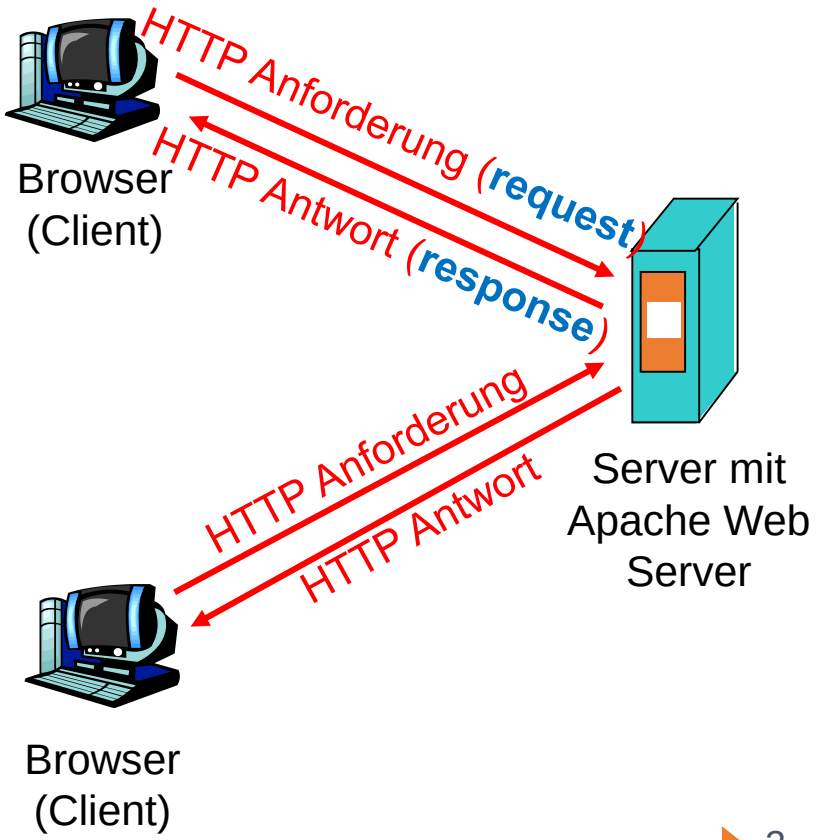
Betriebssysteme und Netzwerke Vorlesung N03

Artur Andrzejak

Fortsetzung:
HTTP - Hypertext Transfer Protocol

- HTTP verwendet TCP
  - Client (Browser) iniitert eine TCP-Verbindung zum Port 80 des Servers
  - Server akzeptiert TCP Verbindung
  - HTTP Nachrichten werden zwischen Browser und Server ausgetauscht
  - TCP Verbindung wird geschlossen
    - Meist vom Client

- HTTP ist **zustandslos**
  - Server behält keinen „HTTP-basierten“ Zustand zwischen Anfragen



Wiederholung: HTTP-Nachrichtenformat

- Zwei Typen von HTTP-Nachrichten
  - Anforderung (**Request**)
  - Antwort (**Response**)
- HTTP-Request-Nachricht
  - Besteht aus einer **Request-Zeile** (Anforderungszeile)
    - 3 Felder: Methoden-, URL- und HTTP-Versionsfeld
  - Sowie **Header-Zeilen** (Kopfzeilen), hier: Webserver-Adresse, Browsertyp, Verbindungstyp, bevorz. Sprache

Request-Zeile → `GET /somedir/page.html HTTP/1.1`

Header-Zeilen

```
Host: www.someschool.edu
User-agent: Mozilla/4.0
Connection: close
Accept-language:fr
```

(extra carriage return, line feed)

Bedingtes GET

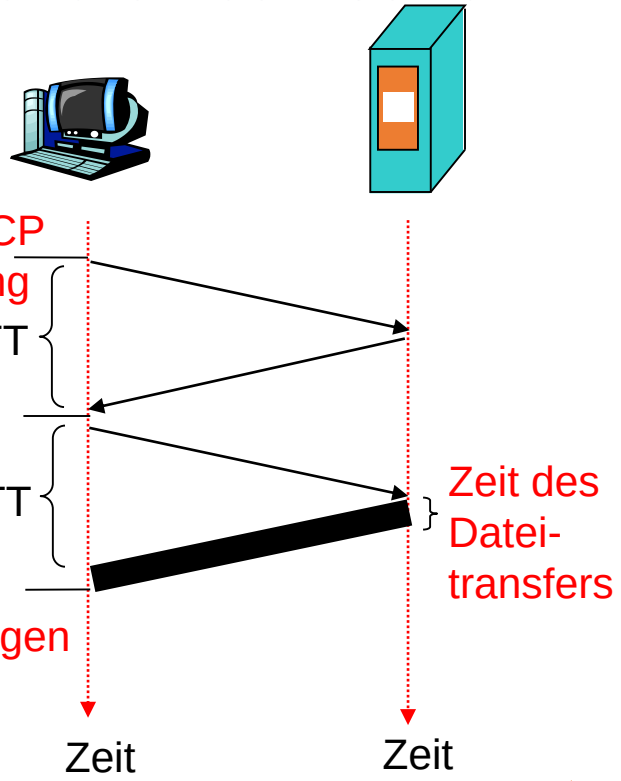
- **Bedingtes GET** ist eine modifizierte HTTP-Request-Nachricht, auf die der Webserver nur dann mit dem Dokumentinhalt antwortet, wenn das Dokument *nach einem bestimmten Zeitpunkt* modifiziert wurde
  - Entlastet Leitungen, wenn Cache-Inhalt noch gültig ist
- Mechanismus: Der Client benutzt in der Request-Nachricht die GET-Methode mit Header-Zeile
  - **If-Modified-Since**:<Datum/Zeit des Dokuments im Cache>
- Falls das Dokument auf dem Server in der Zwischenzeit nicht geändert wurde, antwortet der Server mit:
  - **HTTP/1.1 304 Not Modified**
  - …

HTTP Verbindungstypen

- **Nichtpersistente** Verbindungen
  - Jedes Paar HTTP-Anforderung / HTTP-Antwort wird über eine separate (neu erstellte) TCP-Verbindung geschickt
    - Nach jeder Antwort with die TPC-Verbindung geschlossen
- **Persistente** Verbindungen (Standard bei HTTP)
  - Viele Paare „HTTP-Anforderung/HTTP-Antwort“ werden über dieselbe TCP-Verbindung geschickt
- Vergleich beider Varianten anhand der Zeitdauer von Anforderung bis zum vollständigen Empfang
  - Dazu definieren wir die **Round-Trip-Time** (**RTT**, **Rundlaufzeit**) als die Zeit, die ein Paket vom Client zum Server und zurück benötigt

Antwortzeit bei nichtpersistenen Verbindungen

- Um eine TCP-Verbindung zum Webserver herzustellen, startet der Browser ein **Drei-Wege-Handshake**
  - Browser sendet ein kurzes TCP-Segment an den Server, der Server bestätigt es mit einem kurzen TCP-Segment
  - Dessen Empfang bestätigt wiederum der Client dem Server („Ack_3“)
  - Diese 2 Schritte kosten eine RTT



- Nachdem diese beendet sind, sendet der Client eine Request-Nachricht zusammen mit Ack_3
- Bis zum Empfang des ersten Bytes vergeht ein weiterer RTT
- Dazu kommt noch die Zeit für den Dateitransfer

(Nicht) Persistente HTTP-Verbindungen

- Nichtpersistente Verbindungen
  - Benötigen mindestens **zwei RTT pro Objekt**
  - BS-Overhead für jede TCP-Verbindung
- Persistente Verbindungen
  - Verbindung bleibt nach der Antwort geöffnet
  - Client sendet die Anforderung sobald ein Objekt benötig wird
  - Nach der Anfangsphase braucht ein Paar „Anfrage / Antwort“ **ein RTT**

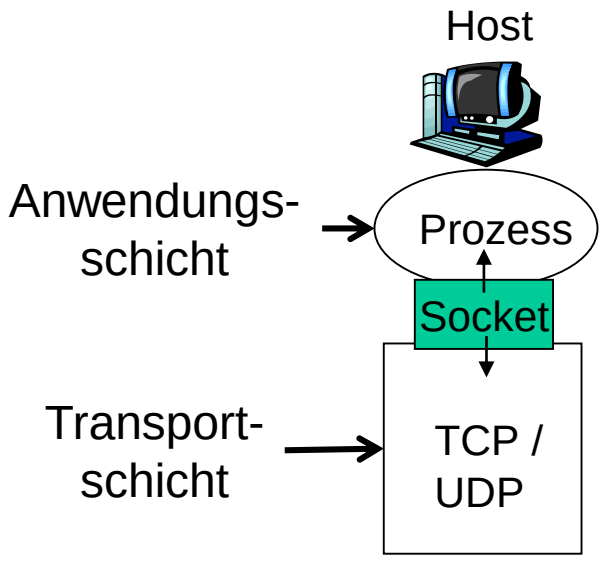
Wiederholung: Internet-Protokollstapel

- Die Gesamtheit der Protokolle aller Schichten bildet den **Protokollstapel** (**protocol stack**)

Name	Funktion	Bsp-Protokolle
Anwendungsschicht (application layer)	Netzwerkanwendungen	HTTP, FTP, SMTP
Transportschicht (transport layer)	Überträgt Nachrichten zwischen BS-Prozessen	TCP, UDP
Netzwerkschicht (network layer)	Leitet die Pakete (Data-gramme) zwischen Hosts	IP, Routing-Protokolle
Sicherungsschicht (data link layer)	Leitet die Pakete zwischen Netzwerkknoten (Routern)	PPP, Ethernet
Bitübertragungss. (physical layer)	Überträgt einzelne Bits zwischen Netzwerkknoten	Hängt vom Medium ab

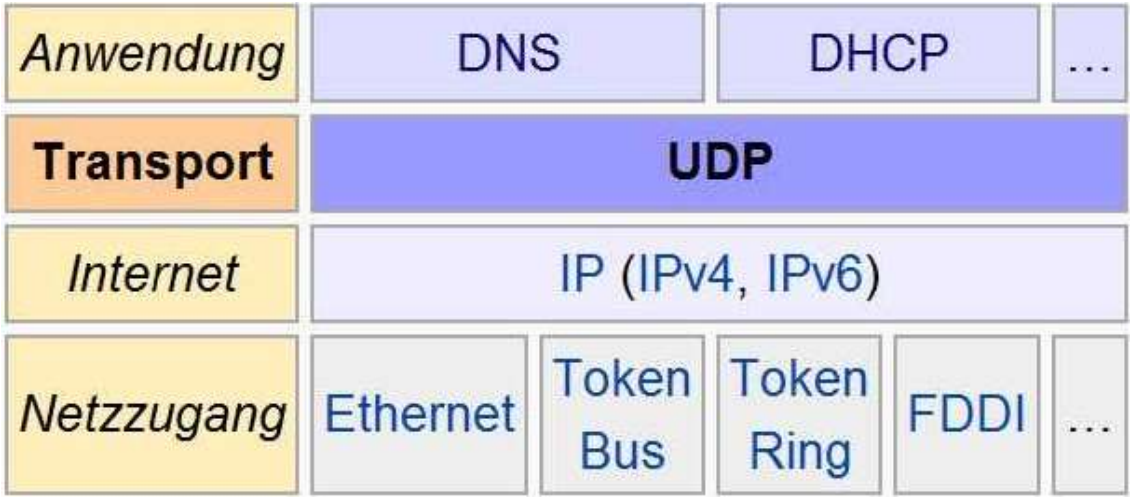
UDP – Protokoll

Achtung: wir sind jetzt bei der **Transportschicht**, d.h. direkt unterhalb der Anwendungs-schicht!



UDP - **User Datagram Protocol** /1

- „Minimales“ Protokoll der Transportschicht
- Verbindungslos; erledigt im Prinzip nur die Prozess-zu-Prozess Adressierung und Übertragung
- Definiert in **RFC 768** von Ravid P. Reed in 1980
- Interessanterweise entwickelt nach TCP, als man ein schnelles Protokoll zur Sprach-übertragung brauchte



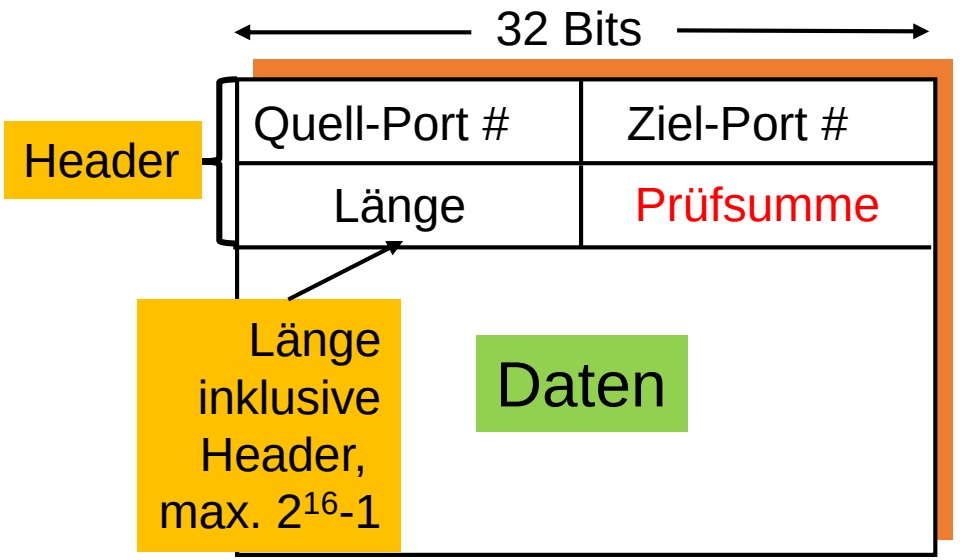
UDP im TCP/IP-Protokollstapel

UDP - User Datagram Protocol /2

- „**Best effort**“-Dienst
  - Verbindungslos, nicht-zuverlässig, ungeschützt
- Was kann schiefgehen?
  - ein Paket kann unbemerkt verlorengehen
  - … mehrmals ankommen
  - Pakete können in falscher Reihenfolge ankommen
  - Keine Gewähr, dass die Daten unverfälscht oder unzugänglich für Dritte eintreffen
- Warum gibt es überhaupt UDP?
  - Kein **Handshake** nötig
    - => Kleinere Verzögerung
  - Kein Verbindungszustand an den Enden
    - Weniger Speicher nötig
  - Kleinerer Header
  - Keine Überlastkontrolle
    - Man kann (theoretisch) so schnell senden wie gewünscht

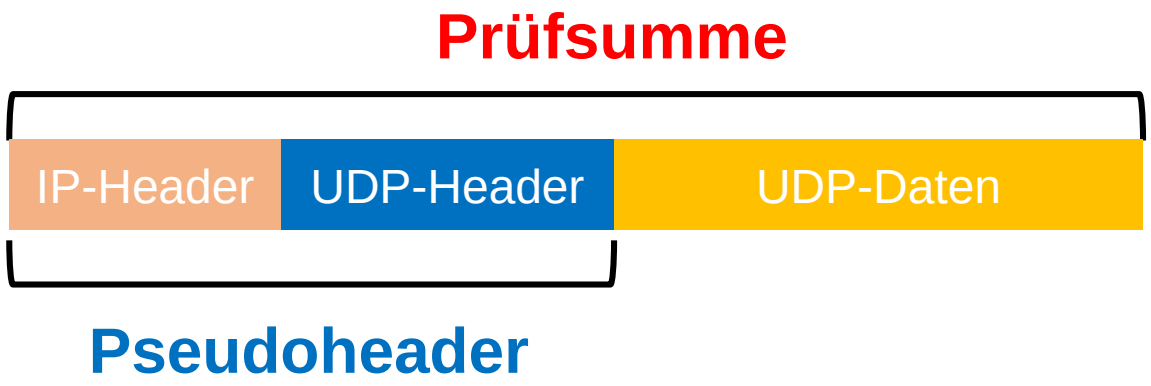
UDP-Paketstruktur

- **Längenfeld**
  - Gibt die Länge des Datagramms, bestehend aus den Daten und dem Header, in Oktetten (= Bytes) an
  - Der kleinstmögliche Wert sind 8 Bytes (d.h. nur Header)
- **Prüfsummenfeld**
  - Enthält eine 16 Bit große Prüfsumme
  - Die Prüfsumme ist optional, wird aber fast immer benutzt
    - Sonst ist Feld auf „0” gesetzt



Prüfsumme und UDP-Pseudoheader

- Die Prüfsumme wird über die eigentlichen Daten und ein sog. **Pseudoheader** gebildet:
  - UDP-Header
  - Einige Daten aus dem IP-Header (tiefere Schicht)
- Daten aus IP-Header: eigentlich eine Verletzung des Schichtenkonzepts!
  - Teile des IP-Headers gehen in die Prüfsumme ein, aber diese Teile werden (in UDP-Prefix) nicht übertragen



UDP-Pseudoheader - Genauer

- Protokoll = Typ des Protokolls, 17 für UDP
- UDP-Länge = Länge des UDP-Headers und der Daten

Bits	0 – 7	8 – 15	16 – 23	24 – 31
0	IP-Quell-Adresse			
32	IP-Ziel-Adresse			
64	0-en	Protokoll	UDP-Länge	
96	Quell-Port #		Ziel-Port #	
128	UDP-Länge (nochmals!)		**Prüfsumme**	
160	Daten			

Teil der Daten aus dem IP-Header

UDP-Header

Prüfsummenberechnung

- Fasse direkt benachbarte Bytes des Pseudoheaders und die des UDP-Paketes zu **16-Bit-Blöcken** zusammen
  - Fülle den letzten Block mit Nullen auf, falls nötig
- Addiere diese Blöcke mit Übertrag zu einer 32-Bit Prüfsumme zusammen; Ergebnis sei **x**
- Addiere die 2 höherwertigen Bytes von **x** zu den 2 niedrigwertigen Bytes auf
  - Falls das Ergebnis >= 2^{16} ist, wiederhole das – warum?
- Wenn diese 16-Bit-Zahl nicht nur aus 1en besteht, dann speichere ihr Einerkomplement im UDP-Header ab

Socket-Programmierung mit UDP

Sockets – Grundlagen /1

- „**Sockets** sind eine plattformunabhängige, standardisierte Schnittstelle (API) zwischen der Netzwerkprotokoll-Implementierung des Betriebssystems und der eigentlichen Anwendungssoftware“ (Wikipedia)
- Sockets arbeiten i.A. bidirektional, d.h. können Nachrichten senden und empfangen
- Jedem Socket wird ein **Port** zugeordnet
  - Eine 16-Bit Integerzahl, „ID“ des Sockets auf dem Host
- Die Entwicklung von Client/Server-Programmen wird als Socket-Programmierung bezeichnet
  - Sockets spielen eine zentrale Rolle in Client / Server-Anwendungen

Sockets – Grundlagen /2

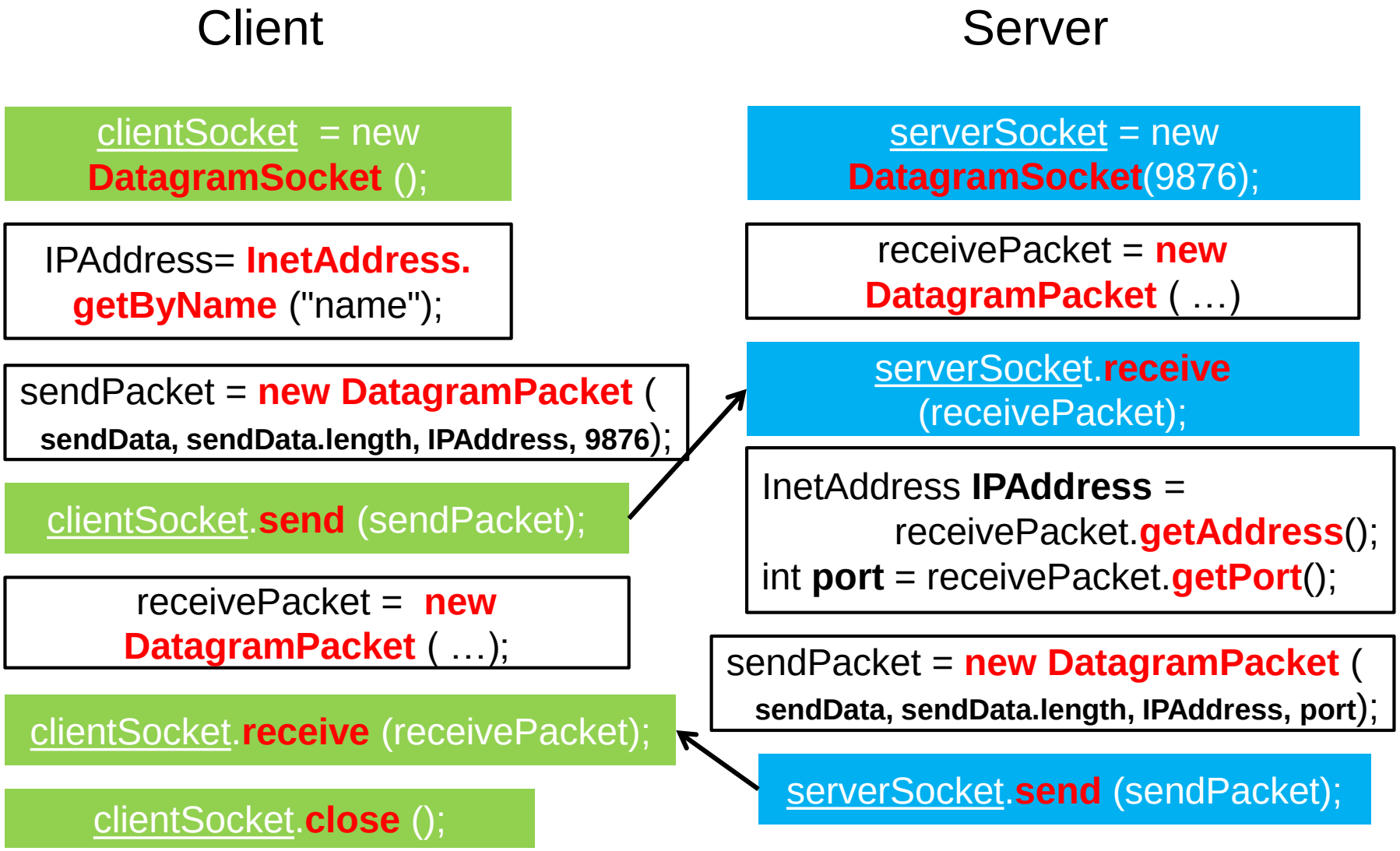
- Eingeführt in BSD4.1 UNIX in 1981
  - Client-Server Modell
- Zwei Typen
  - **Verbindungslos via UDP**
  - **Verbindungsorientiert via TCP**
- Voraussetzungen
  - Server muss laufen und ein Socket erstellt haben, bevor Clients sich mit ihm verbinden können
  - Um sich mit dem Server zu verbinden, muss der Client
    - Einen Socket erstellt haben
    - Die IP-Adresse und die Port-Nummer des Server-Sockets kennen

D.h. wir wählen das Protokoll via die Socketwahl

Sockets mit UDP (Verbindungslos)

- Kein „Handshaking“, da verbindungslos
- Sender fügt **jeder** Nachricht (Segment) die IP-Adresse + Port des Ziels (Empfängers) hinzu
- BS fügt „heimlich“ jedem Paket die IP-Adresse + Port des Senders hinzu
  - Empfänger kann diese Daten extrahieren
- Laufendes Beispiel:
  - **1. Client**
    - Benutzer tippt eine Zeile ein
    - Diese wird an den Server geschickt
  - **2. Server**
    - Empfängt die Zeile
    - Wandelt den Text (Zeile) in Großbuchstaben um
    - Schickt das zurück
  - **3. Client**
    - Empfängt die modifizierte Zeile
    - Zeigt sie auf dem Bildschirm an

Übersicht UDP Kommunikation (Java)



Java-Client (UDP)

Erzeuge Input-Stream

Erzeuge Datagramm Socket

Übersetze den Servernamen zu einer IP-Adresse durch den DNS-Dienst

```
import java.io.*;
import java.net.*;
class UDPClient {
    static string server = “www.my-nice-server.de”;
    public static void main(String args[]) throws Exception
    {
      BufferedReader inFromUser =
        new BufferedReader(new InputStreamReader(System.in));

      DatagramSocket clientSocket = new DatagramSocket();

      InetAddress IPAddress=InetAddress.getByName(server);

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

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

Java-Client (UDP) /2

Erzeuge ein Datagramm mit:
1. Daten
2. Länge der Daten
3. IP-Adresse
4. Port (des Ziels)

Sende das Datagramm

Empfange die Antwort des Servers

```
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 ();
}
}
```

Blockierender Aufruf

Java-Server (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);
```

Erzeuge Datagramm Socket am lokalen Port 9876

Erzeuge Speicher für den zu empfangenen Datagramm

Empfange Datagramm

Blockierender Aufruf

Java-Server (UDP) /2

Ermittle die IP-Adresse und Port-Nr. des Senders

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

InetAddress IPAddress = receivePacket.getAddress();

int port = receivePacket.getPort();
```

```
String capitalizedSentence = sentence.toUpperCase();
```

Der eigentliche „Dienst“ des Servers

Erzeuge Datagramm zum Senden

```
sendData = capitalizedSentence.getBytes();

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

Schicke Datagramm an den Client

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

Ende der Endlosschleife, warte auf das nächste Datagramm

Datagramm (UDP) Kommunikation in C

Sender (Client)

```
s = socket (AF_INET,
           SOCK_DGRAM, 0)
...
bind (s, ClientAddress)
...
sendto (s, "message",
           ServerAddress)
```

Empfänger (Server)

```
s = socket (AF_INET,
           SOCK_DGRAM, 0)
...
bind (s, ServerAddress)
...
amount = recvfrom (s, buffer, from)
```

link

socket()

AF_INET – Kommunikationsbereich ist Internet

SOCK_DGRAM – Socket-Typ ist für Datagram-Socket (d.h. “UDP”)

0 – Das System wählt das Protokoll zu Datagram (d.h. hier UDP)

ServerAddress und ClientAddress sind vom Typ **struct sockaddr_in**

Enthalten Host-IP-Adresse bzw. DNS-Namen und die Portnummer:

```
ServerAddress -> sin_family = AF_INET;
ServerAddress ->sin_port = htons (port);
hostinfo = gethostbyname (hostname);
ServerAddress ->sin_addr = *(struct in_addr *) hostinfo->h_addr;
```

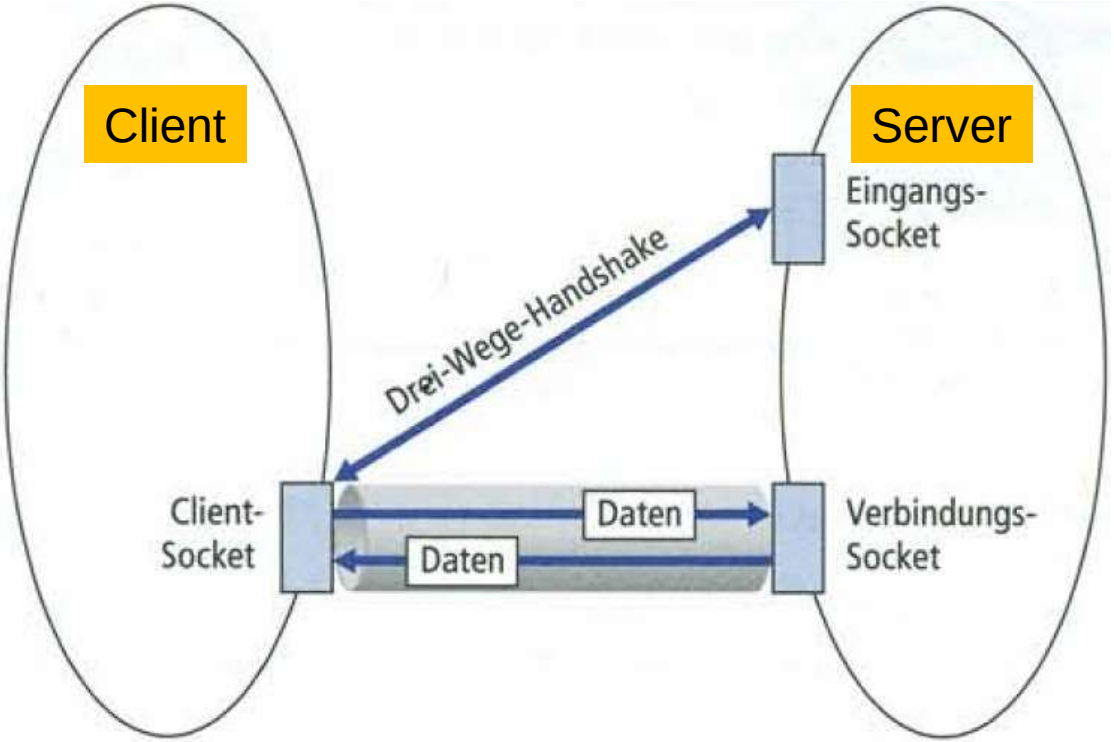
```
#include <sys/socket.h>
#include <netinet/in.h>
```

Socket-Programmierung mit TCP

TCP-Verbindung als ein „virtuelles Rohr“

- Eine TCP-Verbindung verhält sich wie ein *virtuelles Rohr* zwischen zwei Sockets (d.h. wie ein Pipe)
- TCP bietet einen **zuverlässigen Bytestrom-Dienst** zwischen Client-und Server-Prozessen an

- TCP garantiert, dass jedes vom Client versandte Byte in der ursprünglichen Reihenfolge den Server-Prozess erreicht



Sockets mit TCP – das Verbinden

- 1. Server erstellt einen **Eingangssocket** (**welcome socket**) an einem bekannten Port
- 2. Client erstellt eine TCP-Verbindung zum Server, indem er (lokal) einen Socket erstellt
  - *Dabei muss er schon die IP-Adresse + Port (des Eingangssockets) des Servers angeben*
- 3. Wenn angesprochen von einem Client, erstellt der Server einen neuen **Verbindungs-Socket** (**connection socket**) nur für diese Verbindung

Laufendes Beispiel mit TCP

Server (läuft auf IP-Adresse **`hostid`**)

Client

Erzeuge Socket mit Port = **`x`** für eingehende Verbindungsanfragen:
welcomeSocket = **ServerSocket** (**`x`**)

Warte auf ankommende Verbindungsanfragen:
connectionSocket = welcomeSocket.accept()

Daten annehmen von connectionSocket

Antwort verschicken über connectionSocket

Schließen von connectionSocket

TCP-Verbindungs-aufbau

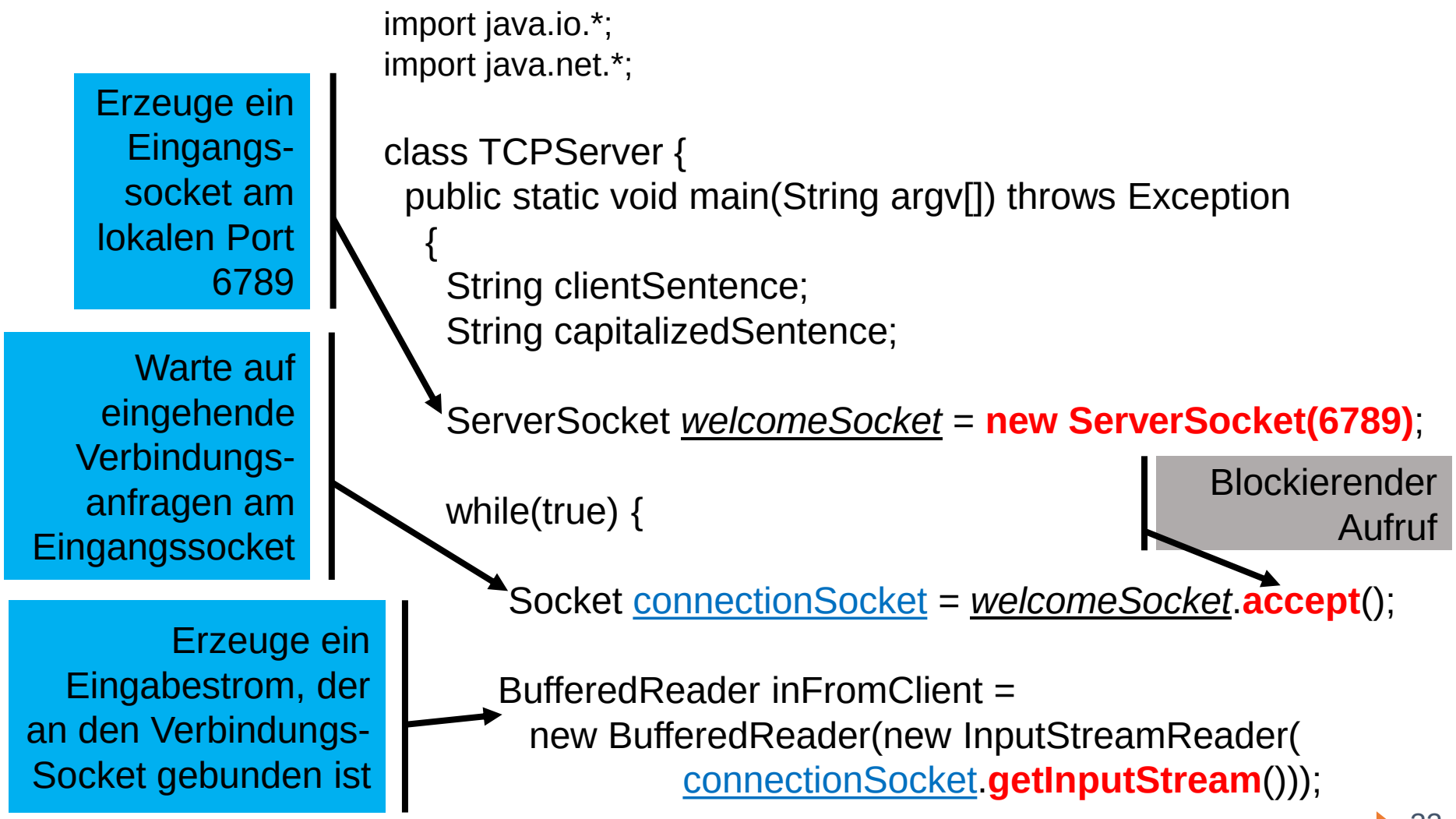
Erzeuge Socket verbunden zu **`hostid`**, port=**`x`**
clientSocket = **Socket** (hostid, x);

Daten verschicken über clientSocket

Antwort entgegennehmen von clientSocket

Schließen von clientSocket

Java-Server (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()));
```

Java-Server (TCP) /2

Erzeuge ein Ausgabestrom zum Senden der Daten über den Socket

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

Empfange eine Zeile vom Client

```
clientSentence = inFromClient.readLine();

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

Sende das Ergebnis an den Client

```
outToClient.writeBytes( capitalizedSentence );
        }
    }
}
```

Ende der Endlosschleife, warte auf die nächste Verbindung

TCP Anmerkungen

- Server hat zwei Arten von Sockets (Java Klassen):
  - **ServerSocket** (hier: *welcomeSocket*) und **Socket** (hier: connectionSocket)
  - Bei TCP erzeugt *welcomeSocket* bei jeder neuen Verbindungs-annahme ein neues Verbindungs-Socket vom Typ **Socket**
- Können mehrere Clients den gleichen Server (zur selben Zeit) verwenden, ggf. mit je mehreren TCP-Verbindungen?
- Im Prinzip ja, aber das obige Prg ist nicht multi-Threaded, also wird nur Client auf einmal abgearbeitet
- Für die Kommunikation mit mehreren Clients muss nach jedem welcomeSocket.accept() ein neuer Thread erzeugt werden, der den entsprechenden Verbindungs-Socket behandelt

Portnummern und Sockets

- Kann die gleiche Portnummer (unter derselben IP-Adresse) an mehrere Sockets vergeben werden?
- UDP: NEIN
  - Wenn man **DatagramSocket**(x) mit schon vergebener Portnummer x aufruft, gibt es einen Fehler (Exception)
- TCP: TEILWEISE
  - NEIN: bei **ServerSocket** (d.h. welcome-Socket des Servers) und **Socket** des Clients: immer eindeutige, neue Portnummer nötig
  - JA: bei den automatisch generierten connectionSockets sind gleiche Portnummern möglich
    - D.h. bei Socket connectionSocket = *welcomeSocket*.**accept**();

TCP-Kommunikation in C

The argument addr is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. (man page)

Sender (Client)

```
s = socket (AF_INET,
           SOCK_STREAM, 0)
...
connect (s, ServerAddress)
...
write (s, "message",  length)
```

Empfänger (Server)

```
s = socket (AF_INET,
           SOCK_STREAM, 0);
...
bind (s, ServerAddress);
listen (s,5);
sNew = accept (s, ClientAddress);
...
n = read (sNew, buffer, amount)
```

Hier wird ein neuer Socket (Referenz „sNew“) erzeugt

socket():
SOCK_STREAM - Socket-Typ für Kommunikation via Datenströme (d.h. TCP)

listen(s, N)
N – Maximale Anzahl der Nachrichten, die an diesem Socket gepuffert werden (= Länge der Eingangsschlange)

ServerAddress und ClientAddress sind vom Typ **struct sockaddr_in**

Video: Sockets in Python

- Programmierung von Sockets mit Threads in Python
- Python 3 Programming Tutorial - Sockets: client server system [N03a]
  - https://www.youtube.com/watch?v=WrtebUkUssc
  - Von 2:00 bis ca. 8:00 (min:sec)

Demultiplexing / Multiplexing

Was ist Demultiplexing / Multiplexing?

- Herausforderung: ankommende Nachrichten müssen an den richtigen Socket gelangen
  - Ein Host kann viele aktive Sockets besitzen
- Analogie: Institut für Informatik (=Host) empfängt täglich einen Korb mit Post
  - Diese Briefe / Pakete müssen an die richtigen Büros (=Sockets) verteilt werden
- Diese Aufgabe der Transportschicht nennt man **Demultiplexing**
  - Die Transportschicht nutzt dafür Informationen in den Headern (u.a. Portnummern)
- Die Umkehrung nennt man **Multiplexing**
  - Sammeln der Datenblöcke von verschiedenen Sockets, verkapseln und verschicken aller über eine Netzwerkkarte

Demultiplexing / Multiplexing

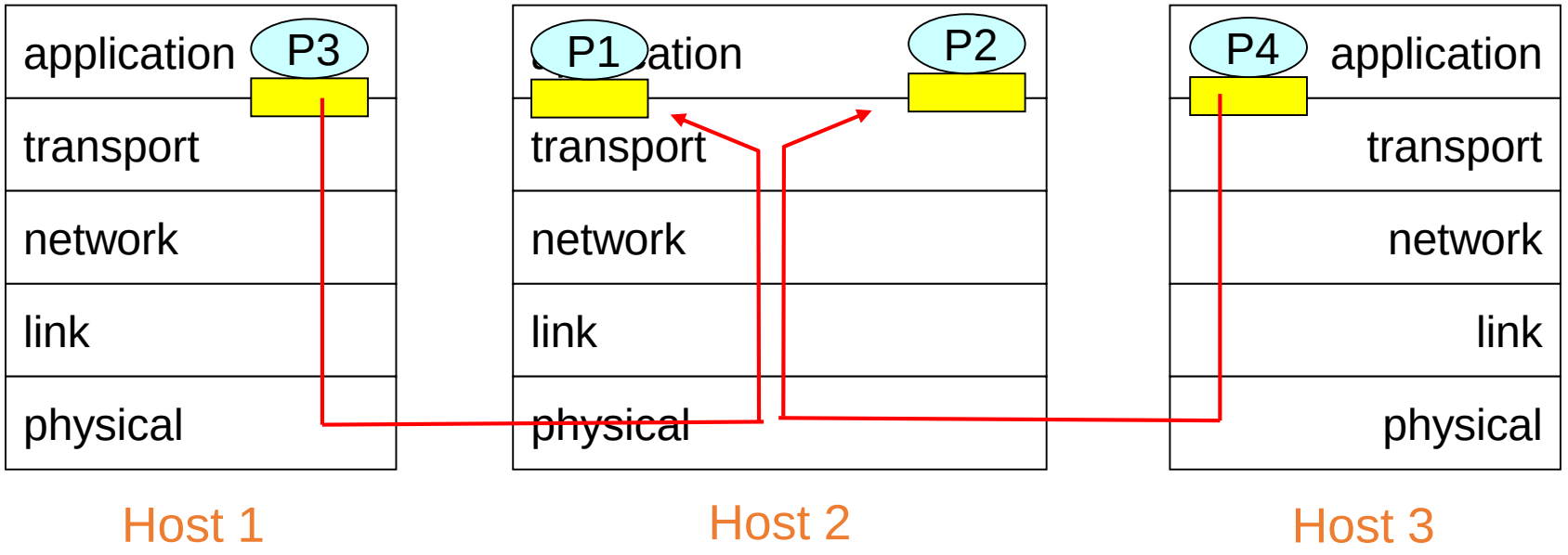
Demultiplexing beim Empfang:

Abliefern der emfangenen Segmente an den richtigen Socket

Multiplexing beim Senden:

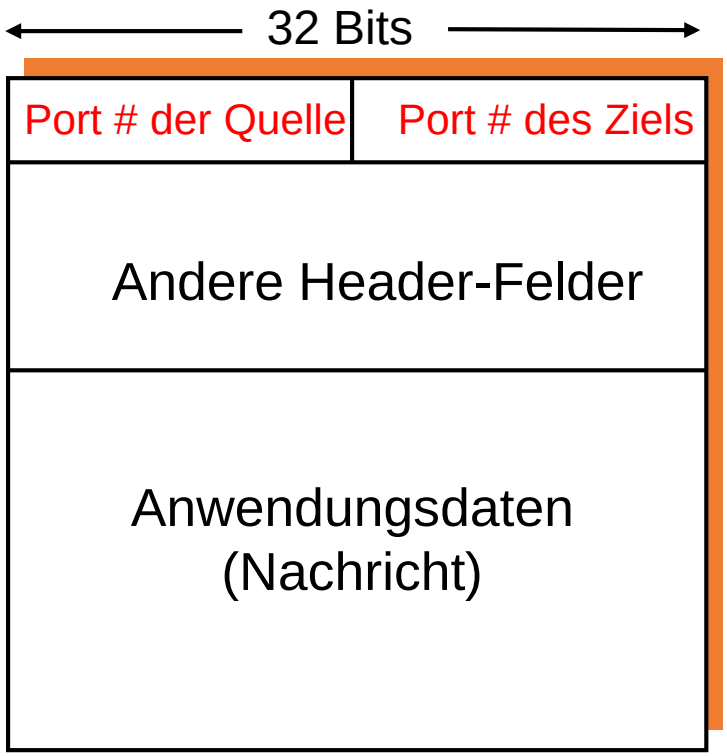
Sammeln der Daten von verschiedenen Sockets und das Verkapseln dieser (mit Headern) zu Segmenten

= Socket = Prozess



Wie funktioniert das Demultiplexing?

- Ein Segment der Transportschicht hat u.a.:
  - **Portnummernfeld der Quelle** (source port number field)
  - **Portnummernfeld des Ziels** (destination port number field)
- Demultiplexing könnte wie folgt funktionieren:
  - Jeder Socket im Host bekommt eine Portnummer zugeteilt
  - Der Wert des Portnummernfelds des Ziels entscheidet, an welchen Socket das Segment geht
- Genauso wird es bei UDP gemacht!



Generelles Format eines TCP/UDP Segments

Verbindungsloses Demultiplexing (UDP)

- Also: das „Ziel“-UDP-Socket ist vollständig durch das Paar identifiziert:

  (IP-Adresse des Ziels, Portnummer des Ziels)

- Wenn ein Host ein UDP – Segment empfängt:
  - Die Transportschicht liest die Zielportnummer P ab
  - Und leitet das Segment an den UDP-Socket zu P weiter
    - Die IP-Adresse (des Ziels) ist nicht mehr wesentlich

- Merke: Segmente mit verschiedenen Quell-IP-Adressen und/oder verschiedenen Quell-Portnummern landen bei dem selben Zielsocket

- Übrigens (Java):
  - Wenn ein Datagramm-Socket ohne Parameter erzeugt wird, teilt ihm die Transportschicht eine Portnummer zu

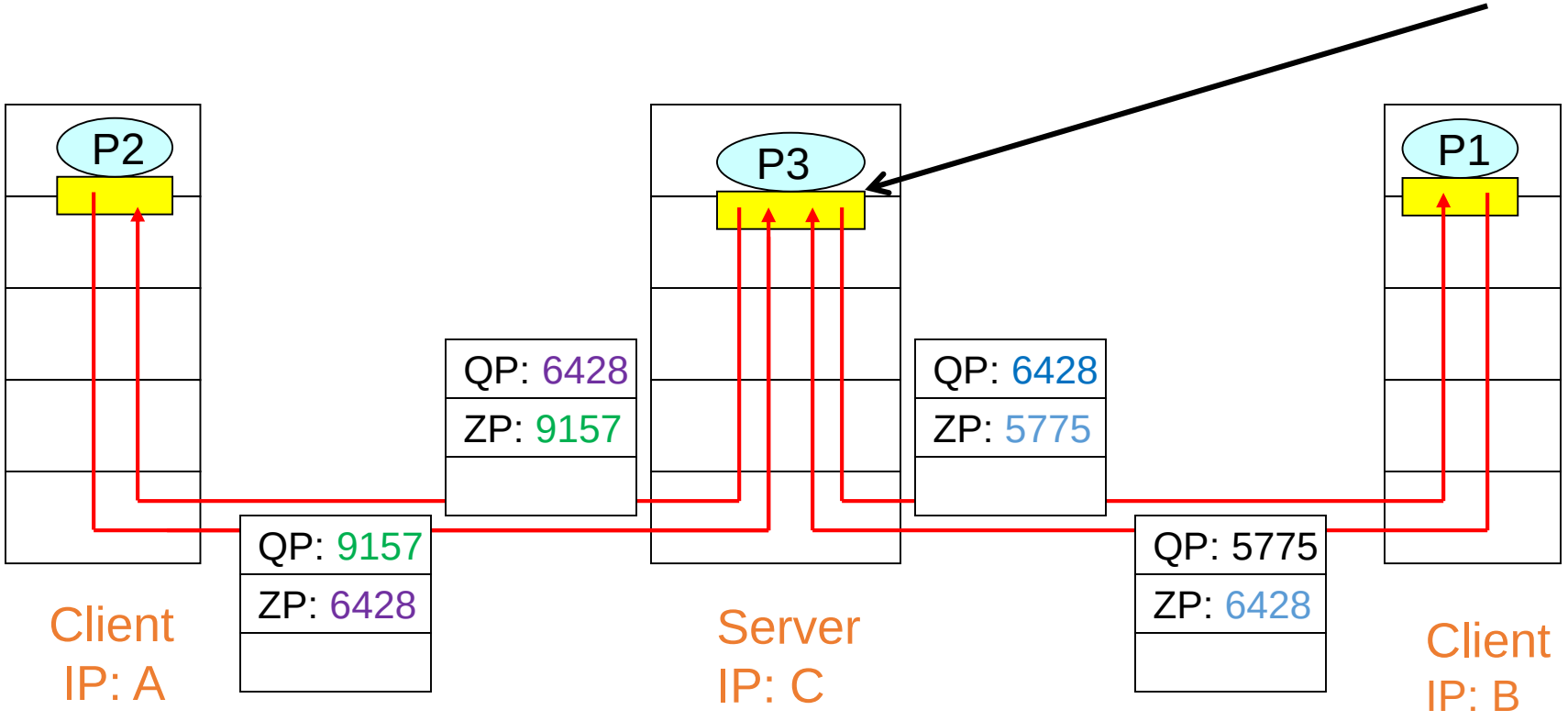
    ```
    DatagramSocket mySocket1 =
    new DatagramSocket();
    ```

  - Wir können aber auch eine Portnummer explizit zuweisen

    ```
    DatagramSocket mySocket2 =
    new DatagramSocket(12535);
    ```

Verbindungsloses Demultiplexing /2

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```



Welchen Zweck hat dann die Portnummer der Quelle (hier QP)?

Sie gibt uns die "Rücksendeadresse" – siehe Java-Sockets mit UDP

Wiederholung: Sockets mit TCP

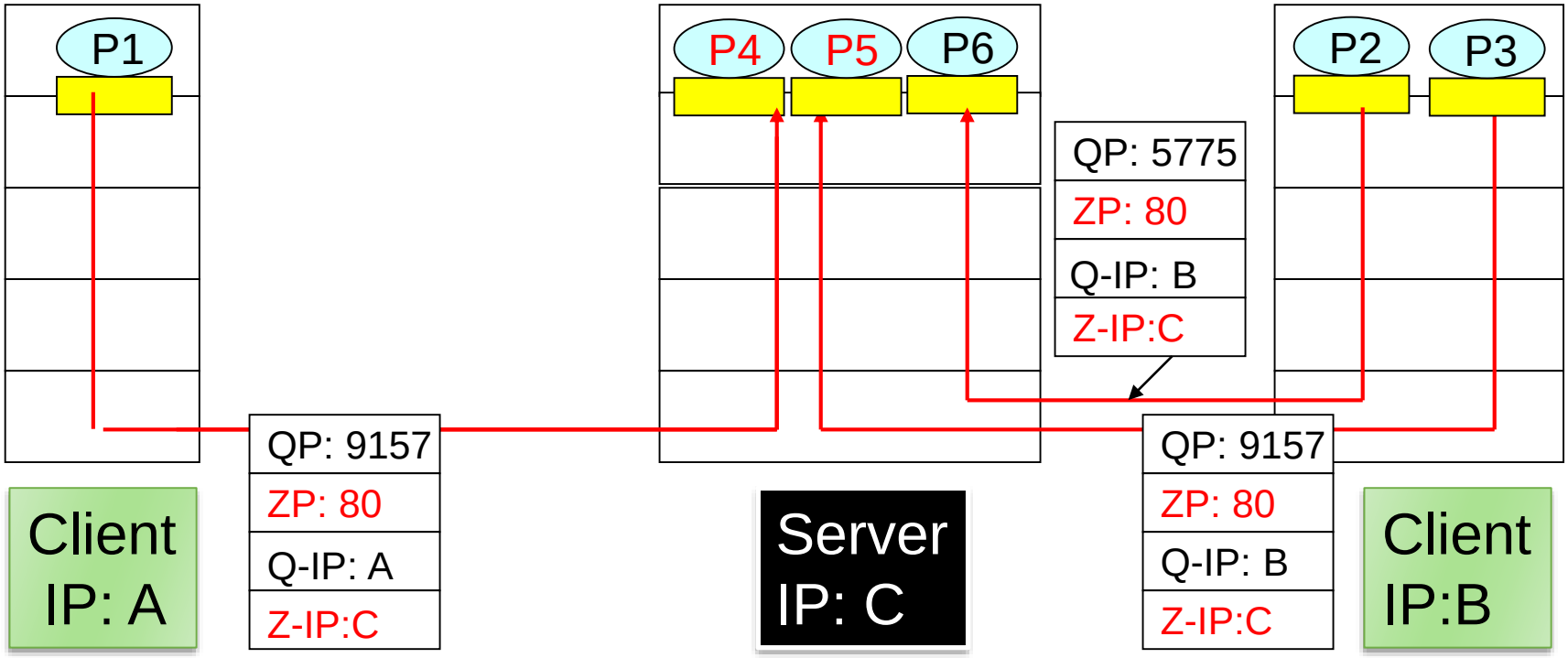
- 1. Server erstellt einen **Eingangssocket** (**welcome socket**) an einem bekannten Port
- 2. Client erstellt eine TCP-Verbindung zum Server, indem er (lokal) einen Socket erstellt
- 3. Wenn angesprochen von einem Client, erstellt der Server einen neuen **Verbindungs-Socket** (**connection socket**) nur für diese Verbindung

Welche Portnummer hat dieses (automatisch erstellte) Verbindungs-Socket?

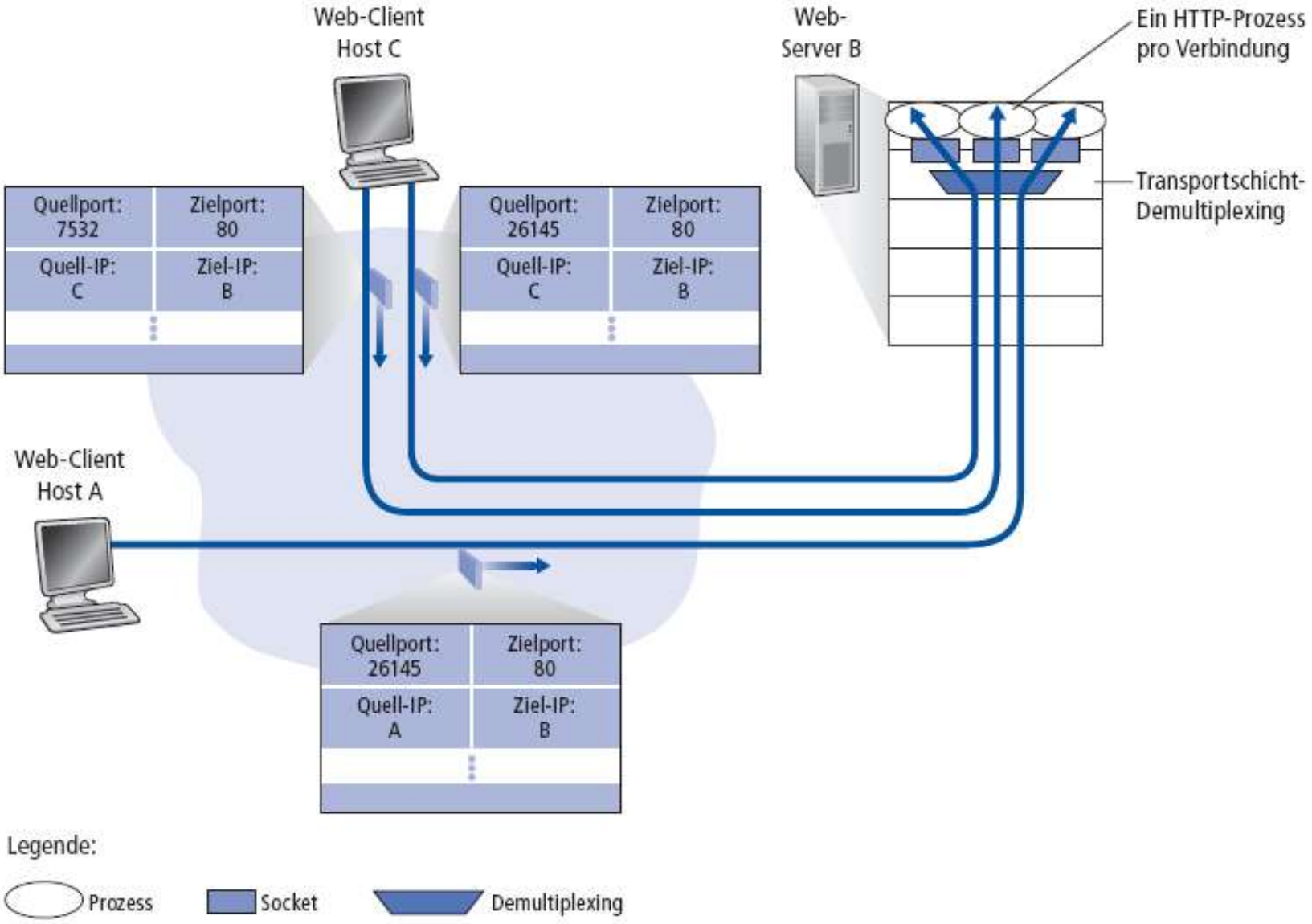
- Die Portnummer des Verbindungs-Sockets ist die gleiche wie vom Eingangs-Socket (z.B. 80 bei WWW)!
- **Problem:** Die ankommenden Pakete für die Verbindungs-Sockets haben die gleiche Zielport-nummer, müssen aber an verschiedene Verbindungs-Sockets ausgeliefert werden!

Verbindungsorientiertes Demultiplexing /1

Problem: Die Zielport-Nummern der ankommenden Pakete sind nicht ausreichend, um die Pakete dem richtigen Verbindungssocket zuzuordnen!



Verbindungsorientiertes Demultiplexing /2



Verbindungsorientiertes Demultiplexing /3

- Ein TCP-Socket wird durch ein 4-Tupel identifiziert:
  1. IP-Adresse der Quelle
  2. Quellportnummer
  3. IP-Adresse des Ziels
  4. Zielportnummer
- Empfänger (Zielhosts) nutzen alle vier dieser Werte, um den richtigen Socket zu finden
  - Zielsocket merkt sich diese Werte beim Handshake (Verbindungsaufbau)
- Deshalb kann ein Server viele Verbindungs-Sockets mit einer gleichen (lokalen) Portnummer haben
  - Sie unterscheiden sich nur durch (IP-Adresse der Quelle, Quellportnummer)

Zusammenfassung

- UDP-Protokoll
- Socket-Programmierung mit UDP / mit TCP
- Multiplexing / Demultiplexing
- Zusatzfolien: Übersicht Email-Protokolle

- Quellen:
  - Kurose / Ross Kapitel 2, Abschnitte 2.8, 2.7
  - Kurose / Ross Kapitel 3, Abschnitte 3.2 und 3.3
  - Zusatzfolien: Kurose / Ross Kapitel 2, Abschnitt 2.4

Danke.

Zusatzfolien:
Übersicht Email-Protokolle
Zurück zur Anwendungsschicht

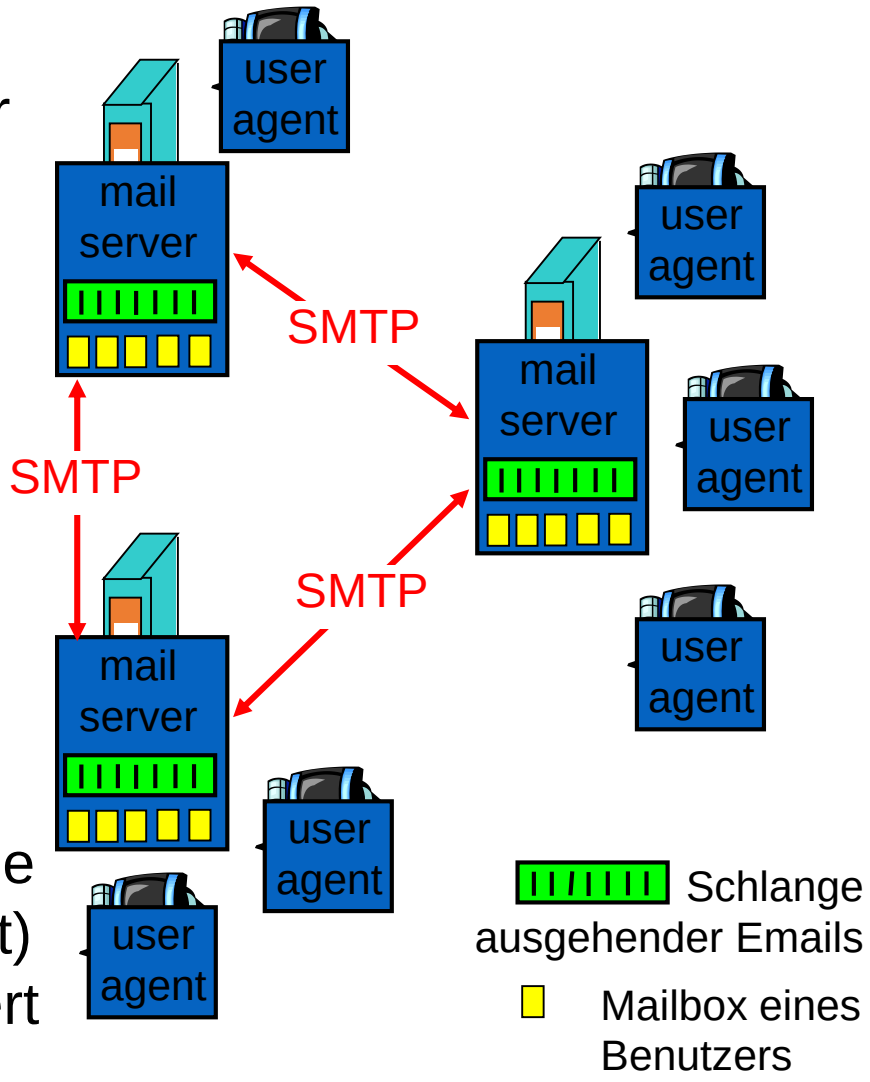
Email

Drei Hauptkompenenten

- Anwendungsprogramme für Email (**user agents**)
- **Mailserver**
- **SMTP**: simple mail transfer protocol

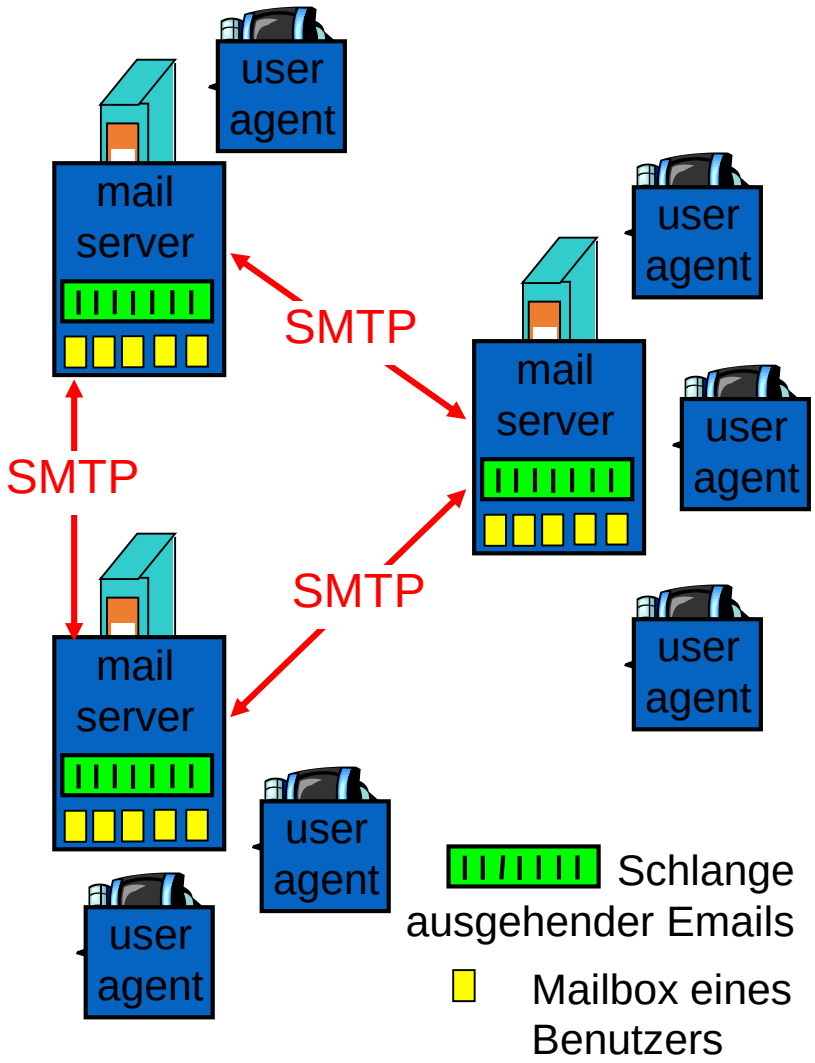
Anwendungsprogramme

- Oder “Email-Reader”
- Outlook, elm, Thunderbird, ..
- Ausgehende und ankommende Nachrichten werden (zunächst) auf dem Mailserver gespeichert



Mailserver

- Jeder Benutzer hat eine eigene **Mailbox** mit eingehenden Nachrichten
- Es gibt eine (gemeinsame) Schlange von zu sendenden Nachrichten
- Das Protokoll SMTP protocol für den Austausch zwischen den Mailservern
  - Aber nicht zwischen den Anwendungsprogrammen und Mailservern – dazu später

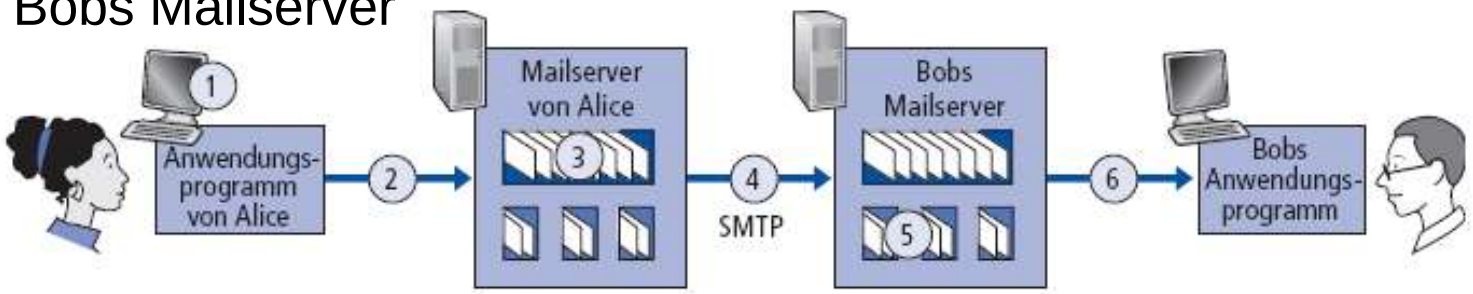


SMTP Protokoll - [RFC 2821]

- Benutzt **TCP**, Port 25, für verlässliche Zustellung der Nachrichten zwischen Mailservern
- **Direkter Transfer** (normalerweise über keine Zwischenstationen) zwischen Mailservern (identifiziert durch die Domäne nach "@")
  - Man kann Mailserver so konfigurieren, so dass es Zwischenstationen gibt
- Interaktion als "**Befehl/Antwort**" (command/ response)
  - Befehle: ASCII-Text
  - Antworten: Status-Code und ein Erklärungssatz
- Nachrichten müssen als **7-bit ASCII** codiert sein!
- Der Sender meldet sich „von selbst" bei Empfänger:
  - HTTP: Pull, SMTP: **Push**

Beispiel: Alice schickt Bob eine Nachricht

1) Alice verwendet ihr Anwendungsprogramm, um eine Nachricht an `bob@someschool.edu` zu erstellen
2) Alices Anwendung versendet die Nachricht an ihren Mail-Server
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



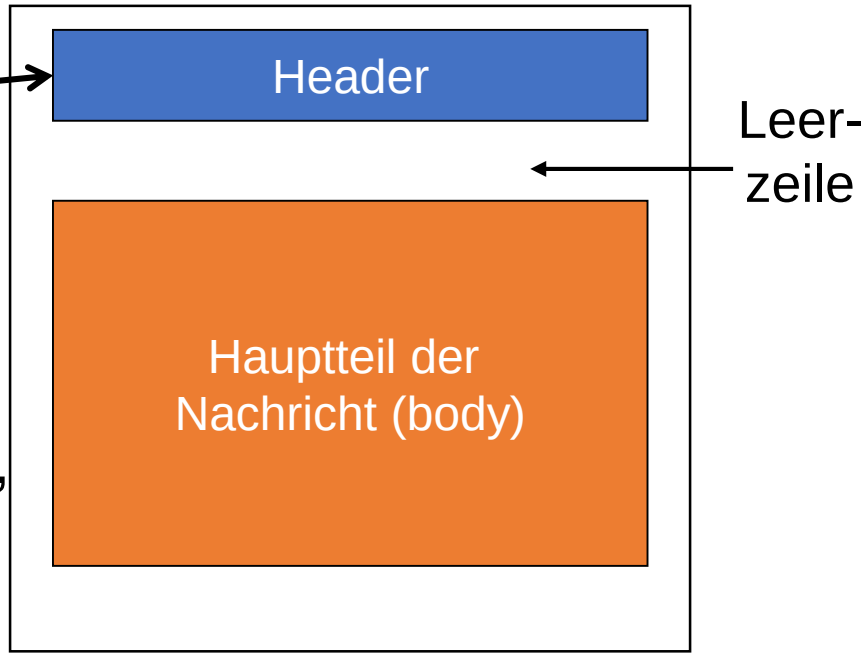
Eine SMTP-Interaction

S: Server (Empfänger)
C: Client (Sender)

```
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
```

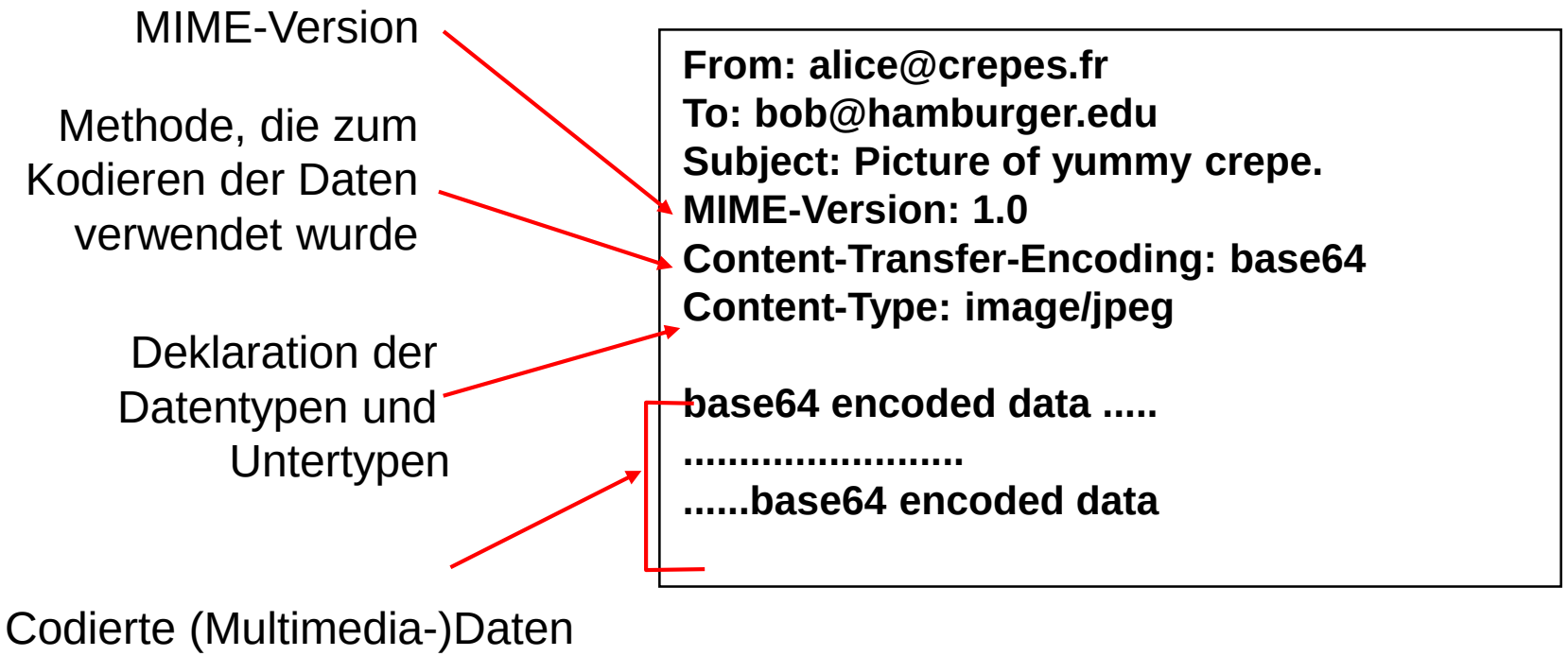
Format einer Nachricht

- RFC 822 legt fest: Innerhalb der Nachricht selbst gibt es zwei Teile:
  - **Header** (Kopf)
  - (Leerzeile zurTrennung)
  - **Hauptteil – Benutzertext**
- Für SMTP ist beides der Inhalt der Nachricht („**Data**“), sie sieht den Header nicht
- Header-Zeilen sind z.B. (keine SMTP-Befehle!)
  - To:
  - From:
  - Subject:



Multimedia-Erweiterungen

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



Datentypen in MIME

Text

Beispiele für Subtypen:

- **`plain`**
- **`html`**

Bilder

Beispiele für Subtypen:

- **`jpeg`**
- **`gif`**
- **`png`**

Audio

Beispiele für Subtypen:

- **`basic`** (8-bit mu-law encoded),
- **`32kadpcm`** (32 kbps coding)

Video

Beispiele für Subtypen:

- **`mpeg`**
- **`quicktime`**

Anwendungen

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

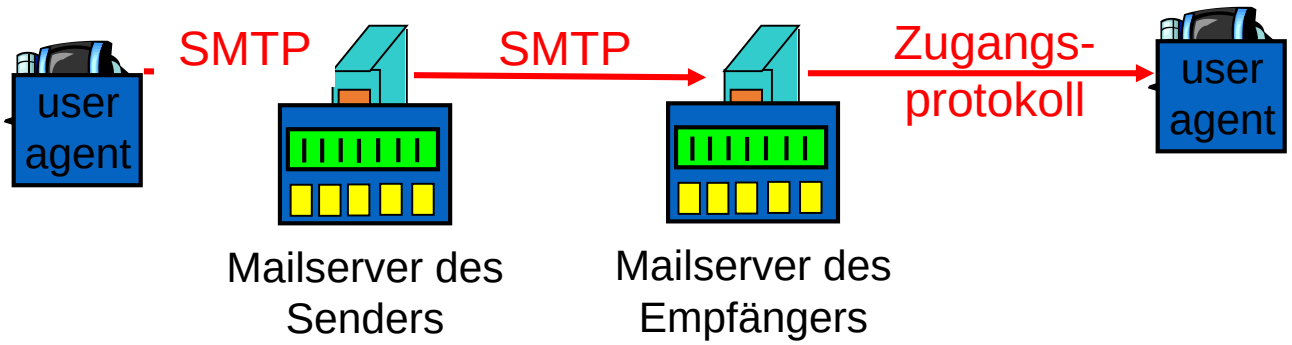
Multipart-Typ

```
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Picture of yummy crepe.
MIME-Version: 1.0
Content-Type: multipart/mixed;
boundary=StartOfNextPart

--StartOfNextPart
Dear Bob, Please find a picture of a crepe.
--StartOfNextPart
Content-Transfer-Encoding: base64
Content-Type: image/jpeg
base64 encoded data .....
.........................
......base64 encoded data
--StartOfNextPart
Do you want the recipie?
```

Mail-Zugriffsprotokolle

- SMTP: Zustellung/Speicherung auf dem Mailserver des Empfängers
- Zugriffsprotokoll: Protokolle zum Zugriff auf E-Mails
  - POP: Post Office Protocol [RFC 1939]
    - Autorisierung und Zugriff/Download
  - IMAP: Internet Mail Access Protocol [RFC 1730]
    - Größere Funktionalität (deutlich komplexer)
  - HTTP: Hotmail, Yahoo!Mail etc.



POP3- Protocol

Authorisierung

- Client-Befehle:
  - `user:` declare username
  - `pass:` password
- Server-Antworten:
  - `+OK`
  - `-ERR`

Transaktionen

Client-Befehle:

- `list:` Liste die Nachrichtennummern auf
- `retr:` hole Nachricht mit Nr. x
- `dele:` Markiere x zum Löschen
- `quit:` Verlasse und führe Änderungen aus

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

POP3 und IMAP

POP3

- "**Download-and-Delete**"-Modus, d.h., andere E-Mail- Clients haben danach keine Möglichkeit mehr, die Mails zu lesen
- Der "**Download-and-Keep**"-Modus ermöglicht den reinen Lesezugriff auf Nachrichten, d.h., verschiedene Clients haben Zugriff
- POP3 ist zwischen einzelnen Sitzungen zustandslos

IMAP

- Alle Nachrichten bleiben 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