

2.1 Terminologie

2.2 Grundlagen der Datenübertragung

2.3 Protokolle und Dienste

2.4 Sockets

2.4.1 Übersicht

2.4.2 Adressierung über Ports

2.4.3 Socket-Programmierung (Übung)

2.4.4 Live-Coding und Programmierung in Python (Übung)

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

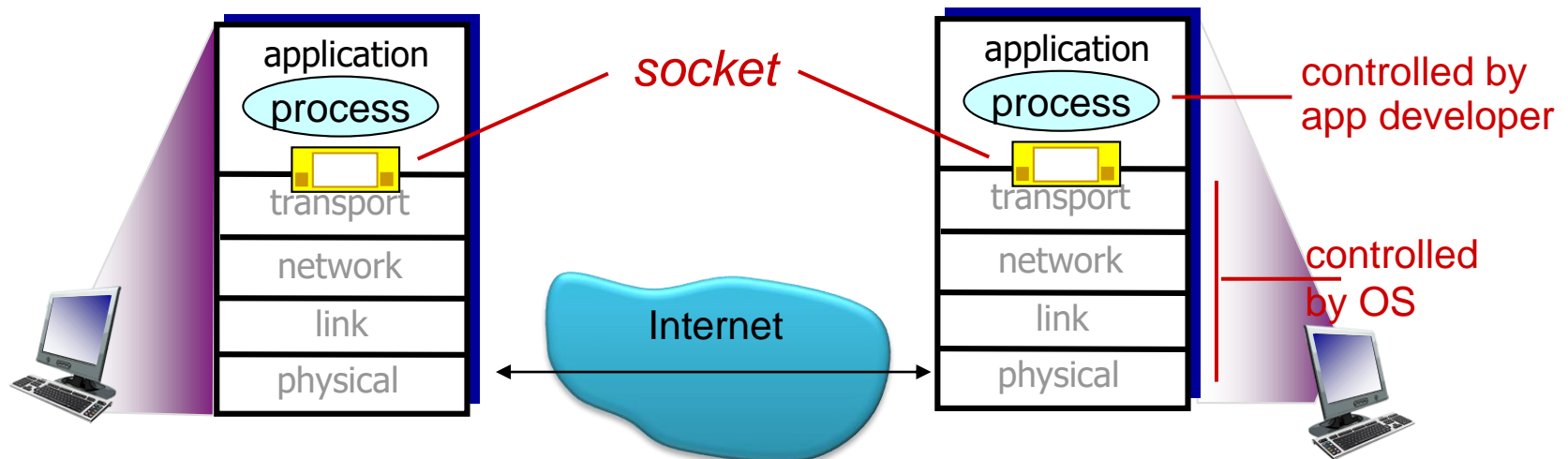
Server-Prozess: Prozess, der darauf wartet, kontaktiert zu werden

Anmerkung:

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

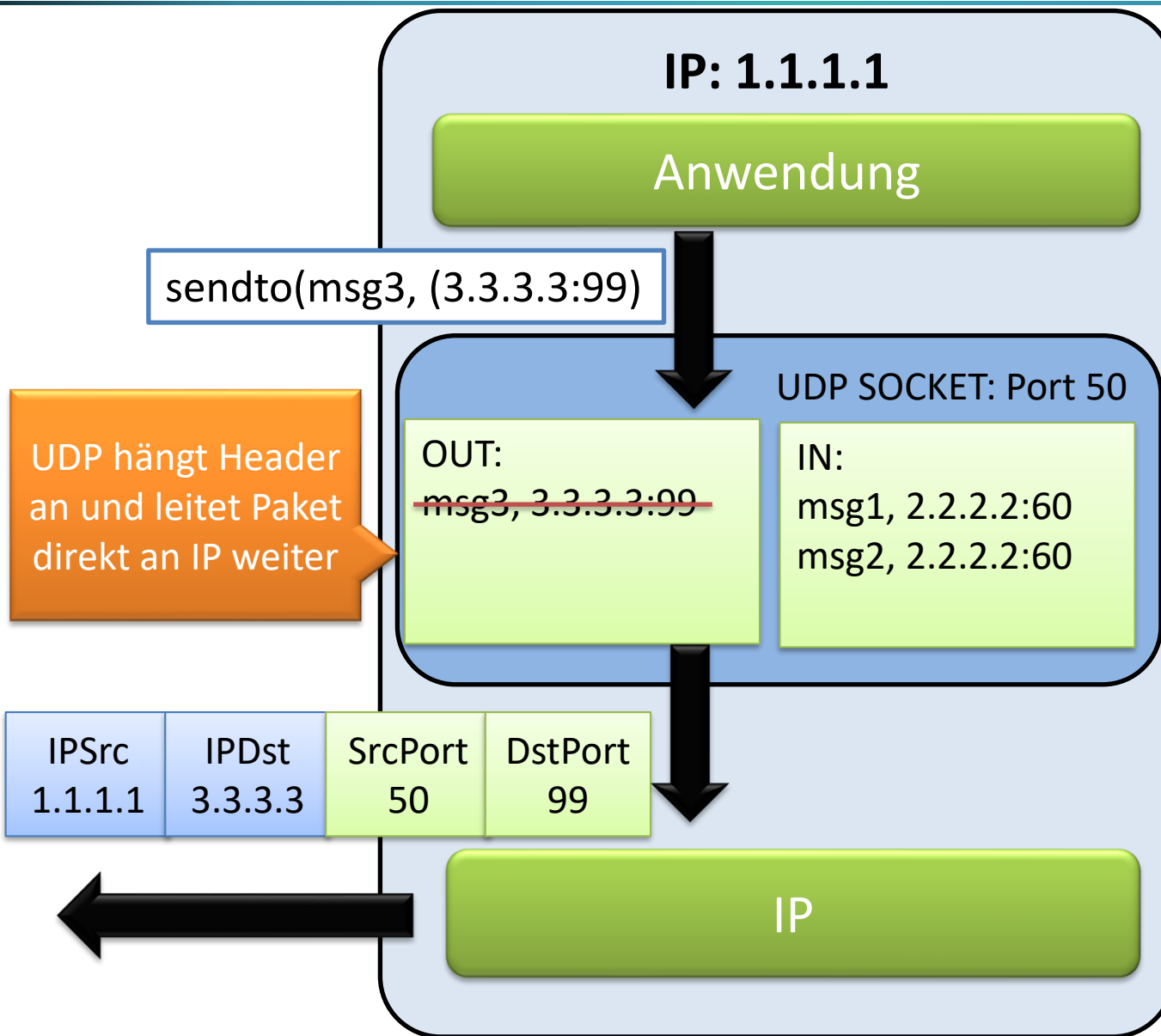
Sockets

- 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

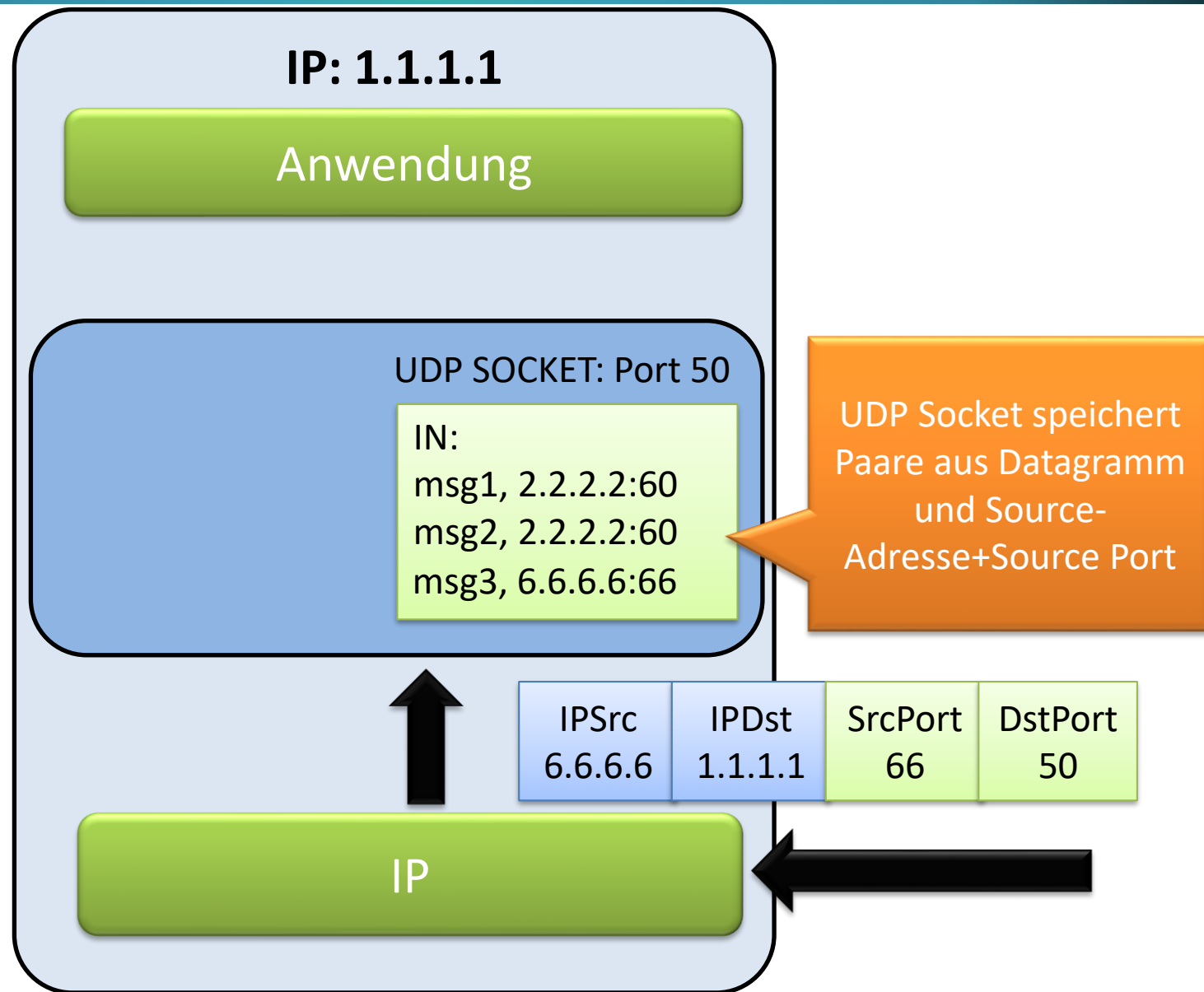


- Die wichtigsten vom Betriebssystem bereitgestellten Sockets sind **UDP** (User Datagram Protocol) Sockets und **TCP** (Transmission Control Protocol) Sockets
- UDP Sockets sind **verbindungslos**
 - über UDP Sockets können direkt Nachrichten versendet werden, ohne eine Verbindung aufzubauen
 - eine 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

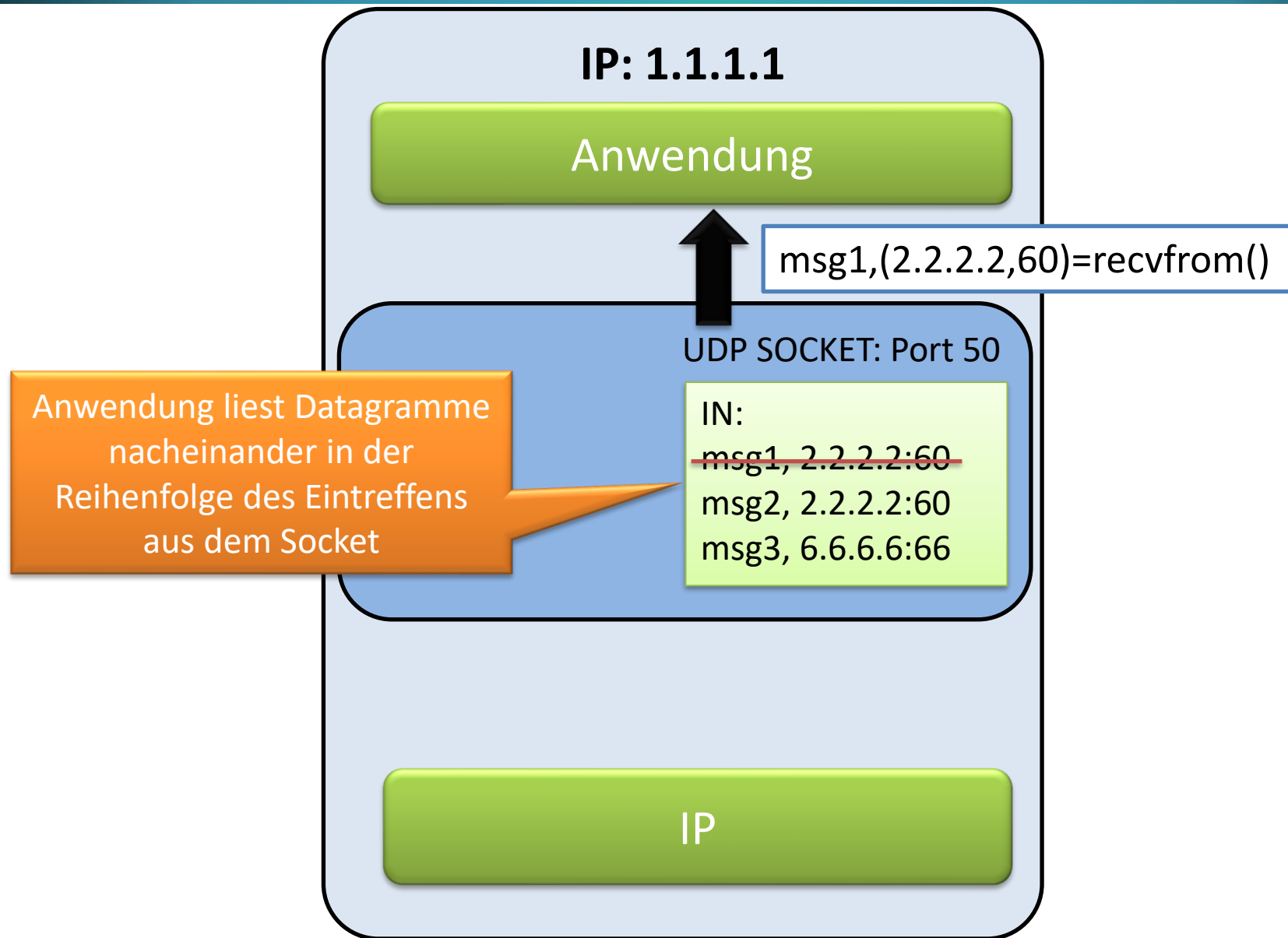
UDP – Socket



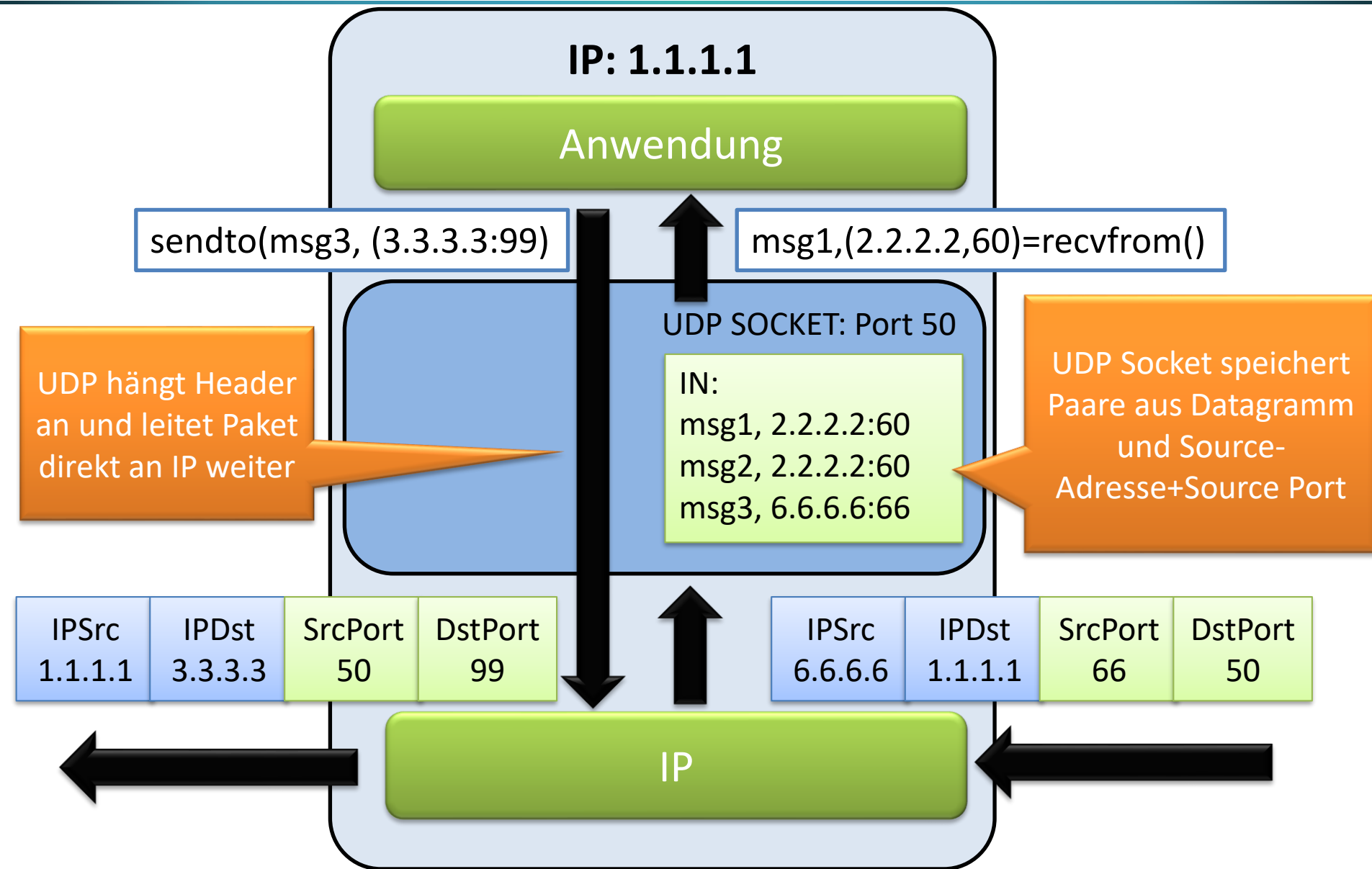
UDP – Socket



UDP – Socket

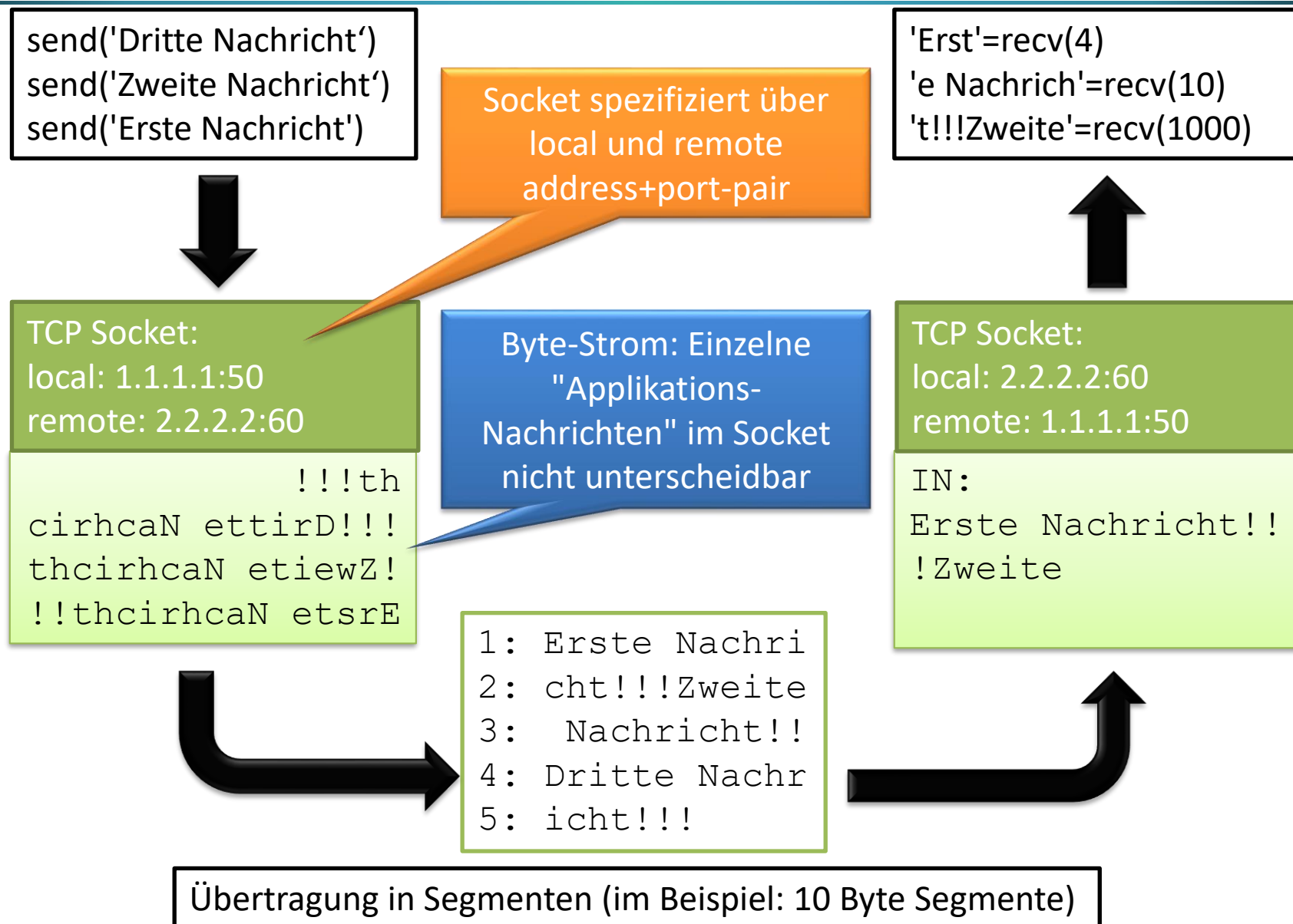


UDP – Socket



- 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 Terminologie

2.2 Grundlagen der Datenübertragung

2.3 Protokolle und Dienste

2.4 Sockets

2.4.1 Übersicht

2.4.2 Adressierung über Ports

2.4.3 Socket-Programmierung (Übung)

2.4.4 Live-Coding und Programmierung in Python (Übung)

2.5 Aufbau des Internets

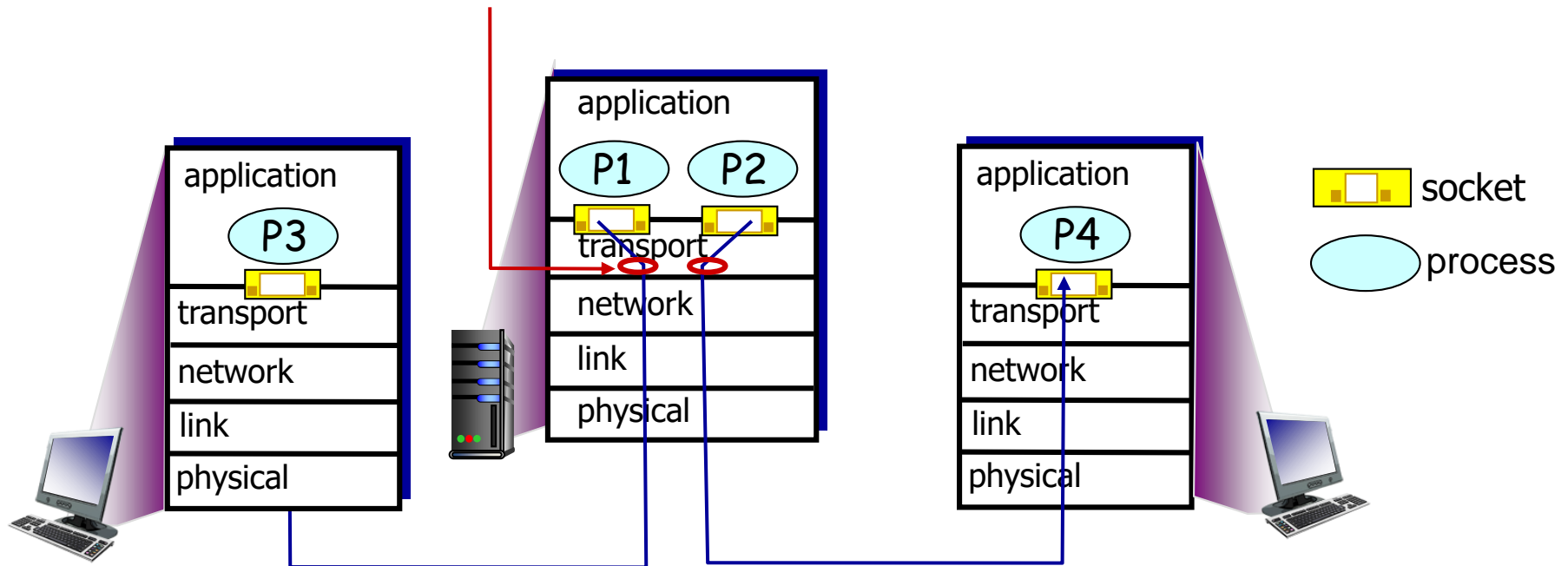
2.6 Zusammenfassung

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)

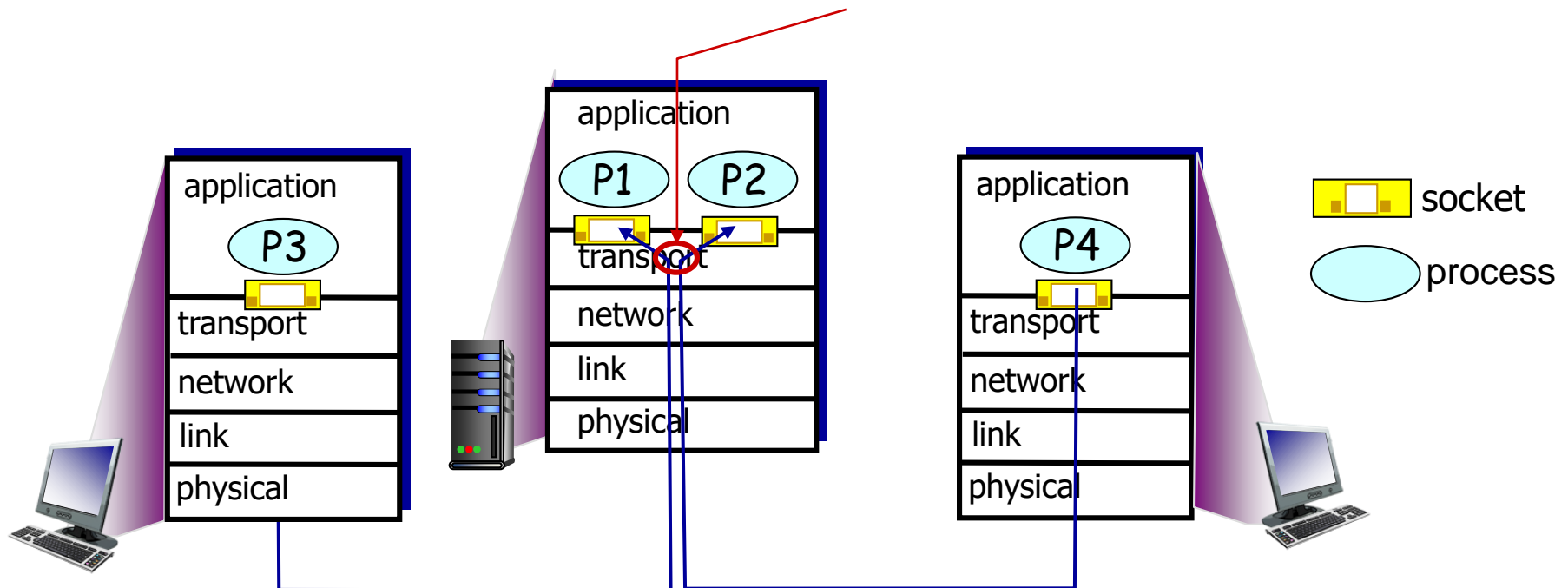


Transportschicht: De-Multiplex

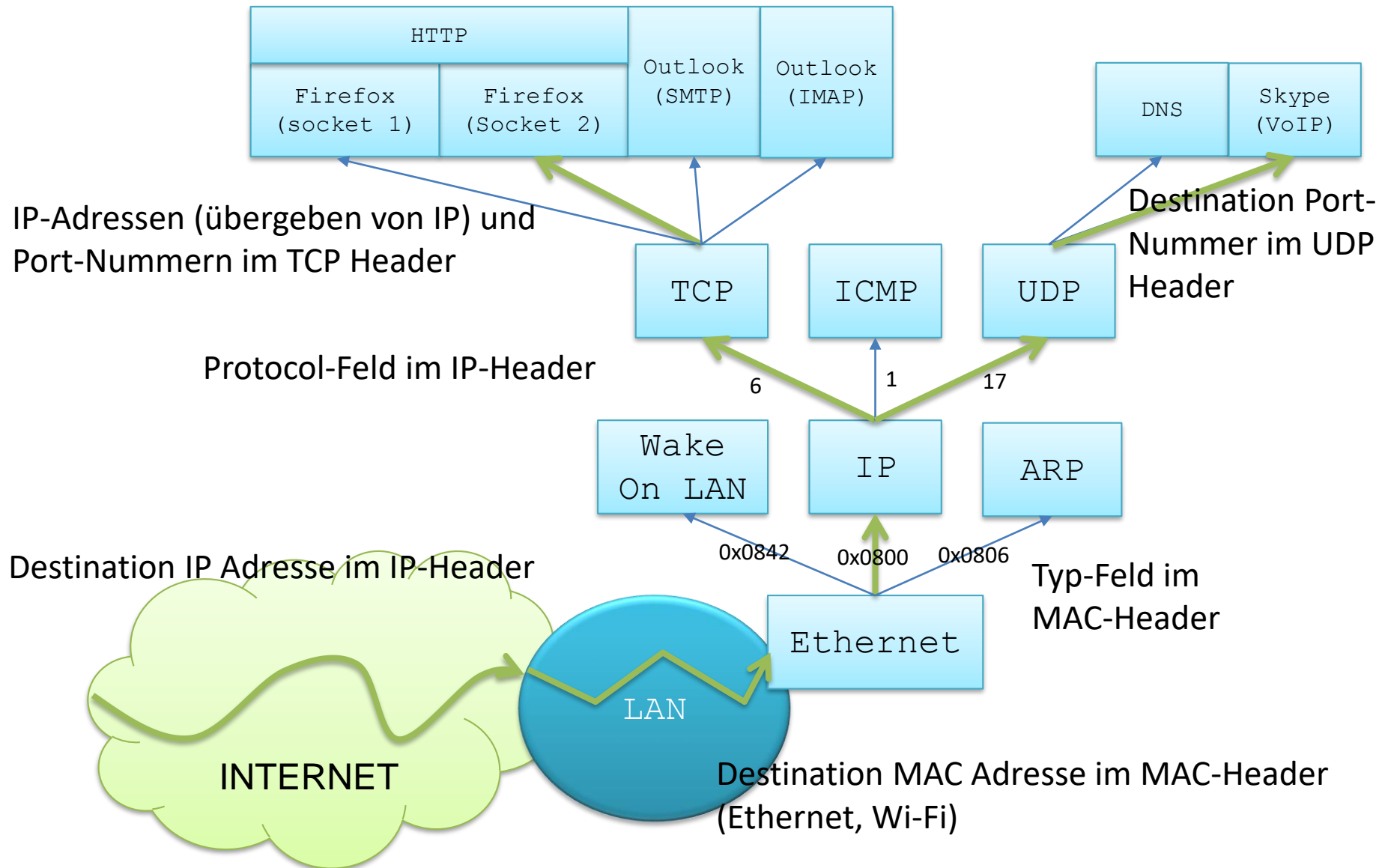
- Multiplex/De-Multiplex: Zusammenführen/Trennen mehrerer Datenströme auf einen Datenstrom
- Viele Datenströme zwischen Anwendung und Transportprotokoll, ein Datenstrom zwischen Transportprotokoll und IP

De-Multiplex am Empfänger:

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

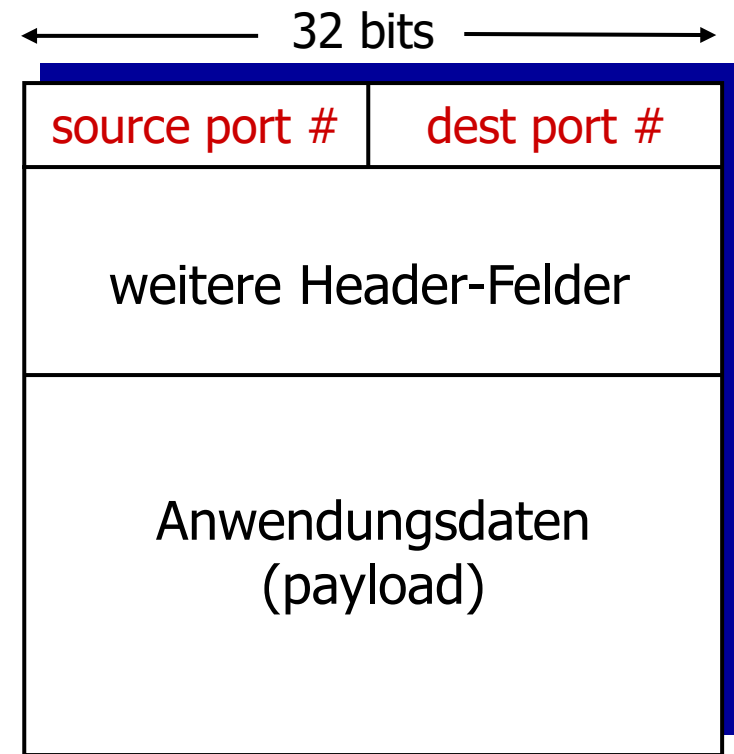


Wo steht das Ziel in einem Paket?



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



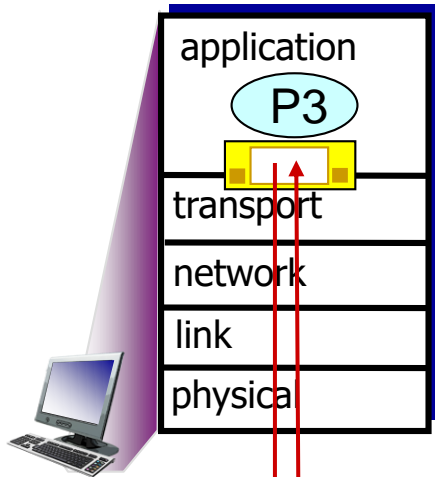
TCP/UDP Segment Format

UDP - Demux

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

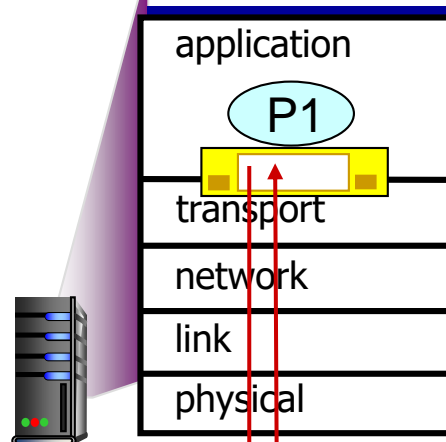
DatagramSocket

```
mySocket2 = new  
DatagramSocket (9157);
```



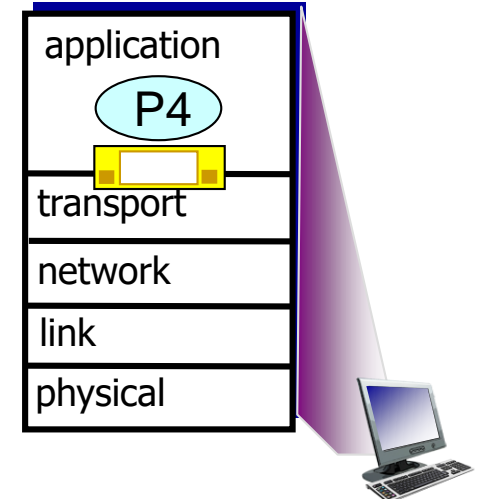
DatagramSocket

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



DatagramSocket

```
mySocket1 = new  
DatagramSocket (5775);
```

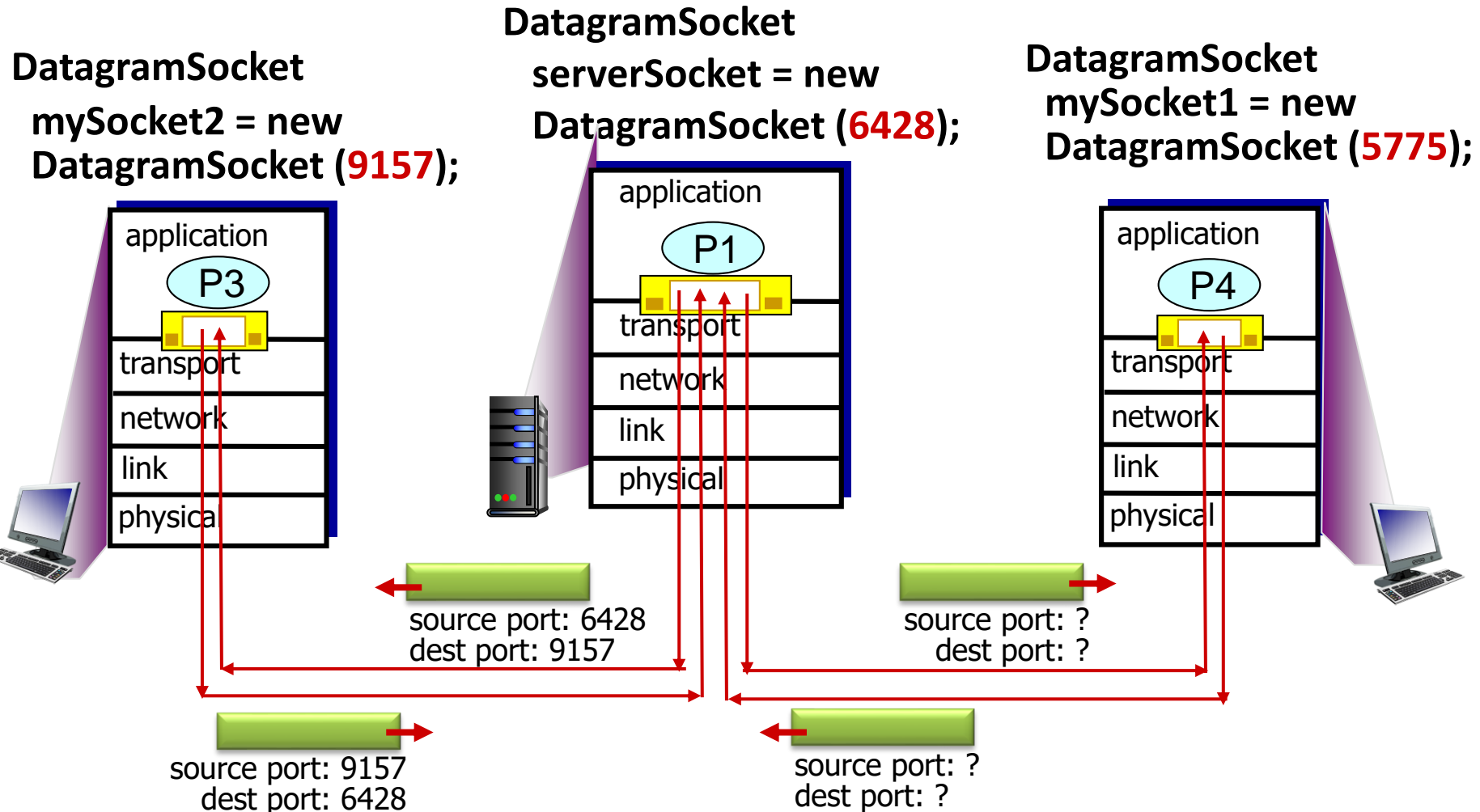


source port: 6428
dest port: 9157

source port: 9157
dest port: 6428

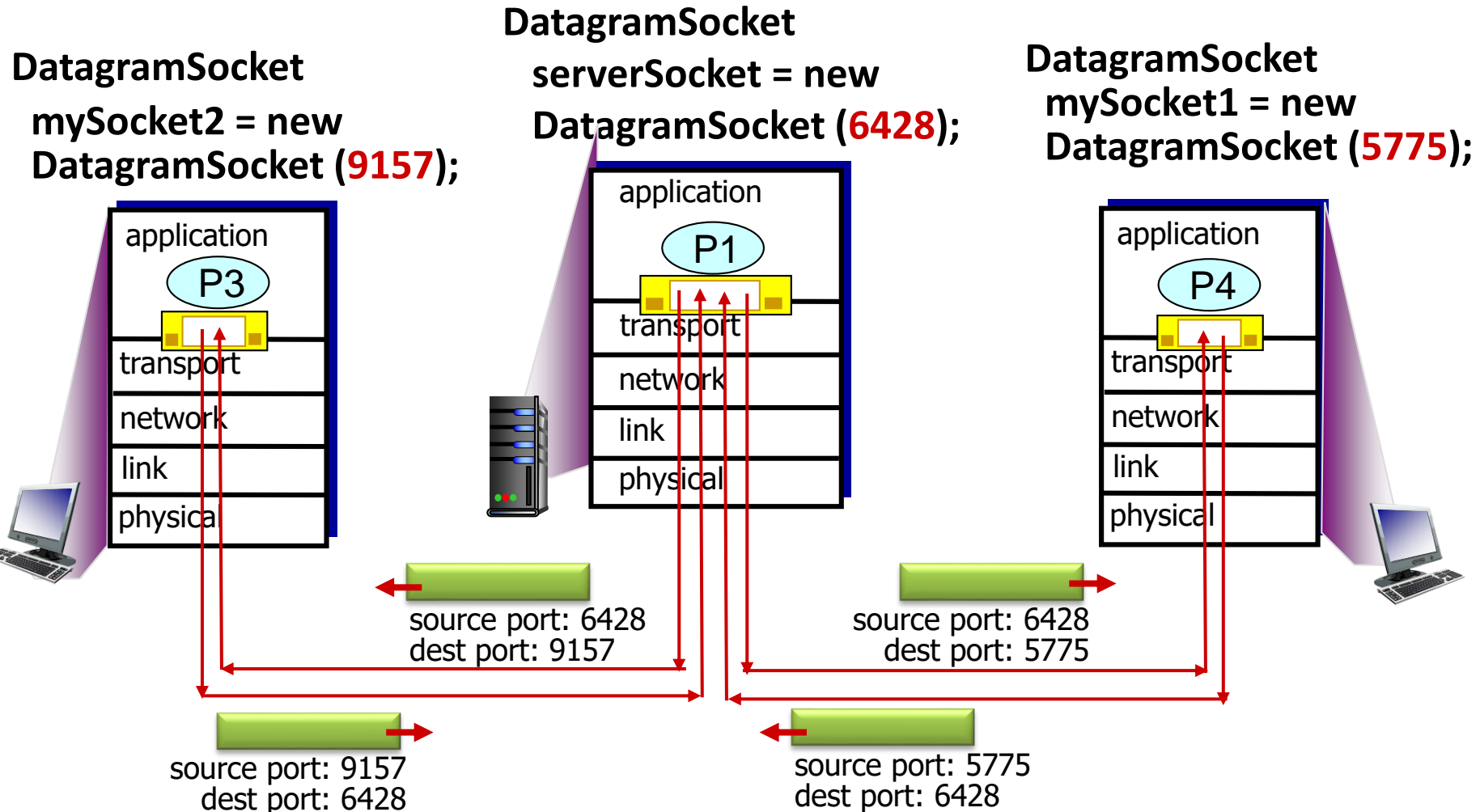
UDP - Demux

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

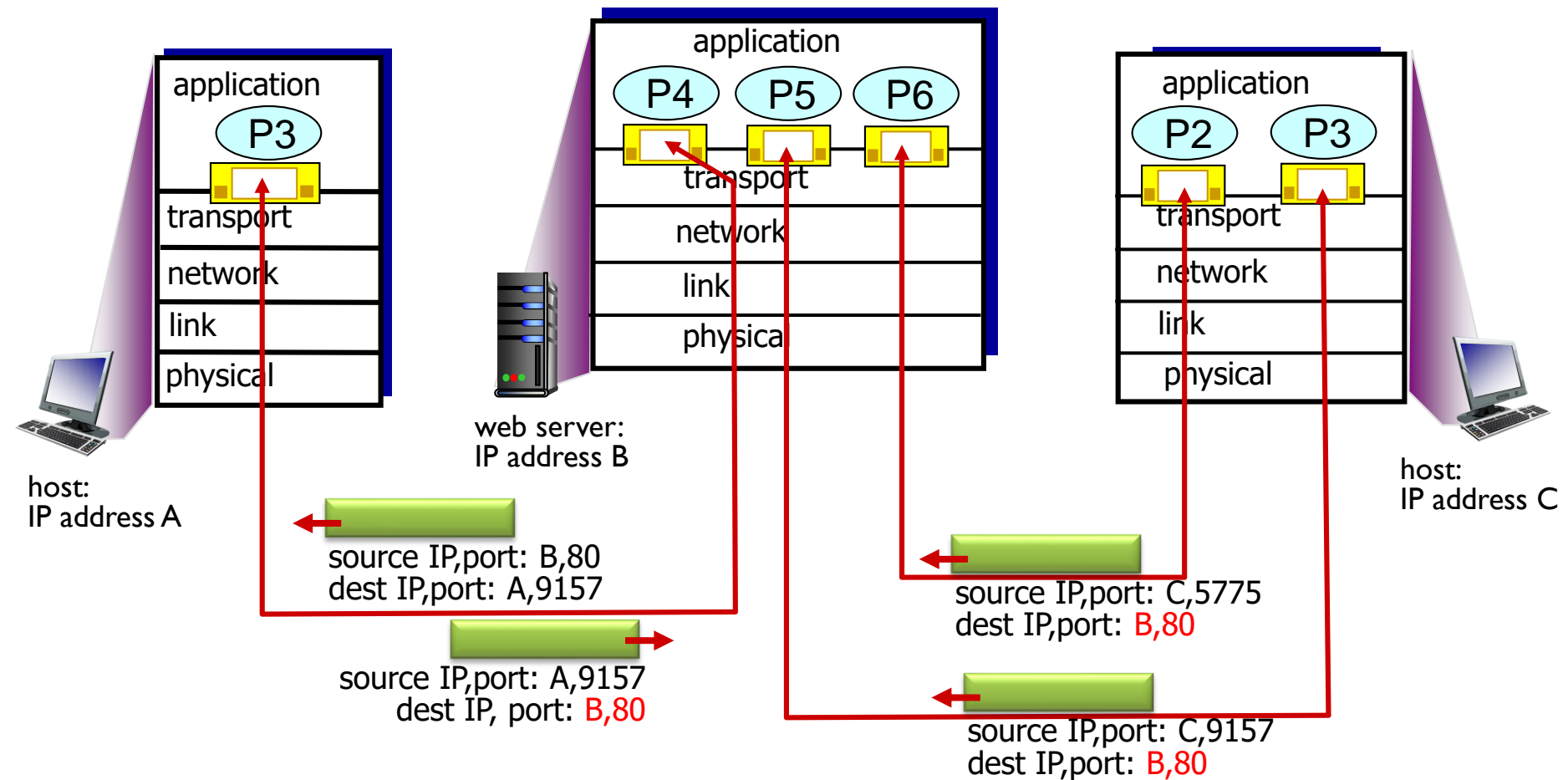


UDP - Demux

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



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 Terminologie

2.2 Grundlagen der Datenübertragung

2.3 Protokolle und Dienste

2.4 Sockets

2.4.1 Übersicht

2.4.2 Adressierung über Ports

2.4.3 Socket-Programmierung (Übung)

2.4.4 Live-Coding und Programmierung in Python (Übung)

2.5 Aufbau des Internets

2.6 Zusammenfassung

- Zwei verschiedene Arten von Sockets für die beiden Transportprotokolle
 - Datagram Socket für UDP
 - unzuverlässiger Transport eines Datagramms
 - ein Socket zur Kommunikation mit mehreren Remote-Sockets
 - Stream Socket für TCP
 - zuverlässiger Transport eines Bytestroms in einer Verbindung
 - Verbindungsauf- und abbau
 - Paketverluste werden erkannt und durch Übertragungswiederholung korrigiert
 - in falscher Reihenfolge empfangene Bytes (Segmente) werden am Empfänger wieder in die richtige Reihenfolge gebracht
 - ein Socket kommuniziert mit genau einem Remote-Socket
 - Listening-Socket zur Annahme von Verbindungswünschen

Datagram Sockets (UDP)

UDP: keine "Verbindung" zwischen Client und Server

- Daten werden direkt gesendet, vorher keine Kontaktaufnahme (Handshaking) zwischen Client und Server
 - Erstes Paket enthält direkt Daten
- Beim Senden (Schreiben in den Socket) werden explizit mit jedem Datagramm auch die IP-Adresse und Port-Nummer des Ziels angegeben
- Empfänger extrahiert die IP Adresse und Port-Nummer des Senders aus den empfangenen Pakete

UDP: übertragene Daten können verlorengehen oder in veränderter Reihenfolge (out-of-order) empfangen werden

UDP aus Sicht der Applikation:

- UDP bietet unzuverlässige Übertragung eines Byte-Blocks (Datagramm) zwischen Client und Server
- Anwendung muss sich selbst um Datenverlust und Wiederherstellung der Reihenfolge der Daten kümmern

UDP Socket (Sender)

UDP Socket: 100.100.100.100:9000

Nachricht 3 → 3.3.3.3:3000



`sock.sendto(Nachricht3,(2.2.2.2,2000))`

Nachricht 2 → 1.2.3.4:4000

Nachricht 1 → 1.2.3.4:1000



UDP Datagram

Ausgehender UDP Socket verwaltet Paare von Nachrichten und Zieladressen. Über einen UDP Socket können Nachrichten an verschiedene Ziele gesendet werden.

UDP Datagramm

Source: 100.100.100.100:9000

Destination: 1.2.3.4:1000

Payload: Nachricht 1

UDP Socket (Empfänger)

UDP Socket: 1.2.3.4:1000

Nachricht 1 ← 100.100.100.100:9000

nachricht,(ip,port) = sock.recvfrom()

Nachricht 5 ← 2.2.2.2:6000

Nachricht 7 → 4.3.2.1:99

Eingehender UDP Socket verwaltet Paare von eingehenden Nachrichten und Absenderadressen. Über einen UDP Socket werden Nachrichten von mehreren sendenden Sockets empfangen werden.

UDP Datagram

UDP Datagram
Source: 4.3.2.1:99
Destination: 1.2.3.4:1000
Payload: Nachricht 7

Stream Sockets (TCP)

TCP: Verbindung zwischen Client und Server

- vor dem Senden der ersten Daten wird mittels Three-Way-Handshake eine Verbindung von einem Client zu einem Server (IP,Port) aufgebaut
- nach einem erfolgreichen Verbindungsaufbau bestehen auf Client- und Serverseite jeweils ein Socket, die nur miteinander bidirektional kommunizieren können
 - beide Seiten können asynchron Bytes in den Socket schreiben, die von TCP so schnell wie möglich (best effort) übertragen werden
 - beide Seiten können asynchron Bytes aus dem Socket lesen, die nach dem korrekten Empfang in der richtigen Reihenfolge zur Verfügung gestellt werden

TCP: Bytes kommen immer vollständig und in der korrekten Reihenfolge an

TCP aus Sicht der Applikation:

- zuverlässige Übertragung eines Byte-Stroms zwischen Client und Server
- Anwendung muss sich um Aufbau, Abbau und unerwarteten Abbruch der Verbindung kümmern
- Anwendung muss sich nicht um Datenverlust und Wiederherstellung der Reihenfolge der Daten kümmern

TCP Socket (Sender)

TCP Socket: 1.2.3.4:10↔5.6.7.8:50

Nachricht: **Rechnernetze!!!**

sock.send(Nachricht)

!!!eztenren
hceR gnuselroV red n
i nemmoklliW!!!ollaH

Ausgehender TCP Socket enthält zu sendende Bytes, die in Segmenten übertragen werden. Die Segmentgröße ist durch die Maximum Segment Size (MSS) beschränkt.

TCP Segment

TCP Segment (hier MSS:20 Bytes)
Source: 1.2.3.4:10
Destination: 5.6.7.8:50
Payload: **Hallo!!!Willkommen i**

TCP Socket (Empfänger)

TCP Socket: 5.6.7.8:50 ↔ 1.2.3.4:10

Nachricht 3: Hall



nachricht=sock.recv(4)

Begrenzte Anzahl von Bytes (hier:4) werden aus dem Socket gelesen

Hallo!!! Willkommen i
n der Vorlesung Rechn
ernetze!!!

Empfangender TCP Socket enthält zu empfangene Bytes in der richtigen Reihenfolge. Anwendung liest eine begrenzte Anzahl von Bytes aus dem Socket.



TCP Segment

TCP Segment (hier MSS:20 Bytes)
Source: 1.2.3.4:10
Destination: 5.6.7.8:50
Payload: ernetze!!!

2.1 Terminologie

2.2 Grundlagen der Datenübertragung

2.3 Protokolle und Dienste

2.4 Sockets

2.4.1 Übersicht

2.4.2 Adressierung über Ports

2.4.3 Socket-Programmierung (Übung)

2.4.4 Live-Coding und Programmierung in Python (Übung)

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	→	<code>serverSocket = socket(AF_INET, SOCK_DGRAM)</code>
bind socket to local port number 12000	→	<code>serverSocket.bind(('', serverPort))</code> <code>print "The server is ready to receive"</code>
loop forever	→	<code>while 1:</code>
read from UDP socket into message, getting client's address (client IP and port)	→	<code>message, clientAddress = serverSocket.recvfrom(2048)</code>
create upper case string	→	<code>modifiedMessage = message.upper()</code>
send upper case string back to this client	→	<code>serverSocket.sendto(modifiedMessage, clientAddress)</code>

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

Aufbau eines TCP-Sockets

Vorgehen im Client:

- Öffnen eines Client-TCP-Sockets
- Spezifikation von IP-Adresse und Portnummer des Server-prozesses
- Öffnen eines Client-TCP-Sockets bewirkt Aufbau einer TCP-Verbindung zum Server-Prozess her

TCP aus Sicht der Applikation:

- TCP stellt einen zuverlässigen, reihenfolgeerhaltenden Transfer eines Bytestroms zwischen Client und Server zur Verfügung

Vorgehen im Server

- Server-Prozess öffnet einen Socket, der Client-Anfragen entgegennimmt
- Wenn der Server-Prozess von einem Client kontaktiert wird, dann erzeugt er einen neuen Socket, um mit diesem Client zu kommunizieren
 - Server einen Socket pro Client und kann mit mehreren Clients kommunizieren
 - Unterscheidung der Sockets durch IP-Adresse und Port-Nummer der Clients
 - ein Client kann mehrere Sockets mit unterschiedlichen Port-Nummern zu einem Web-Server öffnen

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

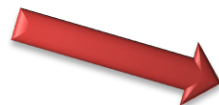
3. Client öffnet TCP Client-Socket
clientSocket mit Ziel-IP=a.b.c.d und
Ziel-Port=x

4. Client initiiert Aufbau einer TCP-
Verbindung

6. Client schreibt Anfrage in den
clientSocket

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

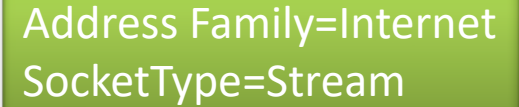
Handshake



Beispiel: TCP Client in Python

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

Address Family=Internet
SocketType=Stream



create TCP socket for
server, remote port 12000 →

```
clientSocket = socket(AF_INET, SOCK_STREAM)  
clientSocket.connect((serverName,serverPort))  
sentence = raw_input('Input lowercase sentence:')  
clientSocket.send(sentence)
```

No need to attach server
name, port →

```
modifiedSentence = clientSocket.recv(1024)  
print 'From Server:', modifiedSentence  
clientSocket.close()
```

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) →

Python – Live-Coding – TCP Sockets

```
import socket
serv_sock=socket.socket ...
    (socket.AF_INET,socket.SOCK_STREAM)
serv_sock.bind((' ',50000))
serv_sock.listen(1)
```

```
conn,addr=serv_sock.accept()
conn.send(b"Hi I'm your upper/lowercase converter.
Message format U|L:<message>")
conn.recv(2)
```

```
msg=conn.recv(1500)
conn.send(...
    (msg.decode('utf-8')).upper()).encode('utf-8'))
```

```
conn.recv(2)
msg=conn.recv(1500)
conn.send(...
    (msg.decode('utf-8')).upper()).encode('utf-8'))
```

```
conn.recv(2)
conn.close()
```

```
import socket
client_sock=socket.socket ...
    (socket.AF_INET, socket.SOCK_STREAM)
client_sock.connect(('127.0.0.1', 50000))
```

```
client_sock.recv(1500)
client_sock.send(b'U:lower')
client_sock.recv(1500)
```

```
client_sock.send(b'L:')
client_sock.send(b'UPPER\n')
client_sock.send(b'L:CONT\n')
```

```
client_sock.recv(1500)
client_sock.send(b'C:')
client_sock.close()
```


Python – Live-Coding – UDP Sockets

```
udp_sock=socket.socket ...
    (socket.AF_INET,socket.SOCK_DGRAM)
udp_sock.bind(('',50000))
```

```
msg,addr=udp_sock.recvfrom(1500)
udp_sock.sendto(...
    (msg.decode('utf-8')).upper()).encode('utf-8'),...
    addr)
```

```
msg,addr=udp_sock.recvfrom(2)
msg,addr=udp_sock.recvfrom(1500)
udp_sock.sendto(...
    (msg.decode('utf-8')).upper()).encode('utf-8'),...
    addr)
```

```
msg,addr=udp_sock.recvfrom(1500)
udp_sock.sendto(...
    (msg.decode('utf-8')).upper()).encode('utf-8'),...
    addr)
```

```
msg,addr=udp_sock.recvfrom(1500)
```

```
udp_sock=socket.socket ...
    (socket.AF_INET, socket.SOCK_DGRAM)
udp_sock.sendto(...
    (b'U:lower'),('127.0.0.1', 50000))
```

```
msg,addr=udp_sock.recvfrom(1500)
udp_sock.sendto(...
    (b'L:UPPER\n'),('127.0.0.1', 50000))
udp_sock.sendto(...
    (b'L:CONT\n'),('127.0.0.1', 50000))
```

```
msg,addr=udp_sock.recvfrom(1500)
msg,addr=udp_sock.recvfrom(1500)
udp_sock.sendto(...
    (b'Bye'),('127.0.0.1', 50000))
udp_sock.close()
```

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

- 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.