

2.1 Grundlagen

2.2 Protokolle und Dienste

2.3 Sockets

2.3.1 Übersicht

2.3.2 Adressierung über Ports

2.3.3 Socket-Programmierung

2.3.4 Live-Coding und Programmierung in Python (Übung)

2.4 Grundlagen der Datenübertragung

2.5 Aufbau des Internets

2.6 Zusammenfassung

- 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

Client-Prozess: Prozess, der die Kommunikation beginnt

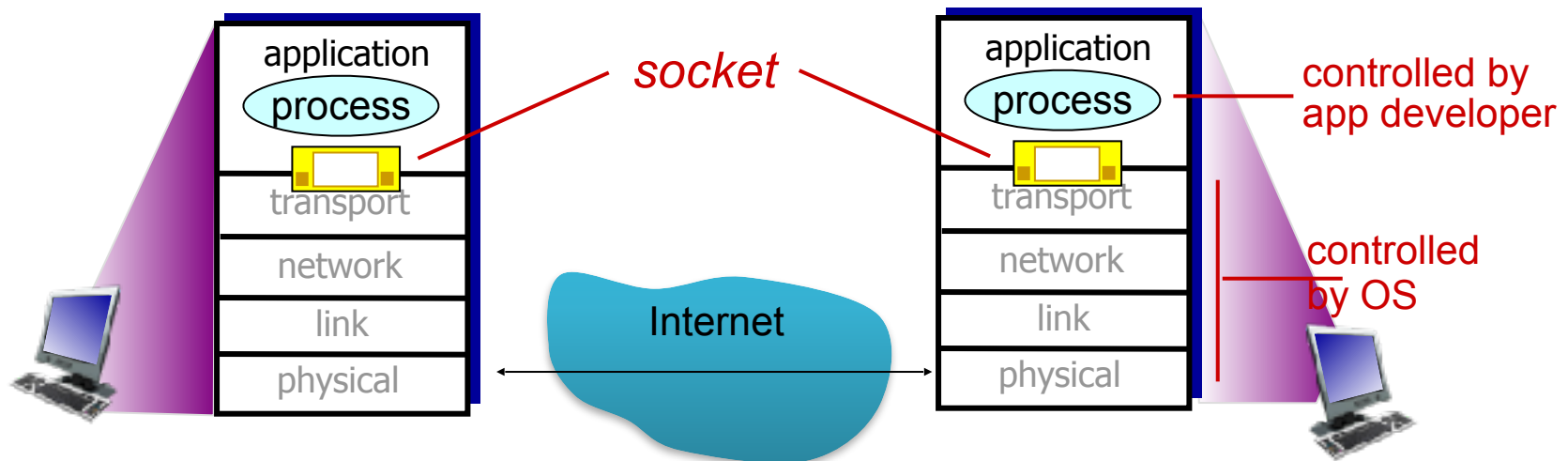
Server-Prozess: Prozess, der darauf wartet, kontaktiert zu werden

Anmerkung:

Anwendungen mit einer P2P-Architektur haben Client- und Server-Prozesse

Sockets

- Prozesse kommunizieren über **Sockets** miteinander
- Transportprotokolle transportieren die Daten zwischen dem Sender- und Empfänger-Socket
 - die Adressierung von Sockets auf einem Rechner erfolgt über den Port
 - die Anwendung "HTTP-Server" hat beispielsweise den Port 80
- Anwendung und Transportprotokoll arbeiten **asynchron**:
 - Die Anwendung kann jederzeit Daten in den Sende-Socket schreiben und das Transportprotokoll überträgt sie so schnell wie möglich
 - Das Transport-Protokoll schreibt empfangene Daten in den Empfangs-Socket und die Anwendung liest die Daten bei Bedarf
- Sockets sind Schnittstellen des Betriebssystems für den Zugriff auf Transportprotokolle



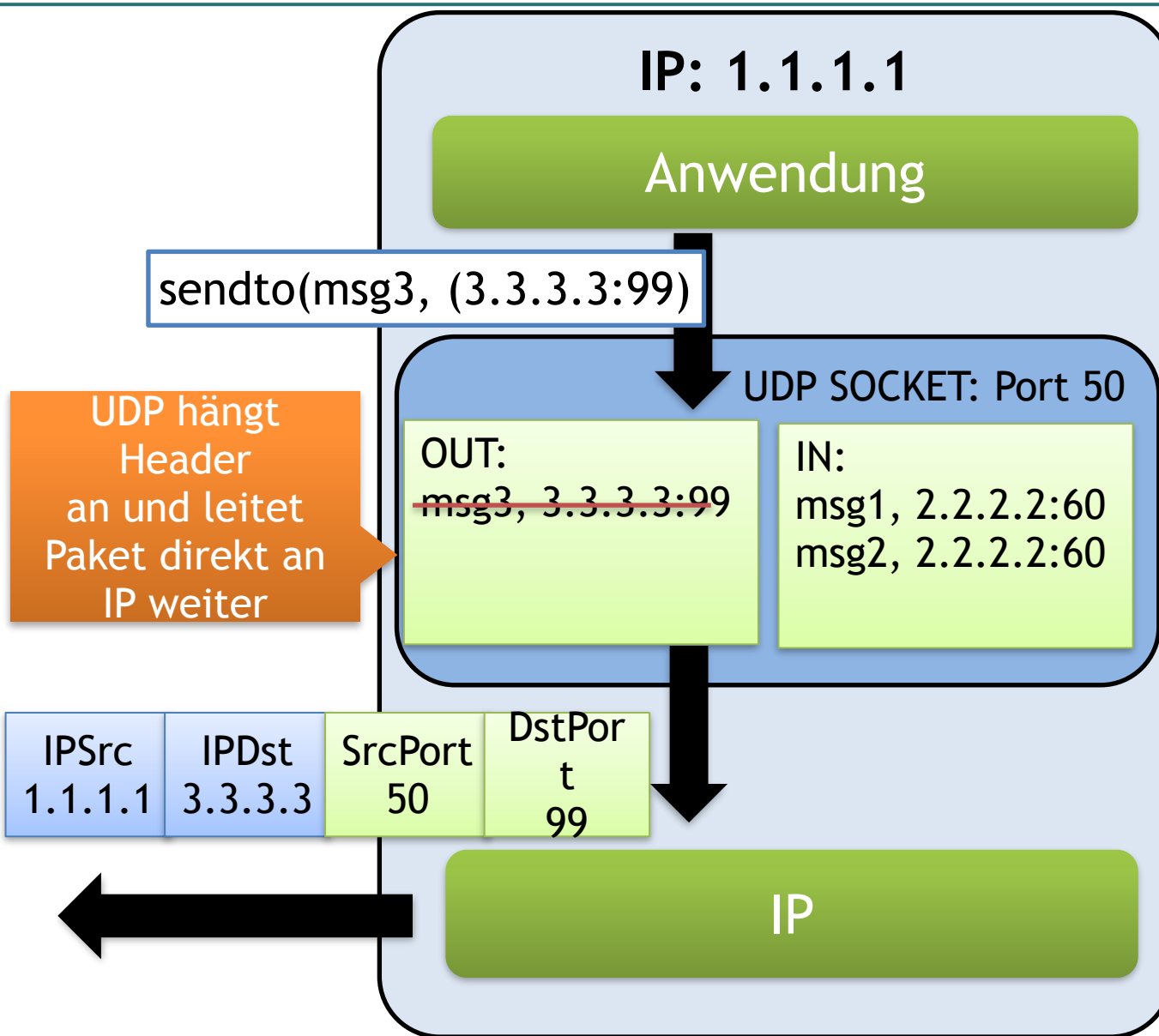
Adressierung von Prozessen

- Hosts besitzen IP-Adressen (32 Bit für IPv4, 128 Bit für IPv6)
- Beispiele:
 - 141.37.121.85
 - 2001:7c0:5f0:f128:5185:9501:4395:183b
- mehrere Prozesse pro Host
- Portnummern identifizieren Prozesse
- Beispiele:
 - HTTP: Port 80
 - IMAP: Port 143
 - SMTP: Port 25

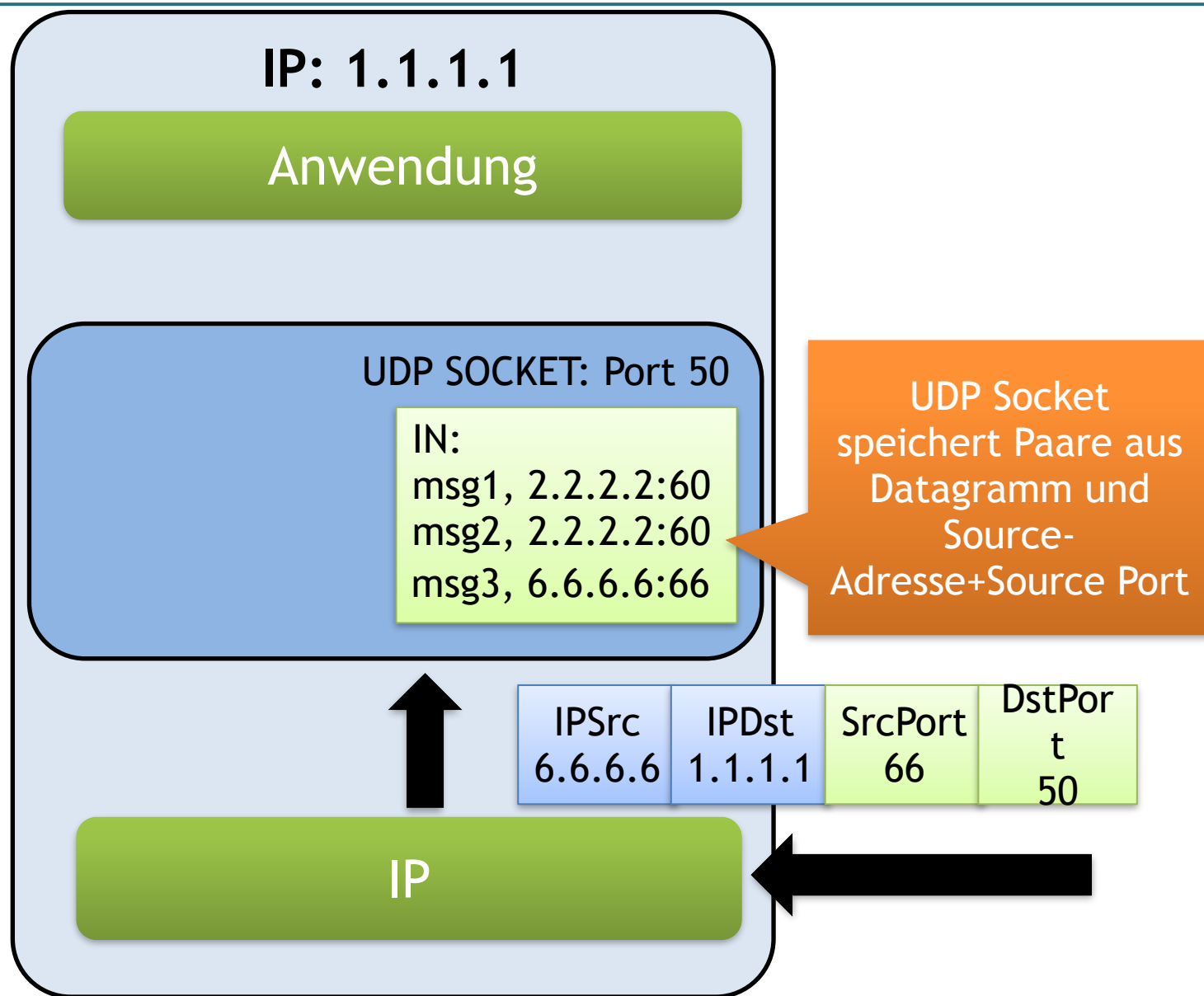
Sockets - UDP und TCP

- Die wichtigsten vom Betriebssystem bereitgestellten Sockets sind **UDP** (User Datagram Protocol) Sockets und **TCP** (Transmission Control Protocol) Sockets
- UDP Sockets sind **verbindungslos**
 - über UDP Sockets können direkt Nachrichten versendet werden, ohne eine Verbindung aufzubauen
 - eine Anwendung kann über einen UDP Socket **mit verschiedenen UDP Sockets** kommunizieren
 - über UDP Sockets werden **Datagramme** mit maximal 65536 Bytes versendet, UDP segmentiert die Datagramme nicht
 - wird eine Datagrammgröße von IP nicht unterstützt, wird das Datagramm verworfen
 - über UDP versendete Datagramme können **verloren** gehen und in einer **veränderten Reihenfolge** am Ziel-Socket ankommen
 - **ungesicherte** Übertragung

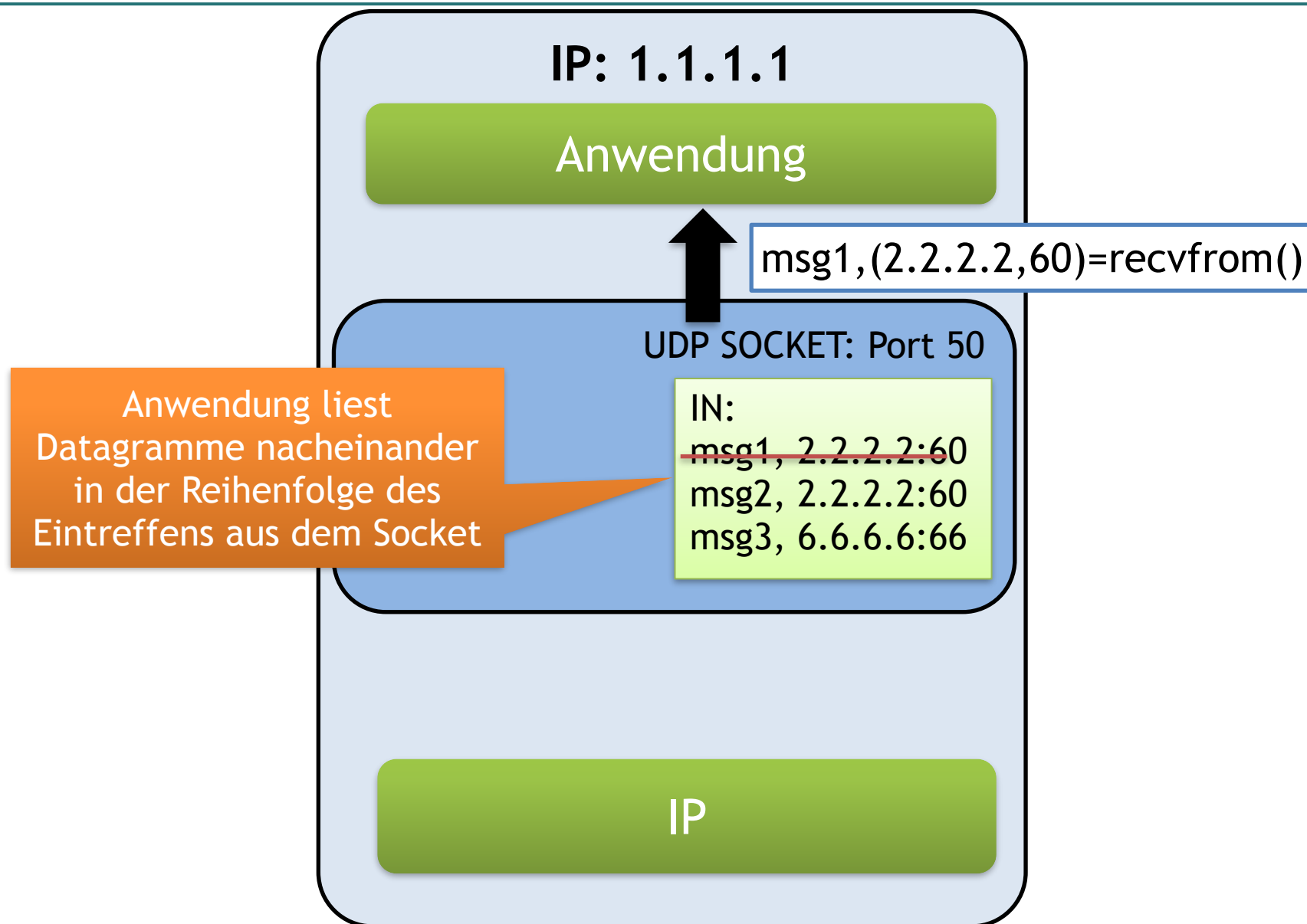
UDP - Socket



UDP - Socket

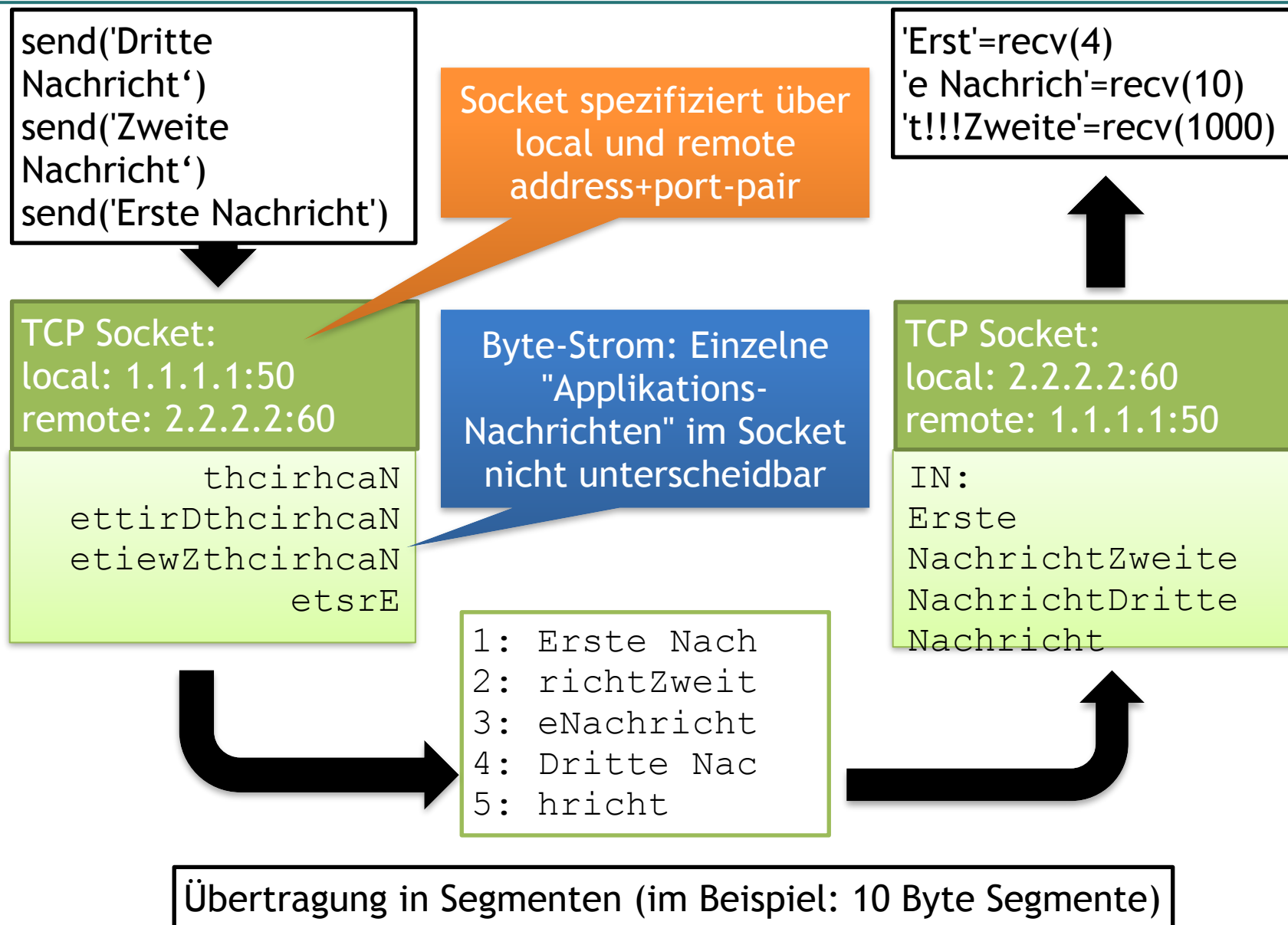


UDP - Socket



- TCP Sockets sind **verbindungsorientiert**
 - die Server-Anwendung muss zunächst einen **Port öffnen** (listen), zu dem die Client-Anwendung eine **Verbindung aufbauen** kann (connect)
 - Three-Way-Handshake: SYN → SYN+ACK → ACK
 - TCP Verbindungen sind **bi-direktional**, über einen TCP-Socket kommuniziert eine Anwendung mit **genau einem anderen** TCP Socket
 - über einen TCP Socket wird ein **Byte-Strom** übertragen
 - der Sender schreibt Nachrichten (Bytes) in den Socket
 - TCP überträgt **asynchron** die Nachrichten in **Segmenten** und schreibt die übertragenen Bytes **vollständig** und in **richtiger Reihenfolge** in den Empfangssocket
 - **gesicherte Übertragung**
 - der Empfänger liest eine beliebige Anzahl von Bytes aus dem Socket
 - in einem Socket ist die Grenze zwischen zwei Nachrichten nicht erkennbar
 - die Anwendung muss **Nachrichten aus dem Byte-Strom filtern** können
 - eindeutiges Ende von Nachrichten
 - Länge der Nachricht im Header

TCP Socket - Bi-Direktionale Übertragung von Byte-Strom



2.1 Grundlagen

2.2 Protokolle und Dienste

2.3 Sockets

2.3.1 Übersicht

2.3.2 Adressierung über Ports

2.3.3 Socket-Programmierung

2.3.4 Live-Coding und Programmierung in Python (Übung)

2.4 Grundlagen der Datenübertragung

2.5 Aufbau des Internets

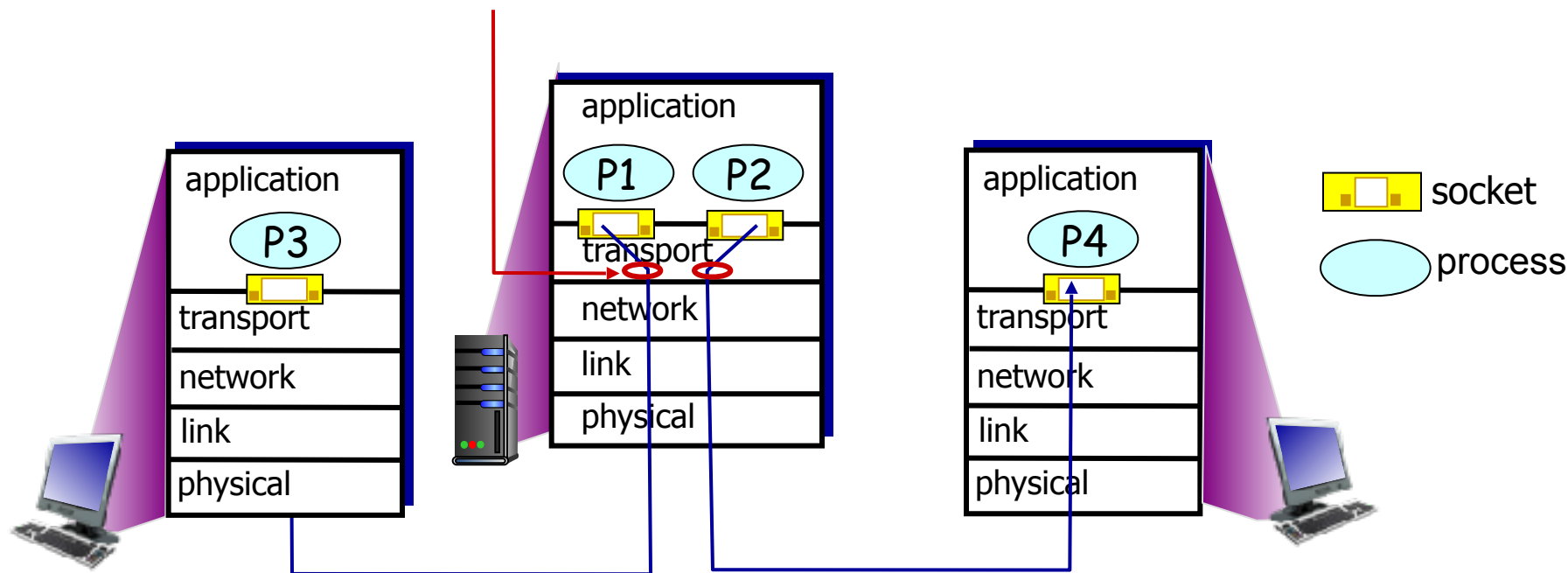
2.6 Zusammenfassung

Transportschicht: Multiplex

- Multiplex/De-Multiplex: Zusammenführen/Trennen mehrerer Datenströme auf einen Datenstrom
- Viele Datenströme zwischen Anwendung und Transportprotokoll, ein Datenstrom zwischen Transportprotokoll und IP

Multiplex am Sender:

Abfertigung von Daten vieler Sockets, Hinzufügen des Transport-Header (horizontale Kommunikation, De-Multiplex)

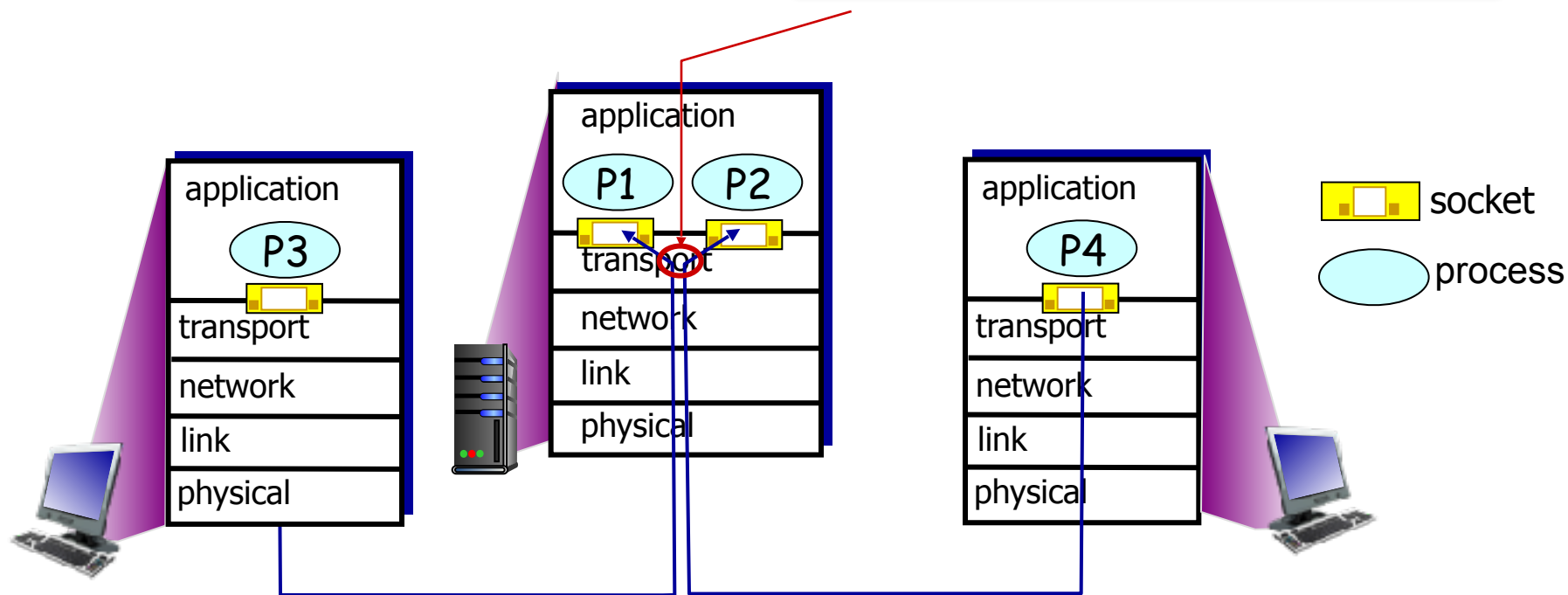


Transportschicht: De-Multiplex

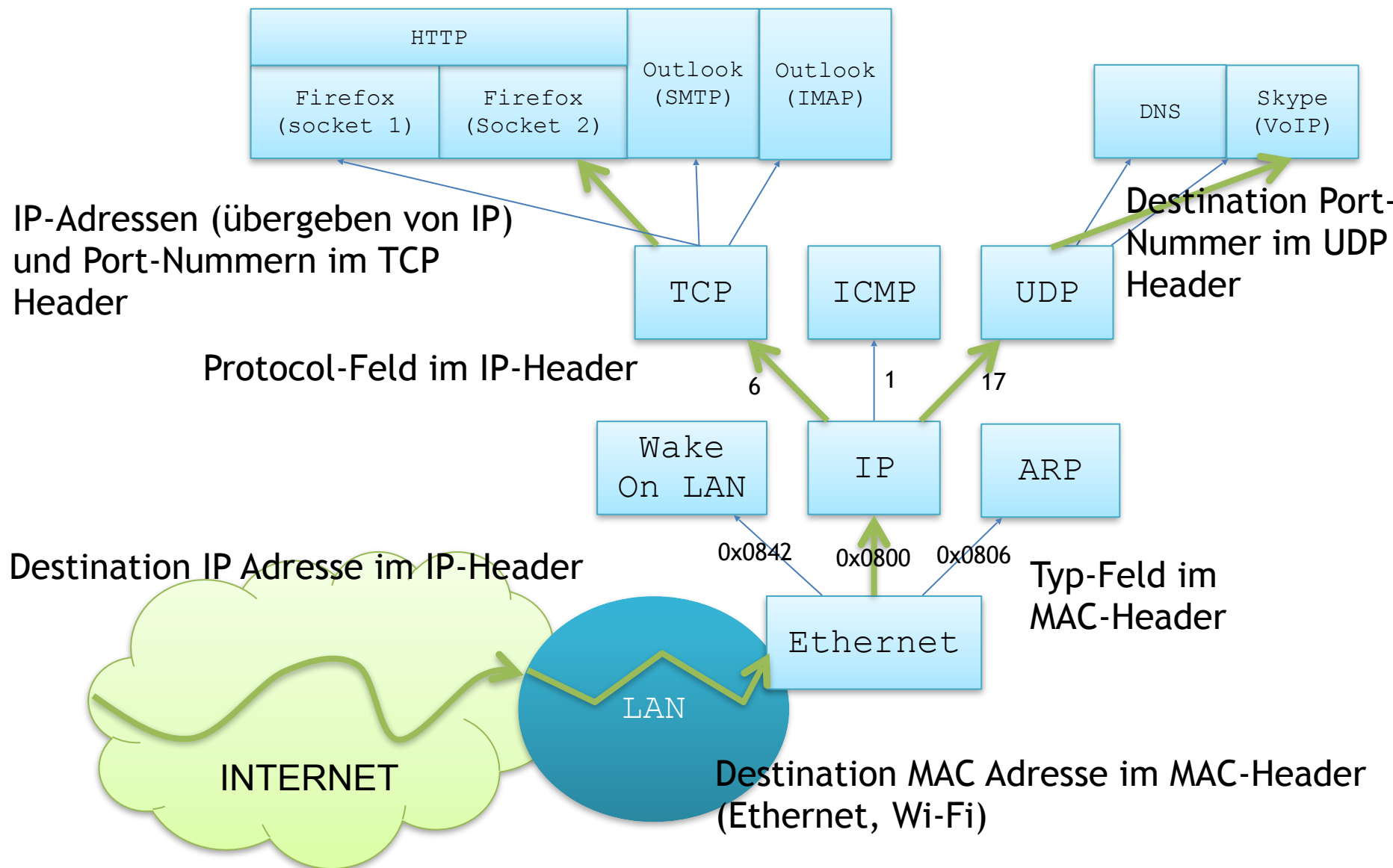
- Multiplex/De-Multiplex: Zusammenführen/Trennen mehrerer Datenströme auf einen Datenstrom
- Viele Datenströme zwischen Anwendung und Transportprotokoll, ein Datenstrom zwischen Transportprotokoll und IP

De-Multiplex am Empfänger:

Ausliefern der Daten an den richtigen Socket, Identifikation über Transport-Header

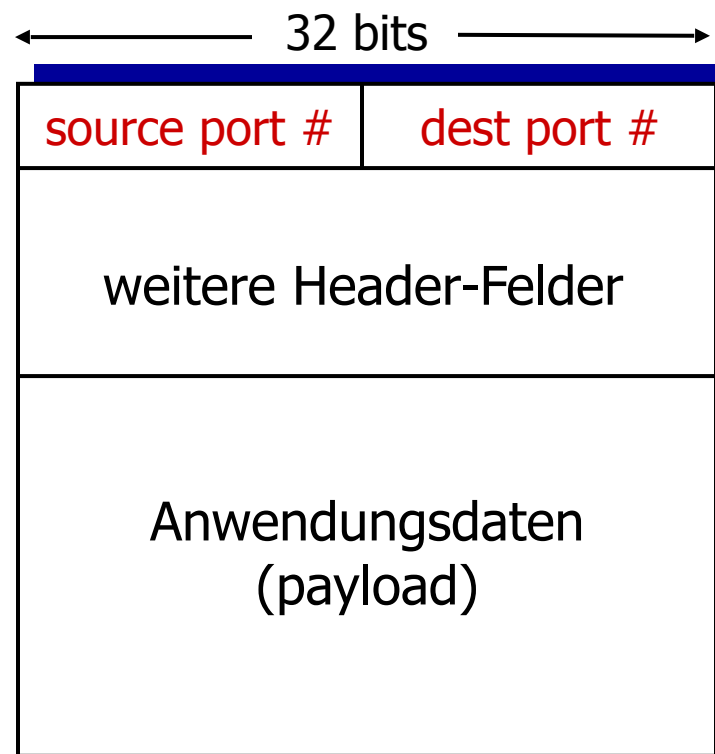


Wo steht das Ziel in einem Paket?



Wie De-Multiplex funktioniert

- Transport-Protokoll identifiziert richtiges Socket über IP Adressen und Ports
- Host empfängt IP Datagramm
 - im IP Header stehen Source und Destination IP Adresse
 - jedes Datagramm enthält ein Segment der Transportschicht
 - im Transport-Header jedes Segments stehen Source und Destination Port
- im Host werden Segmente über IP Adressen und Port Nummern an das richtige Socket (den richtigen Prozess) übergeben

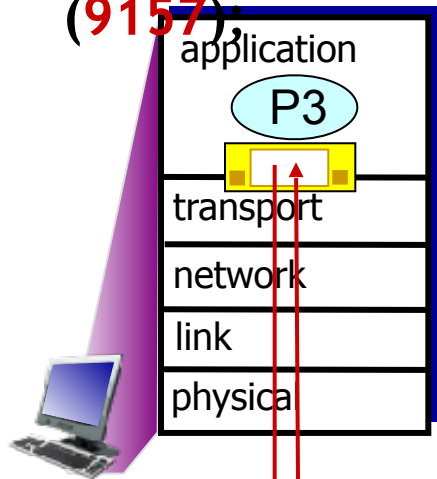


TCP/UDP Segment Format

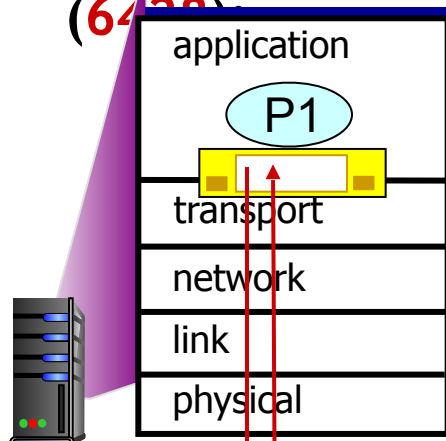
UDP - Demux

- UDP identifiziert Socket NUR über Destination Port
- Pakete mit gleichem Destination Port aber unterschiedlichen IP-Adressen/Source Ports gehen an gleiches Socket

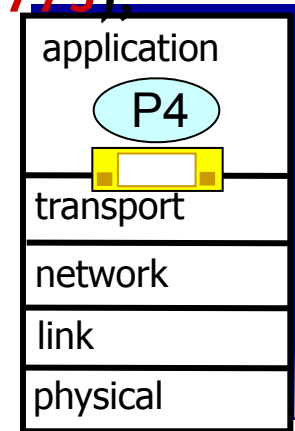
DatagramSocket
mySocket2 = new
DatagramSocket
(9157);



DatagramSocket
serverSocket = new
DatagramSocket
(6428);



DatagramSocket
mySocket1 = new
DatagramSocket
(5775);



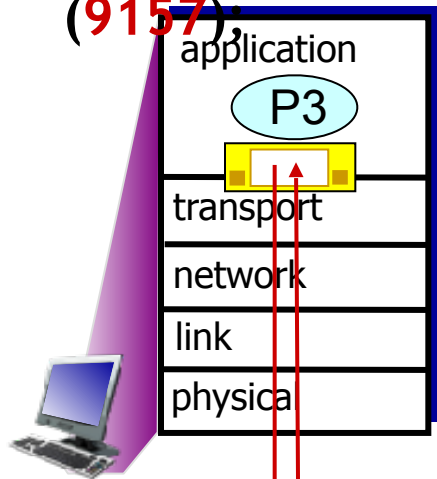
source port: 6428
dest port: 9157

source port: 9157
dest port: 6428

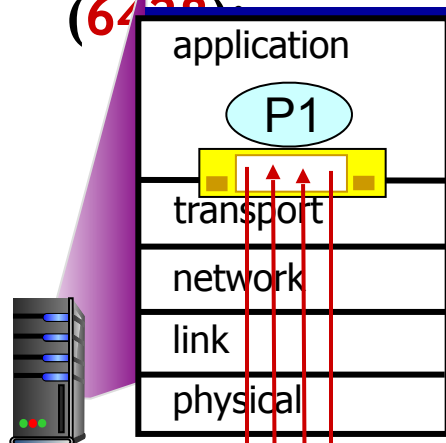
UDP - Demux

- UDP identifiziert Socket NUR über Destination Port
- Pakete mit gleichem Destination Port aber unterschiedlichen IP-Adressen/Source Ports gehen an gleiches Socket

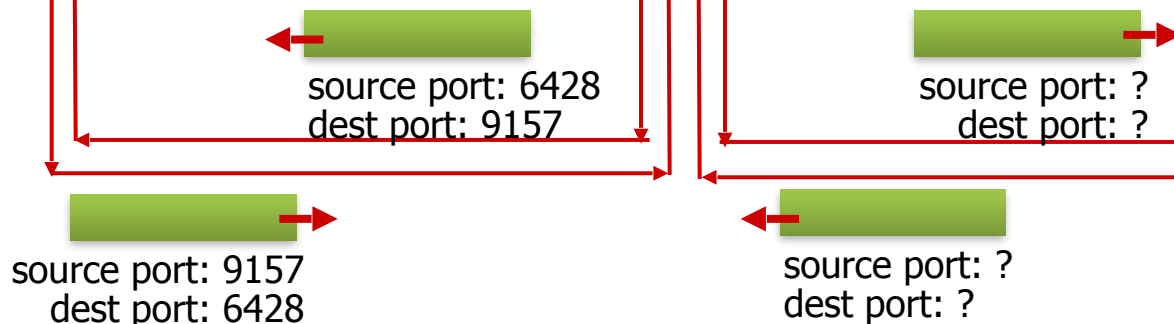
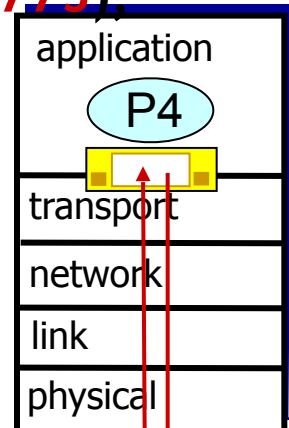
DatagramSocket
`mySocket2 = new
DatagramSocket
(9157);`



DatagramSocket
`serverSocket = new
DatagramSocket
(6428);`



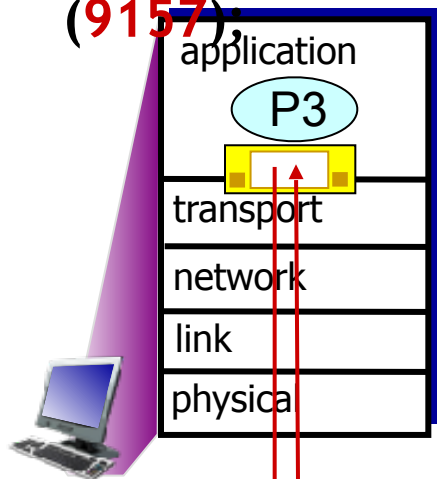
DatagramSocket
`mySocket1 = new
DatagramSocket
(5775);`



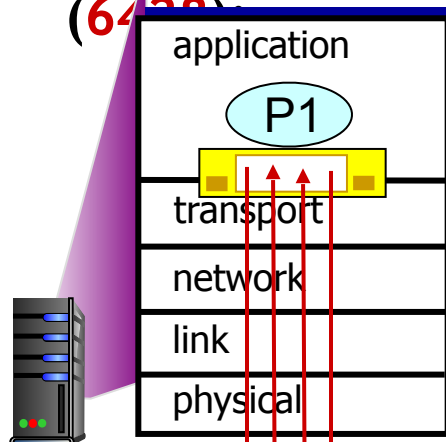
UDP - Demux

- UDP identifiziert Socket NUR über Destination Port
- Pakete mit gleichem Destination Port aber unterschiedlichen IP-Adressen/Source Ports gehen an gleiches Socket

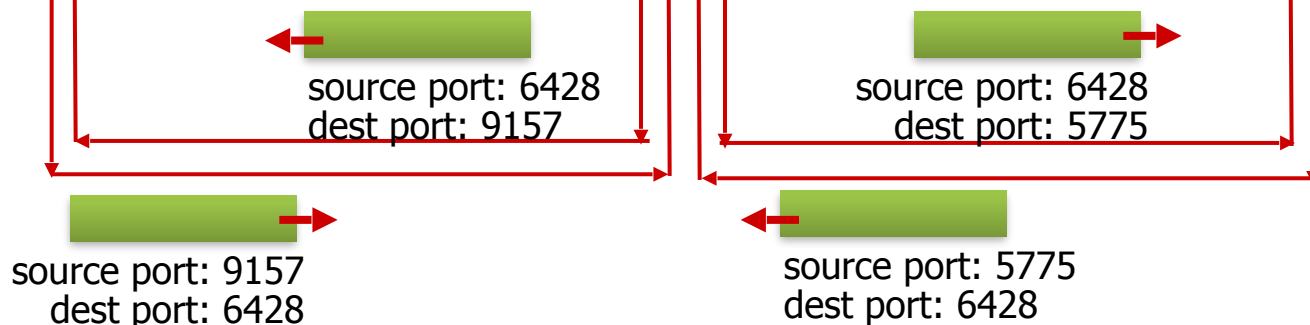
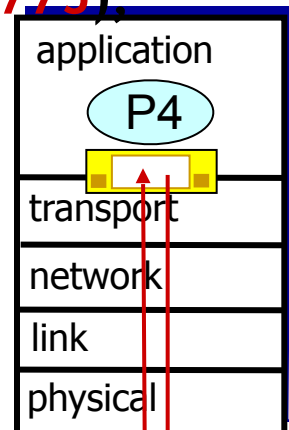
DatagramSocket
`mySocket2 = new
DatagramSocket
(9157);`



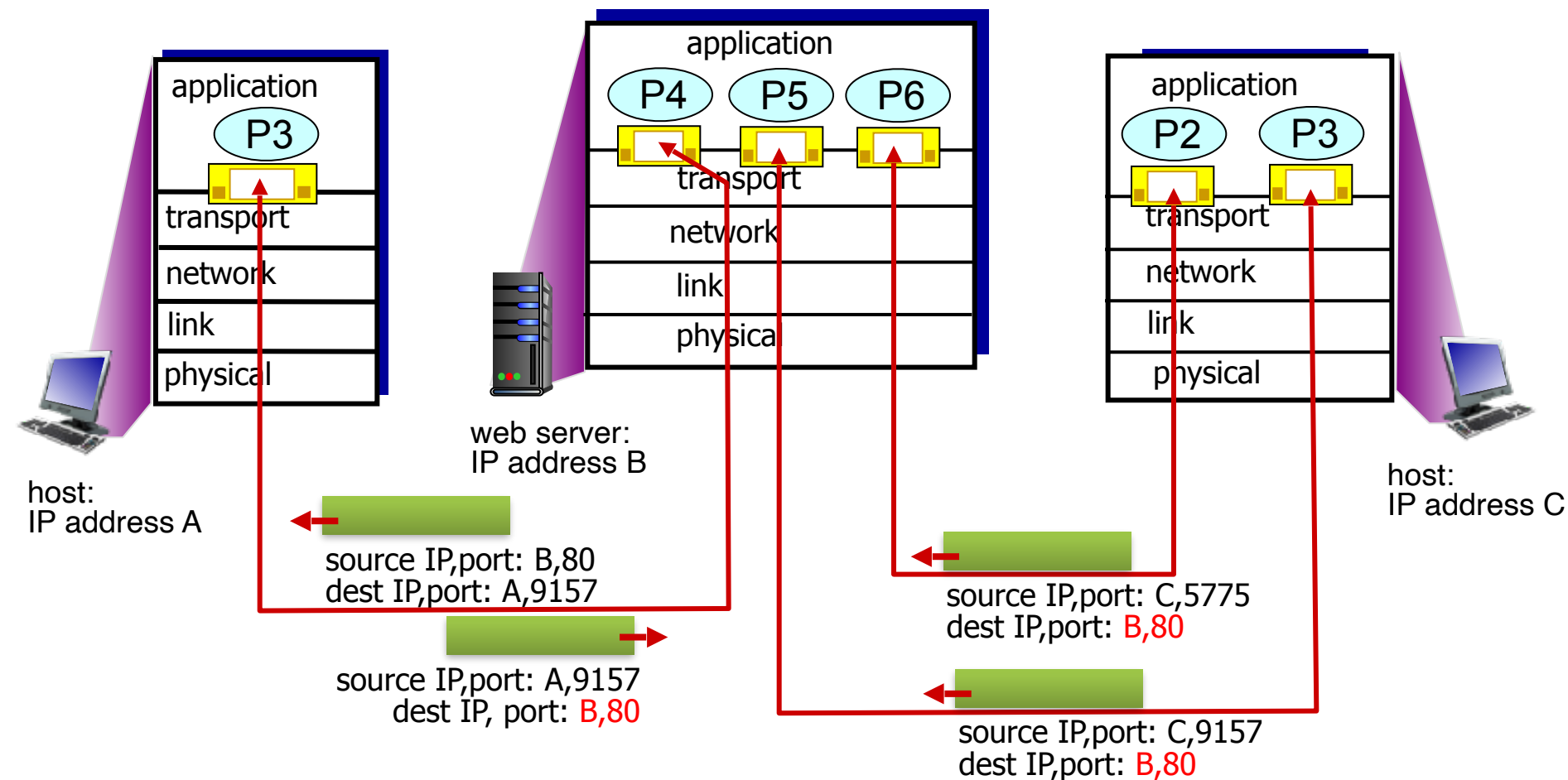
DatagramSocket
`serverSocket = new
DatagramSocket
(6428);`



DatagramSocket
`mySocket1 = new
DatagramSocket
(5775);`



TCP Demux



three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to different sockets

Übersicht der geöffneten Sockets

- Alle Anwendungen auf einem Rechner kommunizieren über Sockets
- **netstat** ist ein unter Windows und Linux verfügbares Tool, um die geöffneten Sockets anzuzeigen
- Das Tool CurrPorts (<https://www.nirsoft.net/utils/cports.html>) stellt die geöffneten Sockets komfortabler in einer Tabelle dar.
- Auch der PacketSniffer WireShark bietet eine Möglichkeit, die geöffneten Sockets und den darüber laufenden Verkehr zu analysieren.
- CurrPorts und WireShark werden Sie in der Übung kennenlernen.

2.1 Grundlagen

2.2 Protokolle und Dienste

2.3 Sockets

2.3.1 Übersicht

2.3.2 Adressierung über Ports

2.3.3 Socket-Programmierung

2.3.4 Live-Coding und Programmierung in Python (Übung)

2.4 Grundlagen der Datenübertragung

2.5 Aufbau des Internets

2.6 Zusammenfassung

- Anwendungsbeispiel
 1. Client erhält eine Zeile mit Zeichen (Daten) über Tastatureingabe und sendet diese an den Server
 2. Der Server erhält die Daten und wandelt alle Zeichen in Großbuchstaben um
 3. Der Server sendet die veränderten Daten an den Client zurück
 4. Der Client empfängt die Zeile und gibt sie auf dem Display aus

2.1 Grundlagen

2.2 Protokolle und Dienste

2.3 Sockets

2.3.1 Übersicht

2.3.2 Adressierung über Ports

2.3.3 Socket-Programmierung

2.3.4 Live-Coding und Programmierung in Python (Übung)

2.3.4.1 Datagram (UDP) Sockets

2.3.4.2 Stream (TCP) Sockets

2.3.4.3 Tools und Tipps

2.4 Grundlagen der Datenübertragung

2.5 Aufbau des Internets

2.6 Zusammenfassung

Übertragung eines UDP-Datagramms

Server: IP a.b.c.d

1. Server öffnet UDP Socket **serverSocket** mit Port= x

5. Server liest das Datagramm aus dem **serverSocket**

6. Server schreibt die Antwort in den **serverSocket** und spezifiziert Client-Adresse und -Portnummer

Client

2. Client öffnet UDP Socket **clientSocket**

3. Client erstellt Datagramm mit Ziel-IP=a.b.c.d und Ziel-Port=x
4. Client schreibt das Datagramm in den **clientSocket**

6. Client liest das Datagramm aus dem **clientSocket**

7. Client schließt den **clientSocket**

Beispiel: UDP Client in Python

include Python's socket
library

from socket import *
serverName = 'hostname'
serverPort = 12000

Address Family=Internet
SocketType=Datagram

create UDP socket for
server

clientSocket = socket(socket.AF_INET,
socket.SOCK_DGRAM)

get user keyboard
input

message = raw_input('Input lowercase sentence:')

Attach server name, port to
message; send into socket

clientSocket.sendto(message,(serverName, serverPort))

read reply characters from
socket into string

modifiedMessage, serverAddress =
clientSocket.recvfrom(2048)

print out received string
and close socket

print modifiedMessage
clientSocket.close()

Beispiel: UDP Server in Python

```
from socket import *  
serverPort = 12000
```

create UDP socket → `serverSocket = socket(AF_INET, SOCK_DGRAM)`
bind socket to local port
number 12000 → `serverSocket.bind(("", serverPort))`
`print "The server is ready to receive"`

loop forever → `while 1:`

read from UDP socket into
message, getting client's
address (client IP and port) → `message, clientAddress = serverSocket.recvfrom(2048)`

create
upper case string → `modifiedMessage = message.upper()`

send upper case string
back to this client → `serverSocket.sendto(modifiedMessage, clientAddress)`

2.1 Grundlagen

2.2 Protokolle und Dienste

2.3 Sockets

2.3.1 Übersicht

2.3.2 Adressierung über Ports

2.3.3 Socket-Programmierung

2.3.4 Live-Coding und Programmierung in Python (Übung)

2.3.4.1 Datagram (UDP) Sockets

2.3.4.2 Stream (TCP) Sockets

2.3.4.3 Tools und Tipps

2.4 Grundlagen der Datenübertragung

2.5 Aufbau des Internets

2.6 Zusammenfassung

Übertragung eines TCP-Bytestroms

Server: IP a.b.c.d

1. Server öffnet Server-Socket
serverSocket mit Port= x

2. Server wartet auf Anfragen

5. Server akzeptiert TCP-Verbindung
und öffnet **connectionSocket**

7. Server liest Anfrage als "Bytestrom"
aus **connectionSocket**

8. Server schreibt Antwort in
connectionSocket

Client

3. Client öffnet TCP Client-Socket
clientSocket mit Ziel-IP=a.b.c.d und
Ziel-Port=x

4. Client initiiert Aufbau einer TCP-
Verbindung

6. Client schreibt Anfrage in den
clientSocket

6. Client liest Antwort als Bytestrom
aus dem **clientSocket**


Handshake



Beispiel: TCP Client in Python

```
from socket import *  
serverName = 'servername'  
serverPort = 12000
```

Address Family=Internet
SocketType=Stream



```
clientSocket = socket(AF_INET, SOCK_STREAM)  
clientSocket.connect((serverName,serverPort))  
sentence = raw_input('Input lowercase sentence:')  
clientSocket.send(sentence)  
modifiedSentence = clientSocket.recv(1024)  
print 'From Server:', modifiedSentence  
clientSocket.close()
```

create TCP socket for
server, remote port 12000



No need to attach server
name, port



Beispiel: Server-Anwendung in Python für TCP Socket

```
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 1:
    connectionSocket, addr = serverSocket.accept()

    sentence = connectionSocket.recv(1024)
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence)

    connectionSocket.close()
```

create TCP welcoming socket →

server begins listening for incoming TCP requests →

loop forever →

server waits on accept() for incoming requests, new socket created on return →

read bytes from socket (but not address as in UDP) →

close connection to this client (but *not* welcoming socket) →

2.1 Grundlagen

2.2 Protokolle und Dienste

2.3 Sockets

2.3.1 Übersicht

2.3.2 Adressierung über Ports

2.3.3 Socket-Programmierung

2.3.4 Live-Coding und Programmierung in Python (Übung)

2.3.4.1 Datagram (UDP) Sockets

2.3.4.2 Stream (TCP) Sockets

2.3.4.3 Tools und Tipps

2.4 Grundlagen der Datenübertragung

2.5 Aufbau des Internets

2.6 Zusammenfassung

- Wireshark - <https://www.wireshark.org/>
 - most popular packet sniffer
 - captures packet from network interface
 - displays and analyzes packet traces
- Rawcap - <https://www.netresec.com/?page=RawCap>
 - packet sniffer for windows
 - command line tool
 - capable of capturing packets on localhost interface (127.0.0.1)
 - packet trace can be viewed by Wireshark
- CurrPorts - <https://www.nirsoft.net/utils/cports.html>
 - displays information on open TCP and UDP ports
 - combined information from Windows tools
 - netstat for viewing open ports
 - taskmanager for process information

Socket Programmierung in Python

- Einige Socket-Befehle sind blockierend, d.h. das Programm läuft erst weiter, wenn die erwarteten Pakete eintreffen
 - `accept` - wartet auf Verbindungswunsch (→`connect`)
 - `recv` - warten auf Daten im Socket (→`send`)
- Umgang mit blockierenden Socket-Befehlen
 - Verwendung von Timeouts
 - Timeouts bewirken, dass ein blockierender Befehl nach einer bestimmten Zeit mit einer „`socket.timeout`“-Exception beendet wird
 - Abfangen von Fehlern
 - Ausführung von Code mit einem „`try: ... except: ...`“ Statement bewirkt, dass ein Fehler abgefangen wird und nicht zu einem unkontrollierten Programmabsturz führt
 - Hilfreich ist vor allem das Abfangen von „vorhersehbaren“ Fehlern wie Timeouts oder nicht-erfolgreichen Verbindungsversuchen
 - um die Kontrolle über das Programm zu erhalten, kann beispielsweise ein `recv()` nach einer Sekunde über einen Timeout beendet und dann wieder gestartet werden
 - nebenläufige Programmierung mit Threads

Socket Programmierung in Python

- Programme müssen gelegentlich auf mehrere externe Ereignisse warten und es ist nicht absehbar, in welcher Reihenfolge diese auftreten
- Beispiel Chat:
 - Ein Chat-Client öffnet einen Socket zu jedem aktiven Buddy. Jeder Buddy kann zu beliebiger Zeit Nachrichten schicken, die aus dem Socket gelesen und unmittelbar ausgegeben werden sollen. Auch im Eingabefenster kann der Nutzer jederzeit eine Nachricht eingeben und versenden.
 - Ein „händisches“ serielles Lesen aller Sockets ist mit Hilfe von Timeouts zwar möglich aber sehr umständlich
- Besser ist es, pro Socket einen Thread zu starten, der die Eingabe liest und verarbeitet bzw. zur Verarbeitung einen neuen Thread startet, um unmittelbar auf neue Daten reagieren zu können
 - in Python bietet das Modul „threading“ Interfaces für die Programmierung mit Threads
 - Kern ist die Klasse „Thread“, deren Objekte als Threads gestartet werden
- Alternativ dazu kann in Python das Modul „selectors“ genutzt werden, um mehrere I/O-Events zu behandeln.