Internet-Technologie

Prof. Jürgen Plate

1 Grundlagen Netzwerkprogrammierung

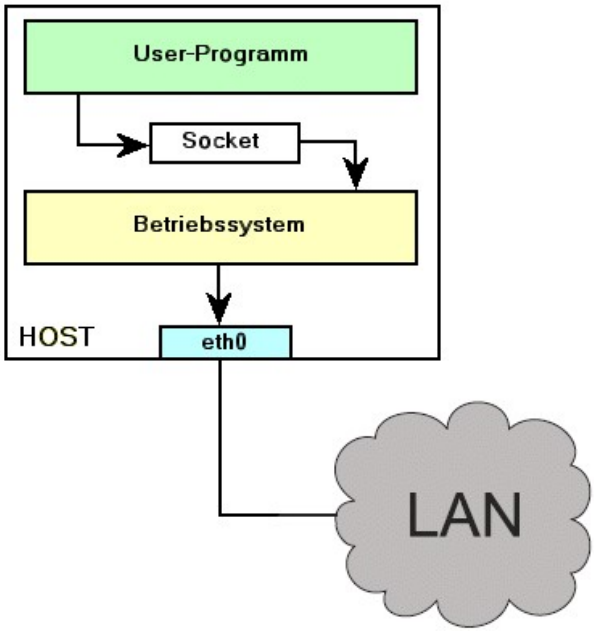
Internetanwendungen mit den TCP/IP-Protokollen werden nach dem Client-Server-Prinzip erstellt. Der Client ist hier in den meisten Fällen das Benutzer-Interface und nimmt vom Server bestimmte Dienste in Anspruch. Er baut in Abhängigkeit vorher definierter Ereignisse (z. B. dem Starten einer Internetanwendung durch einen Anwender) die Verbindung zum Server auf und ist somit der aktive Teil.
Der Server stellt nun den gewünschten Dienst zur Verfügung. Er muß sich ständig in einem Zustand befinden, in dem er Verbindungsaufforderungen von Clients entgegennehmen kann - er ist der passive Teil. Ein Server darf niemals einen Dienst vom Client anfordern.
Client und Server müssen die gleiche Sprache sprechen: Sie müssen sich also an einem gemeinsamen Protokoll orientieren. Das unterschiedliche Verhalten von Client und Server läßt allerdings eine Asymetrie entstehen, die sich in der Verwendung unterschiedlicher Schnittstellen-Befehle bei der Realisierung einer Client- oder Serverapplikation niederschlägt.

TCP/IP wird nicht nur verwendet, um mit anderen Rechnern Kontakt aufzunehmen. Bei Systemen, die nicht über einen Netzwerkanschluß vewrfügen, wird eine Schnittstelle gebraucht, die eine Netzwerkschnittselle emuliert. Dies ist das loopback-Interface, weil es wie eine Schleife auf den Rechner selbst zurückführt. Notwendig ist das loopback-Interface, weil verschiedene Dienste (X-Window, Drucksysteme etc.) netwerkorientiert arbeiten. Für die Progammierung ist es ideal, weil Client und Server auf demselben Rechner laufen können.

In diesem Skript wird gleich mit der Programmierung losgelegt. Grundlagen über TCP/IP, Nameserver usw. finden Sie im Netzwerk-Skript.

1.1 Die Socket-Netzwerkschnittstelle

Was sind Sockets? Ein Socket ist eine Schnittstelle zwischen einem Prozess und einem Transportprotokoll, z. B. TCP oder UDP. Das Socket-Prinzip entspricht dem von File-Deskriptoren. Dort bildet nach dem Öffnen einer Datei das Handle die Verbindung zu dieser Datei und mittels Referenz auf das Handle sind Lese- oder Schreibzugriff möglich. Bei Sockets geht es nicht um Dateien sondern um Kommunikationskanäle, über die Daten gesendet und empfangen werden können.



Die Socket-Schnittstelle ist damit die Grundlage der Programmierung verteilter Anwendungen unter TCP/IP. Ein Socket ("Steckdose") ist ein also Verbindungsendpunkt, der vom Programm wie eine gewöhnliche Datei beschrieben und gelesen werden kann. Anfang der 80er Jahre wurde mit 4.2 BSD in UNIX-Systemen die sogenannte "Socket"-Schnittstelle, das "Socket Application Programming

Interface" (Socket API) für die Kommunikation zwischen Prozessen eingeführt. Es basiert nach wie vor auf dem Berkley Socket API, das für die Programmiersprache C entwickelt wurde. Ein "Socket" ist dabei der Name für einen Endpunkt einer Kommunikationsverbindung. Seine Schnittstelle ist im wesentlichen konzipiert für:

- verbindungsorientierte Kommunikation, aufsetzend auf TCP (`SOCK_STREAM`)
- verbindungslose Kommunikation, aufsetzend auf UDP (`SOCK_DGRAM`)
- direkten Zugriff auf die IP-Schicht (`SOCK_RAW`)
- Kommunikation zwischen lokalen Prozessen (`AF_UNIX`)
- Kommunikation zwischen im TCP/IP-Netz verteilten Prozessen (`AF_INET`)
- Kommunikation für andere Protokoll- und Adressierungsfamilien

Dabei gibt es normalerweise ein Programm, das Anfragen von anderen Programmen entgegennimmt und sie beantwortet (ein sog. Server-Socket) und beliebig viele andere Programme, die ihre Anfragen an das Server-Socket schicken und mit den Antworten weiterarbeiten (die sog. Client-Sockets). Das ganze System ist auch bekannt als Client-Server-Programmierung. Ein sehr typisches Beispiel sind Webserver: der Webserver wartet auf Anfragen von Browsern (oder User Agents o.ä.) und gibt Webseiten zurück. Die Browser arbeiten dann mit diesen Webseiten weiter, indem sie sie anzeigen und Operationen darauf erlauben (Anzeigen des Quelltextes etc.).

In der Berkley Socket API wird der Socket als Datei aufgefasst. Er wird somit im Betriebssystem über einen Filediscriptor identifiziert und auf ähnliche Weise wie reguläre Dateien genutzt. Neben den grundlegenden Funktionen für Lesen und Scheiben, dem Empfangen und Senden von Daten, stellt der Socket die Funktionalitäten der im Rechner verfügbaren Transportschichten bereit. Nach der Initialisierungsphase, in der Adressen, Protokolle und Optionen festgelegt werden kann die Kommunikation sogar über die klassischen IO-Funktionen `read()` und `write()` erfolgen. Die von einem Benutzerprogramm in einen Socket geschriebenen Daten werden vom Betriebssystem mit Headerdaten versehen und über das Netz gesendet. Analog werden über das Netz empfangene Daten vom Betriebssystem ohne Headerdaten im zugehörigen Socket abgelegt. Von dort können sie vom Benutzerprogramm ausgelesen werden. Folglich operiert ein Benutzerprogramm stets nur auf den Nutzdaten (Payload) der Netzwerkkommunikation.

Die Socket-Schnittstelle ist zwar von keiner Institution genormt, stellt aber den Industriestandard dar. Wichtige Gründe sind u.a.:

- leicht verständliches Konzept
- leicht zu handhaben und zu programmieren
- fügt sich harmonisch in die UNIX-Welt ein
- verfügbar für viele Systeme, z.B. die Windows Sockets, WinSock von Microsoft und anderen Firmen aus der PC-Welt definiert und kompatibel zu den UNIX-Sockets.

Um mit einem bestimmten Dienst (Programm) auf einem anderen Rechner zu kommunizieren, reicht es allerdings nicht, einfach den anderen Rechner als solchen anzusprechen. Es ist vielmehr jedes Server-Programm auf einer bestimmten Port-Nummer zu erreichen. Jeder Client muß dem entfernten Rechner diese Port-Nummer mitteilen, damit dieser die Anfrage dem richtigen Programm zuleiten kann.

Die Netz-Ein- und Ausgabe wurde an die Datei-Ein- und Ausgabe angelehnt und etliche Ein- und Ausgabe-Systemaufrufe lassen sich auf Dateien und Sockets anwenden. Es gibt jedoch einige Unterschiede:

- Die typische Client-Server-Beziehung ist unsymmetrisch. Zur Einrichtung einer Netzverbindung muß ein Programm seine Rolle kennen (Client oder Server).
- Eine Netzverbindung kann verbindungsorientiert oder verbindungslos sein. Der erste Fall ähnelt einer Datei-E/A, der zweite Fall kennt kein "open" oder dergleichen.
- Namen spielen bei Netzverbindungen eine größere Rolle. So kann ein Kindprozeß einen vom Elternprozeß übergebenen Dateideskriptor verwenden, ohne den Dateinamen zu kennen.
- Es gibt mehr Parameter zur Spezifizierung einer Netzverbindung als bei der Date-E/A. Eine Verbindung wird durch fünf Parameter beschrieben:
  1. Protokoll
  2. lokale Adresse
  3. lokaler Prozeß
  4. ferne Adresse
  5. ferner Prozeß

  Das Protokoll ist an eine Portnummer der TCP/IP-Verbindung gekoppelt.
- Das Netz-Interface muß mehrere Kommunikationsprotokolle unterstützen und daher allgemeingütiger gehalten werden.
- Der Serverprozess muss von außen gezielt angesprochen werden können. Dazu wird er an einen festen Socket, den sogenannten Port, gebunden, über den er erreichbar ist.
- Der Client braucht keinen festen Port. Er holt sich beim Programmstart einen normalen Socket, dem vom System eine freie Portnummer zugeteilt wird. Der Server erfährt die Nummer des Clients aus der Anfrage und kann ihm unter diesem Port antworten.

Verbindungsorientierte und verbindungslose Kommunikation

- Die "verbindungsorientierte Kommunikation" geht davon aus, daß vor dem eigentlichen Datentransfer vom Client-Prozess eine logische Verbindung aufgebaut wird, über die dann in der Folge beliebig viele Datenpakete hin und her geschickt werden können - so lange bis einer der beiden Partner die Verbindung wieder abbricht ("hangup"). Bei dieser Art der Kommunikation wird Empfänger- und Absenderadresse nur beim Verbindungsaufbau angegeben - danach erfolgt die Kommunikation über Verbindungsnummern, die vom Betriebssystem zugeteilt werden. Die Telefonie funktioniert nach diesem Prinzip.
  Diese Form der Kommunikation eignet sich besser für Anwendungen, bei denen große Datenmengen übertragen werden. Der Verbindungsaufbau dauert zwar etwas, die eigentlichen Daten werden allerdings vollständig und in der richtigen Reihenfolge von der Quelle zum Ziel übermittelt.
- Bei der "verbindungslosen Kommunikation" gibt es - wie der Name sagt - keine bestehende Verbindung zwischen den Partnern. Die Datenpakete werden einzeln mit den vollständigen Adressangaben versehen und auf den Weg gebracht. Im täglichen Leben entspricht dies der Kommunikation mittels Briefen.
  Bei diesem Modell - man spricht dabei auch von "Datagrammen" - wird vom Betriebssystem die korrekte Reihenfolge der einzelnen Datenpakete nicht garantiert. Es ist noch nicht mal gewährleistet, daß ein Paket überhaupt ankommt. Da jedoch der zeitaufwendige Verbindungsaufbau entfällt, ist es in Fällen, bei denen es mehr auf Schnelligkeit als auf Sicherheit ankommt, oft die bessere Wahl. Eventuell nötige Reihenfolge-Überprüfungen bzw. Zeitüberwachngen müssen dann aber vom Anwendungsprozess selbst vorgenommen werden.

Sockets

Zur Kommunikation zwischen Prozessen, die auch auf verschiedenen Rechnern ablaufen können, wurde mit den Sockets im BSD-Unix ein leistungsfähiger Mechanismus der Datenübertragung definiert. Sockets sind heute Grundlage der meisten höheren Datenübertragungsprotokolle und in fast allen Betriebssystemen realisiert. Sie stellen die Schnittstelle zwischen Anwendungsprogramm und den Betriebssystemroutinen zur Datenkommunikation dar. Dabei besteht der Vorteil für den Benutzer darin, daß einem Socket ein Dateideskriptor zugeordnet wird, über den das Anwendungsprogramm fast genauso kommunizieren kann, wie über Pipes oder normale Dateien. Im Gegensatz zu einer Pipe, die grundsätzlich nur in einer Richtung betrieben werden kann, ist ein Socket-Deskriptor jedoch bidirektional - wie eine zum Lesen und Schreiben geöffnete Datei. Sockets sind, wie die Client-Server-Beziehung, unsymmetrisch: Einer der beiden beteiligten Prozesse ist "Server", der andere "Client". Der Server (Diensterbringer) wartet darauf, daß irgendein Client (Kunde) mit ihm Kontakt aufnehmen möchte. Der Client ist der aktive Part und veranlasst den Beginn der Kommunikation.

Über Sockets kann der Datenaustausch auf zweierlei Art erfolgen:

1. Datenströme (Streams): Zwischen Client und Server wird eine Verbindung aufgebaut, die einzelnen Datenpakete werden gesichert und in korrekter Reihenfolge übertragen und zum Schluß wird die Verbindung wieder abgebaut. Dies entspricht dem Zyklus "open" - "read"/"write" - "close" bei einer normalen Datei. Bei einer Verbindung über IP wird dafür TCP benutzt.
2. Einzelpakete (Datagrams): Datagramme werden gleichsam als "Pakete" mit Absender- und Empfängeradresse verschickt. Das entsprechende Internet-Protokoll heißt UDP. Es wird keine Verbindung zwischen den beiden Prozessen aufgebaut, weshalb UDP wesentlich schneller ist. Allerdings gibt es keine Garantie für das Ankommen des Paketes bei der Gegenseite und keine Gewähr für die Einhaltung der richtigen Reihenfolge.

Die Konstanten der BSD-Library wären SOCK_STREAM bzw. SOCK_DGRAM. Daneben werden Raw Sockets (SOCK RAW)verwendet, um auf die Netzwerkebene direkt zuzugreifen.

Sockets sind noch über verschiedenen "Domänen" definiert: Es gibt neben der "Internet-Domäne" noch weitere Domänen, z. B. die "Unix-Domäne" für die Kommunikation zwischen reinen Unix-Prozessen. So kennen die meisten Systeme:

```
/* Supported address families. */
#define AF_UNIX       1   /* Unix domain sockets */
#define AF_LOCAL      1   /* POSIX name for AF_UNIX */
#define AF_INET       2   /* Internet IP Protocol */
#define AF_IPX        4   /* Novell IPX */
#define AF_APPLETALK  5   /* AppleTalk DDP */
#define AF_INET6     10   /* IP version 6 */
#define AF_IRDA      23   /* IRDA sockets */
#define AF_BLUETOOTH 31   /* Bluetooth sockets */
```

Thema dieses Skripts ist aber ausschließlich die Internet-Domäne. Zur Übergabe der Parameter einer Socketverbindung dienen zwei Strukturen, *sockaddr* und für die Internet-Protokolle *sockaddr_in*. Erstere ist so konzipiert, dass sie unabhängig von der verwendeten Adressfamilie ist:

```
struct sockaddr {
  sa_family_t sa_family;         /* address family, AF_xxx       */
  char sa_data[14];              /* 14 bytes of protocol address */
  };
```

Für Intermet-Anwendungen existiert mit *sockaddr_in* eine spezielle Struktur, die es erlaubt, IP-Adresse und Portnummer getrennt

einzutragen. Im Speicher sind diese beiden Strukturen kompatibel, es reicht also eine einfache Typumwandlung, um die gewünschten Informationen zu übergeben:

```
struct sockaddr_in {
  sa_family_t sin_family;        /* Address family                    */
  unsigned short int sin_port;   /* Port number                       */
  struct in_addr sin_addr;       /* Internet address                  */
  unsigned char pad[8];          /* Pad to size of `struct sockaddr'. */
  };
```

Alle Zahlenwerte müssen in *Network Byteorder* (Big-Endian) vorliegen. Ein Beispiel für eine Zuweisung an eine Variable *my_addr* vom Typ *struct sockaddr_in* könnte lauten:

```
my_addr.sin_family = AF_INET;
/* Umwandlung in Network Byteorder mit htons() */
my_addr.sin_port = htons(25);
/* Umwandeln in 32-Bit-Zahl in Network-Byteorder */
my_addr.sin_addr.s_addr = inet_addr("10.27.210.232");
```

Die Systemaufrufe im Überblick

Einige wichtige Socket-Primitive bzw. -Systemcalls sind:

- *Erstellen eines Socket:*
  `s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)` zum Eröffnen einer verbindungsorientierten Kommunikation im Internet
- *Verbindungsaufbau:*
  `bind(s, address, addresslength)` zum Verbinden des Sockets mit der lokalen Endpunktadresse
  `cs = connect(s, address, addresslength)` zum Verbindungsaufbau über einen virtuellen Kanal zum entfernten Prozess
- *Kommunikation:*
  `nwrite = write(s, buffer, nbytes)` (bzw. `send(...)`) und
  `nread = read(s, buffer, maxbytes)` (bzw. `recv(...)` zum Schreiben bzw. Lesen auf/vom virtuellen Kanal
- *Verbindungsabbau:*
  `close(s)`

Die folgende Tabelle gibt einen Überblick über die wichtigsten Socket-Systemcalls in Client- und Serverprogrammen:

Phase	**Client**	**Server**
Endpunkt erzeugen	socket()	socket()
Binden einer Adresse	bind()	bind()
Verbindung aufbauen	connect()	
Warteschlange festlegen		listen()
Warten auf Verbindung		accept()
Daten senden	write() send() sendto() sendmsg()	write() send() sendto() sendmsg()
Daten empfangen	read() recv() recvfrom() recvmsg()	read() recv() recvfrom() recvmsg()
Verbindung schließen	shutdown()	shutdown()
Endpunkt abbauen	close()	close()
Ereignisse annehmen	select()	select()
Verschiedenes	getpeername() getsockname() getsockopt() setsockopt()	getpeername() getsockname() getsockopt() setsockopt()

Für die Kommunikation bei verbindungslosen, d.h. UDP-basierten Socketanwendungen sind die speziellen `send()`- und `receive()`-Systemcalls empfehlenswert, während bei TCP-Verbindugen daneben die Standard-Systemcalls `read()` und `write()` einsetzbar sind.

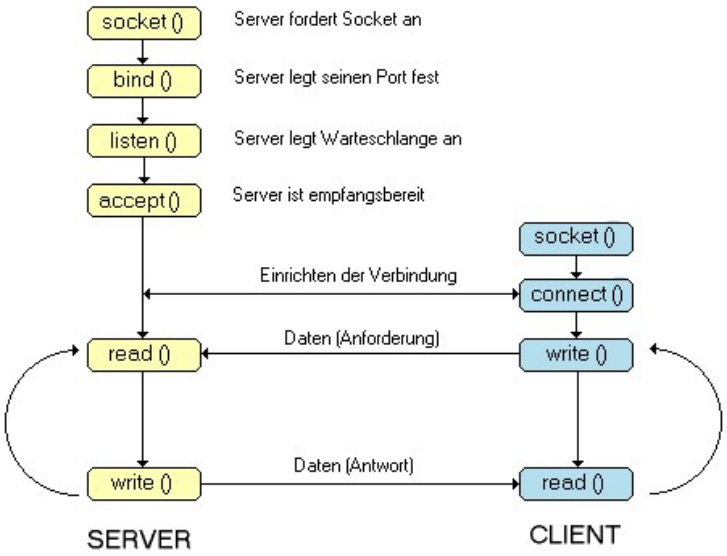
Eine TCP/IP-Verbindung ist, wie wir gesehen haben, durch eine Client-Server-Architektur geprägt und damit asymmetrisch. Vor der Kommunikation muß die Verbindung stehen. Das betrifft einmal die Verbindung zwischen den Rechnern, als auch jene zwischen den Prozessen. Die Adressierung der Rechner erfolgt per Hostname, der vom System auf die IP-Nummer umgesetzt wird.

Vom Client aus muß nicht nur der richtige Rechner, sondern auch der richtige Serverprozeß angesprochen werden können. Dazu bindet sich der Serverprozeß an einen festen Port, über den er erreichbar ist. Damit die Nummer des Ports mit einem Namen versehen werden kann, verwendet man die Datei `/etc/services`. Im Programm wird die Servicenummer durch den Aufruf der Systemfunktion `getservbyname()` bestimmt.

Für bekannte Dienste werden bestimmte Portnummern von der IANA (Internet Assigned Number Authority) festgelegt. Portnummern sind in drei Kategorien eingeteilt:

- 1 - 1023: "Wohlbekannte" (well known) Portnummern für Standardanwendungen (z. B. 22 für SCP, 80 für HTTP, 25 für SMTP usw.). Sie können meist nur von Superuser- bzw. privilegierten Prozessen benutzt werden.
- 1024 - 49151: Registrierte Portnummern, die von der IANA nicht kontrolliert, aber registriert und gelistet werden (z. B. Ports für X Window-Server).
- 49152 - 65535: Dynamische oder private (kurzlebige) Portnummern. Diese werden vom System automatisch an Benutzerprozesse vergeben, die nicht auf eine bestimmte Portnummer angewiesen sind (oftmals der Client).

Der Client braucht normalerweise keinen festen Port. Er erbittet sich auf der lokalen Maschine eine freie Nummer und ruft damit den Port des Servers. Der Server erfährt die Nummer des Clients aus der Anfrage und kann ihm unter diesem Port antworten. Das Szenario zwischen Server und Client sieht wie folgt aus:



Betrachten wir beispielhaft einmal das Listing eines ganz einfachen Servers in der Programmiersprache C. Die einzelnen Systemaufrufe werden weiter unten genauer behandelt, das Listing soll zunächst nur einen Überblick des Ablaufs geben:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <netdb.h>

#define MAXPUF 1023

main()
  {
  int MySocket, ForeignSocket;
  struct sockaddr_in AdrMySock, AdrPartnerSocket;
  struct servent *Service;
  int AdrLen;

  char Puffer[MAXPUF];
  int MsgLen;

  /* Socket einrichten */
  MySocket = socket(AF_INET, SOCK_STREAM, 0);

  /* Socket an Port-Nummer binden */
  memset(&AdrMySock, 0, sizeof (AdrMySock));
```

```
  AdrMySock.sin_family = AF_INET;         /* Internet-Protokolle */
  AdrMySock.sin_addr.s_addr = INADDR_ANY; /* akzept. jeden Client-Host */
  Service = getservbyname("echo","tcp");   /* bestimme Port */
  AdrMySock.sin_port = Service->s_port;   /* (Get Service by Name) */
  bind(MySocket, &AdrMySock, sizeof(AdrMySock));

  /* Empfangsbereitschaft signalisieren und warten */
  listen(MySocket, 5);

  for (;;) /* forever */
    {
    /* Verbindungswunsch vom Client annehmen */
    ForeignSocket = accept(MySocket, &AdrPartnerSocket, &AdrLen);
    /* Datenaustausch zwischen Server und Client */
    MsgLen = recv(ForeignSocket, Puffer, MAXPUF, 0); /* String empfangen */
    send(ForeignSocket, Puffer, MsgLen, 0); /* und zuruecksenden */
    /* Verbindung beenden und wieder auf Client warten */
    close(ForeignSocket);
    }
  }
```

Dieser Server bearbeitet jede Anfrage, die über den Port "echo" an ihn gestellt wird. Nach jeder Anfrage wird die Verbindung wieder gelöst und ein anderer Client kann anfragen. Ein solcher Server dürfte auf jedem Betriebssystem arbeiten können, das TCP/IP unterstützt, selbst wenn es kein Multitasking beherrscht.

Es gibt zwei Variablen pro Socket. Die eine ist wie bei Dateizugriffen ein einfaches Handle (`MySocket`), die andere hält die Adresse der Verbindung (`AdrSock`), also die IP-Nummer des Rechners und die Nummer des verwendeten Ports. Der Server erlaubt Verbindungen von jedem Rechner aus, weil die Konstante `INADDR_ANY` benutzt wird.

Der zugehörige Client gibt dagegen die Adresse des anzusprechenden Servers an. Das Programm sieht wie folgt aus:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <netdb.h>

#define MAXPUF 1023

main()
  {
  int MySocket;               /* Socket-Handle */
  struct sockaddr_in AdrSock; /* Socketstruktur */
  int len;                    /* Die Laenge der Socketstruktur */

  struct hostent *RechnerID;  /* ferner Rechner */
  struct servent *Service;    /* Dienst auf dem fernen Rechner */

  char Puffer[MAXPUF] = "Wir erschrecken zu guten Zwecken!";

  MySocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
  memset(&AdrSock, 0, sizeof (AdrSock));

  /* Bestimme den Zielrechner */
  RechnerID = gethostbyname("server");
  bcopy(RechnerID->h_addr,&AdrSock.sin_addr.s_addr,RechnerID->h_length);

  /* Bestimme den Port */
  Service = getservbyname("echo","tcp");
  AdrSock.sin_port = Service->s_port;

  connect(MySocket, (struct sockaddr *)&AdrSock, sizeof(AdrSock));
  send(MySocket, Puffer, MAXPUF, 0);  /* String senden */
  recv(MySocket, Puffer, MAXPUF, 0);  /* und wieder empfangen */
  printf("%s\n", Puffer);             /* ausgeben */
  close(MySocket);
  }
```

Die `recv`-Funktion liefert als Rückgabewert die Größe des versandten Speicherbereichs (max. 1 KByte, siehe unten).

Grundsätzlich liefern fast alle Netz-Funktionen bei Fehlern den Wert 0 zurück. In den obigen Beispiel-Listing fehlt jegliche Fehlerbehandlung, damit das Prinzip übersichtlich dargestellt werden kann. Im "richtigen" Programm ist eine umfassende Fehlerbehandlung unumgänglich.

Das Server-Programm hat noch einen Nachteil. Nach dem Start des Servers ist die Konsole oder das Shell-Fenster für weitere Zwecke blockiert. Auch fehlt eine korrekte Möglichkeit, den Server zu beenden. Bei einem Abbruch des Programms wird es dem Betriebssystem überlassen, den Socket zu schließen. Zweckmässigerweise wird vom Programm ein Dämon erzeugt, der per `fork`-Aufruf in den

Hintergrund gestellt.

Server und Client können auch unterschiedliche Prozessor-Architekturen haben. Die Speicherung von Zahlen können als Big-Endian oder als Little-Endian erfolgen. Um aus einer lokal verwendeten Byte-Reihenfolge (Host Byte Order) eine Network-Byte-Order-Reihenfolge oder umgekehrt zu erstellen, stehen die vier Funktionen `htonl()`, `htons()`, `ntohl()` und `ntohs()` zur Verfügung (siehe auch Zahlenformat: ntoh und hton).

1.2 Behandlung von Signalen und Timeouts

Mitunter kommt es vor, dass eine Netzverbindund unterbrochen wird. In solchen Fällen stellt sich die Frage, wie das Programm verfahren soll. Wie lange soll auf Daten von der Gegenstation gewartet werden? Wie oft soll ein Verbindungsaufbau wiederholt werden, wenn der Kontakt nicht zustande kommt? Wie kann mit anderen Prozessen kommuniziert werden? Und so weiter?

Grundlage dieses Themenkreises bildet die **Einführung der C-Systemaufrufe zur Behandlung von Prozessen**. Dort können Sie alles genau nachlesen. Hier soll nur ein Beispiel für die Programmierung eines Signal-Handlers für die Taste [Strg]-[C] und die Alarm-Funktion gezeigt werden:

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

void propeller(void)
  {
  /* Aktivitaetsanzeige in Spalte 1 der aktuellen Zeile */
  static char sign[] = {'|','/','-','\\'};
  static int pos = 0;
  putchar('\r');
  putchar(sign[pos]);
  fflush(stdout);        /* sonst sieht man nix */
  pos = (pos + 1) % 4;
  }

void tick (int dummy)    /* den Wecker wieder aufziehen */
  { alarm(1); }

void beenden(int dummy)  /* STRG-C behandeln */
  {
  printf("\nHasta la vista, baby!\n");
  exit(0);
  }

int main(void)
  {
  /* Signalhandler aktivieren */
  signal(SIGINT,beenden);
  signal(SIGALRM,tick);

  alarm(1);  /* Wecker aufziehen */
  for (;;)
    {
    pause();     /* auf Signal warten */
    propeller(); /* Anzeigen, dass sich was tut */
    };
  return(0);     /* never reached */
  }
```

So kann z. B. beim Drücken von [Strg]-[C] das Programm alle Dateien und Sockets ordentlich schließen bevor es beendet wird.

Für die Anwendung bei Timeouts würde man beispielsweise beim Empfang eines Blocks jedesmal den Wecker für einen geeigneten Zeitraum aufziehen. Der Signalhandler wird nicht aktiviert, solange alles wunschgemäß läuft. Tritt ein längerer Timeout auf, sorgt der Signalhandler dann für ein geordnetes Ende des Programms.

1.3 TCP/IP-Sockets: Die Funktionen

Da die Programmierung von Client und Server primär in Perl stattfinden wird, erfolgt die Vorstellung der Systemfunktionen recht kurz. Die Ähnlichkeit der später besprochenen Perl-Funktionen und -Methoden mit den C-Systemaufrufen ist jedoch nicht rein zufällig.

Mit Sockets läßt sich der Austausch von Nachrichten zwischen Prozessen verbindungsorientiert oder mit Datagrammen recht einfach programmieren. Durch Angabe eines Sockettyps wird die Art der Kommunikation festgelegt: SOCK STREAM: verbindungsorientiert, SOCK DGRAM: Datagramm (daneben gibt es noch weitere Typen). Die kommunizierenden Prozesse können auf demselben Rechner

ablaufen oder auf vernetzten Maschinen. Die Programmierschnittstelle unterstützt verschiedene Protokollfamilien und, daran gekoppelt, verschiedene Adressierungsarten. Ein Beispiel für eine Adressfamilie ist AF_UNIX. Sie definiert einen Adressierungsmechanismus für die rechnerinterne Kommunikation zwischen UNIX-Prozessen. Als Adressobjekte werden Pfade im Dateisystem verwendet, genau wie bei Pipes. Welche Adressfamilien unterstützt werden, hängt davon ab, welche Netzwerkprotokolle das Betriebssystem beherrscht. UNIX-Systeme werden zumindest AF_UNIX und AF_INET unterstützen.

Kommunikationsendpunkt: socket

Um mit Sockets zu arbeiten, muß zuerst eine Verbindung geöffnet werden. Hier gibt es Analogien zu Dateizugriffen. Als erstes muß also ein Socket vom Betriebssystem angefordert werden. Dies geschieht mit dem `socket()`-Systemaufruf. Der Aufruf entspricht einem `fopen` bei Dateien. Die Funktion `socket()` hat drei Parameter:

```
int socket(int Family, int Sockettype, int Protocol);
```

- `Family` legt die Protokoll-Familie fest:
  - `AF_UNIX` : UNIX-interne Protokolle
  - `AF_INET` : Internet-Protokolle
  - `AF_NS` : Xerox-NS-Protokolle
  - `AF_IMPLINK`: IMP-Link-Schicht

  Wir werden nur mit `AF_INET` zu tun haben.

- `Sockettype` legt den Typ des Sockets (und damit auch teilweise das Protokoll) fest (in Klammern die Anwendung für die Familie `AF_INET`):
  - `SOCK_STREAM` : Stream-Socket (TCP)
  - `SOCK_DGRAM` : Datagramm-Socket (UDP)
  - `SOCK_RAW` : Raw-Socket (IP)
  - `SOCK_SEQPACKET`: Paket-Socket
  - `SOCK_RDM` : Nachrichten-Socket

- `Protocol` legt das genaue Protokoll fest. Für `AF_INET` sind die möglichen Werte:
  - `IPPROTO_UDP` : UDP-Protokoll (`SOCK_DGRAM`)
  - `IPPROTO_TCP` : TCP-Protokoll (`SOCK_STREAM`)
  - `IPPROTO_ICMP`: ICMP-Protokoll (`SOCK_RAW`)
  - `IPPROTO_RAW` : Raw-IP-Protokoll (`SOCK_RAW`)

Der `socket()`-Systemaufruf liefert einen Integerwert zurück, der einem Dateidescriptor ähnelt. Dieser Wert wird daher "Socketdeskriptor" oder "sockfd" genannt. Im Fehlerfall hat er den Wert -1. Beispiel:

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

int MySocket, ForeignSocket;
...

MySocket = socket(AF_INET, SOCK_STREAM, IPPPROTO_TCP);
...

close(MySocket);
```

Nach Aufruf von "socket" ist der Socket jedoch noch nicht betriebsbereit. Es muss jetzt noch festgelegt werden, für welchen Port (d.h. für welches Protokoll der Anwendungsebene) der Socket zuständig sein soll, ob es sich um einen Server- oder Client-Socket handeln soll, etc.

Jeder eröffnete Socket muß auch wieder geschlossen werden. Eine Nachlässigkeit an dieser Stelle kann sich bitter rächen, da insbesondere bei Serverprozessen Verbindungen sehr oft eröffnet werden und die Systemresourcen für Netzverbindungen irgendwann zur Neige gehen, was meist zum Stillstand des Servers führt. Das Schließen des Sockets erfolgt unter UNIX mit dem `close`-Aufruf.

Socket einrichten: bind

Der Serverprozess muß von außen erreichbar sein. Dazu bekommt er einen sogenannten *well known port*. Diese Nummer ist also den Clientprozessen bekannt. Um einen Socket an diese Nummer zu binden, wird der `bind`-Aufruf verwendet. Als Parameter verwendet `bind` den Socket und eine Struktur `sockaddr_in`, die diesen Port beschreibt.

```
int bind (int sockfd, struct sockaddr *Myaddr, int Addrlen);
```

Mit dem Aufruf wird ein Speicherbereich bereitgestellt, der zur Festlegung der Protokoll-Familie und der Portnummer vorgesehen ist. Bei einem Server-Socket erfolgt damit die Zuordnung zu dem gewünschten Port - er erklärt sich damit zuständig für ein bestimmtes Anwender-Protokoll. Der Aufruf von "bind" ist sowohl bei Datenströmen als auch bei Datagrammen erforderlich. Der Parameter

"sockfd" ist ein Dateideskriptor, der mit einem vorangegangenen "socket"-Aufruf erzeugt wurde. Der zweite Parameter ist ein Zeiger auf eine protokollspezifische Adresse und der dritte Parameter gibt die Größe der Adreßstruktur an. `bind` wird in drei Fällen angewendet:

1. Server registrieren ihre eigene Adresse innerhalb des Systems.
2. Ein Client kann eine spezifische Adresse selbst speichern.
3. Ein verbindungsloser Client muß vom System eine individuelle Adresse anfordern, damit er eine gültige Adresse für die Rückantwort hat.

`bind` füllt also im oben angeführten Fünfertupel die Felder "lokale Adresse" und "lokaler Prozeß".

`struct sockaddr` ist eine allgemeingültige Datenstruktur, die für verschieden Protokollfamilien existiert. Bei Verwendung der Internet-Protokollfamilie kann sie vom Anwenderprogramm überlagert werden durch eine Struktur `sockaddr_in`, die ausschließlich für IP geeignet ist. Unter der Annahme, daß der Anwender eine Variable vom Typ "sockaddr_in" in der Form `struct sockaddr_in adresse;` deklariert hat, enthält "adresse" u. a. die folgenden Komponenten:

```
adresse.sin_family      /* vorzeichenlose 16bit-Ganzzahl (Protokoll-Familie) */
adresse.sin_port        /* vorzeichenlose 16bit-Ganzzahl (Portnummer) */
adresse.sin_addr.s_addr /* vorzeichenlose 32bit-Ganzzahl (Internetadresse) */
```

In die Komponente `sin_family` wird die Konstante AF_INET (2) eingetragen. In die Komponente `sin_port` ist die Portnummer einzutragen - allerdings in sog. "Netzwerk-Anordnung": Portnummer und Internetadresse sind Zahlen, die über das Netz verschickt werden und demnach unabhängig von der internen Zahlendarstellung des jeweiligen Rechners sein müssen (siehe später).

```
struct sockaddr_in adresse;

adresse.sin_family = AF_INET; /* Internet-Protokoll-Familie */
adresse.sin_port = htons(80); /* Port festlegen */
adresse.sin_addr.s_addr = 0;  /* Internetadresse irrelevant */

int ergebnis = bind(descriptor,(struct sockaddr *)&adresse,sizeof(adresse));
```

Der "Typecast"-Operator `(struct sockaddr *)` ist erforderlich, wenn der Compiler auf strenge Typprüfung eingestellt ist. Die Funktion `bind()` erwartet ja einen Zeiger vom Typ `struct sockaddr *`. Beispiel:

```
   .
   .
   .
  s = socket(AF_INET, SOCK_STREAM, 0);
  if (s < 0)
    {
    fprintf(stderr, "Error: Socket\n");
    return -1;
    }

  my_addr.sin_family = AF_INET;
  my_addr.sin_port = htons(4711); /* Portnummer */
  my_addr.sin_addr.s_addr = INADDR_ANY; /* An jedem Device warten */

  if (bind(s, (struct sockaddr *)&my_addr, sizeof(my_addr)) < 0)
    {
    close(s);
    fprintf(stderr, "Error: bind\n");
    return -1;
    }
   .
   .
   .
```

Warteschlange festlegen: listen

Der `listen`-Aufruf gibt an, wieviele Anfragen gepuffert werden können. In fast allen Programmen wird hier ein Wert von 5 für *backlog* verwendet (der derzeitige Höchstwert).

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

`listen` folgt normalerweise nach `socket` und `bind` und unmittelbar vor `accept`.

Falls die `listen()`-Warteschlange voll ist, werden weitere Verbindungswünsche von Clients abgewiesen. Ein Server für Datagramme (UDP) braucht `listen()` nicht aufzurufen, da er keine Verbindungen zu Clients einrichtet.

Verbindungswunsch entgegennehmen: accept

Der `accept`-Aufruf wartet auf eine Anfrage eines Clients. Der Aufruf von `accept()` liefert als Rückgabewert die Socket-ID des Partners.

Des weiteren wird per Parameter in einer Variablen der Struktur `sockaddr_in` die Adresse des Partners geliefert.

```
int accept(int sockfd, struct sockaddr_in *Peer, int *Addrlen);
```

`accept` nimmt die erste Anforderung von der Warteschlange und generiert einen weiteren Socket mit der gleichen Eigenschaft wie `sockfd`. Der Parameter `Peer` verweist auf einen Speicherbereich, dessen Inhalt beim Aufruf undefiniert sein kann. In diesen trägt `accept()` die Internetadresse des Absenders eines eintreffenden Verbindungswunsches ein (die Adresse des Clients). Auf diese Datenstruktur kann genauso zugegriffen werden, wie dies bereits bei `bind()` erklärt wurde. `Peer` und `Addrlen` liefern also die Felder "ferne Adresse" und "ferner Prozeß" des Fünfertupels. Der Parameter "Addrlen" ist die Adresse eines Variablen-Parameters: Vor dem Aufruf muß dort die maximale Länge des Speicherbereiches stehen, auf den der Parameter `Peer` zeigt. Nach dem Aufruf enthält er die Anzahl Bytes, die das Betriebssystem tatsächlich dort eingetragen hat.

`accept` generiert (bei einem concurrent server) automatisch einen neuen Socketdescriptor für die aktuelle Verbindung. Der Rückgabewert von `accept()` ist also ein neuer Dateideskriptor, über den in der Folge die Kommunikation mit dem Client erfolgt (z.B. mit `read()` und `write()`). Der als erster Parameter angegebene Deskriptor bleibt für weitere Verbindungswünsche reserviert. Zum Beispiel:

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

int MySocket, ForeignSocket, Partnerlen;
struct sockaddr_in AdrMySock, AdrPartnerSocket;
...

  MySocket = socket(AF_INET, SOCK_STREAM, IPPPROTO_TCP);
...

  AdrMySock.sin_family = AF_INET;
  AdrMySock.sin_addr.s_addr = INADDR_ANY; /* akzept. jeden */
  AdrMySock.sin_port = PortNr;  /* wird per getservbyname bestimmt */
  bind(MySocket, &AdrMySock, sizeof(AdrMySock));
  listen(MySock, 5);
  for(;;)
    {
    ForeignSocket = accept(MySocket, &AdrPartnerSocket, &Partnerlen);
    ...
    close(ForeignSocket);
    }
...
```

Hinweise: Nicht vergessen: `ForeignSocket` muß nach Ende der Kommunikation geschlossen werden, sonst gehen dem System nach einiger Zeit die Sockets aus.

`accept` ist eine blockierende Funktion, die den aufrufenden Server-Prozess so lange blockiert, bis eine Verbindung vorhanden ist. Ändert man die Eigenschaften des Socket-Deskriptors auf nicht-blockierend, gibt `accept` einen Fehler zurück, wenn beim Aufruf keine Verbindungen vorhanden sind.

Clientaufruf: connect

Sobald der Server läuft, kann der Client Verbindung zum *well known port* des Servers aufnehmen. Der entsprechende Aufruf lautet `connect`.

```
int connect(int sockfd, struct sockaddr_in *ServAddr, int Addrlen);
```

Der Parameter `sockfd` ist natürlich wieder der Socket-Deskriptor. Die weiteren Parameter entsprechen jenen von `bind`. Für die meisten verbindungsorientierten Protokolle richtet `connect` eine Verbindung vom lokalen zum fernen Rechner ein.
Allerdings muß diesmal in der Datenstruktur, auf die `ServAddr` zeigt, die Internetadresse des gewünschten Servers eingetragen werden. Die Verbindung erfolgt zu dem angegebenen Rechner. Weiterhin ist ein Port anzugeben. Der Ziel-Server wird durch seine IP-Nummer festgelegt. Diese steht in der Struktur `sockaddr_in` im Element `sin_addr`. Beispiel:

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

struct sockaddr_in AdrSock;
...
  AdrSock.sin_family = AF_INET;
  AdrSock.sin_addr = HostID;
  AdrSock.sin_port = htons(PortNr);
  connect(MySocket, (struct sockaddr *)&AdrSock, sizeof(AdrSock));
...
```

`HostID` stellt eine 32-Bit-Ganzzahl dar. Im Normalfall liegt die Host-Adresse natürlich nicht Ganzzahl vor, sondern als Zeichenkette der Form "`www.netzmafia.de`" oder "`192.168.234.77`". Zu korrekten Umwandlung in eine ganze Zahl, die in Netzwerk-Anordnung

vorliegen muss, stehen in der C-Bibliothek zwei Routinen zur Verfügung, `gethostbyname()` und `inet_addr()`, die weiter unten besprochen werden.

Das folgende Beispiel zeigt den kompletten Verbindungsaufbau eines Clients:

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

struct sockaddr_in server;

/* Deklaration des Sockets */
int descr = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);

/* Datenstruktur "server" vorbereiten */
server.sin_family = AF_INET;    /* Internet-Protokoll-Familie */
server.sin_port = htons(80);    /* Port 80 festlegen (HTTP) */

/* 1.Versuch die Internetadresse zu ermitteln, Form: "a.b.c.d" */
server.sin_addr.s_addr = inet_addr(server_name);
if (server.sin_addr.s_addr == -1) /* keine korr. Punktnotation */
  {
  /* 2. Versuch: symbolisch */
  struct hostent *host = gethostbyname(server_name);
  if (host != NULL)
    {
    server.sin_addr.s_addr = *((unsigned long*)host->h_addr_list[0]);
    }
  else
    {
    printf("Internetadresse nicht gefunden\n");
    exit(1);
    }
  }

/* jetzt kann verbunden werden */
int i = connect(descr, (struct sockaddr *)&server, sizeof(server));
if (i == 0)
  { /* jetzt steht die Verbindung! */
  ...
  }
```

Datenaustausch: send, sendto, recv und recvfrom

Mit diesen Aufrufen werden Daten über die bestehenden Verbindungen transportiert. Unter UNIX könnten dafür auch die Dateiaufrufe `read` und `write` verwendet werden.

```
int send(int sockfd, char *Buffer, int NBytes, int Flags);

int sendto(int sockfd, char *Buffer, int NBytes, int Flags,
           struct sockaddr_in *To, int AddrLen);

int recv(int sockfd, char *Buffer, int NBytes, int Flags);

int recvfrom(int sockfd, char *Buffer, int NBytes, int Flags,
           struct sockaddr_in *From, int AddrLen);
```

Die ersten drei Parameter dieser vier Systemaufrunfe sind den ersten drei Parametern von `read` und `write` ähnlich. Der Parameter `sockfd` identifiziert wieder den gewünschten Socket, `Buffer` ist ein Zeiger auf einen beliebigen Speicherpuffer, `NBytes` bestimmt die Anzahl der zu übertragenden Bytes und `Flags` hat im Normalfall den Wert Null oder er stellt das das Resultat einer Oder-Verknüpfung mit einer der folgenden Konstanten dar:

- `MSG_OOB` : Sende/empfange Out-of-Band-Daten
- `MSG_PEEK` : Peek auf ankommende Nachricht (`recv, recvfrom`)
- `MSG_DONTROUTE`: Leite Routing um (`send, sendto`)

Wird `Flags` beispielsweise auf den Wert 1 gesetzt (MSG_OOB), dann soll die Übertragung "out of band" erfolgen. Bei dieser Übertragung werden nach Möglichkeit bisher bereits abgeschickte Daten überholt. Es handelt sich dann beispielsweise um hochpriore Informationen, wie beispielsweise das Abbruchsignal Crtl-C beim Telnet-Protokoll.

Alle Funktionen liefern als Rückgabewert die Größe der empfangenen bzw. gesendeten Datenmenge. Die `recv`-Funktion liefert die Sendung in Blöcken von maximal 1 KByte Größe. Wurden größere Pakete verschickt, müssen sie stückweise gelesen werden. Das Senden ist nicht beschränkt. Da der Rückgabewert nichts über die Grösse des tatsächlich gesendeten Pakets aussagt, muß dies vom Programm geregelt werden. Wenn die Pakete nicht immer gleiche Größe besitzen, wird die Paketlänge meist in den ersten Bytes des

ersten Paketes kodiert.

Für den Normalfall (Flags gleich Null) kann statt `send()` auch die Systemfunktion `write` verwendet werden. Zum Beispiel kann statt

```
send(sock,"Hello World",11,0);
```

auch wie folgt programmiert werden:

```
write(sock,"Hello World",11);
```

Darüberhinaus besteht natürlich die Möglichkeit, eine Datei für Standard-Ein/Ausgabe über dem betreffenden Deskriptor zu definieren:

```
FILE *f = fdopen(descr,"rw");
```

wodurch nun auch mit Routinen der "stdio"-Bibliothek auf den Socket zugegriffen werden kann, z. B.:

```
fprintf(f,"Hello World");
```

Dies ist insbesondere wichtig, wenn die Standardeingabe oder Standardausgabe eines beliebigen Programmes auf einen Socket umgeleitet werden soll.

Der Rückgabewert von `recv()` gibt Auskunft über die tatsächliche Anzahl empfangener Bytes. Ist dieser Wert -1, handelt es sich um einen Fehler, beim Wert 0 wurde die Verbindung von der Gegenseite geschlossen. Andernfalls ist der Wert immer größer 0 und kleiner gleich dem Parameter `NBytes`.
Bei Verwendung des Flags `MSG_PEEK` während des Empfangs werden die Daten zwar zum Anwenderprogramm übertragen, sie verbleiben jedoch auch noch in der Empfangswarteschlange, so daß sie mit einem nachfolgenden `recv()`-Aufruf nochmals gelesen werden können.
Ruft der Empfänger die `recv()`-Funktion mit `NBytes > 0` auf und stehen im Empfangspuffer bereits Daten bereit (aber weniger als erwartet - beispielsweise weil der Rest noch nicht angekommen ist), dann kehrt die Funktion trotzdem sofort zurück und übergibt die tatsächliche Anzahl der übertragenen Bytes. Erfordert es die Logik des Anwenderprogrammes, daß vor einer Fortsetzung die Gesamtzahl der erwarteten Bytes eingetroffen ist, so muß der Aufruf von `recv()` so lange wiederholt werden, bis alle Daten eingetroffen sind. Die Daten müssen vom Empfänger in geeigneter Form zusammengesetzt werden.
Wird `recv()` mit `Flags = 0` aufgerufen, kann stattdessen die Systemfunktion `read()` verwendet werden.

Die Funktionen `sendto()` und `revcfrom()` dienen dazu, Daten auf einer UDP-Verbindung zu senden und zu empfangen. Die ersten vier Parameter haben die gleiche Bedeutung wie bei `send()` und `recv()`. Beim vorletzten Parameter von `sendto()` muss man einen Pointer auf eine Variable vom Typ `struct sockaddr_in`, in der festgehalten ist, wohin das Paket genau gesendet werden soll. Der letzte Parameter gibt die genaue Länge der vorangegangenen Variablen an. In der Funktion `recvfrom()` dient der Parameter `From` als Platzhalter, in den bei einem empfangenen Paket Informationen über den Sender gespeichert und an das Programm zurückgegeben werden.

Wenn Sie für eine UDP-Verbindung vorher `connect()` aufgerufen haben, so können Sie einfach `send()` und `recv()` verwenden.

Socket schließen: close

Eine bidirektionale Socket-Verbindung kann mit dem Aufruf

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

geschlossen werden. Dabei legt der Parameter `how` fest, ob künftig keine Daten mehr empfangen werden sollen (how=0), keine mehr gesendet werden (how=1), oder beides (how=2). Wird statt `shutdown()` die Systemfunktion `close()` benutzt, dann entspricht dies einem `shutdown(sock,2)`.

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

Zahlenformat: ntoh und hton

Portnummer und Internetadresse sind Zahlen, die über das Netz verschickt werden und demnach unabhängig von der internen Zahlendarstellung des jeweiligen Rechners sein müssen. Die Reihenfolge der Bytes eines Datenwortes ist auf den verschiedenen Computern unterschiedlich definiert. So besteht eine Variable vom Typ `short` aus zwei Byte. Auf einer Maschine mit Intel-Architektur kommt dabei das niederwerte Byte zuerst ("little endian"), während es auf einem 68000-Prozessor oder einer Sun genau umgekehrt ist. Aus diesem Grund wurde eine eindeutige Netzwerk-Anordnung der zu übertragenden Bytes definiert (höherwertige Bytes zuerst!).

Bei Rechnerarchitekturen, bei denen der Speicher nach der Host-Order ausgewertet wird (das niederwertige Byte also vor dem höherwertigen im Speicher steht), ist es notwendig, alle Werte mit dem Type LONG oder WORD vor der Übergabe an den Treiber in die Network-Order zu konvertieren. Um Zahlen der Maschine in die passende Form für das Netz zu bringen und die Programme portabel zu halten, gibt es die Makros `ntoh()` (Net to Host) und `hton()` (Host to Net). Beide wirken auf `short`-Variablen. Für `long`-Variablen gibt es die analog funktionierenden Makros `htonl()` und `ntohl()`. Vorsicht ist auch bei Vergleichen geboten: Sie liefern in Network- und Host-

Order nicht das gleiche Ergebnis!

- `unsigned long int htonl(unsigned long int hostlong);` wandelt eine lange Ganzzahl (32bit) von der Rechner-Anordnung ("host order") in die Netzwerk-Anordnung um.
- `unsigned short int htons(unsigned short int hostshort);` wandelt eine kurze Ganzzahl (16bit) von der Rechner-Anordnung ("host order") in die Netzwerk-Anordnung um.
- `unsigned long int ntohl(unsigned long int netlong);` wandelt eine lange Ganzzahl (32bit) von der Netzwerk-Anordnung ("host order") in die Rechner-Anordnung um.
- `unsigned short int ntohs(unsigned short int netshort);` wandelt einekurze Ganzzahl (16bit) von der Netzwerk-Anordnung ("host order") in die Rechner-Anordnung um.

Speicher-adresse	Little-Endian (Host-Order)	Big-Endian (Net-Order)	Little-Endian (Host-Order)	Big-Endian (Net-Order)
n+3			31 ... 24	7 ... 0
n+2			23 ... 16	15 ... 8
n+1	15 ... 8	7 ... 0	15 ... 8	23 ... 16
n	7 ... 0	15 ... 8	7 ... 0	31 ... 24
	16 Bit WORD		32-Bit DWORD	

Um beispielsweise den Port des POP3-Dienstes (110) numerisch an die Struktur `sock_add_in` zu übergeben, würde man `hton` verwenden (eigentlich sollte man dazu `getservbyname` verwenden):

```
struct sockaddr_in AdrSock;
...
   AdrSock.sin_port = hton(110);
...
```

Das Socket-Interface stellt hier eine Reihe von Konvertierungs-Funktionen zur Verfügung. Neben den einfache Funktionen für die richtige Byte-Order gibt es auch Konvertierung für die IP-Adressen. Oft werden auch Funktionen für das Bearbeiten von Rohdaten (Byte-Arrays) benötigt, die im folgenden beahndelt werden.

Byte-Operationen

In den verschiedenen Socket-Adreßstrukturen existieren unterschiedliche Byte-Felder, die alle behandelt werden müssen. Einige dieser Felder sind, wie auch immer, keine C-Integer-Felder, so daß hier andere Techniken angewandt werden müssen, um mit ihnen allen gleich operieren zu können. BSD definiert die folgenden drei Routinen, die auf benutzerdefinierten Byte-Strings basieren. Darunter ist zu verstehen, daß es sich um keine Standard-Strings in C handelt, die bekanntermaßen mit einem Nullbyte abgeschlossen werden, sondern die benutzerdefinierten Byte-Strings können innerhalb des Strings durchaus Nullbytes besitzen. Deshalb muß die Länge des Strings den Funktionen als Parameter mitgegeben werden.

```
bcopy (char *Src, char *Dest, int NBytes);
```

Kopiert NBytes vom Ursprung (SRC) zum Ziel (Dest). Achtung: Parameterreihenfolge anders als bei `strcpy`.

```
bzero (char *Dest, int NBytes);
```

Schreibt NBytes Null-Bytes an das angegebene Ziel.

```
int bcmp (char *Ptrl, char *Ptr2, int NBytes);
```

vergleicht zwei Byte-Strings. der Rückgabewert ist gleich Null, wenn beide Byte-Strings gleich sind, sonst ungleich Null (also auch anders als bei `strcmp`).

Die "b..."-Funktionen haben ihre Wurzeln beim BSD-UNIX und werden bald nicht mehr überall verfügbar sein (auch wenn sie in der Literatur noch häufig anzutreffen sind. An ihre Stelle treten die ANSI-Funktionen:

Die Funktion `memcpy` kopiert n Bytes aus dem Puffer Src in den Puffer Dst. Die Funktion gibt die Anfangsadresse von Dst zurück.

```
void *memcpy(void *Dst, const void *Src, size_t n);
```

Die folgende Funktion füllt die ersten n Bytes der Adresse Ptr mit dem Wert byt auf.

```
void *memset(void *Ptr, int byt, unsigned int n);
```

Mit `memcmp` werden die ersten n Bytes im Puffer Ptr1 mit dem Puffer Ptr2 lexikografisch verglichen. Der Rückgabewert ist derselbe wie bei `strcmp`. Ptr1 > Ptr2 → Rückgabewert < 0, Ptr1 < Ptr2 → Rückgabewert > 0 und bei Gleichheit beider Speicherbereiche wird 0

zurückgegeben.

```
int memcmp( const void *Ptr1, const void *Ptr2, size_t n);
```

Konvertierung von IP-Adressen

Für die folgenden Funktionen werden diverse Headerfiles benötigt:

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

Mit den Funktionen `inet_aton()` und `inet_addr()` konvertieren Sie die String-Darstellung der IP-Adresse ("dottet quad", z. B. "192.168.0.1") in eine 32-Bit-Internet-Adresse in Netzwerk-Byteorder. Die in der Literatur noch oft zu findende Funktion `inet_addr()` ist veraltet und wird hier nicht mehr behandelt. Die Syntax zu `inet_aton()`:

```
int inet_aton(const char *ptr, struct in_addr *inp);
```

Hiermit wird die IP-Adressen-Zeichenkette `ptr` eine 32-Bit-Adresse konvertiert. Der Wert dieser 32-Bit-Adresse befindet sich anschließend in `in_addr` der Strukture `struct sockaddr_in`. Zur Erinnerung hier nochmals die Strukturen:

```
struct sockaddr_in {
  /* Adressfamilie normalerweise AF_INET  */
  short int          sin_family;
  /* Port der Verbindung                  */
  unsigned short int sin_port;
  /* Adresse, zu der verbunden werden soll */
  struct in_addr     sin_addr;
  /* Fuelldaten, um auf 14 Bytes zu kommen */
  unsigned char      sin_zero[8];
  };

struct in_addr {
  unsigned long int sin_addr;
  }
```

`in_addr` ist in der Headerdatei `<netinet/in.h>` definiert.

Bei einem Fehler gibt die Funktion `inet_aton()` den Wert 0, bei Erfolg einen Wert ungleich 0 zurück.

Für die Zukunft wichtig ist die Funktion `inet_pton()`, die im Gegensatz zu `inet_aton()` nicht nur IPv4-Adressen umwandelt:

```
int inet_pton(int af, const char *src, void *dst);
```

Für `af` wird die Adressfamilie (ähnlich wie bei `socket()`) angegeben, `src` enthält die Adresse als String und `dst` ist ein Zeiger auf die Zielstruktur, die je nach Adressfamilie variiert. Zum Beispiel:

```
  struct in_addr addr;
...
  inet_pton(AF_INET, "127.0.0.1", &addr);
```

Soll dagegen aus einer 32-Bit-Darstellung der IP-Adresse wieder ein "dotted quad" String entstehen, steht die Funktion `inet_ntoa()` zur Verfügung:

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

Damit wird die übergebene 32-Bit-Adresse `ip`, die als Network-Byteorder vorliegen muss, in einen String konvertiert. Der String wird als Rückgabewert der Funktionen in einem statischen Puffer abgelegt. Bei einem Fehler wird `NULL` zurückgegeben.

Die Alternative ist das Gegenstück zu `inet_pton()`, `inet_ntop()`:

```
const char *inet_ntop( int af, const void *src, char *dst, socklen_t cnt );
```

`af` gibt wieder die Adressfamilie an, `src` enthält die 32-Bit-Darstellung der IP-Adresse und `dst` ist ein Zeiger auf einen String, in dem der konvertierte Wert von `cnt` Bytes länge kopiert wird. Zum Beispiel:

```
  char buf[16];
  ...
  inet_ntop(AF_INET, &addr, buf, 16);
```

Das folgende Beispiel zeigt die Anwendung von `inet_aton()` und `inet_ntoa()`:

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

int main (int argc, char **argv)
  {
  char *ip_addr;
  struct in_addr ip;
  char *ip_str;

  if (argc != 2)
     {
     printf ("Usage: %s IP-Adresse\n", argv[0]);
     return -1;
     }
  ip_addr = argv[1];
  if (inet_aton (ip_addr, &ip) == 0) return -1;
  printf ("IP-Adresse (argv[1])      : %s\n", ip_addr);
  printf ("IP-Adresse als 32-Bit Wert: %08X\n", ip.s_addr);
  ip_str = inet_ntoa (ip);
  if (ip_str == NULL) return -1;
  printf ("IP-Adresse als String     : %s\n", ip_str);
  return 0;
  }
```

Um aus einer IP-Adresse in Form eines Strings die Netzwerkadresse zu extrahieren, dient die Funktion `inet_network()`:

```
in_addr_t inet_network(const char * ptr);
```

Bei der Netzadresse wird bekanntlich der Hostanteil zu 0. Bei Erfolg gibt die Funktion die Netzadresse in Host-Byteorder zurück. Bei einem Fehler wird -1 zurückgegeben.

Um aus einer nummerischen 32-Bit-IP-Adresse eine Netzadresse zu erhalten, wird die Funktion `inet_netof()` verwendet:

```
in_addr_t inet_netof(struct in_addr in);
```

Aus der nummerischen 32-Bit-Darstellung der IP-Adresse, die als Parameter angegeben wird, wird die Netzadresse extrahiert. Diese Adresse wird bei Erfolg als Rückgabewert von `inet_netof()` in Host-Byteorder zurückgegeben. Bei einem Fehler wird -1 zurückgegeben.

Um den Host-Anteil einer nummerischen 32-Bit-IP-Adresse zu ermitteln, steht die Funktion `inet_lnaof()` zur Verfügung:

```
in_addr_t inet_lnaof(struct in_addr in);
```

Damit ermitteln Sie die Adresse des Host-Anteils (z. B. ist bei 192.168.0.23 der Host-Anteil 0.0.0.23). Die Funktion gibt diesen Wert in Form eines 32-Bit-Wertes in Host-Byteorder zurück. Bei Fehler wird -1 zurückgegeben.

Mit der Funktion `inet_makeaddr()` wird aus der Host- und Netzadresse eine IP-Adresse zusammengesetzt.

```
struct in_addr inet_makeaddr(int net, int host);
```

Damit wird aus der Netzadresse `net` und der Host-Adresse `host` eine vollständige 32-Bit-IP-Adresse erzeugt und zurückgeliefert. Die Werte `net` und `host` müssen in Host-Byteorder übergeben werden.

1.4 Namensauflösung

Computer und Dienste werden unter TCP/IP immer über die IP-Nummern angesprochen. Für den Menschen ist jedoch ein (Domain-)Name bequemer. Allerdings gibt es für beides Mechanismen zur Namensauflösung. Im Programm ruft man entsprechende Funktionen auf. Genaueres dazu finden Sie im Netzwerk-Skript.

Normalerweise sind der gewünschte Dienst und der Name des Hosts bekannt, der bezüglich des Dienstes angesprochen werden soll. Daher zuerst ein Blick auf den Host.

```
#include <netdb.h>
...

struct hostent *gethostbyname (char *hostname);
...
```

Die `gethostbyname`-Funktion gibt einen Zeiger auf eine `hostent`-Struktur zurück:

```
struct hostent
  {
  char *h_name;       /* official name of host */
  char **h_aliases;   /* alias list */
  int h_addrtype;     /* host address type */
  int h_length;       /* length of address */
  char **h_addr_list; /* a NULL terminates the list */
  };

#define h_addr h_addr_list[0]; /* first address in list */
```

Gegenwärtig enthält das Feld `h_addrtype` immer den Wert `A_INET` und analog das Feld `h_length` immer den Wert 4 (ist gleich der Länge der Internet-Adressse). Bei Internet-Adressen besteht die Matrix der Zeiger `h_addr_list[0], h_addr_list [1], ...` nicht aus Zeigern auf Zeichen, sondern aus Zeigern auf Strukturen vom Typ `in_addr`. Die `hostent`-Struktur ist sehr allgemein gehalten, wobei momentan vieles davon noch nicht verwendet wird.
Das wichtigste Element der `hostent`-Struktur ist das Feld `h_addr_list`, das in einem Array die IP-Nummer des Rechners enthält. Das Makro `h_addr` liefert die Nummer, wie sie in früheren Versionen üblich war. Das Feld `h_length` liefert die Größe einer IP-Nummer.
Ein Host kann mehr als einen Namen tragen, denn ein universell einsetzbarer Host kann mehr als eine Internet-Schnittstelle besitzen, jede mit einer eindeutigen IP-Adresse. Das folgende Beispiel zeigt die Verwendung der `gethostbyname`-Funktion.

```
/* Print the "hostent" information for every host whose name is
 * specified on the command line. (nach Stevens)
 */
#include <stdio.h>
#include <sys/types.h>
#include <netdb.h>       /* for struct hostent */
#include <sys/socket.h>  /* for AF-INET */
#include <netinet/in.h>  /* for struct in_addr */
#include <arpa/inet.h>   /* for inet_ntoa() */

void pr_inet(char **listptr, int length);

int main(int argc, char **argv)
  {
  char *ptr;
  struct hostent *hostptr;

  while (--argc > 0)
    {
    ptr = *++argv;
    if ((hostptr = gethostbyname(ptr)) == NULL)
      {
      printf("gethostbyname error for host %s\n",ptr);
      continue;
      }
    printf ("official host name: %s\n", hostptr->h_name);
    /* go through the list of aliases */
    while ((ptr = *(hostptr->h_aliases)) != NULL)
      {
      printf("    alias: %s\n", ptr);
      hostptr->h_aliases++;
      }
    printf("    addr type = %d, addr length = %d\n",
           hostptr->h_addrtype, hostptr->h_length);
    switch (hostptr->h_addrtype)
      {
      case AF_INET: pr_inet(hostptr->h_addr_list, hostptr->h_length);
                    break;
      default:      printf("unknown address type\n");
                    break;
      }
    }
  return 0;
  }

void pr_inet(char **listptr, int length)
/* Go through a list of internet addresses,
   printing each one in dotted-decimal notation. */
  {
  struct in_addr *ptr;
  while ( (ptr = (struct in_addr *) *listptr++) != NULL)
    printf (" Internet address: %s\n", inet_ntoa(*ptr));
  }
```

Es gibt auch den Fall, daß ein Server die Internet-Adresse des Clients weiß, aber dessen Namen wissen möchte. Die Funktion `gethostbyaddr` erledigt in diesem Fall die Konvertierung von Adresse zu Namen:

```
#include <netdb.h>
...

struct hostent *gethostbyaddr (char *Addr, int Len, int Type);
...
```

Der `Addr`-Parameter ist ein Zeiger auf eine `sockaddr_in`-Struktur, welche die Internet-Adresse enthält. `Len` ist die Größe dieser Struktur. `Type` muß mit `AF_INET` angegeben werden. Ähnlich wie bei der `gethostbyname`-Funktion gibt es auch hier viel Allgemeingültiges, von dem jedoch nicht viel verwendet wird.

Den eigenen Hostnamen erhät man mit der Funktion `gethostname()`, die zwei Parameter besitzt:

```
int gethostname(char *hostname, int len);
```

Der Parameter `hostname` nimmt den nullterminierten Hostnamen auf. `len` spezifiziert dabei die maximale Länge des char-Arrays. Ist der Hostname länger, wird er auf `len` Zeichen gekappt. In diesem Fall kann es sein, dass kein Nullbyte als Terminierung vorhanden ist. Kann der Hostname nicht ermittelt werden, gibt die Funktion -1 zurück, sonst 0.

Um die eigene IP-Adresse zu ermitteln, kann man `hostname()` und `gethostbyname()` kombinieren:

```
#define INTERFACE eth0

...

char *GetLocalIP(char IPaddress[])
  {
  /* Mike Niedermayr */
  int sock_fd;
  struct ifreq ifr;

  sock_fd = socket(AF_INET, SOCK_DGRAM, 0);
  if (sock_fd == -1)
    {
    sprintf(IPaddress, "unknown");
    return IPaddress;
    }
  else
    {
    strcpy(ifr.ifr_name, "INTERFACE");
    if (ioctl(sock_fd, SIOCGIFADDR, &ifr) == -1)
      {
      sprintf(IPaddress, "unknown");
      }
    else
      {
      sprintf(IPaddress, "%s", inet_ntoa(((struct sockaddr_in *) (&ifr.ifr_addr))->sin_addr));
      }
    close(sock_fd);
    return IPaddress;
    }
  }
```

Die Funktion `getservbyname` sucht nach einem Dienst - letztendlich nach einem Port:

```
#include <netdb.h>
...
struct servent *getservbyname(char *Servicename, char *Protname);

...
```

Diese Funktion gibt einen Zeiger auf folgende Struktur zurück:

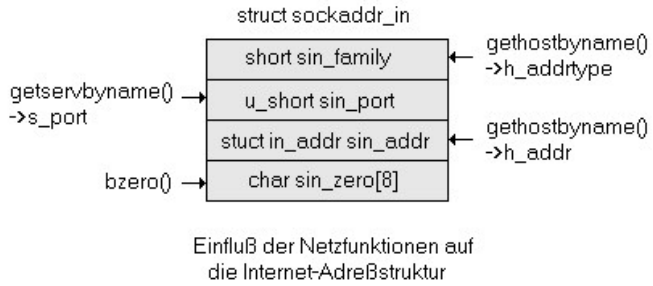
```
struct servent
  {
  char *s_name;      /* official service name */
  char **s_aliases;  /* alias list */
  int s_port;        /* port number, network byte order */
  char *s_proto;     /* protocol to use */
  }
```

Die Information für diese Funktion wird der Datei `/etc/services` entnommen. In dieser Datei wird eine Suche nach dem geforderten Service (`Servicename`) gestartet. Ist auch ein Protokoll angegeben (d. h. `Protname != NULL`), dann muß der entsprechende Eintrag für dieses Protokoll in der Datei vorliegen. Es gibt einige Internet-Dienste, die entweder von TCP oder UDP unterstützt werden (z. B. der Echodienst), und andere, die nur ein Protokoll unterstützen (FTP erfordert beispielsweise TCP). Das Hauptaugenmerk innerhalb der `servent`-Struktur liegt auf der Internet-Portnummer. Zu beachten ist, daß diese Struktur Integer-Portnummern handhaben kann, sogar Intenet-Portnummern in 16 bit-Größe. Beispiel:

```
struct hostent *RechnerID;
struct servent *Service;
...

  RechnerID = gethostbyname("server");     /* Bestimme den Rechner */
  Service = getservbyname("echo","tcp");   /* Bestimme den Port */
...
```

Das wichtigste Element der `servent`-Struktur ist das Feld `s_port`. es enthält die Nummer des Ports, wie sie von der Funktion `connect` verwendet wird.



Einfluß der Netzfunktionen auf die Internet-Adreßstruktur

Nichtprivilegierte Programme (d. h. Programme ohne Root-Rechte) dürfen keine Server-Sockets auf Ports kleiner 1024 öffnen. So wird ein minimaler Schutz davor gewährleistet, daß irgend welche Programme normaler Anwender Ports kidnappen oder auf Ports eigene Services hochfahren, die die Maschine normalerweise nicht bieten würde. Andererseits ist es aus Sicherheitsaspekten nicht sinnvoll, wenn alle Serverprozesse mit root-Privilegien laufen. Die Lösung des Problems ist einfach: sobald man die Server-Sockets gebunden hat, kann man mit `setreuid(2)` die Sonderprivilegien gegen "normale" Userprivilegien tauschen. Alternativ kann man Beispielsweise sicherheitsrelevante setuid-Programme in einem `chroot(2)`-Gefängnis ablaufen lassen, oder das Programm in zwei Prozesse aufteilen, so daß nicht alles mit root-Rechten laufen muß.

Wenn man kurz nachdem ein Programm eine Server-Socket geschlossen hat versucht, einen neuen Socket an denselben Port wie den alten Server-Socket zu binden, erhält man einen "Address already in use"-Fehler. Der Grund dafür ist, daß möglicherweise im Netz noch Pakete herumgeistern, die für den alten Socket bestimmt sind und es deshalb sinnvoll ist, erst einmal zu warten, bis sich das Netz beruhigt hat. Wenn man eine Socket sofort an einen Port binden will, verwendet man die "Reuse"-Option.

1.5 Programmbeispiele

Jetzt sind alle Werkzeuge für das Programmieren von TCP-Servern und -Clients vorhanden. Die folgenden Beispiele verzichten teilweise auf Fehlerbehandlung, damit der eigentliche Programmfluss deutlicher zutage tritt. Für produktive Anwendungen sind sie ohne Fehlerbehandlung etc. nicht geeignet. Alle Beispielprogramme können Sie auch direkt als C-Quelldateien herunterladen.

Für viele Versuche steht per Default bei jedem Linux-System ein Programm zur Verfügung, der gute, alte Telnet-Client. Als ersten Parameter wird der Rechnername (z. B. localhost) und als zweiter Parameter der Port angegeben. Aber man kann natürlich seinen speziellen Client selbst programmieren.

Zum Übersetzen unter Linux genügt folgende Programmzeile:

```
gcc -Wall -o <Binärdatei> <Quelldatei.c>
```

Zu beachten ist noch, dass die Reihenfolge bei den Include-Anweisungen eine Rolle spielt, so muss z. B. `#include <sys/types.h>` - sofern verwendet - vor `#include <sys/socket.h>` stehen.

Demo-TCP-Server

Der erste TCP-Server ist ganz einfach gehalten und zeigt nur die Abfolge der einzelnen Funktionen. Das Programm enthält auch eine grundlegende Fehlerbehandlung. Nach den üblichen Vereinbarungen wird ein Socket angelegt und an den Port 7777 gebunden. Danach wird die Rarteschlange für Client-Requests eingerichtet (`listen()`). Nun geht der Server schlafen und wartet auf eine Anforderung durch den Client (`accept()`). Sobald sich ein Client meldet, wird das unvermeidliche "Hello World" gesendet und der Server beendet.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

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

/* Port fuer die Requests */
#define PORT 7777

int main(void)
  {
  struct sockaddr_in my_addr;
  struct sockaddr_in remote_addr;
  int size;
  int s;
  int remote_s;

  /* Die Socket erzeugen */
  s = socket(AF_INET, SOCK_STREAM, 0);
  if (s < 0)
    {
      fprintf(stderr, "Error: Socket\n");
      return -1;
    }

  memset(&my_addr, 0, sizeof (my_addr));
  my_addr.sin_family = AF_INET;
  my_addr.sin_port = htons(PORT);
  my_addr.sin_addr.s_addr = INADDR_ANY; /* An jedem Device warten */

  if (bind(s, (struct sockaddr *)&my_addr, sizeof(my_addr))==-1)
    {
    fprintf(stderr, "Error: bind\n");
    return -1;
    }

  /* Warteschlange einrichten */
  if (listen(s, 1) == -1)
    {
    fprintf(stderr, "Error: listen\n");
    return -1;
    }

  size = sizeof(remote_addr);
  /* Auf eine eingehende Verbindung warten */
  remote_s = accept(s, (struct sockaddr *)&remote_addr, (socklen_t*)&size);
  if (remote_s < 0)
    {
    fprintf(stderr, "Error: accept\n");
    return -1;
    }
  /* Infos ausgeben */
  printf("\nConnect von: %s\n", inet_ntoa(remote_addr.sin_addr));
  printf("sende Daten...\n");
  size = send(remote_s, "Hello World",11,0);
  if (size == -1)
    {
    fprintf(stderr, "error while sending\n");
    }
  else
    {
    printf("%d Bytes sent\n", size);
    }
  printf("closing sockets\n");
  /* Sockets wieder freigeben */
  close(remote_s);
  close(s);
  printf("terminating\n");
  return 0;
  }
```

Für die ersten Versuche ist das ganz nett, aber nach jeder Kommunikation muss der Server neu gestartet werden - nicht so toll. Zudem wird der Port nicht vom Kernel nicht sofort frei gegeben, sondern erst nach etwa einer halben Minute. Aber immerhin wissen wir nun, dass das Konzept funktioniert. Statt `fprintf(stderr, "...")` hätte ich übrigens auch `perror("...")` nehmen können.

Erweiterter TCP-Server

Nun wird der Server so erweitert, dass er mehrere Anfragen nacheinander bearbeiten kann. Am Anfang bleibt alle wie zuvor - mit zwei Ausnahmen: Erstens gibt es für die Fehlermeldung mit anschließendem Exit eine Funktion `err_exit("...")` und auch die Kommunikation zwischen Server und Client ist in eine Funktion `do_dialog (int sock)` ausgelagert, die entsprechend den Anforderungen verändert und erweitert werden kann.

Da der Server nun mehrere Anfragen bedienen soll, gibt es nach dem Vorspann eine Endlosschleife, innerhalb der dann `accept()` und `do_dialog` aufgerufen werden. Ist die Kommunikation mit dem Client beendet, wird lediglich der mit `accept()` kreierte Client-Socket (`newsockfd`) geschlossen. Der Server-Socket (`sockfd`) bleibt bestehen und so kann der Server die nächste Anfrage bearbeiten. In der Kommunikationsroutine darf diesmal auch der Client Daten senden, die der Server am Bildschirm ausgibt und seinerseits mit "OK" beantwortet - ein erster, zarter Ansatz für ein Protokoll. Beachten Sie jedoch, dass der Server nur immer eine Anfrage auf einmal verarbeiten kann.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>


/* Port fuer die Requests */
#define PORT 7777

/* Puffergroesse */
#define BUFSIZE 16384

void do_dialog (int sock);
void err_exit(char *message);

int main(int argc, char *argv[])
  {
  int sockfd, newsockfd, c_len;
  struct sockaddr_in serv_addr, c_addr;

  /* Socket erzeugen */
  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if (sockfd < 0)
    err_exit("ERROR opening socket");

  /* Socket-Struktur initialisieren */
  memset(&serv_addr, 0, sizeof (serv_addr));
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = INADDR_ANY;
  serv_addr.sin_port = htons(PORT);

  /* Port an die Host-Adresse binden */
  if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
    err_exit("ERROR on binding");

  /* Auf Verbindungsanfragen warten */
  listen(sockfd,5);

  for(;;)
    {
    printf("Server bereit ...\n");
    /* Verbindung akzeptieren */
    c_len = sizeof(c_addr);
    newsockfd = accept(sockfd, (struct sockaddr *)&c_addr, (socklen_t *)&c_len);
    if (newsockfd < 0)
      err_exit("ERROR on accept");

    /* Verbindung steht -> Kommunizieren */
    printf("Connect von: %s\n", inet_ntoa(c_addr.sin_addr));
    do_dialog(newsockfd);
    close(newsockfd);
    }
  return 0;
  }

/* Kommunikation ueber den Socket durchfuehren */
void do_dialog (int sock)
  {
  int n;
  char buffer[BUFSIZE];
  memset(buffer,0,BUFSIZE);

  n = read(sock,buffer,BUFSIZE);
  if (n < 0)
    err_exit("ERROR reading from socket");
  printf("Message: %s\n",buffer);
  n = write(sock,"OK\n",3);
  if (n < 0)
```

```
    err_exit("ERROR writing to socket");
  }

/* Fehlermeldung ausgeben und exit */
void err_exit(char *message)
  {
  perror(message);
  exit(1);
  }
```

TCP-Server mit Fork

Wenn ein Server stärker in Aspruch genommen wird (man denke nur an die Server von Google), ist er ab einer gewissen Anfragefrequenz teilweise nicht mehr erreichbar (die Warteschlange kann ja nur maximal 5 Clients verarbeiten). In so einem Fall bietet es sich an, einen Server zu schreiben, der für jede Anfrage einen eigenen Prozess startet. Dies geschieht unter Unix/Linux mittels `fork()` (siehe auch Parallelität und Signale). Wir erinnern uns: `fork()` verdoppelt beim Aufruf den aktuellen Prozess und sowohl Original als auch die Kopie bekommt einen Rückgabewert. Dabei erhält der Elternprozess die PID des erzeugten Kindprozesses, während der Kindprozess 0 bekommt (beim Fehlschlag liefert `fork()` -1 an den Aufrufer retour.

Nach dem Verbindungsaufbau mit `accept()` wird nun geforkt. Hier kommt ein weiteres Konzept von Unix/Linux zum tragen: Jeder Kindprozess "erbt" alles Wichtige von seinen Eltern. In diesem Fall wird auch die offene TCP-Verbindung vererbt und der Kindprozess kann nun mit dem Client kommunizieren, während der Elternprozess schon wieder für den nächsten Verbindungswunsch bereit steht. der Kindprozess schließt deshalb auch den Server-(Eltern-)Socket (den er nicht braucht) und führt den Dialog mit dem Client durch. Der Elternprozess schließt seinerseits den Client-Socket und kann nun wieder mit `accept()` auf Anfragen warten.

Ein Problem mit den Kindern bei Unix ist, dass sie zu Zombies werden wenn sie sterben. Solche Zombies entstehen, wenn Kindprozesse sich beenden und es den Elternprozess "nicht interessiert". Ein ordentlicher Elternprozess wartet darauf, dass ein Kind stirbt (eine schrecklich nette Familie!). Dazu ruft er `wait()` auf. Doch `wait()` ist eine blockierende Funktion, was bedeuten würde, dass der Server so lange warten muss, bis der Kindprozess beendet ist. Also lässt er doch wieder nur eine Anfrage zu, oder? Zum Glück nicht, denn die Alternative `waitpid()` ist nicht blockierend. Und da der Server ja dauernd läuft, kann er weitermachen und alles klappt wie gewünscht.

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


/* Port fuer die Requests */
#define PORT 7777

/* Puffergroesse */
#define BUFSIZE 16384

void do_dialog (int sock);
void err_exit(char *message);

int main(int argc, char *argv[])
  {
  int sockfd, newsockfd, c_len, status;
  struct sockaddr_in serv_addr, c_addr;
  pid_t pid, wpid;

  /* Socket erzeugen */
  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if (sockfd < 0)
    err_exit("ERROR opening socket");

  /* Socket-Struktur initialisieren */
  memset(&serv_addr, 0, sizeof (serv_addr));
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = INADDR_ANY;
  serv_addr.sin_port = htons(PORT);

  /* Port an die Host-Adresse binden */
  if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
    err_exit("ERROR on binding");

  /* Auf Verbindungsanfragen warten */
  listen(sockfd,5);
  c_len = sizeof(c_addr);
```

```
  for(;;)
    {
    printf("Server bereit ...\n");
    /* Verbindung akzeptieren */
    newsockfd = accept(sockfd, (struct sockaddr *) &c_addr, (socklen_t *)&c_len);
    if (newsockfd < 0)
      err_exit("ERROR on accept");

    /* Verbindung steht */
    printf("Connect von: %s\n", inet_ntoa(c_addr.sin_addr));

    /* Kindprocess erzeugen (erbt die akt. Verbindung) */
    pid = fork();
    if (pid < 0)
      err_exit("ERROR on fork");
    else if (pid == 0)
      {
      /* Das ist der Kindprozess */
      close(sockfd); /* Eltern-Socket schliessen */
      /* Kommunizieren */
      do_dialog(newsockfd);
      exit(0);
      }
    else
      {
      /* Dies ist der Elternprozess */
      close(newsockfd); /* Kind-Socket schliessen */
      printf("Parent PID =  %d, Child PID = %d\n", getpid(), pid);
      waitpid(pid, &status, WNOHANG);
      printf("Kindprozess-Exitstatus: %d\n", status);
      }
    } /* end of while */
  }

/* Kommunikation ueber den Socket durchfuehren */
void do_dialog (int sock)
  {
  int n;
  char buffer[BUFSIZE];
  memset(buffer,0,BUFSIZE);

  n = read(sock,buffer,BUFSIZE);
  if (n < 0)
    err_exit("ERROR reading from socket");
  printf("Message: %s\n",buffer);
  n = write(sock,"OK\n",3);
  if (n < 0)
    err_exit("ERROR writing to socket");
  }

/* Fehlermeldung ausgeben und exit */
void err_exit(char *message)
  {
  perror(message);
  exit(1);
  }
```

TCP-Client

Auf der Client-Seite ist das Öffnen des Socket praktisch schon alles. Man kann danach Schreiben und Lesen. Hier ein Beispiel für ein Client-Programm, das einen Socket einrichtet, Daten empfängt und den Socket wieder schließt. Der Server wird auf der Parameterzeile als Domainname angegeben (z .B. localhost), weshalb der Client die IP-Adresse mittels `gethostbyname()` ermittelt:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

/* Port fuer die Requests */
#define PORT 7777

/* Puffergroesse */
#define BUFSIZE 16384
```

```
int main(int argc, char **argv)
  {
  struct sockaddr_in host_addr;
  int size;
  int sock;
  struct hostent *host;
  char buffer[BUFSIZE];

  if (2 != argc)
    {
    fprintf(stderr, "Angabe des Servers fehlt\n");
    return -1;
    }

  host = gethostbyname(argv[1]);
  if (host == NULL)
    {
    fprintf(stderr, "Unbekannter Host %s\n",argv[1]);
    return -1;
    }

  /* Socket erzeugen */
  sock = socket(AF_INET, SOCK_STREAM, 0);
  if (sock < 0)
    {
    fprintf(stderr, "Error: socket\n");
    return -1;
    }
  /* Socket an das Ziel binden */
  memset(&host_addr, 0, sizeof (host_addr));
  host_addr.sin_family = AF_INET;
  host_addr.sin_addr = *((struct in_addr *)host->h_addr);
  host_addr.sin_port = htons(PORT);
  /* Verbindung aufbauen */
  if (connect(sock, (struct sockaddr *)&host_addr, sizeof(host_addr)) == -1)
    {
    fprintf(stderr, "Error: connect\n");
    return -1;
    }
  /* Daten empfangen */
  size = recv(sock, buffer, BUFSIZE, 0);
  if (size == -1)
    {
    fprintf(stderr, "reading data failed\n");
    return -1;
    }

  printf("%d Bytes: %s\n",size,buffer);
  /* Socket wieder freigeben */
  close(sock);
  return 0;
  }
```

Bei diesem sehr einfachen Client wird ein Problem der Netzwerkprogrammierung offenbar: Auch wenn beim Lesen mit `recv()` keine Daten empfangen wurden (`size == 0`), kann daraus nicht geschlossen werden, dass der Server schon alle Daten gesendet hat - es könnte ja auch eine längere Latenz im Netz schuld sein. Deshalb müssen immer Vereinbarungen darüber getroffen werden, wie die Kommunikation zwischen Server und Client ablaufen soll (Protokoll). Zumindest eine End-of-Data-Markierung ist notwendig. Im obigen Fall wird nur ein Datenblock empfangen. Wie der Empfang von mehreren Datenblöcken programmiert werden kann, zeigt der HTTP-Client weiter unten.

Daytime-Server und -Client

Die aktuelle Zeit im lesbaren Format (daytime) wird normalerweise auf Port 13 bereitgestellt - sofern da überhaupt ein Server läuft. Der folgende Daytimeserver erlaubt für Nicht-root-User die Angabe eines Alternativports auf der Kommandozeile. Ausserdem handelt es sich wieder um einen Server, der mittels `fork()` einen Kindprozess als Daemon im Hintergrund absetzt. Auch ist hier eine grunglegende Fehlerbehandlung implementiert.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <arpa/inet.h>
```

```
#define DEFAULTPORT 13
#define LINEMAX 1000

int main(int argc, char *argv[])
  {
  int server, client;
  socklen_t len;
  struct sockaddr_in server_addr;
  struct sockaddr_in client_addr;
  time_t t;
  struct tm *tm;
  char timemsg[LINEMAX];
  int portnumber;

  /* Rest der struct nullsetzen */
  memset(&(server_addr.sin_zero), '\0', 8);

  /* Kommandozeile einlesen */
  switch (argc)
    {
    case 1:
      /* Keine Argumente - default port */
      portnumber = DEFAULTPORT; break;
    case 2:
      /* Portnummer von der Kommandozeile */
      portnumber = atol(argv[1]);
      if (portnumber <= 0 || portnumber > 65535)
        {
        perror("Falsche Portnummer (muss zwischen 0 und 65535 liegen)");
        return 1;
        }
      break;
    default:
      fprintf(stderr, "Usage: %s [portnumber] (0 < port_number < 65535)\nBeispiel: %s 1024\n", argv[0], argv[0]);
      return 1;
    }

  /* Set protocol to "0" to have socket() choose the correct protocol. */
  if ((server = socket(PF_INET, SOCK_STREAM, 0)) < 0)
    {
    perror("Fehler beim Anlegen des Socket");
    return 2;
    }

  /* Server-Adress-Struktur vorbereiten */
  memset( &server_addr, 0, sizeof (server_addr));
  server_addr.sin_family = AF_INET;
  server_addr.sin_port = htons(portnumber);
  server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
  /* Listen to an interface with a specific IP (requires <arpa/inet.h>). */
  /* inet_aton("127.0.0.1", &(server_addr.sin_addr)); */

  if (bind(server, (struct sockaddr *)&server_addr, sizeof server_addr) < 0)
    {
    perror("Fehler bei bind (Portnummer < 1024? Dann musst Du root sein)");
    return 3;
    }

  /* Daemonisieren */
  switch (fork())
    {
    case  0: break;
    case -1: perror("Fork fehlgeschlagen"); return 4; break;
    default: close(server); return 0; break;
    }

  listen(server, 5);

  /* warten auf Anfragen */
  for (;;)
    {
    len = sizeof(client_addr);

    if ((client = accept(server, (struct sockaddr *)&client_addr, &len)) < 0)
      {
      perror("Accept fehlgeschlagen");
      return 4;
      }
    fprintf(stderr, "Connect von %s\n",inet_ntoa(client_addr.sin_addr));
    /* Zeit bestimmen */
    t = time(NULL);
```

```
    tm = localtime(&t);
    sprintf(timemsg, "%.4i-%.2i-%.2i %.2i:%.2i:%.2i %s\n",
          tm->tm_year + 1900,
          tm->tm_mon + 1,
          tm->tm_mday,
          tm->tm_hour,
          tm->tm_min,
          tm->tm_sec,
          tm->tm_zone);

    /* Uhrzeit zum Client senden */
    if ((send(client, timemsg, strlen(timemsg), 0)) < 0)
      {
      perror("Senden fehlgeschlagen");
      return 6;
      }

    if (close(client) < 0)
      {
      perror("daytimed-tcp close");
      return 7;
      }
    }
  }
```

Der zugehörige Client erlaubt ebenfalls die Angabe der Portnummer. Der abzufragende Host wird als Domainname angegeben - das Programm erfragt intern die IP-Adresse über `gethostbyname()`. Die vom Host abgefragte Zeit- und Datumsinfo wird auf der Standardausgabe ausgegeben.

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

#define MAXDATASIZE 100
#define DEFAULTPORT 13

int main(int argc, char *argv[])
  {
  int server;
  int numbytes;
  struct sockaddr_in server_addr;
  struct hostent *host;
  char buf[MAXDATASIZE];
  int portnumber;

 /* Kommandozeile einlesen */
  switch (argc)
    {
    case 3:
      /* Host und Portnummer von der Kommandozeile */
      portnumber = atol(argv[2]);
      if (portnumber <= 0 || portnumber > 65535)
        {
        perror("Falsche Portnummer (muss zwischen 0 und 65535 liegen)");
        return 1;
        }
      host = gethostbyname(argv[1]);
      if (host == NULL)
        {
        perror("Unbekannter Host");
        return 1;
        }
      break;
    default:
      fprintf(stderr, "Usage: %s [host] [port] (0 < port < 65535)\nBeispiel: %s localhost 1024\n",
                      argv[0], argv[0]);
      return 1;
    }

  /* Set protocol to "0" to have socket() choose the correct protocol. */
  if ((server = socket(PF_INET, SOCK_STREAM, 0)) < 0)
    {
    perror("Fehler beim Anlegen des Socket");
    return 2;
    }
```

```
  memset(&server_addr, 0, sizeof (server_addr));
  server_addr.sin_family = AF_INET;
  server_addr.sin_port = htons(portnumber);
  server_addr.sin_addr = *((struct in_addr *)host->h_addr);
  if (connect(server, (struct sockaddr *)&server_addr, sizeof server_addr) < 0)
    {
    perror("Connect fehlgeschlagen");
    close(server);
    return 3;
    }

  /* Receive the message with recv(). */
  /* (The message was sent in the server applikation the server with send().) */
  if ((numbytes = recv(server, buf, MAXDATASIZE-1, 0)) == -1)
    {
    perror("Empfangsfehler");
    close(server);
    return 4;
    }

  /* Null-terminate the message so we can print it as a string.  */
  buf[numbytes] = '\0';
  printf("%s",buf);
  close(server);
  return 0;
  }
```

Fileserver und -client

Nach dem gleichen Schema sind auch die Beispiele für einen Fileserver und einen Client dazu ausgeführt. Das Fileserver-Programm wartet auf einen Connect vom zugehoerigen Client. Der Client sendet einen Dateinamen und der Server schickt diese Datei (sofern vorhanden) dann an den Client.

Beim Client werden als Kommandozeilenparameter werden der Servername und der Dateiname übergeben. Das Programm versucht dann, diese Datei vom Server zu laden, indem es den Dateinamen sendet. Es gibt diese dann auf der Standardausgabe aus.

UDP-Server

Wenn mit UDP eine Nachricht versendet wird, ist nicht unbedingt sicher, dass diese Nachricht ihr Ziel erreicht. Auch wird nicht garantiert (wie bei TCP, wo die Pakete ggf. sortiert werden), in welcher Reihenfolge die einzelnen Pakete am Ziel angekommen. Allerdings wird mit UDP ein höherer Datendurchsatz erreicht, als dies bei TCP der Fall ist. Im Gegensatz zu TCP gibt es bei UDP keine virtuelle Verbindung, die anfangs aufgebaut und am Ende wieder abgebaut wird. Bei UDP werden die Daten gesendet, ohne eine Verbindung zu etablieren und ohne auf irgendeine Bestätigung zu warten.

Deshalb entfallen auch `listen()` und `accept()`. Der folgende UDP-Server wartet auf eine Nachricht vom Client, gibt diese aus und beendet sich. Das Paket wird also komplett auf einmal übertragen, was bedeutet, dass es mit einem einzigen Leseaufruf gelesen werden kann. Trotzdem hat auch ein UDP-Paket eine maximale Größe, abhängig vom Transportweg und der Hardware (Ethernet: 1500 Bytes). Der Server ruft lediglich die Funktion `recvfrom()` auf und wartet darauf, dass irgendein Client Daten schickt.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>


/* Port fuer die Requests */
#define PORT 7777

/* Puffergroesse */
#define BUFSIZE 4096

int main(void)
  {
  int sockfd;                                   /* unsere Socket   */
  struct sockaddr_in my_addr, remote_addr;      /* 2 Adressen      */
  int remote_addr_size = sizeof(remote_addr);   /* fuer recvfrom() */
  char buf[BUFSIZE];                            /* Datenpuffer     */

  if ((sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
    {
```

```
    fprintf(stderr, "Error: socket()\n");
    exit(1);
    }

  memset(&my_addr, 0, sizeof (my_addr));
  my_addr.sin_family      = AF_INET;
  my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
  my_addr.sin_port        = htons(PORT);

  if (bind(sockfd, (struct sockaddr*)&my_addr, sizeof(my_addr)) < 0)
    {
    fprintf(stderr, "Error: bind()\n");
    close(sockfd);
    exit(1);
    }

  if (recvfrom(sockfd, buf, sizeof(buf), 0,
              (struct sockaddr*)&remote_addr, (socklen_t*)&remote_addr_size) > 0)
    {
    printf("Getting Data from %s\n", inet_ntoa(remote_addr.sin_addr));
    printf("Data : %s\n", buf);
    }
  close(sockfd);
  return(0);
  }
```

Den Server so umzubauen, dass er sich nicht beendet, sondern auf weitere Client-Anfragen zu warten, geht genauso wie bei den TCP-Servern. Um die Empfangsroutine herum kommt wieder eine Endlosschleife:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

/* Port fuer die Requests */
#define PORT 7777

/* Puffergroesse */
#define BUFSIZE 4096

int main (int argc, char **argv)
  {
  int sock, client, rc, len;
  struct sockaddr_in cliAddr, servAddr;
  char buf[BUFSIZE];

  const int y = 1;  /* fuer setsockopt */

  /* Socket erzeugen */
  sock = socket (AF_INET, SOCK_DGRAM, 0);
  if (sock < 0)
    {
    fprintf(stderr, "Kann Socket nicht Ã¶ffnen\n");
    exit(1);
    }

  /* Lokalen Server Port binden */
  memset(&servAddr, 0, sizeof (servAddr));
  servAddr.sin_family = AF_INET;
  servAddr.sin_addr.s_addr = htonl (INADDR_ANY);
  servAddr.sin_port = htons (PORT);
  /* sofortiges Wiederverwenden des Ports erlauben */
  setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &y, sizeof(int));
  rc = bind(sock, (struct sockaddr *) &servAddr, sizeof (servAddr));
  if (rc < 0)
    {
    fprintf (stderr, "Kann Port nicht binden\n");
    exit (1);
    }
  printf ("Warte auf Daten ...\n");

  /* Serverschleife */
  for (;;)
    {
    /* Puffer initialisieren */
    memset (buf, 0, BUFSIZE);
```

```
    /* Nachrichten empfangen */
    len = sizeof (cliAddr);
    client = recvfrom (sock, buf, BUFSIZE, 0,
                  (struct sockaddr *) &cliAddr, (socklen_t*)&len );
    if (client < 0)
      {
      fprintf(stderr, "Kann keine Daten empfangen ...\n");
      continue;
      }

    /* Erhaltene Nachricht ausgeben */
    printf("Getting Data from %s, UDP-Port %u\n",
            inet_ntoa(cliAddr.sin_addr), ntohs(cliAddr.sin_port));
    printf("Data: %s\n", buf);
    }
  return (0);
  }
```

Jeder ankommende Datenblock wird getrennt vom vorhergehenden betrachtet, zusammengehörige Daten können nur anhand der IP-Adresse des Absenders detektiert werden. Auf die gleiche Weise kann der Server nach dem Connect eines Clients auch Daten versenden, anstatt sie zu empfangen.

UDP-Client

Der Client "passt" zum obigen Server. Auf der Kommandozeile werden der Hostname (z. B. localhost) und die zu sendende Nachricht angegeben. Auch hier gibt es keinen Verbindungsaufbau per `connect()`, sondern der Client schiesst gleich mit `sendto()` Datagramme an den Server.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

/* Port fuer die Requests */
#define PORT 7777

/* Puffergroesse */
#define BUFSIZE 4096

int main(int argc, char **argv)
  {
  int sockfd;
  struct sockaddr_in remote_addr;
  struct hostent *host_addr;

  if (argc != 3)
    {
    fprintf(stderr, "Usage: %s <HOST> <MESSAGE>\n", argv[0]);
    }

  if ((host_addr = gethostbyname(argv[1])) == NULL)
    {
    fprintf(stderr, "Cannot resolv hostname: %s\n", argv[1]);
    exit(1);
    }

  if ((sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
    {
    fprintf(stderr, "Error: socket()\n");
    exit(1);
    }

  memset(&remote_addr, 0, sizeof (remote_addr));
  remote_addr.sin_family = AF_INET;
  remote_addr.sin_addr   = *((struct in_addr *) host_addr->h_addr);
  remote_addr.sin_port   = htons(PORT);

  if (sendto(sockfd, argv[2], strlen(argv[2]) + 1, 0,
             (struct sockaddr *)&remote_addr, sizeof(remote_addr)) > 0)
    {
    printf("Message sent\n");
    }
  else
```

```
    {
    fprintf(stderr, "Error while sending data\n");
    }
  close(sockfd);
  return(0);
  }
```

Der Server kann dem Client dann gegebenenfalls noch mit `sendto()` antworten, da der Client mit seinem `sendto()` auch seine Adresse mitgeschickt hat. Der Client kann die Daten vom Server mit `recvfrom()` einlesen.

Einfacher HTTP-Server

Als praktische Anwendung soll nun ein ganz einfacher Webserver programmiert werden. Er nimmt ausschließlich HTTP-GET-Requests entgegen und sendet daraufhin die gewünschte Datei. So ein Server könnte seinen Dienst beispielsweise in irgendeiner Appliance verrichten, wo ein Webserver wie der Apache die Rechen- und Speicherkapazität sprengen würde. Der erste Teil des Programms (main) sieht genauso aus, wie bei den TCP-Server weiter oben. Die Bearbeitung des HTTP-Requests wird von der Funktion `serv_request()` übernommen.

Diese Funktion liest zunächst die Anfrage des Browsers in einen Puffer. Im Fall eines GET besteht der Request nur aus dem Header, der neben der Zeile mit dem Request noch etliche Zeilen mit Angaben zum Client/Browser enthält (auf diese Weise erfährt ein Webserver auch mehr über den User. Auch Cookies sind im Header enthalten.) Das Aufdröseln des Pufferinhalts in einzelne Zeilen kann mittels `strtok()` erfolgen, wobei als Delimiter der Zeilenwechsel "\r\n" angegeben wird:

```
 /* ptr ist ein Hilfszeiger, der mit ptr = buffer initialisiert ist */
 ptr = strtok(buffer,delimiter);
  while ((ptr != NULL))// && !eoh)
    {
    printf ("Headerzeile: %s\n", ptr);
    /* GET-Request? Dann URL extrahieren */
    sscanf(ptr, "GET %1023s HTTP/", url);
    /* naechste Zeile nehmen */
    ptr = strtok(NULL,delimiter);
    }
```

Ausnahmsweise mache ich hier mal keinen weiten Bogen um die Funktion `sscanf()`, denn diese Funktion kann Zeichenmuster in der Eingabezeile erkennen. In diesem Fall suche ich nach dem Muster `"GET <Dateipfad> HTTP/"`. Für den Dateipfad steht der Platzhalter "%1023s", der für einen String von maximal 1023 Zeichen steht. Passt das Muster auf die aktuelle Zeile, packt mir `sscanf()` brav die Dateiangabe in die Variable `url`, in allen anderen Fällen wird die Zeile ignoriert. Gegebenenfalls muss noch ein an der URL hängendes '\r' beseitigt werden.

Bei der URL selbst, genauer bei der Dateiangabe des Browsers, muss auch noch unterschieden werden:

- Keine Dateiangabe (nur z. B. "http://www.netzmafia.de"):
  Es wird als Defaultwert die Datei bzw. der Pfad "/index.html" hinzugefügt.
- Angabe eines Pfades, also eines Verzeichnisses (Dateityp kann mittels `stat()` ermittelt werden):
  Ebenfalls Anhängen von "/index.html" an den Dateipfad. So wird z. B. aus "http://www.netzmafia.de/skripten" nun "http://www.netzmafia.de/skripten/index.html".
- Angabe einer Datei, ggf. mit Pfad:
  keine Veränderungen - ausliefern der Datei

Falls die so aus der URL ermittelte Datei existiert, wird sie zum Client übertragen, wobei unser Server noch einen minimalen Header hinzufügen muss (wer will, kann den Header noch erweitern, wichtig ist eine Leerzeile als Trennung zwischen Header und Body). Gibt es die Datei nicht, wird ein Fehler-404-Header gesendet.

```
/* Implementierung eines einfachen HTTP-Servers,
 * der ausschließlich GET-Requests bearbeiten kann */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>


/* Port fuer die HTTP-Requests */
#define HTTP_PORT 8080
```

```
static void serv_request(int client_fd, char* rootpath);
static void err_exit(char *error_message);

int main( int argc, char **argv)
  {
  struct sockaddr_in server, client;
  int sock, clientd;
  int len;

  /* Die "Document Root" wird per Kommadozeilenargument
   * uebergeben */
  if (2 != argc)
    err_exit("Angabe von document_root fehlt\n");

  /* Erzeuge das Socket */
  sock = socket( PF_INET, SOCK_STREAM, 0);
  if (sock < 0)
    err_exit("Kann Socket nicht anlegen");

  /* Erzeuge die Socketadresse des Servers */
  memset(&server, 0, sizeof (server));
  server.sin_family = AF_INET;
  server.sin_addr.s_addr = htonl(INADDR_ANY);
  server.sin_port = htons(HTTP_PORT);

  /* Binde Port an die Serveradresse  */
  if (bind(sock, (struct sockaddr*)&server, sizeof( server)) < 0)
    err_exit("Kann Socket nicht an Port binden");
  listen(sock, 5);

  /* Bearbeite die Verbindungswuensche von Clients
   * in einer Endlosschleife  */
  for (;;)
    {
    printf("Server wartet auf Anfrage ...\n");
    len = sizeof(client);
    clientd = accept(sock, (struct sockaddr*)&client, (socklen_t*)&len);
    if (clientd < 0)
      err_exit("accept failed");
    printf("client address: %s\n", inet_ntoa(client.sin_addr));
    /* Bearbeite den HTTP-Request */
    serv_request(clientd, argv[1]);
    /* Schließe Verbindung */
    close(clientd);
    }
  }

/* Bearbeite den ankommenden HTTP-Request */
static void serv_request(int client_fd, char* rootpath)
  {
  char delimiter[] = "\r\n";  /* Zeilentrenner im Puffer */
  char buffer[16384];         /* Empfangspuffer */
  char *ptr = NULL;           /* Hilfspointer */
  struct stat info;           /* fileinfo */
  char url[1024];             /* URL */
  char path[1024];            /* Dateipfad */
  int count;                  /* Bytezaehler */
  int fd;                     /* Filedesc. fuer Ausgabedatei */
  int eoh = 0;                /* End of Header Flag */

  *url = '\0';
  /* HTTP-Request einlesen */
  count = recv(client_fd, buffer, sizeof(buffer), 0);
  eoh = (count > 0)? 0 : 1;
  buffer[count] = '\0'; /* Stringterminator setzen */
  ptr = strtok(buffer,delimiter);
  while ((ptr != NULL))// && !eoh)
    {
    printf ("Headerzeile: %s\n", ptr);
    /* GET-Request? Dann URL extrahieren */
    sscanf(ptr, "GET %1023s HTTP/", url);
    /* naechste Zeile nehmen */
    ptr = strtok(NULL,delimiter);
    }

  if (url[strlen(url)-1] == '\r') url[strlen(url)-1] = '\0';
  if (strlen(url) > 1)
    { /* URL im Header gefunden */
    printf( "--- Request: GET %s ", url);
    sprintf(path, "%s/%s", rootpath, url);
    printf( "--- Path: %s ", path);
```

```
    }
  else
    { /* als default "index.html" nehmen */
    printf( "--- Request: GET /index.html ");
    sprintf(path, "%s/index.html", rootpath);
    printf( "--- Path: %s ", path);
    }
  if (stat(path, &info) == 0 && S_ISDIR(info.st_mode))
    { /* bei Directory "/index.html" anhaengen */
    sprintf(path, "%s/%s/index.html", rootpath, url);
    printf( "--- Path: %s ", path);
    }

  /* gewuenschte Datei oeffnen */
  fd = open(path, O_RDONLY);
  if (fd > 0)
    { /* Datei vorhanden, also ausliefern */
    sprintf(buffer, "HTTP/1.0 200 OK\nContent-Type: text/html\n\n");
    send(client_fd, buffer, strlen(buffer), 0);
    do
      {
      count = read(fd, buffer, sizeof(buffer));
      send(client_fd, buffer, count, 0);
      }
    while (count > 0);
    close(fd);
    }
  else
    { /* Datei nicht vorhanden - Fehler senden */
    sprintf(buffer, "HTTP/1.0 404 Not Found\n\n");
    send(client_fd, buffer, strlen(buffer), 0);
    }
  printf(" --- done!\n");
  }


/* Funktion gibt aufgetretenen Fehler aus und
 * beendet das Programm */
static void err_exit(char *error_message)
  {
  fprintf(stderr, "%s: %s\n", error_message, strerror(errno));
  exit(EXIT_FAILURE);
  }
```

Einfacher HTTP-Client

Eigentlich hat ja jeder einen HTTP-Client auf dem Computer, den Browser. Für die Kommandozeile hat man die Wahl zwischen `lynx`, dem Textbrowser sowie `wget` bzw. `curl` als Kommandozeilentools. Wozu dann den Client selbst programmieren? Eine Antwort wäre "Weil ich es kann!" oder auch "Weil ich hier ein Beispiel brauche."

Der folgende HTTP-Client ist auch ein Muster an Einfachheit. Er entspricht im Grund dem Muster-TCP-Client oben. Die empfangenen Daten werden einfach auf die Standardausgabe kopiert. Das hat den Vorteil (oder auch Nachteil), dass der Header auch zu sehen ist. Zwischen Header und Body (der eigentlichen Webseite) befindet sich eine Leerzeile. Wer will, kann das Programm ja dahingehen erweitern, dass Header und Body getrennt werden (oder man schaltet per Pipe passende Unix-Kommandos dahinter).

```
/* GET-Request via HTTP an einen Webserver */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdlib.h>
#include <string.h>

/* HTTP-Port */
#define PORT 80

/* Groesse des Puffers */
#define BUFSIZE 8192

/* Filehandle-Nummer der Standard-Ausgabe */
#define STD_OUT 1

static void err_exit(char *error_message);
```

```
int main( int argc, char **argv)
  {
  struct sockaddr_in server;
  struct hostent *host_info;
  char buffer[BUFSIZE];
  int sock;
  int count;

  if (argc != 3)
     err_exit("usage: httpget <server> <path/file>\n");

  /* Erzeuge das Socket */
  sock = socket(PF_INET, SOCK_STREAM, 0);
  if (sock < 0)
     err_exit("failed to create socket");

  memset(&server, 0, sizeof (server));
  server.sin_family = AF_INET;
  server.sin_port = htons(PORT);

  /* Wandle den Servernamen in eine IP-Adresse um */
  host_info = gethostbyname( argv[1]);
  if (host_info == NULL)
      err_exit("unknown server");
  memcpy((char *)&server.sin_addr, host_info->h_addr, host_info->h_length);

  /* Baue die Verbindung zum Server auf */
  if (connect(sock, (struct sockaddr*)&server, sizeof(server)) < 0)
     err_exit("can't connect to server");

  /* Sende den HTTP-GET-Request */
  sprintf(buffer, "GET %s HTTP/1.0\r\n\r\n", argv[2]);
  send(sock, buffer, strlen( buffer), 0);

  /* Hole die Serverantwort und gib sie auf Konsole aus */
  do
    {
    count = recv(sock, buffer, sizeof(buffer), 0);
    write(STD_OUT, buffer, count);
    }
  while (count > 0);

  /* Schliesse Verbindung und Socket */
  close(sock);
  return(EXIT_SUCCESS);
  }


/* Funktion gibt aufgetretenen Fehler aus und
 * beendet das Programm */
static void err_exit(char *error_message)
  {
  fprintf(stderr, "%s: %s\n", error_message, strerror(errno));
  exit(EXIT_FAILURE);
  }
```

Ein TCP-Portscanner

Das folgende Programm stellt einen einfachen TCP-Portscanner dar, der mittels `connect()` nach offenen Ports sucht. Das kann wegen des Timeouts für eine fehlgeschlagene Verbindung schon eine Weile dauern. Dieses Beispiel zeigt zudem, wie man IP-Adressen einfach hochzählen kann - letztendlich sind sie ja nichts weiter als 32-Bit-Ganzzahlen.. Es werden nur die "well known ports" (0 - 1023) gescannt. Außerdem werden die Namen der Services zu den Ports ermittelt, sofern dieses möglich ist (Zugriff auf die Datei */etc/services* mittels `getservbyport()`).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <netdb.h>

int main(int argc, char *argv[])
  {
  struct sockaddr_in addr;
  struct servent *serv;
  int sock, i;
```

```
  unsigned long begin, end, curr;

  if (argc < 3)
    {
    fprintf(stderr, "usage: %s <begin> <end>\n", argv[0]);
    exit(1);
    }

  /* numerische IP-Adressen ermitteln */
  begin = ntohl(inet_addr(argv[1]));
  end = ntohl(inet_addr(argv[2]));

  /* Anfangadresse < Endadresse sicherstellen */
  if (begin > end)
    {
    curr = end; end = begin; begin = curr;
    }

  memset(&addr, 0, sizeof (addr));
  /* Adressbereich abscannen */
  for (curr = begin; curr <= end; curr++)
    {
    addr.sin_addr.s_addr = htonl(curr);
    printf("%s:\n", inet_ntoa(addr.sin_addr));

    /* well known ports abscannen */
    for (i = 0; i < 1024; i++)
      {
      sock = socket(PF_INET, SOCK_STREAM, 0);
      if (sock == -1)
        {
        perror("socket() failed");
        exit(1);
        }
      addr.sin_addr.s_addr = htonl(curr);
      addr.sin_port = htons(i);
      addr.sin_family = AF_INET;
      /* aktuelle Portnummer in Spalte 1 anzeigen */
      printf("--> %4i\r", i);
      fflush(stdout);

      if (!connect(sock, (struct sockaddr*)&addr, sizeof(addr)))
        {
        /* Namen des Services zum Port ermitteln */
        serv = getservbyport(addr.sin_port, "tcp");
        if (serv)
          printf("%i (%s) open\n", i, serv->s_name);
        else
          printf("%i (unknown) open\n", i);
        }
      close(sock);
      }
    puts("\r        \n");
    }
  return 0;
  }
```

1.6 Ein-/Ausgabe-Polling mit `select()`

In Unix steht ein leistungsfähiger Mechanismus zur Verfügung, der es einem Anwenderprogramm ermöglicht, verschiedene Eingabekanäle "abzuhorchen", darüber festzustellen, ob mindestens einer davon Daten "anbietet" (oder zur Aufnahme von Daten bereit ist - s.u.), und im Anschluss daran diesen zum Lesen bzw. Schreiben auszuwählen. Ein solches Szenario ist beispielsweise dann gegeben, wenn das Programm auf Daten wartet, die ihm ein anderes über eine Pipe liefern soll, und es gleichzeitig auf Tastatureingaben reagieren möchte. In diesem Fall sind ja zwei Dateideskriptoren betroffen: Der Deskriptor 0 für die Standardeingabe und der Eingabedeskriptor der Pipe. Schematisch könnte man dies wie folgt beschreiben:

```
Wiederhole:
  |  Warte bis an der Standardeingabe ODER an der Pipe Daten anliegen
  |
  |  WENN Tastaturdaten vorliegen
  |       lies diese ein
  |       verarbeite sie
  |
  |  WENN an der Pipe Daten anliegen
  |       lies diese ein
  |       verarbeite sie
bis "fertig";
```

Außerdem kann `select()` verwendet werden, wenn ein Server als einzelner Prozeß mehrere Clients bedienen soll, da hier der Server erkennen kann, auf welchem Socket etwas gesendet oder empfangen werden soll. Dies ist notwendig, da der Aufruf von `recv()` so lange wartet, bis etwas empfangen wurde (er ist also blockierend). Der Server würde nun stehen bleiben, und das beim ersten Socket den er überprüft. Eine weitere Möglichkeit wäre nichtblockierende Ein-/Ausgabe (siehe `fcntl()`), die jedoch mehr Ressourcen braucht. `select()` wartet, bis etwas auf einem Socket aus der Socket-Liste ankommt bzw. gesendet werden kann. Nicht zuletzt kann man `select()` verwenden, um den Programmfluss für eine bestimmte Zeit zu unterbrechen (wie `sleep()` respektive `usleep()`). Zuerst die Deklaration von `select()` und dazugehöriger Makros:

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int n, fd_set *readfds, fd_set *writefds,
                   fd_set *exceptfds, struct timeval *timeout);

FD_CLR(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
FD_ISSET(int fd, fd_set *set);
```

Die Parameter dieser Funktion haben die folgende Bedeutung:

- **`n`** ist der höchste benötigte Datei-Deskriptor plus 1. Wenn der höchste relevante Deskriptor beispiesweise 7 ist, so muß das Bitmuster zu seiner Verwaltung acht Bits lang sei (0 bis 7) - n ist dann als 8 anzugeben.

- **`readfds`** ist (wie auch `writefds` und `exceptfds`) ein Zeiger auf ein Bitmuster, bei dem jedes gesetzte Bit einer Deskriptornummer entspricht. Wenn beispielsweise in `*readfds` die Bits 0 und 5 gesetzt sind, so bedeutet dies, daß der aufrufende Prozeß warten möchte, bis entweder über den Dateideskriptor 0 oder über den Deskriptor 5 Daten gelesen werden können.

- In **`*writefds`** bringt ein gesetztes Bit zum Ausdruck, daß gewartet werden soll, bis die entsprechende Datei zur Aufnahme von Daten bereit ist. Bei "normalen" Dateien geht das natürlich immer. Anders verhält es sich bei Deskriptoren, die auf eine Pipe oder auf einen Socket verweisen: Wenn der Sender seine Daten wesentlich schneller produziert, als sie vom Empfänger verarbeitet werden können, laufen die vom Betriebssystem bereitgestellte Puffer irgendwann voll. Der Sender muß nun warten, bis wieder Platz zur Verfügung steht.

- **`*exceptfds`** nimmt Bits auf, die eine Fortsetzung des aufrufenden Prozesses dann veranlassen, wenn an den betreffenden Dateideskriptoren Fehlerzustände auftreten - in diesem Fall kann werder geschrieben noch gelesen werden, es besteht aber die Möglichkeit, auf die Fehlersituation geeignet zu reagieren.

- **`timeout`** ist der Zeiger auf eine Datenstuktur, die einen Timeout vereinbart - also die Zeit, die `select()` verstreichen lassen soll bevor sie mit 0 zurückkehrt.
  Bei manchen Implementationen wird hier die Restzeit gespeichert wenn vor dem Ablaufen auf einem Deskriptor die geforderten Bedingungen zutreffen. Man sollte sich nicht darauf verlassen, jedoch ist es für portable Programme unbedingt notwendig, daß der Timeout vor einem erneuten Aufruf von `select()` wieder gesetzt wird, da er eventuell doch verändert worden sein kann. Die Struktur timeval ist in <sys/time.h> wie folgt deklariert:

  ```
  struct timeval
    {
    long tv_sec;  /* seconds      */
    long tv_usec; /* microseconds */
    };
  ```

  Wenn die darin angegebene Zeit 0 ist (`tv_sev == 0` und `tv_usec == 0`), blockiert `select()` überhaupt nicht, sondern kehrt sofort zurück. Wenn der Parameter "timeout" selbst gleich NULL ist (Nullzeiger), wartet `select()` unbegrenzt lange - bis sich auf einem der angesprochenen Dateideskriptoren etwas tut.

In der Manpage von `select()` unter Linux ist folgendes Beispiel angegeben:

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
  {
  fd_set rfds;
  struct timeval tv;
  int retval;

  /* Watch stdin (fd 0) to see when it has input. */
  FD_ZERO(&rfds);
  FD_SET(0, &rfds);
```

```
  /* Wait up to five seconds. */
  tv.tv_sec = 5;
  tv.tv_usec = 0;

  retval = select(1, &rfds, NULL, NULL, &tv);
  /* Don't rely on the value of tv now! */

  if (retval)
    printf("Data is available now.\n");
    /* FD_ISSET(0, &rfds) == true */
  else
    printf("No data within five seconds.\n");
  exit(0);
  }
```

Auch hier wird darauf hingewiesen, daß man nach dem Aufruf von `select()` nicht mehr auf den Wert in der Struktur `timeval` verlassen kann.

Nach Ausführung von `select()` enthalten auch die drei Bitmuster-Parameter nicht mehr die **vor** dem Aufruf gesetzten Bits, sondern es sind nur noch diejenigen gesetzt, deren zugeordnete Kanäle die Fortsetzung des Prozesses veranlasst haben.
Der **Rückgabewert** von `select()` enthält die Gesamtzahl der (noch) gesetzten Bits. Dieser Wert kann auch 0 sein, wenn der Timeout abgelaufen ist, ohne daß eine Verbindung eingegangen wurde. Bei einem Fehler wird -1 zurückgegeben.

Beispiel:

Bitnummer	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Variable IBM:	0	0	0	1	0	0	0	1
Variable OBM:	1	0	0	0	0	0	0	0
Variable EBM:	1	0	0	0	0	0	0	0

Mit dem Aufruf

```
int i = select(8, &IBM, &OBM, &EBM, NULL);
```

wird ohne Zeitbeschränkung darauf gewartet, daß entweder auf den "Kanälen" 0 oder 4 Eingabedaten zur Verfügung stehen, oder daß auf "Kanal" 7 ein Schreiben möglich ist (oder ein Fehler auftrat). Nach dem Verlassen der Funktion mit Rückgabewert 1 (nur noch 1 Bit gesetzt) sehen die Variablen wie folgt aus:

Bitnummer	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Variable IBM:	0	0	0	0	0	0	0	1
Variable OBM:	0	0	0	0	0	0	0	0
Variable EBM:	0	0	0	0	0	0	0	0

Die C-Bibliothek stellt Makros zur Verfügung, die das Setzen, Löschen und Abfragen von Bits in den bei `select()` benutzten Bitmuster erleichtern. Die Deklaration ist oben schon aufgelistet. Hier einige Beispiele:

- `FD_SET(4,&IBM)` setzt das Bit 4 im Bitmuster IBM
- `FD_CLR(2,&OBM)` löscht das Bit 2 im Bitmuster OBM
- `FD_ISSET(4,&IBM)` liefert "wahr", wenn Bit 4 in IBM gesetzt ist
- `FD_ZERO(&EBM)` setzt EBM auf 0

Abschließend noch ein Beispiel: Ein Programm soll auf Eingaben von der Tastatur warten, aber alle 3 Sekunden den Benutzer zur Eingabe auffordern, wenn er nicht reagiert.

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/time.h>

fd_set EBM;
struct timeval Zeit;
char buffer[1000];

int main()
  {
  do
    {
    printf("\nGibs mir:");
    fflush(stdout);                    /* Ausgabepuffer leeren */

    FD_ZERO(&EBM);   /* Eingabebitmuster = 0 */
```

```
    FD_SET(0,&EBM);  /* Bit 0 setzen         */

    Zeit.tv_sec=3;                     /* Timeout = 3 Sekunden */
    Zeit.tv_usec=0;
    }
  while (!select(1, &EBM, NULL, NULL, &Zeit));

  /* Wenn select() mit 0 zurückkommt, ist
     die Uhr abgelaufen andernfalls steht eine Eingabe an */

  fgets(buffer,1000,stdin);
  printf("Eingabe war: %s\n",buffer);
}
```

Der Aufruf `fflush(stdout)` wird in diesem Beispiel eingesetzt, damit die Eingabeaufforderung sofort auf dem Bildschirm erscheint.

Noch ein Beispiel: `select()` erlaubt es beispielsweise, ein Programm zu schreiben, das auf einem Port wartet und alle engehenden Daten an einen anderen Port weitergibt. Fertig ist der Proxy-Server! Angenommen man hat einen Rechner der per Modem eine Verbindung zum Internet aufgebaut hat als Gateway für ein lokales Netz dient (mittels IP-Masquerading). Nur der Gateway ist von aussen sichtbar, die Rechner des lokalen Netzes jedoch dahinter versteckt. Nehmen wir weiter an, daß auf einem der lokalen Rechner ein Web-Server läuft, der nach aussen Daten anbieten soll. Man braucht also ein Programm, das die Anfragen die an Port 80 des Gateway gelangen, zum Web-Server weitergereicht werden sollen (was einen einfachen Portforwarder aus dem Rennen wirft). Um das zu verwirklichen, braucht man also ein Programm das zwei Sockets geöffnet hat: einen zum Benutzer ausserhalb des Netzes, und einen zweiten der zum Web-Server führt. Das Programm muß erkennen, auf welchem Socket gerade etwas ankommt und diese Daten dann über den anderen Socket schicken. Eine Lösungsmöglichkeit wäre es, einen Firewall zu installieren. Mit `select()` geht es aber auch.

Das folgende Programmfragment zeigt, wie es geht.

```
int data_interchange(int src, int dest)
  {
  /* Implementierung der Polling-Methode + select() um
   * Systemressourcen zu sparen */

  char buffer[BUFFER_SIZE];
  int src_sent, src_recvd, dest_sent, dest_recvd, max, total, i;
  fd_set rfds;
  struct timeval tv;

  if (src > dest) max = src;
  else            max = dest;

  total = 0;
  fcntl(src, F_SETFL, O_NONBLOCK);
  fcntl(dest, F_SETFL, O_NONBLOCK);

  for (;;)
    {
    FD_SET(src, &rfds);
    FD_SET(dest, &rfds);
    tv.tv_sec = 300;
    tv.tv_usec = 0;
    select(max + 1, &rfds, NULL, NULL, &tv);

    src_recvd = recv(src, buffer, sizeof(buffer), 0);
    dest_recvd = recv(dest, buffer, sizeof(buffer), 0);

    if (src_recvd > 0)
      send(dest, buffer, src_recvd, 0);

    if (dest_recvd > 0)
      send(src, buffer, dest_recvd, 0);

    if ((src_recvd == 0) || (dest_recvd == 0))
      break;
    }
  return 0;
  }
```

Wie man sieht, wartet `select()` darauf, daß von einem der beiden Sockets gelesen werden kann. Ist dies der Fall, wird gelesen. Man hätte auch mit `FD_ISSET()` testen können, von welchem Socket gelesen werden kann, doch so haben wir gleich noch ein Beispiel für nicht-blockierende Ein-/Ausgabe. Durch den Aufruf von `fcntl()` mit dem Attribut `O_NONBLOCK` blockiert ein `recv()`-Aufruf nicht, bis Daten eingetroffen sind, sondern kehrt sofort zurück. Die beiden if-Abfragen überprüfen, von welchem der Sockets eingetroffen sind (Wert > 0). Falls von keinem der beiden Sockets Daten kommen, ist ein Timeout aufgetreten. Das bedeutet, daß der Server-Prozeß nach fünf Minuten Inaktivität automatisch endet.

1.7 Eingabepuffer splitten

Hinter den Sockets verbergen sich komplexe Netzwerkfunktionen des Betriebssystems, die unter anderem auch die ein- und ausgehenden Daten puffern. Da diese Puffer nicht unendlich groß sind, muss Ihre Anwendung Sorge dafür tragen, dass die Empfangspuffer regelmäßig geleert und die Sendepuffer nicht übermäßig gefüllt werden, sonst gehen Daten verloren. Wenn Sie die Daten in Form von Strings übertragen, gibt es normalerweise keine Probleme mit der Pufferung, sofern die Strings ordentlich mit '\0' abgeschlossen sind.

Wie groß Sie Ihre Puffer für Empfang und Senden machen, hängt vom Anwendungsfall ab. Allerdings ist ein Byte-Puffer genauso sinnlos wie ein überdimensional großes Pufferarray. Puffergrößen von 512 oder 1024 Byte sind oft zu finden. Im Netzwerk ist nur interessant, dass ein ankommendes Paketim Prinzip nur so gross sein kann, wie es die MTU (Maximum Transfer Unit) des Netzes erlaubt. Das wären z. B. 1500 Bytes für Ethernet. Es können jedoch auch größere Datagramme ankommen, wenn die sendende Einrichtung die MTU nicht beachtet, und es zur Fragmentierung von Datagrammen kommt. Insofern sind 4096 - 8192 Byte Puffergröße keine schlechte Wahl. Sie sollten auch beachten, dass Funktionen wie `recv()` die Bytes abliefern, die bis zum Funktionsaufruf empfangen wurden - unabhängig davon, ob da noch was kommt. Wie bei allen bekannten Internetanwendungen wie SMTP, POP3, IMAP, HTTP etc. werden bzw. wurden Protokolle auf eine höheren Ebene definiert, die fast immer aus einen Frage- und Antwort-Spiel bestehen. Damit läuft die Kommunikation nicht nur in geregelten Bahnen ab, sondern ist auch leichter zu debuggen. Näheres zu einigen höheren Protokollen finden Sie im Netzwerk-Skript.

Bei den bisherigen Beispielen habe ich mich elegant um die Verarbeitung von etwas größeren Datenmengen gedrückt. Entweder es genügte ein einmaliges Lesen von der Schnittstelle, um alle Daten des Absenders im Puffer zu haben oder ich habe den Empfangspuffer einfach mittels `write()` in Richtung Standardausgabe weitergeleitet - nach dem Motto "Soll sich doch ein anderes Programm um die Weiterverarbeitung kümmern". Aus diesem Grund soll hier das Splitten des Puffers in einzelne Zeilen etwas näher betrachtet werden. Normalerweise wäre es ein Zufall, wenn ein Zeilenende genau mit dem Pufferende zusammenfällt, und man den String per Standardfunktion splitten kann. Die Regel ist, dass ein Puffer den Anfang und der nächgste Puffer das Ende einer Zeile enthält. Darum soll jetzt die Arbeit mit mehreren aufeinanderfolgenden Datenblöcken näher betrachtet werden:

Die erste Möglichkeit, die sich anbietet, ist das zeichenweise Lesen von der Netzwerk-Schnittstelle (das Betriebssystem puffert ja die Pakete für uns). Sobald dann ein Newline ('\n') auftritt, wird die Funktion verlassen und gibt den String zurück. Es versteht sich von selbst, dass vom aufrufenden Programm ein Zeichen-Array zur Verfügung gestell werden muss und dass man dessen Länge nicht überschreitet. Die folgende Funktion `get_line()` liest von einen vorhergeöffneten Socket genau eine Zeile ein (der Unterstrich in `get_line()` ist notwendig, weil `getline()` eine Bibliotheksfunktion ist). Als Parameter werden neben dem Socket das Array und dessen maximale Länge übergeben:

```
int get_line(int fd, char *buffer, unsigned int len)
  {
  /* read a '\n' terminated line from socket fd into buffer
   * of size len. The line in the buffer is terminated
   * with '\0'. It returns -1 in case of error and -2 if the
   * capacity of the buffer is exceeded.
   * It returns 0 if EOF is encountered before reading '\n'.
  */
  int numbytes = 0;
  int ret;
  char buf;

  buf = '\0';
  while ((numbytes <= len) && (buf != '\n'))
    {
    ret = recv(fd, &buf, 1, 0);/* read a single byte */
    if (ret == 0) break;       /* nothing more to read */
    if (ret < 0) return -1; /* error or disconnect */
    buffer[numbytes] = buf;    /* store byte */
    numbytes++;
    }
 if (buf != '\n') return -2;   /* numbytes > len */
 buffer[numbytes-1] = '\0';    /* overwrite '\n' */
 return numbytes;
 }
```

Nachteil dieser Lösung ist die Geschwindigkeit bzw. deren Fehlen. Durch die vielen `recv()`-Aufrufe ist die Funktion ziemlich langsam. Besser wäre eine Lösung, bei der ein Datenpaket komplett eingelesen (z. B. bei Ethernet 1500 Bytes) und dann Zeile für Zeile ans aufrufende Programm weitergereicht wird. Genau das macht die folgende Funktion, bei der die Parameter die gleiche Aufgabe haben wie oben. Diese Funktion hat einen internen Puffer, der mittels `recv()` gefüllt wird und dessen Inhalt Stück für Stück bei jedem Aufruf weitergegeben wird. Dazu verwendet die Funktion die statischen Variablen `bufptr`, `count` und `mybuf`, deren Werte erhalten bleiben und bei jedem Aufruf wieder zur Verfügung stehen. Werden mit `recv()` mehrere Zeilen gelesen, bleibt der jeweilige Rest in `mybuf` erhalten und wird beim nächsten Aufruf der Funktion verarbeitet:

```
int readline(int fd, char *buffer, unsigned int len)
  {
```

```
/* read a '\n' terminated line from socket fd into buffer
 * bufptr of size len. The line in the buffer is terminated
 * with '\0'. It returns -1 in case of error or -2 if the
 * capacity of the buffer is exceeded.
 * It returns 0 if EOF is encountered before reading '\n'.
 * Notice also that this routine reads up to '\n' and overwrites
 * it with '\0'. Thus if the line is really terminated with
 * "\r\n", the '\r' will remain unchanged.
*/
static char *bufptr;
static int count = 0;
static char mybuf[1500];
char *bufx = buffer;
char c;

while (--len > 0)            /* repeat until end of line  */
  {                             /* or end of external buffer */
  count--;
  if (count <= 0)            /* internal buffer empty --> read data */
    {
    count = recv(fd, mybuf, sizeof(mybuf), 0);
    if (count < 0) return -1;/* error or disconnect */
    if (count == 0) return 0;   /* nothing to read - so reset */
    bufptr = mybuf;             /* internal buffer pointer    */
    }
  c = *bufptr++;                /* get c from internal buffer  */
  if (c == '\n')
    {
    *buffer = '\0';             /* terminate string and exit  */
    return buffer - bufx;
    }
  else
    {
    *buffer++ = c;              /* put c into  external buffer */
    }
  }
return -2;                      /* external buffer to short */
}
```

Beim Senden von Daten sollte man eigentlich nicht zwischenpuffern, sondern jede Zeile sofort auf die Reise schicken - schließlich wartet der Empfänger darauf.

1.8 Was noch zu beachten ist

Die Programmierung von Netzwerkanwendungen kann mitunter recht komplex werden; auf jeden Fall komplexer als die Beispiele in diesem Skript. Einige Fallen und Fußangeln sollen hier besprochen werden. In den Beispielen wurden lediglich Zeichenketten gesendet und empfangen. Meist liegen die Daten aber nicht in einem derartigen, für uns Menschen lesbaren Format vor. Falls Ganzzahlen oder Gleitkommazahlen zu übertragen sind, muss auf jeden Fall auf die interne Darstellung der Daten geachtet werden (big endian, little endian). Auch wissen Sie nicht unbedingt, was geschieht, wenn Sie von einem 64-Bit-Rechner eine Integer-Zahl an einen 32-Bit-Rechner verssenden. Sie wissen fast nie genau, welche Größe die Datentypen short, int, long, float oder double auf der Gegenseite haben. Besser wandeln Sie die Binärformate vor dem Senden in Zeichenketten um, am besten mit `sscanf()`, `strtol()` und `snprintf()`.

Auch bei binären Strukturen sollte man die komplette Struktur vor dem Senden in eine Zeichenkette konvertieren. Bei komplexen Strukturen würde sich XML als Beschreibungssprache anbieten, insbesondere da es für die XML-Konvertierung fertige Bibliotheken gibt. Auch auf Empfangseite müssen selbstverständlich ebenfalls bestimmte Vorkehrungen getroffen werden. Zusätzlich müssen Sie nationale Umlaute berücksichtigen. Durch die Umwandlung in Strings wird zwar etwas Bandbreite verschwendet, aber dafür sind die übertragenen Informationen lesbar und man kann ggf. einen Telnet-Client für das Debugging einsetzen.

Pufferüberläufe waren (Buffer Overflows) in der Geschichte der C-Programmierung schon immer eine der häufigsten Fehlerursache, weil sie so einfach zu programmieren sind und weil man sie sehr gerne übersieht. Bei einem Pufferüberlauf werden im Grunde ganz einfach mehr Daten in einen Puffer geschrieben, als die Puffergrösse zuläßt. Wenn die Daten, die überlappen, von einer bestimmten Beschaffenheit sind, dann kann der Angreifer damit unter Umständen Schadcode einschleusen und ausführen lassen. Die Socket-Funktionen (z. B. `recv()`) besitzen alle einen Parameter, der die Datenmenge im Zielpuffer begrenzt. Wenn man hier den richtigen Wert verwendet, kann eigentlich nichts passieren. Wenn man Text erwartet, ist es günstig hier ein Byte weniger anzugeben, um den String im Puffer anschließend mit '\0' zu terminieren.

Potentiell anfällige Funktionen sind z. B. `strcpy()`, `strcat()` und `sprintf()`, die keinen Parameter zur Längenbegrenzung besitzen. Sie sollten deshalb nur dort verwendet werden, wo ganz sicher nichts passieren kann. In C90 (ANSI-C) gab es bereits teilweise Abhilfe in Form von `strncpy()` und `strncat()`, die eine Grössenbeschränkung besitzen. Neu in C99 hinzugekommen ist `snprintf()`. Diese Funktionen sollten bevorzugt verwendet werden. Man darf übrigens einem Puffer wirklich nur so weit vertrauen, wie man ihn selbst mit Inhalt gefüllt hat - insbesondere, was Null-Terminierung von Strings angeht.

Zeilen werden in der Netzwerkwelt zumeist mit Carriage Return **und** Linefeed (\r\n) abgeschlossen. Das kann zum seltsamen Verhalten des Programms führen, wenn man beispielsweise nur das Linefeed (Newline) abschneidet, oder mit einem Server kommunizieren will, und der einfach so tut, wie er soll.

1.9 Der Internet-Superserver

Rein theoretisch müßte jeder Daemon bei Systemstart hochgefahren werden, für die eine Anforderung eines entfernten Rechner (Host) auftreten könnte. Dies würde aber die Zahl der laufenden Prozesse unnötig in die Höhe treiben und Systemresourcen verbrauchen. Deshalb wurde der Daemon `inetd`, der Internet-Superserver, entwickelt. Er "lauscht" auf alle Diensteanforderungen, die an dem von ihm überwachten Ports eingehen. Tritt eine solche Anforderung auf, prüft der Daemon die Zugriffsberechtigung (exakt: Die Kontrolle wird an den TCP-Warapper `tcpd` übergeben und dieser macht weiter) und startet im positivem Fall den entsprechenden Daemon, der dann die Anforderungen des Clients bearbeitet. Die Konfigurationsdatei ist `/etc/inetd.conf`. In ihr sind alle Dienste und die entsprechenden Dämonen mit Parametern verzeichnet.

Der `inetd` vereinfacht zudem das Schreiben von Server-Daemonen, da etliche Start-Details bereits durch den `inetd` selbst abgehandelt werden. Der Nachteil besteht darin, daß der `inetd` für jede Anfrage sowohl ein `fork` als auch ein `exec` ausführen muß, um den aktuellen Serverprozeß zu starten. Der Ablauf entspricht in etwa folgendem Schema:

1. Beim Starten liest der Daemon die Datei `/etc/inetd.conf` und generiert für jeden der angegebenen Server einen Socket.
2. Danach wird für jeden Socket ein `bind()` ausgeführt.
3. Für jeden Stream-Socket wird nun ein `listen()` ausgeführt.
4. Nun wird auf einen Verbindungswunsch von außen gewartet (per `select()`-Aufruf).
5. Kommt ein Verbindungswunsch, wird er mit `accept()` angenommen.
6. Nun erzeugt der `inetd` einen Kondprozeß zum Bearbeiten der Anforderung. Das Kind schließt alle Dateideskriptoren (außer dem Socket). Mittels `dup2()` wird der Socket dupliziert und Deskriptoren für `stdin`, `stdout` und `stderr` angelegt. Anschließend wechselt der Prozeß die Benutzeridentität und startet schließlich mittels `exec()` den Prozeß für den gewünschten Dienst.
7. Bei eine Stream-Socket wird der angeschlossene Socket geschlossen.

Der `inetd` und die von ihm gestarteten Server stüten sich auf folgenden Dateien.

- Die Datei `/etc/inetd.conf` hat folgende Zeilenstruktur:

  ```
  Programmname Sockettyp Protokoll Flags User Programmpfad Programmargumente
  ```

  - Der Programmname ist frei wählbar, aber es ist darauf zu achten, dass der gleiche Name auch in der Datei `/etc/services` benutzt wird.
  - Als Sockettyp kommen `stream`, `dgram`, `raw`, `rdm` und `seqpacket` in Frage.
  - Das Protokoll gibt eines der in der Datei `/etc/protocols` aufgelisteten Protokolle an.
  - Mögliche Flags sind `wait, nowait, udp` und `tcp`.
  - Danach folgt der User, unter dessen Rechten das Programm ausgeführt wird.
  - Der Programmpfad gibt an, wo das Programm zu finden ist.
  - Schließlich folgen eventuelle Parameter für das Programm.

  Beispiel:

  ```
  ...
  ftp     stream  tcp  nowait  root  /usr/sbin/in.ftpd     in.ftpd
  telnet  stream  tcp  nowait  root  /usr/sbin/in.telnetd  in.telnetd
  ...
  shell   stream  tcp  nowait  root  /usr/sbin/in.rshd     in.rshd
  login   stream  tcp  nowait  root  /usr/sbin/in.rlogind  in.rlogind
  exec    stream  tcp  nowait  root  /usr/sbin/in.rexecd   in.rexecd
  ...
  #tftp    dgram   udp  wait    root  /usr/sbin/in.tftpd    in.tftpd -s /tftpboot
  ...
  ```

  Nach Änderungen der Konfiguration (z.B. Ein- und Ausschalten des TFTP-Dienstes durch Entfernen bzw. Hinzufügen des Kommentarzeichens) muß der `inetd` nicht neu gestartet werden, sondern kann durch das Versenden des Signal `SIGHUP` angewiesen werden, sich neu zu konfigurieren.

  Das Hauptproblem beim Einsatz des `inetd` ist die Reaktionszeit auf Dienstanforderungen. Das ist bei Diensten wie Telnet oder FTP unproblematisch, kann aber z.B. bei HTTP-Servern sehr kritisch werden, da ja HTTP für jede einzelne Datein eines Webdokuments eine Verbindung aufbaut, demnach für jede einzelne Datei der `httpd` vom `inetd` neu gestartet werden muß. Daher werden zeitkritischen Dienste und Server als Standalone-Server betrieben.

- Die Datei `/etc/services` hat folgende Zeilenstruktur:

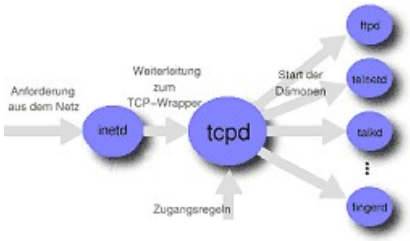
  ```
  Programmname Port/Protokoll
  ```

`Programm` muß der gleiche Name sein wie er in `/etc/inetd.conf` an erster Position angegeben wurde. `Port/Protokoll` bezeichnet den Port, auf dem gelauscht werden soll, und das dazugehörigen Protokoll, z. B.:

```
37 telnet/tcp
```

- Der TCP-Wrapper und die Dateien `/etc/hosts.allow` und `/etc/hosts.deny`:

  Im `inetd` selbst sind keinerlei Sicherheitsmechanismen implementiert. Deswegen wurde ein Filter geschaffen, der vom `inetd` anstelle des für den Port zuständigen Dienstes gestartet wird, der sogenannte TCP-Wrapper `tcpd`. Er erhält den Programmnamen des Dienstes als Argument. Der `tcpd` arbeitet für Server und Client transparent, er wird durch den zu startenden Dienst ersetzt. Zuvor protokolliert und überprüft er die Zulässigkeit des Zugriffs.



  Falls also in `inetd.conf` an der vorletzten Position zusätzlich noch der tcp-Wrapper (`tcpd`) aufgerufen wird (also beispielsweise statt `/etc/ftpd` nun `/etc/tcpd /etc/ftpd`), wird vom `tcpd` vor dem Programmaufruf geprüft, ob der entsprechende Host überhaupt das Recht besitzt, Serverdienste in Anspruch zu nehmen. Dazu werden die Dateien `/etc/hosts.allow` und `/etc/hosts.deny` herangezogen.

  - In keiner der beiden Dateien verzeichnete Hosts, Domains oder Dienste werden immer zugelassen.
  - `/etc/hosts.deny`
    Hier verzeichnete Hosts, Domains oder Dienste werden **nicht** zugelassen. Will man nur einige Bösewichte aussperren, wird man diese in dieser Datei eintragen uns alles andere zulassen, z.B.:

    ```
    # Host sperren
    192.168.0.2: ALL
    # Domain sperren
    .boese.org: ALL
    ```

    Sicherer ist es jedoch, in dieser Datei alles zu sperren, was nicht ausdrücklich erlaubt ist:

    ```
    ALL: ALL
    ```

  - `/etc/hosts.allow`
    Hier verzeichnete Hosts, Domains oder Dienste werden zugelassen. Man kann wahlweise Dienste sperren oder Hosts und Domains nach IP-Nummer oder Namen. Als Beispiel eine aktuelle Datei:

    ```
    # See tcpd(8) and hosts_access(5) for a description.
    sshd:           ALL     : ALLOW
    proftpd:        ALL     : ALLOW
    sendmail:       ALL     : ALLOW
    popper:         ALL     : ALLOW
    in.telnetd:     localhost : ALLOW
    bcpd:           e-technik.fh-muenchen.de, ariacenter.rz.fh-muenchen.de : ALLOW
    ```

    Die Daemons für SSH, FTP, SMTP und POP sind für alle offen, Telnet ist nur am lokalen Rechner erlaubt (Das ":ALLOW" kann man auch weglassen) und der Backup-CLient (`bcpd`) darf nur innerhalb des Fachbereichs und vom Backupserver des Rechenzentrums angesprochen werden.

Eine Zeile in den beiden Dateien hat folgenden Aufbau:

```
Dienst : Rechner : [Kommando]
```

An Stelle von `Dienst` kann entweder ein `Programmname` stehen oder das Schlüsselwort `ALL`, falls die Zeile alle Dienste betreffen soll. Einem `ALL` kann ein `EXCEPT Dienstname` folgen, dann sind alle Dienste mit Ausnahme der benannten gemeint. Per Komma getrennt, lassen sich mehrere Dienste angeben. Die möglichen Einträge sind Rechnernamen, IP-Adressen oder die folgenden Schlüsselworte:

- ALL: Alle Rechner
- KNOWN: Rechner, deren Namen der `tcpd` ermitteln kann (DNS und reverse DNS)
- LOCAL: Alle Rechner, deren Namen keinen Punkt enthalten (Rechner, die in `/etc/hosts` unter einem Kurznamen aufgeführt sind)
- UNKOWN: Rechner, deren Namen der `tcpd` nicht ermitteln kann.

- PARANOID: Alle Rechner, deren Namens- und Adressauflösung über DNS widersprüchliche Angaben ergibt.

Schließlich kann ein optionales Kommando angegeben werden, das immer dann ausgeführt wird, wenn diese Zeile für eine Anforderung zutrifft. Falls auf diese Möglichkeit zurückgegriffen wird, dann meist für ein detailliertes Protokoll. Im Argument des Kommandos können einige Sonderzeichen verwendet werden:

%a liefert die IP des rufenden Rechners
%c gibt den Namen des Nutzers und Rechners zurück (sofern ermittelbar)
%d Name des gewünschten Dienstes
%h Name des zugreifenden Rechners oder dessen IP
%n Name des zugreifenden Rechners (oder unknown oder paranoia)
%p Prozessnummer des Dienstes
%s Name des Serves in Verbindung mit dem Rechnernamen
%u Name des Nutzers auf Clientseite, sofern er ermittelt werden kann

Die Konfiguration kann mit dem Kommando `tcpdchk` überprüft werden (Unstimmigkeiten in `inetd.conf`, Syntaxfehler in `hosts.deny / hosts.allow`, unbekannte Rechnernamen).

1.10 Einen Daemon beschwören

Unter einem Daemon versteht man bei UNIX einen Prozess, der im Hintergrund arbeitet. Bei Wondows heisst so etwas "Dienst". Es ist oft sinnvoll, Server-Prozesse als Daemon einzurichten. Programmnamen für Daemons enden in der Regel auf 'd' (sshd, httpd, inetd etc.).

In Gegensatz zu einem "normalen" Programm, das an ein Terminal gebunden ist, löst sich der Daemon nach dem Start vom Terminal und läuft unabhängig im Hintergrund weiter. Normalerweise wird der Prozess auch vom init-Prozess adoptiert. An die Stelle der Terminalausgabe tritt oft ein Logfile; der Daemon kann aber auch bequemerweise den Syslog-Mechanismus nutzen.

Um einen Daemon zu erzeugen, werde die Funktionen `fork()`, `setsid()` und `chdir()` benötigt. Mittels `chdir()` bekommt der Daemon sein Standardverzeichnis zugewiesen. Hier können "/" oder "/tmp" verwendet wewrden, wenn man kein spezielles Verzeichnis vorsehen will. Auch müssen die Standarddateien `stdin`, `stdout` und `stderr` geschlossen werden,damit sich der Daemon vom Terminal löst. Normelerweise werden sie nicht einfach geschlossen, sondern auf "/dev/null" umgeleitet. Das folgende Beispielprogramm zeigt, wie man den Übergang zum Daemon programmiert, macht aber sonst nichts Sinnvolles.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/fcntl.h>
#include <unistd.h>

int start_daemon(void)
  /* macht das ganze daemon-Zeugs */
  {
  if (chdir ("/tmp") < 0)      /* Basisverzeichnis des Daemons */
    return -1;
  if (setsid ( ) < 0)          /* Erzeugen einer neuen Prozessgruppe */
    return -1;
  umask(0);                    /* unmask the file mode */
  close(STDIN_FILENO);         /* Standarddateien schliessen */
  close(STDOUT_FILENO);
  close(STDERR_FILENO);

  /* Standarddateien auf /dev/null umleiten */
  open("/dev/null", O_RDWR);  /* stdin */
  dup(STDOUT_FILENO);          /* stdout */
  dup(STDERR_FILENO);          /* stderr */

  /* hier kommt dann die grosse Daemon-Magie */
  for(;;)
    {
    sleep(10);
    }
  }

int main(void)
  {
  pid_t pid;     /* Prozess-Id des Kindes */

  if ((pid = fork()) < 0)
    {
    perror("fork() ging schief");
    return 1;
```

```
    }

  if (pid == 0)  /* dies ist der Kindprozess */
    {
    if (start_daemon() < 0)
      {
      perror("start_daemon() ging schief");
      return 1;
      }
    }
  else
    printf("Daemonprozess gestartet, ID %i\n", pid);
  return 0;
  }
```

Siehe auch Linux Daemon Howto.

Auch sind im Zusammenhang mit Daemon-Prozessen die Funktionen `setuid()` (Set User Id) und `setgid()` (Set Group Id) interessant, um die Rechte einzuschränken. Wenn man seinen Daemon nämlich beim Bootvorgang mittels einer Startdatei hochfährt, läuft er natürlich mit "root"-Identität - kann also im Fehlerfall größtmöglichen Schaden anrichten. Daher ist es sinnvoll, möglichst bald eine weniger privilegierte Identität anzunehmen. Das alles und noch einiges mehr ist im Programm dummy_daemon.c ausführlicher skizziert. Bei den Stellen im Programm, die noch geändert bzw. ergänzt werden müssen, beginnen die Kommentare mit `/***`.

1.11 Server ganz ohne Programmieren

Schon vor Jahren hat der Programmierer R. Tudorica gezeigt, wie man mit dem Programm Netcat (`nc`) und einigen Zeilen Shellscript mal schnell einen Server aufsetzen kann:

```
#!/bin/bash
while :
  do
  { echo -e 'HTTP/1.1 200 OK\r\n' ; cat <Pfad/Datei> } \
  | nc -l 8000
  done
```

Wird das Script auf dem eigenen Server gestartet, kann jeder andere, für den der Server erreichbar ist, mit dem Browser über die URL `http://<hostname>:8000` die Datei herunterladen. Das kann man auf dem Server sogar beobachten, da Netcat seine Status-Infos auf der Standardausgabe ausgibt.

Zum Inhaltsverzeichnis Zum nächsten Abschnitt
