

„Systemnahe Programmierung“ (BA-INF 034) Wintersemester 2023/2024

Dr. Matthias Frank, Dr. Matthias Wübbeling

Institut für Informatik 4
Universität Bonn

E-Mail: {matthew, matthias.wuebbeling}@cs.uni-bonn.de

Sprechstunde: nach der (Online) Vorlesung bzw. nach Vereinbarung

3. Netzwerkprogrammierung in C

Bisher (Programmiervorlesungen ADiP + OOSE) wurden hauptsächlich **eigenständige Programme** betrachtet, die ihre **Eingaben vom Nutzer** (Tastatur, Maus, ...) oder **aus Dateien** beziehen sowie **Ausgaben auf dem Bildschirm** oder ebenfalls **Dateien** erzeugen.

Nun sollen die **Programme** (bzw. der/die Programmierer/in) in die Lage versetzt werden, **mit anderen Programmen** bzw. Prozessen (auf anderen Rechnern) **zu kommunizieren**.

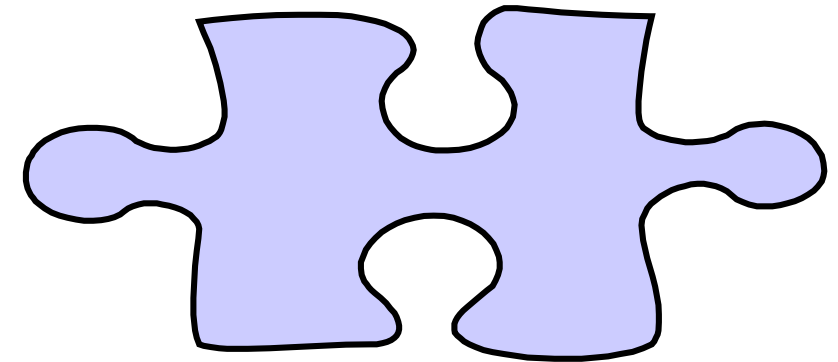
3.1. Motivation

3.2. Netzwerkprogrammierung: Sockets & Co.

3.3. I/O-Multiplexing

3.4. Server-Strukturen

3.5. Zusammenfassung



3. Netzwerkprogrammierung in C
„ein Programm möchte über
ein Netzwerk kommunizieren“

Literaturhinweise

Markus Zahn

Unix-Netzwerkprogrammierung mit Threads, Sockets und SSL

Verlag: Springer, Berlin; Auflage: 1 (August 2006)

Sprache: Deutsch

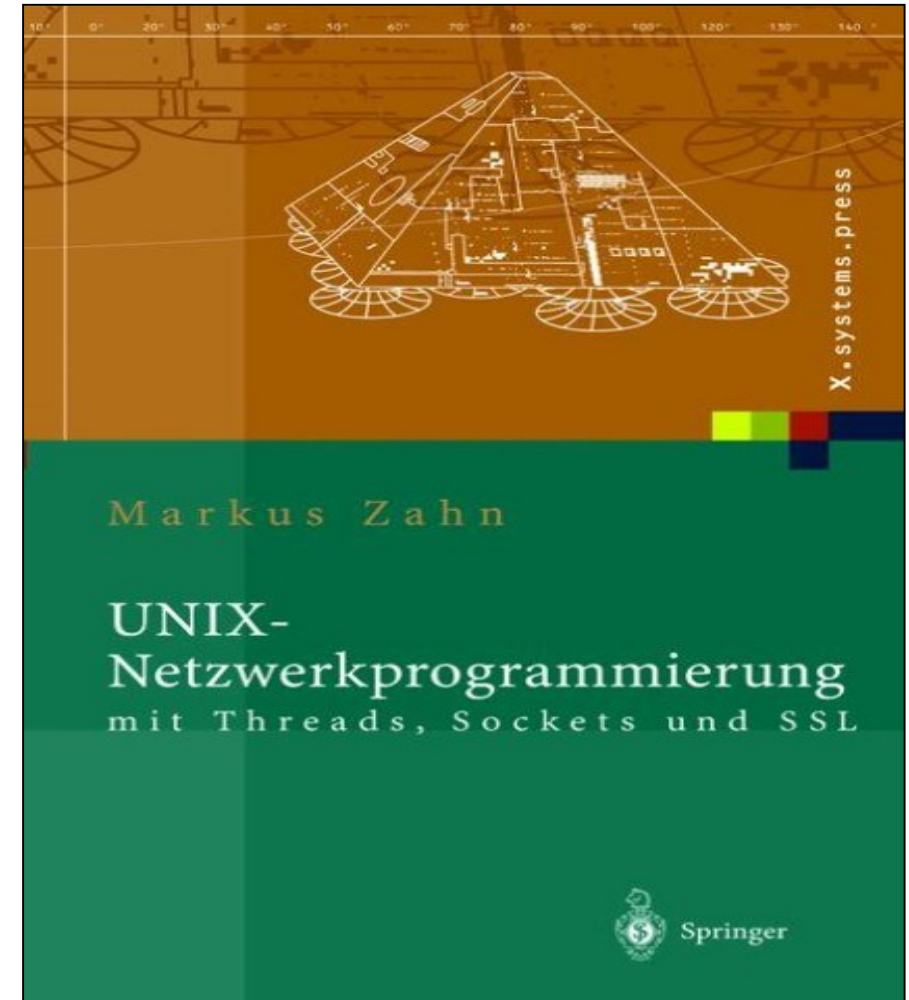
ISBN-10: 3540002995

ISBN-13: 978-3540002994

Inhaltliche Korrekturen zum Buch
sowie Sourcecodes aller Beispielprogramme
online erhältlich unter
<http://unp.bit-oase.de/>

Zusätzlich:

- diverse [Tutorials im Internet](#) auffindbar
- Stichworte „Sockets“, „Network Programming“, „Netzwerkprogrammierung“, ...



Literaturhinweise

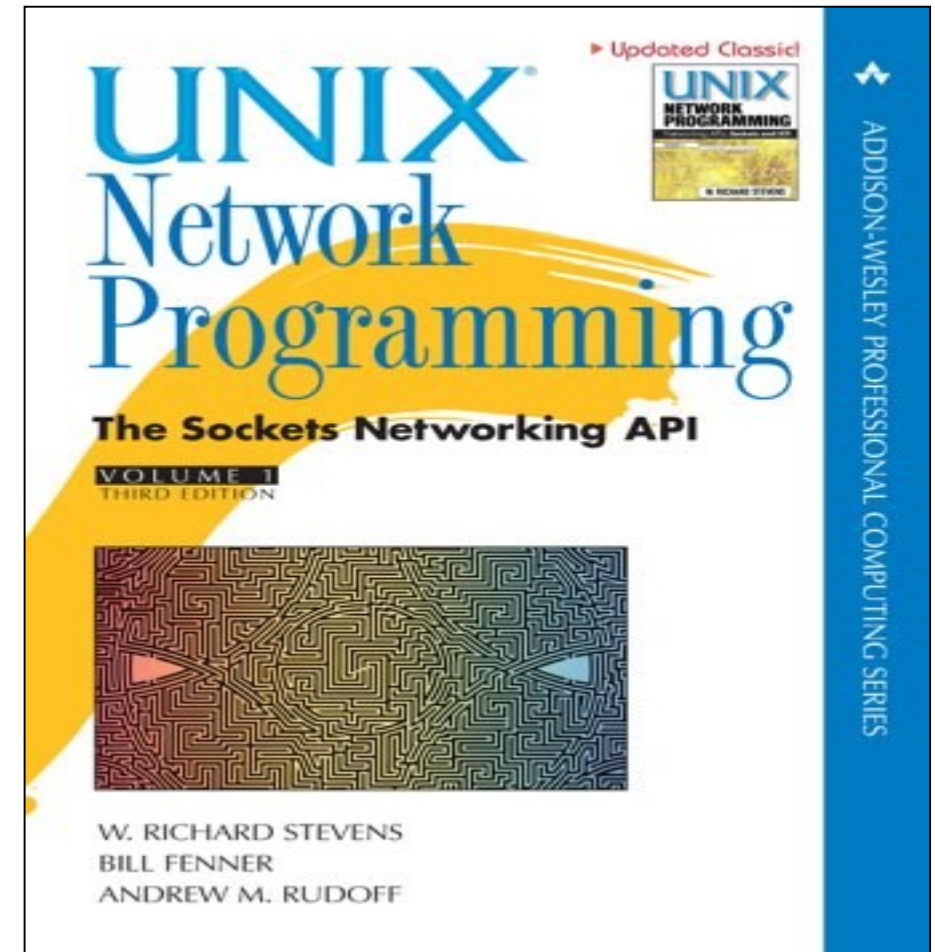
Ein Klassiker:

W. Richard Stevens (et al.)
UNIX Network Programming
The Sockets Networking API
(Third Edition, November 2003)
ISBN: 0131411551

(auch als Deutsche Ausgabe bei
Hanser Fachbuch, Februar 2000)

Zusätzlich:

- diverse [Tutorials im Internet](#) auffindbar
- Stichworte „Sockets“, „Network Programming“, „Netzwerkprogrammierung“, ...



3.1. Motivation für Netzwerkkommunikation

Allgemeines Ziel:

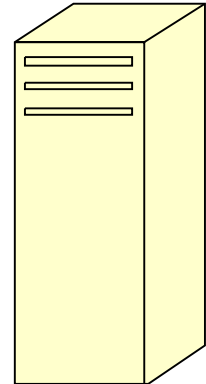
Ein Programm auf einem Rechner möchte mit einem anderen Programm auf einem anderen Rechner kommunizieren!

Beispiel 1:

1. Eingabe einer „Webadresse“



3. Server schickt Antwort mit Inhalt zurück

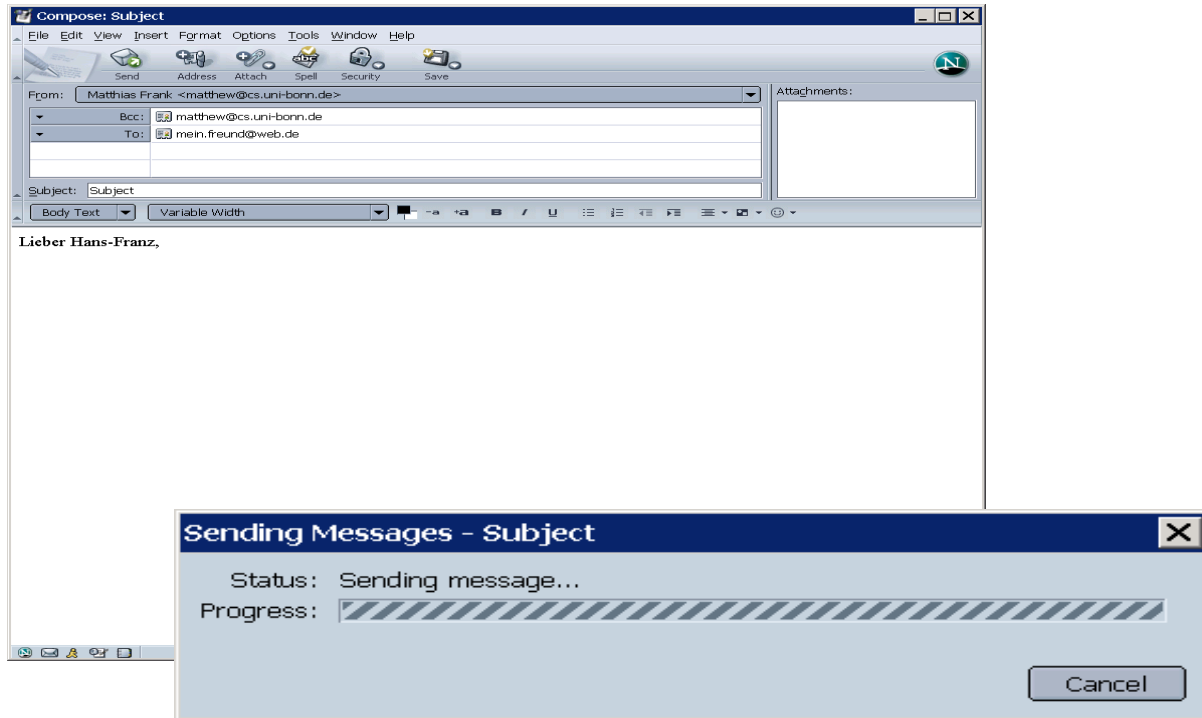


2. Anfrage wird an Server geschickt

Motivation

Beispiel 2:

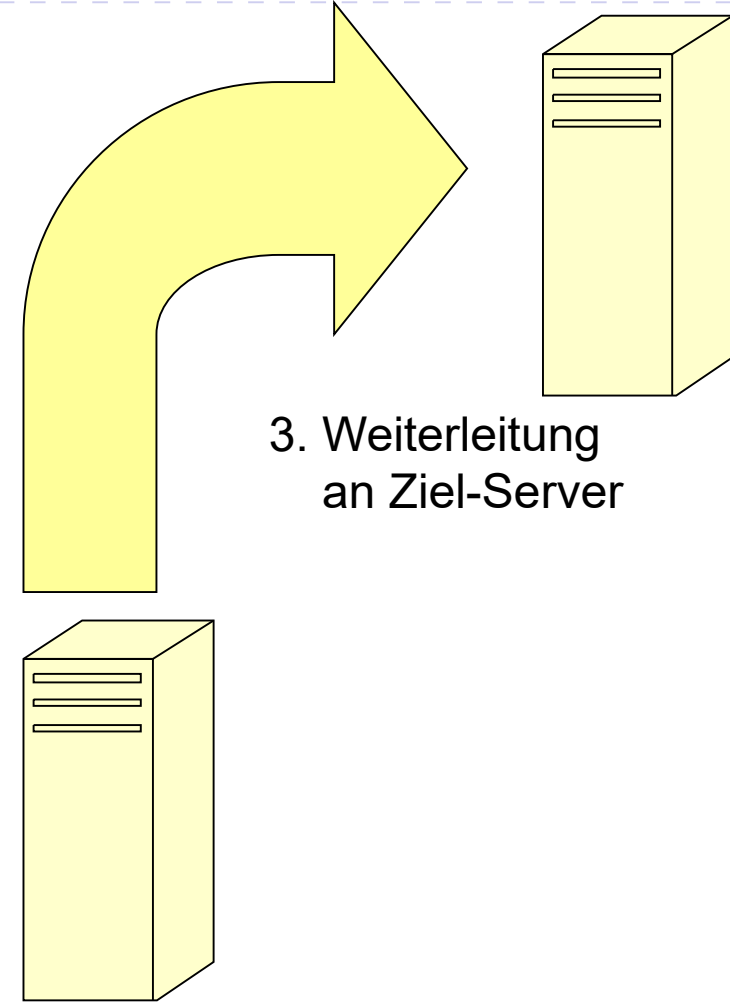
1. Schreiben und Abschicken einer E-Mail



2. Weiterleiten an lokalen Mail-Server



3. Weiterleitung an Ziel-Server



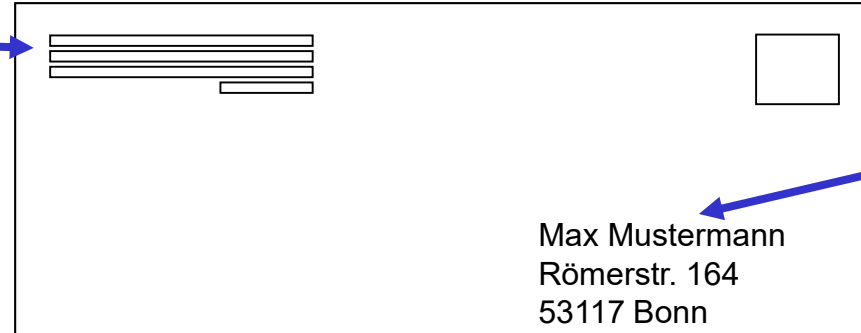
Wie funktioniert das Internet?

(auf 21 Folien)

Analogie: Briefversand

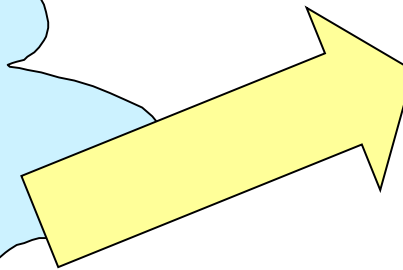
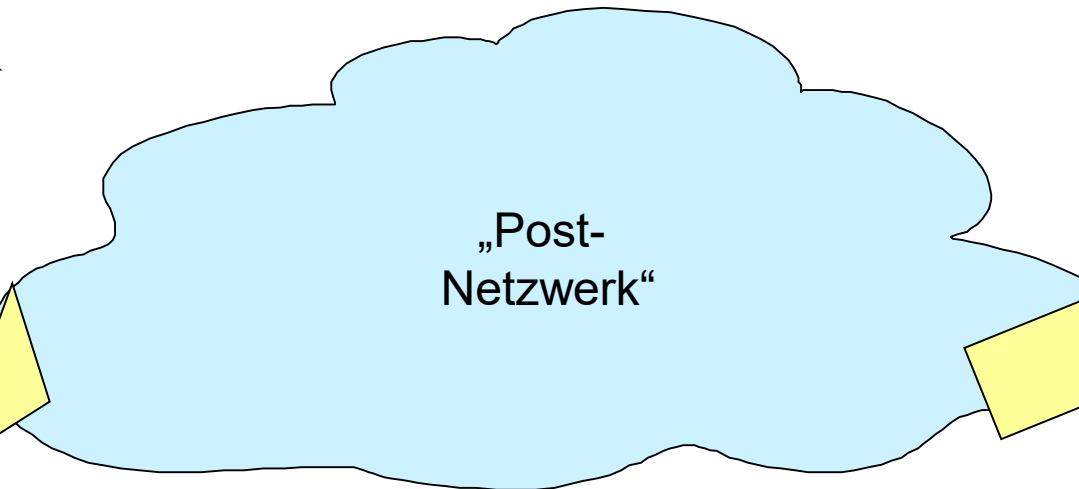
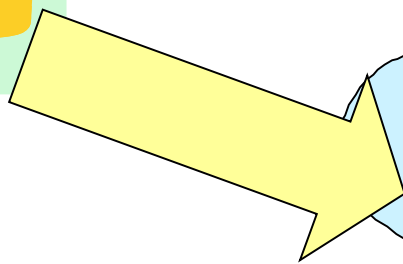
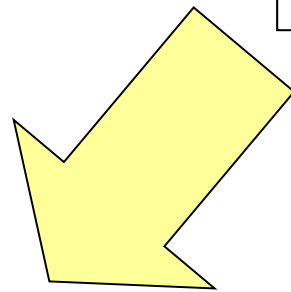
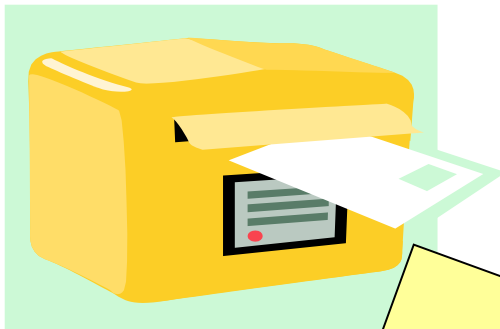
Absender:

- Name
 - Straße + Hausnr.
 - PLZ + Ort
- (mehrere Komponenten)



Zielanschrift:

- Name
 - Straße + Hausnr.
 - PLZ + Ort
- (mehrere Komponenten)



Abstrakte Wolke (für uns keine Details)

Wie funktioniert das Internet?

Analogie zum Briefversand:

- Ein Anwendungsprogramm möchte „Informationen“ (Daten) zu einem anderen Anwendungsprogramm schicken!
- Die Daten werden in „**Informationseinheiten**“ eingepackt und abgeschickt (wie ein Brief bzw. Paket)
- ein solches Paket benötigt mindestens eine **Zielanschrift**
(und auch eine **Absenderanschrift**, wenn eine Antwort erwartet wird)
- passen **nicht** alle Daten **in ein einziges Paket**, so muss die Anwendung **mehrere Pakete** schicken!
(z.B. Download einer großen Datei)

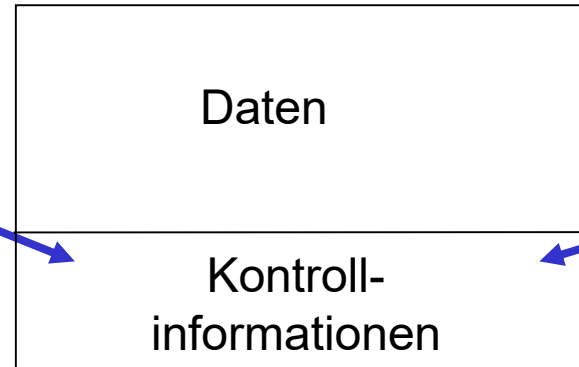
Wie funktioniert das Internet?

Konkrete Adressen im Internet:

Absender:

- IP-Adresse
 - Protokoll
 - Portnummer
- (mehrere Komponenten)

„IP-Datagramm“ (IP = Internet Protocol)

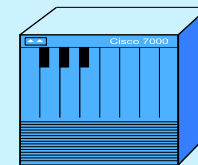
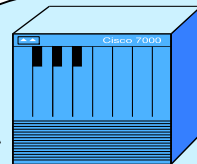
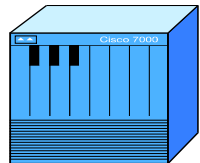


„Zielanschrift“:

- IP-Adresse
 - Protokoll
 - Portnummer
- (mehrere Komponenten)



Router



„Internet“

Abstrakte Wolke (für uns keine Details)

Wie funktioniert das Internet?

Aber: Nutzt der **menschliche Benutzer** „IP-Adresse“, „Portnummer“, ...?

Meist: Nein!

Beispiel 1: WWW



Absender:

- der eigene Rechner (kennt seine genauen Adressen)

Ziel: **www.uni-bonn.de**

Wird (automatisch) übersetzt in

- konkrete IP-Adresse
- benutzt Protokoll „TCP“
- Portnummer „80“ (reserviert für WWW-Server)

Beispiel 2: E-Mail

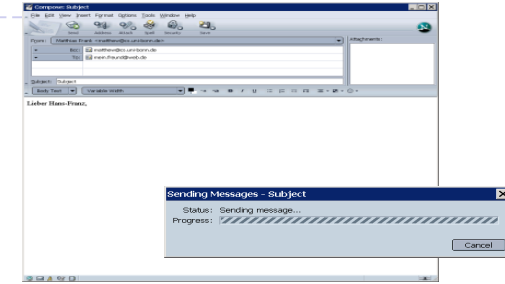
Absender:

- E-Mail-Adresse „mm@cs.uni-bonn.de“
- der eigene Rechner (kennt seine genauen Adressen)
- zusätzlich vermitteln noch dazwischen liegende Mailserver

Ziel: **mein.freund@web.de**

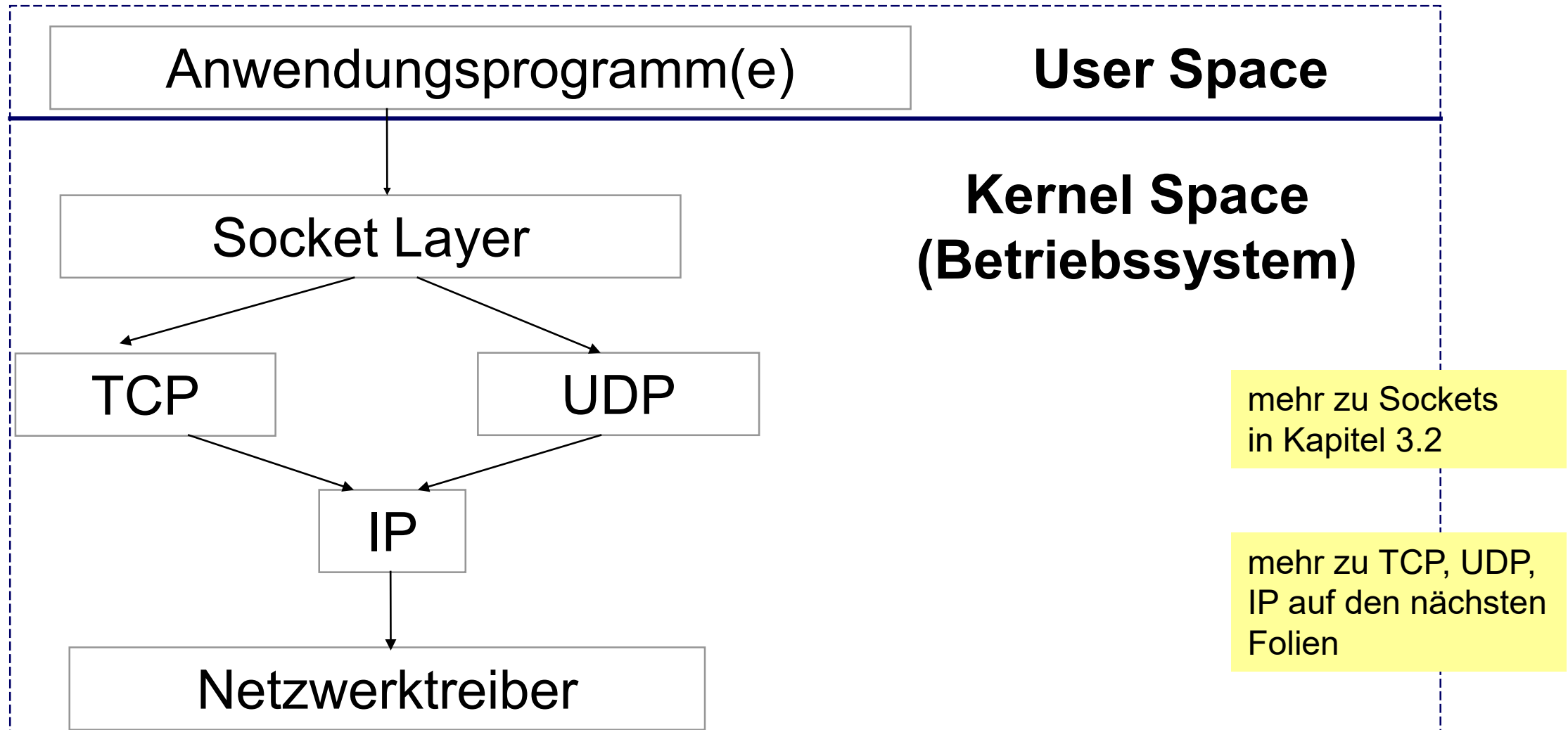
web.de wird (automatisch) übersetzt in die Adressen des zuständigen Mailservers:

- konkrete IP-Adresse
- benutzt Protokoll „TCP“
- Portnummer „25“ (reserviert für Mail-Server)



(Einen Teil dieser) Übersetzung liefert das sog. „Domain Name System“

Die Schnittstelle für Anwendungsprogramme



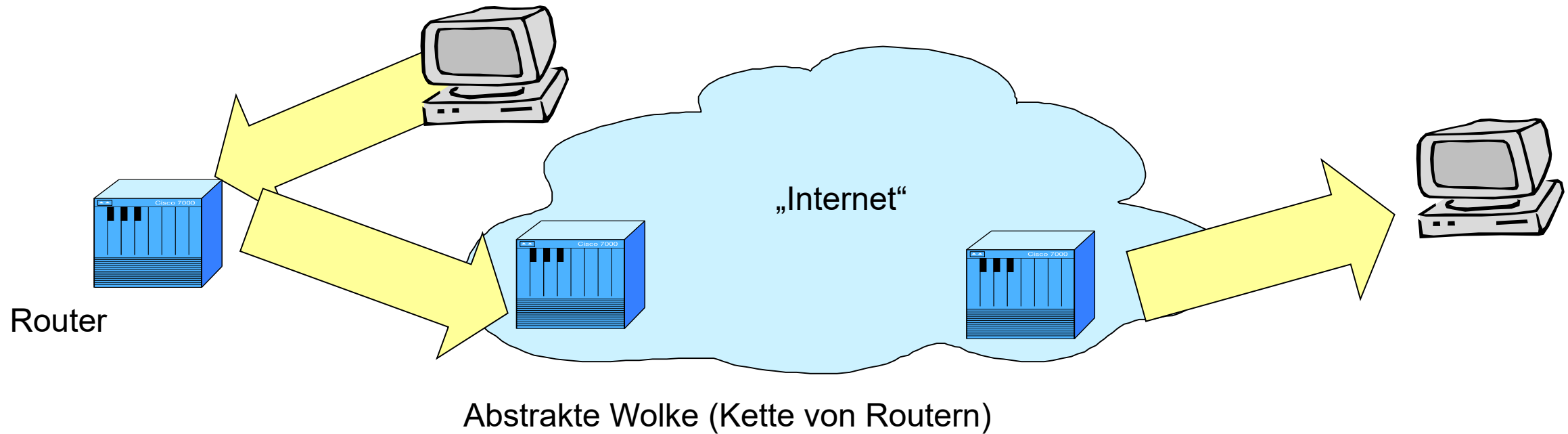
Das Internet Protocol (IP)

- Die **Basis des Internet**: IP (z.Zt Version 4, **IPv4**)
- IP ist ein **Netzwerkprotokoll** (Kommunikation Rechner-Router und Router-Router im Netzwerk)
- sendet **Datenpakete begrenzter Länge**
(theoretisch max. 64 KByte, praktisch begrenzt durch die Netzwerke)
- diese Datenpakete werden „**IP-Datagramme**“ genannt
- das **Ziel** des Datagrammes wird **durch eine IP-Adresse** angegeben
Beispiel: 131.220.6.15 (32 Bit, in Dotted-Decimal Schreibweise)
- **IP bietet keinerlei Garantien** bzgl. des Dienstes:
 - Datagramme können **verloren** gehen
 - Datagramme können **verdoppelt** werden
 - Datagramme können in **falscher Reihenfolge** abgeliefert werden
 - Datagramme können **verfälscht** werden (Bitfehler)

→ vgl. **Analogie zum Brief-/Paketversand !**

Das Internet Protocol (IP)

- IP ist verantwortlich für das „**Routing**“ (Wegfindung zum Ziel)
- IP arbeitet in jedem Rechner (Endgerät) und in allen Routern im Internet

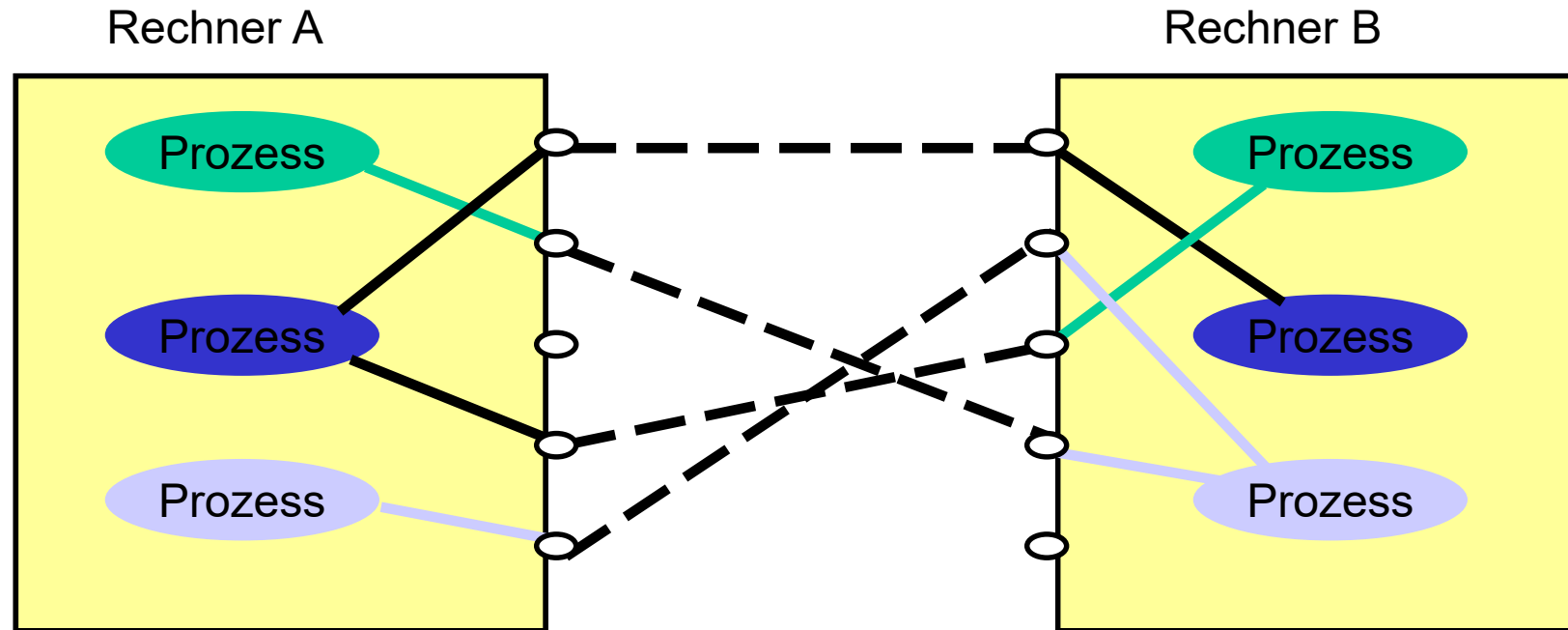


- Anwendungsprogramme greifen (normalerweise) nicht direkt auf IP zu
- Alle anderen Internetprotokolle bauen auf IP auf

Das User Datagram Protocol (UDP)

UDP ist ein **Transportprotokoll** (Kommunikation zwischen Prozessen von Anwendungsprogrammen)

- UDP arbeitet **nur in den kommunizierenden Endgeräten**
- UDP **nutzt den Dienst von IP** (merke: **keine Garantien**) um Datagramme an den richtigen Zielrechner zu schicken
- UDP **nutzt sog. „Ports“** um Kommunikation **für mehrere Prozesse gleichzeitig** anbieten zu können

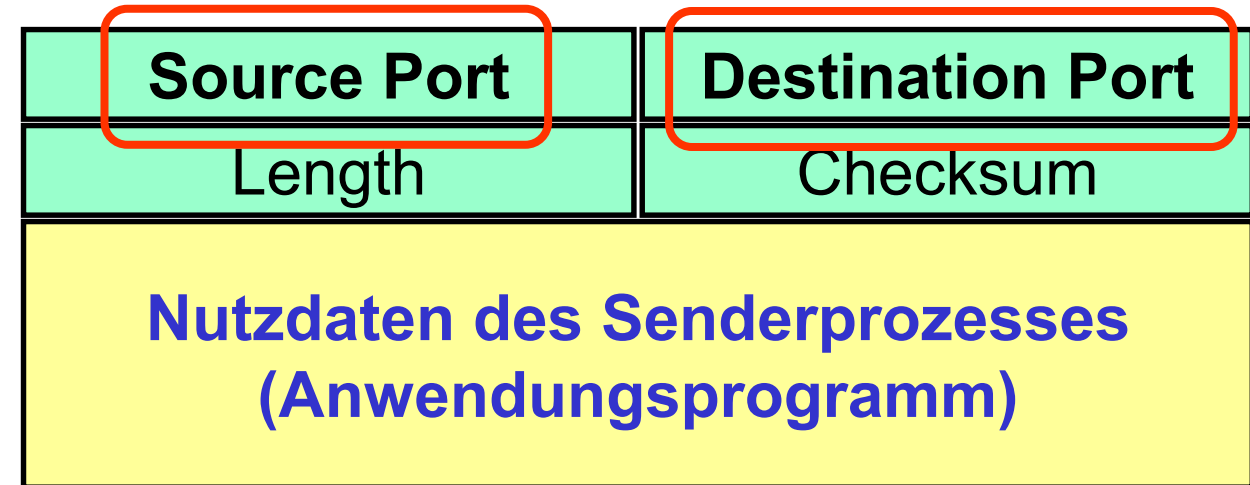


Das User Datagram Protocol (UDP)

- Auslieferung von Datagrammen
- **verbindungslos** (im Gegensatz zu TCP, mehr gleich)
- **unzuverlässig**, keine Garantien (genau wie IP)
- „**minimales**“ Transportprotokoll

Die Checksumme ist der einzige (optionale!) Schutz des Datagramms vor Fehlern (Bitfehler bei der Übertragung mit IP und den Netzwerken)

Format des UDP Datagramms:



Wichtig:

Die **Ports von Quelle und Ziel** sind die relevanten Adressen für das Transportprotokoll UDP!

Das Transmission Control Protocol (TCP)

- TCP ist ebenfalls ein **Transportprotokoll**
(Kommunikation zwischen Prozessen von Anwendungsprogrammen)
- TCP arbeitet (wie UDP) **nur in den kommunizierenden Endgeräten**
- TCP **nutzt den Dienst von IP** (merke: **keine Garantien**) um **Segmente** an den richtigen Zielrechner zu schicken
- TCP **nutzt (wie UDP) sog. „Ports“** um Kommunikation für mehrere Prozesse gleichzeitig anbieten zu können

Im **Gegensatz zu UDP hat TCP** folgende wichtige Eigenschaften:

- der Dienst von TCP ist „**zuverlässig**“
- TCP arbeitet **verbindungsorientiert**
(Grund: einige Kontrollfunktionen wg. der zu realisierenden Zuverlässigkeit)
- Datenübertragung in derselben Verbindung ist **full-duplex** möglich
(in beiden Richtungen, z.B. Request und daraufhin Response)
- TCP arbeitet **Byte-orientiert**:
 - Nutzdaten werden als Byte-Strom interpretiert,
 - TCP entscheidet selbst wie der Strom segmentiert und als IP-Datagramme versendet wird

Das Transmission Control Protocol (TCP)

TCP bietet **Zuverlässigkeit**, nutzt aber **unzuverlässigen Dienst von IP**:

IP:

- Datagramme können verloren gehen
- Datagramme können verdoppelt werden
- Datagramme können in falscher Reihenfolge abgeliefert werden
- Datagramme können verfälscht werden (Bitfehler)

TCP:

- Wiederholung, wenn nötig
- Verwerfen, wenn nötig
- Umsortieren, wenn nötig
- Checksumme prüft, ggf. Wiederholung

→ **einige Kontrollfunktionen** neben dem reinen Datentransport werden nötig

→ es muss eine **feste „Beziehung“** zwischen einem TCP-Sender und Empfänger bestehen

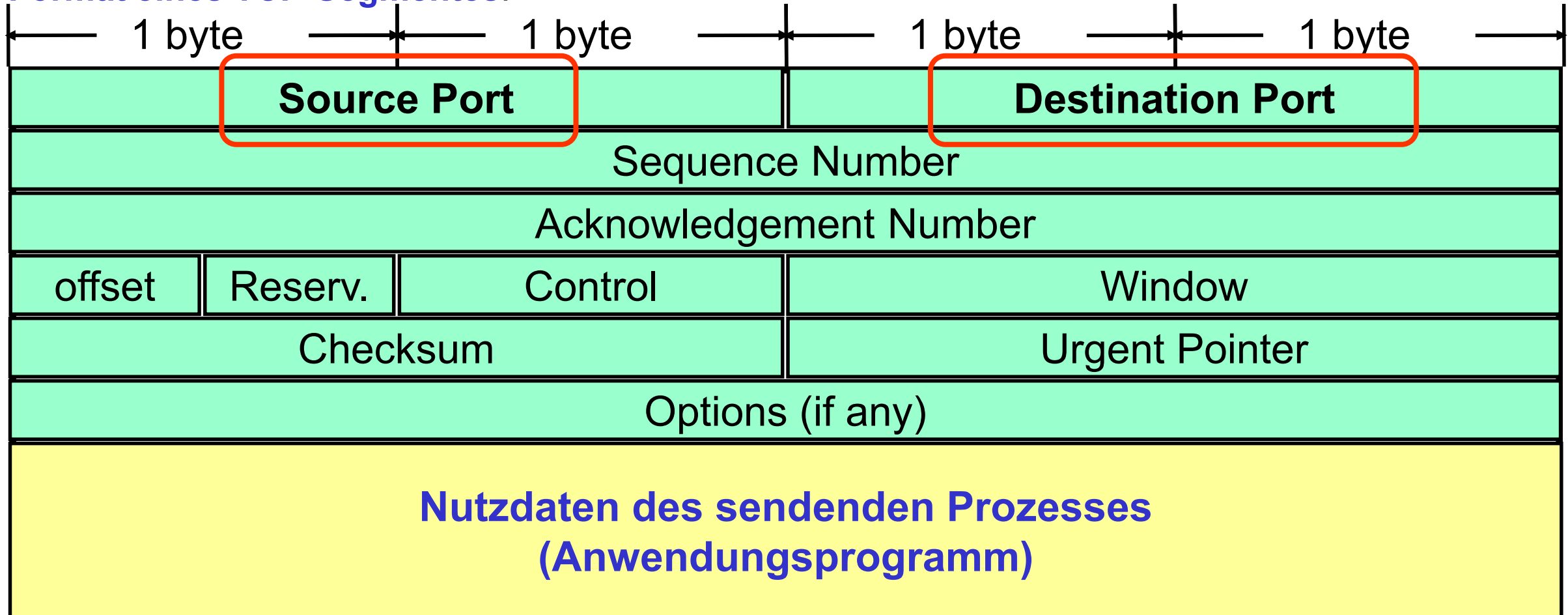
→ vor dem Datentransport **wird eine Verbindung aufgebaut!**

Ein **Anwendungsprogramm initiiert** einen TCP-Verbindungsaufbau.

Wenn die Gegenseite die Verbindung akzeptiert, können Daten **in beiden Richtungen** (full-duplex) versendet werden.

Das Transmission Control Protocol (TCP)

Format eines TCP-Segmentes:



Wichtig:

Die **Ports von Quelle und Ziel** sind die relevanten Adressen auch für das Transportprotokoll TCP!
 Alle weiteren Felder (Kontrollfunktionen) sind für uns nicht relevant.

Einige „well-known ports“ (statische Adressen) bei TCP

Decimal	Description
0	Reserved
1	TCP Multiplexer
5	Remote Job Entry
7	Echo
9	Discard
11	Active Users
13	Daytime
15	Network status program
17	Quote of the Day
19	Character Generator
20	File Transfer Protocol (data)
21	File Transfer Protocol
22	ssh (secure shell)
23	Telnet
25	SMTP (Mail Transfer)
37	Time
42	Host Name Server
43	Who is

Decimal	Description
53	Domain Name Server
77	Any private RJE service
79	Finger
80	World Wide Web HTTP
93	Device Control Protocol
95	SUPDUP Protocol
101	NIC Host Name Server
102	ISO-TSAP
103	X.400 Mail Service
104	X.400 Mail Sending
111	Sun Microsystems RPC
113	Authentication Service
117	UUCP Path Service
119	USENET News Transfer Protocol
129	Password Generator Protocol
139	NETBIOS Session Service
160-223	Reserved



Einige „well-known ports“ (statische Adressen) bei UDP

Decimal	Description
0	Reserved
7	Echo
9	Discard
11	Active Users
13	Daytime
15	Who is up or NETSTAT
17	Quote of the Day
19	Character Generator
22	ssh (secure shell)
37	Time
42	Host Name Server
43	Who is
53	Domain Name Server

Decimal	Description
67	Bootstrap Protocol Server
68	Bootstrap Protocol Client
69	Trivial File Transfer
80	World Wide Web HTTP
111	Sun Microsystems RPC
123	Network Time Protocol
161	SNMP net monitor
162	SNMP traps
512	UNIX comsat
513	UNIX rwho daemon
514	system log
525	Time daemon

Merke: Manche „well-known“ Portnummern sowohl für TCP als auch UDP (Bsp.: DNS, WWW).

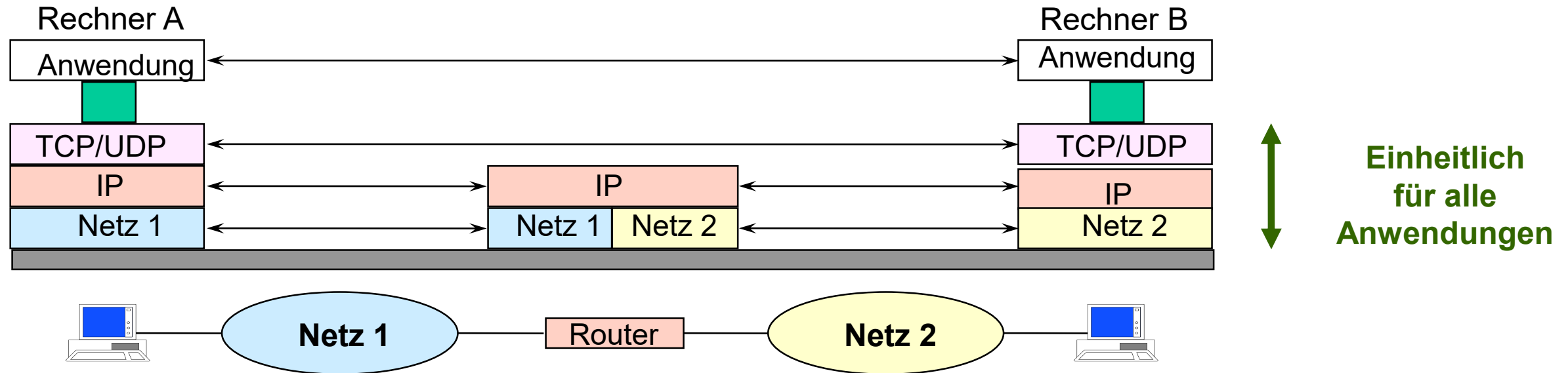
„Well-known“ Ports (< 1024) werden typisch für Server-Prozesse verwendet (diese sollten bekannt sein).

Client-Prozesse können „beliebige“ Portnummern verwenden (bis max. 65535), die z.B. vom Betriebssystem dynamisch vergeben werden.



TCP und UDP über IP

- **Anwendung auf Rechner A** möchte mit **Anwendung auf Rechner B** kommunizieren dazu greifen beide Anwendungen auf TCP oder UDP zu
- **TCP bzw. UDP sind Transportprotokolle**, sie arbeiten nur in den kommunizierenden Endgeräten
- **IP ist ein Netzwerkprotokoll**, arbeitet in den Endgeräten und in jedem Router
(Kommunikation Endgerät-Router sowie Router-Router)



Beliebig viele Router
können auf dem Weg
zwischen A und B stehen!

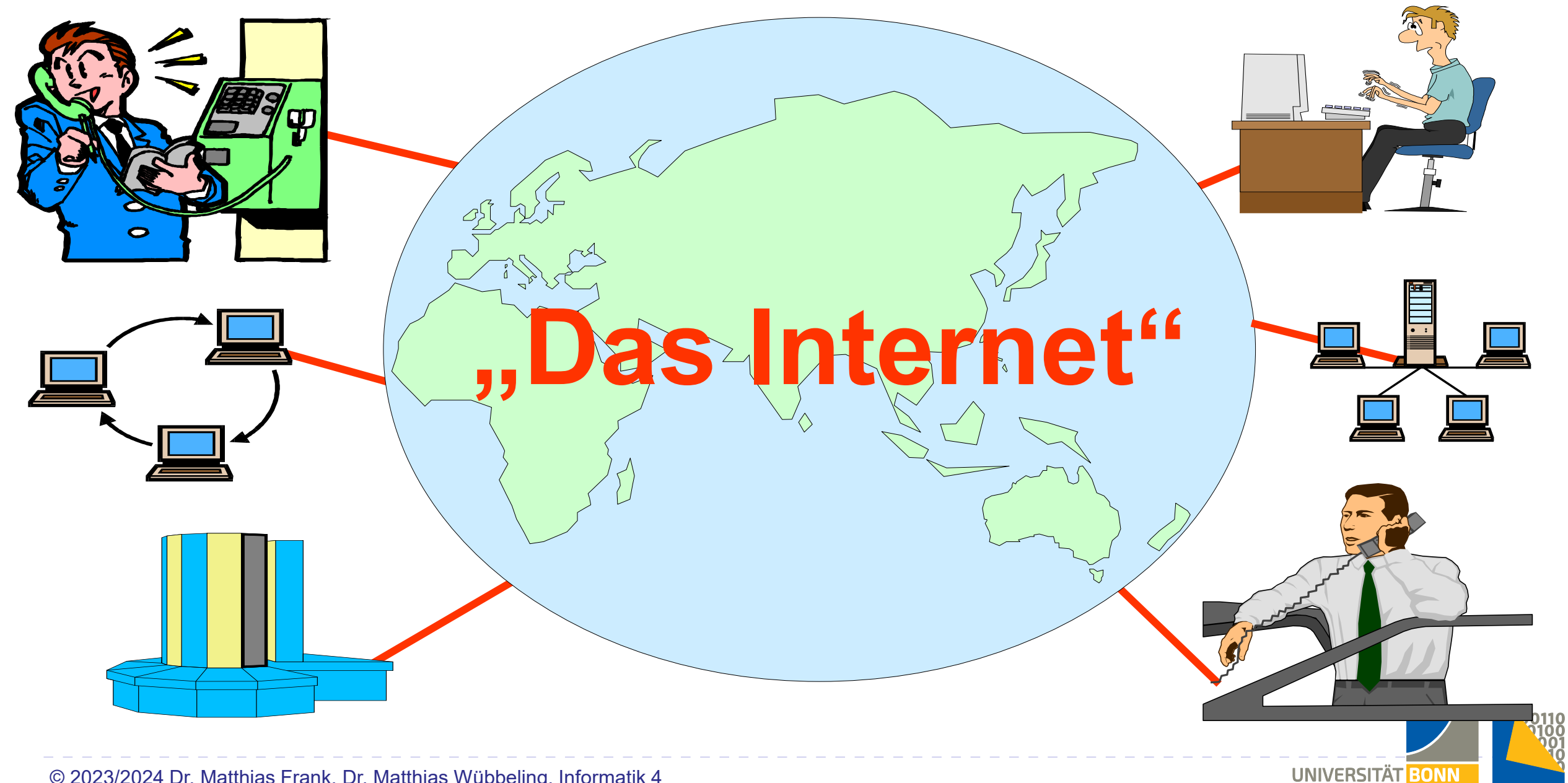
Beispiele:

- A und B im selben Netz → 0 Router
- A und B im Uni-Netz → wenige Router
- B am anderen Ende der Welt von A
→ „einige“ Router und „einige“ Netze

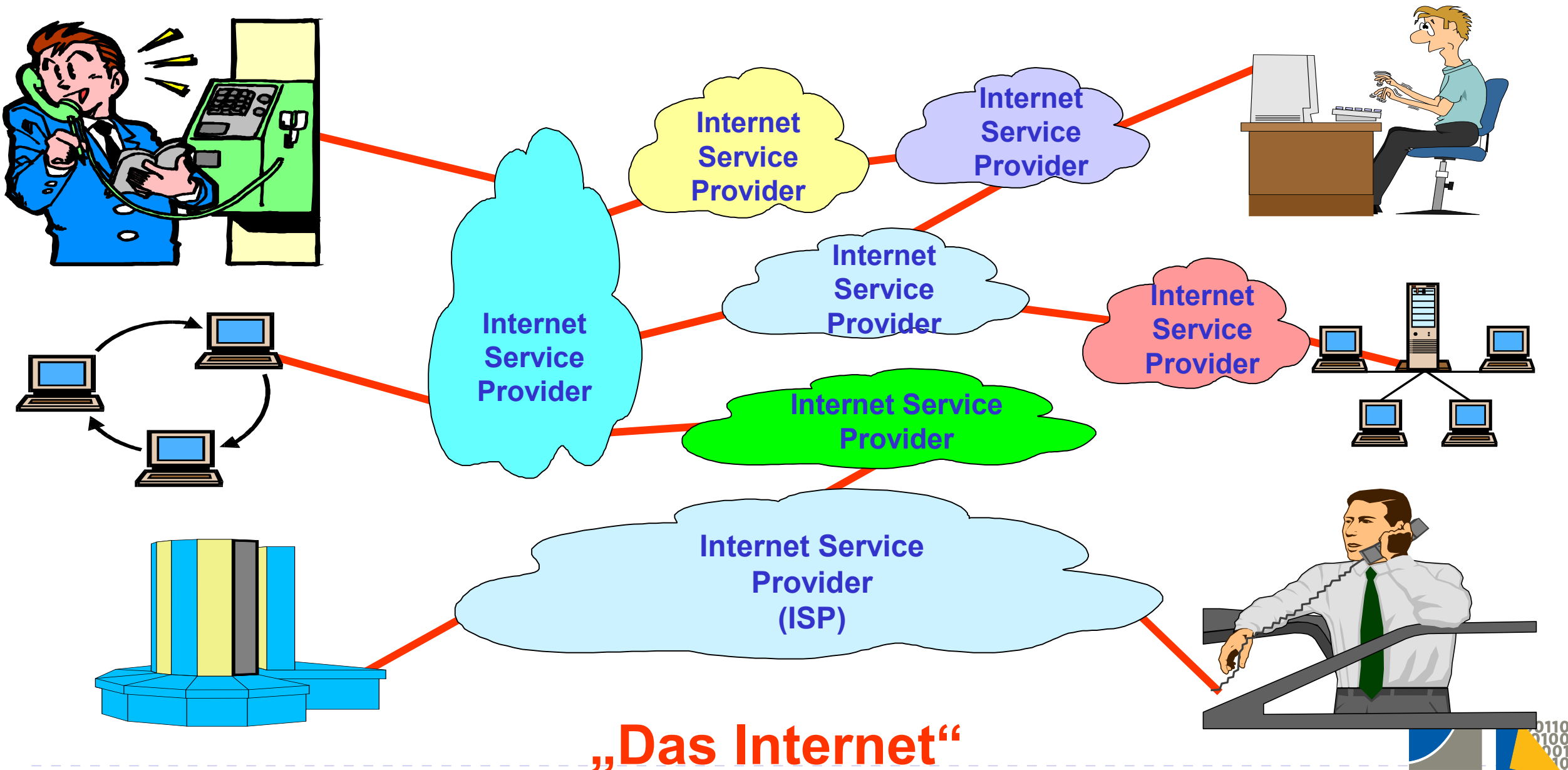
A und B wissen keine Details über den Weg zwischen ihnen!



Das weltweite Internet aus globaler Sicht



Das Internet - ein Netz von Netzen



Das Format eines IP-Datagrammes

Länge in Byte

1	Version	IHL
1	type of service	
2	total length	
2	identification	
1	0	DF MF offset
1	offset (continued)	
1	time to live	
1	protocol	
2	header checksum	
4	source address	
4	destination address	
variabel	options	
variabel	padding	
	Daten des Protokolls der höheren Schicht	

Wichtig:

- Quelle und Ziel eines IP-Datagramms sind jeweils durch eine IP-Adresse (32 Bit) angegeben!
- Das Protokoll der höheren Schicht (z.B. TCP oder UDP) wird durch das Feld „protocol“ identifiziert!
- (Alle weiteren Felder sind für uns nicht relevant,
- Kontrollfunktionen von IP)

Format von Datagrammen
(gemäß IPv4)

Zusammenfassung Adressierung im Internet

Briefversand

Absender/Zielanschrift:

- Name
- Straße + Hausnr.
- PLZ + Ort

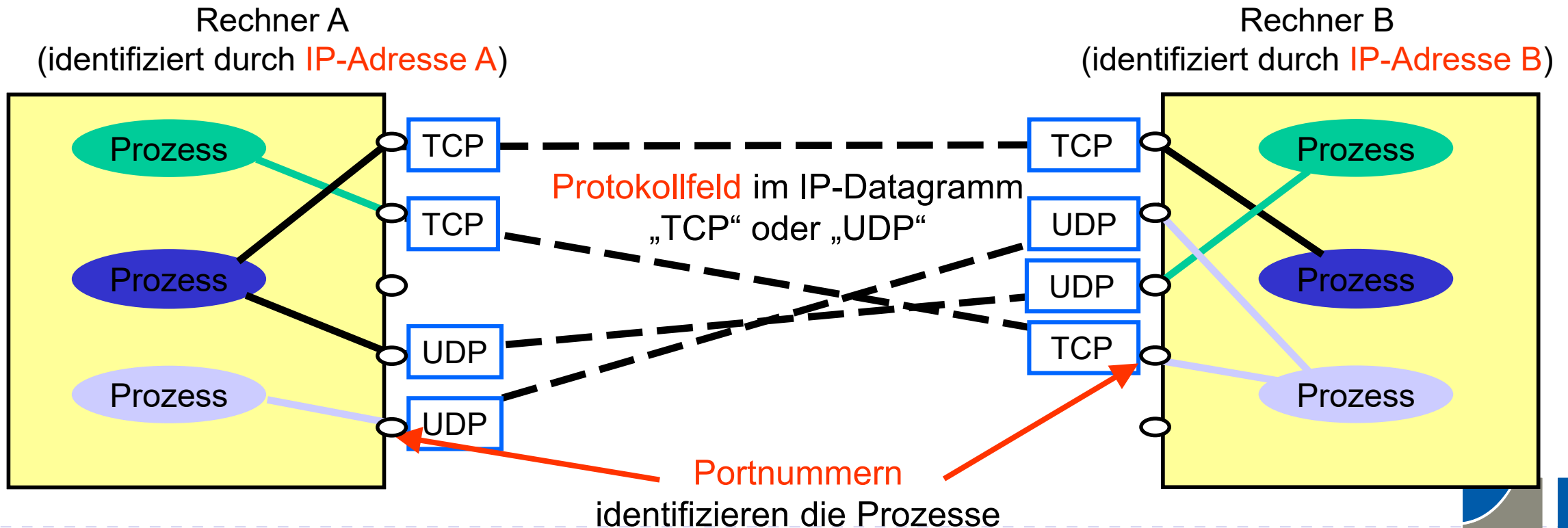
(mehrere Komponenten)

Internet

Quelle/Ziel:

- IP-Adresse
- Protokoll
- Portnummer

(mehrere = 3 Komponenten)



Zusammenfassung TCP/UDP - IP

- **Netzwerkprotokoll IP:**
Unzuverlässige Auslieferung von Datagrammen zwischen Rechnern
- **Transportprotokoll UDP:**
Unzuverlässige Auslieferung von Datagrammen zwischen Prozessen
- **Transportprotokoll TCP:**
Zuverlässige Übertragung von Byte-Strömen zwischen zwei Prozessen

Frage: Welches Transportprotokoll ist besser?

Antwort: Dies hängt von der Anwendung ab!

Also: Der Designer der Anwendung muss es entscheiden, der Programmierer muss es umsetzen!

TCP oder UDP?

TCP bietet **verbindungs-orientierten, zuverlässigen Dienst** zur Übertragung von Byte-Strömen (dazu **erheblicher Overhead**: im Header, in Übertragungszeit, ...)

UDP **minimalen Dienst** zur Auslieferung von Datagrammen (so **wenig Overhead** wie möglich)

TCP oder UDP (das könnte für Designer/Programmierer die Frage werden):

- Internet E-Commerce, Transaktionen, ...?
- Video/Audio streaming server/client ?
- File transfer ?
- E-Mail ?
- Chat groups ?
- Operation durch einen Roboter, ferngesteuert über ein Netzwerk ?
- ...
- Anwendungen im lokalen Netz oder über das (weite) Internet?

Werbeeinblendung:

Wer sich mehr für Internet und Kommunikationssysteme interessiert, der ist herzlich willkommen bei weiteren Lehrveranstaltungen der Informatik 4 (Vorlesungen „KiVS“ u/o „ReSi“ / „IT-Si“, Projektgruppe, BA-Arbeit ... Master Studium).

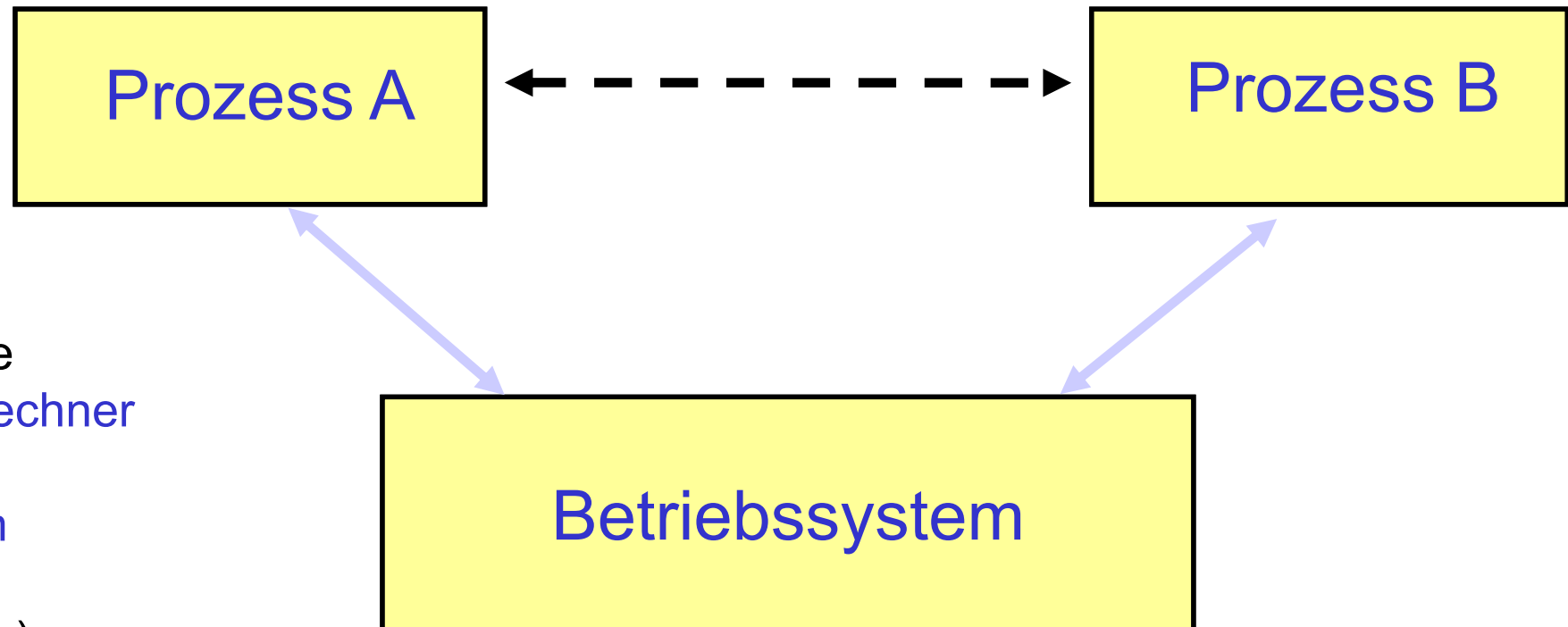


Ein Blick über den Tellerrand: Interprozess- Kommunikation

Für eine **Inter-Process-Communication (IPC)** gibt es diverse Ansätze:

Vgl. Kap 2.3 IPC
SysProg

- Shared Memory, Semaphore, Monitore
- Unix Signale
- Message Passing, Netzwerkkommunikation
- Remote Procedure Call (RPC), Remote Method Invocation (RMI)



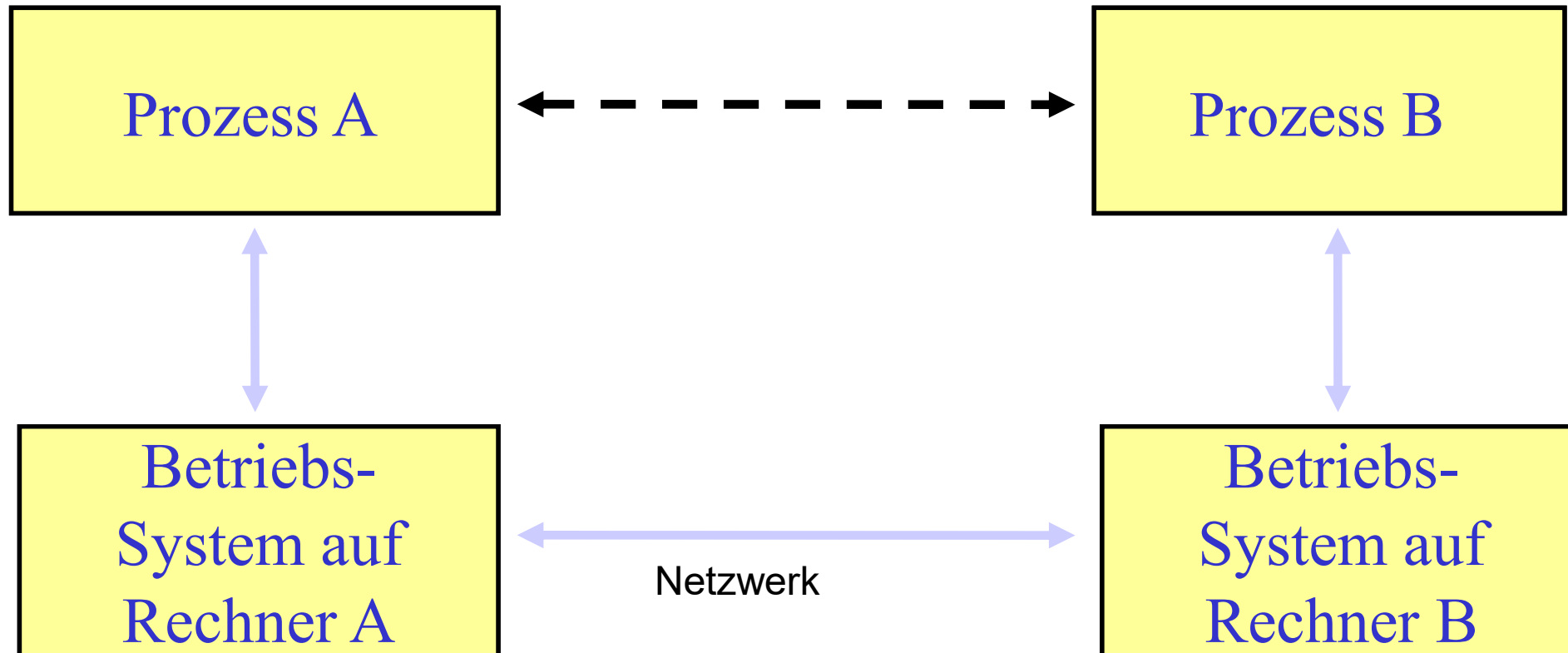
Zu unterscheiden:

1. Parallele Prozesse
auf dem selben Rechner
2. Prozesse in einem
verteilten System
(über ein Netzwerk)

Interprozess-Kommunikation (2)

Die **Netzwerkkommunikation** stellt einen universellen Fall von **IPC** dar:

- IPC zwischen Prozessen auf verschiedenen Rechnern wird möglich
- IPC zwischen Proz. auf dem selben Rechner ist genauso realisierbar
Quell-IP-Adresse = Ziel-IP-Adresse, bzw. nutze sog. Loopback-Adresse 127.0.0.1



(3.2.3.3.) Sockets

Folie aus Kap 2.3 IPC
SysProg

- Will be discussed in detail in [Chapter 3](#)
 - Process **persistence**
 - Full-duplex
 - Record or stream oriented
 - Networking support
 - Hostname + port as identifier
 - Also usable on a single system
-
- Specific for single systems: Unix domain Sockets (in contrast to [internet domain sockets](#))
 - No protocol overhead → increased efficiency

Unix Domain Sockets

Folie aus Kap 2.3 IPC
SysProg

- Identical API to internet domain socket
- Different naming scheme: `sockaddr_un` instead of `sockaddr_in`

```
struct sockaddr_un {  
    sa_family_t sun_family;           /* AF_UNIX */  
    char sun_path[108];               /* pathname */  
};
```

- Name will be mapped to file system, but can't be opened

Persistence ???

Unix Domain Sockets – Example “bind”

Folie aus Kap 2.3 IPC
SysProg

```
#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>

int main(void) {
    int fd, size;
    struct sockaddr_un un;

    un.sun_family = AF_UNIX;
    strcpy(un.sun_path, "foo.socket");
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        err_sys("socket failed");
    size = offsetof(struct sockaddr_un, sun_path) + strlen(un.sun_path);
    if (bind(fd, (struct sockaddr *)&un, size) < 0)
        err_sys("bind failed");
    printf("UNIX domain socket bound\n");
    exit(0);
}
```

“apue.h”

comes from <http://www.apuebook.com/Resources> to Stevens et al. Advanced Programming in the Unix Environment

[Source](#)

3.2. Netzwerkprogrammierung: Sockets & Co.

Das Anwendungsprogramm (bzw. der Anwendungsprogrammierer) benötigt eine Schnittstelle zum Zugriff auf Netzwerkfunktionen und die dazugehörigen Kommunikationsprotokolle.

[3.2.1. Einführung zu Sockets](#)

[3.2.2. Socket-Adressen](#)

[3.2.3. Network Byte Order](#)

[3.2.4. Kommunikation über Sockets mit UDP](#)

[3.2.5. Kommunikation über Sockets mit TCP](#)

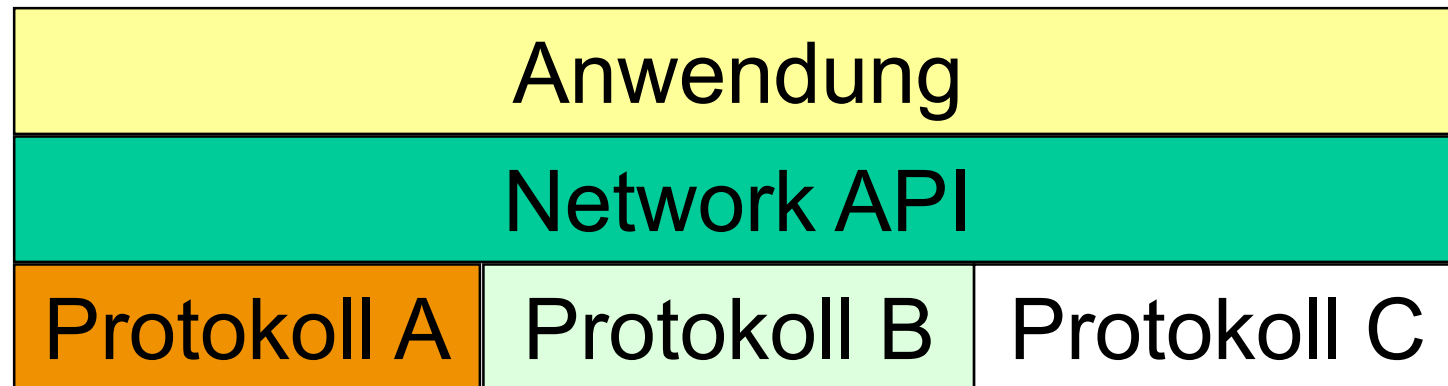
[3.2.6. Zwischenresümee, Adressübersicht, Hilfsfunktionen](#)

Netzwerkprogrammierung: Network API

Das Anwendungsprogramm (bzw. der Anwendungsprogrammierer) benötigt eine Schnittstelle zum Zugriff auf Netzwerkfunktionen und die dazugehörigen Kommunikationsprotokolle:

→ **Network Application Programming Interface**, Network API

Das Network API wird vom Betriebssystem zur Verfügung gestellt.



Es sollen möglichst mehrere Protokolle (gleichzeitig) und bei Bedarf auch mehrere Protokollfamilien unterstützt werden. (Bsp.: TCP/IP Internet vs. Appletalk)

Generisches API

- Die Network API ist ein generisches Programmierungs-Interface.
- Unterstützung von nachrichten-orientierter (vgl. UDP als Beispiel) sowie verbindungs-orientierter Kommunikation (vgl. TCP als Beispiel)
- greift auf die existierenden I/O Dienste des Betriebssystems zu
- API ist unabhängig vom Betriebssystem
(z.B. Sockets unter Linux oder Sun Solaris, ...)
- **Bedeutung von „generisch“** für Network API:
 - verschiedene Protokollfamilien können unterstützt werden
 - Unabhängigkeit der API von der konkreten Adress-Repräsentation der kommunizierenden Endpunkte
 - Ggf. spezielle Unterstützung für Clients und Server

Zugriff auf Protokolle TCP/UDP und IP

Die Protokolle der sog. „TCP/IP-Familie“ (TCP, UDP, IP, ...) haben **keine eigene**, spezielle **API Definition**.

Es gibt **verschiedene APIs** für den Zugriff auf TCP/IP:

- **Sockets**, ursprünglich „Berkeley-Sockets“ aus ihrem Ursprung vom Berkeley-Unix (=> diese werden wir genauer betrachten)
- **TLI**, System V Transport Layer Interface, entwickelt von AT&T
- **XTI**, X/Open Transport Interface, Weiterentwicklung von TLI
- **Winsock**, Socket API im Windows Betriebssystem
- **MacTCP**, Zugriff auf TCP/IP vom Apple Macintosh
- plattformunabhängig: **Asio C++ Library** (Varianten Boost.Asio und Non-Boost)
<http://think-async.com/Asio>

Benötigte Funktionen des Network API

Vorbereitung:

- Identifikation der zwei Endpunkte der Kommunikation: lokaler und entfernter Endpunkt
- Aufbau einer Verbindung zwischen den zwei Endpunkten (ggf. ohne = verbindungslos)
- Warten auf eingehende Verbindungswünsche (typisch für Server)

Datentransferphase:

- Senden und Empfangen von Daten
- Fehlerbehandlung

Nachbereitung:

- Ordentlicher Abbau einer Verbindung
- ggf. Fehlerbehandlung

Berkeley-Sockets

(Ursprung der Sockets aus dem Berkeley-Unix Berkeley-Software-Distribution 4.2BSD, 1983)

Sockets stellen ein **generisches Network API** dar.

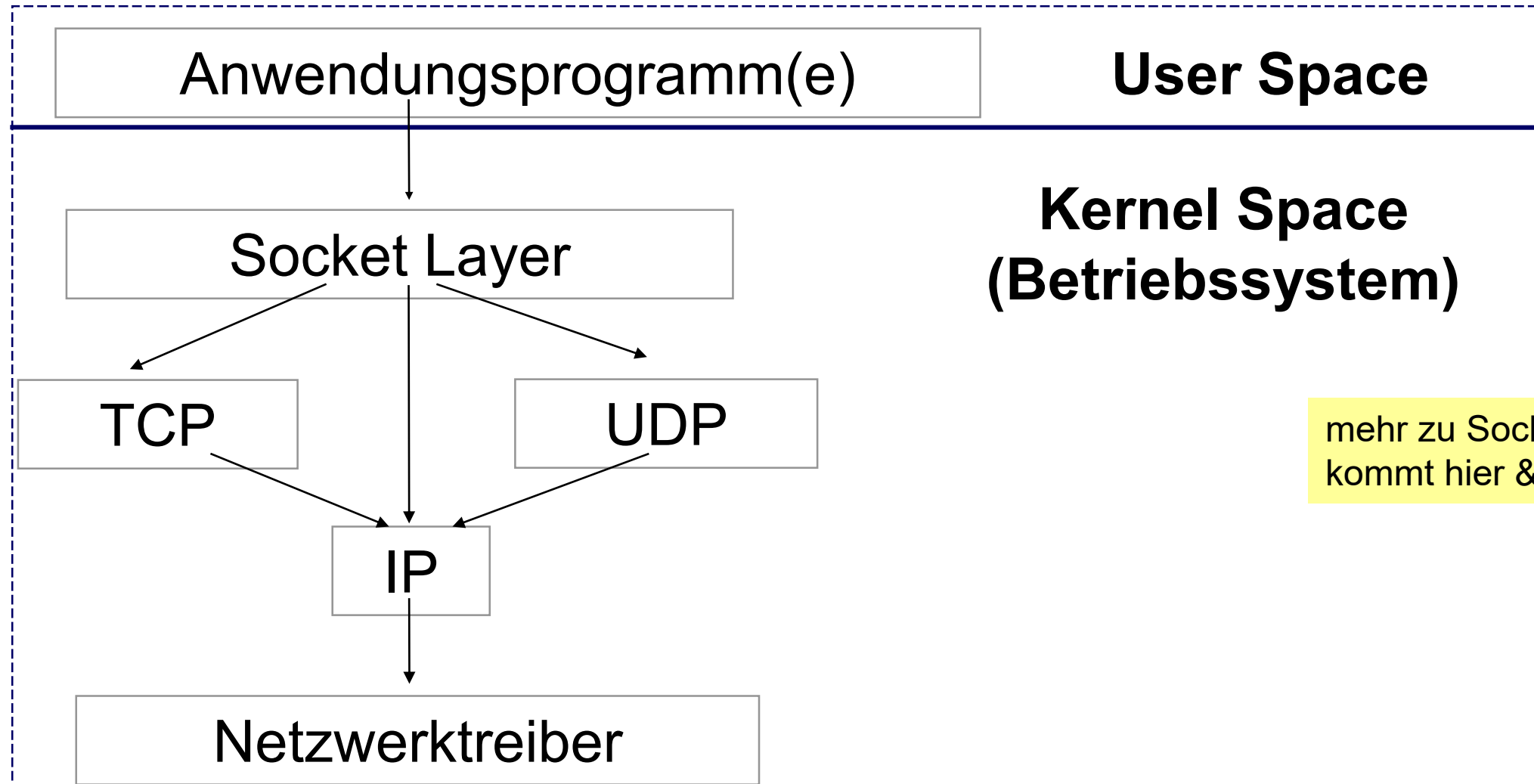
Insbesondere:

- **Unterstützung verschiedener Protokollfamilien**
 - TCP/UDP – IP Familie
 - Novell IPX
 - Appletalk
 - ...
- **Unabhängigkeit von konkreter Adressdarstellung** (Details später)

Wichtige Eigenschaften von Sockets:

- Implementiert durch eine Menge von Systemfunktionen
- Ein Socket ist eine Datenstruktur innerhalb eines Programms
- Ein Socket ist eine **abstrakte Repräsentation eines kommunizierenden Endpunktes**
- Sockets arbeiten ähnlich wie Unix I/O-Dienste, wie z.B. Files, Pipes, FIFO Queues
- Spezielle Bedürfnisse von Sockets:
 - Identifikation der Adressen der beiden Endpunkte: eigener und entfernter
 - ggf. Verbindungsaufbau

Socket Framework für TCP/UDP mit IP



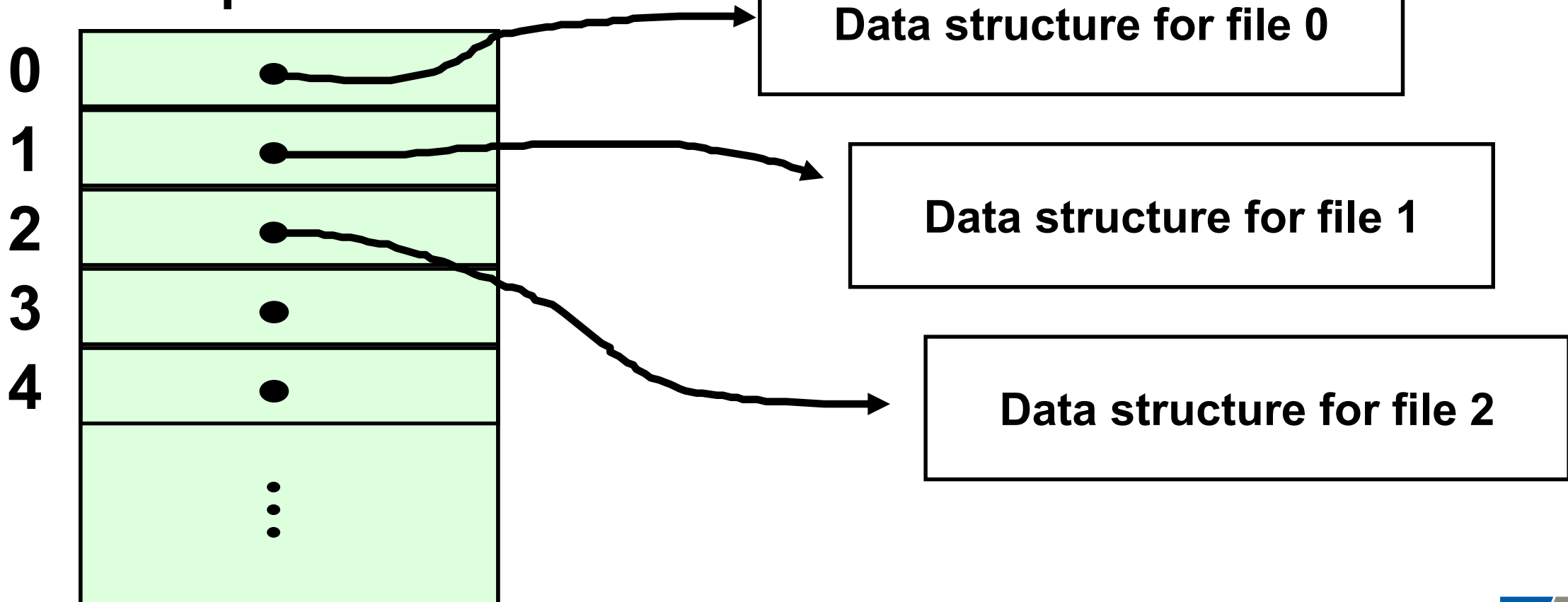
Unix Descriptor Table

Bekannte File-Deskriptoren von Unix/C

Well-known descriptors

0: STDIN
1: STDOUT
2: STDERR

Descriptor Table

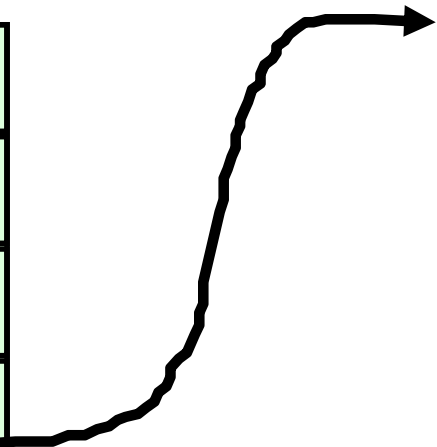


Datenstruktur eines Socket Descriptor

Ein **Socket Descriptor** ist ein „normaler“ **Unix-File-Descriptor** mit speziellem Inhalt der Datenstruktur:

Descriptor Table

0	•
1	•
2	•
3	•
4	•
	⋮



Family: AF_INET
Service: SOCK_STREAM
Local IP: 111.22.3.4
Remote IP: 123.45.6.78
Local Port: 4711
Remote Port: 3726

Socket Domain Families

Die **Protokollfamilie eines Sockets** wird bei der Initialisierung festgelegt. Für den entsprechenden Funktionsaufruf können festgelegte Schlüssel verwendet werden:

- **Internet Domain Sockets** (Schlüssel AF_INET)
 - implementiert mittels IP-Adressen und Portnummern
 - für IP Version 4
 - Schlüssel AF_INET6 für IP Version 6
- **Unix Domain Sockets** (Schlüssel AF_UNIX oder AF_LOCAL)
 - implementiert über Unix Dateinamen (sog. „named pipes“)
- **Novell IPX** (Schlüssel AF_IPX)
- **AppleTalk** Datagram Delivery Protocol (Schlüssel AF_APPLETALK)
- ... weitere

AF_ = Address Family, ursprünglich auch PF_ = Protocol Family
jedoch AF_ und PF_ werden synonym benutzt!



Service Typ eines Sockets

Es gibt drei wichtige **Typen von Sockets**.

Für den entsprechenden Funktionsaufruf können (auch hier) festgelegte Schlüssel verwendet werden:

- **Stream** (Schlüssel SOCK_STREAM)
 - benutzt das Transportprotokoll TCP
 - verbindungs-orientierter zuverlässiger Dienst
 - Byte-Strom orientiert
- **Datagram** (Schlüssel SOCK_DGRAM)
 - benutzt das Transportprotokoll UDP
 - verbindungsloser unzuverlässiger Dienst
- **Direktzugriff** auf das Netzwerk (Schlüssel SOCK_RAW)
 - greift direkt auf IP zu („übergeht“ die Transportprotokolle)
 - genutzt z.B. für Routing und andere Spezialzwecke



Protokolfamilie vs. Socket Type

Kombinationsmöglichkeiten von Protocol Family und Socket Type im Falle der Internet-Protokolle:

	AF_INET	AF_INET6	AF_LOCAL AF_UNIX	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP	TCP	Ja		
SOCK_DGRAM	UDP	UDP	Ja		
SOCK_RAW	IPv4	IPv6		Ja	Ja

AF_ROUTE und AF_KEY sind Beispiele für weitere Protokoll-/Adressfamilien, Zugriff auf den Kernel für Funktionen des Routing bzw. Verschlüsselung/Sicherheit.

Erzeugung des Sockets - Parameter

Programmsourcesequenz:

```
#include <sys/types.h>
#include <sys/socket.h>
```

```
int socket(int domain, int type, int protocol);
```

- **Parameter domain** bestimmt die Protokollfamilie
benutze Schlüssel **AF_INET**, AF_INET6, AF_LOCAL etc.
(wir verwenden im weiteren nur AF_INET)
- **Parameter type** bestimmt den Typ des Socket: TCP oder UDP
benutze Schlüssel SOCK_STREAM oder SOCK_DGRAM
- **Parameter protocol** identifiziert speziellen Wert für Protokoll-Feld des IP-Headers
 - im Normalfall 0 (TCP und UDP werden automatisch eingetragen)
 - für SOCK_RAW muss der Wert entsprechend gesetzt werden

Erzeugung des Sockets - Resultat

Programmsourcesequenz:

```
#include <sys/types.h>
#include <sys/socket.h>
```

```
int socket(int domain, int type, int protocol);
```

- **Rückgabewert** ist ein Socketdescriptor (nicht-negativer Integerwert; 0, 1, 2, ...)
- Im Falle eines **Fehlers** Rückgabewert -1
- socket() stellt die Ressourcen bereit, die für die (abstrakte) Repräsentation des kommunizierenden Endpunktes benötigt werden
- socket() kümmert sich nicht um Adressierung!
(denn der Socket ist generisch) → die konkrete Adresse des Endpunktes wird dem Socket im nächsten Schritt mitgeteilt (**verschiedene Fälle!**)



Binden einer Adresse – bind()

Falls ein Programm **einen Socket öffnet, um empfangsbereit zu sein**, so folgt als nächster Schritt der Aufruf von `bind()` – sowohl für UDP als auch TCP:

Programmsourcesequenz:

```
#include <sys/types.h>
#include <sys/socket.h>
```

```
int bind(int sockfd, const struct sockaddr *myaddr,
        socklen_t addrlen);
```

- **Parameter sockfd** (Socket File Descriptor) bestimmt den konkreten Socket, den wir an eine Adresse binden wollen
- **Parameter myaddr** ist ein Zeiger auf die Adresse (generisch!)
- **Parameter addrlen** ist die konkrete Längenangabe des Adress-Structs unter `myaddr`
- **Rückgabewert** 0 wenn alles ok, bei Fehler -1

3.2.2. Socket-Adressen

Generische Socket-Adressen:

Der **generische Typ struct sockaddr** ist wie folgt definiert:

```
typedef uint16_t    sa_family_t;

struct sockaddr {
    sa_family_t    sa_family;    /* address family */
    char          sa_data[14]; /* up to 14 bytes of
                                direct address */
};
```

- `sa_family` gibt die Adressfamilie an (`AF_INET`, `AF_LOCAL`, ...)
- `sa_family_t` entspricht `uint16_t` bzw. `unsigned short`
- `sa_data[]` kann 14 weitere beliebige Bytes beinhalten (abhängig von der konkreten Adressfamilie)

Generische Socket-Adressen

Es gibt auch folgende Repräsentation (seit 4.3BSD-Reno) **zur Unterstützung von variabler Länge der Adressstrukturen**:

```
struct sockaddr {  
    uint8_t    sin_len;    /* length of structure (16) */  
    sa_family_t sa_family; /* address family */  
    char       sa_data[14]; /* up to 14 bytes of  
                           direct address */  
};
```

Die Präsenz von `sin_len` hängt vom Betriebssystem ab.

Der Programmierer muss sich jedoch (fast) nie mit `sin_len` beschäftigen, dies wird intern in den Kernelroutinen erledigt
(Ausnahme: Funktionen zum Routing im Kernel, vgl. Schlüssel `AF_ROUTE`)

Einige Typdefinitionen – Übersicht

Bei den folgenden Codesequenzen bzw. Datenstrukturen werden folgende Typdefinitionen aus den System Header-Includefiles häufig vorkommen:

Datentyp	Beschreibung	Zugeh. Headerfile
<code>int8_t</code> <code>uint8_t</code> <code>int16_t</code> <code>uint16_t</code> <code>int32_t</code> <code>uint32_t</code>	signed 8-bit Integer unsigned 8-bit Integer signed 16-bit Integer unsigned 16-bit Integer signed 32-bit Integer unsigned 32-bit Integer	<code><sys/types.h></code> <code><sys/types.h></code> <code><sys/types.h></code> <code><sys/types.h></code> <code><sys/types.h></code> <code><sys/types.h></code>
<code>sa_family_t</code> <code>socklen_t</code>	address family of socket address structure, normally <code>uint16_t</code> length of socket address structure, normally <code>uint32_t</code>	<code><sys/socket.h></code> <code><sys/socket.h></code>
<code>in_addr_t</code> <code>in_port_t</code>	IPv4 address, normally <code>uint32_t</code> TCP or UDP port, normally <code>uint16_t</code>	<code><netinet/in.h></code> <code><netinet/in.h></code>



Konkrete Socket-Adressen, IPv4 (AF_INET)

Die **Adressen der Internet-Protokollfamilie** sind wie folgt definiert:

```
struct in_addr {
    in_addr_t s_addr;          /* 32 bit IPv4 address */
};                             /* network byte order! */

struct sockaddr_in {
    sa_family_t sin_family;    /* here: AF_INET */
    in_port_t sin_port;        /* 16-bit TCP/UDP port number */
                                /* network byte order! */
    struct in_addr sin_addr;    /* 32 bit IPv4 address */
                                /* network byte order! */
    char sin_zero[8];          /* unused, total size 16 bytes */
};
```

- Das konkrete struct `sockaddr_in` hat die gleiche Länge (16 Byte) wie das generische struct `sockaddr`
- struct `sockaddr_in` enthält „nur“ zwei der bekannten Adress-Elemente der TCP/IP-Familie (!?), nämlich Port und IP-Adresse



Konkrete Socket-Adressen

Adressen der TCP/IP-Familie:

```
struct sockaddr_in {
    sa_family_t sin_family;    /* here: AF_INET */
    in_port_t    sin_port;     /* 16-bit TCP/UDP port number */
                                /* network byte order! */
    struct in_addr sin_addr;   /* 32 bit IPv4 address */
                                /* network byte order! */
    char        sin_zero[8];   /* unused, total size 16 bytes */
};
```

Das **zu verwendende Protokoll** wurde beim Erzeugen des Sockets mit Parameter type bereits festgelegt.
(SOCK_STREAM = **TCP**, SOCK_DGRAM = **UDP**)

Somit enthält der **Socket als eindeutige Identifikation** des kommunizierenden Endpunktes die bereits kennen gelernte **IP-Adresse, Protokoll-Typ sowie Portnummer**.

Internet (vgl. Folie 25)

Quelle/Ziel:

- **IP-Adresse**
 - **Protokoll**
 - **Portnummer**
- (mehrere = 3 Komponenten)

Was bedeutet „**Network Byte Order**“ für Port bzw. Adresse?



3.2.3. Network Byte Order

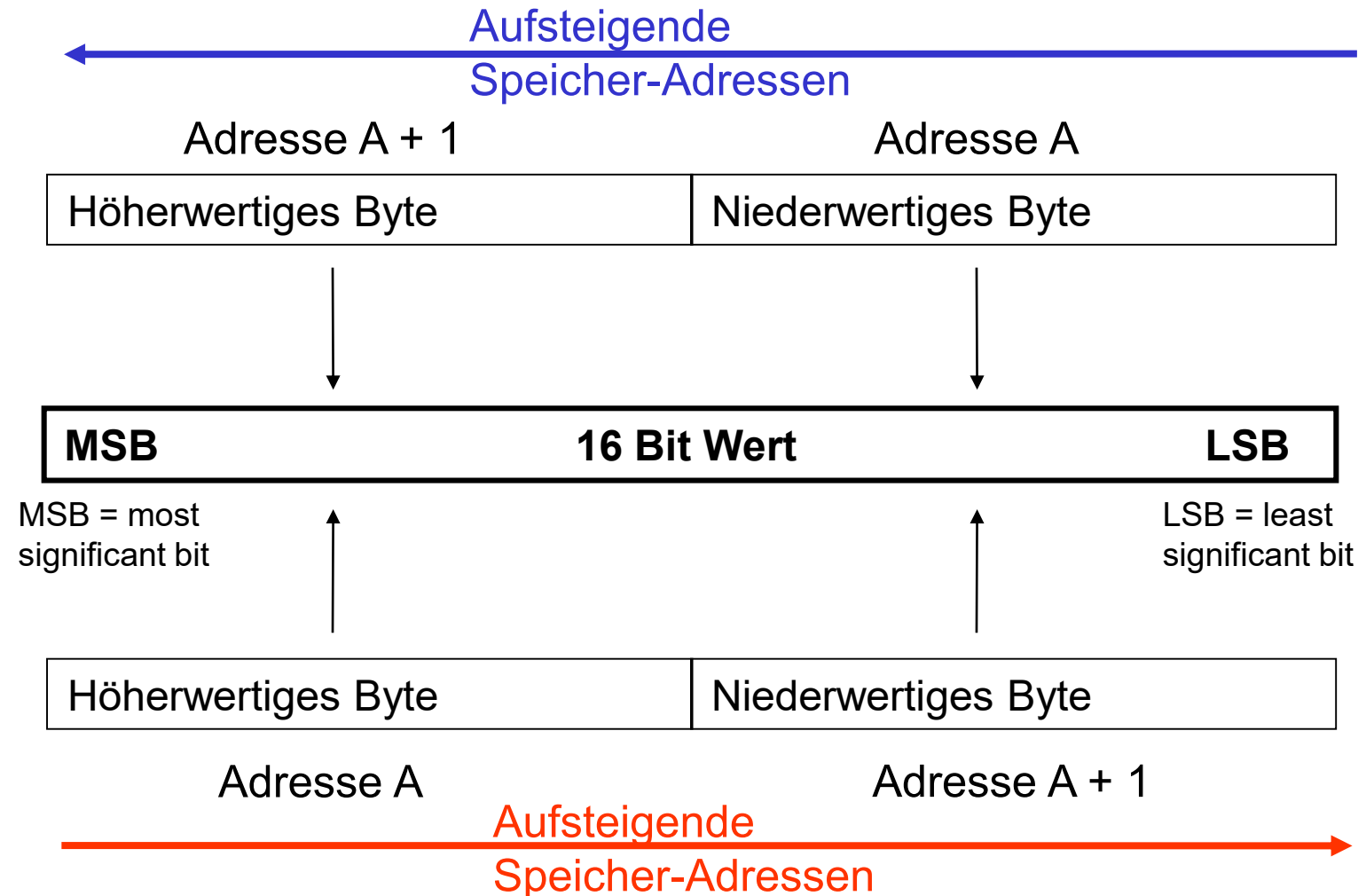
Die Werte in `sin_port` (16 Bit Integer, Portnummer) sowie `sin_addr` (32 Bit Integer, IPv4 Adresse) müssen in der sog. Network Byte Order abgelegt sein.

Hintergrund:

**Little-Endian-
Byte-Order**

16 Bit Portnummer:
(32 Bit Adresse analog)

**Big-Endian-
Byte-Order**



**Wie wird ein Zahlenwert, der mehrere Bytes umfasst, im Speicher abgelegt?
(der Speicher ist mit Byte-weiser Adressierung organisiert!)**

Unterschiedliche Speicherarchitekturen

Little-Endian- Byte-Order

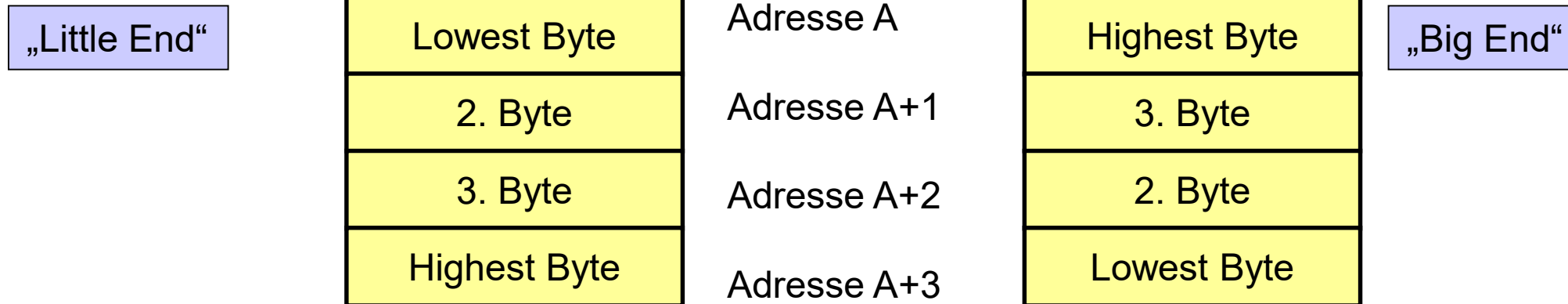
Big-Endian- Byte-Order

Vgl. Kap 1. Assembler
SysProg

Architektur:

Alpha
DEC VAX
DEC PDP-11
IBM 80x86
...

IBM 370
Motorola 68000
PowerPC
Hewlett Packard
Sun (Ultra) Sparc
(Java VM)
...



Eselsbrücke: „Little Endian“ bzw. „Big Endian“ besagt, welches Byte (little oder big) an der Startadresse (also der niedrigsten) abgelegt ist!

Host vs. Network Byte Order

Die **Host Byte Order** ist genau die architektur-abhängige Darstellung (also z.B. Little-Endian auf PCs, Big Endian auf Sun Workstations).

Für die Netzwerkfunktionen und deren Protokolle wird jedoch eine **architektur-unabhängige Darstellung** benötigt, um z.B. korrekte Kommunikation zwischen einem PC und einem Sun Server zu gewährleisten!

→ Definition und Einführung der sog. **Network Byte Order**

Für die Internet-basierten Protokolle (TCP, UDP, IP, ...) ist die **Network Byte Order** identisch zur **Big-Endian Byte Order** definiert!

→ auf einer Sun Sparc Architektur ist „alles ok“

→ auf einer **PC-Architektur müssen die Darstellungen ggf. gewandelt werden!**

Beispiel zur Wandelung

```
...  
struct sockaddr_in mysockaddr;  
...  
mysockaddr.sin_port = 4711;    /* weise festen Port zu! */  
...  
err = bind(mysock, &mysockaddr, sizeof(mysockaddr));
```

Auf **Sun Sparc** wäre **alles in Ordnung** denn Network = Host Byte Order!

Läuft **derselbe Programmcode auf einem PC**, so gibt es Probleme bei der Verarbeitung der derart zugewiesenen Portnummer 4711.

Dezimalzahl 4711 = Binärzahl 0001001001100111, High Byte = 00010010
Low Byte = 01100111

Gemäß Little-Endian steht „01100111“ in der niedrigeren Speicherstelle,
die **Network Byte Order erwartet dort aber das High Byte ...**

→ die Netzwerkprotokolle verarbeiten die **Portnummer 0110011100010010** (binär)
(dies entspricht der **Dezimalzahl 26.386**, und **nicht der gewollten 4711**)



Hilfsfunktionen zur Wandelung

Es gibt entsprechende **Funktionen zur Wandelung** zwischen Network und Host Byte Order:

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

```
uint16_t htons(uint16_t hvalue); /* Host to Network, 16 bits */  
uint32_t htonl(uint32_t hvalue); /* Host to Network, 32 bits */  
uint16_t ntohs(uint16_t hvalue); /* Network to Host, 16 bits */  
uint32_t ntohl(uint32_t hvalue); /* Network to Host, 32 bits */
```

Merkweise: h = host, n = network, s = short, l = long
(also ntohs = network to host für short Integer, usw.)

In guten Programmen sollten diese Funktionen an den entsprechenden Stellen ***IMMER*** benutzt werden, damit der Programmierer nicht mehr nachdenken muss, ob das Programm auf Big-Endian oder Little-Endian Architekturen laufen soll!

Beispiel von eben korrigiert:

```
...  
mysockaddr.sin_port = htons(4711); /* weise festen Port zu! */  
...                               /* Network Byte Order! */
```

Komplettbeispiel: socket() und bind() mit UDP

```
int fd, err;
struct sockaddr_in addr;

fd = socket(AF_INET, SOCK_DGRAM, 0);
if (fd < 0) { ... }

addr.sin_family = AF_INET;
addr.sin_port = htons(4711);
addr.sin_addr.s_addr = htonl(INADDR_ANY);

err = bind(fd, (struct sockaddr *) &addr, \
           sizeof(struct sockaddr_in));
if (err < 0) { ... }
```

Socket für UDP wird generiert

Fester Port wird zugewiesen

INADDR_ANY bedeutet, dass das Betriebssystem die IP-Adresse automatisch bestimmt (nämlich die des „eigenen“ Rechners, auf dem das Programm läuft)

Anmerkung:

addr ist vom Typ struct sockaddr_in, der 2. Parameter von bind() ist jedoch vom generischen Adresstyp struct sockaddr (vgl. Folie 48).

→ Daher wird in einem sauberen Programm (Vermidung von Compiler-Warnung) eine Typumwandlung (Casting) mittels (struct sockaddr *) vorgenommen!

[Source](#)



Besonderheiten bei der Adresswahl

Beim **Aufruf von bind()** können die benötigten **Adressen** (Port, IP-Adresse) entweder **konkret gewählt** oder **vom Betriebssystem (BS) automatisch zugewiesen** werden.

Prozess wählt		Ergebnis
IP-Adresse	Portnummer	
INADDR_ANY	0	BS wählt IP-Adresse und Port automatisch
INADDR_ANY	konkret, > 0	BS wählt IP-Adresse, Prozess wählt Port selbst
konkrete IP-Adresse	0	Prozess wählt IP-Adresse selbst, BS wählt Port autom.
konkrete IP-Adresse	konkret, > 0	Prozess wählt IP-Adresse und Port selbst

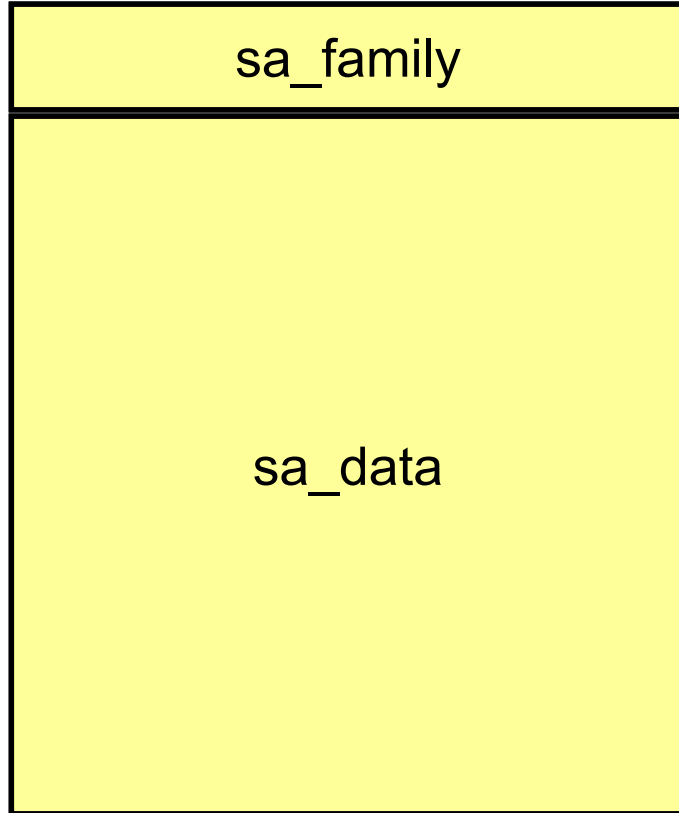
- **BS wählt Portnummer automatisch:** Ist sinnvoll für Client-Prozesse, die nicht mit einer festen (bzw. bekannten, „well-known“) Portnummer kommunizieren müssen.
BS wählt eine verfügbare Portnummer aus Bereich > 1023 aus.
- **INADDR_ANY:** BS verwendet automatisch die IP-Adresse des Rechners, auf dem der Socket geöffnet wurde (... auf dem das Programm läuft).
Hat der Rechner mehrere IP-Adressen (mehrere Netzwerkanschlüsse), so wählt das BS „geeignet“ aus (z.B. gemäß des Routing)



Übersicht weiterer Socket-Adress-Strukturen

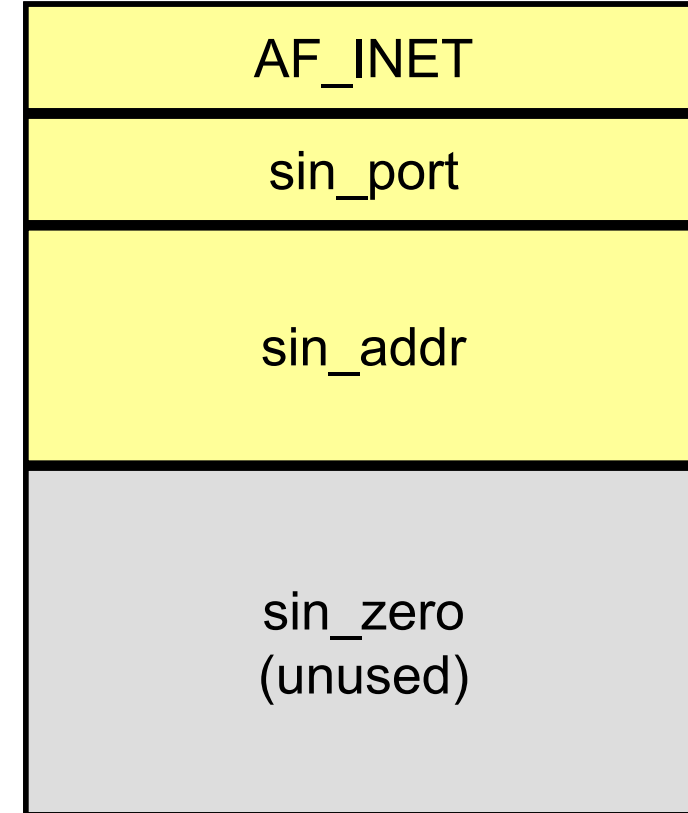
Bereits kennen gelernt:

Generische Socket-Adress-Struktur



Gesamtlänge: 16 Bytes

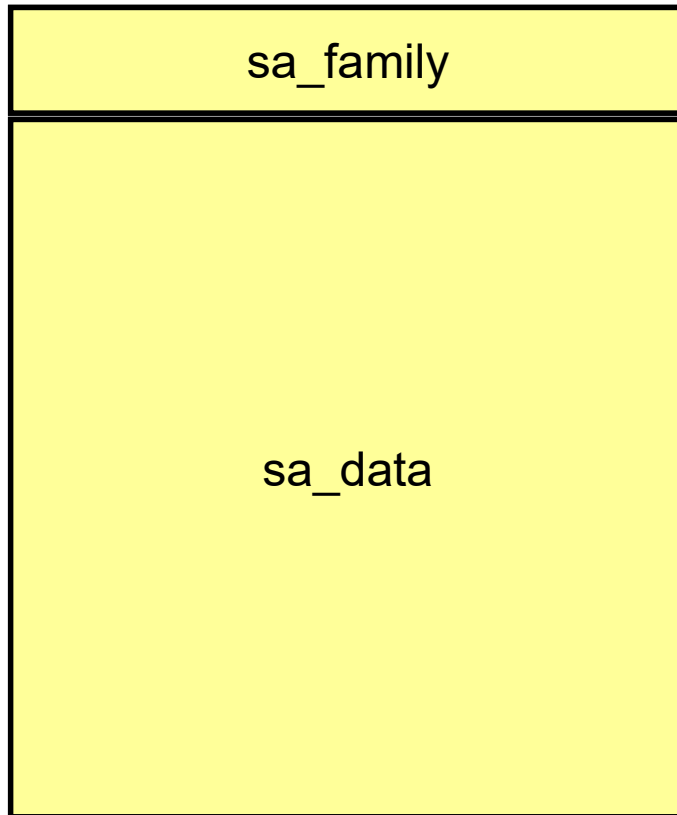
Konkrete Adress-Struktur für die IPv4-Familie



Länge der Struktur wird auf 16 Byte aufgefüllt (Mindestlänge!)

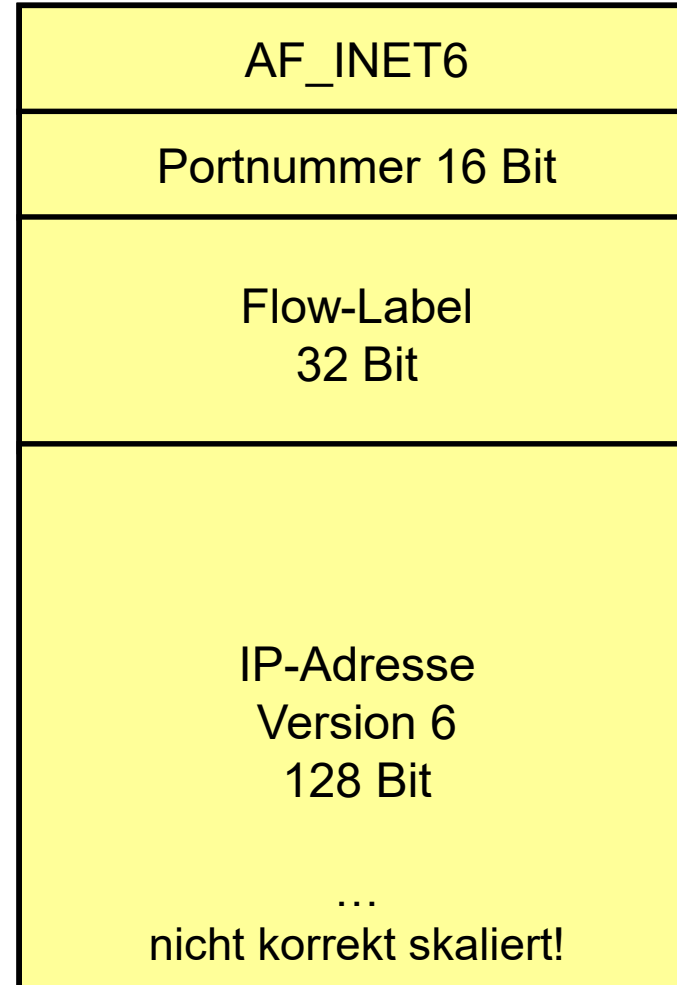
Weitere Socket-Adress-Strukturen

Generische Socket-
Adress-Struktur



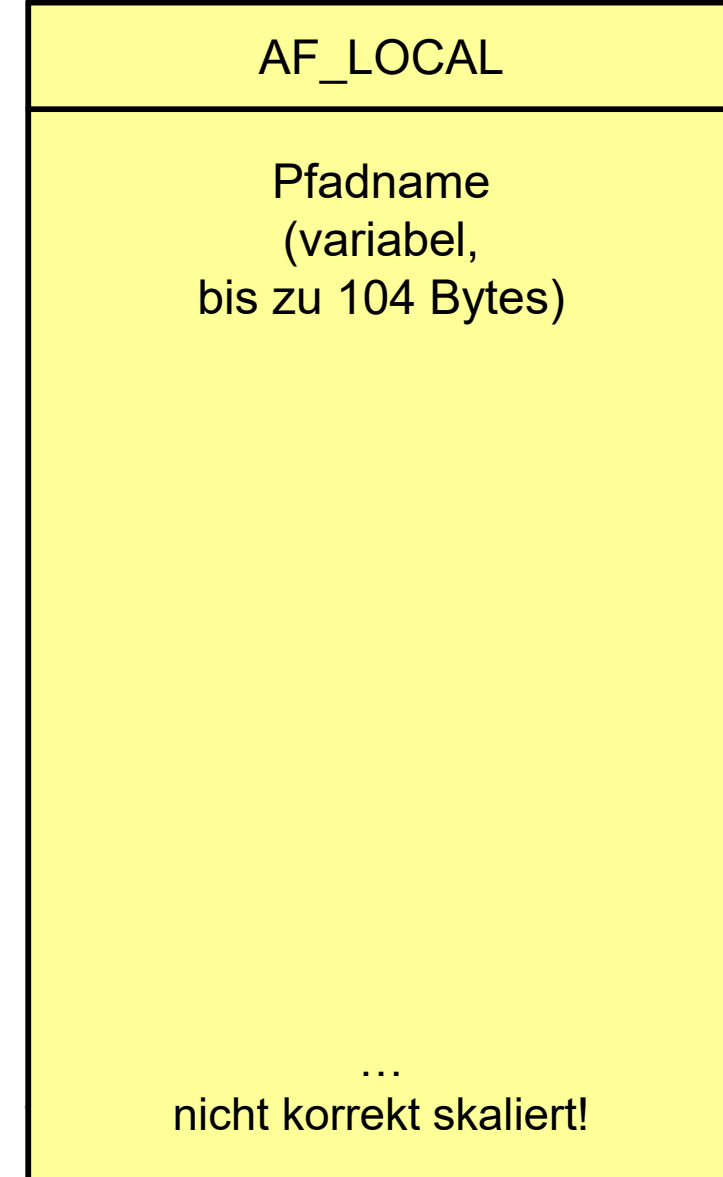
Gesamtlänge: 16 Bytes

Konkrete Adress-Struktur
für die **IPv6-Familie**



Gesamtlänge: 24 Bytes

Konkrete Adress-Struktur
für **Unix-Domain Sockets**



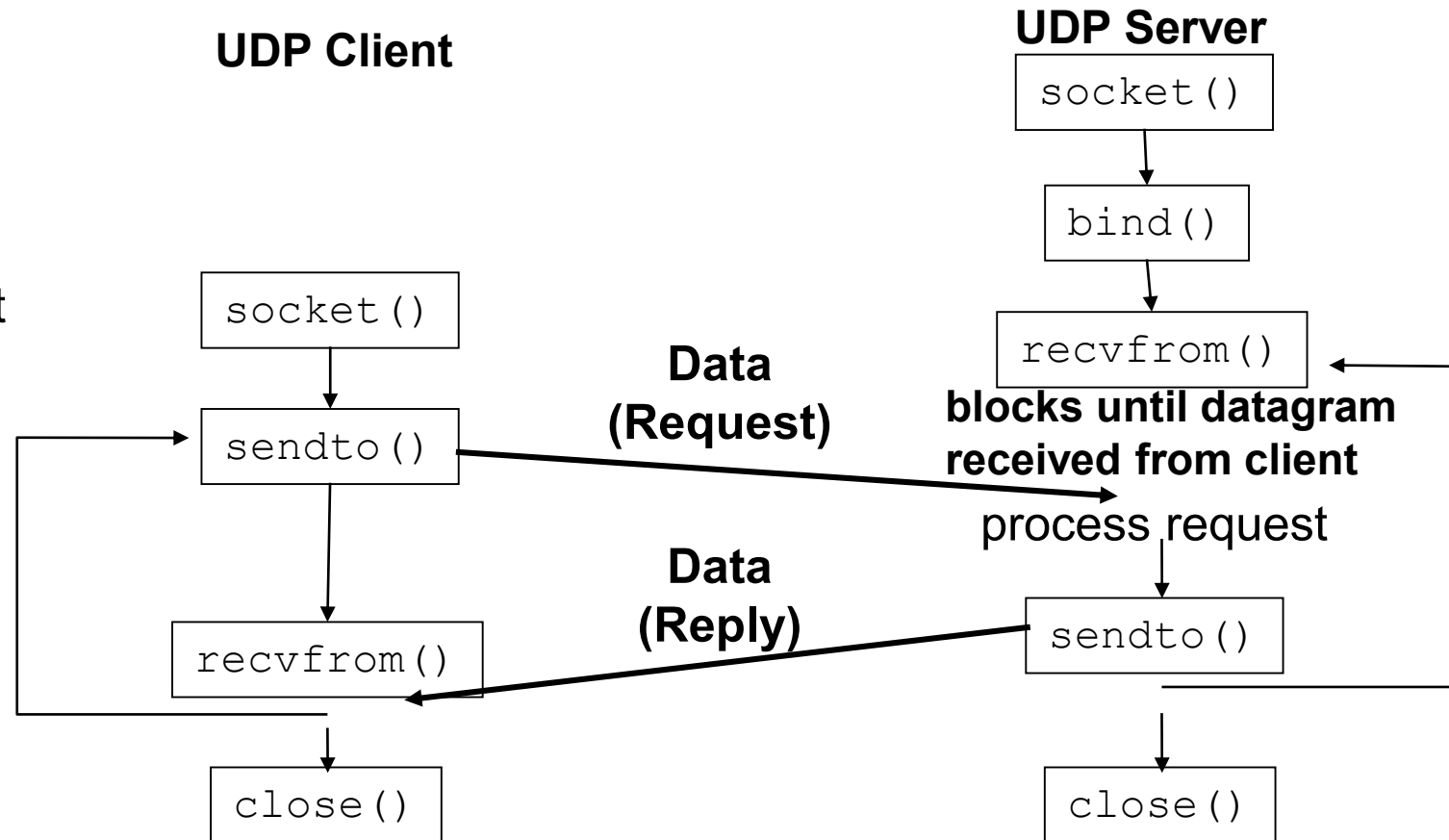
Gesamtlänge: variabel (!)

3.2.4. Kommunikation über Sockets mit UDP

Ablaufschema:

socket() wird in jedem Fall benötigt (aber kein bind()!)

Typisch:
Zyklus aus
Senden,
Warten,
Empfangen



socket() und bind() kennen wir bereits – „**UDP Server** bereitet sich vor“

Typisch:
Zyklus aus
Empfangen,
Verarbeiten,
Beantworten

close(), beim ordentlichen Abschluss schließen des Sockets

Bemerkenswert:

Beim **Client** wird **typischerweise kein bind()** durchgeführt, die sende-seitigen Adressen des Clients werden automatisch vom BS gesetzt.

sendto() – Senden eines UDP Datagramms

Programmsourcesequenz:

```
#include <sys/types.h>
#include <sys/socket.h>

int sendto(int sockfd, const void *buff, size_t nbytes,
           int flags, const struct sockaddr *to, socklen_t tolen);
```

Parameter:

- sockfd File Descriptor des Sockets
- buff **Zeiger auf den Puffer** mit den zu sendenden Daten
- nbytes **Länge** der zu sendenden Daten in Bytes
- flags Optionen, standard ist 0 (weitere ggf. später)
- to **Zeiger** auf Socket-Adress-Struktur mit der **Zieladresse (Port + IP-Adr.)**
- tolen **Länge** der Adress-Struktur

Rückgabewert:

- ≥ 0 Anzahl der gesendeten Bytes
- -1 Fehler

Beispiel von sendto()

```
/* the socket sockfd has been created */
```

```
char msg[64];  
int err;  
struct sockaddr_in dest;  
  
strcpy(msg, "hello, world!");  
  
dest.sin_family = AF_INET;  
dest.sin_port = htons(4711);  
dest.sin_addr.s_addr = inet_addr("130.37.193.13");  
  
err = sendto(sockfd, msg, strlen(msg)+1, 0, \  
(struct sockaddr*) &dest, sizeof(struct sockaddr_in));  
  
if (err < 0) { ... }  
Casting der Adress-  
Struktur
```

Konkrete Zieladresse
wird vorbereitet

Hilfsfunktion `inet_addr()`
auf nächster Folie

Source



Hilfsfunktionen für IP-Adressen

- Eine **IP-Adresse** (von IP Version 4) hat die **Länge 32 Bit**, kann also als ein 32 Bit-Integer-Wert aufgefasst werden.
- Zur **besseren Lesbarkeit** verwenden wir gerne die sog. **Dotted-Decimal-Schreibweise**, die 32-Bit Adresse wird durch 4 Dezimalzahlen (je 8 Bit, Werte von 0 ... 255) getrennt durch Punkte dargestellt:

Bsp.: 131.220.6.15

- Für den Programmierer gibt es **zwei Hilfsfunktionen zur Wandlung**:

```
#include <arpa/inet.h>
```

```
in_addr_t inet_addr(const char *dotted); /* Dotted to Network */  
char *inet_ntoa(struct in_addr network); /* Network to Dotted */
```

Erinnerung:

in_addr_t ist ein Unsigned 32 Bit Integer

recvfrom() – Empfang eines UDP Datagramms

Programmsourcesequenz:

```
#include <sys/types.h>
#include <sys/socket.h>

int recvfrom(int sockfd, void *buff, size_t nbytes,
             int flags, struct sockaddr *from, socklen_t *fromlen);
```

Parameter:

- sockfd File Descriptor des Sockets
- buff Zeiger auf einen **Puffer, in den empfangene Daten geschrieben werden**
- nbytes **Länge des Puffers**
- flags Optionen, standard ist 0 (weitere ggf. später)
- from Zeiger auf Socket-Adress-Struktur mit der **Quelladresse (Port + IP-Adr.)**
- fromlen Zeiger auf Integer mit **Länge der Quell-Adress-Struktur**

recvfrom() – Empfang eines UDP Datagramms

Programmsourcesequenz:

```
#include <sys/types.h>
#include <sys/socket.h>

int recvfrom(int sockfd, void *buff, size_t nbytes,
             int flags, struct sockaddr *from, socklen_t *fromlen);
```

Im Normalfall blockiert ein Aufruf von `recvfrom()` solange, bis ein Datagramm empfangen wurde (Ausnahmen später).

Rückgabewert:

- `>= 0` Anzahl der empfangenen Bytes
- `-1` Fehler
- `from` ist ein Zeiger auf eine **Adress-Struktur**, die die **Adressen des Absenders** des UDP-Datagramms enthält (IP-Adresse, Portnummer)
- wenn der Empfänger eine Antwort schicken möchte, benötigt er diese Adressen!
- wenn den Empfänger die Quelladresse nicht interessiert, so können `*from` und `*fromlen` NULL-Zeiger sein

Es ist auch möglich, 0 Byte zu empfangen bzw. zu senden!

Beispiel von recvfrom()

```
/* the socket sockfd has been created
   and bound to a port number */
```

```
char msg[64];
int len, flen;
struct sockaddr_in from;
```

Adress-Struktur
für Quelladresse
wird bereitgestellt

```
flen = sizeof(struct sockaddr_in);
```

Buffer kann max.
64 Byte aufnehmen

```
len = recvfrom(sockfd, msg, sizeof(msg), 0, \
                (struct sockaddr*) &from, &flen);
```

```
if (len < 0) { ... }
```

```
printf("Received %d bytes from host %s port %d: %s", len, \
       inet_ntoa(from.sin_addr), ntohs(from.sin_port), msg);
```

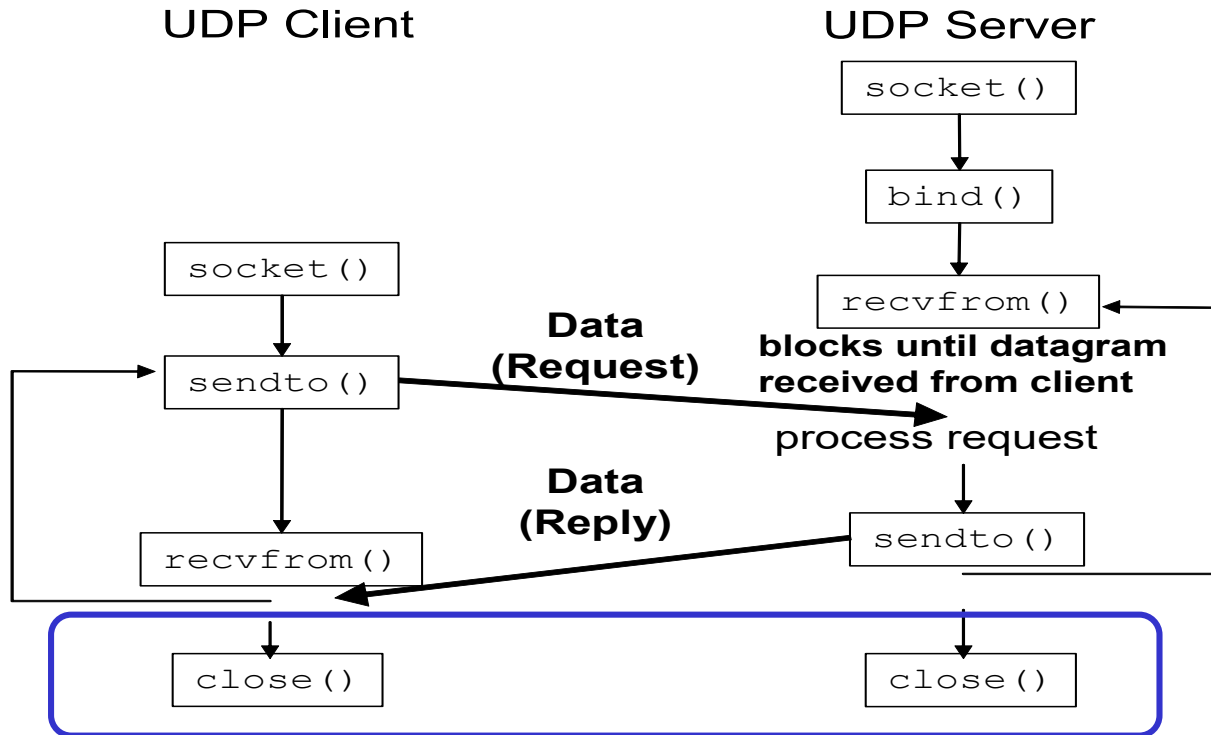
Hilfsfunktion zur Ausgabe der IP-Adresse als Dotted-Decimal

Wandlung der Portnummer in
Host-Byte-Order nicht vergessen!

Source



Schließen des Socket



Wenn ein Socket nicht mehr benötigt wird, so sollte er geschlossen werden.

Programmsourcesequenz:

```
#include <unistd.h>
```

```
int close(int sockfd);
```

Rückgabewert:

- 0 bei Erfolg
- -1 bei Fehler

Nach dem Schließen eines Sockets darf auf den Socket-Descriptor nicht mehr zugegriffen werden.

Die **benutzte Portnummer wird vom BS wieder freigegeben**. Am Client kann z.B. der Port dynamisch wieder neu vergeben werden, am Server kann auf einen spezifischen „well-known“ Port ein neuer Prozess zugreifen.