TCP/IP

1 Internet Protocol

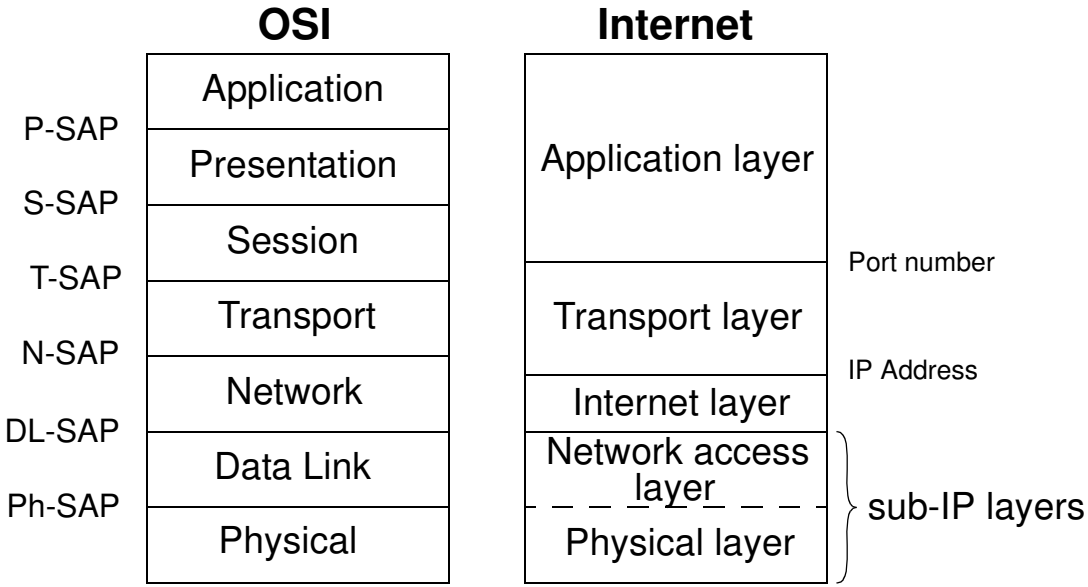
2 Funktionen des Internet Protokolls

3 Transportschichtprotokolle

1 Internet Protocol

1.1 Internet Protocol Layer Architecture

Die Internet Architektur unterscheidet sich grundlegend vom **OSI Schichtenmodell** (OSI Referenz Model), wie im folgenden Bild dargestellt ist:

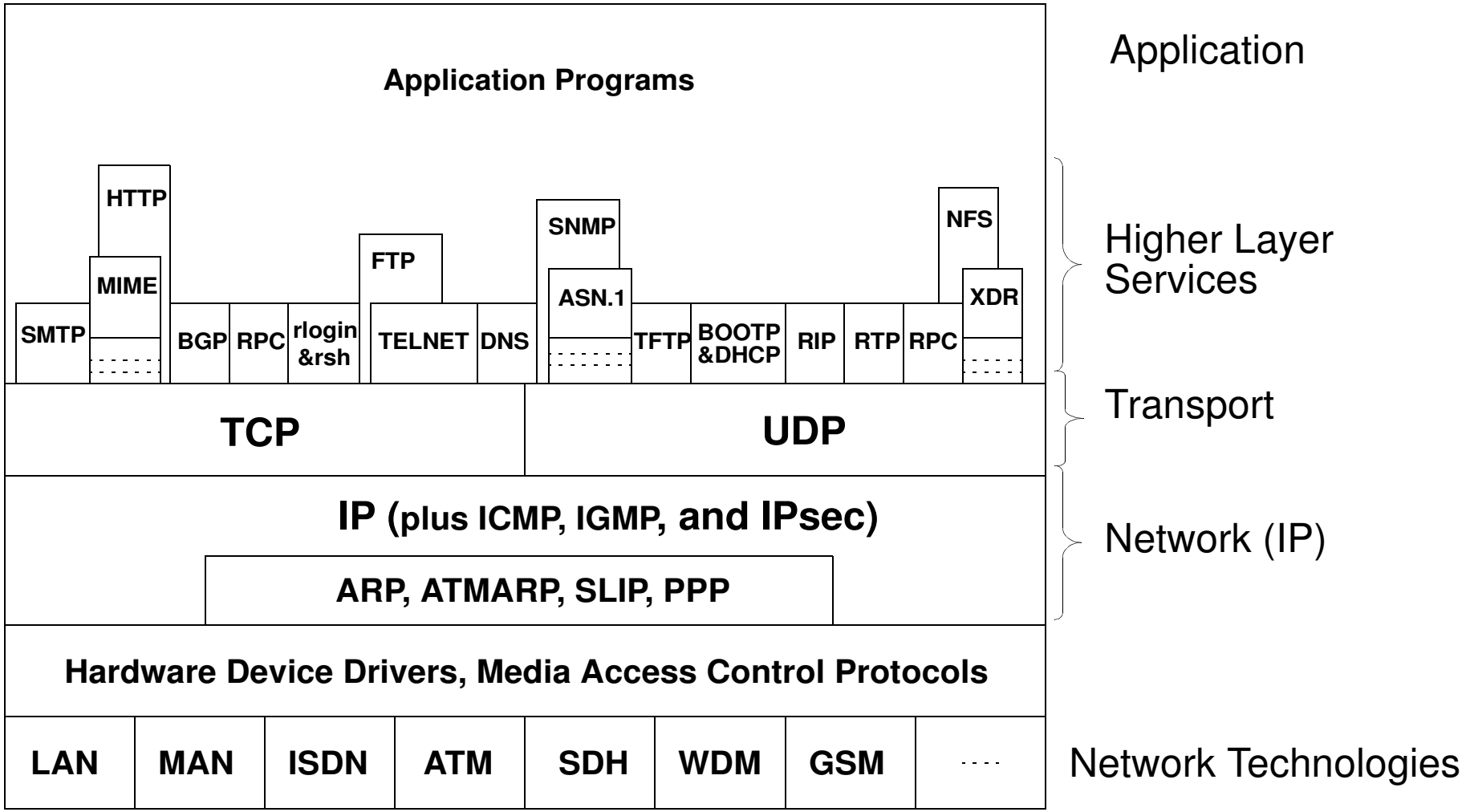


Unterschiede zwischen OSI Schichtenmodell und Internet Protokoll Architektur

- Nur 5 Protocolschichten
- Keine strikte Trennung zwischen den Schichten

Im Gegensatz zum OSI Modell ist das Internet Modell aufgrund konkreter Anforderungen entstanden.

1.2 Internet Protokoll Suite



AbkürzungslisteList of Acronyms

LAN	Local Area Netwowrk
MAN	Metropolitan Area Network
ISDN	Integrated Services Digital Network
ATM	Asynchronous Transfer Mode
SDH	Synchronous Digital Hierarchy
WDM	Wavelength Division Multiplex
GSM	Global System for Mobile Comm.
IP	Internet Protocol
ICMP	Internet Control Message Protocol
IGMP	Internet Group Management Protocol
ARP	Address Resolution Protocol
SLIP	Serial Line IP
PPP	Point-to-Point Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
TELNET	Remote Terminal Service Protocol
FTP	File Transfer Protocol
RPC	Remote Procedure Call
BGP	Border Gateway Protocol
RLOGIN	Remote Login
DNS	Domain Name System
TFTP	Trivial File Transfer Protocol
ASN.1	Abstract Syntax Notation .1 Standard (for SNMP)
SNMP	Simple Network Management Protocol
RTP	Real Time Protocol
SMTP	Simple Mail Transfer Protocol
HTTP	Hypertext Transfer Protocol
MIME	Multipurpose Internet Mail Extensions
NFS	Network File System
XDR	External Data Representation Standard
RIP	Routing Information Protocol
BOOTP	Bootstrap Protocol
DHCP	Dynamic Host Configuration Protocol

2 Funktionen des Internet Protokolls

Der Hauptzweck des Internet Prokolls ist die Weiterleitung von Datagrammen zwischen Endsystemen

- **Hop-by-Hop Routing:** Jedes Paket wird individuell in jedem Router gerouted und weitergeleitet (Forwarding), basierend auf der Zieladdresse. Pakete können unterschiedliche Pfade durchs Netz nehmen und sich gegenseitig überholen.
- "**Best Effort Service**", d.h. keinerlei Garantien bezüglich
  - Paketverlust und Reihenfolge
  - Bitfehlern in transportierten Daten
  - Flusssteuerung oder Stauregelung (Congestion Control)
  - minimalem Durchsatz oder Verzögerung

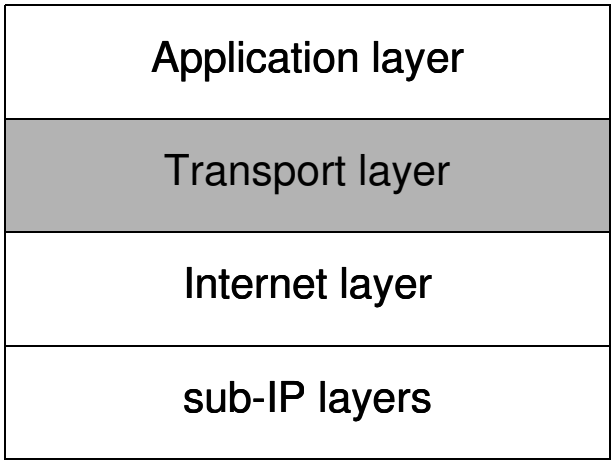
Geschichte des Internet Protokoll - IPv4 and IPv6

Die erste weitflächig eingesetzt Version von IP is IP version 4 (IPv4), welches 1981 im RFC 791 standardisiert wurde. Dieser Standard ist fast 30 Jahre später im noch im Einsatu. Mitte der 90ziger begann die Entwicklung des Nachfolgeprotokolls IPv6, welches derzeit nur in sehr begrenztem Umfang eingesetzt wird. Das Prinzip von IPv6 ist dem von IPv4 sehr ähnlich mit Ausnahme der Addresslänge und einer zusätzlicher Merkmale.

3 Transportschichtprotokolle

3.1 Einführung

Die IP-Schicht bietet einen unzuverlässigen Datagramzustelldienst zwischen Endsystemen (Rechner, Router, ...) an. Darauf aufsetzend kommen Transportprotokolle zum Einsatz um Daten zwischen zwei Prozessen oder Applikationen auszutauschen. Im Internet Protokollstack befindet sich die Transportschicht zwischen Anwendungsschicht und Internetschicht..

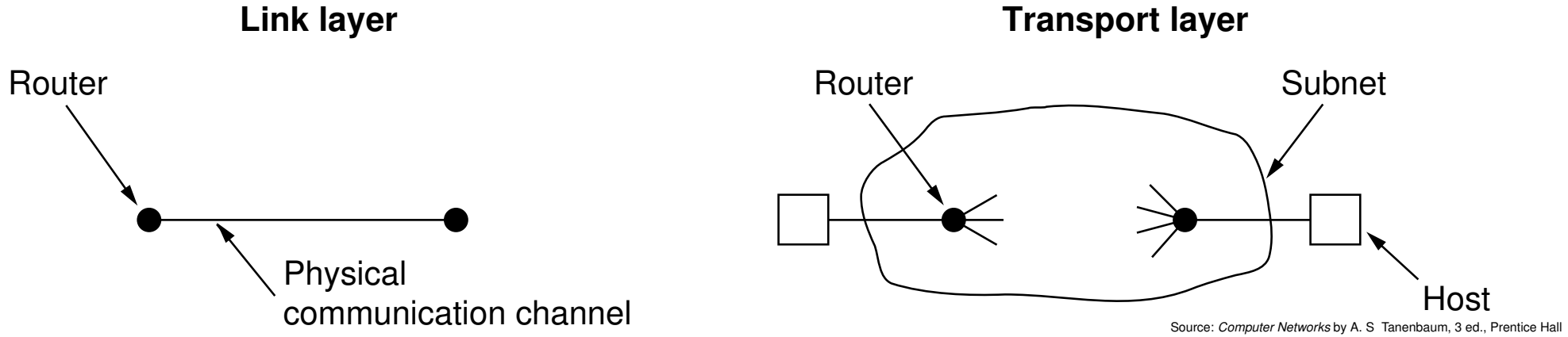


Im Allgemeinen können zwei Arten von Transportprotokollen unterschieden werden:

- **Verbindungslose** Protokolle
- **Verbindungsorientierte** Protokolle

3.1.1 Datensicherungsschicht (link layer) im Gegensatz zur Transportschicht

Es gibt einige Gemeinsamkeiten zwischen der Datensicherungsschicht und der Transportschicht, aber auch deutliche Unterschiede:



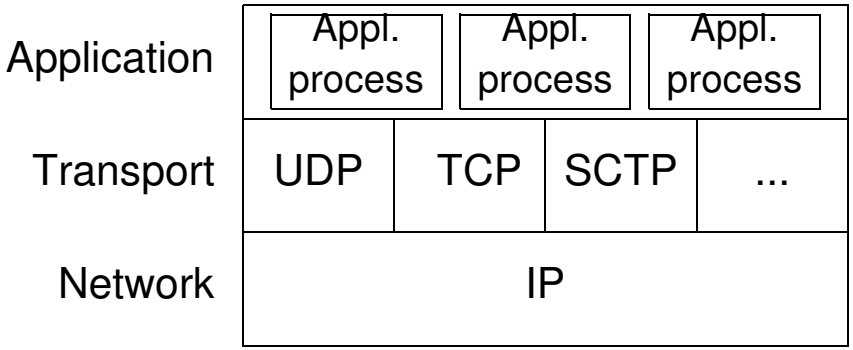
	Datensicherungsschicht	Transportschicht
Art der Verbindung	Direkte Kommunikation über physikalischen Kommunikationskanal.	Ende-zu-Ende Kommunikation über mehrere Verbindungen mit unterschiedlichen Technologien
Bandbreite	Eher konstant	Variabel aufgrund statistischem Multiplexing des Verkehrs, variabler Anzahl an Verbindungen, ...
Verzögerung	Eher konstant	Kann über mehrere Größenordnungen varieren (ms to s)

3.1.2 Internet Transportprotokolle

Im Internet werden hauptsächlich zwei Transportprotokolle verwendet:

- User Datagram Protocol (UDP)
- Transmission Control Protocol (TCP)

Sowohl UDP als auch TCP verwenden Portnummern um sendende und empfangende Prozesse auf Endsystemen zu identifizieren.



3.2 User Datagram Protocol (UDP)

3.2.1 Header Format

Das User Datagram Protokoll bietet einen einfachen Mechanismus um Datagramme zwischen Anwendungen auszutauschen. UDP verwendet Quell- und Zielport für die Zustellung von Nachrichten. UDP ist verbindungslos, d.h. es gibt keine Funktionen um mit Fehlern umzugehen. UDP erlaubt das Multiplexing/Demultiplexing von UDP Datagrammen auf verschiedene Ports. UDP wird in RFC 768 beschrieben.

Das UDP Nachrichtenformat ist sehr einfach:

- UDP Quell-/Zielport
- UDP Datagramm Länge (einschließlich UDP header)
- UDP Prüfsumme (16 bit 1-Komplement)

Source port (16)	Destination port (16)
Datagram length (16)	UDP checksum (16)
Data (var.)	

3.2.2 Beispiele für die Anwendung von UDP

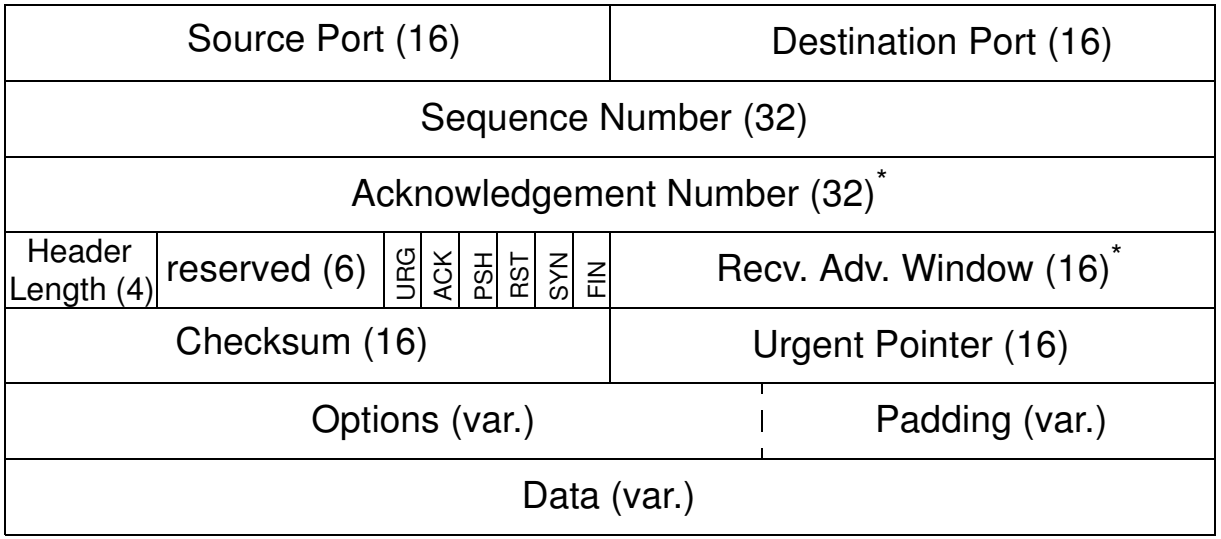
UDP wird überall dort verwendet wo pünktliche Ankunft wichtiger ist als Zuverlässigkeit, z.B. bei Audioströmen, Videoströmen, ... Es wird auch dort verwendet wo die Kosten für einen Verbindungsaufbau im Vergleich zu den übertragenen Daten zu hoch sind (z.B. Query/Response Anwendungen). In solchen Fällen wird die Zuverlässigkeit durch Mechanismen auf der Anwendungsebene realisiert.

- Multimedia-Anwendungen: Real-time Transport Protocol (RTP)
- Netzbasierter Zugriff auf Dateien: Network File System (NFS)
- Netzmanagement: Simple Network Management Protocol (SNMP)
- Adressauflösung: Domain Name System (DNS)
- ...

3.3 Transmission Control Protocol (TCP)

- Verbindungsorientiert
- Bidirektional
- Punkt-zu-Punkt Transport
- Zuverlässig
- Reihenfolgesicherung
- Bytestrom ("Byte-Pipe") zwischen Anwendungen oder Prozessen
  Sendestrom wird in Segmente bestimmter Größe aufgeteilt und an die IP-Schicht weitergeleitet und bei der empfangenden Station wieder entsprechend zusammengesetzt.
- Verwendete Bandbreite wird an die Eigenschaften des Netzes angepasst (Congestion Control)

3.3.1 Kopfformat



Flags:

URG Urgent pointer valid
ACK Acknowledgement field valid
PSH Push request
RST Connection reset
SYN Synchronous sequence number
FIN End of byte stream

* refers to data transmission in reverse direction (piggy-backed)

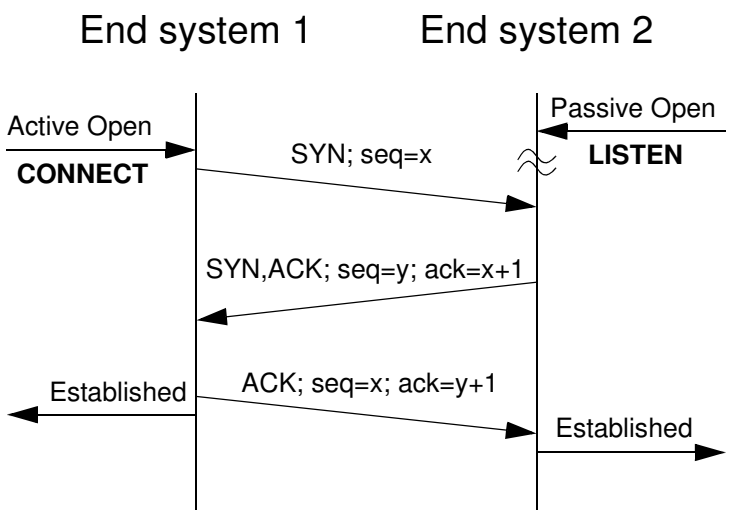
- Quell- und Zielport
- Sequenznummer: Zeigt auf das nächste zu sendende Byte
- Acknowledgementnummer: Zeigt auf das nächste aus der Gegenrichtung zu erwartende Byte.
- TCP Kopflänge: Gibt die Anzahl an 32 Bit Worten im TCP Kopf an (Länge des Kopfes kann aufgrund der Optionen varieren).
- Receiver Advertised Window: Wird zur Flusskontrolle verwendet.

- Flags
  - ACK: Acknowledgementnummer ist gültig; im Allgemeinen im normalen Datenaustausch gesetzt
  - SYN, RST and FIN: Genutzt während des Verbindungsaufbau und -abbau genutzt
  - PSH: Push function: Empfänger soll Daten sofort an Anwendung ausliefern
  - URG: Der "Urgent-Zeiger" ist gültig. Die entsprechenden Daten sollen sofort an die Anwendung ausgeliefert werden
- **Checksum**
  - 16-bit 1-complement Prüfsumme über Kopf und Daten
  - Die Prüfsumme umfasst auch einen Pseudoheader: IP Quelladresse, IP Zieladresse, TCP Protokollnummer (6), and the TCP Länge (Länge des Headers plus Payload).

3.3.2 Verbindungsmanagement

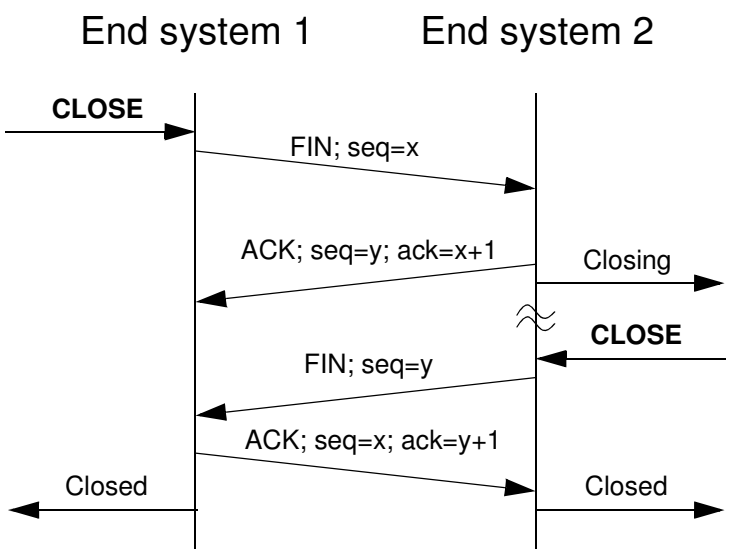
Verbindungsauf- und abbau

Aufbau: "Three-way handshake"



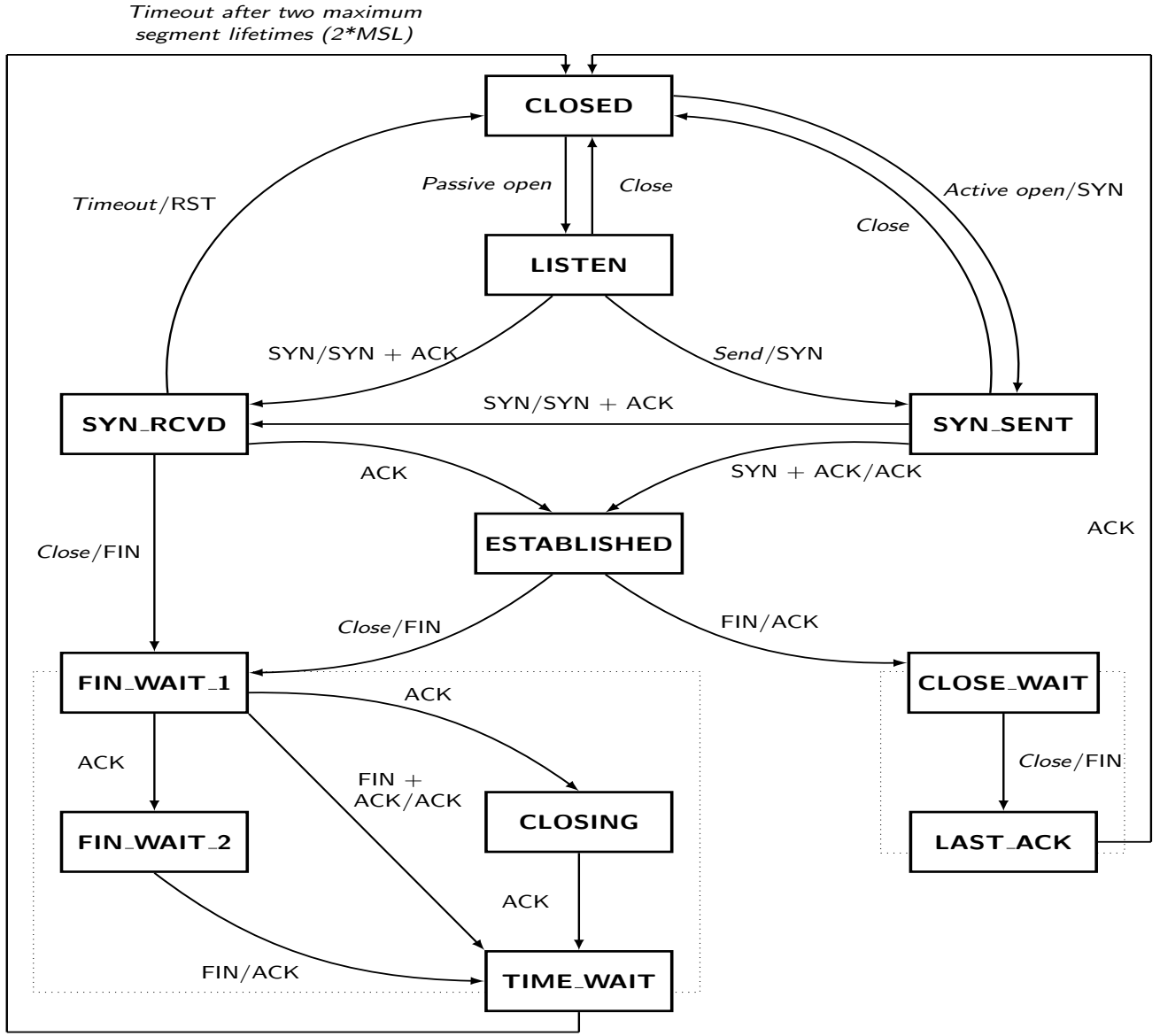
- Durch ein "**passive open**" wird eine TCP Entität erzeugt die Verbindungen erwartet
- **"Active open"** baut Verbindung auf
- Während des "three-way handshake", werden die initialen Sequenznummern ausgetauscht (mit gesetztem SYN bit)

Abbau: "Four-way handshake"



- Abbau muß zwischen Endsystemen koordiniert werden.
- FIN bit signalisiert Verbindungsabbau
- RST bit signalisiert Verbindungsreset
- In Grenzfällen kann Verbindung "halboffen" bleiben

3.3.3 TCP Zustandsautomat



Socket-Programmierung

1 Einführung

2 TCP Socket Programmierung in C

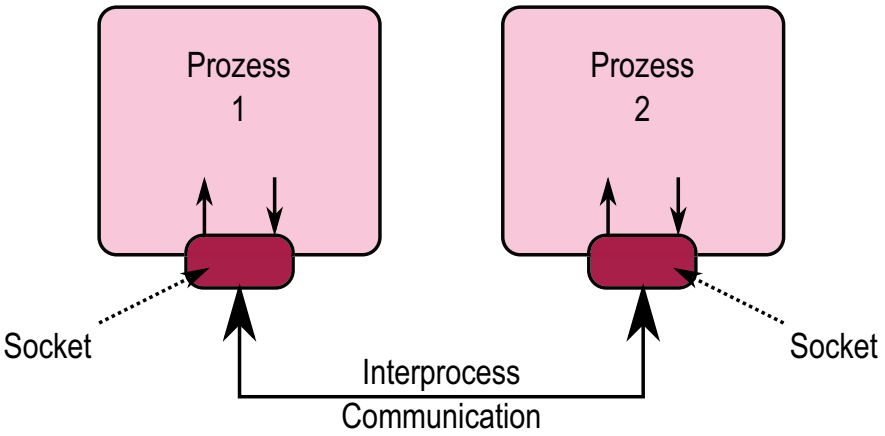
3 UDP Socket Programmierung in C

4 Socket Programmierung in Java

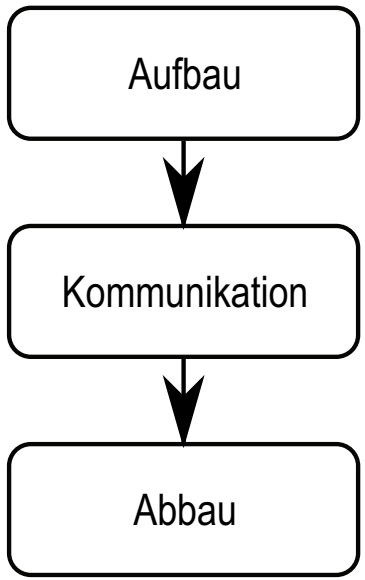
1 Einführung

1.1 Grundkonzepte von Sockets

- Endpunkt eines Kanals zur **bidirektionalen** Kommunikation zwischen zwei Prozessen
- Schreiben und Lesen analog zu einer Datei "Everything in Unix is a file"
- API einer betriebssystemnahen Bibliothek (libc, winsock.dll,...)
- Berkeley Socket API de facto Standard
  - Zunächst nicht standardisiert, mittlerweile im POSIX-Standard
  - 1983 eingeführt in BSD an der Universität von Kalifornien in Berkeley (UC Berkeley)
- Unterscheidung von
  - Server Socket: Stellt Endpunkt zur Verfügung
  - Client Socket: Verbindet sich mit Server Socket



1.2 Lifecycle von Sockets



- Aufbau
  - Festlegung der Eigenschaften
  - Client Socket: Kontaktaufnahme mit anderem Prozess
  - Server Socket: Wartet Verbindungsaufbau durch Client ab
- Kommunikation
  - Austausch von Daten
- Abbau
  - Abbau der Verbindung
  - Freigeben von Ressourcen

1.3 Eigenschaften von Sockets

- Art übertragenen Daten
  - Stream: kontinuierlicher Bytestrom
  - Datagram: Menge an zusammenhängenden Bytes

Einführung

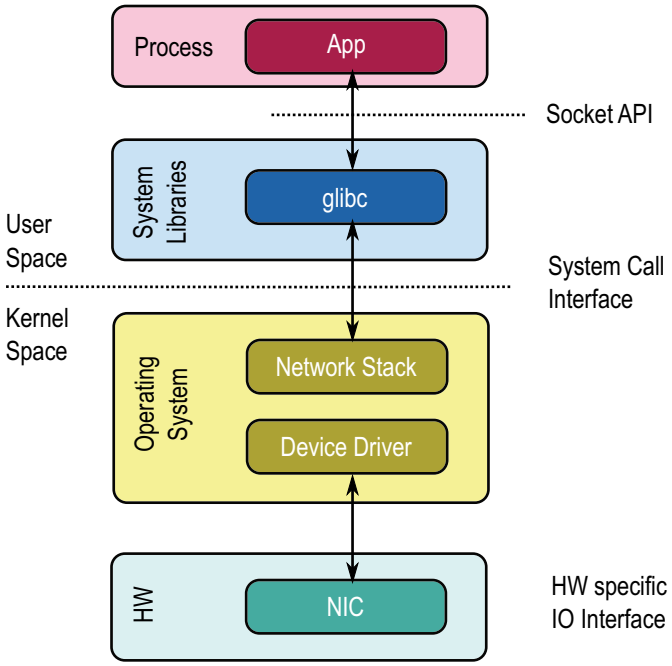
- Zuverlässigkeit
  - Bytes können verloren gehen (Unreliable)
  - Bytes können nicht verloren gehen (Reliable)
- Unicast vs. Multicast
  - Unicast:
    - Daten werden an genau einen anderen Prozess gesendet
    - Daten werden von genau einem anderen Prozess empfangen
  - Multicast
    - Daten werden gleichzeitig an mehrere andere Prozesse gesendet
    - Daten werden von mehreren anderen Prozessesn empfangen
- Addressierung: Wie wird ein anderer Prozess addressiert?
  - Dateisystem (Lokaler Namensraum): /home/user/filex
  - Netzaddressen (Internet Namensraum): 192.168.1.1:80

$\rightarrow$ Eigenschaften lassen sich auf konkrete Protokolle und Mechanismen abbilden

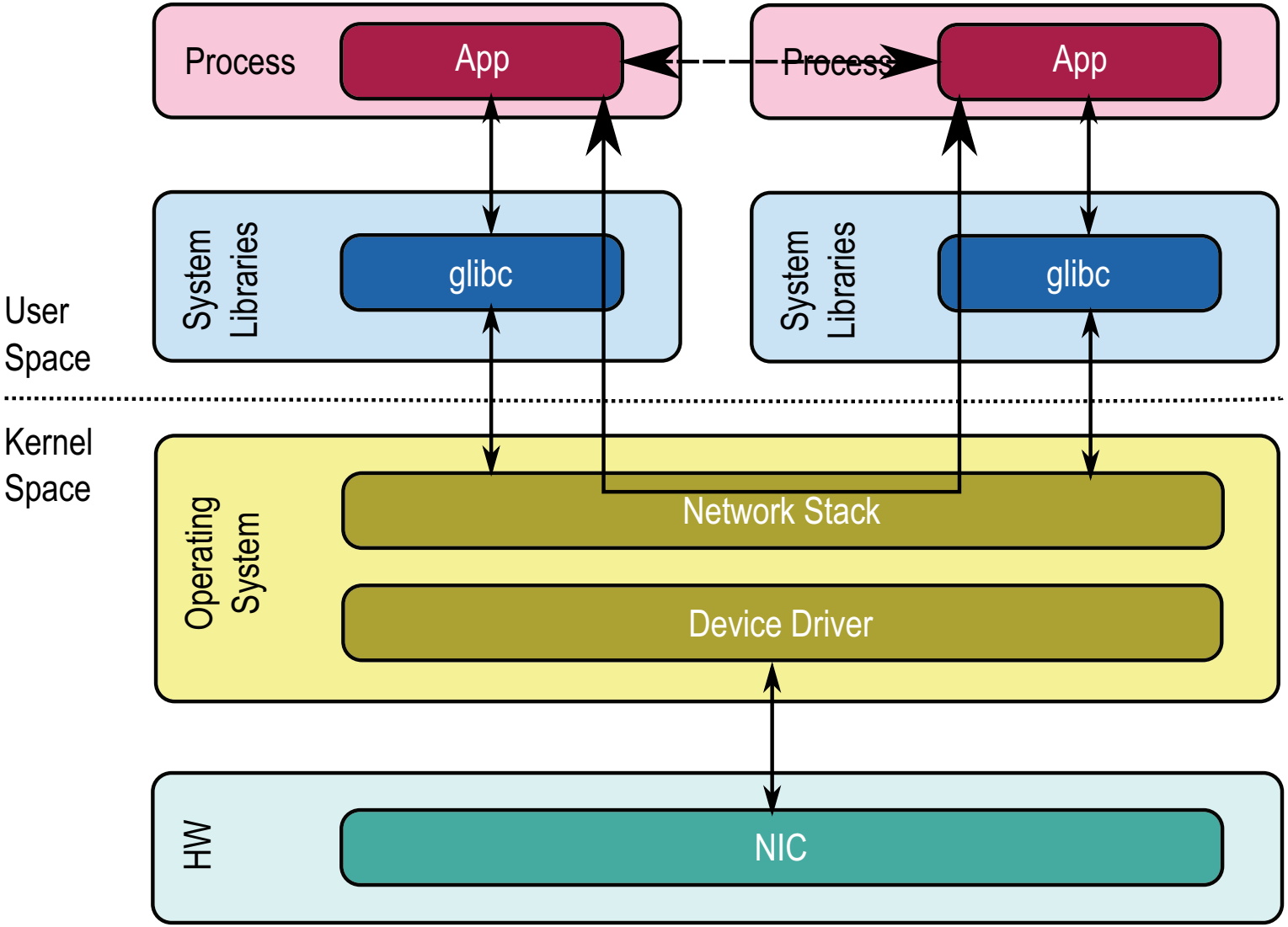
1.4 Einordnung im Betriebssystem

1.4.1 Allgemeine Übersicht

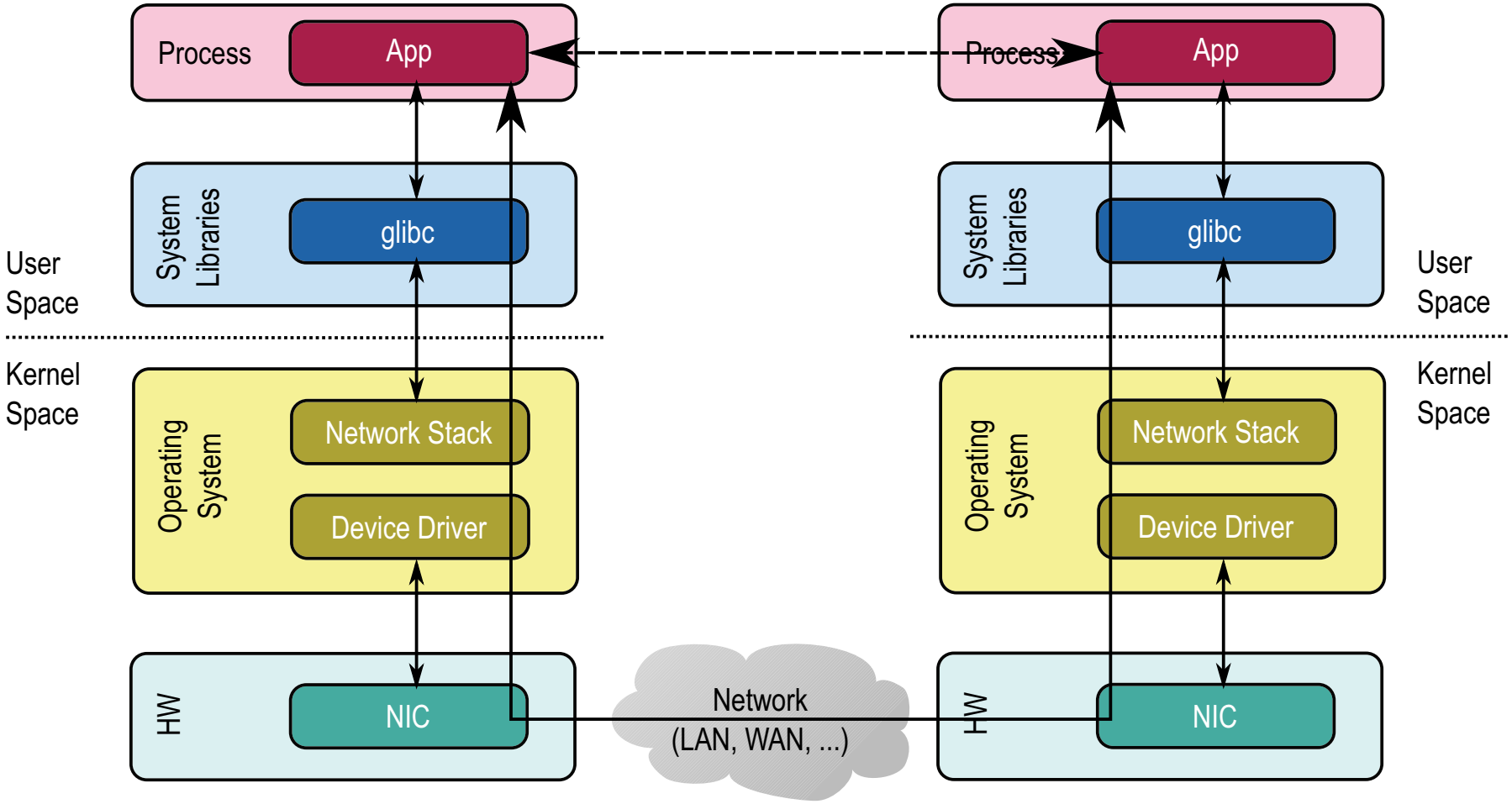
- Socket API: Schnittstelle zwischen Prozess/Applikation und systemnaher Bibliothek (libc, winsock.dll,...)
- System Call Interface:
  - Bindeglied zwischen User Space und Kernel Space
  - Signalisert Kernel die Allozierung der entsprechenden Ressourcen
- Hardware spezifisches IO Interface: Schnittstelle zwischen Betriebssystem und Hardware (z.B. Netzinterface)



1.4.2 Sockets zur Lokalen Interprozesskommunikation



1.4.3 Sockets zur Verteilten Interprozesskommunikation



1.5 Klassifikation von Sockets

Sockets lassen sich mittels der oben eingeführten Eigenschaften anhand von
Domänen, Typen, und Protokollen unterscheiden.

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

- Domäne: Bestimmt die Art des Sockets. Die folgenden Socketarten können unterschieden werden
  - AF_UNIX: Lokale Kommunikation über Dateien (Pipes)
  - AF_INET: Kommunikation über IPv4 Protokoll
  - AF_INET6: Kommunikation über IPv6 Protokoll
  - AF_NETLINK: Spezieller Sockettyp der es erlaubt mit dem UNIX/LINUX Kernel zu kommunizieren. Zum Beispiel stellt der Linux Kernel einen speziellen NetLink Socket zur Verfügung über den man Ereignisse über zur Verfügung stehende Netzinterfaces erhalten kann.
  - AF_PACKET: Direkte Kommunikation über das Ethernet Netzwerkinterfaces
  - $\rightarrow$ Darüberhinaus gibt es weitere Socketarten für spezielle Protokolle (IPX, Appletalk, ...)

- Type: Jede Socketart bietet ein oder mehrere verschiedene Kommunikationstypen an. Im Falle von AF_INET Sokkets ist offensichtlich dass es in aller Regel zwei verschiedene Arten von Protokollen über IP gibt. Es kann zwischen UDP und TCP basierter Kommunikation unterschieden werden. Die folgenden Sockettypen abstrahieren von den konkreten Protokollnamen.
  - SOCK_STREAM: Sichert die folgenden Eigenschaften zu
    - Zuverlässigkeit
    - Reihenfolgesicherung
    - Verbindungsorientiert
    - Austausch bidirectionaler Byteströme

    $\rightarrow$ Die Kombination von PF_INET und SOCK_STREAM stellt einen Socket dar welcher Daten über das TCP/IP Protokoll austauscht.

- SOCK_DGRAM: Beschreibt einen Datagram-Service mit den folgenden Eigenschaften
  - Unzuverlässig
  - Verbindungsloss
  - Austausch von Nachrichten mit fester Länge
  - Unidirectional

  → Die Kombination von PF_INET und SOCK_DGRAM stellt einen Socket dar welcher Daten über UDP und IP austauscht.
- SOCK_RAW: Erlaubt den direkten Zugriff auf das Protokoll des Sockets der durch die Domäne beschrieben wurde.

→ Darüberhinaus gibt es weitere Typen die hier nicht weiter von Interesse sind

- Protokoll: Beschreibt die Kombination aus Domäne und Typ ein Kommunikationsprotokoll nicht eindeutig, kann dies hiermit erfolgen. Für die hier betrachteten Sockets spielt das Feld keine Rolle.

Nicht jede Kombination aus Domäne, Type und Protokoll lässt sich verwenden. Im folgenden sollen die folgenden Kombinationen näher betrachtet werden

- TCP-Socket: (AF_INET, SOCK_STREAM, NULL)
- UDP-Socket: (AF_INET, SOCK_DGRAM, NULL)
- Unix-Socket: (AF_UNIX, SOCK_STREAM, NULL)

1.6 Beschreibung der Kommunikatonsprimitive für TCP-Socket.

1.6.1 Eigenschaften von TCP/IP

Es muß zwischen Client-Sockets und Server-Sockets unterschieden werden

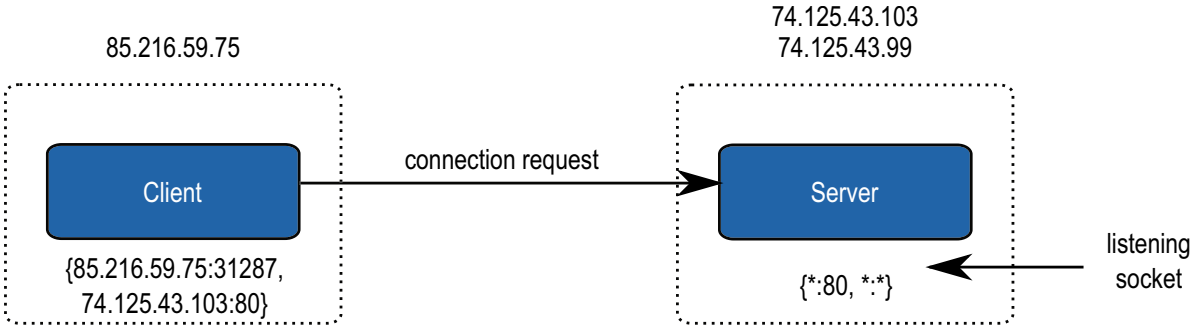
- Server muss auf einer bestimmten IP-Addresse und einem bestimmten Port für Verbindungsanfragen hören
  Zum Beispiel: Web-Server von www.spiegel.de
  IP-Addresse: 195.71.11.67
  Port: 80 (Well-known Port)
- Client muß sich zu dieser Addresse über das lokale Netzinterface verbinden
- Nach erfolgreichem TCP Verbindungsaufbau (SYN, SYN/ACK, ACK) steht eine zuverlässige Verbindung zwischen Client und Server zur Verfügung
- Eine Verbindung wird durch das sogenannte 5-Tupel beschrieben

- Quell und Ziel IP-Addresse
- Quell und Ziel TCP Port
- Transportprotokoll: TCP, UDP, ...

1.6.2 Übersicht über Verbindungsaufbau

Notation: {Lokale-IP-Addresse:Lokaler-Port, Entfernte-IP-Addresse:Entfernter-Port }

- Ein Server erwartet Verbindungsanfragen **auf allen IP Addressen**, jeweils auf **Port 80** (Listening Socket oder auch Server Socket) und akzeptiert Verbindungsanfragen von allen Clients. Dies wird durch die Notation {*:80, *:*} symbolisiert.



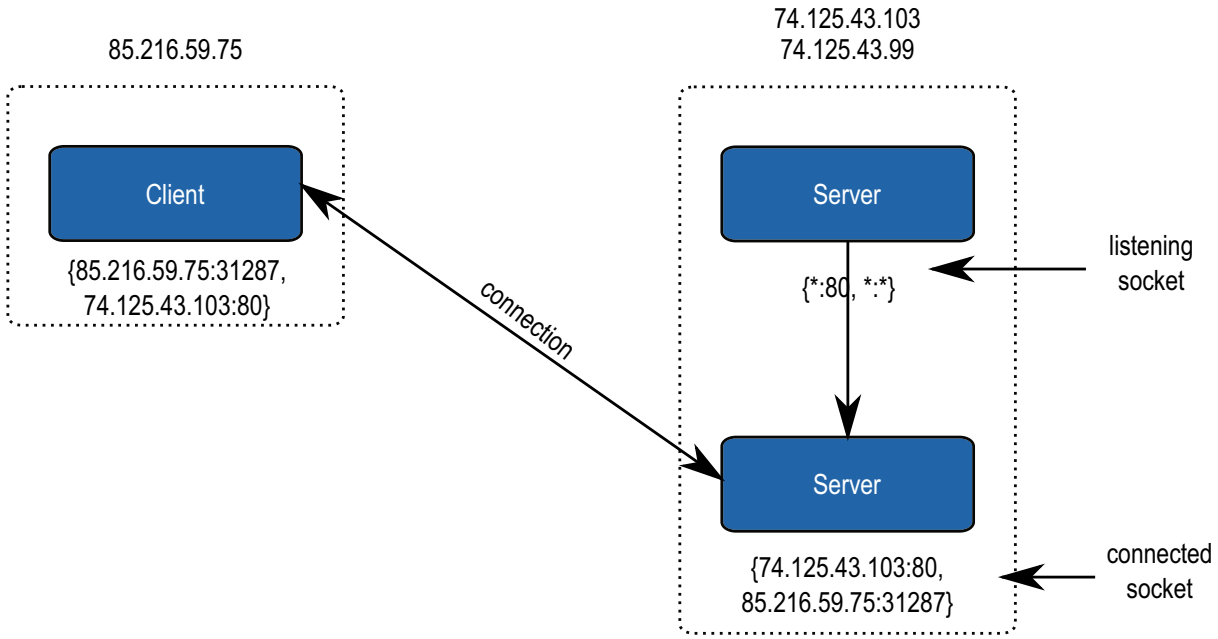
- Der **Client verbindet sich** mit dem "Listening Socket". Hierzu verwendet der Client eine seiner IP-Addressen und wählt einen **nichtbenutzten Port als Quellport** aus. Dieser Port wird als **ephemeral ("flüchtig")** Port bezeichnet . Für ephemeral Ports steht ein vorgegebener Bereich zur Verfügung der betriebsysstem- und konfigurationsabhängig ist.
  Unter Linux lässt sich die Einstellung wie folgt überprüfen.

```
cat /proc/sys/net/ipv4/ip_local_port_range
```

```
32768    61000
```

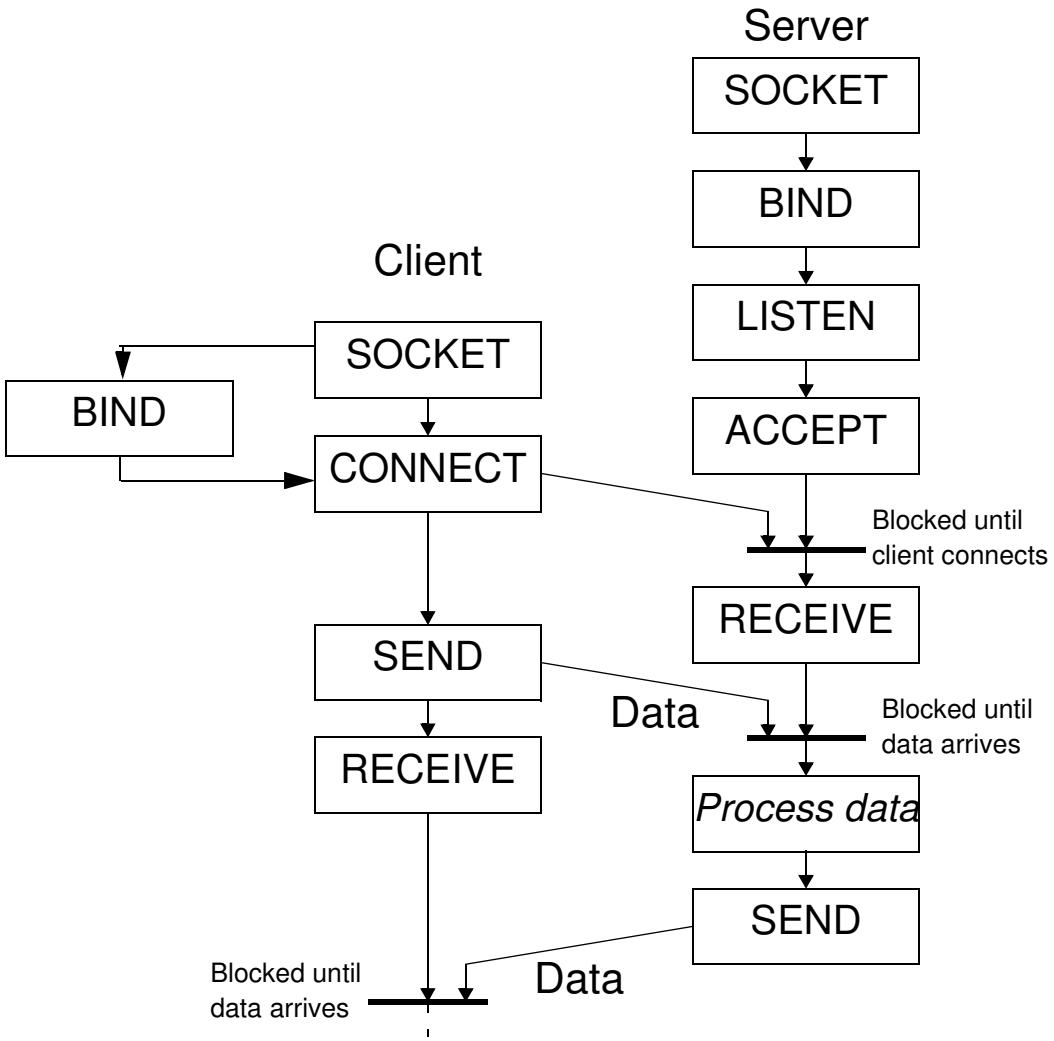
Üblicherweise merkt das Betriebssystem sich den zuletzt verwendeten ephemeral Port und inkrementiert diesen um 1 für die nächste Verwendung des Sockets.

- Nach erfolgreichem Verbindungsaufbau (SYN, SYN-ACK, ACK) **erzeugt der Server einen neuen Sokket**, welcher für die Kommunikation mit dem Client zuständig ist.
- **Der Server Socket existiert weiter** und kann weitere Verbindungsanfragen entgegen nehmen. Dies ist allerdings vom multithreading Modell abhängig (s.u.).



1.6.3 Socketprimitive

- **SOCKET:** Erzeugt einen neuen Kommunikationsendpunkt.
- **BIND:** Bindet einen Socket an eine lokale Netzaddresse und ggf. an einen lokalen Port.
- **LISTEN:** Signalisiert die Absicht auf diesem Port Verbindungen anzunehmen. Hiermit lässt sich die Größe der Warteschlange für noch nicht bediente Verbindungsanfragen einstellen
- **ACCEPT:** Ein Serversocket wartet auf Verbindungsanfragen von Clients. Der aufrufende Prozess wird solange **blockiert** bis entweder ein Timer abläuft (einstellbar) oder eine Verbindung aufgebaut wird.
- **CONNECT:** Ein Clientsocket baut eine Verbindung zu einem Serversocket auf.

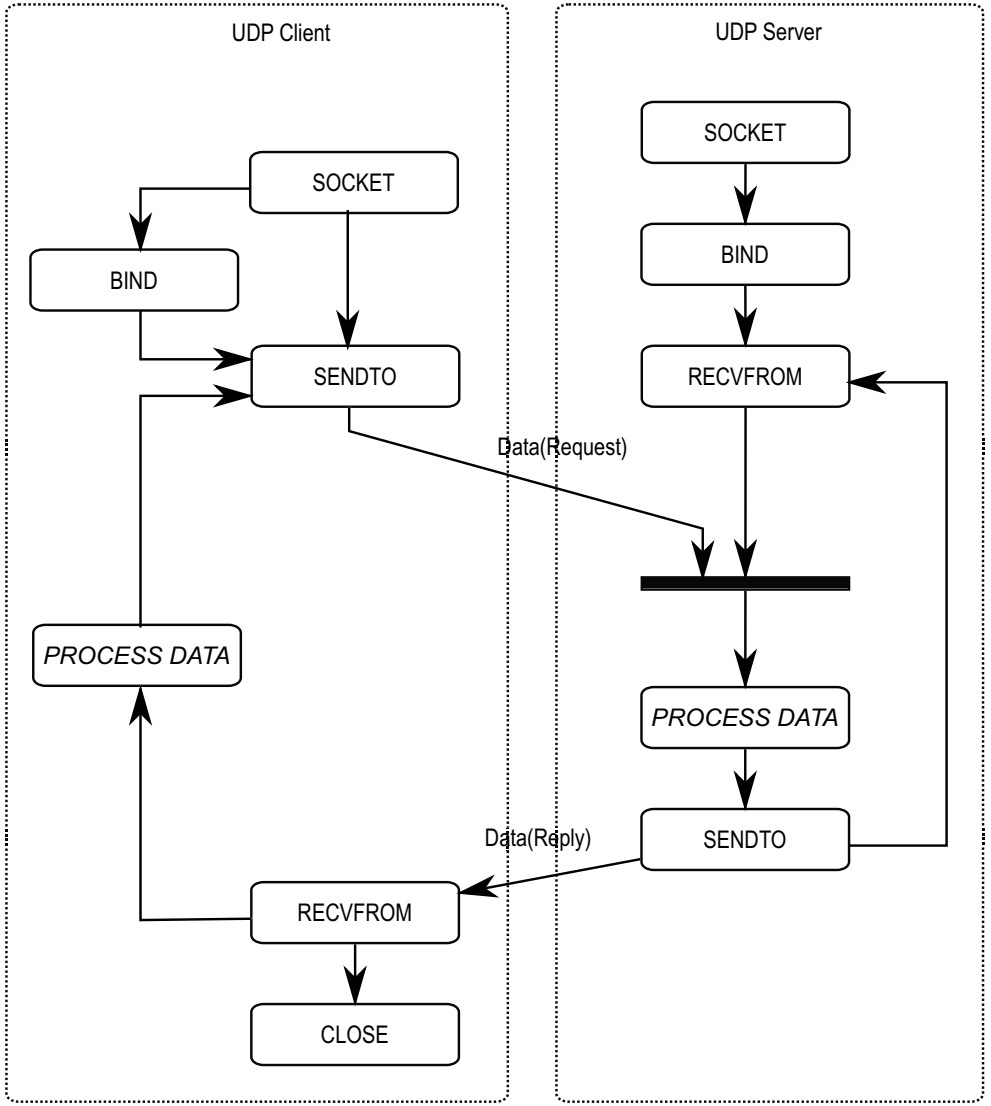


- **SEND:** Senden von Daten über die Verbindung durch den Client oder den Server.
- **RECEIVE:** Empfangen von Daten auf dem Client oder auf dem Server.
- **CLOSE:** Beenden der Verbindung.

1.7 Beschreibung der Kommunikationsprimitive für UDP-Sockets

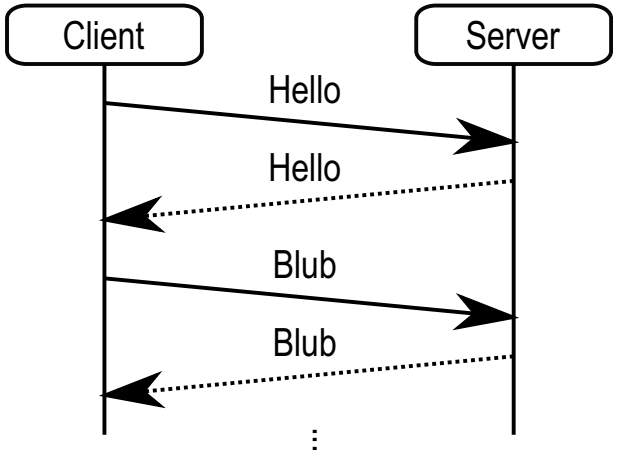
1.7.1 Socketprimitive

- **SOCKET:** Erzeugt einen neuen Kommunikationsendpunkt.
- **BIND:** Bindet einen Socket an eine lokale Netzaddresse und ggf. an einen lokalen Port.
- **SENDTO:** Senden von Daten an eine Ziel IP Addresse und einen Zielport.
- **RECVFROM:** Empfangen von Daten auf einer bestimmten IP-Addresse und einem vorgegebenen Port



2 TCP Socket Programmierung in C

Im Folgenden soll die Programmierung von Network Sockets anhand eines einfaches Echo-Servers und des dazugehörenden Clients illustriert werden.



Sowohl Client als auch Server sollen auf dem gleichen Rechner laufen, um die Demonstration zu vereinfachen und abhängig von konkreten IP-Addressen zu sein. Jeder Rechner (betriebssystemunabhängig) hat ein sogeanntes "Loopback"-Interface. Dies stellt ein virtuelles Netzinterface dar, welches die Addresse 127.0.0.1 hat.

2.1 echo Server

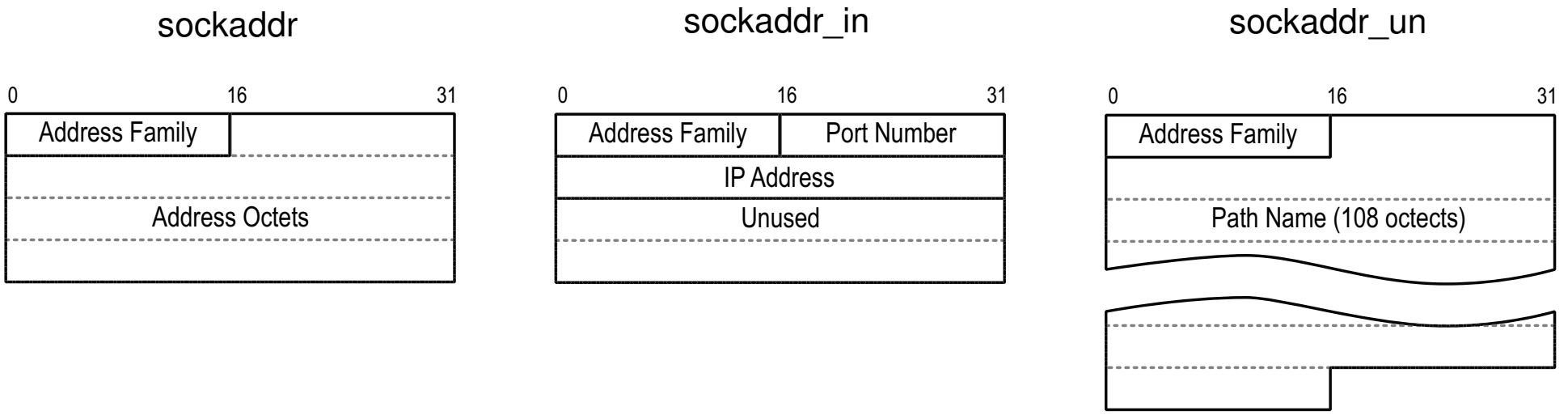
2.1.1 Deklaration der Variablen

```
int main(int argc, char *argv[])
{
  //File Descriptor, welcher den Socket wiederspiegelt
  int sockfd;
  //Eine Struktur welche die Addresse des Servers beschreiben soll
  struct sockaddr_in serverAddress;
  //Groesse der Addresse in bytes
  int serverAddressLength;

  //File Descriptor, welcher den Socket nach erfolgreichem Verbinden wiederspiegelt
  int clientfd;
  //Die folgende Struktur enthaelt die Addresse des Clients nach erfolgreicher
  //Verbinnung
  struct sockaddr_in clientAddress;
  //Groesse der ClientAddresse
  int clientAddressLength;

  // Zeiger auf Speicher zum Einlesen des Sockets
  char readBuffer[BUFFER_SIZE];

  // Begruessungsstring
```

```
char * welcomeMsg="I am your first server\n";
```

2.1.2 Repräsentation von Addressen

- Strukturen sind unterschiedlich lang
- Network-Byte Order muss beachtet werden
- Die Socket-Schnittstelle ist allgemein gehalten und verwendet Addressen vom Typ `struct sockaddr`. Je nach Anforderungen gibt es hiervon Spezialisierungen, z.B.
  - `struct sockaddr_in`: Zur Beschreibung von Addressen zur IP-basierten Kommunikation
  - `struct sockaddr_un`: Zur Beschreibung von Unix Sockets, welche über das lokale Filesystem die Kommunikation zwischen zwei Prozessen möglich machen.



Die Struktur `sockaddr_in` welche in `<socket.h>` definiert ist (vereinfachte Darstellung)

```
typedef uint16_t in_port_t;

struct sockaddr_in{
   sa_family_tsin_family    /* Address family */
   in_port_t sin_port;           /* Port number.  */
   struct in_addr sin_addr;      /* Internet address.  */
};
```

Die Struktur `in_addr` welche in `<socket.h>` definiert ist

```
typedef uint32_t in_addr_t;
struct in_addr
{
    in_addr_t s_addr;
};
```

Folgend soll die Serveraddresse mit den entsprechenden Werden belegt werden um den Echoserver auf Port 12345 des Loopback-Interfaces zu betreiben.

```
//Die Serveraddresse muss mit den entsprechenden Werten belegt werden
  serverAddress.sin_family = PF_INET;
  serverAddress.sin_addr.s_addr = inet_addr("127.0.0.1");
  //Die Portnummer muss in die network byte order gebracht werden
  serverAddress.sin_port = htons(12345);
  serverAddressLength = sizeof(serverAddress);
```

- Da wir IP als zugrundliegendes Network Layer Protokoll verwenden muss die Addressfamilie `PF_INET` sein.
- Die Funktion `in_addr_t inet_addr(const char *cp)` erlaubt die Umwandlung einer IP-Addresse welche als String angegeben wurde in den Typ `in_addr_t`.

2.1.3 socket(...) Funktion

Die `socket(...)` Funktion erzeugt einen Filedeskriptor auf welchem geschrieben und gelesen werden kann wie auf einer gewöhnlichen Datei.

```
//Erzeugen des Filedescriptors
```

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

Weitere Details hierzu siehe vorne.

2.1.4 bind(...) Funktion

Anschliend muß festgelegt werden auf welcher Addresse und auf welchem Port der Socket hören soll. Hierzu wird der Filedeskriptor des oben erzeugten Sockets sowie ein Pointer auf die Struktur, welche die lokale IP-Addresse enthält übergeben. Damit bekannt ist wie groß der Speicherbereich ist muß noch die entsprechende Größe mitübergeben werden.

```
//Binden des Sockets an die Adresse
bind(sockfd, (struct sockaddr *)&serverAddress, serverAddressLength);
```

2.1.5 listen(...) Funktion

Anschließend muss noch festgelegt werden was der angelegte und gebunde Socket machen soll. Hier gibt es zwei Möglichkeiten. Entweder wird der Socket als Clientsocket verwendet und es wird die Methode `connect(..)` aufgerufen oder der Socket wird als Serversocket verwendet und man hört (listen) auf dem diesem Port.

```
//Absicht ausdruecken auf dem entsprechenden Socket zu hoeren
//Eine Verbindung im backlog stehen.
listen(sockfd,1);
```

Mit dem Aufruf von `listen(...)` wird dem Kernel mitgeteilt, dass er eingehende Verbindungsanfragen (TCP SYN) akzeptieren soll. Die 1 drückt aus, dass maximal eine akzeptierte Verbindung im backlog (Nachholbedarf, Auftragsüberhang, ...) stehen darf, welche noch nicht abgearbeitet wurde. Hier kann eine beliebig große Zahl stehen in Abhängigkeit des zur Verfügung stehenden Speichers.

```
int listen(int sockfd, int backlog);
```

2.1.6 accept(...) Funktion

Die accept(...) Funktion ist **blockierend**, d.h. der echoServer-Prozess wird hier solange warten bis sich ein Client verbindet. Dies kann entweder schon vor Aufruf von accept passieren, d.h. eine Verbindungsanfrage steht im backlog, oder zu einem Zeitpunkt in der Zukunft.

```
//Warten bis sich ein Client verbindet
clientfd = accept(sockfd, (struct sockaddr *)&clientAddress,
                  &clientAddressLength);
```

Die `accept(...)` Funktion benötigt einen Zeiger auf eine Struktur die die Clientaddresse nach erfolgreicher Verbindung erhalten wird. Hiertmit lässt sich zum Beispiel die IP-Addresse und der Port, welche vom Client benutzt werden bestimmten. Als Rückgabe Wert liefert `accept(...)` einen Filedeskriptor, der zur Kommunikation mit dem Client verwendet werden kann.

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

- `int sockfd`: Filedeskriptor, welcher auf den Socket zeigt auf welchem aufgebaute Verbindungen bearbeitet werden sollen.
- `struct sockaddr *addr`: Struktur, welche die Address des gegenüberliegenden Endpunktes aufnehmen soll.
- `socklen_t *addrlen`: Die Größe der Addresse
- Gibt einen Filedeskriptor zurück. Im Fehlerfall wird ein Wert <0 zurück gegeben.

2.1.7 send(...) Funktion

Nach erfolgreichem Verbindungsaufbau soll zunächst eine Begrüßungsmitteilung mittels der Funktion `send(...)` an den Client versendet werden.

```
//Ausgabe der Begruessungsmessage
send(clientfd, welcomeMsg, strlen(welcomeMsg), 0);
```

Die `send(...)` Funktion ist der Funktion `write(...)`, welche zum Schreiben in Dateien verwendet wird sehr ähnlich. Sie unterscheidet sich lediglich dadurch, dass ein viertes Argument vorhanden ist, welches für den Socket spezifische Information (flags) enhalten kann. Sind die flags null, ist `send(...)` äquivalent zu `write(...)`.

```
ssize_t send(int s, const void *buf, size_t len, int flags);
```

- `int s`: Filedeskriptor
- `const void * buf`: Zeiger auf Speicherplatz, welcher die zu sendenden Daten enthält
- `size_t len`: Größe/Menge an Daten die zu senden ist
- `int flags`: Angabe von zusätzlichen Optionen die die Kommunikation beeinflussen (s. `man send` für weitere Informationen).
- Gibt die Anzahl an geschriebenen Zeichen zurück.

2.1.8 receive(...) Funktion

Für die `receive(...)` Funktion gelten die gleichen Regeln wie für `send(...)`.

```
while(1){
  int bytesRead = recv(clientfd, readBuffer, BUFFER_SIZE, 0);
  if (bytesRead > 0){
  int bytesSend = send(clientfd, readBuffer, bytesRead,0);
    if (bytesSend ==-1)
    perror("Problem during send");
  }
  else if (bytesRead == 0){
    close(clientfd);
    exit(0);
  }
  else{
    perror("Problem during recv");
    close(clientfd);
    exit(-1);
  }
}
```