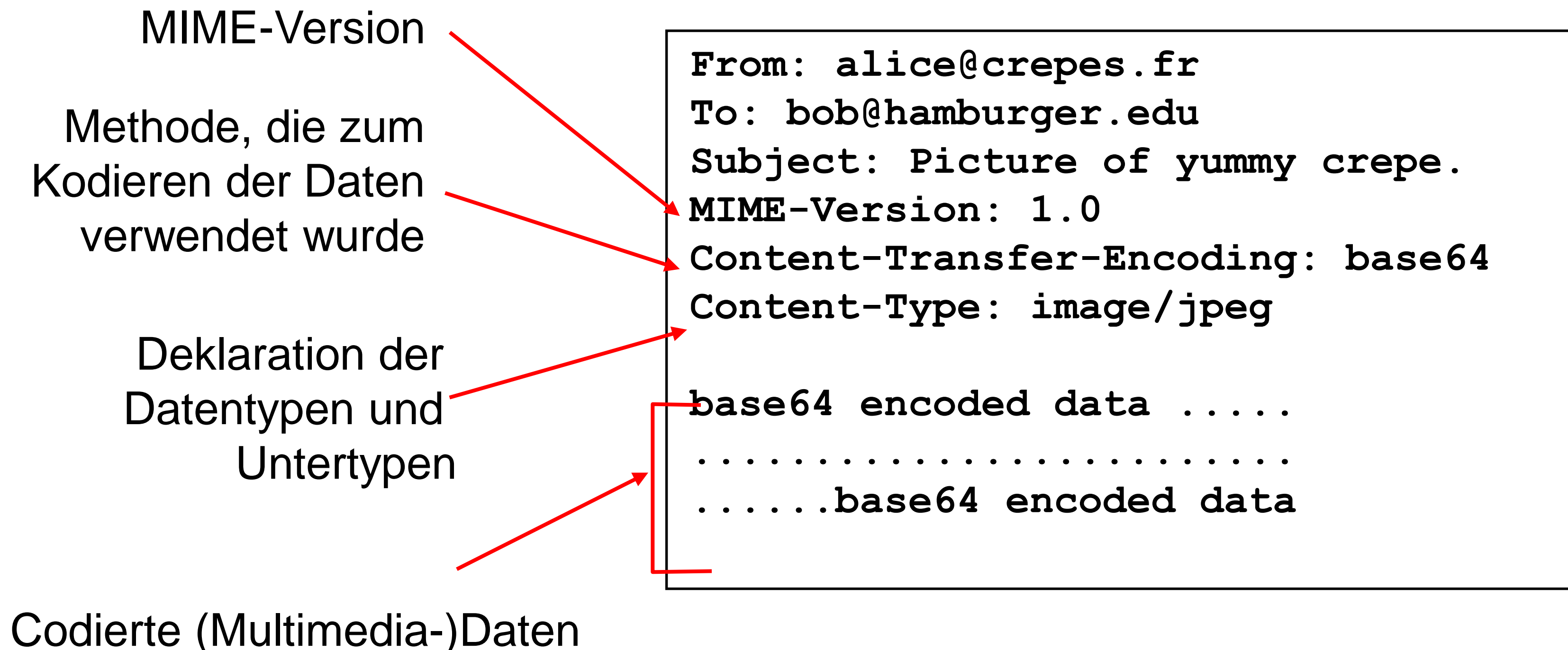
- Header-Zeilen, z.B.
  - To:
  - From:
  - Subject:

  Keine SMTP-Befehle!
- Body
  - Die eigentliche Nachricht in ASCII



2.4.3 MIME - Erweiterung des Nachrichtenformates für Nicht-ASCII-Daten

- MIME: Multimedia Mail Extension, definiert in RFC 2045 und 2056
- Zusätzliche Zeilen im Header deklarieren den MIME-Typ des Inhaltes



2.4.3 Datentypen in MIME

Text
Beispiele für Subtypen:
- **`plain`**
- **`html`**

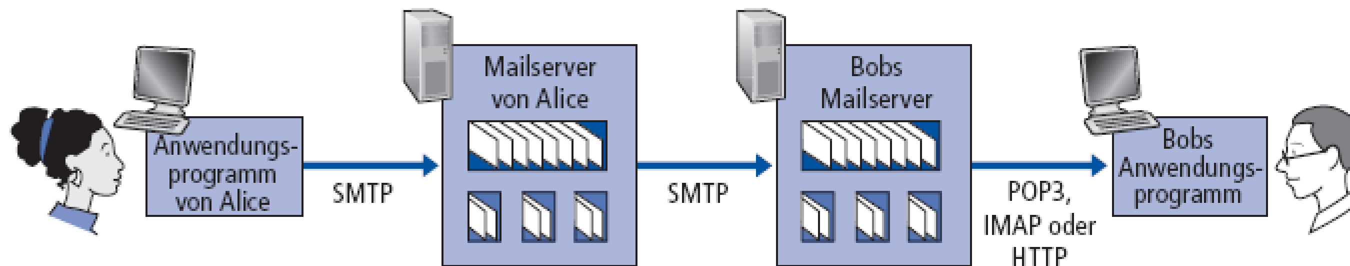
Audio
Beispiele für Subtypen:
- **`basic`** (8-bit mu-law encoded),
- **`32kadpcm`** (32 kbps coding)

Bilder
Beispiele für Subtypen:
- **`jpeg`**
- **`gif`**
- **`png`**

Video
Beispiele für Subtypen:
- **`mpeg`**
- **`quicktime`**

Anwendungen
Daten müssen von der Anwendung vor der Wiedergabe interpretiert werden.
Beispiele für Subtypen:
- **`msword`**
- **`octet-stream`**

2.4.4 Mail-Zugriffsprotokolle



- Das Senden einer E-Mail ist ein zweistufiger Prozess:
  - Alices Anwendungsprogramm auf ihrem Computer sendet die Nachricht zuerst per SMTP an Alices Mailserver
  - Alices Mailserver speichert die Nachricht und versucht in regelmäßigen Abständen die Nachricht per SMTP an Bobs Mailserver zu übermitteln, bis er Erfolg hat. Bobs Mailserver speichert dann die Nachricht in Bobs Postfach.
- Mail-Zugänge benutzen oft eine Client-Server-Architektur
- Will Bob mit seinem Anwendungsprogramm (Client) die Nachricht von Bobs Mailserver (Server) empfangen, kann er dazu kein SMTP verwenden, da das Abrufen der Nachricht eine Pull-Operation ist.
- Beliebte Mail-Zugriffsprotokolle (Pull-Protokolle): **POP3**, **IMAP** und **HTTP**

2.4.4 Mail-Zugriffsprotokolle

- **POP: Post Office Protocol** [RFC 1939]
  - Autorisierung (Anwendung <--> Server) und Zugriff/Download
- **IMAP: Internet Mail Access Protocol** [RFC 1730]
  - Größere Funktionalität (deutlich komplexer)
  - Manipulation der auf dem Server gespeicherten Nachrichten
- **HTTP:**
  - z.B. bei Hotmail, Yahoo!Mail etc.

2.4.4 POP3

Autorisierungsphase:

- Befehle des Clients:
  - **user**: Benutzername
  - **pass**: Passwort
- Antworten des Servers:
  - **+OK**
  - **-ERR**

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

Transaktionsphase:

- **list**: Nachrichten auflisten
- **retr**: Nachrichten herunterladen
- **dele**: Löschen von Nachrichten
- **Quit: Ende**

```
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 1 contents>
S: .
C: dele 2
C: quit
S: +OK POP3 server signing off
```

S: Server,
C: Client

2.4.4 POP3 und IMAP

POP3 Modi:

- "*Download-and-Delete*"-Modus (Beispiel auf voriger Folie)
  Andere E-Mail-Clients haben nach Abruf keine Möglichkeit mehr die Mails zu lesen
- "*Download-and-Keep*"-Modus
  Ermöglicht den reinen Lesezugriff auf Nachrichten; auch andere Clients haben Zugriff
- POP3 ist zustandslos zwischen einzelnen Sitzungen

IMAP:

- Alle Nachrichten bleiben an einem Ort: auf dem Server
- Nachrichten können auf dem Server in Ordnern verwaltet werden
- IMAP bewahrt den Zustand zwischen einzelnen Sitzungen:
  - Namen von Ordnern und Zuordnung von Nachrichtennummer und Ordnername bleiben erhalten

2.5 DNS

Bei Menschen gibt es viele verschiedene Identifikationsmechanismen.

- z.B. Name, Ausweisnummer

Internet-Hosts und Router werden auch über bestimmte Mechanismen identifiziert:

- IP-Adresse (32 Bit) – für die Adressierung in Paketen
- “Name”, z.B., www.yahoo.com – von Menschen verwendet

Frage: Wie findet die Abbildung zwischen IP-Adressen und Namen statt?

→ Domain Name System (DNS)

- *Verteilte Datenbank*, implementiert eine Hierarchie von *Nameservern*
- *Protokoll der Anwendungsschicht*, wird von Hosts verwendet, um Namen *aufzulösen* (Abbildung zwischen Adresse und Name)
  - Zentrale Internetfunktion, implementiert als Protokoll der Anwendungsschicht
  - Grund: Komplexität nur am Rand des Netzwerkes!

2.5.1 Von DNS erbrachte Dienste

DNS-Dienste:

- Übersetzung von Hostnamen in IP-Adressen
- Aliasnamen für Hosts
  - Kanonische Namen und Aliasnamen
- Aliasnamen für Mailserver
- Lastausgleich (Loadbalancing)
  - Replizierte Webserver: mehrere IP-Adressen von einem kanonischen Namen

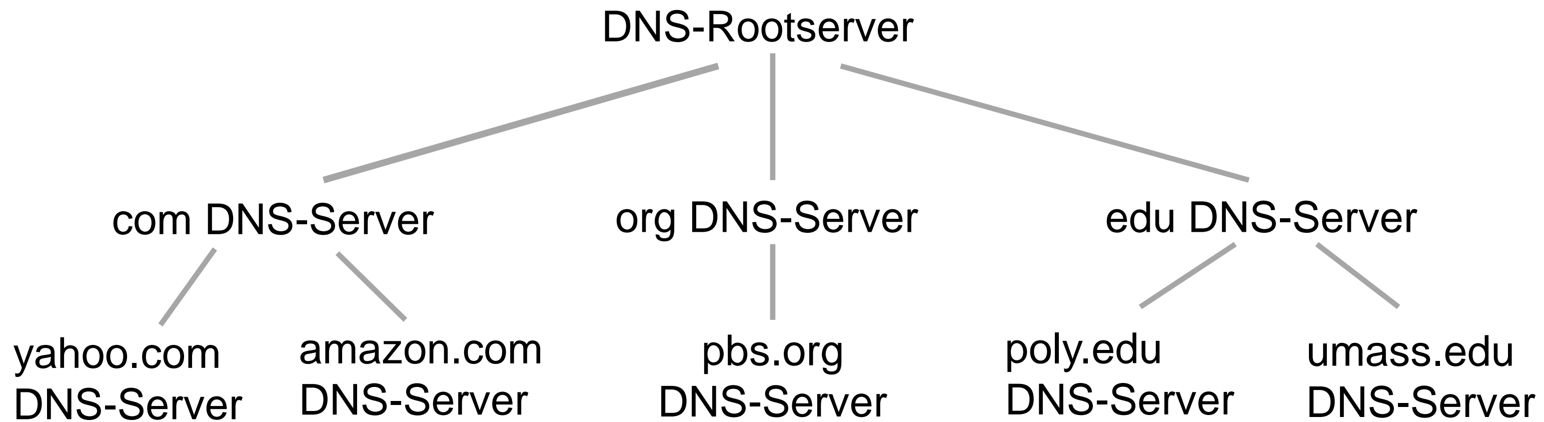
2.5.2 Arbeitsweise von DNS

Warum ist DNS nicht zentralisiert?

- Robustheit gegenüber Fehlern und Angriffen
  - Wenn ein zentraler DNS-Server zusammenbrachen würde, dann auch das ganze Internet!
- Datenverkehrsaufkommen
  - Wenn es nur einen zentralen DNS-Server gäbe müsste dieser *alle* DNS-Anfragen beantworten.
- Große „Distanz" zur zentralisierten Datenbank
  - Ein einzelner DNS-Server kann nicht in der Nähe aller anfragenden Clients sein. Anfragen von weit entfernten Orten mussten über möglicherweise langsame und überlastete Leitungen reisen was zu spürbaren Verzögerungen führen könnte.
- Wartung
  - Ein einziger DNS-Server müsste Datensätze für alle Internethosts beinhalten. Daher wäre diese zentralisierte Datenbank nicht nur riesig, sondern sie müsste auch häufig aktualisiert werden um jeden neuen Host zu enthalten.

→ Fazit: Eine zentralisierte Datenbank in einem einzelnen DNS-Server ist einfach nicht skalierbar. DNS kann als **verteilte, hierarchische Datenbank** im Internet betrachtet werden.

2.5.2 Arbeitsweise von DNS



Client sucht die IP-Adresse von www.amazon.com:

1. Client fragt seinen lokalen DNS-Server
2. Dieser fragt einen DNS-Rootserver, um den DNS-Server für com zu finden
3. Danach fragt er den com-DNS-Server, um den amazon.com-DNS-Server zu finden
4. Dann wird der amazon.com-DNS-Server gefragt, um die IP-Adresse zu www.amazon.com zu erhalten

2.5 Hierarchie der DNS-Server

- DNS-Rootserver
  - 13 DNS-Rootserver weltweit im Internet
  - Jeder dieser DNS-Rootserver ist eigentlich ein Cluster replizierter Server
  - Kennt die Adressen der Top-Level-Domain (com, net, org, de, uk, …) Server
  - Gibt diese Informationen (Adressen) bei Anfragen an die lokalen Nameserver weiter

- Top-level domain (TLD) Server
  - Verantwortlich für com, org, net, edu, gov etc. sowie für alle Länder-Domains, z.B. de, uk, fr, ca, jp
  - Network Solutions ist verantwortlich für den com-TLD-Server
  - Educause hat die Verantwortung für den edu-TLD-Server

- Authoritative DNS-Server
  - DNS-Server einer Organisation, der eine autorisierte Abbildung der Namen dieser Organisation auf IP-Adressen anbietet
  - Verwaltet von der entsprechenden Organisation oder einem Service Provider

2.5.2 Hierarchie der DNS-Server

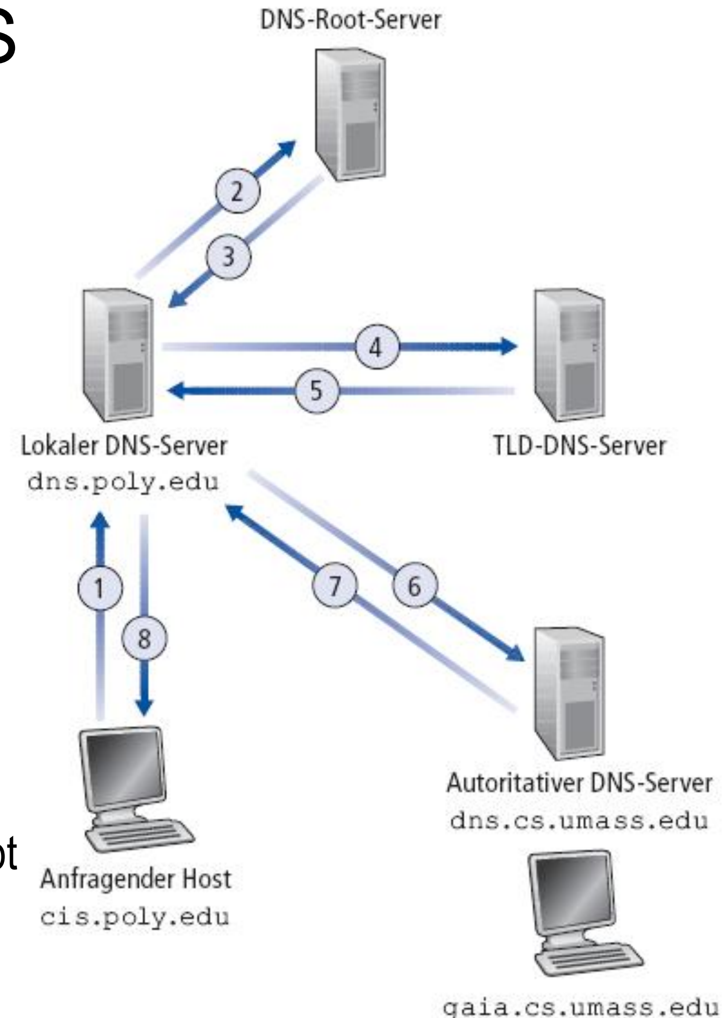
Lokale DNS Server

- Gehören **nicht** strikt zur Hierarchie der DNS-Server
- Jeder ISP (z.B. Firmen, Universität, ISP für Privatkunden) besitzt einen lokalen Nameserver
- Werden auch “Default-Nameserver” genannt
- Agieren als Proxy
- Wenn ein Host eine DNS-Anfrage startet, dann schickt er diese an seinen lokalen Nameserver
  - Dieser kümmert sich um die Anfrage so lange, bis eine endgültige Antwort vorliegt
  - Dazu kontaktiert er bei Bedarf Root-Nameserver, TLD-Nameserver und autoritative Nameserver
  - Dann schickt er die Antwort an den Host zurück

2.5.2 Namensauflösung mit DNS

Host cis.poly.edu fragt nach der IP-Adresse von gaia.cs.umass.edu:

1. **Der Host** sendet eine Anfrage an seinen lokalen DNS-Server. Die Anfrage enthält den zu übersetzenden Hostnamen `gaia.cs.umass.edu`.
2. **Der lokale DNS-Server** leitet die Anfrage an einen DNS-Rootserver weiter.
3. **Der DNS-Rootserver** reagiert auf den Teil `edu` und gibt eine Liste von IP-Adressen von TLD-Servern für `edu` zurück.
4. **Der lokale DNS-Server** sendet dann die Anfrage erneut an einen der TLD-Server.
5. **Der TLD-Server** reagiert auf den Teil `umass.edu` und gibt die IP-Adresse des autoritativen DNS-Servers zurück.
6. **Der lokale DNS-Server** sendet dann die Anfrage erneut an den autoritativen DNS-Server.
7. **Der autoritative DNS-Server** antwortet mit der IP-Adresse für `gaia.cs.umass.edu`.



→ Dieses Beispiel benutzt sowohl rekursive als auch iterative Anfragen.

2.5.2 Namensauflösung mit DNS

Zwei Arten der Anfragen:

- **Iterative Anfragen**

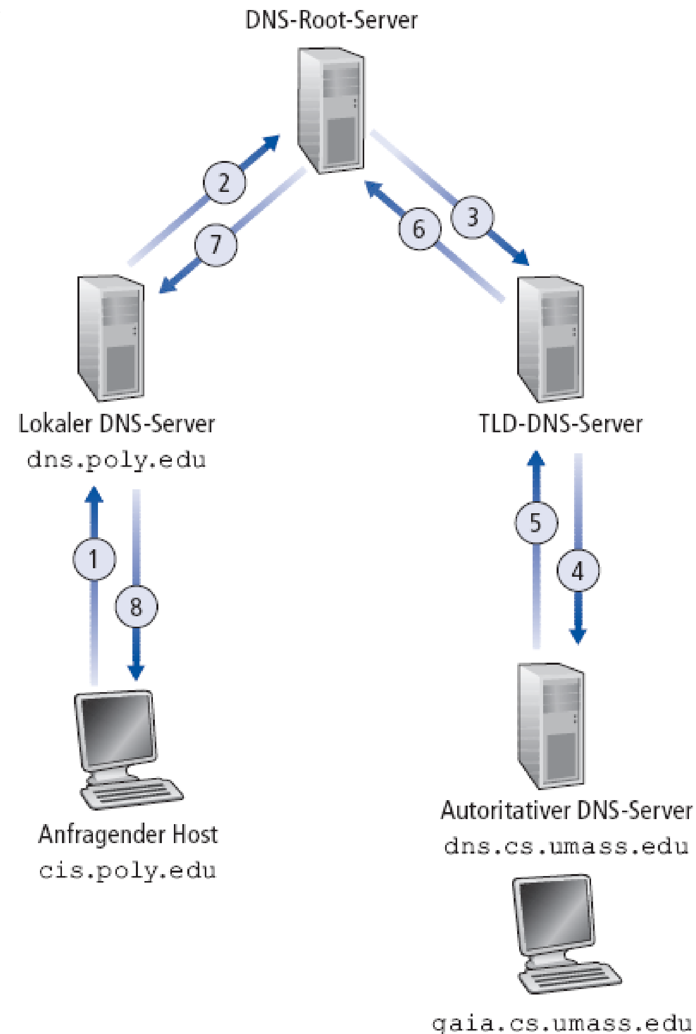
Angesprochene Server in der Hierarchie antworten mit einem Verweis auf andere Server

  - "Ich kenne den Namen nicht, frag diesen Server …"

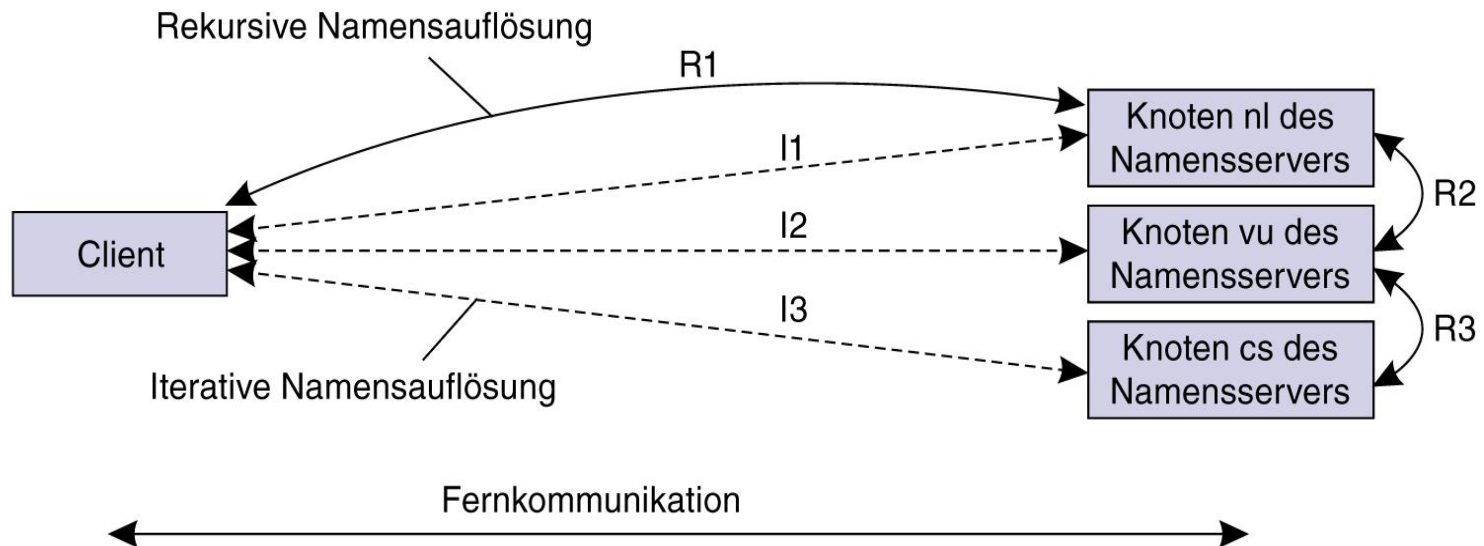
- **Rekursive Anfragen**

Wenn der gefragte Server nicht direkt eine Antwort an den fragenden Host zurückgibt, sondern die Anfrage weiterleitet handelt es sich um eine rekursive Anfrage (→ siehe Abb.).

  - Zusätzliche Belastung!
  - Root-Nameserver erlauben dies häufig nicht, andere Nameserver dagegen schon!



Delay vs. Overhead



2.5.2 DNS Caching

- Sobald ein Nameserver eine Abbildung zur Namensauflösung kennenlernt, merkt er sich diese in einem Cache
  - Die Adressen der TLD-Server werden üblicherweise von den lokalen Nameservern gecacht
    - Root-Nameserver werden eher selten angesprochen

  → Verbessert Leistungen hinsichtlich Verzögerungen

  → Reduziert Anzahl der zur Auflösung benötigten DNS-Nachrichten
  - Die Einträge im Cache werden nach einer vorgegebenen Zeit wieder gelöscht
- Mechanismen zur Pflege von Cache-Einträgen und zur Benachrichtigung bei Änderungen werden derzeit von der IETF entwickelt
  - RFC 2136

2.5.3 DNS Resource Records

Resource Records (=RR, Ressourcendatensätze)

- Ein RR ist ein Viertupel mit den Feldern (`Name, Wert, Typ, TTL`)
- DNS-Server speichern ihre Informationen in Form von RR
- Jede DNS-Antwortnachricht beinhaltet einen oder mehrere RR
- RR werden definiert in RFC 1034 und RFC 1035

2.5.3 DNS Resource Records

RR-Format: (`Name, Wert, Typ, TTL`)

TTL – Bestimmt, wann eine Ressource aus einem Cache entfernt werden sollte.

Name und **Wert** hängen vom Typ ab:

- Typ=A
  `name` ist der Hostname
  `value` ist die IP-Adresse
- Typ=NS
  `name` ist eine Domain (z.B. foo.com)
  `value` ist der Hostname des autoritativen Nameservers für diese Domain
- Typ=MX
  `value` ist der Name des Mailservers für die Domain **`name`**
- Typ=CNAME
  `name` ist ein Alias für einen kanonischen (echten) Namen:
  -`www.ibm.com` ist ein Alias für `servereast.backup2.ibm.com`
  `value` ist der kanonische Name

2.5.3 DNS-Nachrichtenformat

DNS-Anfragen (Query) und DNS-Antwortnachrichten (Reply) haben dasselbe Format.

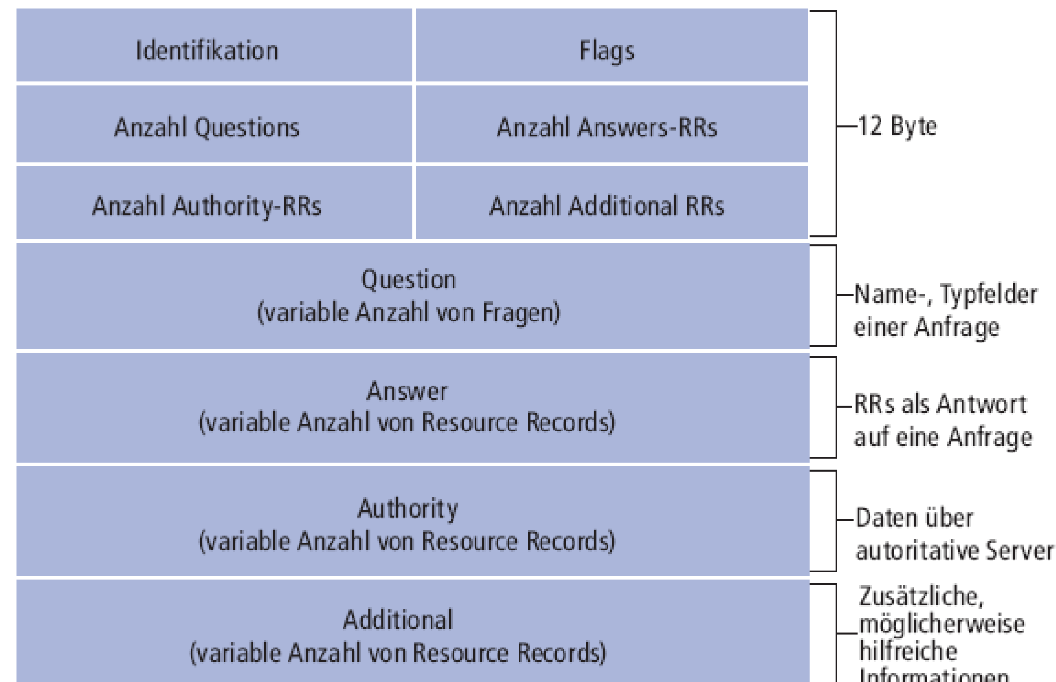
Header-Felder

- Identification: 16-Bit-ID, ermöglicht die Zuordnung einer eingehenden Reply-Nachicht zur vorangegangenen Query-Nachricht (ID ist bei beiden Nachrichten gleich).

- 1Bit-Flags:
  - query/reply
  - recursion desired
  - recursion available
  - reply is authoritative

- Vier Anzahl-Felder:

Diese Felder zeigen, wie häufig die einzelnen Datenabschnitte, die dem Header folgen, auftreten.



2.5.3 DNS-Nachrichtenformat

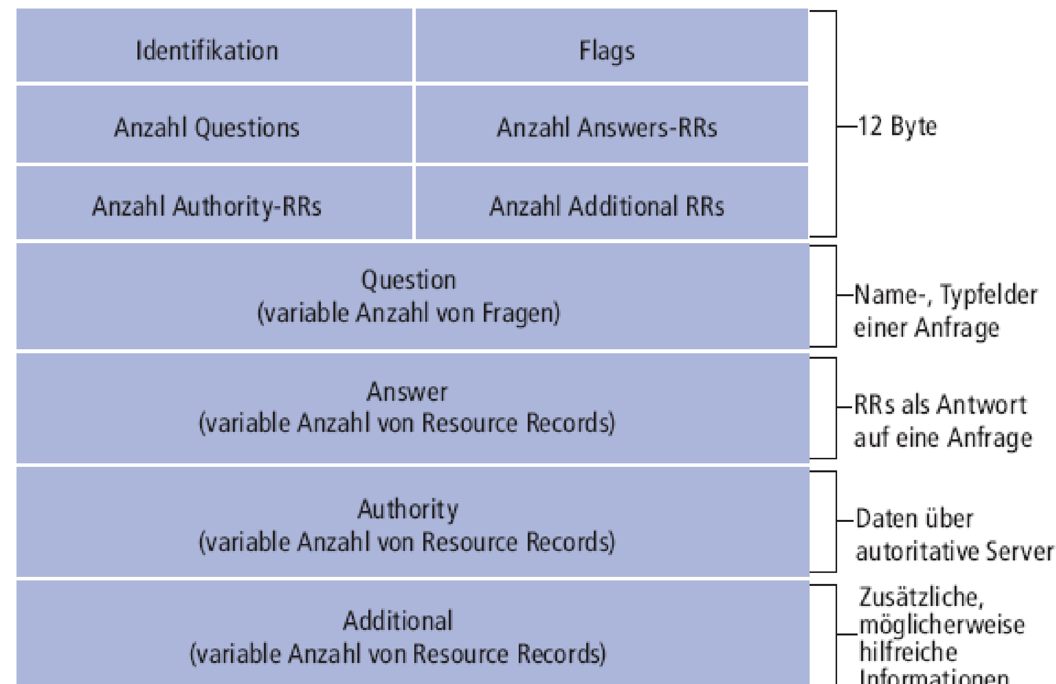
Body-Felder

- Question-Abschnitt: Enthält Informationen über die gestellte Anfrage.
  - Namensfeld – Enthält den angefragten Namen
  - Typfeld – Spezifiziert die Art der Frage

- Answer-Abschnitt:
Enthält die RR für den Namen der ursprünglich angefragt wurde.

- Authority-Abschnitt:
Enthält RR zur Identifikation von authoritativen Servern für die Antworten.

- Additional-Abschnitt:
Enthält je nach Typ der Frage sonstige hilfreiche Datensätze zur Auflösung des angefragten Namens.



2.6 Peer-to-Peer-Anwendungen

- Peers sind gleichwertige Anwendungen
- Keine zentrale Steuerung
- Selbstorganisation
- Emergentes Verhalten
- Schwarmverhalten, Schwarmintelligenz

2.6 Peer-to-Peer-Anwendungen

Beispiel

1. Alice führt eine P2P-Client-Applikation auf ihrem Laptop aus
2. Sie verbindet sich zeitweise mit dem Internet und bekommt bei jeder Verbindung eine andere IP-Adresse zugewiesen
3. Sie sucht im P2P-Netz nach "Hey Jude"
4. Die P2P-Applikation zeigt die Peers an, die eine Kopie von „Hey Jude" besitzen
5. Alice wählt einen der Peers aus, → Bob
6. Die Datei wird von Bobs PC auf den Laptop von Alice kopiert mittels HTTP
7. Während Alice sich eine Datei herunterlädt stellt sie anderen Benutzern Dateien zur Verfügung
8. Alices Anwendung ist sowohl Webclient als auch vorübergehend ein Webserver

Da alle Peers auch Server sind, skaliert das System gut!

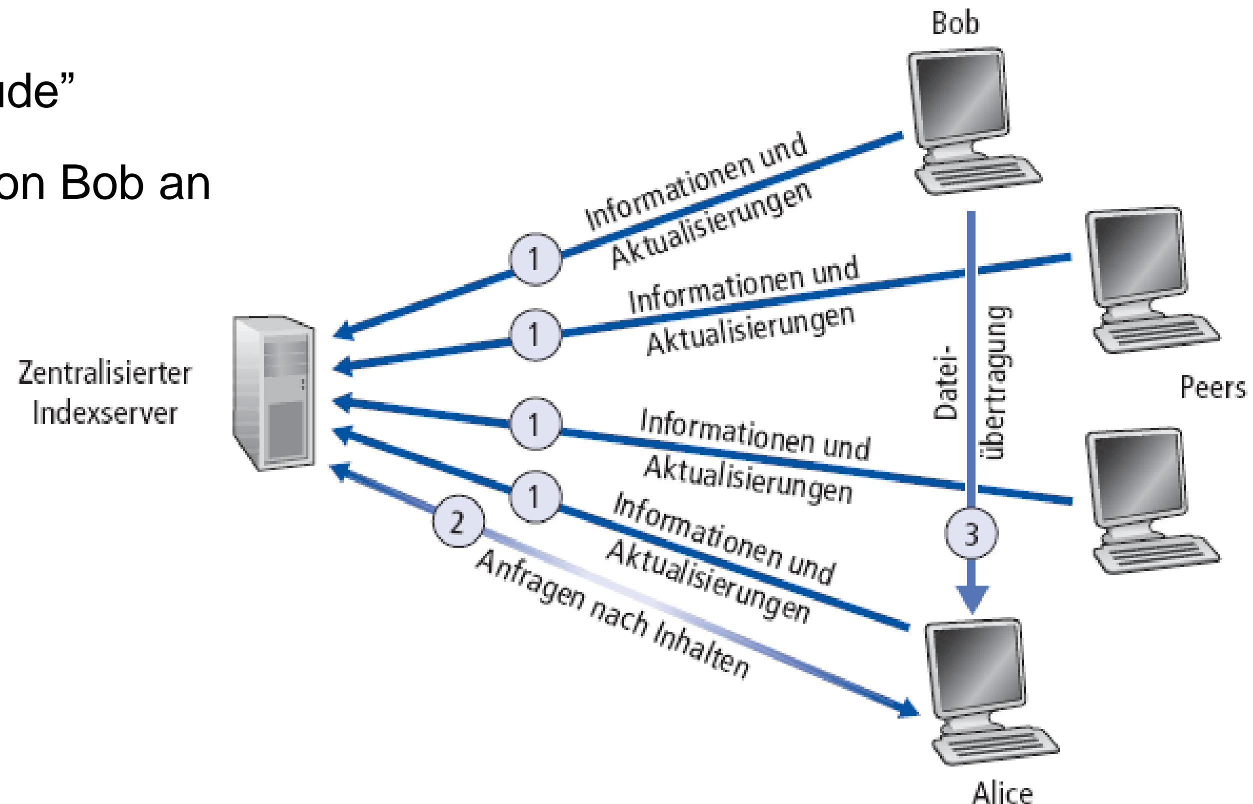
2.6.1 Peer-to-Peer-Anwendungen

Ursprüngliches “Napster”-Design

1) Wenn sich ein Peer verbindet übermittelt er zum zentralen Server:
   - IP-Adresse
   - Verfügbarer Inhalt

2) Alice fragt nach “Hey Jude”

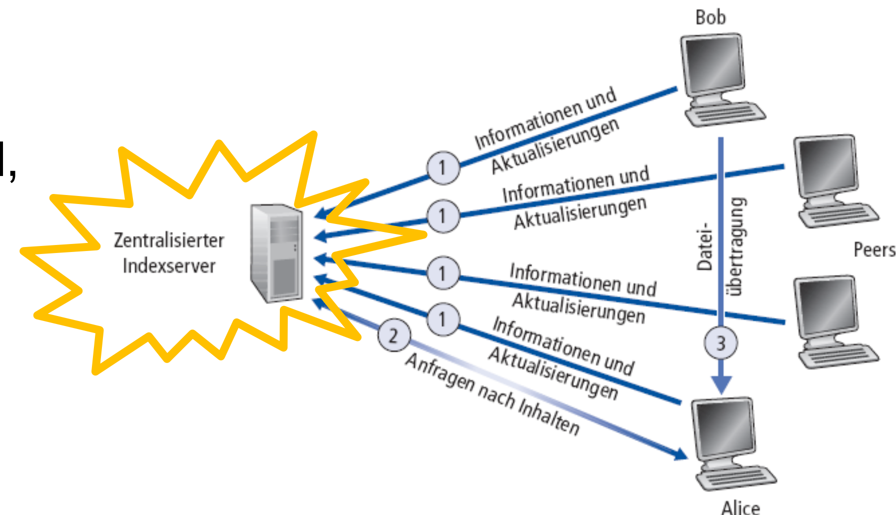
3) Alice fordert die Datei von Bob an



2.6.1 Zentralisierter Index - Probleme

- **Single Point of Failure**
  Es gibt eine zentrale Instanz die nur einmal vorhanden ist: der Server mit dem Verzeichnis!
- **Leistungsengpass und Infrastrukturkosten**
  Bei großen P2P-Systemen muss der Indexserver einen riesigen Index verwalten und jede Sekunde auf Tausende von Anfragen reagieren.
- **Verletzung des Urheberrechts**
  P2P-Filesharing-Systeme können es Benutzern ermöglichen urheberrechtlich geschützte Inhalte kostenlos zu erhalten. Besitzt eine P2P-Filesharing-Firma einen zentralisierten Indexserver, können Strafanträge dazu führen, dass der Indexserver heruntergefahren werden muss. Es ist viel schwieriger dezentrale Architekturen abzuschalten.

→ Der Dateitransfer erfolgt zwar dezentral, aber die Suche nach dem Inhalt ist hochgradig zentralisiert.



2.6.1 Dezentraler Ansatz: Anfrage-Fluten

Beispiel: Gnutella-Protokoll

- Index wird vollständig über die Gemeinschaft von Peers verteilt. Jeder Peer indiziert nur die Dateien, die er zur Verteilung freigibt.
- Peers formen ein abstraktes, logisches Netzwerk

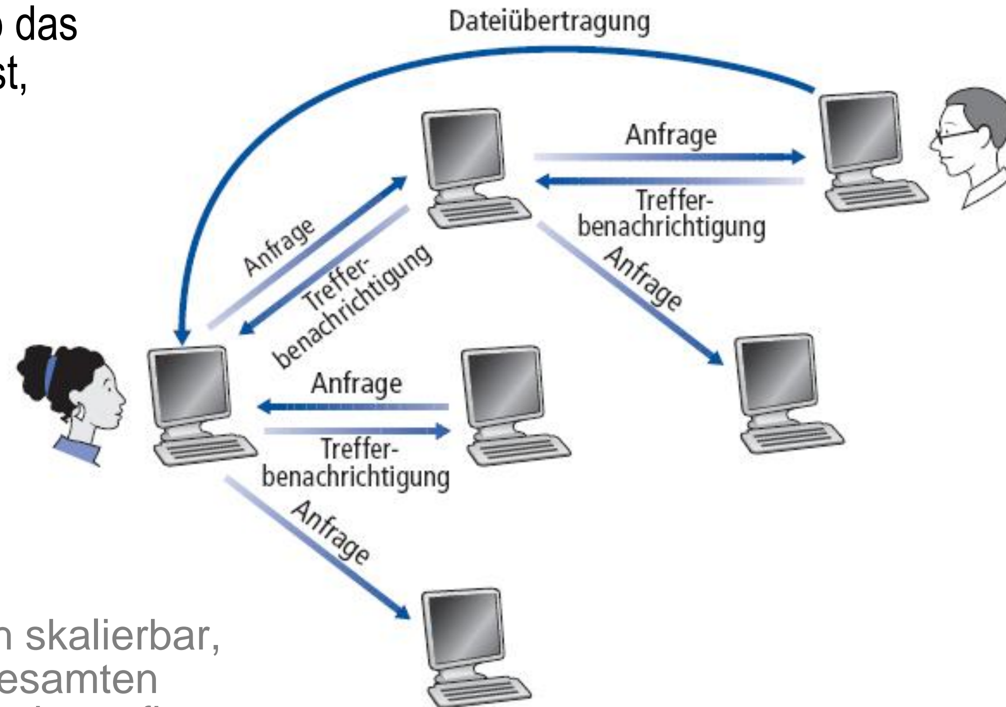
→ wird als Overlay-Netzwerk (Graph) bezeichnet

  - Es besteht eine Kante zwischen den Peers X und Y, wenn eine TCP-Verbindung zwischen ihnen besteht
  - Kanten sind virtuelle (nicht physikalische) Leitungen
  - Ein Peer ist üblicherweise mit weniger als zehn Nachbarn im Overlay verbunden
- Peers senden alle Nachrichten über schon bestehende TCP-Verbindungen an ihre benachbarten Peers im Overlay-Netzwerk.

2.6.1 Dezentraler Ansatz: Anfrage-Fluten

Beispiel

1. Alices Client sendet die Anfrage mit dem Schlüsselwort "Lolcat" an alle ihre Nachbarn.
2. Diese leiten die Anfrage jeweils rekursiv an alle ihre Nachbarn weiter.
3. Erhält ein Peer eine Anfrage überprüft er, ob das Schlüsselwort zu irgendeiner der Dateien passt, die er der Allgemeinheit zur Verfügung stellt.
4. Gibt es eine Übereinstimmung, dann sendet der Peer eine Trefferbenachrichtigung an Alice, die den passenden Dateinamen und die File-Größe enthält.
Diese Trefferbenachrichtigungen werden in umgekehrter Richtung auf demselben Pfad wie die Anfrage geschickt.



→Konzept der Anfrage-Fluten nicht wirklich skalierbar, da eine Anfrage immer an jeden Peer im gesamten Overlay-Netzwerk gesendet wird. Das Verkehrsaufkommen im zugrunde liegenden Netz wird mit der Zeit zu groß!

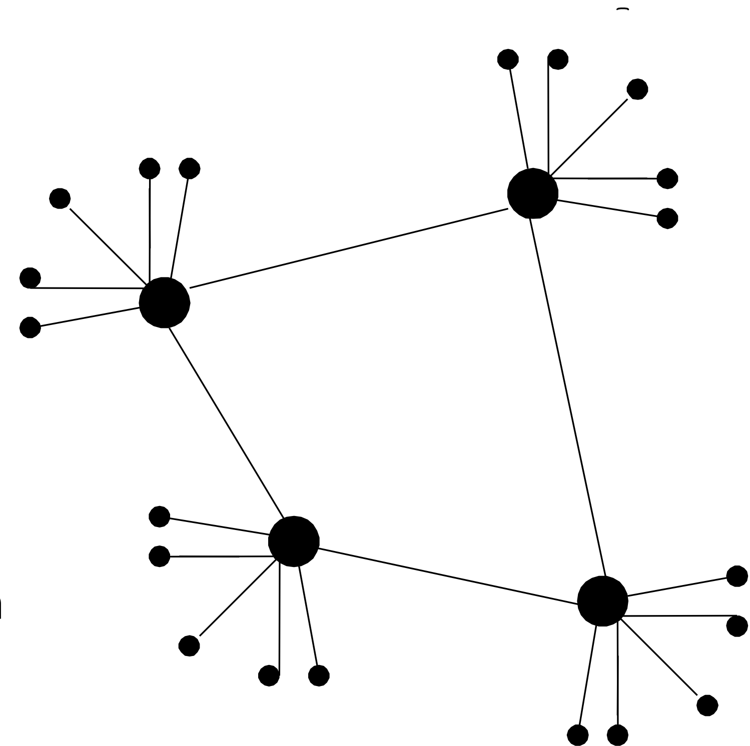
2.6.1 Beitreten neuer Peers

1. Zum Beitreten muss der Peer einen schon in das Gnutella-Netzwerk eingebundenen Peer finden, z.B. über eine Liste von Kandidaten, die zentralisiert gespeichert wird.
2. Der neue Peer probiert so lange nacheinander zu diesen Kandidaten eine TCP-Verbindung aufzubauen bis er einmal Erfolg damit hat.
3. *Fluten:* Der neue Peer flutet eine „Ping“-Nachricht über den Peer, mit dem er sich gerade verbunden hat. Peers die das Ping erhalten antworten dem neuen Peer direkt mit einer Pong-Nachricht.
4. Der neue Peer erhält eine Reihe von Pong-Nachrichten und kann weitere TCP-Verbindungen zu anderen Peers aufbauen.

2.6.1 Hierarchische Overlay-Netzwerke

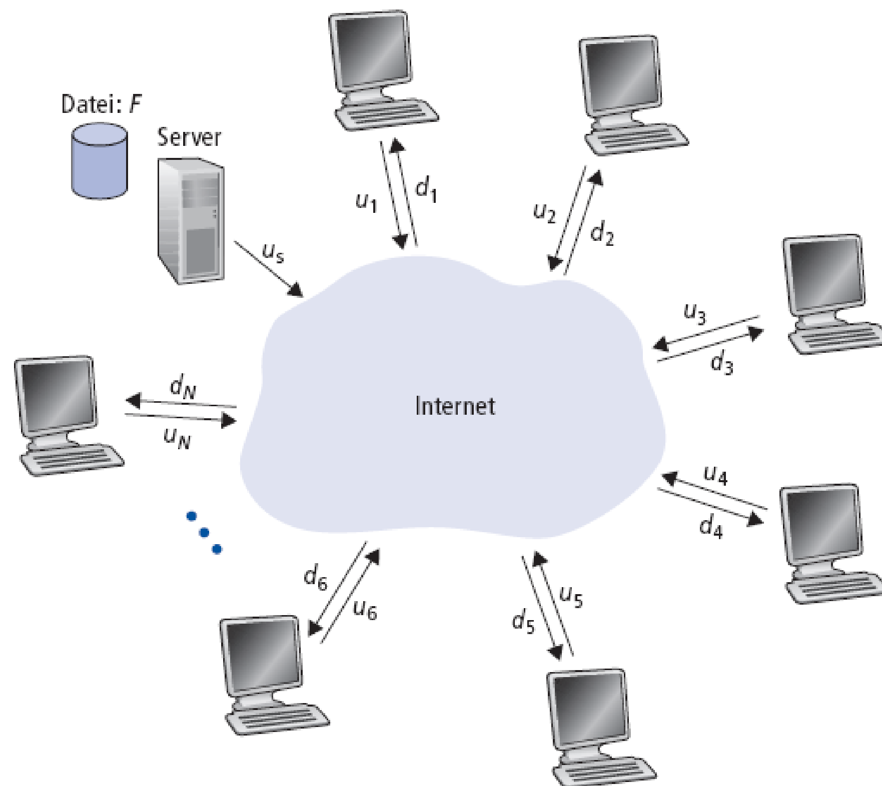
→Ansatz zwischen zentralem Index und dem Fluten von Anfragen:

- Jeder Peer ist entweder selbst ein „Super-Peer“ oder einem anderen Super-Peer zugeordnet
  - TCP-Verbindung zwischen Peer und seinem Super-Peer
  - TCP-Verbindungen zwischen einigen Super-Peers
- Super-Peers kennen die Dateien, welche von ihren Peers angeboten werden



2.6.2 Client-Server- VS. P2P-Architektur

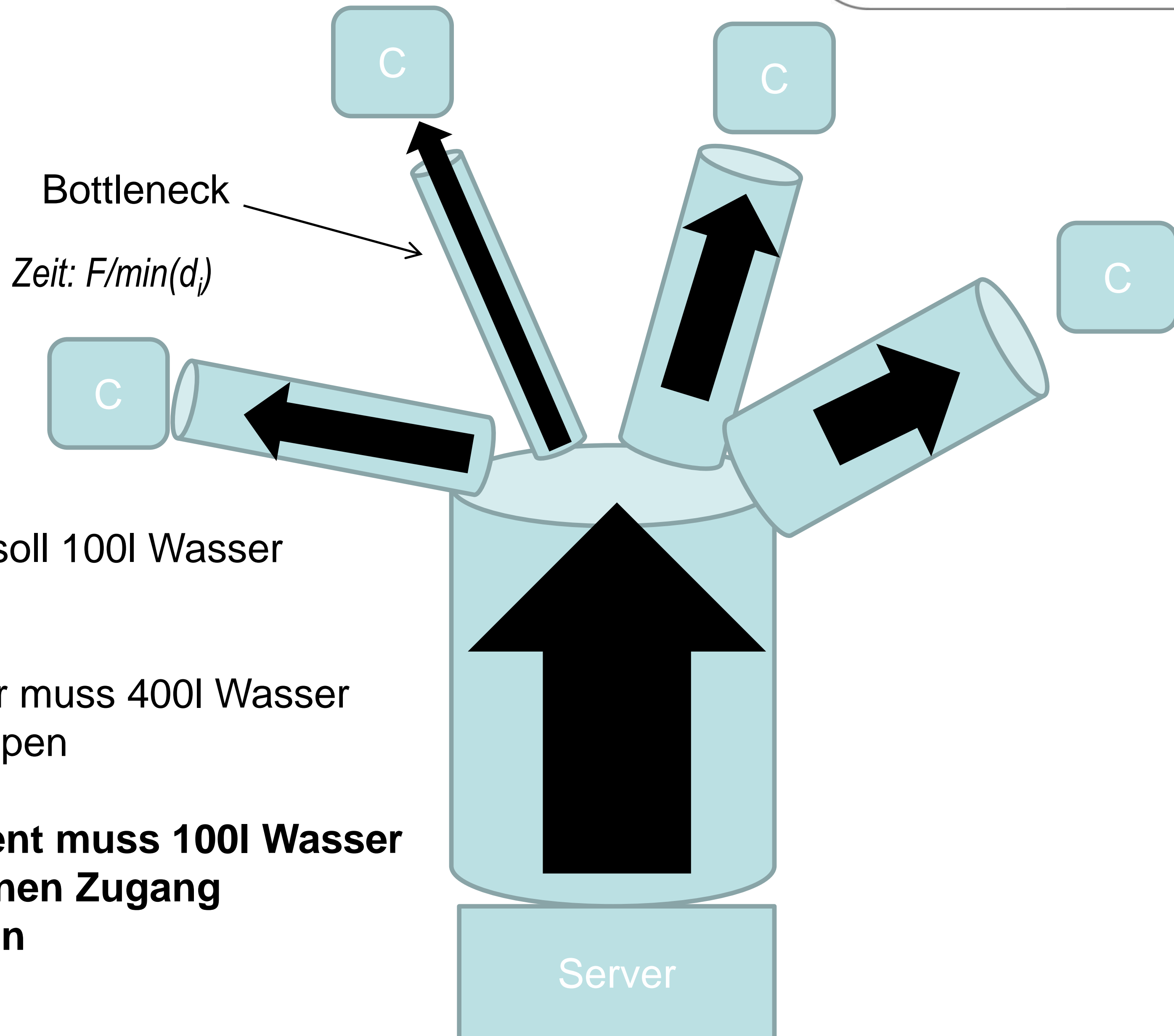
Frage: Wie lange dauert es bei den beiden unterschiedlichen Architekturen eine Datei der Größe F, die zu Anfang nur auf einem Server liegt, an *N* andere Computer zu verteilen?



u_s: Bandbreite vom Server in das Netz

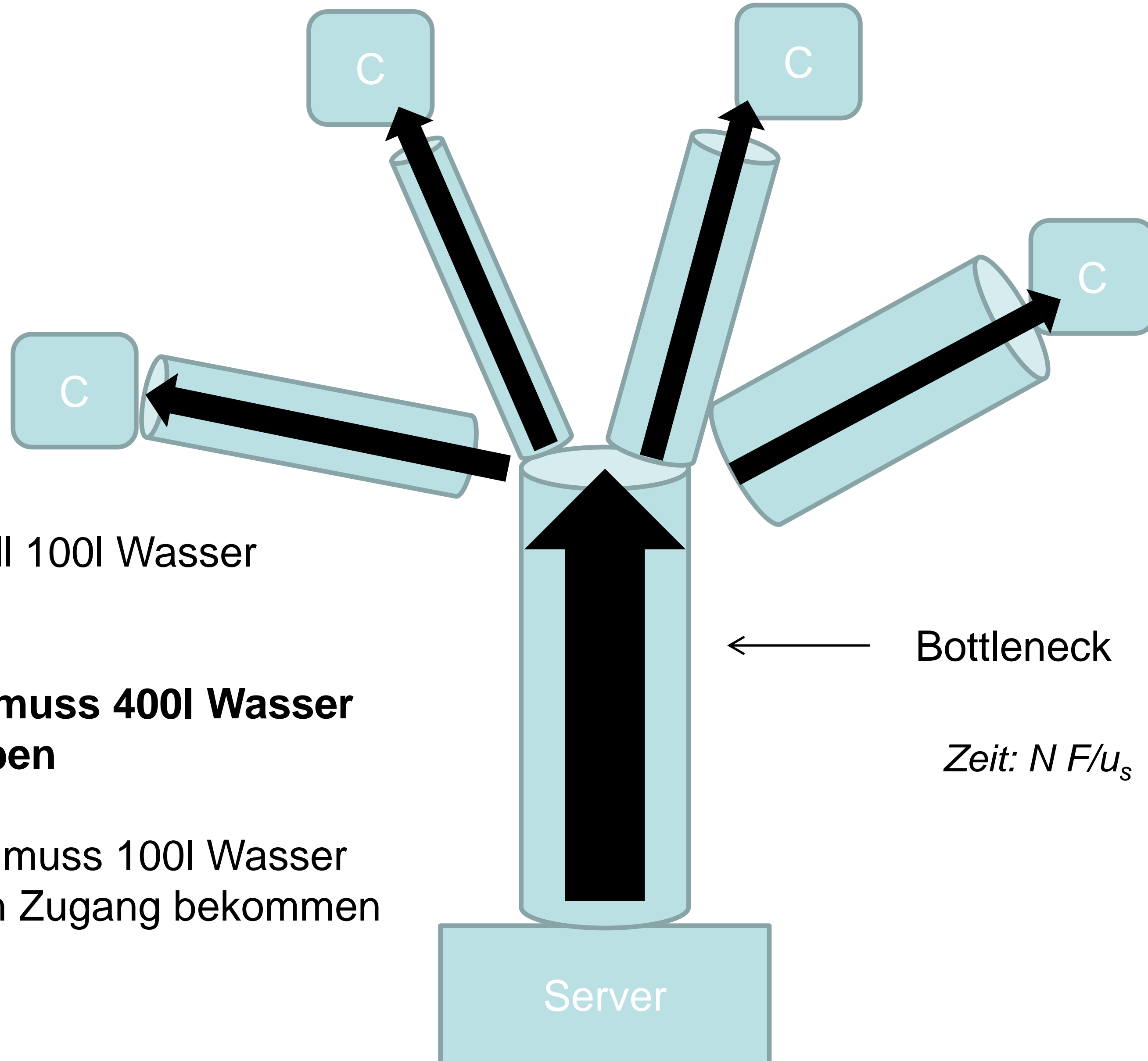
u_i: Bandbreite von Client/Peer i in das Netz

d_i: Bandbreite zu Client/Peer i aus dem Netz



Jeder Client soll 100l Wasser bekommen:

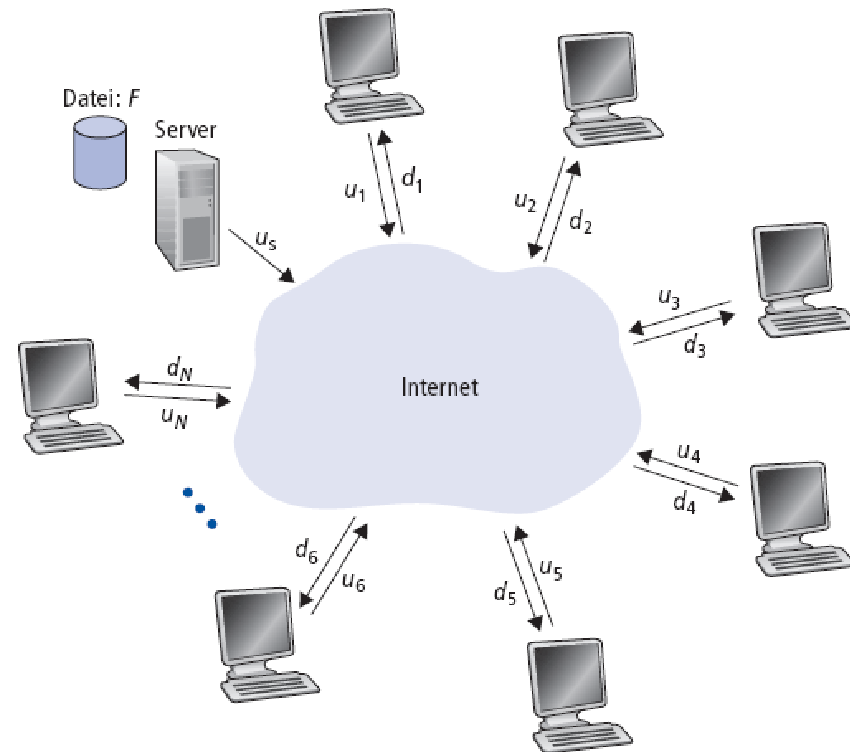
- Der Server muss 400l Wasser hinaufpumpen
- **Jeder Client muss 100l Wasser durch seinen Zugang bekommen**



Jeder Client soll 100l Wasser bekommen:

- **Der Server muss 400l Wasser hinaufpumpen**
- Jeder Client muss 100l Wasser durch seinen Zugang bekommen

2.6.2 Client-Server- VS. P2P-Architektur



Client-Server Architektur:

- Server sendet N Kopien parallel:
  → *Zeit:* $N\,F/u_s$
- Client i benötigt F/d_i Sekunden für den Download

Zeit um die Datei an *N* Computer mittels Client-Server-Architektur zu übertragen:

$$d_{cs} = \max\left\{ N\,F/u_s,\ F/\min(d_i) \right\}$$

Wächst für große N linear mit N!

u_s: Bandbreite vom Server in das Netz

u_i: Bandbreite von Client/Peer i in das Netz

d_i: Bandbreite zu Client/Peer i aus dem Netz

2.6.2 Client-Server- VS. P2P-Architektur

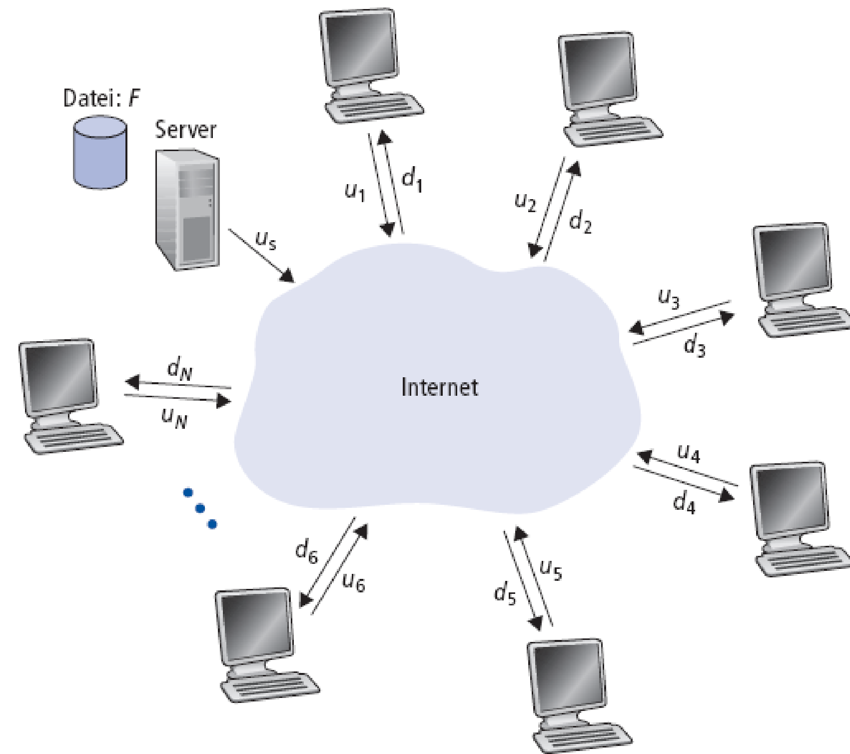
<u>Peer-to-Peer Architektur:</u>

- Server muss eine Kopie senden: F/u_s
- Client i braucht F/d_i Sekunden für den Download
- NF Bits müssen insgesamt heruntergeladen werden
- Höchstmögliche Datenrate ins Netz:

$$u_s + \sum_{i=1,N} u_i$$

Zeit um die Datei an *N* Computer mittels Client-Server-Architektur zu übertragen:

$$d_{p2p} = \max\left\{ F/u_s,\ F/\min(d_i),\ NF/(u_s + \sum_{i=1,N} u_i)\right\}$$

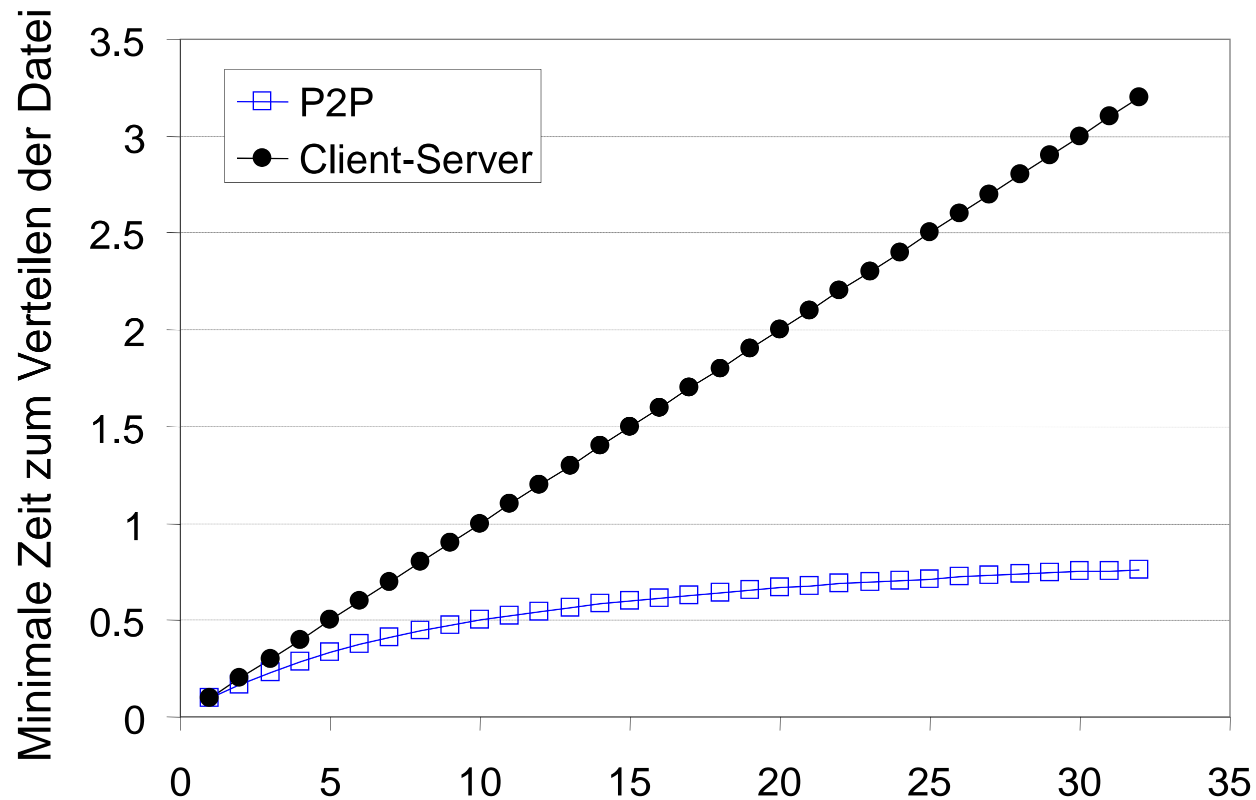


u_s: Bandbreite vom Server in das Netz

u_i: Bandbreite von Client/Peer i in das Netz

d_i: Bandbreite zu Client/Peer i aus dem Netz

2.6.2 Client-Server- VS. P2P-Architektur



2.6.3 Bittorrent

Bestandteile:

- Eine statische „Metainfo“-Datei mit der Endung .torrent
  - Beinhaltet statische Informationen zum Download
- Ein gewöhnlicher Webserver
  - Auf diesem ist die .torrent-Datei abgelegt (und verlinkt)
- BitTorrent-Clients der Endanwender
  - Interpretieren die .torrent-Datei
  - Laden Teile des Downloads von anderen Clients herunter
  - Stellen Teile des Downloads anderen Clients zur Verfügung
  - Sorgen für Fairness
- Ein „Seed“/„Original-Downloader“
  - Wenigstens eine Urquelle für den Download muss bereitgestellt werden
- Ein Tracker
  - Vermittelt die Clients untereinander

2.6.3 Bittorrent

Die .torrent Datei

```
announce: http://tracker.3dgamers.com:6969/announce
creation: date 1132687444
info:
    length=294285337
    name=bf2_v1_12update.exe
    piece length = 262144
    pieces = „59bf6c45....“
```

→ `piece length`: Der Download wird in Teile fester Größe zerlegt. Dieser Wert gibt die Größe dieser Teile an.

→ `pieces`: Für jeden Teil gibt es einen 20-Byte-Hashwert (SHA-1), anhand dessen man die Korrektheit des Downloads überprüfen kann.

2.6.3 Bittorrent

Codierung

- Die .torrent-Datei ist per Bencoding kodiert:
  - Strings: „spam“ -> 4:spam
  - Integer: 3 -> i3e
  - Listen: [2,3] -> li2ei3ee
  - Lexika: {key,42} -> d3:keyi42ee

```
Das Beispiel Bencoded:

d8:announce41:http://tracker.3dgamers.com:6969/announcee
13:creation datei1132687444e4:infod
6:lengthi294285337e4:name19:bf2_v1_12update.exe
12:piece lengthi262144e6:pieces22460:Y¿lE6)4...
```

2.6.3 Bittorrent

Tracker

- Während bei einem Client ein Download läuft, wird in regelmäßigen Abständen der Tracker kontaktiert
- Die Kommunikation erfolgt über HTTP (GET + Response)
- In der GET-Anfrage werden folgende Informationen übertragen:
  - `info_hash:` der Hashwert aus der .torrent-Datei
  - `peer_ID:` eine ID für den Anfragenden, diese wird zufällig bestimmt
  - `IP/Port:` IP-Adresse und Port des Anfragenden
  - `uploaded/downloaded:` übertragenes Datenvolumen
- In der Antwort erhält der Client zwei Informationen:
  - Das Intervall, nach dem er sich wieder beim Tracker melden soll
  - Eine Liste der Kontaktdaten zu anderen Clients

2.6.3 Bittorrent

Client

- Lädt zu Beginn die .torrent-Datei vom Webserver
- Kontaktiert regelmäßig den Tracker

Up-/Download:

- Baut zu allen Peers, die ihm bekannt sind, eine TCP-Verbindung auf
- Die Verbindungen sind bidirektional
- Zuerst wird angekündigt, welche Teile des Downloads der User besitzt
- Erhält er im Laufe der Zeit neue Teile, dann kündigt der Client dies ebenfalls an
- Jede Verbindung hat auf beiden Seiten zwei Zustandsinformationen:
  - 1. `Interest`: ist auf 1 gesetzt, wenn Teile vom Kommunikationspartner heruntergeladen werden sollen (da er Teile hat, die man selbst nicht besitzt)
  - 2. `Choked`: ist auf 1 gesetzt, wenn der Upload auf dieser Verbindung temporär eingestellt werden soll

→ Fairness Algorithmus bei Bittorrent!

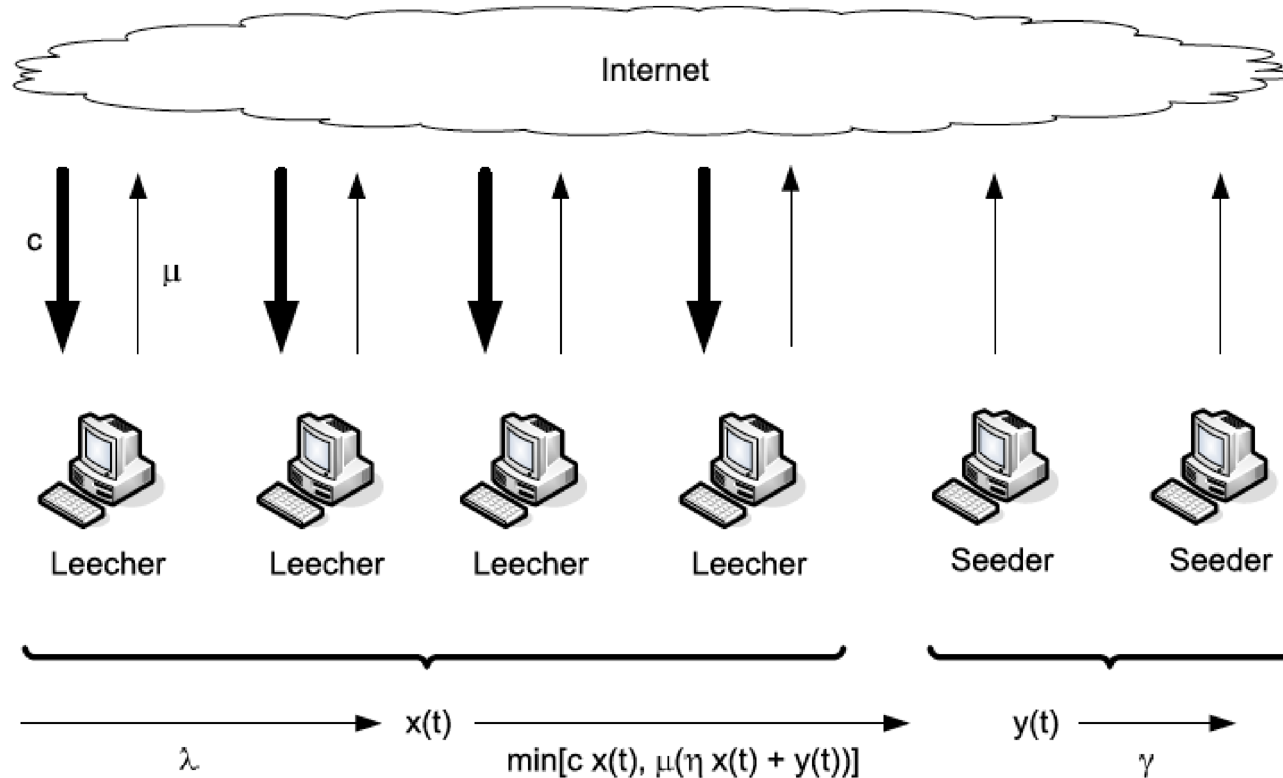
2.6.3 Bittorrent

Bei TCP kann es zu Problemen kommen wenn über mehrere Verbindungen gleichzeitig gesendet wird, deshalb wird mithilfe des Choked-Bits die Anzahl der gleichzeitig aktiven Uploads beschränkt.

Fairness durch Choking Algorithmus

Wie wird das Choked-Bit gesetzt?

- Fairness Strategie: Tit-for-Tat („Wie Du mir, so ich Dir“)
  - Bei den 4 (Standardeinstellung der meisten Clients) Verbindungen von denen man den höchsten Download hat, wird das Choked-Bit gelöscht um eigene Daten an diese Peers zu verteilen.
  - Diese 4 Verbindungen werden alle 10 Sekunden neu bestimmt um zu gewährleisten, dass Verbindungen nicht andauernd gechoked und wieder unchoked werden
- „Optimistic Unchoking“
  - Reihum wird bei jeder Verbindung für 30 Sekunden das Choke-Bit gelöscht um neuen Kommunikationspartnern eine Chance zu geben



Download flow = c x(t)

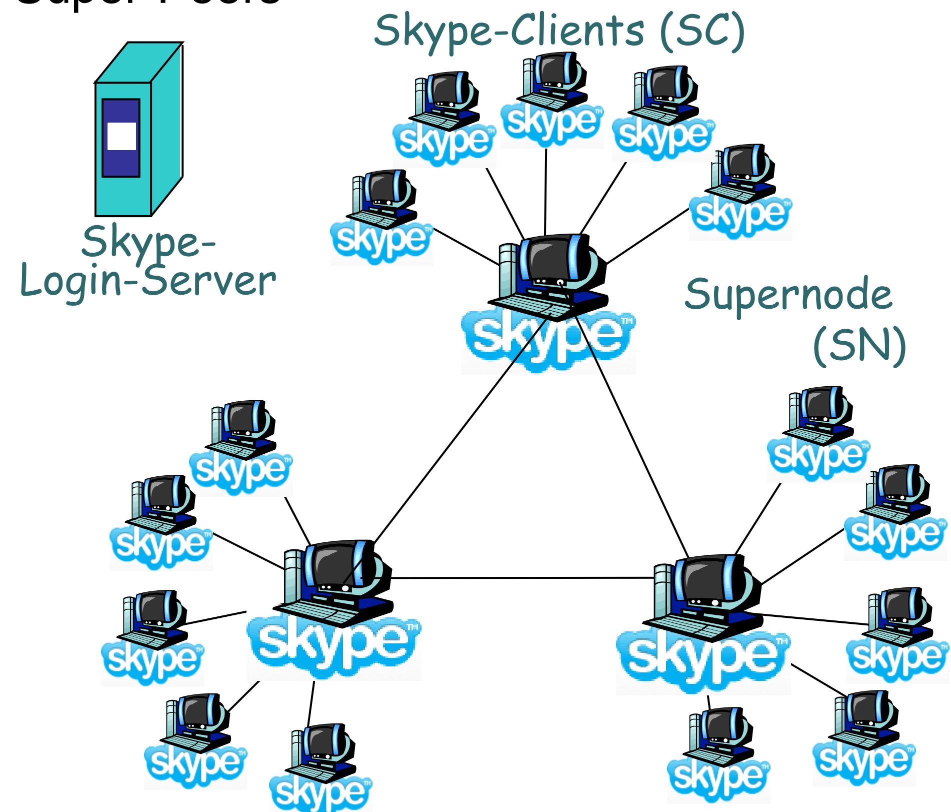
Upload flow = μ(η x(t) + y(t))

$$T = \max\left\{\frac{1}{c}, \frac{1}{\eta}\left(\frac{1}{\mu} - \frac{1}{\gamma}\right)\right\}$$

D. Qiu and R. Srikant. *Modeling and Performance Analysis of BitTorrent-Like Peer-to-Peer Networks*. In Sigcomm'04, pages 367-378, Portland, Oregon, USA, 2004.

2.6.4 Skype

- Skype ist eine P2P-Voice-over-IP- (VoIP) und Instant Messaging-Anwendung (IM)
- Proprietäre Protokolle
- Verschlüsselung aller von Skype übertragenenen Pakete
- Hierarchisches Overlay mit Peers und Super-Peers
- Index, der Skype-Benutzernamen auf aktuelle IP-Adressen und Ports abbildet ist auf Super-Peers verteilt



2.7 Socket-Programmierung

Definition: Socket

Eine Schnittstelle auf einem Host, kontrolliert durch das Betriebssystem, über das ein Anwendungsprozess sowohl Daten an einen anderen Prozess senden als auch von einem anderen Prozess empfangen kann.
Ein Socket ist also eine Art "Tür" zwischen Anwendungsprozess und Transportprotokoll.

Socket-API

- Eingeführt in BSD4.1 UNIX, 1981
- Sockets werden von Anwendungen erzeugt, verwendet und geschlossen
- Client/Server-Paradigma
- Zwei Transportdienste werden über die Socket-API angesprochen:
  - Unzuverlässige Paketübertragung
  - Zuverlässige Übertragung von Datenströmen

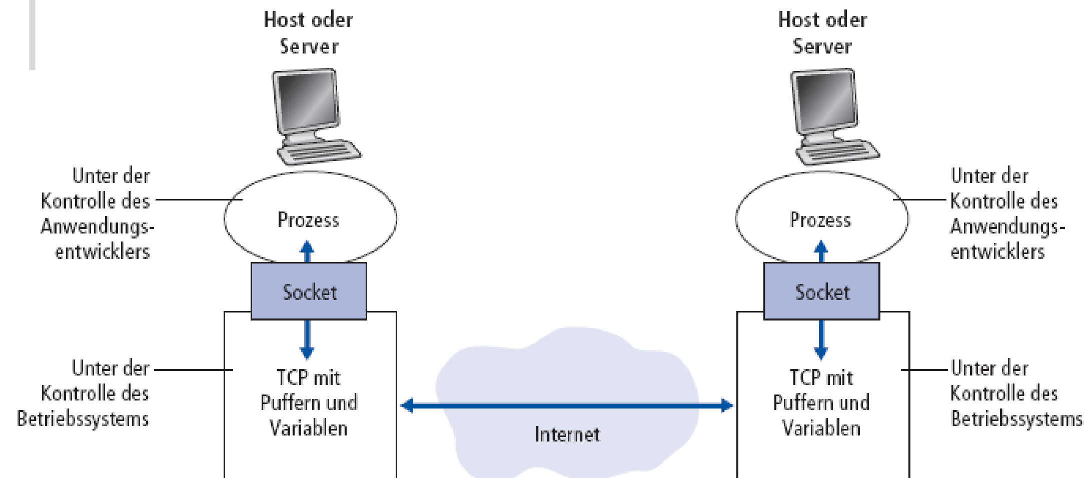
2.7.1 Socket-Programmierung mit TCP

Vorgehen im Server:

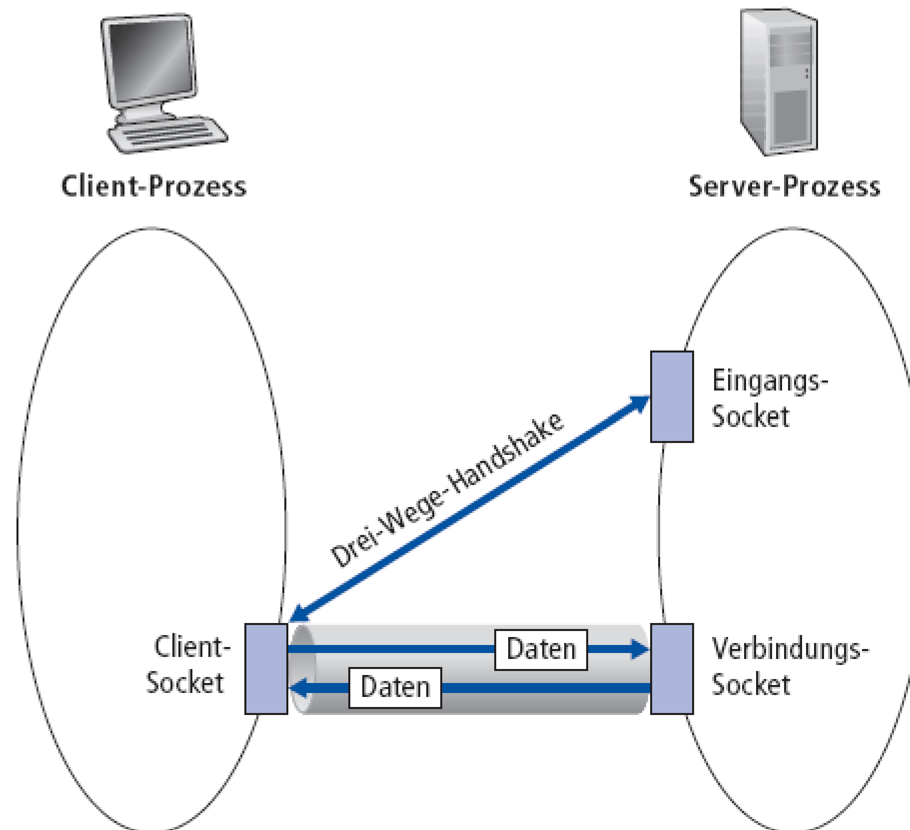
- Server-Prozess muss laufen
- Server muss einen Socket angelegt haben, der Client-Anfragen entgegennimmt
- Wenn der Serverprozess von einem Client kontaktiert wird, dann erzeugt er einen neuen Socket, um mit diesem Client zu kommunizieren
  - So kann der Server mit mehreren Clients kommunizieren
  - Portnummern der Clients werden verwendet, um die Verbindungen zu unterscheiden

Vorgehen im Client:

- Anlegen eines Client-TCP-Sockets
- Angeben von IP-Adresse und Portnummer des Server-Prozesses
- Durch das Anlegen eines Client-TCP-Sockets wird eine TCP-Verbindung zum Server-Prozess hergestellt

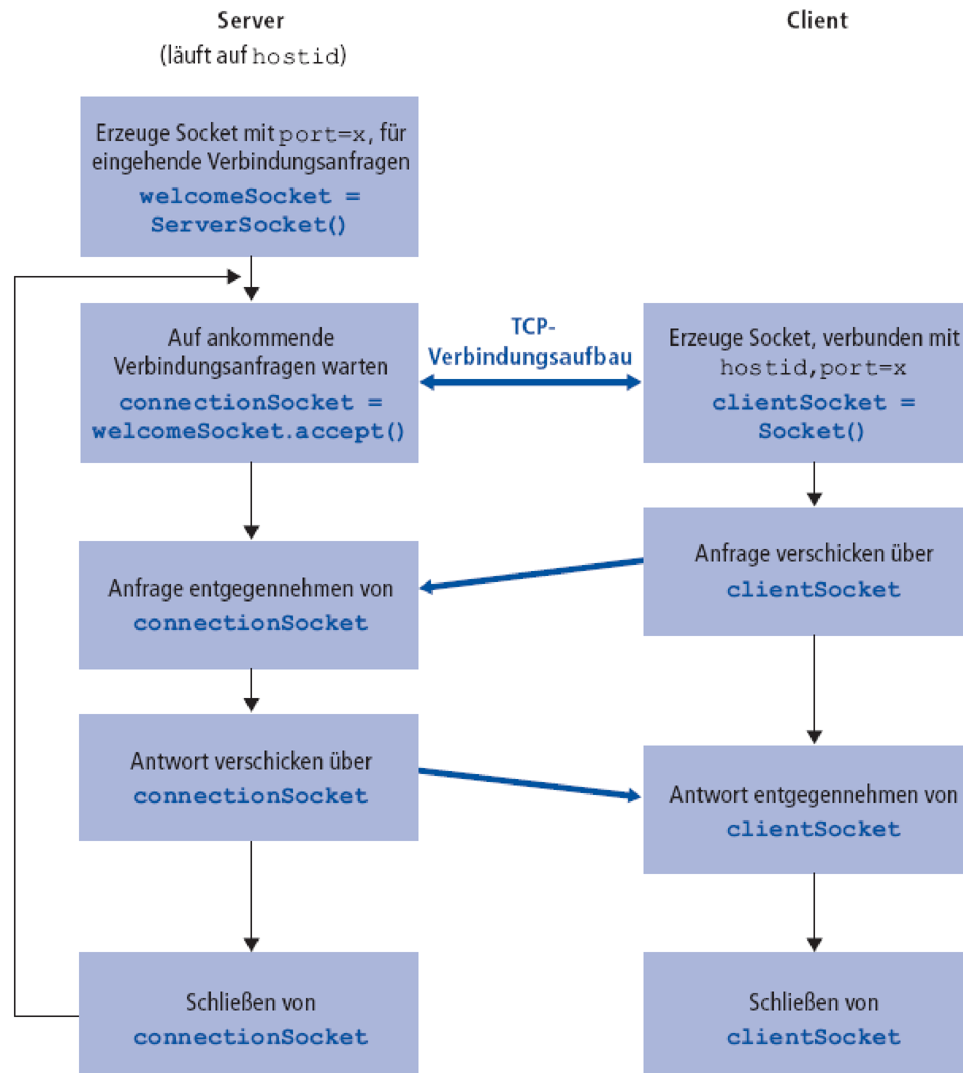


2.7.1 Socket-Programmierung mit TCP



Aus Anwendungsperspektive stellt TCP einen zuverlässigen, reihenfolgeerhaltenden Transfer von Bytes zwischen Client und Server zur Verfügung.

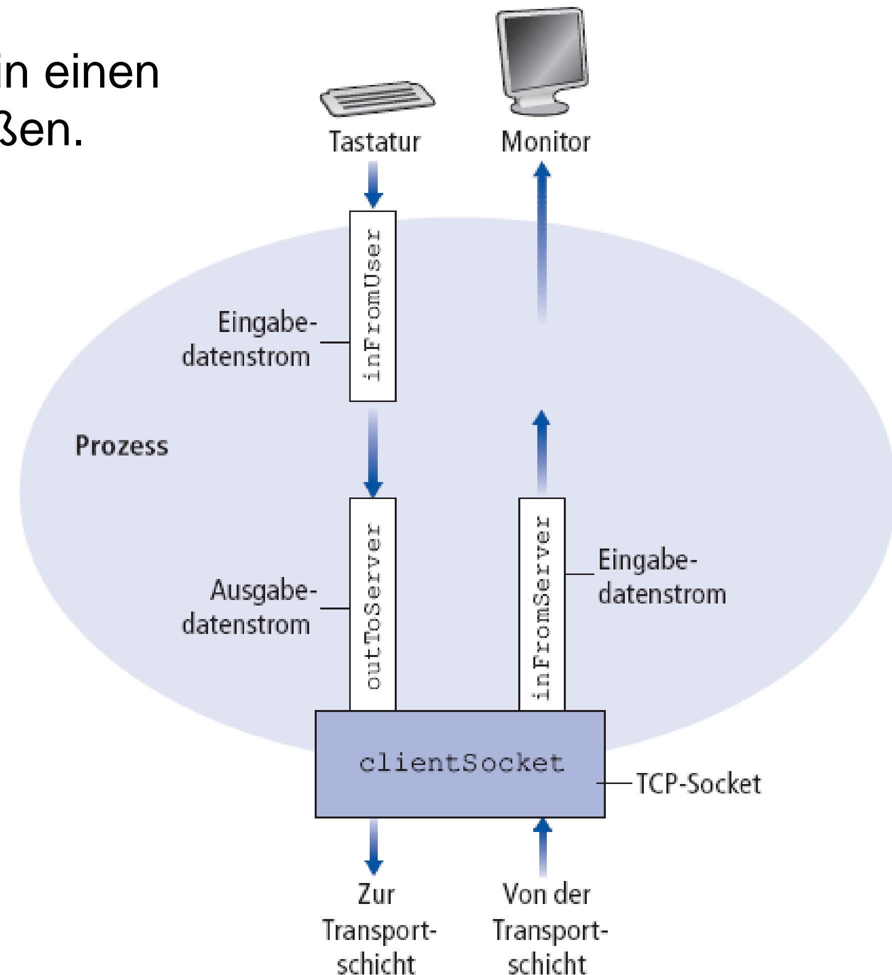
2.7.1 Socket-Programmierung mit TCP



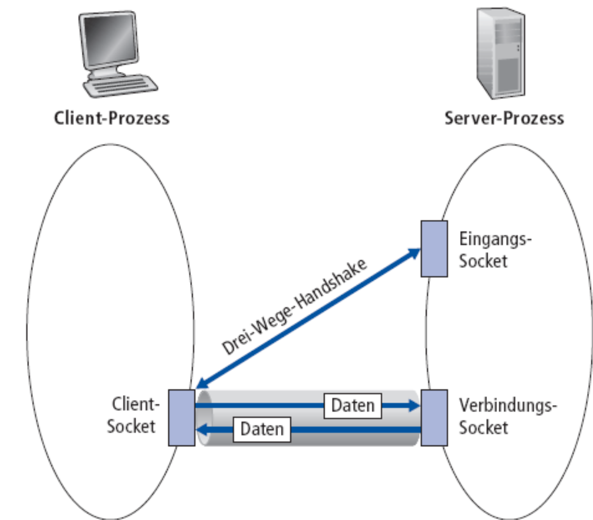
2.7.1 Socket-Programmierung mit TCP

(Daten-) Ströme

- Ein Strom ist eine Folge von Bytes, die in einen Prozess hinein- oder aus ihm hinausfließen.
- Ein *Eingabestrom* ist mit einer Quelle verbunden, z.B. Tastatur oder Socket.
- Ein *Ausgabestrom* ist mit einer Senke verbunden, z.B. dem Monitor oder einem Socket.



2.7.1 Socket-Programmierung mit TCP



Beispiel für eine Client/Server-Anwendung:

1) Client liest Zeilen von der Standardeingabe (`inFromUser` Strom) und sendet diese über einen Socket (`outToServer` Strom) zum Server
2) Server liest die Zeile aus seinem Verbindungs-Socket
3) Server konvertiert die Zeile in **Großbuchstaben** und sendet sie durch seinen Verbindungs-Socket zum Client zurück
4) Client liest die konvertierte Zeile vom Socket (`inFromServer` Strom) und gibt sie auf seiner Standardausgabe (Monitor) aus

2.7.1 Beispiel: Java-Client mit TCP

```
import java.io.*;
import java.net.*;
class TCPClient {

    public static void main(String argv[]) throws Exception
    {
        String sentence;
        String modifiedSentence;

        BufferedReader inFromUser =
          new BufferedReader(new InputStreamReader(System.in));

        Socket clientSocket = new Socket("hostname", 6789);

        DataOutputStream outToServer =
          new DataOutputStream(clientSocket.getOutputStream());

        ...
```

Eingabstrom anlegen → BufferedReader inFromUser

Client-Socket anlegen, mit dem Server verbinden → Socket clientSocket

Ausgabestrom anlegen, mit Socket verbinden → DataOutputStream outToServer

2.7.1 Beispiel: Java-Client mit TCP

Eingabestrom anlegen, mit Socket verbinden

Zeile an Server schicken

Zeile vom Server lesen

```
...
BufferedReader inFromServer =
  new BufferedReader(new
      InputStreamReader(clientSocket.getInputStream()));

sentence = inFromUser.readLine();

outToServer.writeBytes(sentence + '\n');

modifiedSentence = inFromServer.readLine();

System.out.println("FROM SERVER: " + modifiedSentence);

clientSocket.close();

        }
    }
```

2.7.1 Beispiel: Java-Server mit TCP

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

class TCPServer {

  public static void main(String argv[]) throws Exception
    {
     String clientSentence;
     String capitalizedSentence;

     ServerSocket welcomeSocket = new ServerSocket(6789);

     while(true) {

         Socket connectionSocket = welcomeSocket.accept();

        BufferedReader inFromClient =
           new BufferedReader(new
           InputStreamReader(connectionSocket.getInputStream()));
           ...
```

Socket für eingehende Anfragen Anlegen (Port 6789)

Neuer Socket!

An diesem Socket auf Anfragen von Clients warten

Eingabestrom anlegen, mit Socket verbinden

2.7.1 Beispiel: Java-Server mit TCP

```
...

DataOutputStream  outToClient =
  new DataOutputStream(connectionSocket.getOutputStream());

clientSentence = inFromClient.readLine();

capitalizedSentence = clientSentence.toUpperCase() + '\n';

outToClient.writeBytes(capitalizedSentence);

         }
     }
}
```

Ausgabestrom anlegen, mit Socket verbinden

Zeile vom Socket lesen

Zeile an Client schicken

Ende der while-Schleife, auf den nächsten Client warten

2.7.2 Socket-Programmierung mit UDP

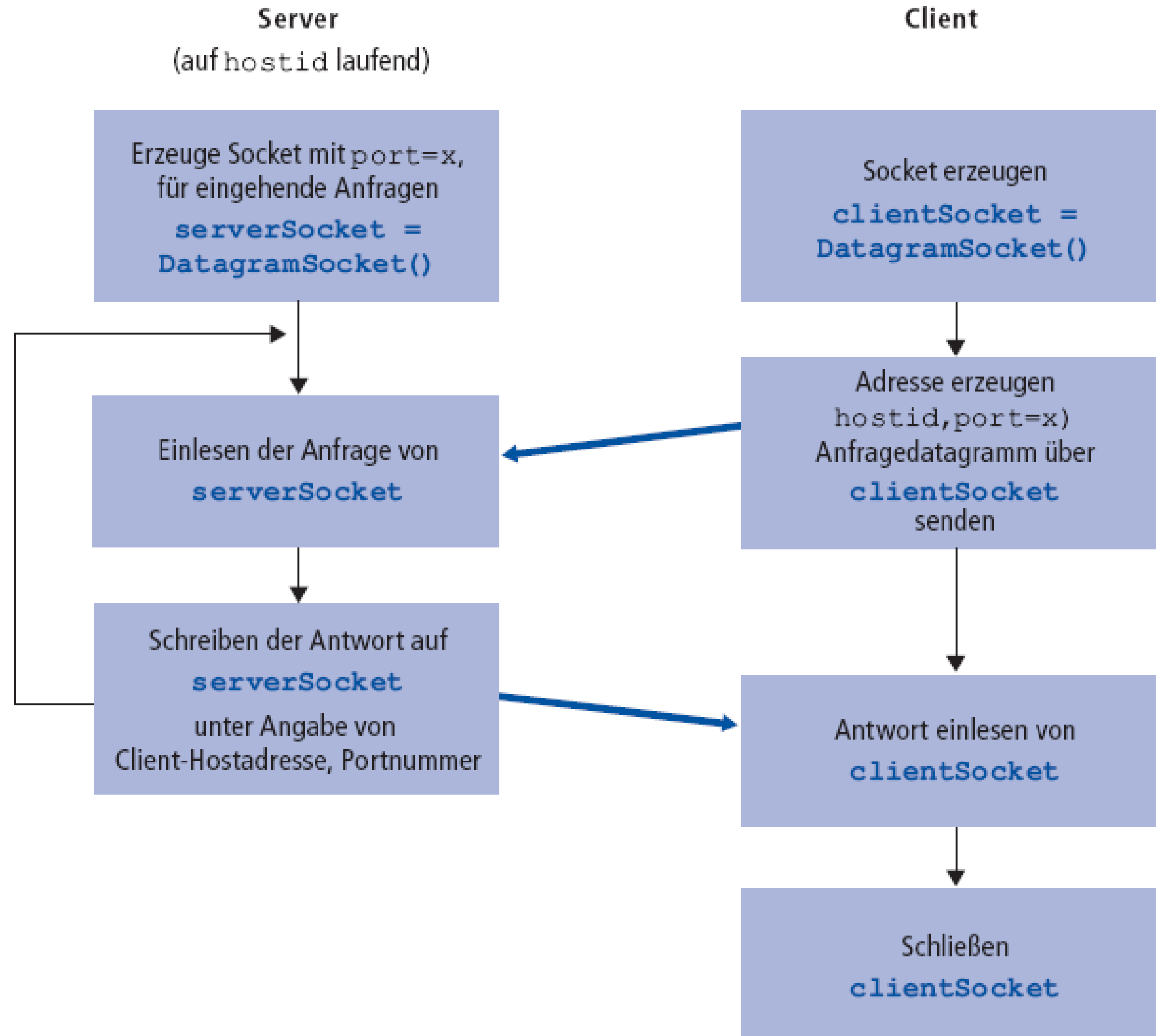
UDP: keine „Verbindung" zwischen Client und Server

- Kein Verbindungsaufbau
- Sender hängt explizit die IP-Adresse und Portnummer des empfangenden Prozesses an jedes Paket an
- Server liest die IP-Adresse und die Portnummer des sendenden Prozesses explizit aus dem empfangenen Paket aus

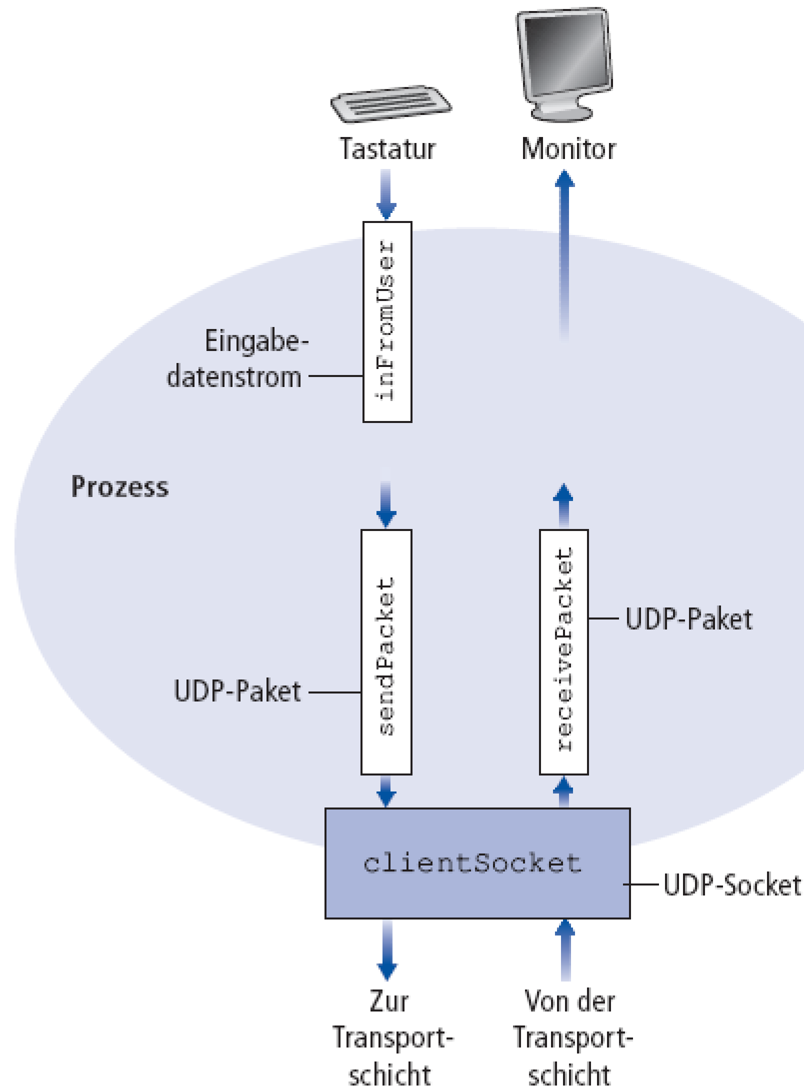
→ Mit UDP können Pakete in falscher Reihenfolge empfangen werden oder ganz verloren gehen!

Aus Anwendungssicht stellt UDP einen unzuverlässigen Transport einer Gruppe von Bytes ("Paket") zwischen Client und Server zur Verfügung.

2.7.2 Socket-Programmierung mit UDP



2.7.2 Beispiel: Java-Client mit UDP



2.7.2 Beispiel: Java-Client mit UDP

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

class UDPClient {
    public static void main(String args[]) throws Exception
    {

      BufferedReader inFromUser =
        new BufferedReader(new InputStreamReader(System.in));

      DatagramSocket clientSocket = new DatagramSocket();

      InetAddress IPAddress = InetAddress.getByName("hostname");

      byte[] sendData = new byte[1024];
      byte[] receiveData = new byte[1024];

      String sentence = inFromUser.readLine();
      sendData = sentence.getBytes();
      …
```

Anlegen eines Eingabestroms → BufferedReader inFromUser

Anlegen des ClientSocket → DatagramSocket clientSocket

Übersetzen von hostname in eine IP-Adresse über DNS → InetAddress IPAddress

2.7.2 Beispiel: Java-Client mit UDP

Paket anlegen: Daten, Länge, IP, Port

Paket an Server schicken

Paket vom Server empfangen

```
...
DatagramPacket sendPacket =
   new DatagramPacket(sendData, sendData.length, IPAddress, 9876);

clientSocket.send(sendPacket);

DatagramPacket receivePacket =
   new DatagramPacket(receiveData, receiveData.length);

clientSocket.receive(receivePacket);

String modifiedSentence =
    new String(receivePacket.getData());

System.out.println("FROM SERVER:" + modifiedSentence);
clientSocket.close();
}
}
```

2.7.2 Beispiel: Java-Server mit UDP

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

class UDPServer {
  public static void main(String args[]) throws Exception
    {

      DatagramSocket serverSocket = new DatagramSocket(9876);

      byte[] receiveData = new byte[1024];
      byte[] sendData  = new byte[1024];

      while(true)
        {

          DatagramPacket receivePacket =
             new DatagramPacket(receiveData, receiveData.length);
           serverSocket.receive(receivePacket);
           ...
```

Datagramm-Socket auf Port 9876 anlegen → `DatagramSocket serverSocket = new DatagramSocket(9876);`

Platz für das zu empfangende Paket reservieren → `DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);`

Paket empfangen → `serverSocket.receive(receivePacket);`

2.7.2 Beispiel: Java-Server mit UDP

```
      String sentence = new String(receivePacket.getData());

      InetAddress IPAddress = receivePacket.getAddress();

      int port = receivePacket.getPort();

      String capitalizedSentence = sentence.toUpperCase();

      sendData = capitalizedSentence.getBytes();

      DatagramPacket sendPacket = new DatagramPacket(sendData,
                         sendData.length, IPAddress, port);

      serverSocket.send(sendPacket);
    }
  }
}
```

IP und Port des Clients bestimmen

Zu sendendes Paket anlegen

Paket über Socket senden

Ende der while-Schleife, auf nächstes Paket warten

Berkeley sockets = C

Beispiel: UDP with C

```
//CLIENT

//includes…

/* Open a datagram socket */
Int sd = socket(AF_INET, SOCK_DGRAM, 0);

if (sd == INVALID_SOCKET) {
   fprintf(stderr, "Could not create socket.\n");
   exit(0);
}

struct hostent *hp;
hp = gethostbyname(“www.host.com“);  //DNS name

struct sockaddr_in server;
memset((void *)&server, '\0', sizeof(struct sockaddr_in));
server.sin_family = AF_INET;
server.sin_port = htons(  6789  );  //port number
server.sin_addr = *((struct in_addr *)hp->h_addr);

char buf[100];
sprintf(buf, "This is packet  1”);
sendto(sd, buf, 100, 0, &server, sizeof(server));

close(sd);
```

```
//SERVER

//includes
…

#define BUFLEN 512
#define NPACK 10
#define PORT  6789

struct sockaddr_in si_me, si_other;
int s, i, slen=sizeof(si_other);
char buf[BUFLEN];

 if ((s=socket( AF_INET, SOCK_DGRAM, IPPROTO_UDP))==-1)
   exit(1);

memset((char *) &si_me, 0, sizeof(si_me));
si_me.sin_family = AF_INET;
si_me.sin_port = htons(PORT);
si_me.sin_addr.s_addr = htonl(INADDR_ANY);
if ( bind(s, &si_me, sizeof(si_me))==-1)
   exit(1);

for (i=0; i<NPACK; i++) {
   if ( recvfrom(s, buf, BUFLEN, 0, &si_other, &slen)==-1)
      exit(1);

   printf("Received packet from %s:%d\nData: %s\n\n",
              inet_ntoa(si_other.sin_addr), ntohs(si_other.sin_port), buf);
}

close(s);
```

Beispiel: TCP with C

//CLIENT

```
…

if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("Socket");
        exit(1);
}

struct hostent *host;
host = gethostbyname(“the.server.com");

struct sockaddr_in server_addr;
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(5000);
server_addr.sin_addr = *((struct in_addr *)host->h_addr);
bzero(&(server_addr.sin_zero),8);

connect(sock, (struct sockaddr *)&server_addr, sizeof(struct sockaddr)) ;

send(sock,send_data,strlen(send_data), 0);

close( sock );
```

//SERVER

```
…

int sock, connected, bytes_received , true = 1;
char recv_data[1024];

if ((sock = socket( AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("Socket");
        exit(1);
}

server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(5000);
server_addr.sin_addr.s_addr = INADDR_ANY;
bzero(&(server_addr.sin_zero),8);

bind(sock, (struct sockaddr *)&server_addr, sizeof(struct sockaddr));

listen(sock, 5);

sin_size = sizeof(struct sockaddr_in);

connected = accept(sock, (struct sockaddr *)&client_addr,&sin_size);

bytes_received = recv(connected,recv_data,1024,0);

close(connected);
close(sock);
```