Systemnahe Software – Netzwerke

Prof. Dr. Franz Schweiggert und Mitarbeiter

23. Mai 2004

Fakultät Mathematik u. Wirtschaftswissenschaften
Abteilung Angewandte Informationsverarbeitung
Vorlesungsbegleiter (gültig ab SS 2000)

Anmerkungen:

- Auf eine systematische Unterscheidung zwischen *BSD Unix* und *System V Unix (Linux, Solaris)* wird hier verzichtet.
- Die enthaltenen Beispiel-Programme wurden zum großen Teil unter Linux entwickelt und sind weitestgehend unter Solaris getestet (fur konstruktive Hinweise sind die Autoren dankbar).
- Die Beispiele sollen jeweils gewisse Aspekte verdeutlichen und erheben nicht den Anspruch von Robustheit und Zuverlässigkeit. Man kann alles anders und besser machen.
- Details zu den behandelten / verwendeten System Calls sollten jeweils im Manual bzw. den entsprechenden Header-Files nachgelesen werden.
- Dieses Skript ersetzt kein Lehrbuch und erhebt auch nicht den Anspruch eines zu sein. Die Erklärungen der Vorlesung sind zum Verständnis unverzichtbar.

Literatur: (jeweils neueste Ausgabe beachten!)

- [Bach86] M. J. Bach: The Design of the UNIX Operating System. Prentice Hall 1986
- [Comer91] D. E. Comer: Internetworking with TCP/IP. Volume I; Principles, Protocols, and Architecture. Prentice Hall 1991
- [Comer91] D. E. Comer, David L. Stevens: Internetworking with TCP/IP. Volume II; Design, Implementation and Internals. Prentice Hall 1991
- [Etter99] M. Etter: Zur Qualitätsverbesserung in der Entwicklung und Pflege verteilter Anwendungen - Konzeption und Realisierung einer Bibliothek auf der Basis von TCP/IP. Dissertation an der Universität Ulm, Abt. Angewandte Informationsverarbeitung, 1999.
- [Rochkind85] M. Rochkind: UNIX-Programmierung für Fortgeschrittene. Hanser Verlag 1985
- [Santifaller] M. Santifaller: TCP/IP and ONC/NFS. Internetworking in a UNIX Environment. Addison Wesley 1994
- [Stevens92] W. R. Stevens: Advanced Programming in the UNIX Environment. Addison-Wesley 1992
- [Stevens95] W. R. Stevens: TCP/IP Illustrated, Volume 2 - The Implementation. Addison-Wesley 1995
- [Stevens96] W.R. Stevens: TCP/IP Illustrated, Volume 3 - TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols. Addison-Wesley 1998
- [Stevens98] W.R. Stevens: UNIX Network Programming, Volume 1 - Networking APIs: Sockets and XTI. Prentice Hall 1998.
- [Stevens99] W.R. Stevens: UNIX Network Programming, Volume 2 - Interprocess Communications. Prentice Hall 1999.
- [Tanenbaum87] A. S. Tanenbaum: Operating Systems - Design and Implementation. Prentice Hall 1987

Inhaltsverzeichnis

Kapitel 1

Netzwerk-Kommunikation

1.1 Übersicht

- räumlich verteiltes System von Rechnern, Steuereinheiten und Peripheriegeräten, verbunden mit Datenübertragungseinrichtungen
- aktive / passive Komponenten

Ausbreitung

- globales Netz (**G**AN *g*lobal area network)
  Internet, EUNet, VNET (IBM), u.a.
- Weitverkehrsnetz (**W**AN *w*ide area network)
  DATEV-Netz, DFN (Deutsches Forschungsnetz)
- lokale Netze (**L**AN *l*ocal area network)
  innerhalb eines Unternehmens; i.a. mehrere, die selbst wieder vernetzt sind
  ☞ **Front-end-Lan** (Netz innerhalb einer Abteilung / Instituts)
  ☞ **Backbone-LAN** (Verbindung von Front-end-LAN's

Netzwerktopologie

- physikalische / logische Verbindung der Rechner im Netz (Stern, Ring, Bus)

Protokolle

- Regeln (Vereinbarungen), nach denen Kommunikationspartner (Rechner) eine Verbindung aufbauen, die Kommunikation durchführen und die Verbindung wieder abbauen

Grundlegendes herstellerunabhängiges Konzept:

DIN/ISO-OSI-Referenzmodell

- **O**SI - *Open Systems Interconnection*

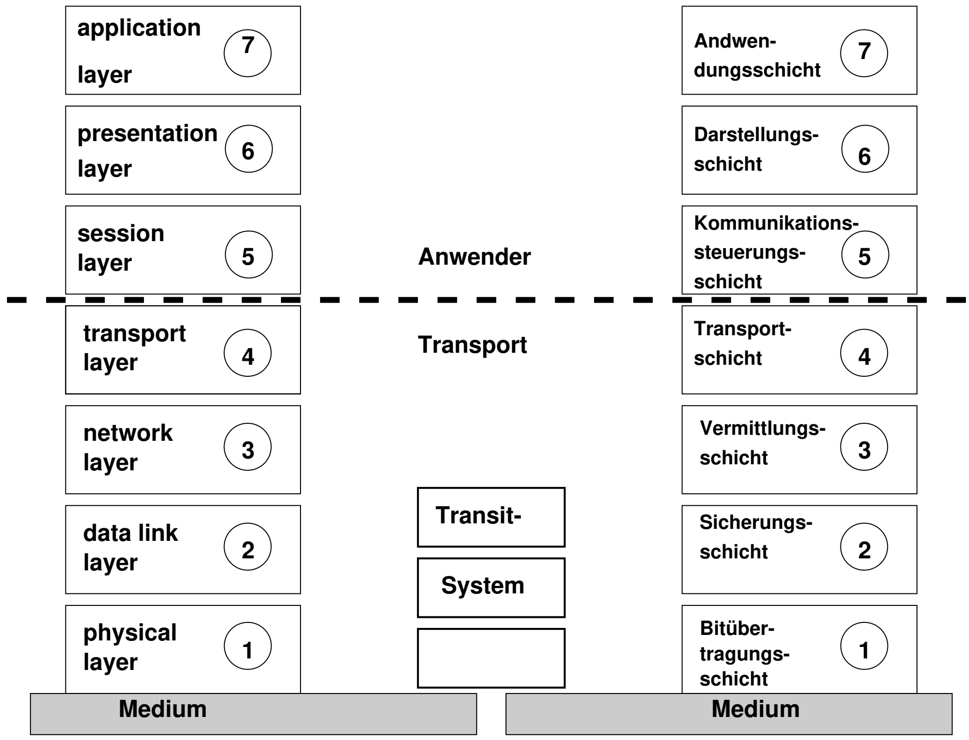

Abbildung 1.1: ISO-OSI-Referenzmodell

1. **B**itübertragungsschicht
   Regelung aller physikalisch-technischer Eigenschaften der Übertragungsmedien zwischen den verschiedenen End-/Transitsystemen
   Darstellung von Bits via Spannungen, Stecker, ...

2. **S**icherungsschicht
   Sicherung der Schicht **1** gegen auf den Übertragungsstrecken auftretenden Übertragungsfehler (elektromagnetische Einflüsse)
   z.B. Prüfziffern, *parity bits*

3. **V**ermittlungsschicht
   Adressierung der Zielssysteme über das (die) Transitsystem(e) hinweg sowie Wegsteuerung der Nachrichten durch das Netz
   Flußkontrolle zwischen End- und Transitsystemen (Überlastung von Übertragungswegen und Rechnern / Transitsystemen, faire Verteilung der Bandbreite)

4. **T**ransportschicht
   Stellt die mithilfe der Schichten **1 2 3** hergestellten Endsystemverbindungen für die Anwender zur Verfügung
   z.B. Abbildung logischer Rechnernamen auf Netzadressen

5. **K**ommunikationssteuerungsschicht
   Bereitsstellung von Sprachmitteln zur Steuerung der Kommunikationsbeziehung (*session*)
   Aufbau, Wiederaufnahme nach Unterbrechung, Abbau

6. **D**atendarstellungsschicht
   Vereinbarungen bzgl. Datenstrukturen für Datentransfer

7. **A**nwendungsschicht
   Berücksichtigung inhaltsbezogener Aspekte (Semantik)

Quasistandard

- **T**CP/IP
  Transmission Control Protocol / Internet Protocol

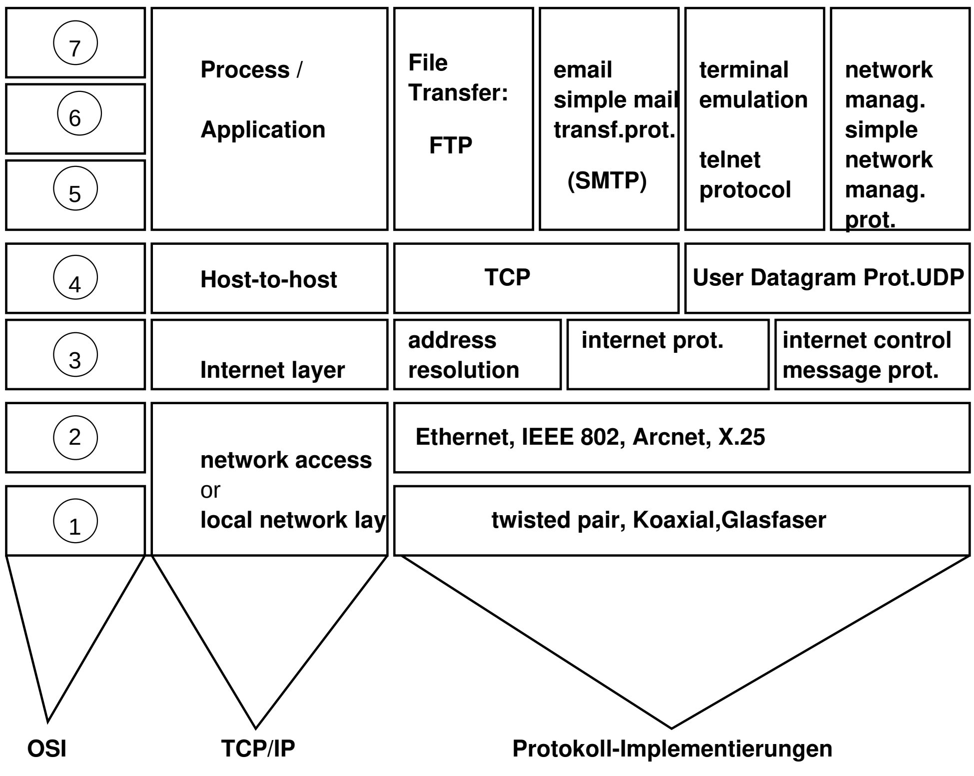

Abbildung 1.2: TCP/IP

Kopplung von Netzen

- Kopplungseinheiten zur Verbindung von Netzen (*internetworking units*) Adreßumwandlung, Wegewahl (*routing*), Flußkontrolle, Fragmentierung und Wiederzusammenfügung von Datenpaketen, Zugangskontrolle, Netzwerkmanagement
  - **Repeater**
    Verstärker; Empfangen, Verstärken, Weitersenden der Signale auf Bitübertragungsschicht; die zu verbindenden Netze müssen identisch sein; Verbindung von Netzsegmenten
  - **Bridge**
    Verbindung von Netzen mit unterschiedlichen ÜBertragungsmedien, aber mit gleichem Schichtaufbau; operiert auf Sicherungsschicht
  - **Router**
    operiert auf Vermittlungsschicht
  - **Gateway**
    Verknüpfung von Netzen, die in Schicht 3 (und aufwärts) unterschiedliche Struktur aufweisen
  - **Hub**
    Eine Art Multiplexer, der das Eingangssignal sternförmig an die angeschlossenen Geräte weiterleitet (keine eigene Adresse); reduziert den Verkabelungsaufwand

1.2 Lokale Netze

Zugangsverfahren

→ wer darf wann senden ?

a) über strenge Vorschrift wird festgelegt, wer wann senden darf

b) jeder sendet wie er will, bis Fehler auftritt, der dann korrigiert wird

ad a) **T**oken-Verfahren
<u>ein</u> besonderes Bitmuster (*Token*) ”kursiert” im Netz
senden darf der, der es besitzt
das Token wird an das Ende der Sendung angefügt

ad b) **C**SMA/CD-Verfahren
carrier sense multiple access with collision detection

1. viele beteiligte Sender (*multiple access*)
2. vor dem Senden in den Kanal ”horchen” (*carrier sense*)
   wenn frei, senden, sonst warten
3. während des Sendens den Kanal prüfen, ob andere senden, um Kollissionen zu erkennen (*collision detection*)
   wenn Kollision, müssen alle Sender abbrechen; jeder wartet eine zufällig gewählte Zeitspanne und wiederholt Sendevorgang - Sender mit der kürzesten Zeitspanne ”gewinnt”

1.3 LAN-Standards

IEEE 802.3 CSMA/CD - Bussystem (ISO 8802-2)
→ ETHERNET
relativ kostengünstig, weit verbreitet;
ursprünglich war *Ethernet* als Standard Anfang der 80er Jahre von Xerox, DEC und Intel erarbeitet worden; ISO-8802-3 ist eine Erweiterung hiervon
ursprüngliche Version: Übertragungsrate bis 10MBits/s, inzwischen sind 1GBit/s realisiert und 10GBit/s geplant

Kabeltypen:

- "Thick Ethernet" (Yellow Cable)
- "Thin Ethernet" (RG-58)
- Koaxialkabel
- verdrillte Kupferkabel (*twisted pair*)

IEEE 802.5 Tokenring mit Token-Zugangsverfahren (ISO 8802-5)

- verdrillte Kupferkabel (1-4MBits/s)
- Koaxialkabel (4-40MBits/s)

ISO-8802-4 Tokenbus

logischer Ring (Token-Zugang) auf physikalischem Bus
Anwendung: MAP-Protokolle (*manufacturing automation protocol*), zur Vernetzung von Robotern, CNC-Maschinen

ISO-8802-8 FDDI *fiber distributed data interface*
für Nahverkehrsnetze mit hoher Übertragungsgeschwindigkeit (100 MBits/s), meist als *Backbone* eingesetzt

IEEE-802.11 WLAN *wireless lan*
drahtlose Datenübertragung via Funkverbindung; Reichweite: ca. 30-50m (je nach Umgebung); Übertragungsrate: 11MBit/s (durch Kanalbündelung auch 22MBit/s); Sterntopologie

1.4 Ethernet

(LAN)

- ca. 1970 XEROX PARC
- Standardisiert 1978 von XEROX, INTEL, DEC
- Kabel: koaxial
  - ☞ jeweils am Ende des Kabels ein Widerstand zur Vermeidung von Reflexionen elektrischer Signale
  - ☞ Kabel rein passiv
  - Anbindung an Ethernet → "T"-Stück

- elektronische Komponenten:
  - **T**ransceiver (Übertragen/Empfangen in/von Ethernet)
  - **h**ost interface (Rechner-Bus)

☐ Eigenschaften:

- alle teilen sich einen Kanal (BUS)
- **b**roadcast - alle Transceiver hören alles, host interface stellt fest, ob Nachricht für diesen Rechner gedacht ist
- **b**est-effort delivery - "Bemühensklausel": ob Sendung wohl ankommt?
- CSMA/CD-Zugangsverfahren

☐ Varianten:

- twisted pair ethernet → "Telefondrähte"
- thin-wire ethernet → "Kabelfernsehen"
  geringere Leistung, geringerer Preis, auch für interface
- Breitbandtechnik → multiplexing (weniger Kabel)

☐ Adressierung
host interface bildet Filter, alle Pakete werden dahin weitergeleitet, nur die dem Transceiver entsprechenden Pakete (Hardware-Adressen) werden an Rechner weitergeleitet
Adresse eines Rechners: 48 Bit Integer, vom Hersteller auf Interface festgelegt, von IEEE gemanaged

- Adreßtypen:
  - physische Adresse <u>einer</u> Schnittstelle
  - network broadcast address (alle Bits auf 1, "an alle")
  - multicast broadcast (Teilmengen broadcast)
    BS initialisiert Schnittstelle: welche Adressen sollen erkannt werden?

Frame Format:

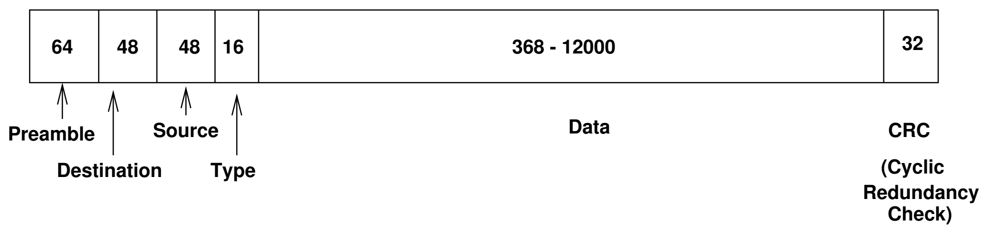

Abbildung 1.3: Ethernet Frame Format

- Preamble: zur Synchronisation der Knoten, alternierende 0/1-Folge
- Type: welches Protokoll (für BS)? → self-identifying

1.5 Internetworking - Concept & Architectural Model

- bislang:
  ein Netz (ein physikalisches Netz) - z.B. Ethernet, Token Ring, Adressen von Hosts waren physikalische Adressen

- jetzt:
  "Netz über Netzen über Netzen über ... über physikalischen Netzen

  ☞ notwendig:
  Abstraktion von zugrundeliegenden physikalischen Netzen
  Ansatz 1: spezielle (Applikations-) Programme, die aus der Heterogenität der physikalischen Netze / Hardware eine softwaremäßige Homogenität herstellen
  Ansatz 2: Verbergen von Details im Betriebssystem des jeweiligen Rechners, layered-system Architecture

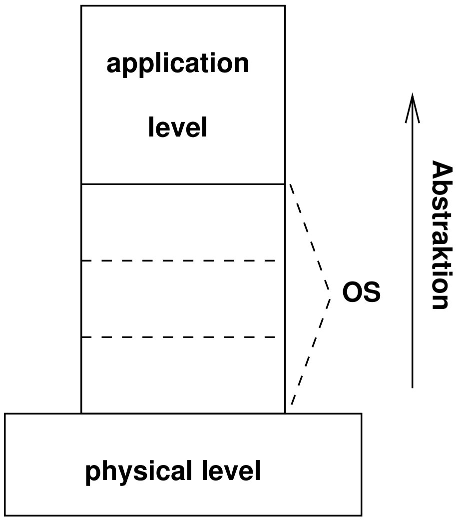

Abbildung 1.4: Layered System Architecture

Internet Architecture

☐ *In a TCP/IP internet, computers called gateways provide all interconnections among physical networks*

Frage: Muß ein Gateway alle in allen Netzen erreichbare Rechner kennen (ein Super-Computer)?

Antwort: Nein, *gateways route packets based on destination network, not on destination host* (alle Netze müssen i.P. bekannt sein, → lernende Gateways, → Mini-Computer)

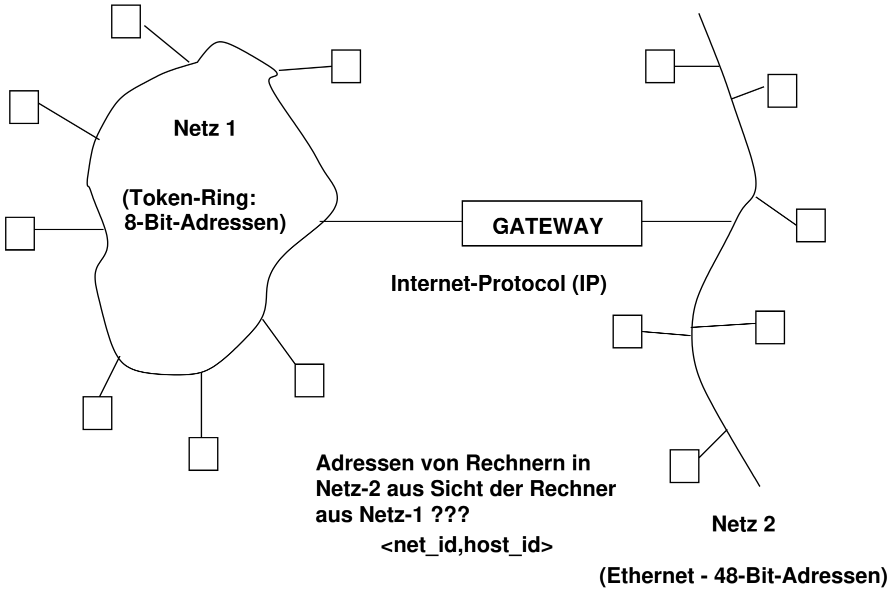

Abbildung 1.5: Internet Architecture

Benutzer-Sicht:
ein Internet als <u>ein</u> großes, virtuelles Netzwerk
☐ *The* **T**CP/IP *internet protocols treat all networks equally (independant of their delay and throughput characteristics, maximum packet size or geographic scale). A local area network like Ethernet, a wide area network like NSFNET backbone, or a point-to-point link between two machines each count as one network*

Internet-Adressen

- globale Identifikation von hosts im (in ihrem) Netz

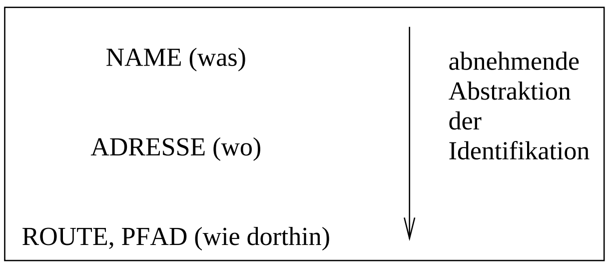

Abbildung 1.6: Abstraktion der Identifikation

- **I**P-Adresse (logische Adresse):
  Integer, die die Route unterstützen (32-Bit): **(netId,hostId)**

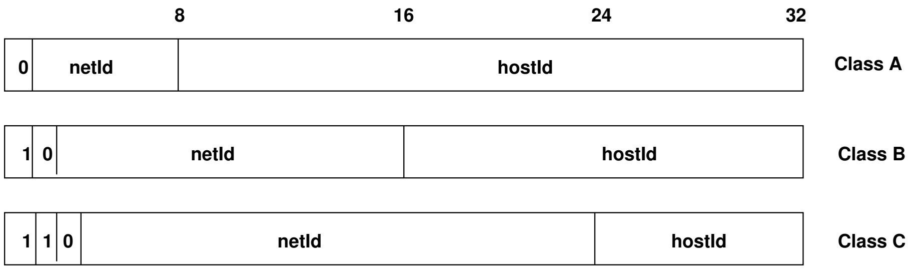

Abbildung 1.7: 32-Bit IP-Adresse

A: wenige Netze mit sehr vielen Hosts (mehr als 2^{16} Hosts)

B: mehr Netze mit ziemlich vielen Hosts (zwischen 2^8 und 2^{16} Hosts)

C: viele Netze mit wenigen Hosts (weniger als 2^8 Hosts)

NB: IP-Adresse spezifiziert die Verbindung eines Hosts zu einem Netz!
NB: HostId == 0: spezifiziert Netz selbst
NB: HostId == 1: Broadcast an alle Host's des Netzes (falls dieses Netz Broadcast kennt)

lesbare Adressen (punktiertes Dezimalformat): (*d*otted decimal notation)

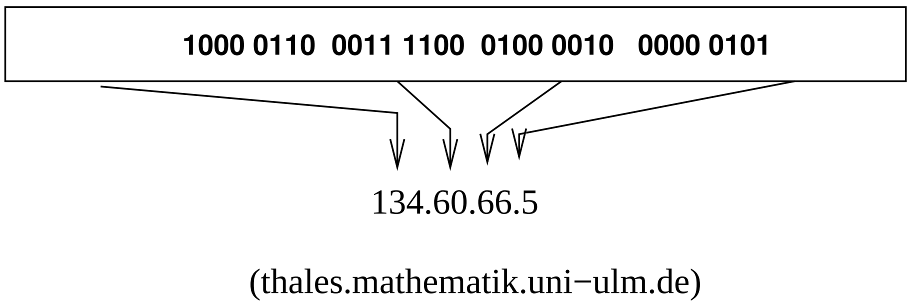

Abbildung 1.8: IP-Adresse: dotted decimal form

Adressenvergabe:

- NIC (*Network Information Center*) vergibt NetId
- Verantwortung für HostId delegiert an jeweilige Organisation
  - Problem: lowest-Byte (Bit) $\leftrightarrow$ highest-Byte (Bit)

  Protocol network standard **b**yte order : most significant byte send first!

Internet Adresse $\rightarrow$ physikalische Adresse: **ARP** - **A**ddress Resolution Protocol
IP-Adresse als ”physikalische” Adresse in einem virtuellen Netzwerk

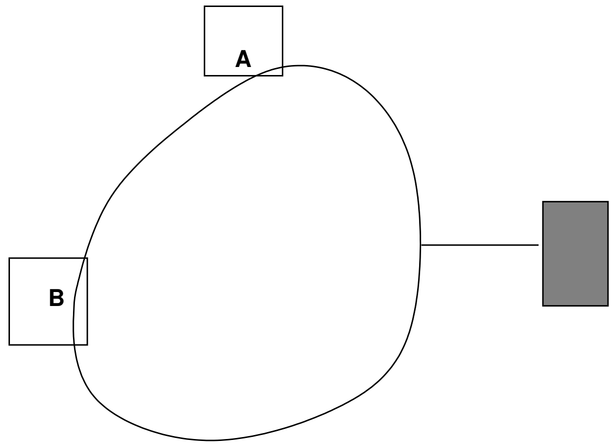

Abbildung 1.9: Adress-Auflösung

I_A, I_B Internet-Adressen (netId, hostId)

P_A, P_B physikalische Adressen (z.B. Ethernet)

Address resolution problem:

$I_A \rightarrow P_A$??? (low level software)

Ethernet	48 Bit
proNet	8 Bit
IP-hostId	8 .. 24 Bit

- proNet: $P_{Adr} \subseteq I_{Adr}$ (kein Problem)
- X.25: Tabelle $\subseteq I_{Adr} \times P_{Adr}$ + hashing (Zuordnungstabelle)
- Bei Ethernet:

  A will an B senden, kennt aber nur I_B und nicht P_B

  1. A sendet via *broadcast* an alle mit I_B, B kennt (hoffentlich) seine eigene Internet-Adresse

→ **broadcasting ist sehr teuer**, insbesondere auch weil ein IP-Paket u.U. in sehr viele Pakete auf dem Ethernet zerlegt wird (später: Fragmentation)

2. A sendet *broadcast*: "Rechner mit Internet-Adresse I_B möge sich melden mit P_B"
   B meldet sich mit (I_B,P_B) bei A

   A merkt sich für die Zukunft (I_B,P_B)

TCP/IP

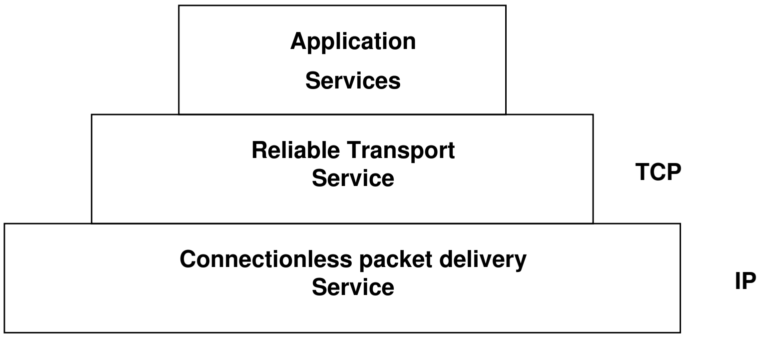

Abbildung 1.10: TCP/IP-Schichtenmodell

1. Concept of **U**nreliable Delivery
   Auslieferung von Paketen ist nicht garantiert (*may be lost, duplicated, delayed, delivered out of order → service will not detect nor inform sender or receiver*)
2. **C**onnectionless
   Jedes Paket wird losgelöst von den anderen behandelt (evt. auch anders gerouted)
3. **b**est-effort delivery
   "bemühe mich um Auslieferung"

IP Internet Protocol:

- Regeln, die den *unreliable, connectionless, best-effort delivery* - Mechanismus definieren
  1. *basic unit of data transfer* durch ein TCP/IP-Internet
  2. enthält routing-Funktion (Auswahl des Pfades)
  3. Regeln, wie Host's und Gateway's Pakete verarbeiten sollen, wie und wann Fehlermeldungen produziert werden sollen, Bedingungen für das "Wegwerfen" von Paketen

Internet Datagram - ***basic transfer unit***

- grober Aufbau (Abb. 1.11, S. **??**):

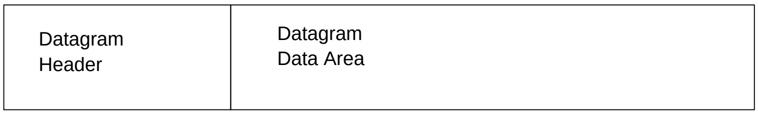

Abbildung 1.11: IP Datagramm - grober Aufbau

- genauer ist dies in Abb. 1.12, S. 12 dargestellt

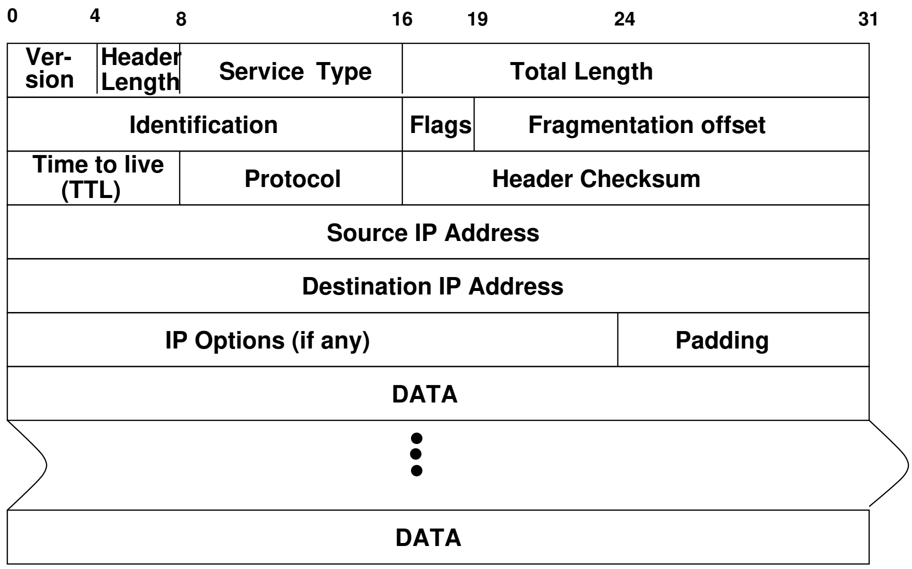

Abbildung 1.12: IP Datagram - im Detail

TCP/IP Internet Layering Model

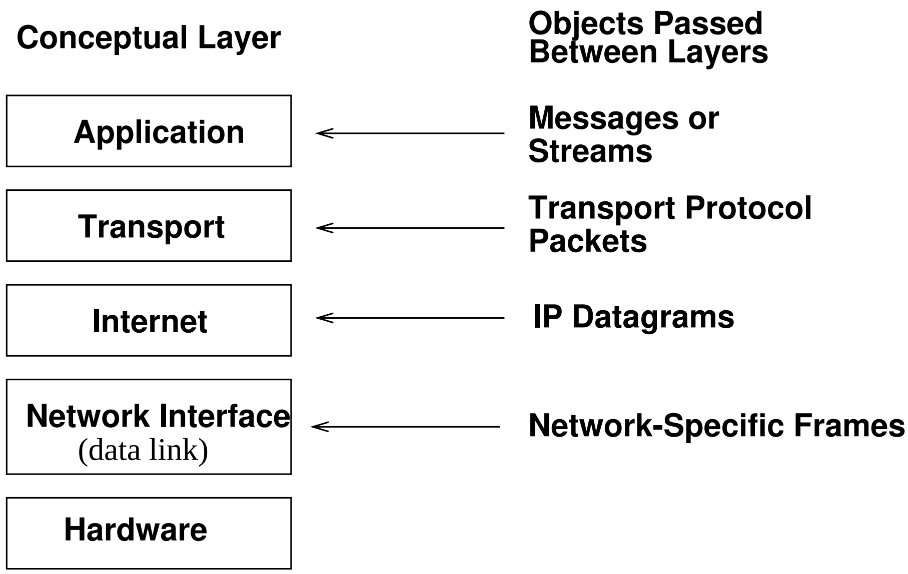

Abbildung 1.13: TCP/IP Layering Model

Application Layer

Auf der obersten Ebene starten Benutzer Anwendungsprogramme, die auf im Netz verfügbare Dienste zugreifen wollen. Ein Anwendungsprogramm interagiert mit dem / den Transport-Protokoll(en), um Daten zu senden / empfangen.
Jedes Anwendungsprogramm wählt die benötigte Transport-Art, z.B. eine Folge individueller Botschaften (*messages*) oder einfach eine Folge von Bytes. Das Anwendungsprogramm übergibt diese Daten in der verlangten Form an die Transport-Ebene zur Auslieferung.

Transport Layer

Die Hauptaufgabe dieser Schicht besteht darin, die Verbindung zur Kommunikation zwischen zwei Anwendungsprogrammen bereitzustellen (*end-to-end-communication*).

- Regulieren des Informationsflusses
- Bereitsstellen eines zuverlässigen Transports
- Sicherstellen, dass Daten korrekt und in Folge ankommen
- Warten auf Empfangsbestätigung des Empfängers
- erneutes Senden verlorengegangener Pakete

- Aufteilen des Datenstroms in kleine Stücke (*packets*)
- Übergeben jeden Paketes mit Zieladresse für die nächste Schicht

i.a. arbeiten viele Applikation mit der Transport-Schicht (Senden, Empfangen)

- jedem Paket wird zusätzliche Information beigefügt (welche Appl.?)
- Prüfsumme

Internet Layer

- Erstellen der IP Datagrams
- (weiter-) senden der Datagrams
- "auspacken" der Datagrams (auf Zielmaschine)
- Übergabe an das richtige Transport-Protokoll (Zielmaschine)
- ICMP

Network Interface Layer

- übernimmt Datagrams und schickt sie auf spezifischem Netz weiter

1.6 Transport-Protokolle

1.6.1 Ports

- Internet Protokolle adressieren *Host's*
- Adressierung von Applikationen (letztliches Ziel) innerhalb des Zielrechners?

Unterstellt seien *multiprocess*-Systeme als Zielrechner (z.B. UNIX-Rechner).

- Prozess als letztliches Ziel?
  Prozesse werden dynamisch erzeugt und terminiert, Sender kann Prozesse auf anderen Maschinen nicht identifizieren
  Dienste (sprich Funktionen, Leistungen) auf anderen Rechnern sind das Ziel, unabhängig von welchem Prozess (welchen Prozessen) diese realisiert sind!
- Jede Maschine hat eine Menge sog. Protokoll- **P**ort's (abstrakter Endpunkt), identifiziert durch positive *Integer*. Das jeweilige Betriebssystem bietet Mechanismus an, mit dem Prozesse Ports spezifizieren und nutzen können.
  Die meisten Betriebssysteme unterstützen synchronen Zugriff auf Ports. Wenn dabei eine Prozess Daten von einem Port lesen will, noch keine Daten da sind, wird er solange blockiert, bis (genügend) Daten eingetroffen sind.
  Ports sind typischerweise gepuffert.
- Ziel-Adresse: (IP Adresse + Port-Nummer)
  Jede Meldung enthält neben *destination port* auch *source port* (z.B. zum Antworten).

1.6.2 UDP

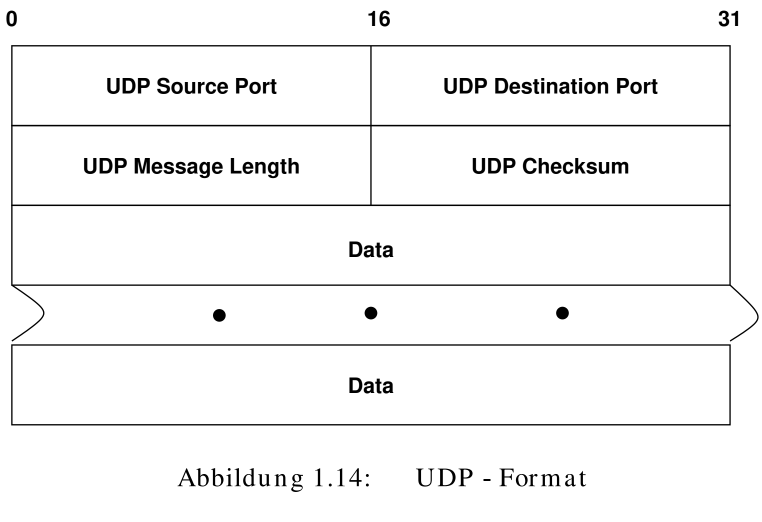

Abbildung 1.14: UDP - Format

☐ *The User Datagram Protocol (UDP) provides unreliable connectionless delivery service using IP to transport messages between machines. It adds ability to distinguish among multiple destinantions within a given host computer*

- keine Empfangsbestätigungen (*Acknowledgement*
- Kein Ordnen ankommender Meldungen
- keinerlei Feedback
  - *UDP messages* können verloren gehen, dupliziert werden, ungeordnet ankommen, schneller ankommen als verarbeitet werden!
  - Anwendungsprogramme, die auf UDP aufbauen, müssen selbst für "Zuverlässigkeit" sorgen.
- *Source Port* ist optional, wenn nicht genutzt, so NULL
- *LENGTH* Anzahl Bytes (*octets*) im UDP Datagram inkl. Header.
- *CECKSUM* ebenfalls optional; wenn keine Prüfsumme berechnet, so NULL.
  NB: IP berechnet keiner Prüfsumme über den Datenteil!
  Berechnung der Prüfsumme (wie bei IP): Daten werden in 16-Bits-Einheiten aufgeteilt, Summe über 1-er-Komplement wird gebildet, davon wieder 1-er-Komplement gebildet; ist Prüfsumme identisch Null, so wird davon das 1-er-Komplement gebildet (alles auf 1); unproblematisch, da es bei 1-er-Komplement zwei Darstellungen des Zahl Null gibt: alle Bits auf 0 oder alle auf 1!

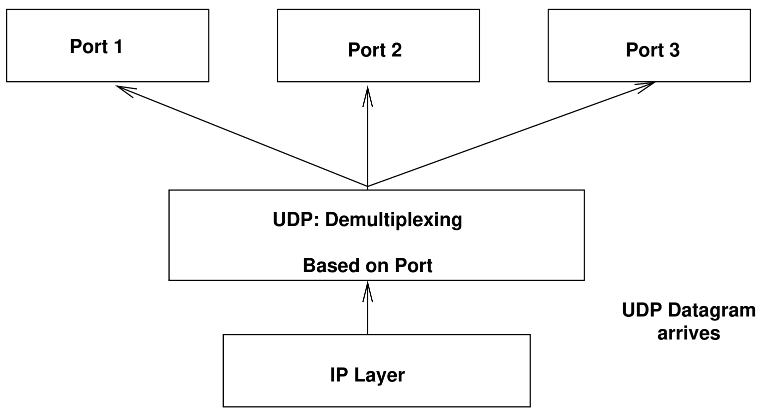

Abbildung 1.15: UDP-Demultiplexing

Port-Nummern:

- einige zentral vergeben (*well-known port numbers, universal assignment*)
- *dynamic binding approach*
  Port ist nicht global bekannt; wenn ein Programm einen Port benötigt, wird von der Netzwerk-Software einer bereitgestellt. Um eine Port-Nummer auf einem anderen Rechner zu erfahren, wird eine entsprechende Anfrage gestellt und beantwortet.

Decimal	Keyword	UNIX Keyword	Description
0	-	-	Reserved
7	ECHO	echo	Echo
9	DISCARD	discard	Discard
11	USERS	systat	Active Users
13	DAYTIME	daytime	Daytime
37	TIME	time	Time
42	NAMESERVER	name	Host Name Server
43	NICNAME	whois	Who Is?
53	DOMAIN	nameserver	Domain Name Server
67	BOOTPS	bootps	Bootstrap Protocol Server
68	BOOTPC	bootpc	Bootstrap Protocol Client
69	TFTP	tftp	Trivial File Transfer
111	SUNRPC	sunrpc	Sun Microsystems RPC
123	NTP	ntp	Network Time Protocol
161	-	snmp	SNMP net protocol
.			
.			
.			

Beispiel einer TCP/IP Sitzung an Port 80 (Webserver):

```
thales$ telnet www.mathematik.uni-ulm.de 80
Trying 134.60.166.6...
Connected to www.mathematik.uni-ulm.de.
Escape character is '^]'.
GET /sai/swg/ HTTP/1.0 # <-- Eingabe
  # <-- noch ein <return>, dann folgt Ausgabe des Servers
HTTP/1.1 200 OK
Date: Tue, 18 May 2004 09:05:22 GMT
Server: Apache/1.3.27 (Unix)
Last-Modified: Tue, 18 May 2004 09:04:47 GMT
ETag: "b40a3-10fe-40a9d1af"
Accept-Ranges: bytes
Content-Length: 4350
Connection: close
Content-Type: text/html

<HTML>
<TITLE>Prof. Dr. Franz Schweiggert</TITLE>
<BODY>
<H1>Prof. Dr. Franz Schweiggert</H1>

....

<ADDRESS><A HREF="/sai/swg/">Franz Schweiggert</A>, 15. Mai 2004
</ADDRESS>

</BODY>
</HTML>
Connection to www.mathematik.uni-ulm.de closed by foreign host.
thales$
```

Kapitel 2

Berkeley Sockets

2.1 Grundlagen

- **API** (*application program interface*) zu Kommunikations-Protokollen
- In UNIX: *Berkeley sockets* und *System V Transport Layer Interface (TLI)*
- Beide Schnittstellen wurden für C entwickelt
- Die Implementierungen der Internet-Protokolle und der Socket-Schnittstelle wurden erstmals 1982 in der Version 4.1c der *Berkeley Software Distributions* (**BSD**) an der Universität von Kalifornien in Berkeley integriert.
- Mit der Version 4.2BSD war 1983 das erste Unix-System mit Netzwerkunterstützung verfügbar.

Die beiden zentralen Abstraktionen der Socket-Schnittstelle sind der *Socket* und die **Kommunikationsdomäne**. Aus Sicht der Anwendung repräsentiert ein Socket einen **Kommunikationsendpunkt** eines Benutzerprozesses innerhalb der gewählten Kommunikationsdomäne. Die Domäne spezifiziert die Protokollfamilie und definiert allgemeine Eigenschaften und Vereinbarungen wie z.B. die Adressierung des Kommunikationsendpunktes.

Durch die Abstraktion Socket und Kommunikationsdomäne wird eine Trennung zwischen den Funktionen der Socket-Schnittstelle und den im System implementierten Kommunikationsprotokollen erreicht. Dem Anwender steht somit eine einheitliche Schnittstelle zur Verfügung, die verschiedene Netzwerkprotokolle und lokale Interprozesskommuniktaionsprotokolle gleichermaßen unterstützt. Die Implementierung ist vollständig im Betriebssystem integriert.

- Der Zugriff einer Anwendung auf die Funktionen der Socketschicht (*socket layer functions*) im Betriebssystemkern erfolgt über die Systemaufrufe der Socket-Schnittstelle. Hier findet die Umsetzung der protokollunabhängigen Operationen in die protokollspezifischen Implementierungen der darunterliegenden Protokollschicht (*protocol layer*) statt.
- Die Auswahl des konkreten Kommunikationsprotokolls wird bei der Erzeugung eines Socket über die Spezifikation der Domäne festgelegt.
- In der Protokollschicht (*protocol layer*) sind die Implementierungen der vom System unterstützen Protokollfamilien zusammengefaßt.

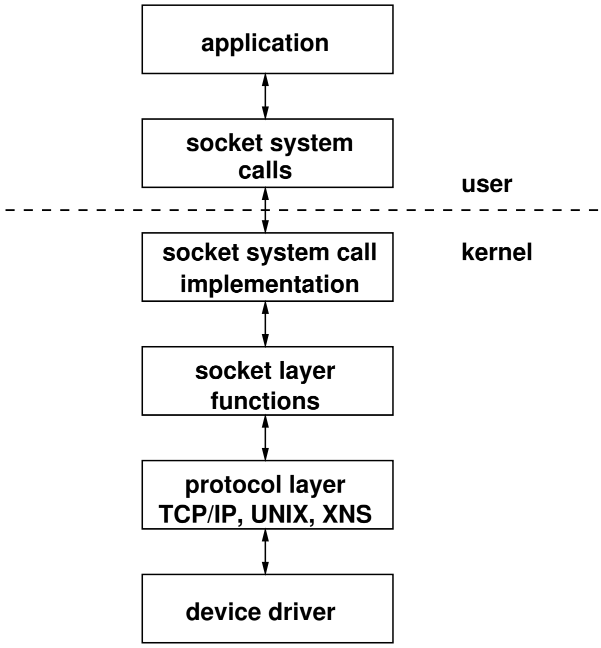

Abbildung 2.1: Vereinf. Modell der Implementierung von Sockets unter BSD

- Die darunterliegende Datenübertragungsschicht enthält die Implementierungen der Gerätetreiber (*device driver*) zur Datenübertragung über verschiedene Medien oder auch systeminterne Mechanismen.

2.2 Ein erstes Beispiel: Timeserver an Port 11011

```
#include <netdb.h>
#include <netinet/in.h>
#include <stdio.h>
#include <strings.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <unistd.h>
#include <stdlib.h>
#define TPORT   11011
int main()
{  struct sockaddr_in addr, client_addr;
   size_t client_add_len = sizeof(client_addr);
   int sfd, fd; int optval = 1;
   bzero(&addr, sizeof(addr));   // mit 0en fuellen
   addr.sin_family = AF_INET;   // TCP/IP-Verbindung
   addr.sin_port = htons(TPORT);// Port eintragen, Network Byte Order
   if ((sfd = socket(AF_INET, SOCK_STREAM, 0)) <0) // socket erzeugen
       perror("socket"),  exit(1);
   if (setsockopt(sfd,SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval))<0)
       perror("setsockopt"),   // rasche Wiederzuweisung ermoeglichen
      exit(1);
   if (bind(sfd, (struct sockaddr *) &addr, sizeof(addr)) <0)
       perror("bind"), exit(1);
   if (listen(sfd, SOMAXCONN) <0)
      perror("listen"), exit(1);
   while ((fd = accept(sfd, (struct sockaddr *) &client_addr,
                    &client_add_len)) >=0) {
      time_t clock; char *tbuf;
      time(&clock);      // aktuelle Uhrzeit holen
      tbuf = ctime(&clock);   // in String schreiben
      write(fd, tbuf, strlen(tbuf));   // in Clientverbindung schreiben
      close(fd);
   }
   exit(0);         // seldomly reached
}
```

Übersetzung und Ausführung:

```
thales$ make -f make_timserv
gcc -o timserv -Wall timserv.c  -lsocket -lnsl
thales$ timserv &
[1] 28753
thales$ telnet thales 11011
Trying 134.60.66.5...
Connected to thales.
Escape character is '^]'.
Tue May 18 11:36:34 2004
Connection to thales closed by foreign host.
thales$ exit
```

2.3 Die Socket-Abstraktion

- Der Socket repräsentiert als zentrale Abstraktion einen Kommunikationsendpunkt eines Benutzerprozesses innerhalb der gewählten Kommunikationsdomäne.

- Letztere wiederum ist eine weitere Abstraktion, die allgemeine Kommunikationseigenschaften der vom System bereitgestellten Mechanismen für die Prozesskommunikation mit Sockets zusammenfaßt. Dazu zählen beispielsweise der zu verwendende Namensraum und die damit verbundene Art der Adreßspezifikation bei der Benennung eines konkreten Socket.

- Die Kommunikation zwischen je zwei Sockets verläuft normalerweise immer innerhalb derselben Domäne.

Die in der Regel von allen Unix-Systemen unterstützten und heute hauptsächlich verwendeten Kommunikationsdomänen sind die **Unix-Domäne** für die Prozesskommunikation auf dem lokalen System und die **Internet-Domäne** für die Prozesskommunikation nach den Internet-Standard-Protokollen. Die innerhalb der Internet-Domäne miteinander kommunizierenden Prozesse können sehr wohl auch auf demselben lokalen System laufen; hierzu bietet jedoch die Verwendung der Unix-Domäne entscheidende Vorteile durch erweiterte protokoll-spezifische Eigenschaften und durch bessere Performance.

Socket-Typen:

- In der Socket-Abstraktion entsprechen die Socket-Typen den für die Anwendung jeweils sichtbaren Kommunikationsverfahren (**verbindungslos - verbindungsorientiert**).

- Zusätzlich definiert ein Socket-Typ spezielle Eigenschaften eines Protokolls wie z.B. fehlerfreie Datenübertragung, Flusskontrolle, Einhaltung von Datensatzgrenzen).

- Letztlich wird über den Socket-Typ die **Kommunikationssemantik** definiert.

- Wie bei den Domänen gilt: Prozesskommunikation nur zwischen Sockets vom selben Typ!

Die wichtigsten Socket-Typen unter Unix:

STREAM: Ein **Stream Socket** stellt ein Kommunikationsverfahren bereit, das einen bidirektionalen, kontrollierten und verlässlichen Datenfluss garantiert, d.h. alle transferierten Datenpakete kommen in derselben Reihenfolge vollständig und ohne Duplikate beim Empfänger an (vgl. in der Semantik zu bidirektionale Unix-Pipes). In BSD-Unix sind Pipes auf diese Weise implementiert.

DGRAM: Ein **Datagram Socket** stellt ebenfalls ein bidirektionales Kommunikationsverfahren bereit, aber ohne Flusskontrolle und ohne Garantie, dass gesendete Pakete den Empfänger erreichen, dass die Reihenfolge erhalten beleibt oder dass Duplikate ankommen. Sie sind zudem im Datenvolumen beschränkt. Datensatzgrenzen bleiben beim Datentransfer allerdings erhalten.

RAW: Ein **Raw Socket** stellt eine allgemeine Schnittstelle zu den meist der Transportschicht zugrundeliegenden Kommunikationsprotokollen bereit, welche die Socket-Abstraktion unterstützen. Dieser Socket-Typ ist normalerweise Datagram-orientiert, obwohl die exakte Charakteristik von der Kommunikationssemantik des konkreten Protokolls abhängt. In der Internet-Protokollfamilie kann mit Raw Sockets beispielsweise direkt das *Internet Protocol* (**IP**), das **ICMP** (*Internet Control Message Protocol*) oder das **IGMP** (*Internet Group Management Protocol*) verwendet werden.

2.4 Die Socket-Programmierschnittstelle

2.4.1 Vorbemerkungen

Ein Ziel bei der Entwicklung der Socket-Schnittstelle war, die bestehenden Systemaufrufe des Unix-I/O-Systems weitestgehend auch für die Netzwerkkommunikation zu nutzen (orthogonale Erweiterung). Aufgrund wesentlicher Unterschiede in der Semantik von File-I/O und Netzwerk-I/O konnte die Abstraktion *Everything is a file* nicht befriedigend erweitert werden. Als Kompromiss wurden deshalb insgesamt 17 neue Systemaufrufe für die Kommunikation mit Sockets bereitgestellt.

In BSD-basierten Systemen sind Sockets vollständig im Betriebssystem realisiert und somit erfolgt der Zugriff auf die Socket-Schnittstelle vollständig über System Calls. In SVR4-basierten Systemen sind Sockets auf der Basis des Streams-Subsystems implementiert. Die Socket-Funktionen sind dabei entweder als Bibliotheks-Funktionen unter Verwendung der Streams-Systemaufrufe oder auch im Systemkern mit einer Systemaufrufschnittstelle realisiert. Für die Anwendungsprogrammierung ist dies in der Regel irrelevant.

Sockets werden in gleicher Weise wie Dateien über Deskriptoren realisiert. Viele Systemaufrufe des I/O-Subsystems (wie z.B. *read()* oder *write()*) können so auch auf Sockets angewandt werden.

2.4.2 Überblick/Einordnung

Zuverlässige vs. nicht zuverlässige Verbindung:

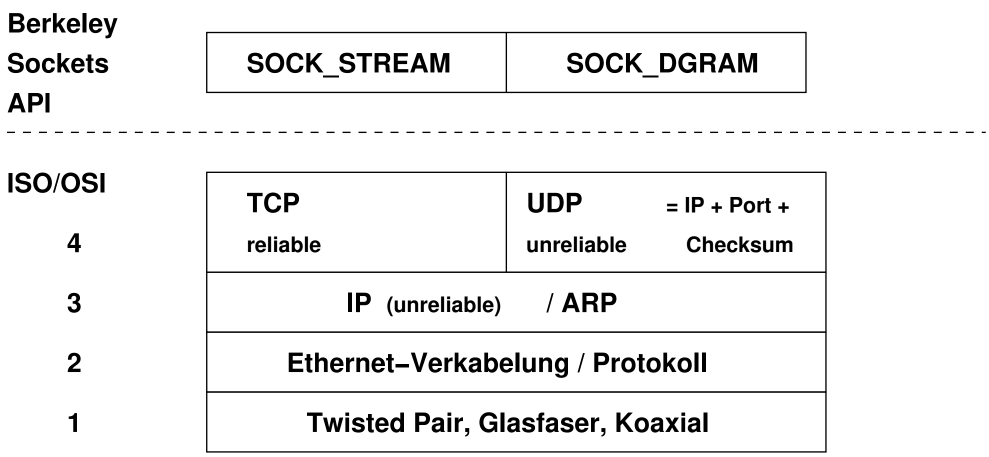

Abbildung 2.2: Sockets – Überblick

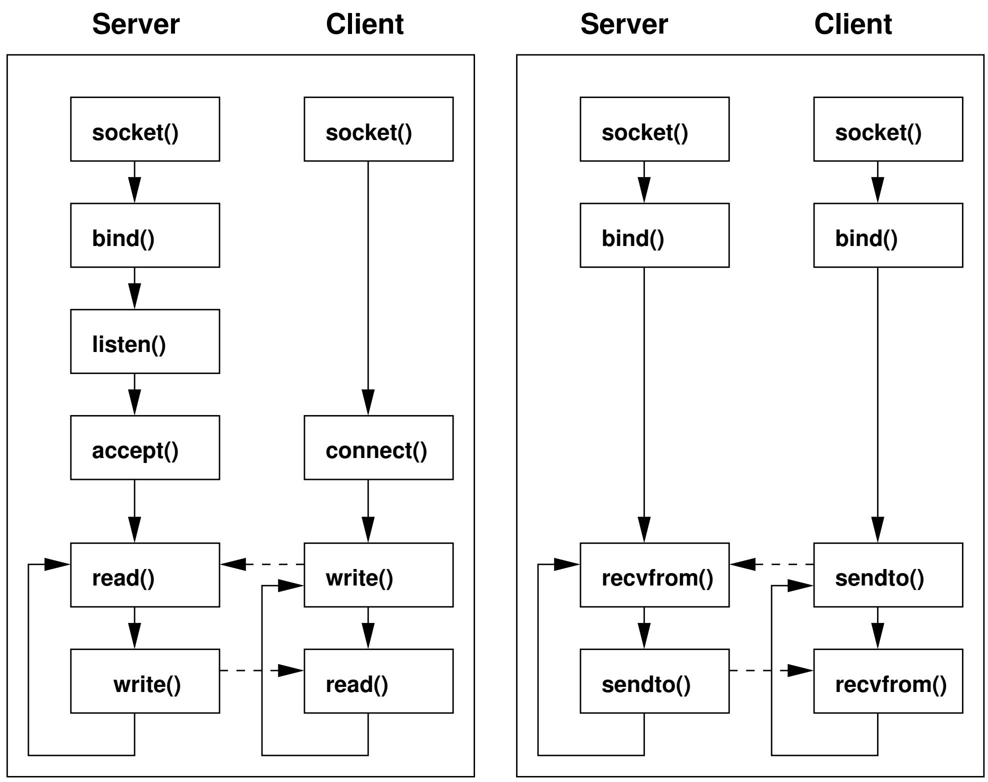

Abbildung 2.3: API-Aufrufe

Kommunikations-Domäne:

- Internet Domain: (bidirektionale) Kommunikation zwischen Rechnern
  - TCP/IP oder UDP/IP Adressierung: das 5er-Tupel

    (Protocol, SendAddr, SendPort, RecvAddr, RecvPort)
- Unix Domain: (bidirektionale) Kommunikation auf **einem** (Unix-) Rechner
  - 3er-Tupel:

    (Protocol, local-pathname, foreign-pathname)

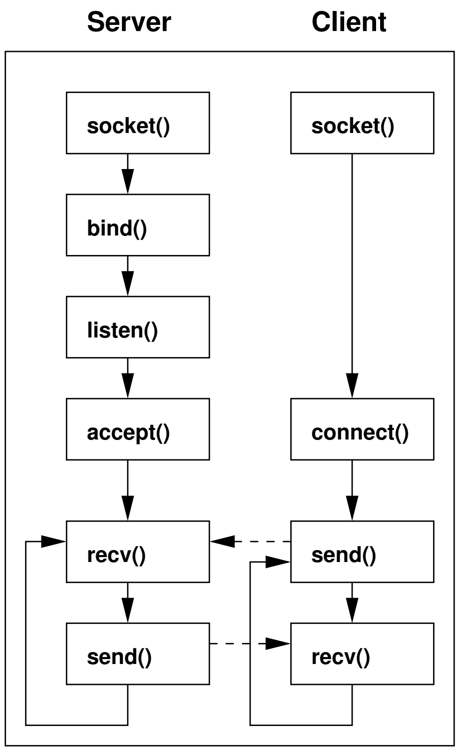

Abbildung 2.4: Kommunikation in der Unix-Doamain

2.4.3 Erzeugung eines Socket

Die Erzeugung einer Socket-Instanz erfolgt mit der Funktion **socket()**:

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

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

- Die zu verwendende Domäne wird über den Parameter *domain* angegeben.
- Die zu verwendende Kommunikationssemantik wird über den Parameter *type* angegeben.
- Das konkrete Kommunikationsprotokoll kann über den Parameter *protocol* angegeben werden. Wird hier **0** angegeben (also nichts spezifiziert), so selektiert das System eine geeignetes Protokoll passend zu den ersten beiden Parametern.
- Rückgabewert ist ein Deskriptor, der den erzeugten Socket referenziert und in allen weiteren darauf operierenden Funktionen benutzt wird.
- Für den Parameter *domain* sind in *sys/socket.h* Konstanten definiert: **AF_UNIX** (auch als **PF_UNIX**) für die Unix-Domäne, **AF_INET** (auch als **PF_INET**) für die Internet-Domäne. **AF** steht für *address family*, **PF** für *protocol family*.
- Für den Parameter *type* sind in *sys/socket.h* ebenfalls Konstanten definiert: **SOCK_STREAM** für den Socket-Typ STREAM, **SOCK_DGRAM** für DGRAM und **SOCK_DGRAM** für RAW.
- Der Rückgabewert -1 signalisiert einen Fehler:
  EPROTONOSUPPORT, falls das spezifizierte Protokoll nicht unterstützt wird
  ENOPROTOTYPE, falls der Socket-Typ innerhalb der gewählten Domäne unzulässig ist oder nicht unterstützt wird;
  weitere Fehler können aufgrund systeminterner Ressourcenknappheit oder mangelnden Zugriffsrechten entstehen.

Beispiel:

```
int sd = socket(PF_INET, SOCK_STREAM, 0);
```

Damit wird ein Stream-Socket der Internet-Domäne erzeugt, welches das voreingestellte Transportprotokoll (hier **TCP**) als das dem Socket zugrundeliegende Kommunikationsprotokoll verwendet.

2.4.4 Benennung eines Socket

Mit **socket()** werden die internen, den Socket repräsentierenden Datenstrukturen allokiert und initialisiert. In diesem Zustand kann der Socket als **unbenannter Socket** bezeichnet werden. Solange der Socket noch mit keiner Adresse verknüpft ist, kann der Socket noch nicht von fremden Prozessen angesprochen werden und somit auch keine Kommunikation stattfinden.

Die Benennung eines Socket erfolgt durch Zuweisung einer Adresse an den Socket:

- **Socket-Adresse in der Internet-Domäne**:

  ```
  <protocol,local-address,local-port,foreign-address,foreign-port>
  ```

  Ein Halb-Tupel *<protocol,address,port>* definiert dabei jeweils **einen** Kommunikationsendpunkt, *foreign* bezieht sich auf den zukünftigen Kommunikationspartner.

- **Socket-Adressen in der Unix-Domäne:**
  Hier werden die Kommunikationsverbindungen über Pfadnamen idenitifiziert, also über Tupel der Form

  ```
  <protocol,local-pathname,foreign-pathname>
  ```

Mit der Funktion **bind()** wird ein Kommunikationsendpunkt (ein Halbtupel, s.o.) festgelegt:

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

int bind(
    int               sd,         /*socket descriptor*/
    struct sockaddr * address,    /*address*/
    int               addresslen  /*length of address*/
);
```

Aufgrund der meist unterschiedlichen Adressformate der einzelnen Domänen sind Socket-Adressen als eine Folge von Bytes variabler Länge mittels einer generischen Datenstruktur anzugeben. Alle Funktionen der Socket-Schnittstelle, die Adressen verwenden, referenzieren diese nur über diese generische Datenstruktur, die in **sys/socket.h** wie folgt definiert ist:

```
struct sockaddr {
   u_char  sa_len;          /*total address length     */
   u_char  sa_family;       /*address family           */
   char    sa_data[14];     /*protocol-specific address*/
};
```

- Die Komponente *sa_len* gibt die gesamte Länge der Socket-Adresse in Bytes an.

- *sa_family* bezeichnet die Socket-Adressfamilie, die dem Adressformat der verwendeten Kommunikationsdomäne entspricht.

- *sa_data* enthält die ersten 14 Bytes der Adresse selbst. Damit wird die Länge der tatsächlichen Adresse nicht beschränkt!

Anm.: Dieser Punkt ist sehr implementierungsspezifisch und wird mit dem Übergang auf IPv6 geändert werden müssen.

Beispiel für die Initialisierung und Zuweisung einer Adresse in der Unix-Domäne (Pfadname */tmp/my_socket*):

```
# include <sys/un.h>
# include <string.h>

int sd;
struct sockaddr_un sun_addr;   /*socket address
                                *defined in <sys/un.h>
                                */
/* Create a socket in the UNIX domain: */

sd = socket(PF_UNIX,SOCK_STREAM,0);

/* Initialize the socket address: */
(void) memset(&sun_addr,0,sizeof(sun_addr));
(void) strcpy(sun_addr.sun_path,"/tmp/my_socket");
sun_addr.sun_family = AF_UNIX;
sun_addr.sun_len    = (u_char) ( sizeof(sun_addr.sun_len) +
                                 sizeof(sun_addr.sun_family) +
                                 sizeof(sun_addr.sun_path) + 1);

/* bind the address to the socket: */
bind(sd,(struct sockaddr *)&sin_addr, sizeof(sun_addr));
```

Benennung einer Adresse in der Internet-Domäne:

- Hier ist aus Rechneradresse und Portnummer eine Netzwerkadresse zu konstruieren
- Dazu gibt es eine ganze Reihe noch zu besprechender Bibliotheksfunktionen

Prinzip:

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

int sd;
struct sockaddr_in sin_addr; /*defined in <netinet.h>*/

/* create a socket: */
sd = socket(PF_INET,SOCK_STREAM,0);

/* Initialize the address: ... */

/* Bind the name to the socket: */
bind(sd, (struct sockaddr *)&sin_addr,sizeof(sin_addr));
```

2.4.5 Aubau einer Kommunikationsverbindung

Der Aufbau einer Kommunikationsverbindung zwischen zwei nicht notwendigerweise verschiedenen Prozessen verläuft in der Regel asymmetrisch, wobei ein Prozess als **Client**, der andere als **Server** bezeichnet wird. Der Server stellt normalerweise einen Dienst zur Verfügung, wartet also auf die Inanspruchnahme dieses Dienstes durch einen anderen Prozess, den Client; entsprechend werden die Kommunikationsendpunkte als passiver bzw. aktiver Kommunikationsendpunkt bezeichnet.

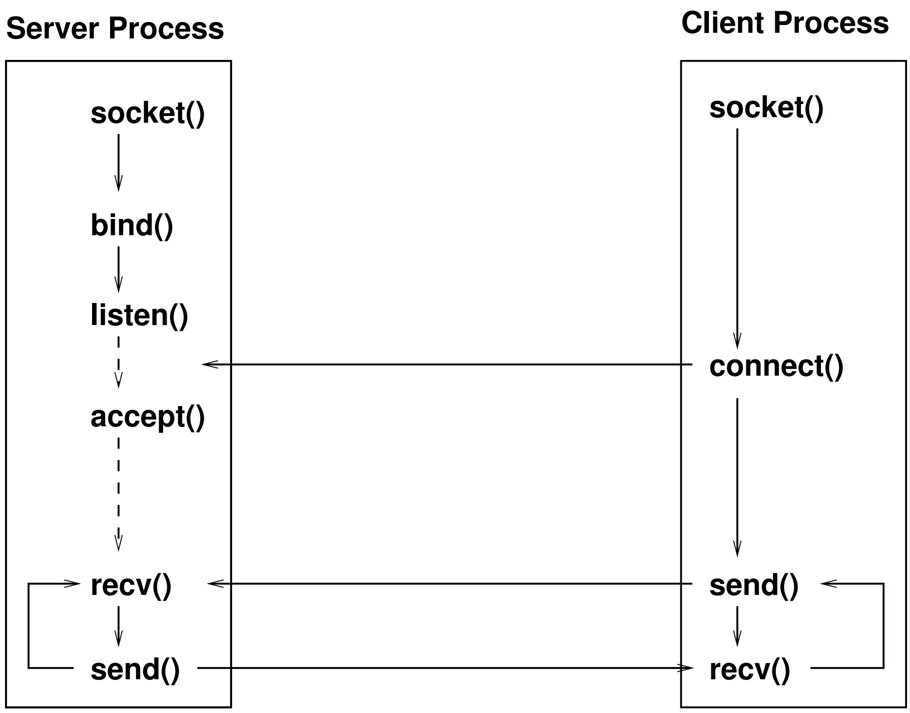

Abbildung 2.5: Verbindungsorientierte Client-Server-Kommunikation

Die Benennung des Server-Socket spezifiziert dabei die eine Hälfte einer möglichen Kommunikationsverbindung; die Vervollständigung wird durch einen Client initiiert, der aktiv die Verbindung zu einem Server anfordert und dabei die bekannte Adresse des Servers spezifiziert und seine eigene (implizit) mitliefert. Dazu dient die Funktion **connect**():

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

int connect(
    int sd,                    /*client's socket descriptor*/
    struct sockaddr * address, /* server's address*/
    int adresslen
);
```

Das zweite und dritte Argument ist analog zu den enstprechenden Parametern von *bind()* auf Server-Seite zu verstehen. Der Server hat kein explizites *bind()*

zur Adresszuweisung an seinen Socket durchzuführen, da es automatisch vom Betriebssystem vorgenommen wird, sofern der Socket zum Zeitpunkt der Verbindungsaufnahme noch unbenannt ist.

Anm.: Die Socket-Adresse des Kommunikationspartners wird in der Socket-Terminologie auch als **Peer-Adresse** bezeichnet.

Beispiel:

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

int sd;
struct sockaddr_in sin_addr;  /* for address of the server*/

sd = socket(PF_INET, SOCK_STREAM,0);

/* Initialize the socket address of the server (see below)*/

/*connect to the specified server:)
connect(sd, (struct sockaddr *)&sin)addr, sizeof(sin_addr));
```

Der Client wird durch die Ausführung von *connect()* solange blockiert, bis entweder die Kommunikationsverbindung vom Server vervollständigt wurde (somit erfolgreich hergestellt wurde) oder bis ein Fehler auftritt.

Bevor der Server Verbindungsanforderungen entgegen nehmen kann, muss sein erzeugter (*socket()*) und benannter (*bind()*) Socket als passiver Kommunikationsendpunkt gekennzeichnet werden (als Bereitschaft, Verbindungen zu akzeptieren) - dazu dient die Funktion **listen**():

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

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

- Der Parameter **backlog** spezifiziert die Länge einer Warteschlange des passiven Socket, in der Verbindungsanforderungen von Clients solange gehalten werden, bis sie vom Server explizit akzeptiert werden.

Dies erfolgt durch die Funktion **accept**():

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

int accept(
    int sd,
    struct sockaddr * address,
    int * addresslen
);
```

- Der Parameter *sd* ist der Deskriptor des benannten, passiven Socket.
- Die Argumente *address* und *addresslen* sind Wert-/Resultat-Parameter: sie müssen mit einer Variablen der domänen-spezifischen Socket-Adresse

bzw. deren Länge initialisiert werden. Nach erfolgreichem Funktionsaufruf enthalten sie die Socket-Adresse des Client bzw. deren tatsächliche Länge (also die *Peer*-Adresse). Ist der Server an der *Peer*-Adresse nicht interessiert, so ist im Parameter *addresslen* der Nullzeiger anzugeben, der Parameter *address* bleibt dabei unberücksichtigt.

- *accept()* blockiert, bis eine Verbindungsanforderung eines Clients in der Warteschlange des passiven Sockets zur Verfügung steht.
- Als Resultat liefert *accept()* - sofern keine Fehler aufgetreten ist - einen Socket-Deskriptor zurück, der einen neuen Socket referenziert; dieser repräsentiert den Kommunikationsendpunkt für die nun fertige neue Verbindung.

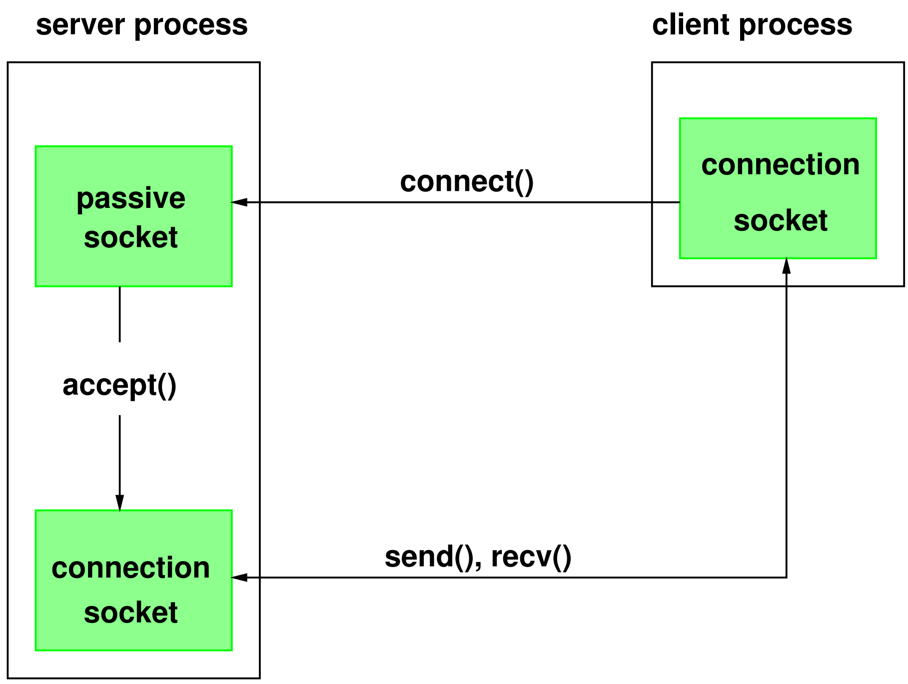

Abbildung 2.6: Aufbau einer verbind.-orient. Client/Server-Kommunikation

- Akzeptieren einer Kommunikationsverbindung in der UNIX-Domäne:

```
int                 sd,          /*server socket descriptor*/
                    conn_sd;     /*connection socket desc. */
struct sockaddr_un  client_addr; /*client socket address   */
int                 client_len;  /*length of client address*/

  /*
   * sd = socket(); bind(sd,...); listen(sd,...);
   */
```

```
client_len = sizeof(client_addr);
conn_sd    = accept(sd, (struct sockaddr *) &client_addr,
                    &client_len);
```

- Akzeptieren einer Kommunikationsverbindung in der Internet-Domäne:

```
int                 sd,           /*server socket descriptor*/
                    conn_sd;      /*connection socket desc. */
struct sockaddr_in  client_addr;  /*client socket address   */
int                 client_len;   /*length of client address*/

  /*
   * sd = socket(); bind(sd,...); listen(sd,...);
   */

client_len = sizeof(client_addr);
conn_sd    = accept(sd, (struct sockaddr *) &client_addr,
                    &client_len);
```

2.4.6 Client-Beispiel: Timeclient für Port 11011

```
#include <netdb.h>
#include <netinet/in.h>
#include <stdio.h>
#include <strings.h>
#include <sys/socket.h>
#include <unistd.h>
#include <stdlib.h>

#define TPORT    11011

int main(int argc, char **argv)
{    struct sockaddr_in addr;
    struct hostent *hp;
    int fd;
    int nbytes;
    char buf[BUFSIZ];

    if (argc!=2)
        printf("usage: %s hostname\n", argv[0]),
        exit(1);

    if (!(hp = gethostbyname(argv[1])))    // IP-Adresse des Servers holen
        fprintf(stderr, "unknown host: %s\n", argv[1]),
        exit(1);

    bzero(&addr, sizeof(addr));    // mit 0en fuellen
    addr.sin_family = AF_INET;    // TCP/IP-Verbindung
    addr.sin_port = htons(TPORT);// Port eintragen, Network Byte Order
                    // Serveraddresse eintragen
    bcopy(hp->h_addr, &addr.sin_addr, hp->h_length);

    if ((fd = socket(AF_INET, SOCK_STREAM, 0)) <0)
         perror("socket"),    // Socket erzeugen
        exit(1);

    if (connect(fd, (struct sockaddr *) &addr, sizeof(addr)) <0)
        perror("connect"),    // an Socketport verbinden
        exit(1);
                    // vom Socket lesen
    while ((nbytes = read(fd, buf, sizeof(buf))) >0)
        printf("%.*s", nbytes, buf);

    close(fd);
    exit(0);
}
```

2.4.7 Überblick: Gebrauch von TCP-Ports

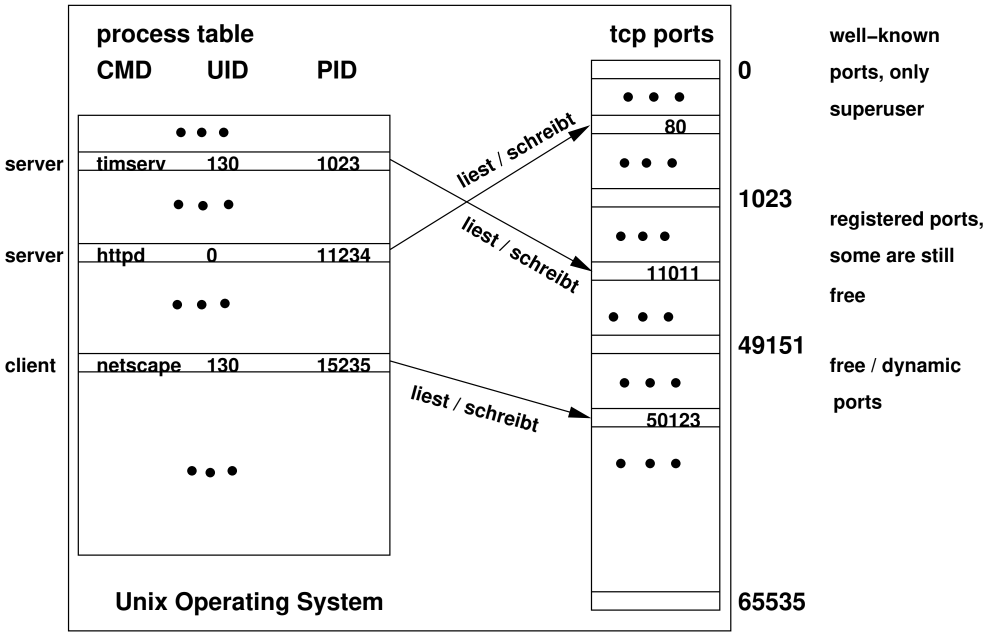

Abbildung 2.7: Ports und Prozesse in Unix

- Wie erfahre ich, welche Prozesse aktuell einen bestimmten Port belegen?
  lsof – *list open files (and ports*
  `lsof | grep TCP | grep 4711`

Beispiel:

- **netscape liest Webseite www.uni-ulm.de**

netscape:

```
gethostbyname("www.uni-ulm.de");  // get IP-Addr. server
fd = socket(AF_INET,SOCK_STREAM,0);
sin_port = htons(8);
sin_family = AF_INET;
connect(fd, [to hsot www.uni-ulm.de; port 80]);
write(fd, "GET /HTTP/1.0\m\n", 16);
read(fd,buf, sizeof(buf));
```

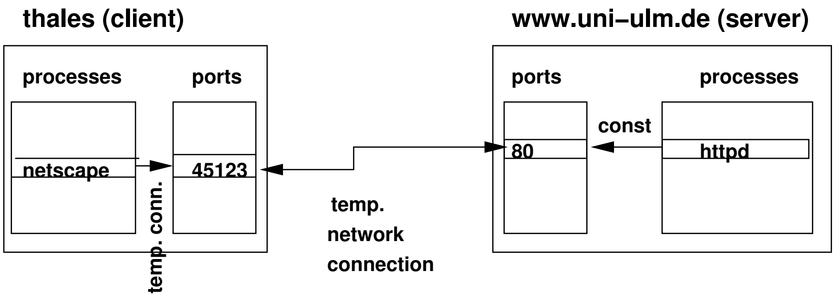

Abbildung 2.8: netscape als Client

- **httpd (Webserver-Prog.) auf www.uni-ulm.de erhält Anfrage von netscape auf thales.mathematik.uni-ulm.de**

http:

```
fd = socket(AF_INET, SOCK_STREAM,0);
sin_port = htons(80);
sin_family = AF_INET;
bind(fd, [Internetdomain; port 80]);
listen(fd, SOMAXCON); // Maximale Anzahl in Queue
nfd = accept(fd, ...);
read(nfd, buf, sizeof(buf));
write(nfd,"<html><body> ... ", 586);
```

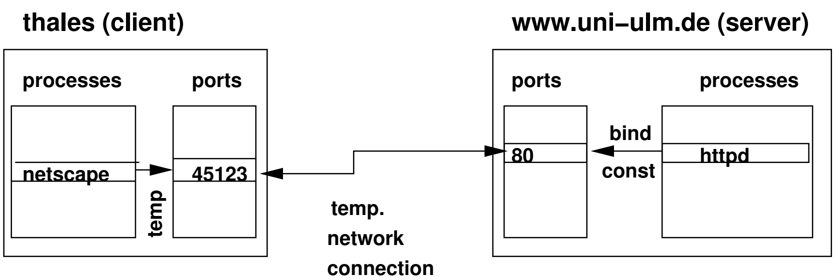

Abbildung 2.9: Server httpd antwortet

struct sockaddr

- zentraler Informationsträger für bind(), accept() und connect()
  - bei AF_INET: **struct sockaddr_in**
  - bei AF_UNIX: **struct sockaddr_un**
- wichtigster Inhalt, wenn Komponente *sin_family* auf AF_INET gesetzt:
  - sin_port = Portnummer
  - sin_addr = IP-Adresse Server / Client (bei accept)
- wer braucht was?
  - bind(): braucht Infos für Server (Protokoll + Port)
  - accept(): schreibt Infos über Client (Protokoll + IP-Adresse + Port)
  - connect(): braucht Infos über Server (Protokoll + IP-Adresse + Port)

Wie erfahre ich, **welcher Rechner** von **welchem Port** aus sich an meinen Server angedockt hat?

- `accept(fd, (struct sockaddr *) & cli_addr, ...)`
  accept() schreibt Infos über den Client in die Struktur `cli_addr`
- `printf("port: %d\n", ntohs(cli_addr.sin_port));`
  ntohs() konvertiert Network-Byte-Order in Host-Byte-Order
- `printf("host: %s\n", inet_ntoa(cli_addr.sin_addr));`
  inet_ntoa() konvertiert IP-Adresse in String (dotted decimal form)

2.4.8 Der Datentransfer

Nachdem zwischen Client und Server eine Kommunikationsverbindung aufgebaut ist, kann ein Datentransfer stattfinden. Dazu können in gewohnter Weise die System Calls des UNIX-I/O-Systems benutzt werden: **read**(), **readv**() (*read from multiple buffers*), **write**(), **writev**() (*write into multiple buffers*). Die Socket-Schnittstelle stellt 3 weitere Funktionspaare zur Verfügung, die

die Semantik der UNIX-I/O-Funktionen um Socket- und protokollspezifische Eigenschaften und Mechanismen erweitern.

- **send**(), **recv**()

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

ssize_t send ( int sd, void * buf, size_t len, int flags);
ssize_t recv ( int sd, void * buf, size_t len, int flags);
```

Ist bei beiden Funktion für *flags* der Wert 0 angegeben, so sind identisch zu *write()* und *read()*. Die Spezifikation bestimmter Methoden des Datentransfers oder das Versenden / Empfangen von Kontrollinformationen kann durch entsprechende *flag*-Werte aktiviert werden:

MSG_OOB Die spezifizierten Daten sollen mit hoher Priorität gesendet bzw. empfangen werden. Solche Daten werden im Kontext von Sockets als **out-of-band**-Daten (OOB-Daten) bezeichnet. Sie werden im Vergleich zu normalen Daten auf einem logisch unabhängigen Kanal gesendet. Bei OOB-Daten kann zudem der übliche Pufferungsmechanismus umgangen werden. Dieser Mechanismus unterliegt allerdings Beschränkungen und ist nur für wenige, verbindungsorientierte Protokolle realisiert.

MSG_PEEK Auf der Seite des Empfängers wird hiermit spezifiziert, dass gepufferte Daten nur gelesen, aber nicht "konsumiert" werden sollen. Der nächste Lesezugriff liefert dieselben Daten noch einmal zutücl.

MSG_WAITALL Der Lesezugriff soll solange blockieren, bis die angeforderte Datenmenge insgesamt zur Verfügung steht.

MSG_DONTWAIT Ausführung der Operation im nicht-blockierenden Modus.

MSG_DONTROUTE Ausgehende Datenpakete werden ohne Berücksichtigung einer Routing-Tabelle mur in das lokal angeschlossene Netzwerk gesendet. Dies ist nur für spezielle Diagnoseprogramme interessant.

MSG_EOR Die Flagge **end-of-record** markiert das logische Ende eines Datensatzes; die Daten werden dabei mit zusätzlicher Kontrollinformation versendet. Sie kann aber nur verwendet werden, wenn das zugrundeliegende Kommunikationsprotokoll das Konzept der Datensatz-Übermittlung unterstützt.

MSG_EOF Hiermit wird das Ende der Datenübertragung markiert und als Kontrollinformation zusammen mit den angegebenen Daten übertragen.

Die für den Datentransfer bereitgestellten Flaggen sind zumeist auf spezielle Kommunikationsprotokolle beschränkt und auch nur definiert, wenn die diese Protokolle auf dem System implementiert sind. Die protokollunabhängigen

Flaggen sind **MSG_PEEK**, **MSG_WAITALL** und **MSG_DONTWAIT**; die beiden letzteren stehen nur in neueren Implementierungen zur Verfügung!

- **sendto()**, **recvfrom()**

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

ssize_t sendto ( int sd, void * buf, size_t len, int flags,
                 struct sockaddr * address, int addresslen);

ssize_t recvfrom ( int sd, void * buf, size_t len, int flags,
                   struct sockaddr * address, int * addresslen);
```

Diese Funktionen stehen für den Datentransfer mit verbindungslosen Kommunikationsverfahren zur Verfügung. Dabei ist die Angabe der Sender- und Empfänger-Adresse in jedem Datenpaket erforderlich, da keine virtuelle Verbindung zwischen den Partnern besteht.
Die Angabe der Adressen (Parameter *address* und *addresslen*) sind wie bei den Funktionen *connect()* bzw. *accept()* anzugeben. Läßt man diese Parameter weg, so entsprechen diese Funktionen den obigen Funktionen *send()* und *recv()*.

- **sendmsg()**, **recvmsg()**

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

ssize_t sendmsg ( int sd, struct msghdr * msg, int flags );

ssize_t recvmsg ( int sd, struct msghdr * msg, int flags );
```

Diese beiden Funktionen stellen die allgemeinste Schnittstelle dar und erweitern die Funktionalität der obigen Funktionen. Mehr dazu siehe z.B. im Manual!

2.4.9 Terminierung einer Kommunikationsverbindung

Das Schliessen und die damit verbundene Freigabe der systeminternen Ressourcen erfolgt mit der Funktion **close()** auf den entsprechenden Socket-Deskriptor. Bei Prozess-Termination werden die Socket-Verbindungen in gleicher Weise wie die Datei- oder terminal-Verbindungen geschlossen.

In verbindungsorientierten Kommunikationsprotokollen, die ja einen verlässlichen Datentransfer garantieren, versucht das System beim Schliessen eines Socket für eine gewisse Zeit evt. noch ausstehende, zwischengepufferte Daten zu transferieren. Dies lässt sich durch entsprechende Operationen ändern.
Eine Verbindung zwischen zwei Sockets ist **voll-duplex**; dies bedeutet, dass Sende- und Empfangskanal logisch voneinander unabhängig sind. Mit der Funktion **shutdown()** lässt sich die Verbindung auch nur in einer Richtung terminieren. Dies wird typischerweise bei verbindungsorientierten Protokollen vom Sender dazu verwendet, dem Empfänger das Ende der Eingabe anzuzeigen. Die Empfängerseite bleibt für den Datentransfer weiterhin geöffnet. Das Schliessen der Empfängerseite bewirkt, dass noch nicht konsumierte, zwischengepufferte Daten wie auch alle zukünftig noch eintreffenden Daten verworfen werden.

- **shutdown()**

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

int shutdown (  int sd, int how );
```

Der Wert von *how*:

0 Leseseite (Empfängerseite) wird geschlossen

1 Schreibseite (Senderseite) wird geschlossen

2 Beide Seiten werden geschlossen

2.4.10 Verbindungslose Kommunikation

In diesem Fall läuft die Kommunikation nach einem symmetrischen Modell, auch wenn ggf. einer der Prozesse die Funktion des Servers, der andere die des Clients einnehmen kann (aber es findet kein Verbindungsaufbau statt). Die Adressen der Kommunikationspartner werden also **nicht** über eine virtuelle Kommunikationsverbindung festgelegt; in jedem Datenpaket muss stattdessen die Empfängeradresse beigefügt werden. Die Datenpakete werden bei verbindungsloser Kommunikation oft auch als **Datagramme** und die Kommunikationsendpunkte als **Datagram Sockets** (Socket-Typ: **SOCK_DGRAM**) bezeichnet.

Soll der Socket mit einer bestimmten lokalen Adresse benannt werden, so muss die Zuweisung der Adresse über die Socket-Funktion **bind()** **vor** dem ersten Datentransfer stattfinden; ansonsten erfolgt die benennung des lokalen Socket beim Senden des ersten Datagrams **implizit** durch das Betriebssystem.

- Versenden von Daten mit gleichzeitiger Spezifikation der Empängeradresse: Funktionen **sendto()** und **sendmsg()**
- Empfang von Daten mit gleichzeitiger Gewinnung der Absenderadresse: Funktionen **recvfrom()** und **recvmsg()**

Ein Datentransfer zwischen zwei Datagram-Sockets kann nur dann stattfinden, wenn beide Kommunikationsendpunkte **explizit** über die Funktion **bind**() oder **implizit** über die Funktionen **sendto**() oder **sendmsg**() benannt sind; ansonsten werden die gesendeten Datagramme verworfen!

Da verbindungslose Kommunikation auch unzuverlässige Datenzustellung bedeutet, sollte man sich in Client/Server-Anwendungen die gesendeten Datagramme vom Empfänger bestätigen lassen, sofern auf eine Anfrage keine Daten vom Empfänger erwartet werden. Ist ein kontrollierter und zuverlässiger Datentransport nötig, so müssen dies die Anwendungen in diesem selbst regeln!

- Prinzip der Kommunikation zwischen zwei Datagram Sockets:

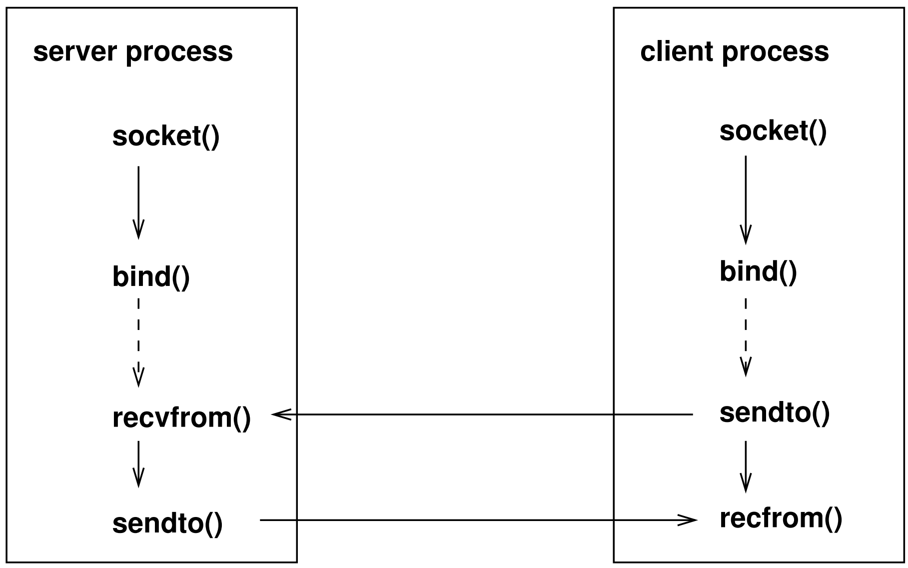

Abbildung 2.10: Aufbau einer verb.-losen Client/Server-Kommunikation

- Der Server-Prozess benennt den erzeugten Datagram Socket mit einer nach aussen hin bekannten Adresse.
- Die **explizite** Benennung des Client-Socket mit *bind()* ist **optional** und i.d.R. nicht erforderlich, da nur der Server die Adresse des Kommunikationspartners als Empfängeradresse für die zu sendende Rückantwort benötigt!
- Wichtig ist in diesem Zusammenhang nur, dass die Kommunikationsverbindung innerhalb der gewählten Domäne eindeutig ist; bei der **impliziten** Benennung eines Socket durch das Betriebssystem ist dies gewährleistet!

Auf Datagram Sockets kann die Funktion **connect()** angewandt werden; damit wird jedoch keine Kommunikationsverbindung aufgebaut, sondern die dabei spezifizierte Adresse dem Socket als Empfängeradresse zugewiesen, an die alle folgenden Datagramme zu senden sind. Ausserdem werden dann an diesem Socket nur Datagramme empfangen, deren Adresse mit der in *connect()* angegebenen Adresse übereinstimmt. Damit erübrigt sich die Identifizierung der empfangenen Datagramme. Da hiermit sowohl die Sender- wie Empfängeradresse festgelegt sind, können zum Datentransfer auch die Funktionen **send()** und **recv()** bzw. die UNIX-I/O-Systemaufrufe verwendet werden. In den Funktionen **sendto()** und **sendmsg()** sollten die Socket-Adressen aus Gründen der Portabilität unspezifiziert bleiben. Die Empfängeradresse kann durch einen erneuten Aufruf von *connect()* jederzeit geändert werden sowie eine Beziehung durch Angabe einer ungültigen Adresse gelöscht werden.

2.4.11 Feststellen gebundener Adresse

Manchmal ist es notwendig, die lokal oder entfernt gebundene Socket-Adresse allein anhand des Socket Deskriptors festzustellen; dazu dienen die Funktionen **getsockname()** und **getpeername()**, die als Resultat die lokale bzw. entfernt gebundene Adresse zurückliefern:

```
# include <sys/socket.h>
int getsockname ( int sd,
                  struct sockaddr * address, int * addresslen
                );

int getpeername ( int sd,
                  struct sockaddr * address, int * addresslen
                );
```

Die Parameter *sd, address, addresslen* sind dabei in gleicher Weise wie in der Funktion *accept()* zu spezifizieren!

Die Funktion **getsockname()** ist insbesondere dann nützlich, wenn die lokale Socket-Adresse vom System zugewiesen wurde. Das Feststellen der entfernt gebundenen Adresse mit der Funktion **getpeername()** ist dann notwendig, wenn ein Prozess diese Adresse benötigt und allein den Socket Deskriptor einer bereits akzeptierten Kommunikationsverbindung erhält und somit keinen Zugriff auf die Peer-Adresse besitzt. Dies ist beispielsweise bei durch den **Internet Superserver inetd** gestarteten Server-Prozessen der Fall.

2.4.12 Socket-Funktionen im Überblick

- **Aufbau**
  - **socket**():
    Erzeugen eines unbenannten Socket
  - **socketpair**():
    Erzeugen eines Paares von miteinander verbundenen Sockets, siehe Manual
  - **bind**():
    Zuweisung einer lokalen Adresse an einen unbenannten Socket
- **Server**
  - **listen**():
    Einen Socket auf Verbindungsanforderungen vorbereiten
  - **accept**():
    Eine Verbindungsanforderung akzeptieren
- **Client**
  - **connect**():
    Eine Verbindung zu einem Socket anfordern
- **Empfangen**
  - **read**():
    Daten einlesen
  - **readv**():
    Daten in mehrere Puffer einlesen
  - **recv**():
    Daten einlesen und Angabe von Optionen
  - **recvfrom**():
    Daten einlesen, optional Senderadresse empfangen, Angabe von Optionen
  - **recvmsg**():
    Daten in mehrere Puffer einlesen, optional Senderadresse und Kontrollinformationen empfangen, Angabe von Optionen
- **Senden**
  - **write**():
    Daten senden
  - **writev**():
    Daten aus mehreren Puffern senden
  - **send**():
    Daten senden und Angabe von Optionen
  - **sendto**():
    Daten an die spezifizierte Empfängeradresse senden, Angabe von Optionen
  - **sendmsg**():
    Daten aus mehreren Puffern senden, und Kontrollinformationen an den spezifizierten Empfäger senden, Angabe von Optionen

- **Ereignisse**
  - **select()**:
    Multiplexen und auf I/O-Bedingungen warten, siehe Manual
- **Terminieren**
  - **shutdown()**:
    Eine Verbindung in eine oder beide Richtungen terminieren
  - **close()**:
    Eine Verbindung terminieren und Socket schliessen
- **Administration**
  - **getsockname()**:
    Feststellen der lokal gebundenen Socket-Adresse
  - **getpeername()**:
    Feststellen der entfernt gebundenen Socket-Adresse
  - **setsockopt()**:
    Ändern von Socket- und Protokoll-Optionen, siehe Manual
  - **getsockopt()**:
    Auslesen von Socket- und Protokoll-Optionen, siehe Manual
  - **fcntl()**:
    Ändern der I/O-Semantik, siehe Manual
  - **ioctl()**:
    Verschiedene Socketoperationen, siehe Manual

2.5 Konstruktion von Adressen

Adressen für die Lokalisierung eines Dienstes auf einem nicht notwendig entfernten System sind protokoll-spezifisch und setzen sich in der Internet-Domäne aus einer **Rechneradresse** und einer den Dienst identifizierenden **Portnummer** zusammen. Anwendungen spezifizieren den anzufordernden Dienst i.r.G. über einen Namen statt über Rechneradresse plus Portnummer, so z.B. den WWW-Server auf dem Rechner *www.mathematik.uni-ulm.de*, dessen bereitgestellter Dienst mit *http* bezeichnet wird. Namen lassen sich schliesslich leichter merken als numerische Adressen, man erreicht dadurch auch eine gewisse Unabhängigkeit (Änderung von Adresse und Portnummer unter Beibehaltung des Namens).

Für die Konvertierung von Namen in Adressen bzw. Portnummern, für die Konstruktion und Manipulation von Netzwerkadressen sowie für die Lokalisierung eines Rechners gibt es eine Reihe von Bibliotheksfunktionen, die allerdings nicht Bestandteil der Socket-Schnittstelle sind.

2.5.1 Socket-Adressen

- Alle Funktionen der Socket-Schnittstelle, die auf Socket-Adressen operieren, verwenden eine generische Socket-Adressstruktur.
- Damit wird die Unabhängigkeit der Socket-Schnittstelle von den konkreten Implementierungen der bereitgestellten Kommunikationsprotokolle erreicht.

- Die Interpretation der Socket-Adressen erfolgt in den protokollspezifischen Funktionen, die bei der Erzeugung einer Socket-Instanz durch die Domäne festgelegt sind.
- Deklaration der Socket-Adressstruktur in auf 4.3BSD basierenden Betriebssystemen:

```
struct sockaddr {
   u_short sa_family;    /* address family              */
   char    sa_data[14];  /* protocol-specific address */
}
```

  - Die Komponente **sa_family** enthält das Adressformat.
  - Die Komponente **sa_data** maximal die ersten 14 Bytes der protokollspezifischen Adresse. Dies ist ein Implementierungsdetail der Socket-Schnittstelle und beschränkt **nicht** die Länge der protokollspezifischen Adresse.

Die Implementierung von Netzwerkprotokollen stellt mit Blick auf die Performance viele Anforderungen an die Speicherverwaltung des Betriebssystems, so z.B.

- die Handhabung von Datenpuffern unterschiedlicher Länge,
- das einfache Hinzufügen / Entfernen von Kopfdaten oder
- die Datenübergabe zwischen den eigenverantwortlichen Funktionen der verschiedenen Netzwerkschichten.

Auf BSD-Systemen ist für die Kommunikation mit Sockets eine spezielle Speicherverwaltung implementiert, die sogenannten **memory buffers**; dabei wird versucht, Kopieroperationen der Datenpakete zu minimieren. SVR6 basierende Systeme verwenden i.d.R. die Mechanismen des Streams-Subsystems.

Die Schnittstellen zwischen den Schichten des Socket-Modells sind zwar wohldefiniert, die Grenzen in der Implementierung sind eher fließend, da die einer Schicht zugrundeliegenden Datenstrukturen oft auch den Funktionen der darüberliegenden Schicht zugänglich sind. So sind das Adressformat und die ersten 14 Bytes der protokollspezifischen in der Socket-Schicht bekannt, sie sind aber auch der Anwendungsschicht bekannt (Abhängigkeit der Anwendung von den konkreten Kommunikationsprotokollen!). Bezüglich der Kompatibilität und Portabilität entsteht eine weitere Abhängigkeit dadurch, dass die generische Socket-Adresse eine Datenstruktur des Betriebssystemkerns ist und dass sich diese Struktur ab der Version *4.3BSD-Reno* (und aller darauf aufbauenden Systeme) wie folgt geändert hat:

```
struct sockaddr {
  u_char  sa_len;       /* total address length      */
  u_char  sa_family;    /* address family            */
  char    sa_data[14];  /* protocol specific address */
}
```

Die Komponente **sa_len** enthält die Länge der protokollspezifisichen Adresse in Bytes; die Gesamtlänge der Datenstruktur ist unverändert 16 Bytes! Die Hinzunahme dieser Komponente ist für die systeminterne Implementierung notwendig, damit Adressen variabler Länge protokollunabhängig behandelt werden können. Die Anwendung hat davon keinen echten Vorteil, da die

Längeninformation Argument der entsprechenden Socket-Funktionen ist, somit bereits verfügbar ist.

Die sehr systemnahe Implementierung von Netzwerkanwendungen mittels der Socket-Schnittstelle hat einige Nachteile, die sich insbesondere auf die Portabilität auswirken. Darauf soll im Folgenden jedoch nicht weiter eingegangen werden (Fragen und Lösungen diesbezüglich wurden in der Dissertation von M. Etter behandelt).

2.5.2 Socket-Adressen der UNIX-Domäne

Hier werden für die Benennung von Sockets UNIX-Pfadnamen benutzt; die folgende Socket-Adressstruktur ist in *sys/un.h* definiert:

```
struct sockaddr_un {
  u_char sun_len;       /*total address length incl. nullbyte*/
  u_char sun_family;    /*address family AF_UNIX            */
  char   sun_path[104]; /*path name                         */
}
```

- In der Komponente **sun_len** ist die Länge der konkreten Socket-Adresse in Bytes anzugeben; diese ergibt sich aus der Länge der beiden Komponenten **sun_len** und **sun_family** sowie der Stringlänge des Pfadnamens in **sun_path** plus 1 (terminierendes Null-Byte!).
- Die Komponente **sun_family** bezeichnet die Adressfamilie bzw. das Adressformat und ist stets mit der Konstanten **AF_UNIX** zu initialisieren.
- Die Dimension der Komponente **sun_path** mit 104 Bytes ist systemabhängig; in den meisten UNIX-Systemen ist die maximale Länge eines Pfadnamens auf 1024 Bytes (Konstante **MAX_PATH** in *limits.h*) festgelegt. In Socket-Implementierungen, die auf dem Streams System basieren, können Sockets mit Pfadnamen bis zu dieser Länge benannt werden. Werden in Socket-Implementierungen die Adressstrukturen der Version *4.3BSD-Reno* verwendet, so ist die maximale Pfadlänge durch den Datentyp **u_char** der Komponente **sun_len** auf 256 Bytes beschränkt!

Die Socket-Adressen sollten einer sicheren Konvention folgend immer mit Null-Bytes initialisiert werden. Die Verwendung von absoluten Pfadnamen ist in einigen System erforderlich.

- Initialisierung:

```
struct sockaddr_un addr;
(void) memset(&addr, 0, sizeof(addr));
(void) strcpy(addr.sun_path, "/tmp/my_socket");
addr.sun_family = AF_UNIX;
# ifdef HAVE_SOCKADDR_SA_LEN
addr.sun_len = (u_char) (sizeof(addr.sun_len) +
                         sizeof(addr.sun_family) +
                         strlen(addr.sun_path) +1);
# endif
```

2.5.3 Socket-Adressen in der Internet-Domäne

Hier ist die Socket-Adressstruktur in **netinet/in.h** wie folgt definiert:

```
struct in_addr {
   u_long s_addr; /* 32 bit netid/hostid */
};

struct sockaddr_in {
  u_char         sin_len;    /*total address length (16 bytes)*/
  u_char         sin_family; /*address family AF_INET         */
  u_short        sin_port;   /*16 bit port number             */
  struct in_addr sin_addr;   /*IP address                     */
  char           sin_zero[8];/*unused                         */
};
```

- Die Datenstruktur **in_addr** enthält nur die Komponente **s_addr**, in der eine 32 Bit lange Adresse des Internetprotokolls IP in Netzwerkbyteordnung gespeichert wird.
- Die Komponente **sin_len** der Datenstruktur **sockaddr_in** ist immer **sizeof(struct sockaddr_in) = 16** Bytes.
- Die Adressfamilie in Komponente **sin_family** ist immer **AF_INET**.
- Die Komponente **sin_port** ist eine 16 Bit lange Port-Nummer in Netzwerkbyteordnung, die zusammen mit der Internet-Adresse **sin_addr** einen Kommunikationsendpunkt eindeutig' definiert.
- Die Komponente **sin_zero** ist aus Gründen der Portabilität mit Null-Bytes zu initialisieren und dient lediglich zur Ausweitung der Datenstruktur auf die Länge der generischen Socket-Adressstruktur **sockaddr**:

Für die implizite Selektierung einer geeigneten lokalen IP-Adresse oder auch Portnummer durch das System existieren jeweils eine ausgezeichnete IP-Adresse und eine Portnummer mit Sonderbedeutung: die **IP-Adresse 0** (**INADDR_ANY**) bzw. **0.0.0.0** in punktiertem Dezimalformat sowie die **Portnummer 0**. Bei der Spezifikation der Komponenten **sin_addr.s_addr** und **sin_port** mit diesen sog. *Wildcards* wird die lokale IP-Adresse nach einem erfolgreichen Verbindungsaufbau entsprechend der ein- / ausgehenden Netzwerkschnittstelle automatisch vom System festgelegt und bei der Benennung eines Socket eine frei Portnummer aus dem Bereich der **kurzlebigen Portnummern** gewählt.

2.5.4 Byte-Ordnung

Die Anordnung von Bytes bei Mehr-Byte-Grössen erfolgt nicht bei allen Computersystemen in der gleichen Reihenfolge. Für die Speicherung einer 2-Byte-Grösse gibt es zwei Möglichkeiten:

- das niederwertige Byte liegt an der Startadresse (**Little-Endian-Anordnung**)
- das höherwertige Byte liegt an der Startadresse (**Big-Endian-Anordnung**)

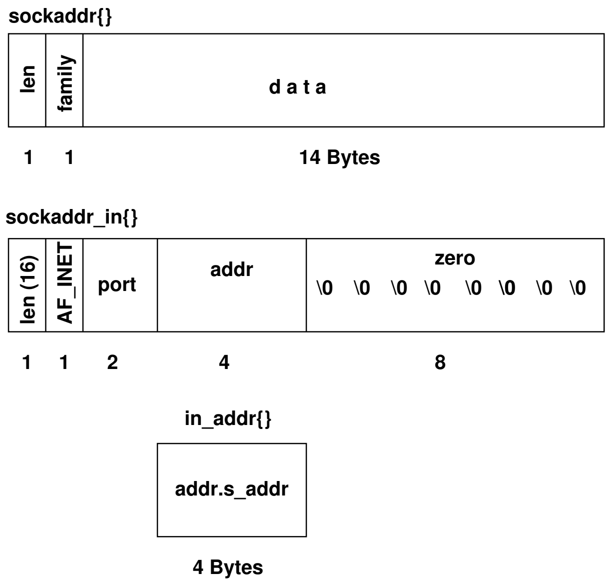

Abbildung 2.11: Organisation der Adress-Struktur sockaddr_in

Bei 4-Byte-Grössen können zusätzlich noch die 2-Byte-Grössen unterschiedlich angeordnet sein!

Für den Austausch protokollspezifischer Daten werden in den Internet-Protokollen nur 2- und 4-Byte-Integerwerte im Big-Endian-Format verwendet (**network byte order**), die Bit-Ordnung selbst ist ebenfalls in diesem Format!

Die Implementierungen von Netzwerkprotokollen sind somit auf jedem System dafür verantwortlich, dass protokollspezifische Daten in Netzwerkbyteordnung transferiert werden, d.h. die Daten sind von der Byteordnung des Computersystems (*host*) in die Netzwerkbyteordnung zu transferieren, sofern sich die Anordnungen unterscheiden. Zur Konvertierung von 2-Byte-Grössen (*short*) und 4-Byte-Grössen (*long*) gibt es folgende Funktionen (Makros):

```
u_short htons(u_short hostshort); /*host-to-network short*/

u_long  htonl(u_long  hostlong); /*host-to-network long*/

u_short ntohs(u_short netshort); /*network-to-host short*/

u_long  ntohl(u_long  netlong); /*network-to-host long*/
```

Die protokollspezifischen Mehrbytegrössen, die bei Internet-Protokollen zu spezifizieren sind, sind die Internet-Adresse und die Portnummer in *sockaddr_in* (Komponenten *sin_addr.s_addr* und *sin_prt*).

2.5.5 Spezifikation von Internet-Adressen

Die 32-Bit-IP-Adressen werden meist in der sog. **punktiertes Dezimalformat** (**dotted decimal notation**) angebenen; diese entsteht dadurch, dass byteweise Dezimalzahlen gebildet werden, die durch Punkt getrennt sind.

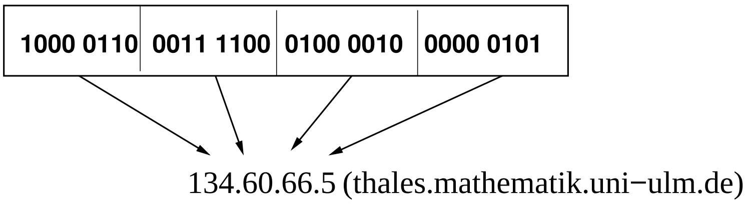

Abbildung 2.12: dotted-decimal notation

Jeder **Host** in einem IP-Netzwerk ist über eine IP-Adresse eindeutig identifiziert. Ist ein Host an mehrere IP-Netze angeschlossen (**multi-homed host**), so muss dieser für jedes angeschlossene Netz eine IP-Adresse besitzen. Über einen Alias-Mechanismus können einem Host auch mehrere IP-Adressen zugeordnet werden.

Zur Manipulation und Konvertierung von IP-Adressen gibt es wieder einige Bibliotheksfunktionen, die im Headerfile **arpa/inet.h** definiert sind.

- Umwandlung eines als String in *dotted-decimal notation* vorliegende IP-Adresse in eine 32-Bit-IP-Adresse: Funktion **inet_addr**()

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

  unsigned long inet_addr ( char * ipaddr);
  ```

  Rückgabewerte ist eine 32-Bit-IP-Adresse oder im Fehlerfall die Konstante **INADDR_NONE** (0xffffffff), die allerdings einer gültigen **Broadcast-Adresse** entspricht. Zu beachten ist auch, dass der Rückgabewert *unsigned long* und nicht *struct in_addr*. Dies behebt die folgende Funktion:

- **inet_aton**()

  ```
  # include <sys/socket.h>
  # include <netinet/in.h>
  # include <arpa/inet.h>

  int inet_aton (
      char           * ipaddr;   /*dotted decimal notation  */
      struct in_addr * in_addr;  /*result: 32-bit-IP-address*/
  );
  ```

  Rückgabewert ist 1 im Erfolgsfall, 0 sonst!

- Konvertierung einer 32-Bit-IP-Adresse in *dotted-decimal notation*: Funktion **inet_ntoa**()

```
# include <sys/socket.h>
# include <netinet/in.h>
# include <arpa/inet.h>

char * inet_ntoa ( struct in_addr inaddr);
```

- Beispiel für eine Anwendung:

```
struct sockaddr_in addr;

if(inet_aton("134.60.66.5", &addr.sin_addr))
   (void) printf("IP-Address: %s\n",
                inet_ntoa(addr.sin_addr));
```

2.5.6 Hostnamen

Die Beziehung zwischen Hostnamen und IP-Adressen werden in der Internet-Domäne jeweils in der Datenstruktur **hostent** repräsentiert, die als Resultat der Funktionen **gethostbyname**() und **gethostbyaddr**() geliefert wird. Diese Funktionen zur **Adress-Resolution** werden als **Resolver** bezeichnet.

Die Struktur *hostent* (in **netdb.h** definiert):

```
struct hostent {
   char *   h_name;      /*official name of host             */
   char **  h_aliases;   /*alias list                        */
   int      h_addrtype;  /*host address type (address family)*/
   int      h_length;    /*length of address                 */
   char **  h_addr_list; /*address list from name server     */
}
# define h_addr h_addr_list[0]
                         /*address for backward compatibility*/
```

Diese Datenstruktur beschreibt den offiziellen Namen des Rechners in der Komponente **h_name**, eine Liste seiner öffentlichen Alias-Namen in **h_aliases**, den Adress-Typ in **h_addrtype**, die Länge einer Adresse in **h_length** und eine Liste der IP-Adressen (in Netzwerkbyteordnung) in **h_addr_list**. Falls es sich bei dem Rechner um einen *multi-homed host* handelt oder Alias-Adressen definiert sind, so enthält die Liste der IP-Adressen entsprechend viele Elemente. Aus Kompatibilitätsgründen verweist die Makrodefinition **h_addr** auf das erste Element dieser Liste.

Die Funktionen **gethostbyname**() und **gethostbyaddr**():

```
# include <netdb.h>
struct hostent * gethostbyname (char * name);
struct hostent * gethostbyaddr ( char * addr,
                                 int    len,
                                 int    type
                               );
```

In der Funktion **gethostbyaddr()** ist die protokollspezifische Adresse in Netzwerkbyteordnung, deren Länge und der Adress-Typ anzugeben. In der Internet-Domäne sind als Parameter eine IP-Adresse, deren Länge, zu erhalten als **sizeof(struct in_addr)**, und die Konstante **AF_INET** anzugeben.

Die Spezifikation von **Hostnamen** erfolgt entweder über **einfache** Namen wie beispielsweise **thales** oder über **absolute** Namen wie **thales.mathematik.uni-ulm.de.**. Ein absoluter Namen wird auch als **Fully Qualified Domain Name** (**FQDN**) bezeichnet; diese müssen mit einem Punkt enden, der die Wurzel des hierarchisch geordneten Namensraums bezeichnet. Relative Hostnamen werden abhängig von der administrativen Konfiguration des Systems mit Hilfe der auf dem lokalen System voreingestellten Namensdomäne vervollständigt. In Benutzeranwendungen wird der abschliessende Punkt bei absoluten Namen meist weggelassen.

Beispiel:

```
/* hostent.c: print hostent */

# include <stdio.h>
# include <netdb.h>
# include <sys/socket.h>
# include <netinet/in.h>
# include <arpa/inet.h>

void print_hostent( char * host ) {
  struct hostent * hp;

  (void) printf("%s:\n", host);
  if ( (hp = gethostbyname(host)) ) {
     char ** ptr;

     (void) printf("Offizieller Host-Name: %s\n", hp->h_name);

     for(ptr = hp->h_aliases; ptr && *ptr; ptr++)
        (void) printf("   alias: %s\n", *ptr);

     if( hp->h_addrtype == AF_INET)
        for(ptr = hp->h_addr_list; ptr && *ptr; ptr++)
           (void) printf("   Adresse: %s\n",
                         inet_ntoa(*((struct in_addr *) *ptr)));
  } else
        (void) printf("----> Kein Eintrag!\n");

  (void) printf("------------------------------------------\n");
}

int main(int argc, char ** argv) {
int i;
  for(i = 1 ; i < argc; i++)
     print_hostent(argv[i]);
exit(0);
}
```

```
thales$ gcc -o hostent -Wall -lxnet hostent.c
thales$ hostent thales turing
thales:
Offizieller Host-Name: thales
   alias: ftp
   alias: www
   alias: pop
   alias: glueck
   alias: adi
   Adresse: 134.60.66.5
------------------------------------------
turing:
Offizieller Host-Name: turing
   alias: loghost
   alias: mailhost
   Adresse: 134.60.166.1
------------------------------------------
thales$
```

Die Beziehungen zwischen **Hostnamen** und **Internet-Adressen** werden in der verteilten Datenbank des **Domain Name System** (**DNS**) verwaltet; die darin enthaltenen Informationen sind über sogenannte **Nameserver** zugänglich. Alternativ dazu werden Informationen über Hostnamen und Internet-Adressen auf dem lokalen System in der Datei **/etc/hosts** oder über den **Network Information Service** (**NIS**) bereitgestellt. Die unterschiedlichen Möglichkeiten der **Adress-Resolution** zeigt die folgende Abbildung:

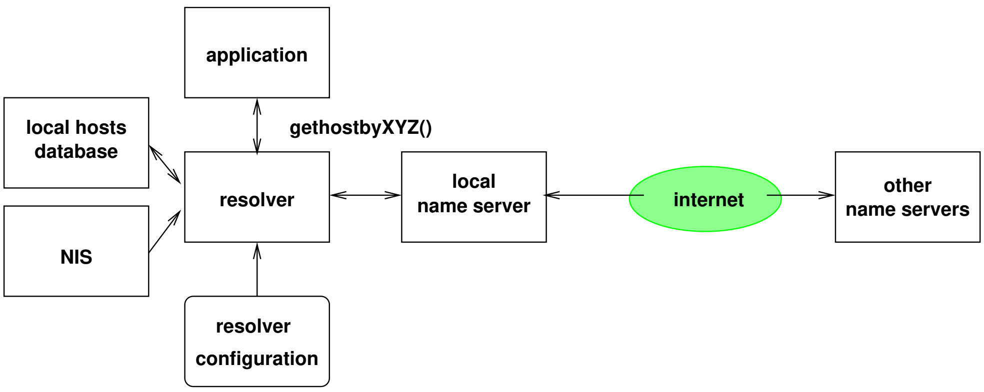

Abbildung 2.13: Methoden der Adress-Resolution

Die auf eine Anfrage gelieferten Informationen variieren aufgrund der verschiedenen Zugangsverfahren und auch unterschiedlichen Organisationen der Datenbanken. Die Zugangsverfahren hängen auch von der Implementierung wie der administrativen Konfiguration des Resolvers ab. Die Resolver-Funktionen liefern auf jeden Fall den offiziellen Hostnamen und eine IP-Adresse in der Struktur **hostent** zurück. Bei Verwendung lokaler Mechanismen liefern einige Systeme jedoch nur einfache und keine absoluten Hostnamen zurück. Die wesentlichen Unterschiede zeigen sich bei Aliasnamen und Aliasadressen. Wird beispielsweise die Host-Tabelle oder NIS verwendet, erhält man genau eine Adresse und alle Aliasnamen. Werden die Informationen über Nameserver angefordert, so erhält man eventuell Aliasadressen und höchstens einen Aliasnamen, sofern es sich bei demi in der Anfrage spezifizierten Hostnamen um einen Aliasnamen handelt; dazu noch einmal obiges Programm:

```
thales$ hostent www   #using NIS
www:
Offizieller Host-Name: thales
   alias: ftp
   alias: www
   alias: pop
   alias: glueck
   alias: adi
   Adresse: 134.60.66.5
------------------------------------------
thales$ hostent www.mathematik  #using DNS
www.mathematik:
Offizieller Host-Name: thales.mathematik.uni-ulm.de
   Adresse: 134.60.66.5
------------------------------------------
```

2.5.7 Lokale Hostnamen und IP-Adressen

Besteht bereits eine Verbindung, so können der lokale **Hostname** und die lokale **IP-Adresse** mit Hilfe der Socket-Funktion **gethostname()** und der Resolver-Funktion **gethostbyaddr()** ermittelt werden.

- Die Funktion **gethostname()**:

  ```
  # include <unistd.h>

  int gethostname(char * name, size_t namelen);
  ```

  Beispiel:

  ```
  hypatia$ cat gethost.c
  /* gethost.c: Hostnamen bestimmen */

  # include <stdio.h>
  # include <unistd.h>

  void main() {

     char name[20];
     if ( gethostname(name,20) == 0 )
        (void) printf("Hostname: %s\n", name);
  }
  hypatia$ gcc -Wall -o gethost gethost.c
  hypatia$  gethost
  Hostname: hypatia
  hypatia$
  ```

  Die maximale Länge eines Hostnamens ist auf den meisten Systemen über die Konstante **MAXHOSTNAMELEN** im Headerfile **sys/param.h** festgelegt.

- Das Kommando **uname**:

```
hypatia$ uname -a
Linux hypatia 2.2.13 #5 Mon Apr 3 12:46:02 MEST 2000 i586 unknown
hypatia$
```

- Die Funktion **uname**():
  Dazu ist im Headerfile **sys/utsname.h** folgende Datenstruktur definiert:

```
struct utsname {
     char  sysname[SYS_NMLN];  /*operating system name          */
     char  nodename[SYS_NMLN]; /*node name (host name)          */
     char  release[SYS_NMLN];  /*operating system release level*/
     char  version[SYS_NMLN];  /*operating system version level*/
     char  machine[SYS_NMLN];  /*hardware type                  */
}
```

Die Funktion selbst ist

```
int uname( struct utsname * buf);
```

Beispiel:

```
hypatia$ cat uname.c
/* uname.c: get host info */

# include <stdio.h>
# include <sys/utsname.h>

void main(){

   struct utsname buf;
   if( uname(&buf) == 0 ) {
     (void) printf("Host: %s\nOS: %s\nRelease: %s\n",
                    buf.nodename, buf.sysname, buf.release);
     (void) printf("Version: %s\nHardware: %s\n",
                      buf.version, buf.machine);
   }
}

hypatia$ gcc -Wall -o my_uname uname.c
hypatia$ my_uname
Host: hypatia
OS: Linux
Release: 2.2.13
Version: #5 Mon Apr 3 12:46:02 MEST 2000
Hardware: i586
hypatia$
```

2.5.8 Portnummern und Dienste

In der Internet-Protokollfamilie werden in den Protokollen der **Transportschicht** (**TCP** und **UDP**) 16 Bit lange **Portnummern** für die Identifizierung eines Dienstes bereitgestellt. Diese 65536 Portnummern werden von der **IANA** (*Internet Assigned Numbers Authority*) in drei Bereiche eingeteilt: **well-known ports** ($0-1023$), **registered ports** ($1024-49151$) und **dynamic and/or private ports** ($49152 - 65535$). Der aktuelle Stand ist in der Datei

- **ftp://ftp.isi.edu/in-notes/iana/assigments/port-numbers**

verfügbar.

Well-known ports identifizieren bekannte Internet-Dienste. So wird in allen TCP/IP-Implementierungen dem **Telnet-Server telnetd** die TCP-Portnummer 23 und dem **TFTP-Server tftpd** (*Trivial File Transfer Protocol*) die UDP-Portnummer 69 zugewiesen, sofern dieses Anwedungsprotokoll unterstützt wird und die entsprechenden Netzwerkanwendungen auf dem System bereitgestellt sind. Auf UNIX-Systemen werden die Portnummern aus dem Bereich $1 - 1023$ als **reservierte Portnummern** bezeichnet und können nur von Prozessen mit Superuser-Privilegien zur Benennung von Sockets verwendet werden. Die sog. *well-known ports* belegen hier die Portnummern $1 - 511$ und die Portnummern $512 - 1023$ sind für Client-Anwendungen mit Supreuser-Privilegien reserviert, die eine reservierte Portnummer als Bestandteil der Client/Server-Authentifizierung benötigen. Beispiele dafür sind **rlogin** und **rsh**.

Von der IANA nicht verwaltet werden Dienste, die registrierte Portnummern verwendet (lediglich als Konvention aufgelistet). So sind z.B. die Portnummern $6000-6063$ für einen **X Window Server** für beide Protokolle (allerdings derzeit nur TCP verwendet) registriert. Nach Konvention binden X-Server für passive Sockets die Portnummern $6000 + x$, wobei x die Nummer des Displays angibt. Wengleich die registrierten Portnummern frei verfügbar sind zur Benennung eines Socket beliebig genutzt werden können, ist dies keinesfalls zu empfehlen. Über die dynamischen und privaten Portnummern, häufig auch als **kurzlebige Portnummern** (*ephemeral ports*) bezeichnet, wird von der IANA nichts ausgesagt. Sie sind für eine implizite Benennung eines Socket durch das Betriebssystem reserviert. Die meisten UNIX-Systeme binden heute noch die Nummern $1024 - 5000$ für kurzlebige Portnummern; damit können maximal 3977 Sockets (typischerweise für Client-Anwendungen) zu einem Zeitpunkt implizit benannt sein. Solaris-Betriebssysteme verwenden kurzlebige Portnummern aus dem Bereich $32768 - 65535$.

Die Beziehungen zwischen Portnummern und den offiziellen Namen der Dienste sowie deren Aliasnamen werden in der Datei **/etc/services** oder einer entsprechenden NIS-Tabelle definiert. Dazu ist im Headerfile **netdb.h** folgende Struktur definiert:

```
struct servent {
  char *   s_name;    /*official service name          */
  char **  s_aliases; /*alias list                     */
  int      s_port;    /*port number (network byte order*/
  char *   s_proto;   /*protocol to use                */
}
```

Die Zugriffsfunktionen:

- **getservbyname()**

```
# include <netdb.h>

struct servent * getservbyname(
      char * name,
      char * protocol
);
```

- **getservbyport()**

```
# include <netdb.h>

struct servent * getservbyport(
      int port;
      char * protocol
);
```

Beispiel:

```
thales$ cat getserv.c
/* getserv.c: get service */

# include <stdio.h>
# include <netdb.h>
# include <netinet/in.h>

void print_servent( char * name, char * protocol) {
    struct servent * sp;

    if( (sp = getservbyname(name,protocol)) ) {
        char ** ptr;

        (void) printf("Offizieller Name des Dienstes: %s\n",
                      sp->s_name);

        for( ptr = sp->s_aliases; ptr && *ptr; ptr++)
             (void) printf("   Alias: %s\n", *ptr);

        (void) printf("   Port-#: %d\n",
                      ntohs( (u_short) sp->s_port));
        (void) printf("   Protokoll: %s\n", sp->s_proto);
   } else
        (void) printf("Kein Eintrag!\n");
}

int main(int argc, char ** argv) {
     if( argc != 3 ) {
        fprintf(stderr, "Usage: %s service protocol\n", argv[0]);
exit(1);
     } else
```

```
        print_servent(argv[1], argv[2]);
     exit(0);
}
thales$ gcc -Wall -o getserv -lxnet getserv.c
thales$ getserv telnet tcp
Offizieller Name des Dienstes: telnet
   Port-#: 23
   Protokoll: tcp
thales$
```

2.5.9 Protokoll- und Netzwerkinformationen

Die Beziehungen zwischen Protokollnamen und Protokollnummern sowie zwischen Netzwerknamen und Netzwerkadressen werden in zwei weiteren Tabellen (auf dem lokalen System in den Dateien **/etc/protocols** und **/etc/networks**) sowie ggf. in entsprechenden NIS-Tabellen gehalten. Der Zugriff darauf erfolgt vergleichbar zu Hostnamen und Diensten über zwei in **netdb.h** definierte Datenstrukturen und entsprechenden Zugriffsfunktionen.

- Die Datenstruktur **struct protoent**:

```
struct protoent {
  char *   p_name;     /*official protocol name */
  char **  p_aliases;  /*alias list             */
  char *   p_proto;    /*protocol number        */
};
```

- Die Funktionen **getprotobyname**() und **getprotobynumber**():

```
# include <netdb.h>

struct protoent * getprotobyname( char * name );

struct protoent * getprotobynumber( int number );
```

- Die Datenstruktur **struct netent**:

```
struct netent {
  char *   n_name;      /*official net name*/
  char **  n_aliases;   /*alias list       */
  int      n_addrtyupe; /*net type         */
  u_long   n_net;       /*net number       */
}
```

- Die Zugriffsfunktionen **getnetbyname**() und **getnetbyaddr**():

```
# include <netdeb.h>

struct netent * getnetbyname( char * name );

struct netent * getnetbyaddr( u_long  addr,
                              int type );
```

Das Internet-Protokoll verwendet eine 8 Bit lange Protokollnummer zur Identifizierung der nächst höheren Protokollschicht, an die das transportierte IP-Datagram weiterzuleiten ist. Die für das Internet-Protokoll definierten Protokollnummern werden von der **IANA** verwaltet und sind in RFC1700 enthalten; die aktuelle Version findet sich unter

- **ftp://ftp.isi.edu/in-notes/iana/assignments/protocol-numbers**

2.5.10 Zusammenfassung der Netzwerkinformationen

Die Erzeugung und Benennung eines Kommunikationsendpunktes erfordert

- die Selektion eines Kommunikationsprotokolls und
- die Konstruktion einer protokollspezifischen Adresse, die in der Internetdomäne aus
  - aus einer Netzwerkadresse,
  - einer Rechneradresse und
  - einer den Diest spezifizierenden Portnummer

  zusammengesetzt ist.

Für eine unabhängige Spezifikation dieser Informationen über Namen statt über protokollspezifische Adressen und Nummern stehen den Anwendungen vier Tabellen zur Verfügung, die die Beziehungen zwischen Namen, deren Aliasnamen und den protokollspezifischen Informationen enthalten.

Information	Tabelle	Datenstruktur	Funktionen
Host	/etc/hosts	hostent	gethostbyname(), gethostbyaddr()
Dienst	/etc/services	servent	getserbyname(), getservbyport()
Protokoll	/etc/protocols	protoent	getprotobyname(), getprotobynumber()
Netzwerk	/etc/networks	netent	getnetbyname(), getnetbyaddr()

Die in der Spalte **Tabelle** angegebenen Informationen werden oftmals zentral mit Hilfe des **Network Information Service** (**NIS**) verwaltet, damit Änderungen und Erweiterungen nicht auf jedem System lokal zu aktualisieren sind. Die Schnittstellen der Zugriffsfunktionen sind unabhängig von der Lokalität und Organisation der Tabellen.

Zusätzlich existieren für jede der vier Tabellen je drei weitere Funktionen, die das sequentielle Auslesen der gesamten Tabelle unabhängig von deren Lokalität und Organisation ermöglichen.
In den Folgenden Funktionsnennungen sind die Buchstaben **XXX** jeweils zu ersetzen durch einen der Namen in der Spalte *Datenstruktur*:

- **struct XXX * getXXX(void);**

  Diese Funktionen öffnen ggf, die entsprechende Tabelle, lesen jeweils den nächsten Eintrag und liefern als Resultat einen Zeiger auf die entsprechend initialisierte Datenstruktur **XXX** zurück. Ist das Ende der Tabelle erreicht, liefern sie den Null-Zeiger.

- **void setXXX(int stayopen);**

  Diese vier Funktionen öffnen jeweils die entsprechende Tabelle und setzen deren internen Positionszeiger auf den Anfang zurück. Ist das Argument *stayopen* ungleich 0, so können auch die in der jeweiligen Spalte *Funktionen* angegebenen Zugriffsfunktionen mit entsprechenden Selektionskriterien zum sequentiellen Auslesen der Tabelle benutzt werden. Die Verbindung zu einer vt. zugrundeliegenden Netzwerkdatenbank (NIS) wird durch die Ausführung der Zugriffsfunktionen in diesem Fall nicht geschlossen.

- **void endXXX(void);**

  Diese Funktionen schliessen die zugehörige Tabelle und beenden eine evt. bestehende NIS-Verbindung;

Das Auslesen von Host- / Netzwerkinformationen, die mit Hilfe des **Domain Name System** (**DNS**) verwaltet werden, ist **nicht** möglich. Hier liefern die Zugriffsfunktionen den Inhalt der lokalen bzw. der via NIS verwalteten Tabellen zurück. Die Funktion *gethostent()* kann also **nicht zum Auslesen der gesamten Hostinformationen des Internet benutzt werden!**

2.5.11 IPv6

Das Internet-Protocol der nächsten Generation **IP next generation** ist die neue version des aktuellen Internet-Protokolls in der Version 4 (**IPv4**). Die offizielle Bezeichnung des neuen Protokolls ist **IPv6** (z.Zt. noch als *Draft Standard*).

- Anstatt der bislang 32 Bit langen Adressen werden in IPv6 128 Bit lange Adressen benutzt (vergrößerter, besser strukturierter Adressraum)
- Limitierungen bzgl. der Wegwahl von IP-Paketen (*routing*) und der Konfiguration von Netzwerken werden beseitigt.
- Verbesserte Sicherheitsmechanismen wie Verschlüsselung und Authentifizierung werden integriert.
- Vorgesehen sind Mechanismen für eine prioritätsgesteuerte Datenflusskontrolle von Echtzeitanwendungen (z.B. Übermittlung von multmedialen Daten in Echtzeit).

Literatur zu IPv6 - z.B.:

- Gilligan, R.E.; Nordmark, E.: Transition Mechanisms for IPv6 Hosts and Routers. RFC 1933, Sun Microsystems, Inc., April 1996
- Huitema, C.: IPv6 - The New Internet Protocol. Prentice Hall, Inc., Englewood Cliffs, 1996

Kapitel 3

Netzwerk-Programmierung

3.1 Client/Server

3.1.1 Vorbemerkungen

In diesem Abschnitt sollen einige der zuletzt dargestellten Konzepte und Funktionen an einer einfachen Client/Server-Implementierung exemplarisch dargestellt werden. Für die Implementierung des Servers gibt es im Prinzip zwei Möglichkeiten:

- Der Server arbeitet die ankommenden Anforderungen sukzessive ab (**iterative server**).
- Der Server arbeitet die ankommenden Anforderungen parallel ab (**concurrent server**).

3.1.2 concurrent server

Server *fork*t, neuer Prozess bearbeitet Anforderung

```
/* Concurrent Server: conc_srv.c
 * simple error handling
 */

int sockfd, newsockfd;

if ( (sockfd = socket( ... ) ) < 0 ) {
   perror("socket error");
   exit(1);
}
if ( bind(sockfd, ...) < 0 ) {
   perror("bind error");
   exit(1);
}
if ( listen(sockfd, 5) < 0 ) {
   perror("listen error");
   exit(1);
}

while (1) {
   newsockfd = accept(sockfd, ...); /* blockiert */
```

```
   if ( newsockfd < 0 ) {
      perror("accept error");
      exit(1);
   }
   switch (pid = fork() ) {
      case -1:
         perror("fork error");
         exit(1);
      case 0:
         close(sockfd);
         /*verarbeite Anforderung:*/
         doit(newsockfd);
         exit(0);
      default:
         close(newsockfd);
         break;
   }
}
```

3.1.3 iterative server

```
/* Iterative Server: iter_srv.c
 * simple error handling
 */

int sockfd, newsockfd;

if ( (sockfd = socket( ... ) ) < 0 ) {
   perror("socket error");
   exit(1);
}
if ( bind(sockfd, ...) < 0 ) {
   perror("bind error");
   exit(1);
}
if ( listen(sockfd, 5) < 0 ) {
   perror("listen error");
   exit(1);
}

while (1) {
   newsockfd = accept(sockfd, ...); /* blockiert */
   if ( newsockfd < 0 ) {
      perror("accept error");
      exit(1);
   }
   /*verarbeite Anforderung:*/
   doit(newsockfd);
   close(newsockfd);
}
```

3.2 echo-Server und echo-Client

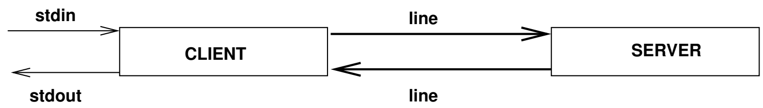

Abbildung 3.1: echo: Client/Server

- Client liest eine Zeile (Folge von Zeichen bis zu einem *newline*!) von *stdin* und übergibt diese an den Server
- Server liest die Zeile über seine Netzwerk-Eingabe und gibt sie über seine Netzwerk-Ausgabe an den Client zurück (*echo*)
- Der Client liest die Zeile von der Socket Schnittstelle, gibt sie an *stdout* aus und terminiert.

3.3 Erste Implementierungen

3.3.1 Headerfiles

Die wesentlichen Include-Anweisungen für die Inet-Domäne sind im Headerfile **inet.h**, die für die Unix-Domäne in **unix.h** zusammengefasst:

```
/* ------------------ inet.h -------------------------
 * Definitions for TCP and UDP client / server programs
 */

#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <errno.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define SERV_UDP_PORT 6000
#define SERV_TCP_PORT 5529
   /* must not conflict with
    * any other TCP server's port
    */
#define SERV_HOST_ADDR  "127.0.0.1"
/* "134.60.66.5": it's thales */

/* "127.0.0.1"  */
   /*local host*/
```

```
#define MAXLINE 256

/* unix.h
 *
 * Definitions for UNIX domain stream and datagram
 * client / server programs
 */

#include   <stdio.h>
#include   <unistd.h>
#include   <strings.h>
#include   <sys/types.h>
#include   <sys/socket.h>
#include   <sys/un.h>

#define UNIXSTR_PATH "./s.unixstr"
#define UNIXDG_PATH  "./s.unixdg"

#define MAXLINE 512
```

3.3.2 TCP-Verbindung - Concurrent Server

Der **echo-Server** wird als *concurrent server* realisiert. Um **Zombie**-Prozesse zu vermeiden, wird der bereits früher behandelte **signal handler** verwendet:

```
/* -------------------- sign.h ------------------------ */
/* Signal Handler                                        */

#ifdef SIGN_H
#include <signal.h>
#include <sys/wait.h>
#include <unistd.h>

typedef void (*Sigfunc)(int);

#else

#define SIGN_H
/* avoid multiple includes */
#include <signal.h>

typedef void (*Sigfunc)(int);

Sigfunc ignoresig(int);
/* ignore interrupt and avoid zombies
         * just for midishell (parent)
         */
Sigfunc ignoresig_bg(int);
/* ignore interrupt  -
         * just for execution of background commands
         */
Sigfunc entrysig(int);
```

```
/* restore reaction on interrupt */
#endif

/* ------------------------ sign.c ------------------------ */

#define SIGN_H

# include <stdio.h>
# include "sign.h"

void shell_handler(int sig){
  if( (sig == SIGCHLD) || (sig == SIGCLD)) {
     int status;
     waitpid(0, &status, WNOHANG);
  }
  return;
}

struct sigaction newact, oldact;

Sigfunc ignoresig(int sig) {
   static int first = 1;
   newact.sa_handler = shell_handler;
   if (first) {
      first = 0;
      if (sigemptyset(&newact.sa_mask) < 0)
          return SIG_ERR;
      newact.sa_flags = 0;
      newact.sa_flags |= SA_RESTART;
      if (sigaction(sig, &newact, &oldact) < 0)
              return SIG_ERR;
      else
              return oldact.sa_handler;
   } else {
      if (sigaction(sig, &newact, NULL) < 0)
              return SIG_ERR;
      else
              return NULL;
   }
}


struct sigaction newact_bg, oldact_bg;

Sigfunc ignoresig_bg(int sig) {
   newact_bg.sa_handler = SIG_IGN;
   if (sigemptyset(&newact_bg.sa_mask) < 0)
          return SIG_ERR;
   newact_bg.sa_flags = 0;
   newact_bg.sa_flags |= SA_RESTART;
   if (sigaction(sig, &newact_bg, &oldact_bg) < 0)
              return SIG_ERR;
   else
              return oldact_bg.sa_handler;
```

```
}



Sigfunc entrysig(int sig)   {
   if (sigaction(sig, &oldact, NULL) < 0 )
      return SIG_ERR;
   else
      return NULL;
}
```

Der Echo-Server:

```
/* -----------------main_srv.c --------------------------
 * server using TCP protocol
 *
 * simple error handling
 */

#include "inet.h"
#include "sign.h"

int main() {
   int sockfd,newsockfd,clilen,childpid, n;
   struct sockaddr_in cli_addr, serv_addr;
   char recvline[MAXLINE];

   if( (ignoresig(SIGCHLD) == SIG_ERR) ) {
       perror("Signal");
       exit(1);
   }
   /*
    * open a TCP socket - Internet stream socket
    */
   if ( (sockfd = socket(AF_INET, SOCK_STREAM,0)) < 0) {
      perror("server: can't open stream socket");
      exit(1);
   }

   /*
    * bind our local address so that the client can send us
    */
   /*bzero((char *) &serv_addr, sizeof(serv_addr));*/
   memset((char *) &serv_addr, 0, sizeof(serv_addr));
   serv_addr.sin_family = AF_INET;

   serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
   /* INADR_ANY: tells the system that we'll accept a connection
    * on any Internet interface on the system, if it is multihomed
    * Address to accept any incoming messages (-> in.h).
    * INADDR_ANY is defined as ((unsigned long int) 0x00000000)
    */

   serv_addr.sin_port = htons(SERV_TCP_PORT);
```

```
   if ( bind(sockfd, (struct sockaddr *)&serv_addr,
         sizeof(serv_addr)) < 0) {
      perror("server: can't bind local address");
      exit(1);
   }

   listen(sockfd,5);

   while(1) {
      /*
       * wait for a connection from a client process
       * - concurrent server -
       */
      clilen=sizeof(cli_addr);
      newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
               &clilen);

      if (newsockfd < 0 ) {
         if (errno == EINTR)
            continue; /*try again*/
         perror("server: accept error");
         exit(1);
      }

      if ( (childpid = fork()) < 0) {
         perror("server: fork error");
         close(sockfd);
         close(newsockfd);
         exit(1);
      }
      else if (childpid == 0) {
         close(sockfd);
         if( (n=recv(newsockfd, recvline,MAXLINE,0)) < 0)
            exit(2);
         if( send(newsockfd,recvline,n,0) < n)
            exit(3);
         close(newsockfd);
         exit(0);
      }

      close(newsockfd); /*parent*/
   }
}
```

Der Echo-Client:

```
/* -------------------- main_cli.c -----------------------
 * client using TCP protocol
 */

#include "inet.h"

int main(){
```

```
   int sockfd;
   struct sockaddr_in serv_addr;
   char sendline[MAXLINE], recvline[MAXLINE];
   int n;

   /*
    * fill in the structure "serv_addr" with address
    * of server we want to connect with
    */

   /*bzero( (char *) &serv_addr, sizeof(serv_addr));*/
   memset( (char *) &serv_addr, 0, sizeof(serv_addr));
   serv_addr.sin_family = AF_INET;
   serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
   serv_addr.sin_port = htons(SERV_TCP_PORT);

   /*
    * open a TCP socket - internet stream socket
    */

   if ( (sockfd = socket(AF_INET, SOCK_STREAM,0)) < 0 ) {
      perror("client: can't open stream socket");
      exit(1);
   }

   /*
    * connect to the server
    */

   if (connect(sockfd, (struct sockaddr *) &serv_addr,
         sizeof(serv_addr) ) < 0) {
      perror("client: can't connect to server");
      exit(2);
   }

   printf("%25s", "Client - give input: ");
   if( fgets(sendline,MAXLINE,stdin) != NULL) {
       n = strlen(sendline);
       if (send(sockfd, sendline,n,0) < n)
           exit(3);
       shutdown(sockfd,1);
       if(recv(sockfd,recvline,n,0) < 0)
           exit(4);
       recvline[n] = '\0';
       printf("%25s%s\n","Client - got: ", recvline);
   }

   close(sockfd);
   exit(0);
}
```

Ausführung:

```
hypatia$ make -f LinMakefile.srv
```

```
gcc -Wall -c main_srv.c
gcc -Wall -c sign.c
gcc -Wall -o server main_srv.o sign.o
hypatia$ make -f LinMakefile.cli
gcc -Wall -c main_cli.c
gcc -Wall -o client main_cli.o
hypatia$ server &
[1] 658
hypatia$  client
    Client - give input: Eine Eingabezeile
           Client - got: Eine Eingabezeile

hypatia$ ps | grep server
  658 pts/0    00:00:00 server
hypatia$ kill 658
[1]+  Terminated              server
hypatia$
```

3.3.3 UDP-Verbindung - Iterative Server

Der Echo-Server:

```
/* ------------------main_srv.c ---------------------------
 * server using UDP protocol
 *
 * simple error handling
 */

#include "inet.h"

int main() {
   int sockfd,clilen, n;
   struct sockaddr_in cli_addr, serv_addr;
   char recvline[MAXLINE];

   /*
    * open a UDP socket - Internet datagramm socket
    */
   if ( (sockfd = socket(AF_INET, SOCK_DGRAM,0)) < 0) {
      perror("server: can't open datagramm socket");
      exit(1);
   }

   /*
    * bind our local address so that the client can send us
    */
   bzero((char *) &serv_addr, sizeof(serv_addr));
   serv_addr.sin_family = AF_INET;

   serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
   /* INADR_ANY: tells the system that we'll accept a connection
    * on any Internet interface on the system, if it is multihomed
    * Address to accept any incoming messages (-> in.h).
    * INADDR_ANY is defined as ((unsigned long int) 0x00000000)
```

```
    */

   serv_addr.sin_port = htons(SERV_UDP_PORT);

   if ( bind(sockfd, (struct sockaddr *)&serv_addr,
         sizeof(serv_addr)) < 0) {
      perror("server: can't bind local address");
      exit(1);
   }
   clilen=sizeof(cli_addr);

   while(1) {
      /*
       * wait for a connection from a client process
       * - iterative server -
       */

      if( (n=recvfrom(sockfd, recvline,MAXLINE,0,
              (struct sockaddr *)&cli_addr,&clilen)) < 0) {
            perror("server - recvfrom");
            exit(2);
      }
      if( sendto(sockfd,recvline,n,0, (struct sockaddr *)&cli_addr,
                 sizeof(cli_addr)) < n) {
            perror("server - sendto");
            exit(3);
      }
  }
}
```

Der Echo-Client:

```
/* -------------------- main_cli.c ----------------------
 * client using UDP protocol
 */

#include "inet.h"

int main(int argc, char * argv){
   int sockfd;
   struct sockaddr_in cli_addr, serv_addr;
   char sendline[MAXLINE], recvline[MAXLINE];
   int n;

   /*
    * fill in the structure "serv_addr" with address
    * of server we want to connect with
    */

   bzero( (char *) &serv_addr, sizeof(serv_addr));
   serv_addr.sin_family = AF_INET;
   serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
   serv_addr.sin_port = htons(SERV_UDP_PORT);
```

```
    /*
     * open a UDP socket - internet datagramm socket
     */

    if ( (sockfd = socket(AF_INET, SOCK_DGRAM,0)) < 0 ) {
       perror("client: can't open datagramm socket");
       exit(1);
    }

    /*
     * bind any local address for us
     */
    bzero( (char *) &cli_addr, sizeof(cli_addr));
    cli_addr.sin_family = AF_INET;
    cli_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    cli_addr.sin_port = htons(0);

    if (bind(sockfd, (struct sockaddr *) &cli_addr,
          sizeof(cli_addr)) < 0 ) {
       perror("client: can't bind local address");
       exit(2);
    }

    printf("%25s", "Client - give input: ");
    if( fgets(sendline,MAXLINE,stdin) != NULL) {
        n = strlen(sendline);
        if (sendto(sockfd, sendline,n,0,(struct sockaddr *)&serv_addr,
                   sizeof(serv_addr)) < n) {
            perror("client - sendto");
            exit(3);
        }
        shutdown(sockfd,1);
        if(( n=recvfrom(sockfd,recvline,n,0,(struct sockaddr *)0,
                        (int *)0)) < 0) {
            perror("client - recvfrom");
            exit(4);
        }
        recvline[n] = '\0';
        printf("%25s%s\n","Client - got: ", recvline);
    }

    close(sockfd);
    exit(0);
}
```

Ausführung:

```
hypatia$ make -f LinMakefile.srv
gcc -Wall -c main_srv.c
gcc -Wall -o server main_srv.o
hypatia$ make -f LinMakefile.cli
gcc -Wall -c main_cli.c
gcc -Wall -o client main_cli.o
hypatia$ server &
```

```
[1] 946
hypatia$ client
    Client - give input: Noch 'ne Zeile
           Client - got: Noch 'ne Zeile

hypatia$ ps | grep server
  946 pts/0    00:00:00 server
hypatia$ kill 946
[1]+  Terminated              server
hypatia$
```

3.3.4 TCP-Verbindung in der UNIX Domain

Der Echo-Server:

```
/* main_srv.c
 * server using UNIX domain stream protocol
 */

#include "unix.h"
#include "sign.h"


int main() {
  int sockfd,newsockfd,clilen,childpid,servlen,n;
  struct sockaddr_un cli_addr, serv_addr;
  char recvline[MAXLINE];

  if( (ignoresig(SIGCHLD) == SIG_ERR) ) {
      perror("Signal");
      exit(1);
  }

  /*
   * open a UNIX domain stream socket
   */
   if ((sockfd=socket(AF_UNIX,SOCK_STREAM,0)) < 0 ) {
     perror("server: can't open a stream socket");
     exit(2);
  }

  /*
   * bind our local address so that the client can send to us
   */

  /* set all with zeros */
  bzero( (char *) &serv_addr, sizeof(serv_addr));

  serv_addr.sun_family = AF_UNIX;
  strcpy(serv_addr.sun_path, UNIXSTR_PATH);

  /* determine length of address: */
  servlen = strlen(serv_addr.sun_path) + sizeof(serv_addr.sun_family);

  if (bind(sockfd, (struct sockaddr *) &serv_addr, servlen) < 0 ) {
    perror("server: can't bind local address");
    exit(3);
  }

  listen(sockfd,5);

  while(1) {
    /*
     * wait for a connection from a client process
     * - concurrent server -
     */
```

```
    clilen = sizeof(cli_addr);
    newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
          &clilen);
    if (newsockfd < 0 ) {
      perror("server: accept error");
      exit(4);
    }

    if ( (childpid = fork() ) < 0 ) {
      perror("server: can't fork");
      exit(5);
    }
    else if (childpid == 0) {  /* child */
      close(sockfd);
      if( (n=recv(newsockfd, recvline, MAXLINE,0)) < 0)
         exit(6);
      if( send(newsockfd, recvline,n,0) < n)
         exit(7);
      close(newsockfd);
      exit(0);
    }
    /*parent:*/
    close(newsockfd);
  }
}
```

Der Echo-Client:

```
/* main_cli.c
 *
 * client using Unix domain stream protocol
 */

#include "unix.h"

int main() {
   int sockfd, servlen;
   struct sockaddr_un serv_addr;
   char sendline[MAXLINE], recvline[MAXLINE];
   int n;

   /*
    * Fill in the structure "serv_addr" with the
    * address of the server that we want to sent do
    */
   bzero( (char *) &serv_addr, sizeof(serv_addr) );
   serv_addr.sun_family = AF_UNIX;
   strcpy(serv_addr.sun_path, UNIXSTR_PATH);
   servlen = strlen(serv_addr.sun_path) + sizeof(serv_addr.sun_family);

    /*
     * open an Unix domain stream socket
     */
   if ( (sockfd = socket(AF_UNIX, SOCK_STREAM, 0) ) < 0) {
```

```
      perror("client: can't open stream socket");
      exit(1);
   }

   /*
    * connect to the server
    */
   if (connect(sockfd, (struct sockaddr *) &serv_addr, servlen) < 0) {
      perror("client: can't connect to server");
      exit(1);
   }

   printf("%25s", "Client - give input: ");
   if( fgets(sendline,MAXLINE,stdin) != NULL) {
        n = strlen(sendline);
        if (send(sockfd, sendline,n,0) < n)
            exit(3);
        shutdown(sockfd,1);
        if(recv(sockfd,recvline,n,0) < 0)
            exit(4);
        recvline[n] = '\0';
        printf("%25s%s\n","Client - got: ", recvline);
   }


   close(sockfd);
   exit(0);
}
```

Ausführung:

```
hypatia$ make -f LinMakefile.sr v
gcc -Wall -c main_srv.c
gcc -Wall -c sign.c
gcc -Wall -o server main_srv.o sign.o
hypatia$ make -f LinMakefile.cl i
gcc -Wall -c main_cli.c
gcc -Wall -o client main_cli.o
hypatia$ server &
[1] 959
hypatia$ ls -l s.*
srwxr-xr-x   1 swg      users           0 Apr 30 12:12 s.unixstr
hypatia$ client
    Client - give input: one line
           Client - got: one line

hypatia$ ps | grep server
  959 pts/4    00:00:00 server
hypatia$ kill 959
[1]+  Terminated              server
hypatia$ server &
[1] 966
hypatia$ server: can't bind local address: Address already in use
```

```
[1]+  Exit 3                  server
hypatia$ rm s.unixstr
hypatia$
```

3.3.5 Modifikation der ersten Implementierung

Der Client liest Zeile für Zeile von der Standardeingabe, schickt diese sukzessive an den Server, liest sie wieder und gibt sie „markiert“ wieder an die Standardausgabe. Im folgenden wird nur der geänderte Teil dargestellt:

Der Client-Teil:

```
   while ( fgets(sendline,MAXLINE,stdin) != NULL) {
       n = strlen(sendline);
       if (send(sockfd, sendline,n,0) < n)
           exit(3);
       if(recv(sockfd,recvline,n,0) < 0)
           exit(4);
       recvline[n] = '\0';
       printf(">>> %s", recvline);
   }
```

Der Server-Teil:

```
      else if (childpid == 0) {
         close(sockfd);
         while ( (n=recv(newsockfd, recvline,MAXLINE,0)) > 0) {
            if( send(newsockfd,recvline,n,0) < n)
                exit(3);
         }
         close(newsockfd);
         exit(0);
      }
```

Ausführung:

```
hypatia$ server &
[1] 1282
hypatia$ client < cli.src
>>>    while ( fgets(sendline,MAXLINE,stdin) != NULL) {
>>>        n = strlen(sendline);
>>>        if (send(sockfd, sendline,n,0) < n)
>>>            exit(3);
>>>        if(recv(sockfd,recvline,n,0) < 0)
>>>            exit(4);
>>>        recvline[n] = '\0';
>>>        printf(">>> %s", recvline);
>>>    }
hypatia$
```

3.3.6 Anmerkungen

Diese Implementierungen sind wenig robust (triviales Fehlerhandling); sie sollten die prinzipelle Anwendung der Socket Funktionen demonstrieren. Leseund Schreiboperationen auf *Stream Sockets* können auch weniger als die spezifizierte Anzahl von Bytes als Resultatwert liefern. Dies ist generell bei allen über einen Deskriptor referenzierten Objekten möglich, wenn beispielsweise der zugrundeliegende Systemaufruf durch ein Signal unterbrochen wurde oder der Deskriptor sich im nicht-blockierenden Modus befindet. Dies kann auch dadurch passieren, dass beim Lesen oder Schreiben Ressourcenbeschränkungen verletzt werden oder momentan keine weitere Eingabe zur Verfügung steht. Die exakte Semantik ist von dem konkret referenzierten Objekt abhängig.

Bei *Stream Sockets* ist das Ein- / Ausgabeverhalten vom Erreichen der Socket-Puffergrenzen im Betriebssystemkern abhängig. Die Kommunikationsverbindung ist hier **bidirektional** und **voll-duplex**; jeder Socket besitzt einen **Sendepuffer** und einen **Empfangspuffer**, die beide voneinander unabhängig sind. Die Verläßlichkeit des Datentransfers wie auch die Datenflusskontrolle werden über das TCP-Protokoll und die internen Algorithmen der TCP-Implementierungen mit Hilfe der Sende- und Empfangspuffer der miteinander verbundenen Sockets geregelt.

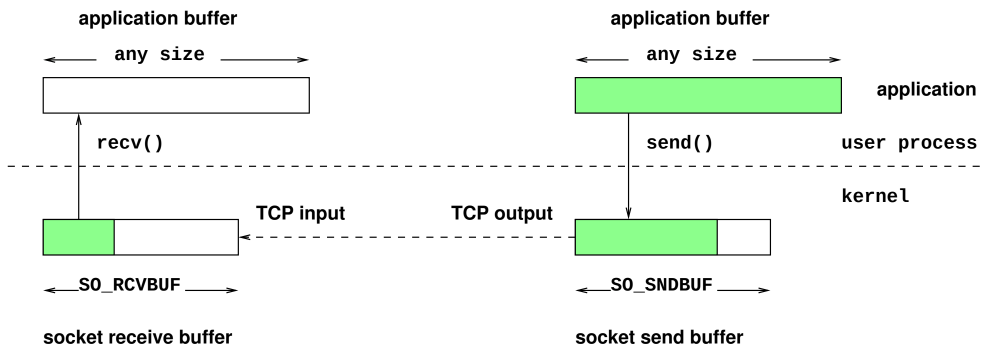

Abbildung 3.2: Socket-Pufferung (vereinfacht)

Die Dimensionen des Sende- und Empfangpuffers sind systemabhängig voreingestellt und lassen sich über die beiden Socket-Optionen **SO_SNDBUF** und **SO_RCVBUF** modifizieren. Die Dimension des Puffers in der Anwendung ist frei wählbar.

Die Ausführung der I/O-Funktionen auf Sockets bewirkt letztlich nur, dass die Daten zwischen dem Puffer der Anwendung (im Benutzeradressraum) und dem enstprechenden Socket-Puffer (im Adressraum des Kernels) kopiert werden. Im Fall einer Schreiboperation zeigt die als Resultat gelieferte Anzahl von Bytes nur an, dass diese Anzahl in den Sendepuffer des Socket kopiert wurden und nicht, dass diese Zahl von Bytes den Empfänger erreicht hat. Entsprechend werden beim Lesen die momentan im Empfangspuffer des Socket gehaltenen Daten in den Anwendungspuffer kopiert. Der tatsächliche Datentransfer über den Kommunikationskanal wird von den Ein- / Ausgabefunktionen der TCP-Implementierung **asynchron** vorgenommen und kann von der Anwendung nicht beeinflusst werden. Das Verhalten der Lese- und Schreibfunktionen ist also von den momentan im Empfangspuffer enthaltenen Daten bzw. dem freien Bereich im Sendepuffer abhängig (ähnlich der Funktionsweise von Pipes).

Lesefunktionen liefern (im Erfolgsfall) immer das Minimum aus der angeforderten Datenmenge und den im Empfangspuffer gehaltenen Bytes zurück – sofern der Systemaufruf nicht durch ein Signal unterbrochen wurde – maximal aber **SO_RCVBUF** Bytes! Schreibfunktionen versuchen, die spezifizierte Anzahl von Bytes zu senden und blockieren solange, bis die gesamte Datenmenge in den Puffer kopiert wurde. Im Erfolgsfall gilt also, dass die als Resultat gelieferte Anzahl der spezifizierten Anzahl entspricht – sofern kein Signal den Systemaufruf unterbricht! Diese Funktionalität kann auch in den Lesefunktionen der Socket-Schnittstelle durch Spezifikation der Flagge **MSG_WAITALL** erreicht werden. Die Existenz diese Flagge wie auch die Eigenschaft, dass Schreibfunktionen versuchen, die spezifizierte Datenmenge insgesamt zu versenden, sind allerdings von der jeweiligen Implementierung der Socket-Schnittstelle abhängig!

In vielen Netzwerkanwendungen ist es oft notwendig, logisch unvollständige Lese- und Schreiboperationen – die im Fall eines unterbrochenen Systemaufrufs ja auch keinen Fehler darstellen – entsprechend zu behandeln. Dazu können z.B. die folgenden beiden Funktionen verwendet werden:

- **sendn():**

```
/* ipc_send.c */
# define IPC_SEND_H
# include "ipc_send.h"

ssize_t sendn(int fd, char * buf, size_t len) {

   char *  ptr = buf;
   size_t  nc;
   ssize_t n;

   for(nc = len; nc > 0; ptr += n, nc -=n) {
      send_again:
      if( (n=send(fd,ptr,nc,0)) <= 0)
          if(errno == EINTR) {
               errno = 0;
               goto send_again;
          } else
               return ( len -= nc) ? len : -1;
    }
    return len;
}
```

- **recvn():**

```
/* ipc_recv.c */
# define IPC_RECV_H
# include "ipc_recv.h"

ssize_t recvn(int fd, char * buf, size_t len) {
```

```
    char *    ptr = buf;
    size_t    nc;
    ssize_t   n;
    int       flags = 0;


    for(nc = len; nc > 0; ptr += n, nc -= n) {
        recv_again:
        if( (n = recv(fd, ptr, nc, flags)) < 0 )
            if(errno == EINTR) {
                errno = 0;
                goto recv_again;
            } else
                return (len -= nc) ? len : -1;
         else  if( n == 0 ) {
              errno = ENOTCONN;
              return (len -= nc) ? len : 0;
         }
         if(buf[n-1] == '\n') return n;
    }
    return len;
}
```

Diese beiden Funktionen implementieren inkrementelles Schreiben und Lesen, die die Anwendung jeweils solange blockieren, bis die spezifizierte Datenmenge insgesamt gesendet resp. empfangen wurde; sie berücksichtigen zudem die Möglichkeit, dass Systemaufrufe durch Signale unterbrochen werden können.

Die Funktion **sendn()** ist unabhängig von der Semantik und Implementierung der Socket-Funktion **send()** realisiert: in einer Schleife werden die noch ausstehenden Daten durch weitere Aufrufe von **send()** gesendet, falls die als Resultat gelieferte Anzahl von Bytes kleiner als die spezifizierte Datenmenge ist. Die Funktion **recvn()** ist ähnlich realisiert; unterstützt die Socket-Funktion **recv()** die Flagge **MSG_WAITALL**, so wird diese auch zur Performance-Steigerung beim Einlesen der Daten genutzt. Ist inkrementelles Lesen erforderlich, so übernimmt in diesem Fall die Funktion **recv()** diese Aufgabe selbst und erspart der Anwendung weitere Systemaufrufe. Dennoch ist auch in diesem Fall die Schleifenkonstruktion erforderlich, falls **recv()** durch ein Signal unterbrochen wird und einen weiteren Aufruf notwendig macht.

Der Resultatwert beider Funktionen ist im Erfolgsfall die in der Komponente *len* angegebene Anzahl Bytes. Dies gilt auch bei der Spezifikation von **0** Bytes. Im Fehlerfall liefert **sendn()** als Resultat **-1**, sofern noch keine Daten gesendet wurden, andernfalls die Anzahl der bis zum Auftreten des Fehlers erfolgreich gesendeten Bytes. Entsprechendes Fehlerverhalten gilt auch für **recvn()** mit der Ausnahme, dass eine ordnungsmäßige Termination der Verbindung durch den Kommunikationspartner als logischer Fehler behandelt wird und den Resultatwert **0** liefert, sofern noch keine Daten empfangen wurden.

3.4 Verbesserte Implementierungen

3.4.1 Zeilenorientierter Echo-Server

Wie in den vorigen Beispielen wird ein zeilenorientierter Echo-Server betrachtet; der Client liest also eine Zeile (Folge von Bytes bis *newline*), schickt diese an den Server und dieser schickt sie wieder zurück. Dazu sollen die beiden Funktionen **sendn**() und **recvn**() verwendet werden. Dabei ist aber zu beachten, dass Zeilen deutlich kürzer sein können als die enstprechenden Puffer – die Flagge **MSG_WAITALL** würde hier zu einer Blockade der Kommunikation führen.

Die Dateien insgesamt:

- **sign.h** und **sign.c** wie bisher
- Das Hauptprogramm des Servers:

```
/* ------------------main_srv.c ---------------------------
 * server using TCP protocol
 *
 * simple error handling
 */

#include "inet.h"
#include "sign.h"
#include "str_echo.h"

int main(int argc, char **argv) {

   int sockfd,newsockfd,clilen,childpid;
   struct sockaddr_in cli_addr, serv_addr;

   if( (ignoresig(SIGCHLD) == SIG_ERR) ) {
       perror("Signal");
       exit(1);
   }
   /*
    * open a TCP socket - Internet stream socket
    */
   if ( (sockfd = socket(AF_INET, SOCK_STREAM,0)) < 0) {
      perror("server: can't open stream socket");
      exit(1);
   }

   /*
    * bind our local address so that the client can send us
    */
   bzero((char *) &serv_addr, sizeof(serv_addr));
   serv_addr.sin_family = AF_INET;

   serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
   /* INADR_ANY: tells the system that we'll accept a connection
    * on any Internet interface on the system, if it is multihomed
```

```
     * Address to accept any incoming messages (-> in.h).
     * INADDR_ANY is defined as ((unsigned long int) 0x00000000)
     */

    serv_addr.sin_port = htons(SERV_TCP_PORT);

    if ( bind(sockfd, (struct sockaddr *)&serv_addr,
          sizeof(serv_addr)) < 0) {
       perror("server: can't bind local address");
       exit(1);
    }

    listen(sockfd,5);

    while(1) {
       /*
        * wait for a connection from a client process
        * - concurrent server -
        */
       clilen=sizeof(cli_addr);
       newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
                &clilen);

       if (newsockfd < 0 ) {
          if (errno == EINTR)
             continue; /*try again*/
          perror("server: accept error");
          exit(1);
       }

       if ( (childpid = fork()) < 0) {
          perror("server: fork error");
          close(sockfd);
          close(newsockfd);
          exit(1);
       }
       else if (childpid == 0) {
          close(sockfd);
          /*****************************/
          str_echo(newsockfd);
          /*****************************/

          exit(0);
       }

       close(newsockfd); /*parent*/
    }
 }
```

- Die Funktionalität des Servers:

```
 /* str_echo.c
  * function used in connection-oriented servers
  * read a stream socket one line at a time,
```

```
 * and write each line back to the sender
 * return when the connection is terminated
 */
# define STR_ECHO_H
# include "str_echo.h"

void str_echo(int sockfd) {
   int n;
   char line[MAXLINE];

   while(1) {
      n = recvn(sockfd,line,MAXLINE);
      if (n == 0)
         return; /*connection terminated*/
      else if ( n < 0 ) {
         perror("str_echo: readline error");
         exit(3);
      }
      /*Ausgabe auf stderr des Servers: */

      if (sendn(sockfd, line, n) != n) {
         perror("str_echo: write error");
         exit(3);
      }
   }
}
```

- Das Hauptprogramm des Client:

```
/* -------------------- main_cli.c -----------------------
 * client using TCP protocol
 */

#include "inet.h"
#include "str_cli.h"
#include "str_echo.h"

int main(){
   int sockfd;
   struct sockaddr_in serv_addr;

   /*
    * fill in the structure "serv_addr" with address
    * of server we want to connect with
    */

   bzero( (char *) &serv_addr, sizeof(serv_addr));
   serv_addr.sin_family = AF_INET;
   serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
   serv_addr.sin_port = htons(SERV_TCP_PORT);

   /*
    * open a TCP socket - internet stream socket
    */
```

```
   if ( (sockfd = socket(AF_INET, SOCK_STREAM,0)) < 0 ) {
      perror("client: can't open stream socket");
      exit(2);
   }

   /*
    * connect to the server
    */

   if (connect(sockfd, (struct sockaddr *) &serv_addr,
         sizeof(serv_addr) ) < 0) {
      perror("client: can't connect to server");
      exit(3);
   }

   /*************************/
   str_cli(stdin, sockfd);
   /*************************/

   close(sockfd);
   printf("\n");
   exit(0);
}
```

- Die Funktionalität des Client:

```
/* str_cli.c
 * function used by connection-oriented clients
 *
 * read the contents of the FILE *fp, write each line to the
 * stream socket (to the server process), then read a line back
 * from the socket and write it to stdout
 *
 * return to caller when an EOF is encountered on the input file
 */

# define STR_CLI_H
# include "str_cli.h"

void str_cli(FILE *fp, int sockfd) {
   int n;
   char sendline[MAXLINE], recvline[MAXLINE+1];

   while (fgets(sendline, MAXLINE,fp) != NULL) {
      n = strlen(sendline);
      if (sendn(sockfd,sendline,n) != n) {
         perror("str_cli: write error on socket");
         exit(1);
      }

      /*
       * now read a line from the socket and
       * write it to stdout
```

```
         */
        n = recvn(sockfd, recvline, n);
        if (n < 0) {
           perror("str_cli: readline error");
           exit(1);
        }
        recvline[n] = '\0';
        printf(">>> ");
        fputs(recvline,stdout);

    }

    if (ferror(fp)) {
        perror("str_cli: error reading file");
        exit(1);
    }
}
```

- Die Funktion **recvn**() — ohne die Flagge **MSG_WAITALL**

```
/* ipc_recv.c */
# define IPC_RECV_H
# include "ipc_recv.h"

ssize_t recvn(int fd, char * buf, size_t len) {

   char *     ptr = buf;
   size_t     nc;
   ssize_t    n;
   int        flags = 0;


   for(nc = len; nc > 0; ptr += n, nc -= n) {
        recv_again:
        if( (n = recv(fd, ptr, nc, flags)) < 0 )
            if(errno == EINTR) {
                errno = 0;
                goto recv_again;
            } else
                return (len -= nc) ? len : -1;
         else  if( n == 0 ) {
               errno = ENOTCONN;
               return (len -= nc) ? len : 0;
         }
         if(buf[n-1] == '\n') return n;
   }
   return len;
}
```

- Ausführung:

```
hypatia$ make -f LinMakefile.srv
```

```
gcc -Wall -c main_srv.c
gcc -Wall -c ipc_send.c
gcc -Wall -c ipc_recv.c
gcc -Wall -c str_echo.c
gcc -Wall -c sign.c
gcc -Wall -o server main_srv.o str_echo.o ipc_send.o ipc_recv.o sign.o
hypatia$ make -f LinMakefile.cli
gcc -Wall -c main_cli.c
gcc -Wall -c str_cli.c
gcc -Wall -o client main_cli.o str_cli.o ipc_recv.o ipc_send.o
hypatia$ server &
[1] 1883
hypatia$ client
eine Zeile
>>> eine Zeile
und noch eine Zeile
>>> und noch eine Zeile
^d
hypatia$
```

3.4.2 Ein „stream“-basierter Echo-Server

Der Client liest eine belibige Folge von Zeichen aus der Standardeingabe, schickt sie an den Server, der schickt sie zurück und der Client kopiert das Ganze an die Standardausgabe. Dazu müssen auf Client-Seite kleinere Änderungen (statt **fgets()** wird **read()** verwendet) vorgenommen werden.

Die Dateien insgesamt:

- **sign.h** und **sign.c** wie bisher
- Das Hauptprogramm des Servers:

```
/* -----------------main_srv.c --------------------------
 * server using TCP protocol
 *
 * simple error handling
 */

#include "inet.h"
#include "sign.h"
#include "str_echo.h"

int main(int argc, char **argv) {

   int sockfd,newsockfd,clilen,childpid;
   struct sockaddr_in cli_addr, serv_addr;

   if( (ignoresig(SIGCHLD) == SIG_ERR) ) {
       perror("Signal");
       exit(1);
   }
   /*
    * open a TCP socket - Internet stream socket
```

```
 */
if ( (sockfd = socket(AF_INET, SOCK_STREAM,0)) < 0) {
   perror("server: can't open stream socket");
   exit(1);
}

/*
 * bind our local address so that the client can send us
 */
bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;

serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
/* INADR_ANY: tells the system that we'll accept a connection
 * on any Internet interface on the system, if it is multihomed
 * Address to accept any incoming messages (-> in.h).
 * INADDR_ANY is defined as ((unsigned long int) 0x00000000)
 */

serv_addr.sin_port = htons(SERV_TCP_PORT);

if ( bind(sockfd, (struct sockaddr *)&serv_addr,
      sizeof(serv_addr)) < 0) {
   perror("server: can't bind local address");
   exit(1);
}

listen(sockfd,5);

while(1) {
   /*
    * wait for a connection from a client process
    * - concurrent server -
    */
   clilen=sizeof(cli_addr);
   newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
            &clilen);

   if (newsockfd < 0 ) {
      if (errno == EINTR)
         continue; /*try again*/
      perror("server: accept error");
      exit(1);
   }

   if ( (childpid = fork()) < 0) {
      perror("server: fork error");
      close(sockfd);
      close(newsockfd);
      exit(1);
   }
   else if (childpid == 0) {
      close(sockfd);
```

```
            /************************/
            str_echo(newsockfd);
            /************************/

            exit(0);
         }

         close(newsockfd); /*parent*/
      }
   }
```

- Die Funktionalität des Servers:

```
/* str_echo.c
 * function used in connection-oriented servers
 * read a stream socket one line at a time,
 * and write each line back to the sender
 * return when the connection is terminated
 */
# define STR_ECHO_H
# include "str_echo.h"

void str_echo(int sockfd) {
   int n;
   char line[MAXLINE];

   while(1) {
      n = recvn(sockfd,line,MAXLINE);
      if (n == 0)
         return; /*connection terminated*/
      else if ( n < 0 ) {
         perror("str_echo: readline error");
         exit(3);
      }
      /*Ausgabe auf stderr des Servers: */

      if (sendn(sockfd, line, n) != n) {
         perror("str_echo: write error");
         exit(3);
      }
   }
}
```

- Das Hauptprogramm des Client:

```
/* -------------------- main_cli.c -----------------------
 * client using TCP protocol
 */

#include "inet.h"
#include "str_cli.h"
#include "str_echo.h"
```

```
int main(){
   int sockfd;
   struct sockaddr_in serv_addr;

   /*
    * fill in the structure "serv_addr" with address
    * of server we want to connect with
    */

   bzero( (char *) &serv_addr, sizeof(serv_addr));
   serv_addr.sin_family = AF_INET;
   serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
   serv_addr.sin_port = htons(SERV_TCP_PORT);

   /*
    * open a TCP socket - internet stream socket
    */

   if ( (sockfd = socket(AF_INET, SOCK_STREAM,0)) < 0 ) {
      perror("client: can't open stream socket");
      exit(2);
   }

   /*
    * connect to the server
    */

   if (connect(sockfd, (struct sockaddr *) &serv_addr,
         sizeof(serv_addr) ) < 0) {
      perror("client: can't connect to server");
      exit(3);
   }

   /***************/
   str_cli(sockfd);
   /***************/

   close(sockfd);
   printf("\n");
   exit(0);
}
```

- Die Funktionalität des Client:

```
/* str_cli.c
 * function used by connection-oriented clients
 *
 * read from stdin (0),  write to
 * stream socket (to the server process), then read back
 * from the socket and write  to stdout
 *
 * return to caller when an EOF is encountered on the input
 */
```

```
# define STR_CLI_H
# include "str_cli.h"

void str_cli(int sockfd) {
   int n;
   char sendline[MAXLINE], recvline[MAXLINE+1];

   for(;;) {
      if( (n = read(0, sendline, MAXLINE)) < 0 ) {
         perror("read");
         exit(1);
      } else if (n == 0)
           return; /*done*/

      if (sendn(sockfd,sendline,n) != n) {
         perror("str_cli: write error on socket");
         exit(2);
      }

      /*
       * now read from the socket and write to stdout
       */
      if( (n = recvn(sockfd, recvline, n)) != n) {
         perror("str_cli: read error on socket");
         exit(3);
      }

      if(write(1,recvline,(size_t) n) < 0) {
         perror("write");
         exit(4);
       }
   }

}
```

- Die Funktion **recvn()** — jetzt mit der Flagge **MSG_WAITALL**

```
/* ipc_recv.c */
# define IPC_RECV_H
# include "ipc_recv.h"

ssize_t recvn(int fd, char * buf, size_t len) {

   char *    ptr = buf;
   size_t    nc;
   ssize_t   n;
   int       flags = 0;

# ifdef MSG_MSG_WAITALL
  flags |= MSG_WAITALL
# endif

   for(nc = len; nc > 0; ptr += n, nc -= n) {
```

```
        recv_again:
        if( (n = recv(fd, ptr, nc, flags)) < 0 )
            if(errno == EINTR) {
                errno = 0;
                goto recv_again;
            } else
                return (len -= nc) ? len : -1;
         else  if( n == 0 ) {
              errno = ENOTCONN;
              return (len -= nc) ? len : 0;
         }
         if(buf[n-1] == '\n') return n;
    }
    return len;
}
```

- Ausführung:

```
hypatia$ make -f LinMakefile.srv
gcc -Wall -c main_srv.c
gcc -Wall -c ipc_send.c
gcc -Wall -c ipc_recv.c
gcc -Wall -c str_echo.c
gcc -Wall -c sign.c
gcc -Wall -o server main_srv.o str_echo.o ipc_send.o ipc_recv.o sign.o
hypatia$ make -f LinMakefile.cli
gcc -Wall -c main_cli.c
gcc -Wall -c str_cli.c
gcc -Wall -o client main_cli.o str_cli.o ipc_recv.o ipc_send.o
hypatia$ server &
[1] 1997
hypatia$ client < ipc_recv.c
/* ipc_recv.c */
# define IPC_RECV_H
# include "ipc_recv.h"

ssize_t recvn(int fd, char * buf, size_t len) {

   char *    ptr = buf;
   size_t    nc;
   ssize_t   n;
   int       flags = 0;

# ifdef MSG_MSG_WAITALL
  flags |= MSG_WAITALL
# endif

   for(nc = len; nc > 0; ptr += n, nc -= n) {
        recv_again:
        if( (n = recv(fd, ptr, nc, flags)) < 0 )
            if(errno == EINTR) {
                errno = 0;
                goto recv_again;
            } else
```

```
                    return (len -= nc) ? len : -1;
            else  if( n == 0 ) {
                  errno = ENOTCONN;
                  return (len -= nc) ? len : 0;
            }
            if(buf[n-1] == '\n') return n;
      }
      return len;
}

hypatia$
```

Index