Netzwerktechnologien
3 VO

Dr. Ivan Gojmerac
ivan.gojmerac@univie.ac.at

4. Vorlesungseinheit, 10. April 2013

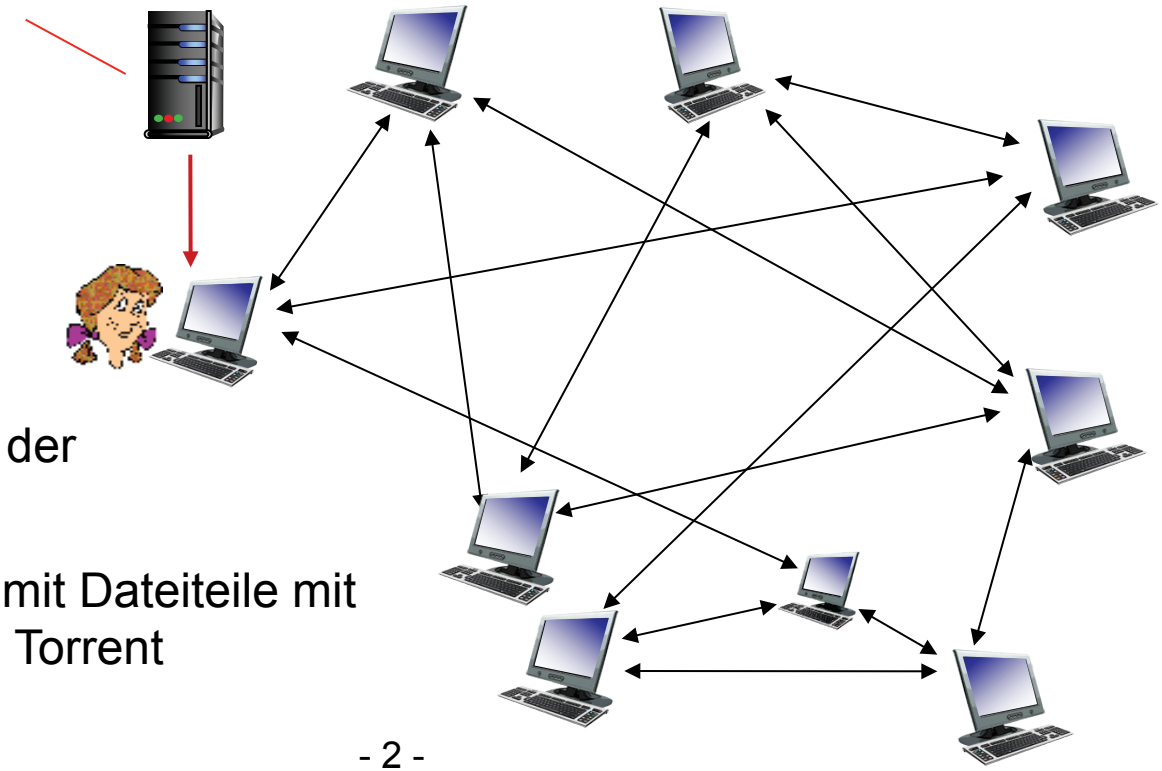
Bachelorstudium Medieninformatik
SS 2013

2.6.1 Bittorrent – Datei Distribution

- Datei wird in Teile gleicher Größe aufgeteilt
- Peers im Torrent senden/empfangen Dateiteile

Tracker: Verfolgt welche Peers am Torrent teilhaben

Torrent: Gruppe von Peers, die Teile einer Datei austauschen



Alice verbindet sich …

… bekommt die Liste der Peers vom Tracker

… und beginnt damit Dateiteile mit anderen Peers im Torrent auszutauschen.

2.6.1 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

2.6.1 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.

2.6.1 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!

2.6.1 Bittorrent – Choked-Bit

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

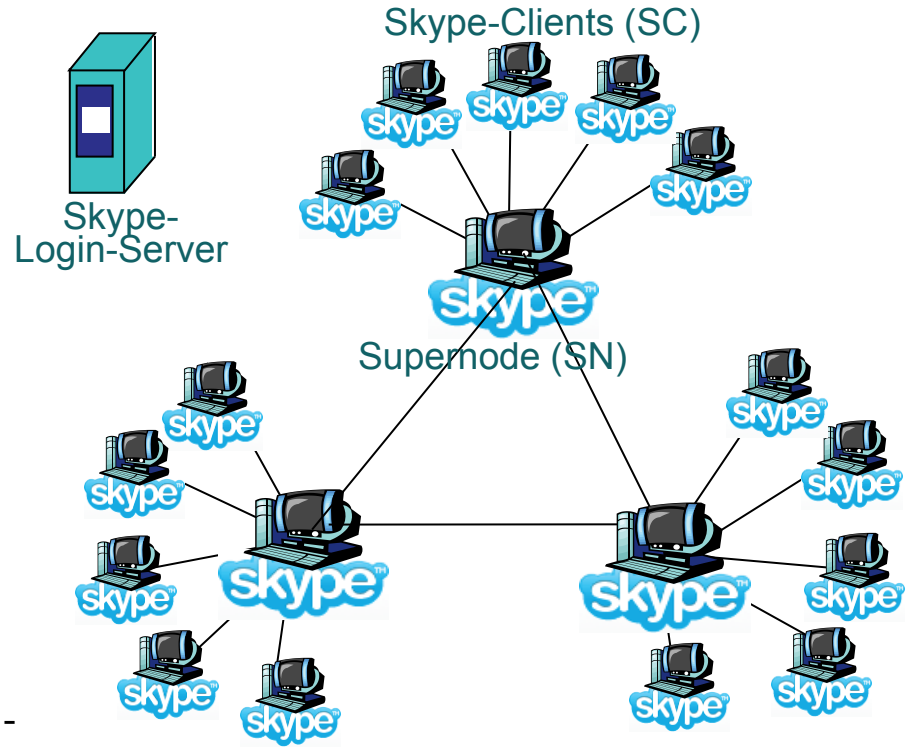
Wie wird das Choked-Bit gesetzt?

→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 lediglich 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

2.6.1 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
- Ein Index, der Skype-Benutzernamen auf aktuelle IP-Adressen und Ports abbildet ist auf die Super-Peers verteilt



2.6.2 Distributed Hash Tables (DHT)

- Eine DHT ist eine verteilte P2P Datenbank
- Besteht aus (key, value) Paaren
  - z.B. key: `[Sozialversicherungsnummer]`, value: `[Voller Name]`
  - key: `[Filmtitel]`, value: `[IP Adresse]`
- (key, value) Paare sind über Millionen von Peers verteilt
- Wenn ein Peer bei der DHT mit einem key anfragt, lokalisiert die DHT Paare mit passendem key und gibt diese an den Peer zurück
- Ein Peer kann auch selbst (key, value) Paare einfügen

2.6.2 Distributed Hash Tables (DHT)

Wie werden die (key, value) Paare auf die Peers verteilt?

Grundlegende Idee:

1. Konvertiere durch Hash jeden Key in eine Integer Zahl in einem definierten Bereich
2. Weise jedem Peer eine Integer Zahl im selben Bereich als ID zu
3. Lege jedes (key, value) Paar auf den Peer mit der **nächstgelegenen ID**

Beispiel:

Integerbereich [0, 2^n-1] bei n=4 (damit jede ID durch 4 bit dargestellt werden kann)
Peers: 1,3,4,5,8,10,12,14

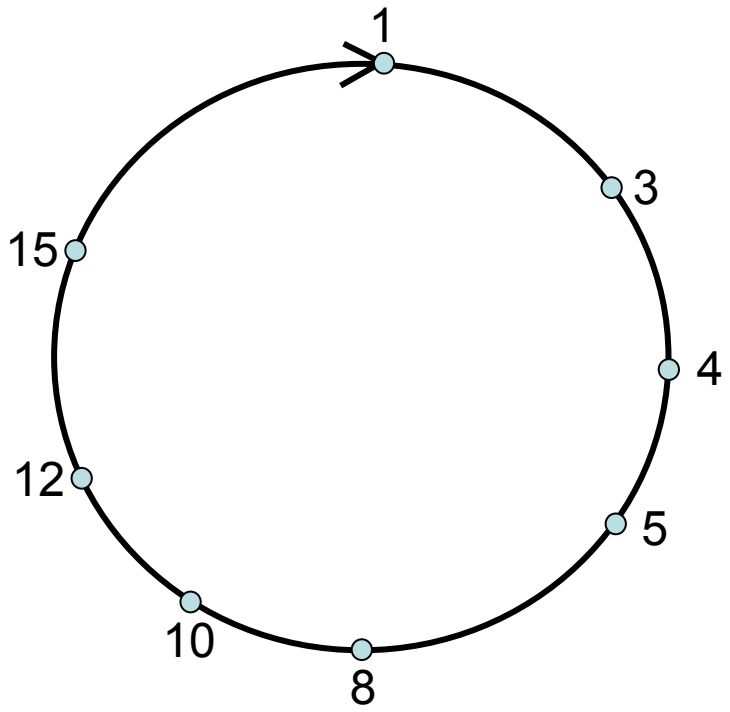
key = `13`, dann ist der nächstegelegene Peer = 14

key = `15`, dann ist der nächstegelegene Peer = 1

2.6.2 Circular DHT

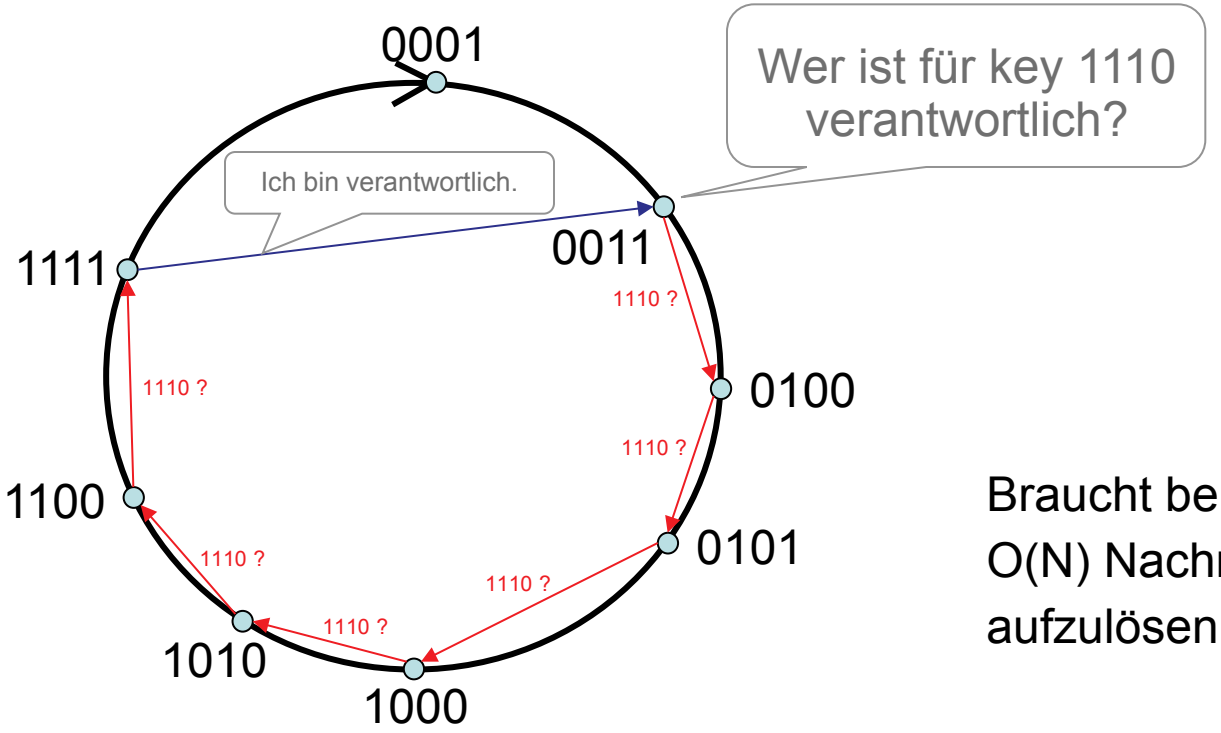
ABER: Eine Lösung bei der jeder Peer die IDs und IP Adressen aller anderen Peers lokal speichert ist für DHTs mit Millionen von Peers nicht skalierbar!

Lösung: Circular DHT



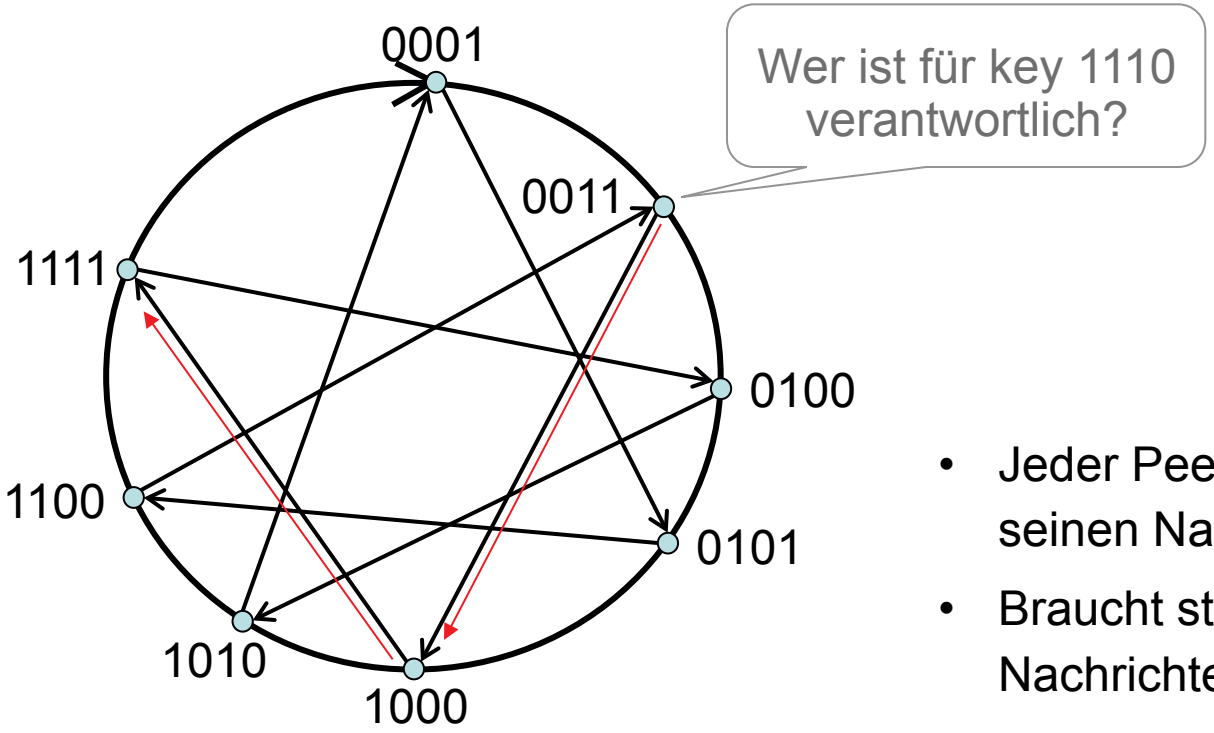
- Jeder Peer kennt *nur* seinen Vorgänger und seinen Nachfolger
- Overlay Netzwerk

2.6.2 Circular DHT



Braucht bei N Peers im Durchschnitt O(N) Nachrichten um eine Anfrage aufzulösen.

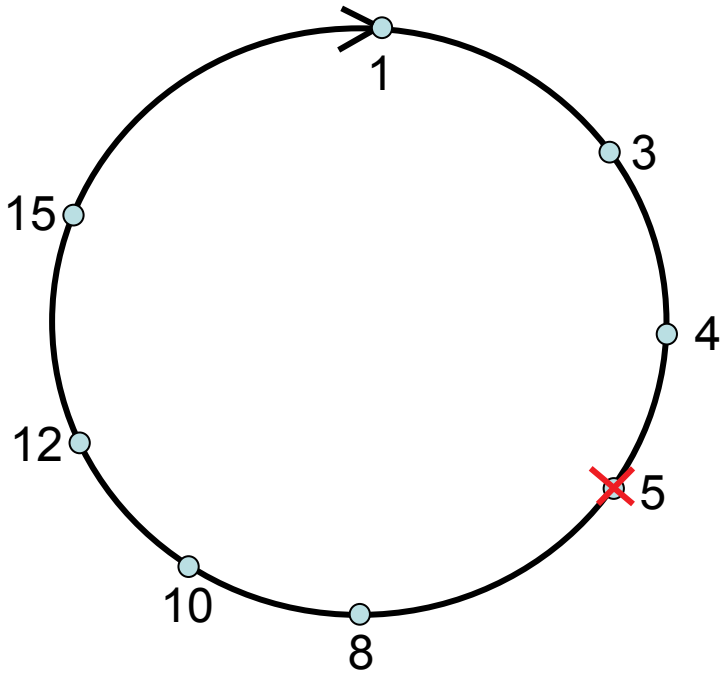
2.6.2 Circular DHT mit Abkürzungen



- Jeder Peer kennt seinen Vorgänger, seinen Nachfolger und Abkürzungen
- Braucht statt 6 nur noch 2 Nachrichten um Anfrage aufzulösen
- Abkürzungen können so angelegt werden, dass eine Anfrage im Durchschnitt nur noch O(log N) Nachrichten zur Auflösung benötigt

2.6.2 Peer Wechsel (Churn) in DHT

- Peers können ständig dazukommen oder wegfallen (=Peer Churn)
- Jeder Peer kennt die Adressen seiner **nächsten zwei Nachfolger**
- Er pingt diese beiden regelmäßig an um zu überprüfen ob sie noch da sind
- Sollte der direkte Nachfolger nicht mehr da sein wird der zweite Nachfolger zum neuen direkten Nachfolger bestimmt



Beispiel: Peer 5 fällt plötzlich weg

- Peer 4 merkt, dass Peer 5 nicht mehr da ist und macht Peer 8 zu seinem direkten Nachfolger
- Peer 4 fragt Peer 8 wer sein direkter Nachfolger ist (Antwort: Peer 10) und macht diesen zu seinem zweiten Nachfolger

2.7 Socket-Programmierung

2.7 Socket-Programmierung

<u>Definition: Socket</u>

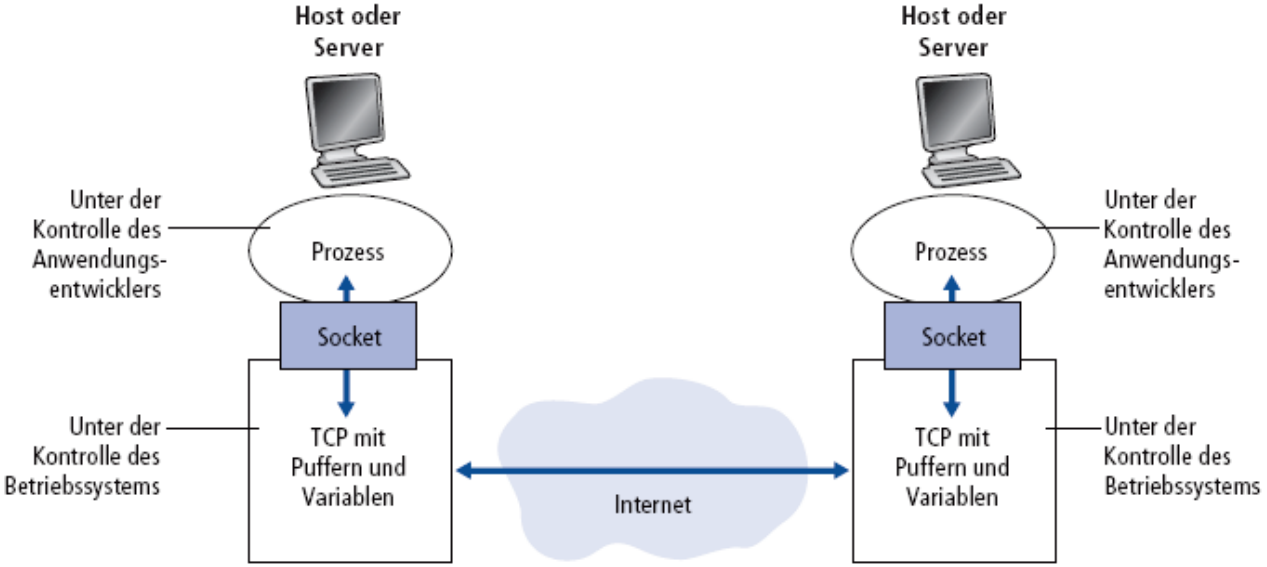
Eine Schnittstelle auf einem Host, kontrolliert durch das Betriebssystem, über das ein Anwendungsprozess sowohl Daten an einen anderen Prozess senden als auch von einem anderen Prozess empfangen kann.

→ Ein Socket ist also eine Art "Tür" zwischen Anwendungsprozess und Transportprotokoll.

<u>Socket-API</u>

- 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

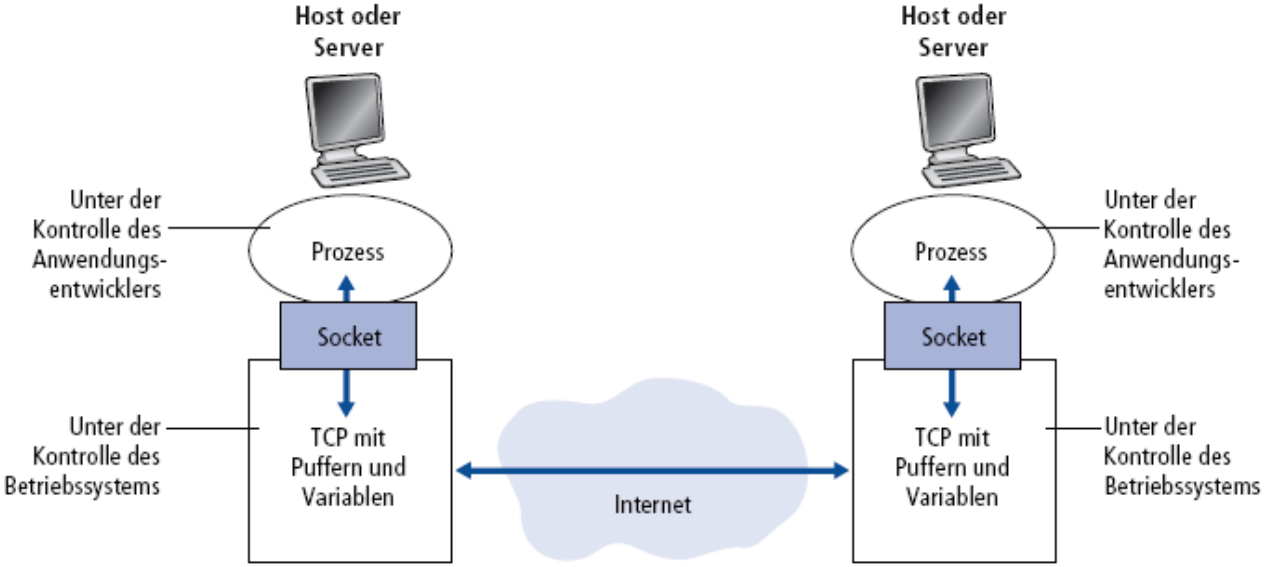
2.7.1 Socket-Programmierung mit TCP



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

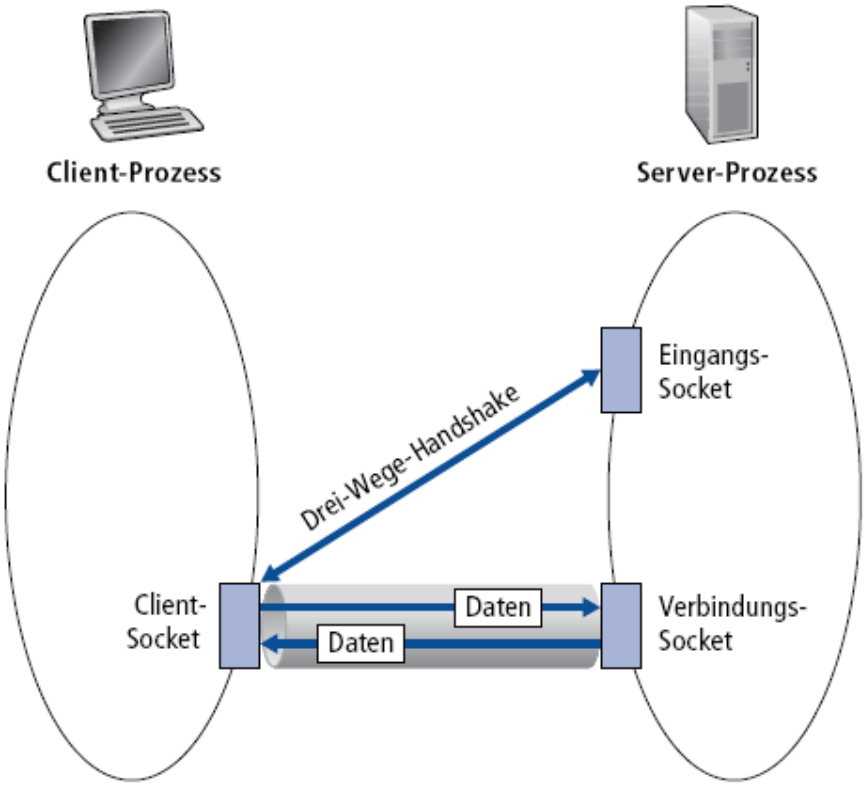
2.7.1 Socket-Programmierung mit TCP



Vorgehen im Client:

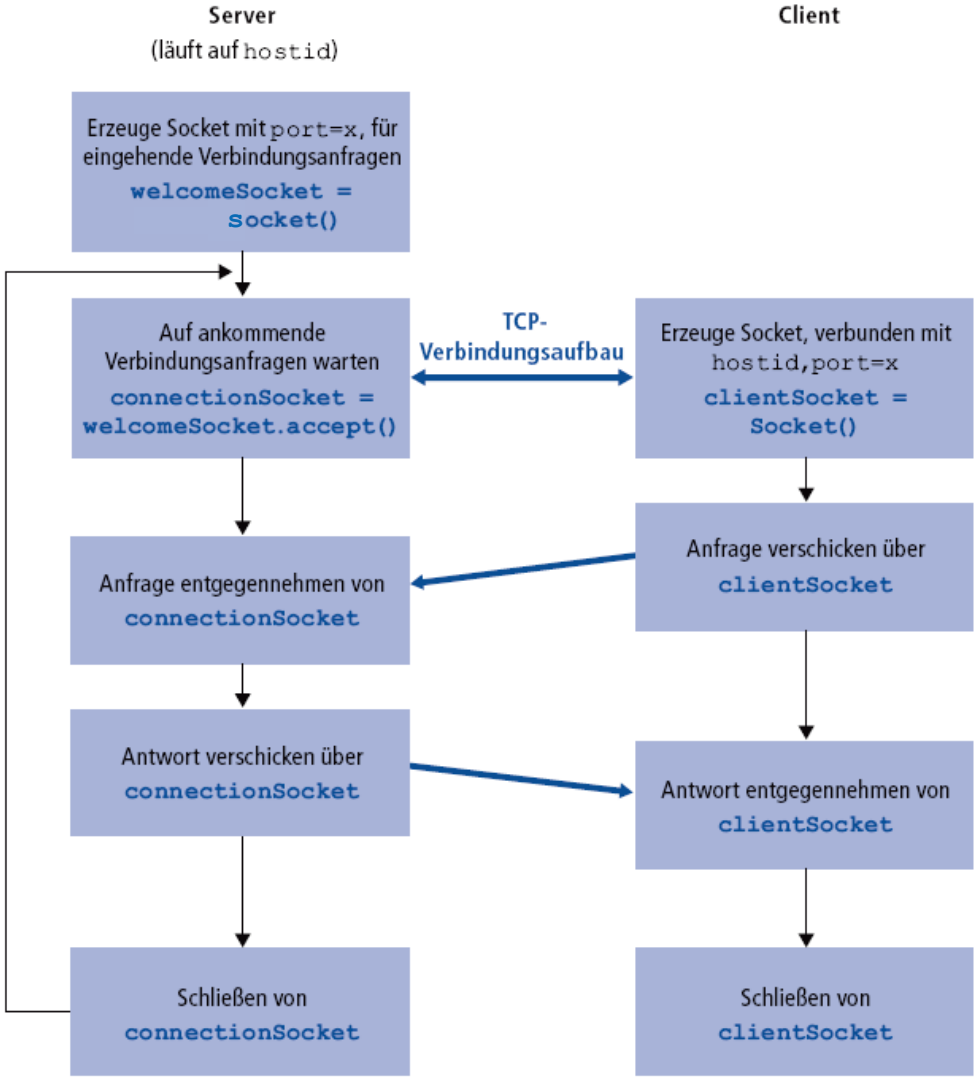
- Anlegen eines Client-TCP-Sockets
- Angeben von IP-Adresse und Portnummer des Server-Prozesses
- Durch das Anlegen eines Client-TCP-Sockets wird eine TCP-Verbindung zum Server-Prozess hergestellt

2.7.1 Socket-Programmierung mit TCP



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

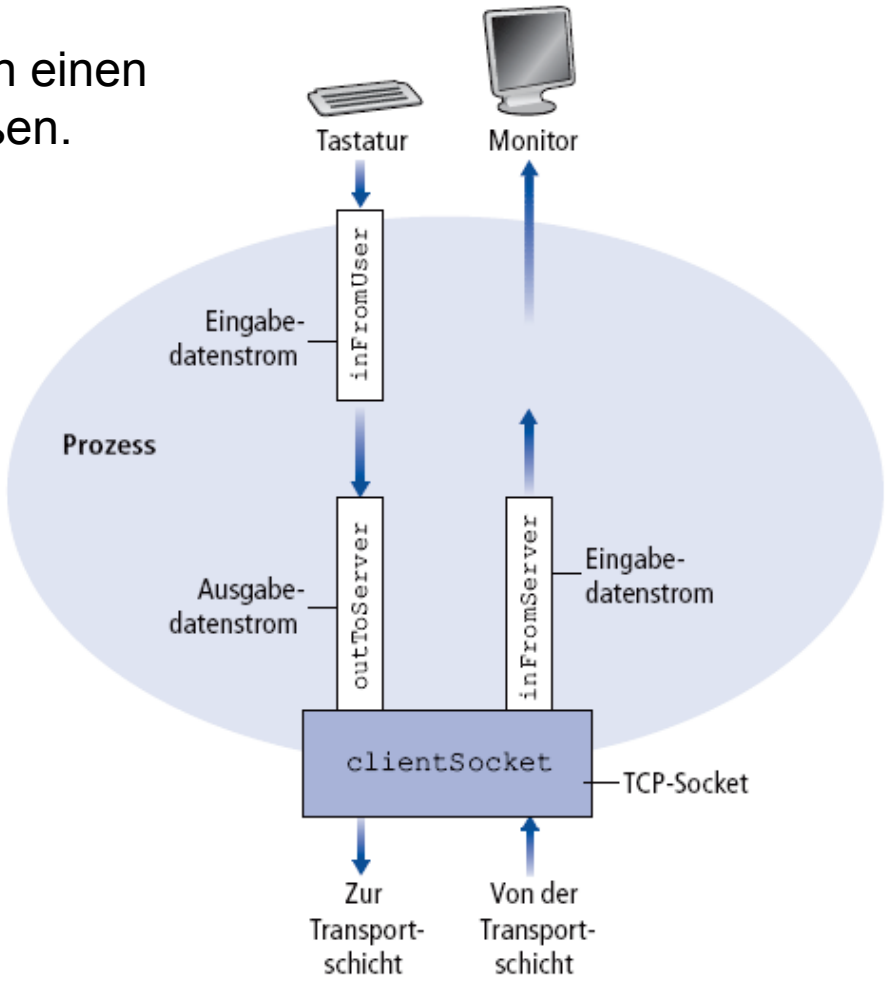
2.7.1 Socket-Programmierung mit TCP



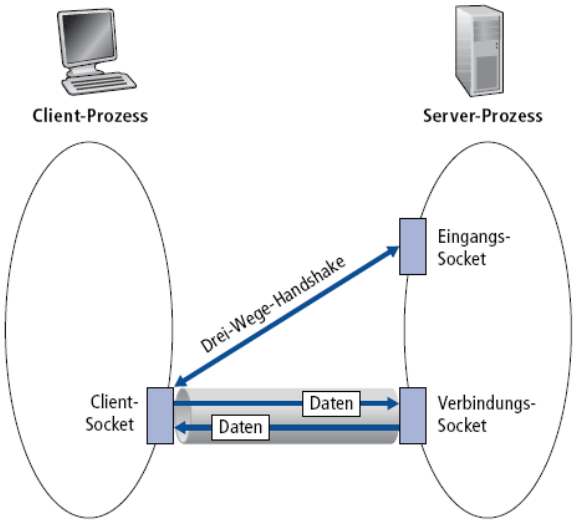
2.7.1 Socket-Programmierung mit TCP

(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.



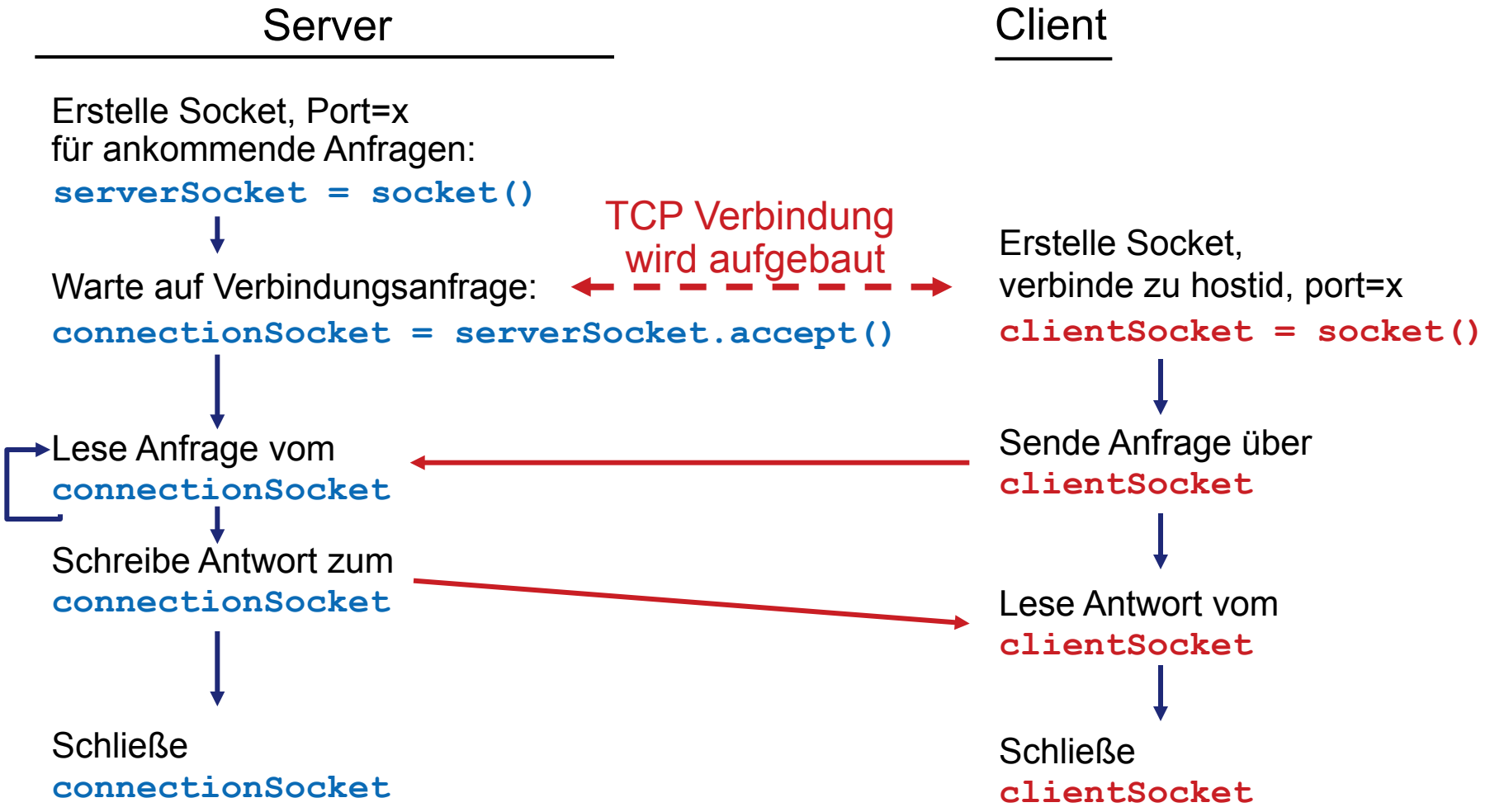
2.7.1 Socket-Programmierung mit TCP



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 Client-Server Socket Interaktion mit TCP



2.7.1 TCP Client

Python TCP Client

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = raw_input('Input lowercase sentence:')
clientSocket.send(sentence)
modifiedSentence = clientSocket.recv(2048)
print 'From Server:', modifiedSentence
clientSocket.close()
```

Erstelle TCP Socket für den Server, remote Port 12000

2.7.1 TCP Server

Python TCPServer

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

    sentence = connectionSocket.recv(2048)
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence)
    connectionSocket.close()
```

Erstelle TCP Socket

Server fängt an auf ankommende TCP Anfragen zu horchen

Endlosschleife

Server wartet im accept() auf Anfragen und erstellt einen neuen Socket im return

Lese Bytes vom Socket

Schließe die Verbindung zu diesem Client (aber schließe nicht das horchende serverSocket)

2.7.2 Socket-Programmierung mit UDP

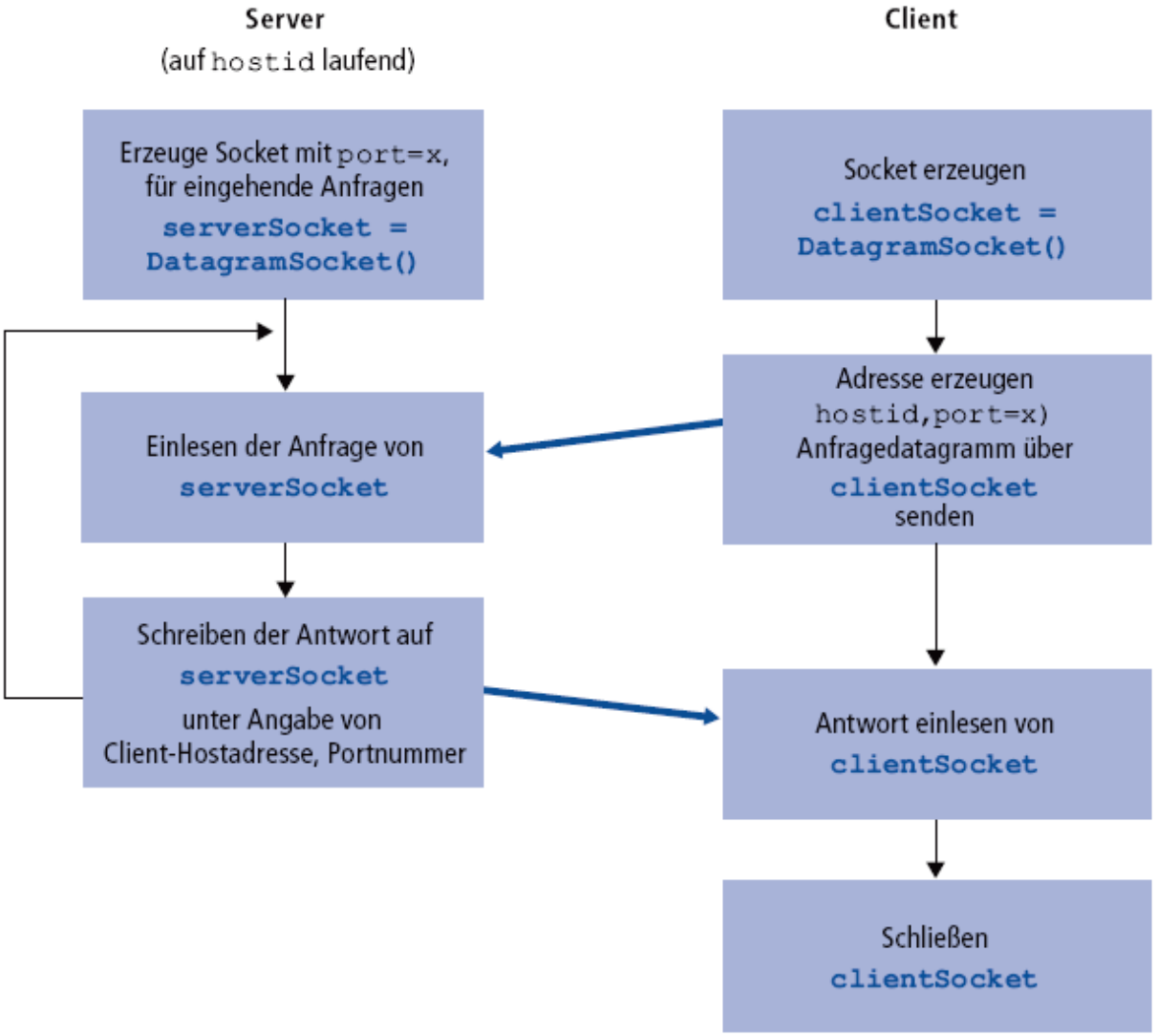
UDP: keine „Verbindung“ zwischen Client und Server

- 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

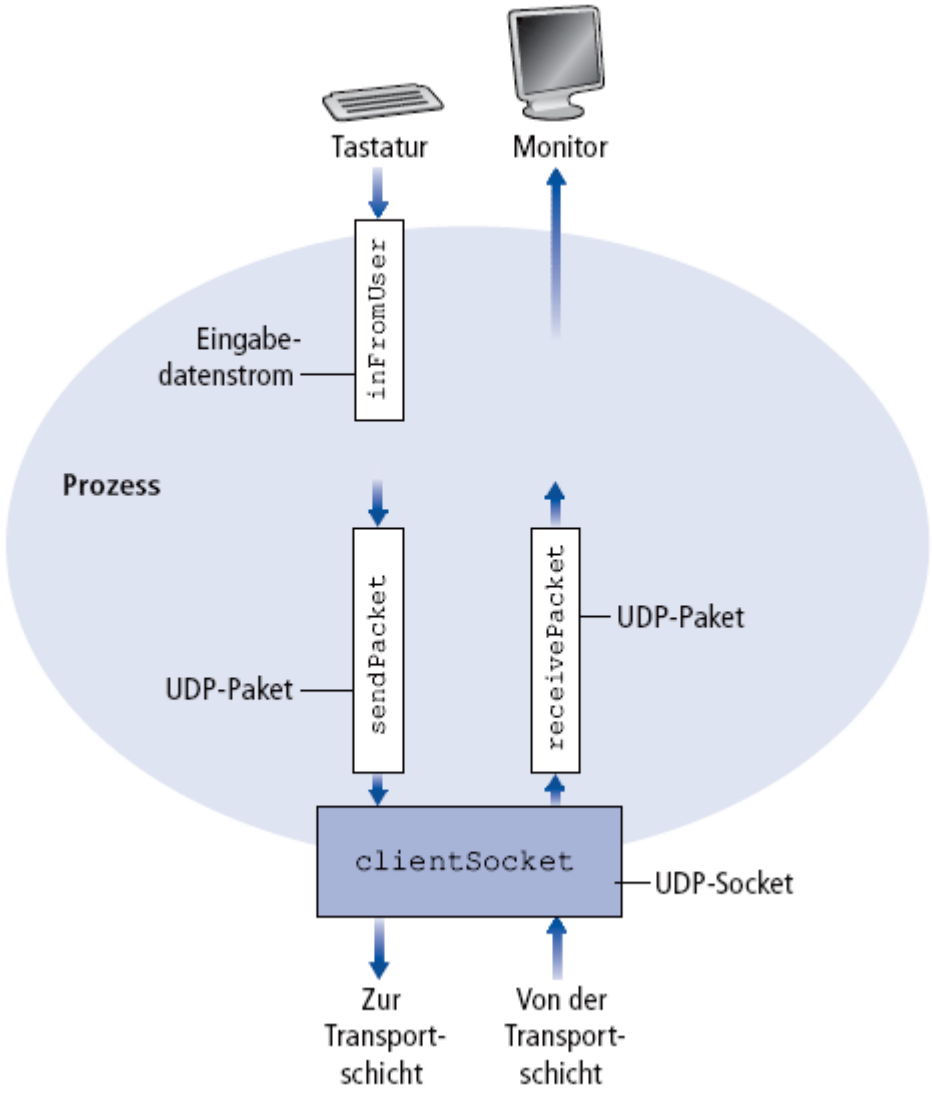
→ Mit UDP können Pakete in falscher Reihenfolge empfangen werden oder ganz verloren gehen!

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

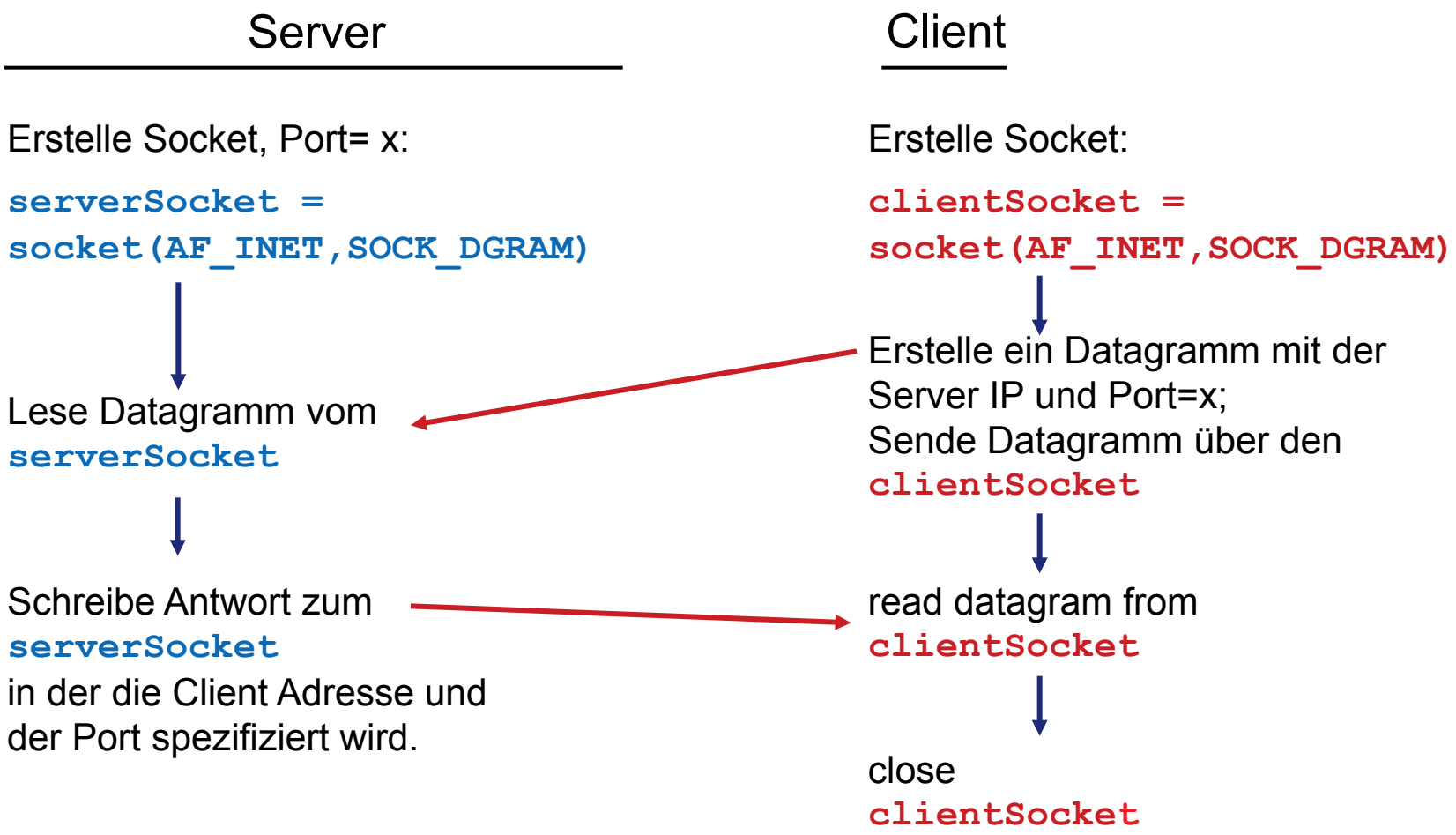
2.7.2 Socket-Programmierung mit UDP



2.7.2 Beispiel: Client mit UDP



2.7.2 Client-Server Socket Interaktion mit UDP



2.7.2 UDP Client

Python UDPClient

```
from socket import *
serverName = 'hostname'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_DGRAM)
message = raw_input('Input lowercase sentence:')
clientSocket.sendto(message,(serverName, serverPort))
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
print modifiedMessage
clientSocket.close()
```

Pythons Socket Bibliothek einbinden

Erstelle UDP Socket für den Server

Server Name und Port anfügen, dann über Socket versenden

Lese Antwort buchstabenweise aus dem Socket und schreibe sie in einen String

Zeige den erhaltenen String an und schließe das Socket

2.7.2 UDP Server

Python UDPServer

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

Erstelle UDP Socket

Binde Socket an den lokalen Port 12000

Endlosschleife

Lese vom UDP Socket in message und extrahiere die Client Adresse (IP und Port)

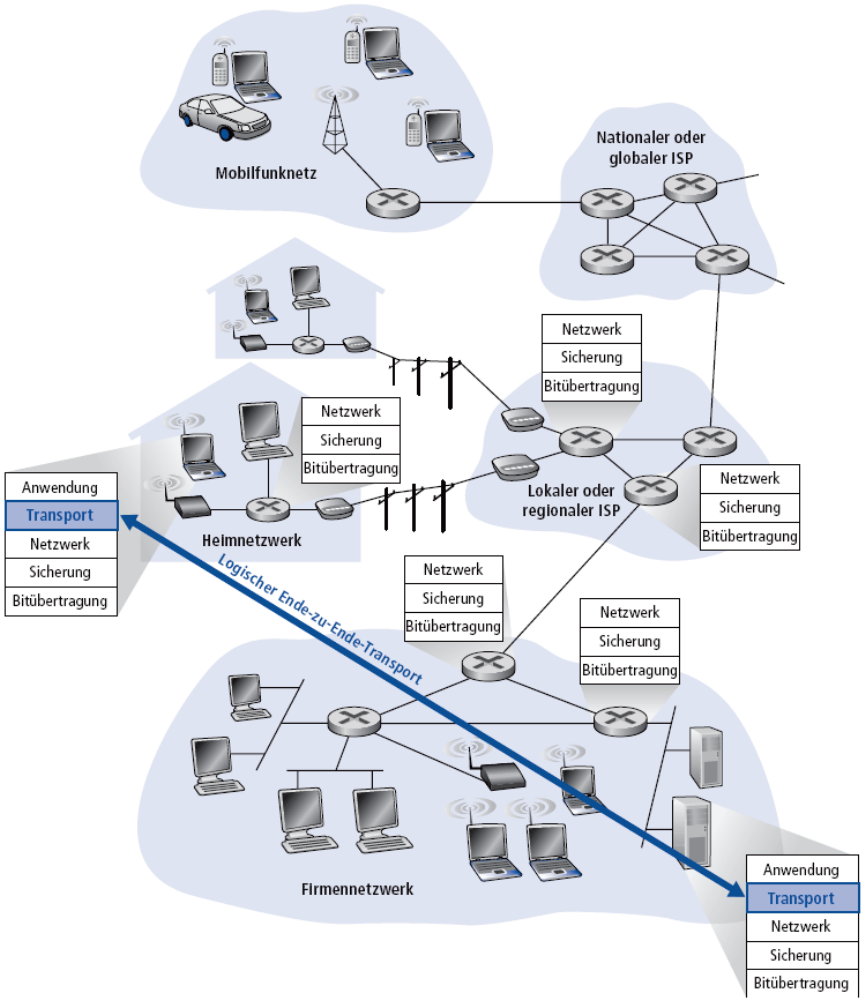
Sende einen String aus Großbuchstaben an den Client zurück

Kapitel 3 – Transportschicht

3.1 Dienste der Transportschicht

3.1 Dienste der Transportschicht

- Stellen **logische Kommunikation** zwischen Anwendungsprozessen auf verschiedenen Hosts zur Verfügung
- Transportprotokolle laufen auf Endsystemen
  - *Sender*: teilt Anwendungsnachrichten in **Segmente** auf und gibt diese an die Netzwerkschicht weiter
  - *Empfänger*: fügt Segmente wieder zu Anwendungsnachrichten zusammen, gibt diese an die Anwendungsschicht weiter
- Es existieren verschiedene Transportschichtprotokolle
  - Internet: TCP und UDP



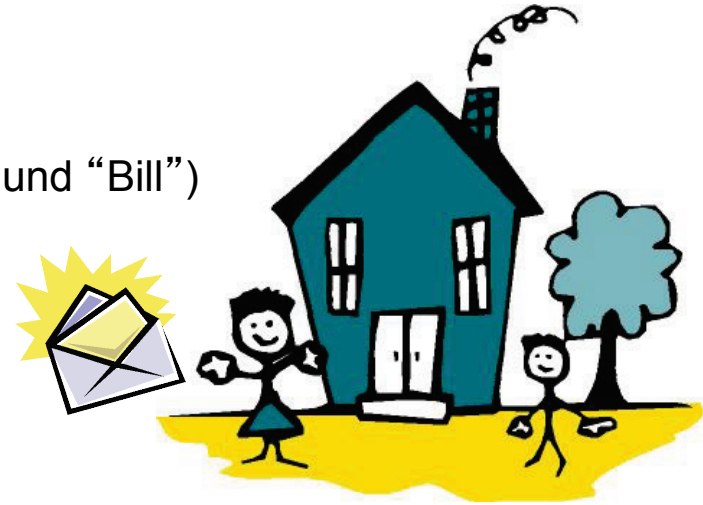
3.1 Unterschied: Transport- und Netzwerkschicht

- *Netzwerkschicht:* logische Kommunikation zwischen Hosts
- *Transportschicht:* logische Kommunikation zwischen Prozessen
  - verwendet und erweitert die Dienste der Netzwerkschicht

<u>Analogie: Briefzustellung an mehrere Bewohner des selben Hauses</u>

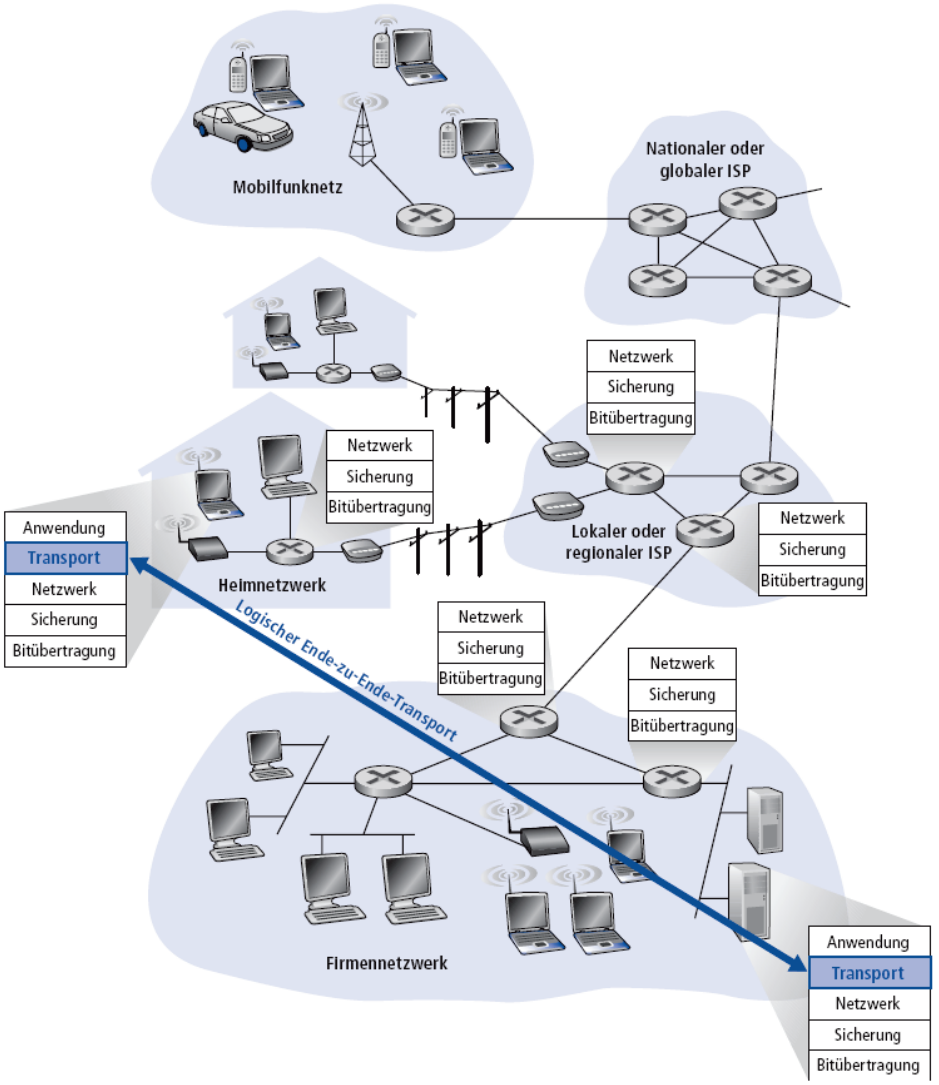
→ 12 Kinder senden Briefe an 12 andere Kinder.

- Prozess = Kind
- Anwendungsnachricht = Brief in einem Umschlag
- Host = Haus
- Transportprotokolle = Namen der Kinder (z.B. “Anne” und “Bill”)
- Netzwerkprotokoll = Postdienst



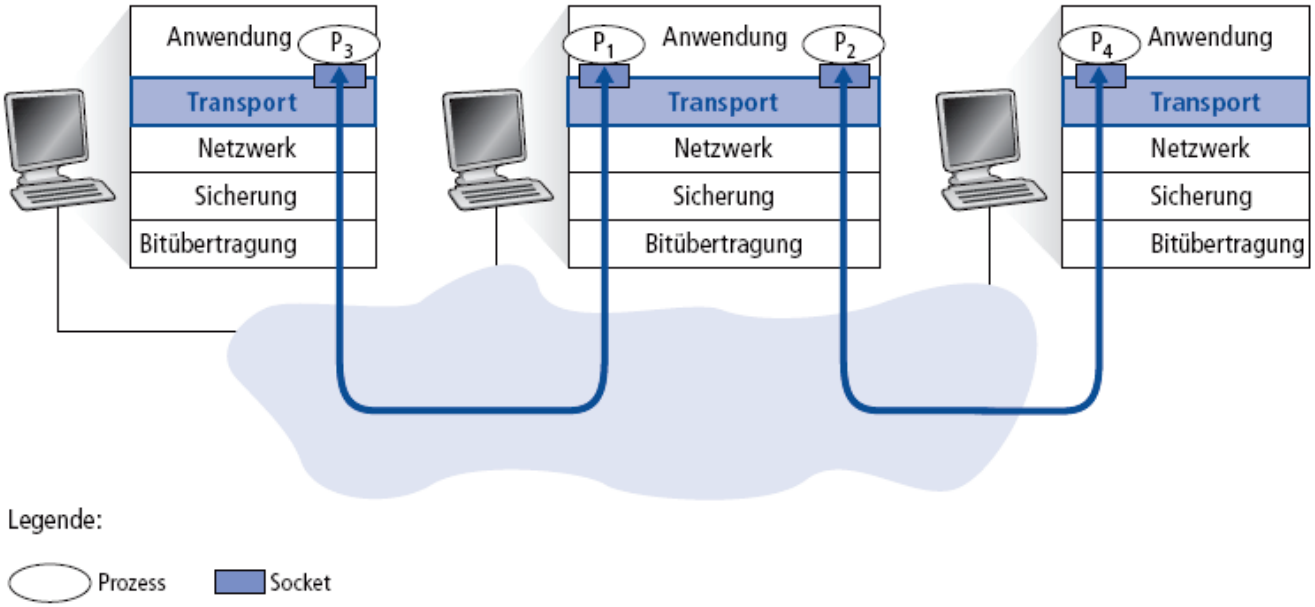
3.1 Transportprotokolle im Internet

- Zuverlässige, reihenfolgeerhaltende Auslieferung → TCP
  - Überlastkontrolle
  - Flusskontrolle
  - Verbindungsmanagement
- Unzuverlässige Datenübertragung ohne Reihenfolgeerhaltung → UDP
  - Minimale Erweiterung der "Best-Effort"-Funktionalität von IP
- Dienste, die nicht zur Verfügung stehen:
  - × Garantien bezüglich Bandbreite oder Verzögerung



3.2 Multiplexing und Demultiplexing

3.2 Multiplexing und Demultiplexing



- **Multiplexing** beim Sender:

Daten von mehreren Sockets einsammeln, Daten mit einem Header versehen (der später für das Demultiplexing verwendet wird).

- **Demultiplexing** beim Empfänger:

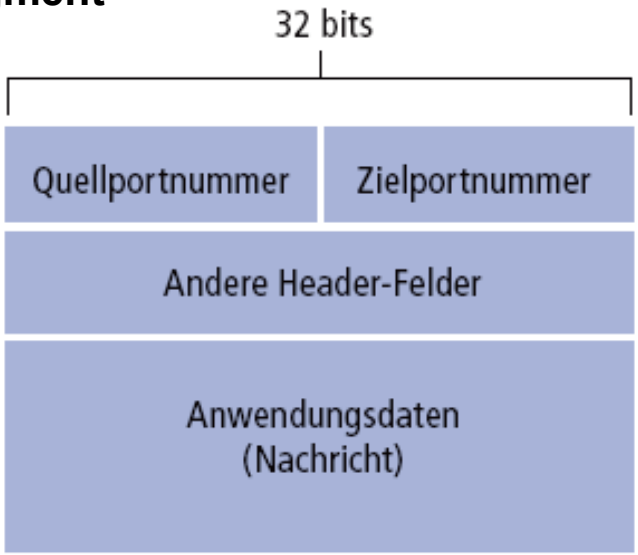
Empfangene Segmente am richtigen Socket abliefern.

3.2.1 Demultiplexing

Analogie: Briefzustellung an mehrere Bewohner des selben Hauses

Bill erhält einen Stapel Briefe vom Briefträger und gibt jeden davon dem Kind, dessen Name auf dem Brief steht.

- Host empfängt IP-Datagramme
  - Jedes Datagramm hat eine Absender-IP-Adresse und eine Empfänger-IP-Adresse
  - Jedes **Datagramm** beinhaltet ein **Transportschichtsegment**
  - Jedes **Segment** hat eine Absender- und eine Empfänger-Portnummer
- Hosts nutzen **IP-Adressen** und **Portnummern**, um Segmente an den richtigen Socket weiterzuleiten



TCP/UDP Segmentformat

3.2.1 Verbindungsloses Demultiplexing (UDP)

- Sockets mit Portnummer anlegen:

```
DatagramSocket mySocket1 = new DatagramSocket(12534);
DatagramSocket mySocket2 = new DatagramSocket(12535);
```

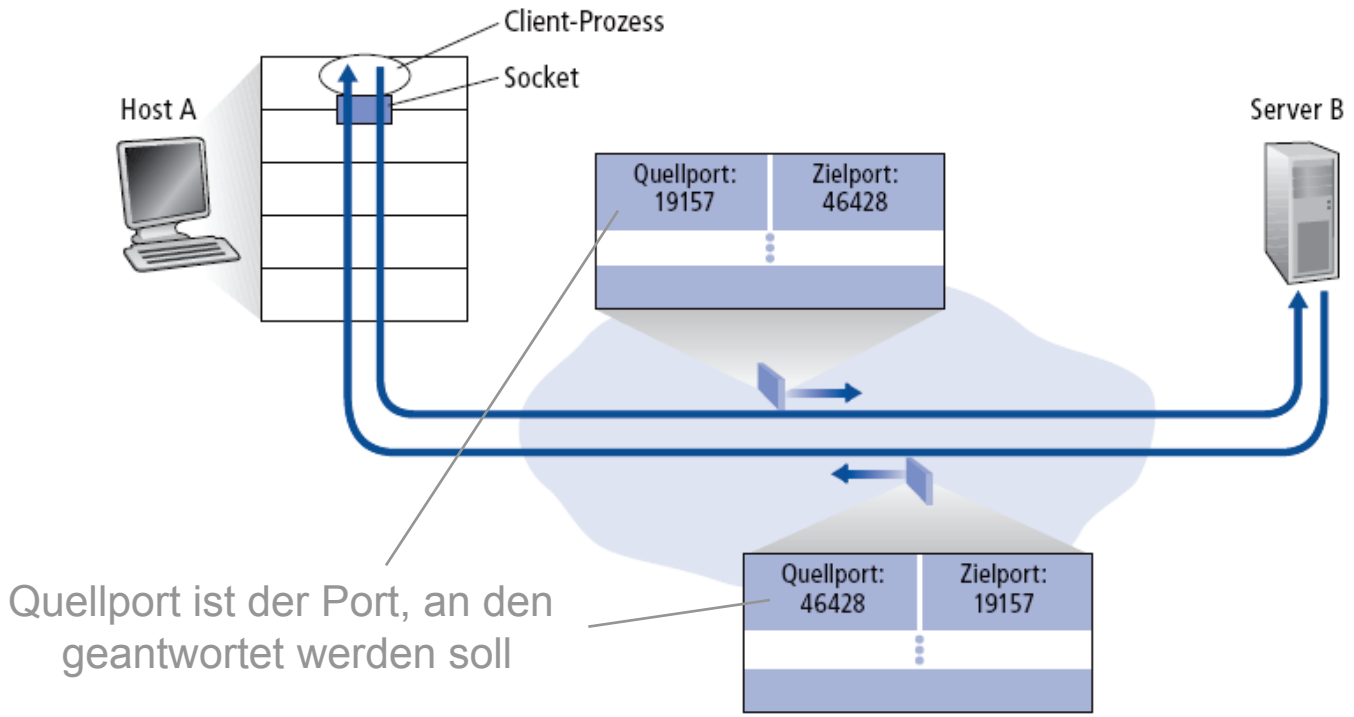
- **UDP**-Socket wird durch ein 2-Tupel identifiziert:

(Empfänger-IP-Adresse, Empfänger-Portnummer)

- Wenn ein Host ein UDP-Segment empfängt:
  1. Lese Empfänger-**Portnummer**
  2. Das UDP-Segment wird an den UDP-Socket mit dieser Portnummer weitergeleitet
- IP-Datagramme mit anderer Absender-IP-Adresse oder anderer Absender-Portnummer werden an **denselben Socket** ausgeliefert

3.2.1 Verbindungsloses Demultiplexing

```
DatagramSocket serverSocket = new DatagramSocket(46428);
```



3.2.2 Verbindungsorientiertes Demultiplexing (TCP)

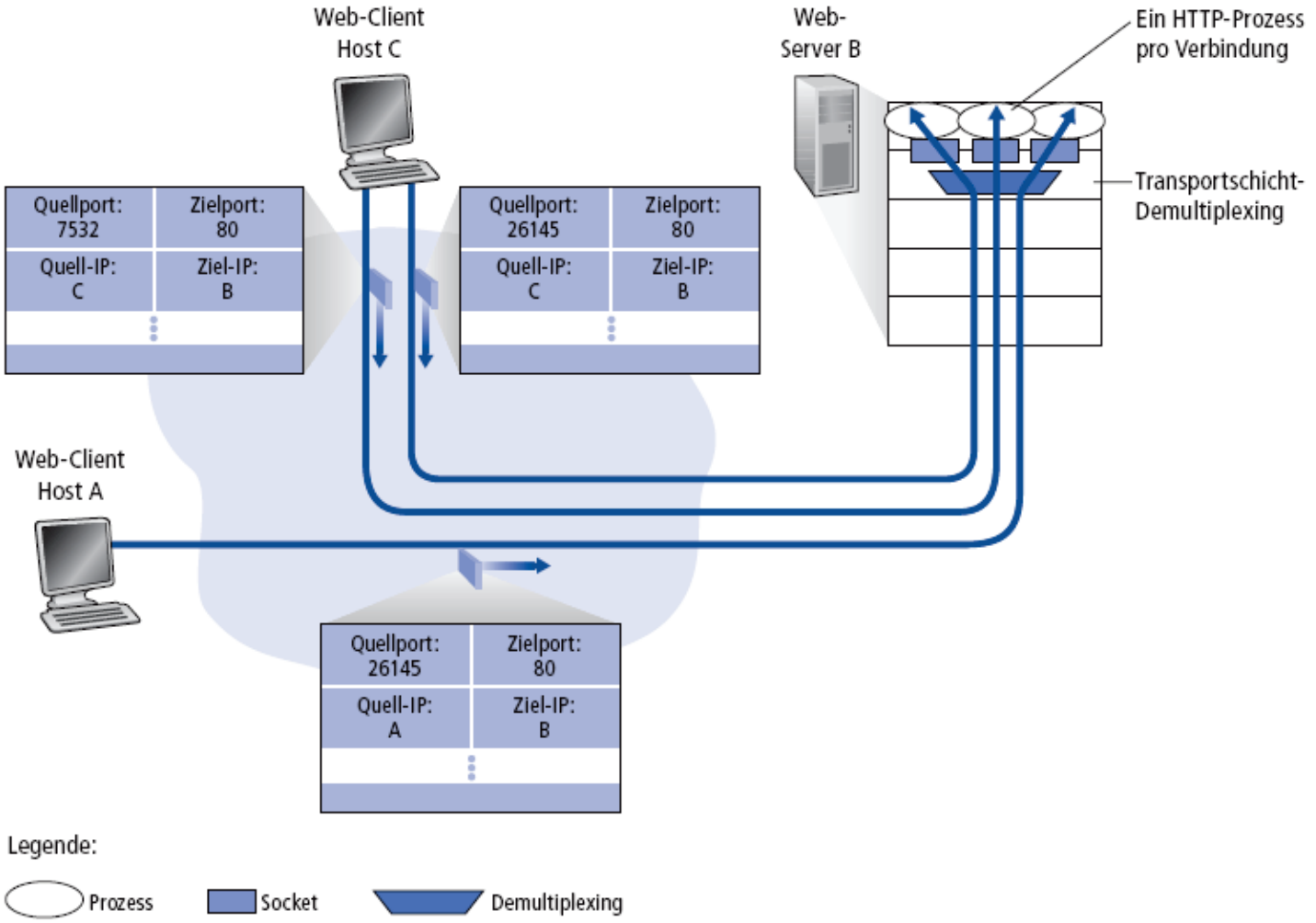
- **TCP**-Socket wird durch ein 4-Tupel identifiziert:
  - Absender-IP-Adresse
  - Absender-Portnummer
  - Empfänger-IP-Adresse
  - Empfänger-Portnummer

→ *Empfänger nutzt alle vier Werte, um den richtigen TCP-Socket zu identifizieren*

- Server kann viele TCP-Sockets gleichzeitig offen haben:
  - Jeder Socket wird durch sein eigenes 4-Tupel identifiziert

- Webserver haben verschiedene Sockets für jeden einzelnen Client
  - Bei nichtpersistentem HTTP wird jede Anfrage über einen eigenen Socket beantwortet (dieser wird nach jeder Anfrage wieder geschlossen)

3.2.2 Verbindungsorientiertes Demultiplexing (TCP)



3.3 Verbindungsloser Transport: UDP

3.3 Verbindungsloser Transport: UDP

- Minimales Internet-Transportprotokoll
- “Best-Effort”-Dienst, UDP-Segmente können:
  - verloren gehen
  - in der falschen Reihenfolge an die Anwendung ausgeliefert werden
- *Verbindungslos:*
  - kein Handshake zum Verbindungsaufbau
  - jedes UDP-Segment wird unabhängig von allen anderen behandelt

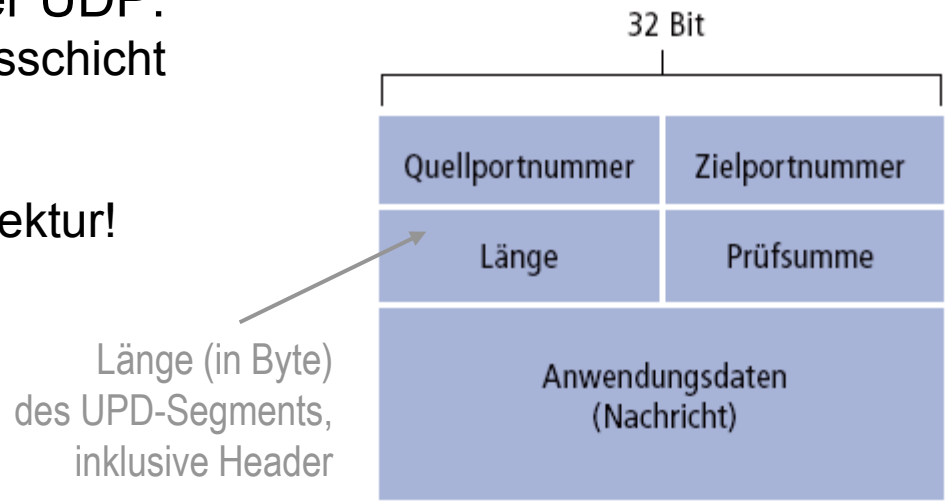
Warum gibt es UDP?

- Kein Verbindungsaufbau (der zu Verzögerungen führen kann)
- Einfach: kein Verbindungszustand im Sender oder Empfänger
- Kleiner Header
- Keine Überlastkontrolle: UDP kann so schnell wie von der Anwendung gewünscht senden

3.3 Verbindungsloser Transport: UDP

- Häufig für Anwendungen im Bereich **Multimedia-Streaming** eingesetzt
  - Verlusttolerant
  - Mindestrate
- Andere Einsatzgebiete
  - DNS
  - SNMP
- Zuverlässiger Datentransfer über UDP:
  Zuverlässigkeit auf der Anwendungsschicht implementieren

→ Anwendungsspezifische Fehlerkorrektur!



UDP Segmentformat

3.3.1 Fehlerkorrekturmechanismus: UDP Prüfsumme

Ziel: Fehler im übertragenen Segment erkennen (z.B. verfälschte Bits)

Sender:

- Betrachte Segment als Folge von 16-Bit-Integer-Werten
- Prüfsumme: Addition (1er-Komplement-Summe) dieser Werte
- Sender legt das *invertierte Resultat* im UDP-Prüfsummenfeld ab

Empfänger:

- Berechne die Prüfsumme des empfangenen Segments **inkl. des Prüfsummenfeldes**
- Sind im Resultat alle Bits 1?
  → NEIN – Fehler erkannt
  → JA – Kein Fehler erkannt

3.3.1 Fehlerkorrekturmechanismus: UDP Prüfsumme

Zahlen werden addiert und ein Übertrag aus der höchsten Stelle wird zum Resultat an der niedrigsten Stelle addiert.

Beispiel:
Addiere zwei 16-Bit-Integer-Werte.

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
+	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Übertrag	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
Summe (mit Übertrag)	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
Prüfsumme	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1