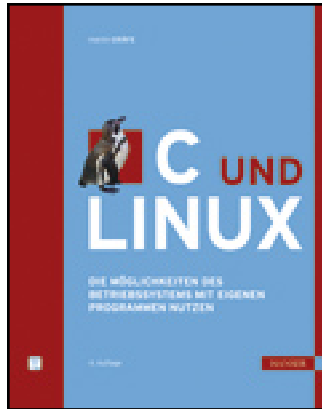
HANSER



Leseprobe

Martin Gräfe

C und Linux

Die Möglichkeiten des Betriebssystems mit eigenen Programmen nutzen

ISBN: 978-3-446-42176-9

Weitere Informationen oder Bestellungen unter

http://www.hanser.de/978-3-446-42176-9

sowie im Buchhandel.

© Carl Hanser Verlag, München

Kapitel 7

Netzwerkprogrammierung

Seit den Anfängen von Linux vor mehr als 10 Jahren ist die Netzwerkkommunikation[1] fester Bestandteil des Betriebssystems. Während zu dieser Zeit auf Windows^{TM} 3.11-Rechner diverse proprietäre Netzwerkprotokolle aufgesetzt wurden, war Linux von Haus aus in der Lage, über den Standard TCP/IP mit Workstations, Großrechnern und Servern in Rechenzentren zu kommunizieren. Im Bereich der PCs und Workstations wurden die proprietären Lösungen fast vollständig von TCP/IP verdrängt, sodass inzwischen die unterschiedlichen Betriebssysteme in einem Netzwerk „die gleiche Sprache“ sprechen.

Möglicherweise fragen Sie sich an dieser Stelle, ob Sie sich mit der komplexen Thematik Netzwerkprogrammierung auseinandersetzen sollen, weil Sie vielleicht gar nicht beabsichtigen, mehrere Computer mit eigenen Programmen zu „vernetzen“. Bei Linux ist jedoch die Netzwerkkommunikation mehr als ein Hilfmittel zum Datenaustausch zwischen zwei Computern; sie ist die logische Fortsetzung der Interprozesskommunikation, und viele Teile des Systems bauen darauf auf. Wenn Sie auf dem Desktop Ihres Linux-PCs ein Fenster schließen oder eine Schaltfläche anklicken, wird dies dem betreffenden Programm von der grafischen Oberfläche X11 über Funktionen zur Netzwerkkommunikation mitgeteilt. Aus diesem Grund ist es sinnvoll, dieses Thema selbst bei „Stand-Alone-Systemen“ ohne einen Netzwerkanschluss zu betrachten.

Die Thematik der Netzwerkprogrammierung ist deutlich umfangreicher als beispielsweise die Interprozesskommunikation. Eine vollständige und detaillierte Beschreibung umfasst leicht mehrere hundert Seiten. Deshalb konzentriert sich dieses Kapitel auf die geläufigsten Funktionen und Methoden, die Sie immerhin in die Lage versetzen werden, einen eigenen Webserver zu programmieren!

---

[1] Netzwerk*kommunikation* wäre die passendere Bezeichnung für dieses Kapitel. In der englischsprachigen Literatur hat sich jedoch der Begriff „Network Programming“ etabliert und wurde später wörtlich ins Deutsche übersetzt.

7.1 Einführung

Ähnlich wie bei den vorangegangenen Kapiteln soll auch hier der Einstieg in die Thematik anhand kleiner Beispielprogramme erleichtert werden. Um die einzelnen Schritte in den Programmen nachvollziehen zu können, sind jedoch ein gewisses Grundlagenwissen und etwas Theorie unerlässlich. Insbesondere sollen die im Umfeld der Netzwerkkommunikation auftauchenden Begriffe sowie das Prinzip der Kommunikation über eine Netzwerkverbindung erläutert werden.

7.1.1 Begriffe

Ethernet

Der Begriff „Ethernet“ wird häufig als Synonym für eine Netzwerkverbindung verwendet. Tatsächlich beschreibt Ethernet ein Verfahren, wie mehrere Teilnehmer auf eine gemeinsame Netzwerkleitung zugreifen können – es handelt sich dabei also um ein *Zugriffsverfahren*. Ursprünglich bestand diese Netzwerkleitung aus einem Koaxialkabel (ähnlich einem Antennenkabel), an das alle Computer eines Netzwerksegments über je ein T-Stück angeschlossen waren. Damit waren Datenraten bis 10 MBit/s möglich.

Heute findet man eine solche Verkabelung nur noch selten. Das Koaxialkabel wurde weitestgehend von der bekannten 8-adrigen „CAT 5“-Leitung verdrängt, und die Netzwerkstruktur ist heutzutage in der Regel sternförmig, mit *Hubs* und *Switches* in den Sternpunkten.

Ethernet definiert also weder die Art der Leitung noch die des Protokolls (s. u.), das über die Leitung übertragen wird.

Protokoll

Damit sich die Teilnehmer eines Netzwerkes untereinander „verstehen“, muss in einem Protokoll festgelegt sein,

- wie ein „Datenpaket“ auf der Netzwerkverbindung aussieht;
- wie der Empfänger des Datenpaketes „adressiert“ wird;
- ob und wie auf Übertragungsfehler reagiert wird;
- …

Einige solcher Protokolle sind:

IP – *Internet Protocol* (regelt die Adressierung der Netzwerkteilnehmer);

TCP – *Transmission Control Protocol* (definiert die gesicherte[1] Datenübertragung zwischen zwei Netzwerkteilnehmern);

[1] „gesichert“ bedeutet hier: Es können keine Daten verloren gehen.

UDP – *User Datagram Protocol* (regelt die Datenübertragung ähnlich TCP, jedoch nicht gesichert);

FTP – *File Transfer Protocol* (definiert die Übertragung von Dateien);

telnet – Protokoll zum Login eines Benutzers auf einem entfernten Rechner;

HTTP – *Hyper-Text Transfer Protocol* (beschreibt den Zugriff auf Web-Seiten);

SMTP – *Simple Mail Transfer Protocol* (regelt die E-Mail-Übertragung);

PPP – *Point to Point Protocol* (definiert die Netzwerkverbindung zwischen zwei Teilnehmern über eine Punkt-zu-Punkt-Verbindung inkl. Authentifizierung und Zuweisung einer IP-Adresse).

In der Regel trifft man auf eine Verschachtelung mehrerer Protokolle. So „laufen" über ein serielles Kabel zwischen PC und analogem Modem beim Aufrufen einer Internet-Seite gleichzeitig die Protokolle HTTP, TCP, IP und PPP.

Bei den meisten PC-Netzwerken wie auch im Internet kommt die Kombination TCP und IP zum Einsatz, kurz als TCP/IP bezeichnet. Das IP sorgt dafür, dass die Daten zum richtigen Teilnehmer gelangen, das TCP stellt sicher, dass alle Datenpakete fehlerfrei und in der richtigen Reihenfolge ankommen. Die Beispiele in diesem Kapitel beziehen sich alle auf TCP/IP-Verbindungen.

In Abschnitt 7.3.3 werden die Grundlagen des HTTP beschrieben.

Port

In TCP/IP-Netzwerken werden die Netzwerkteilnehmer über ihre IP-Adresse adressiert. Doch wie funktioniert es, dass die Daten des Web-Servers (also die HTML-Seiten) im Browser landen, die E-Mails im E-Mail-Client ankommen, die Dateien vom FTP-Server zum FTP-Client (z. B. xftp) übertragen werden und Benutzername und Passwort bei einem „remote login" (`rlogin` oder `telnet`) an der richtigen Stelle ankommen? Dies ist in ähnlicher Weise gelöst, wie auch die richtigen Daten an Drucker, Modem, Monitor, Tastatur, Maus und Scanner ankommen: Die Geräte hängen an unterschiedlichen Anschlüssen oder auch *Ports* des Computers. Ebenso enthalten die TCP/IP-Datenpakete neben der IP-Adresse auch eine Port-Nummer, also eine Information, für welchen „Anschluss" des Computers die Daten bestimmt sind (Abbildung 7.1).

Es gibt eine ganze Reihe von Port-Nummern, die fest für bestimmte *Dienste* vergeben sind, beispielsweise:

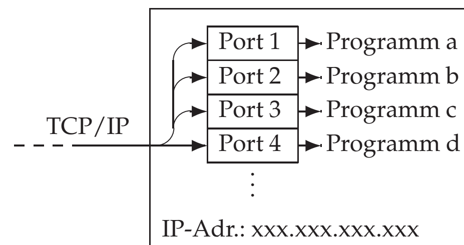


Abbildung 7.1: Aufteilung der IP-Verbindungen auf verschiedene Ports

Port	Dienst
21	FTP-Server
23	telnet-Server
25	SMTP-Server (E-Mail-Dienst)
53	Domain Name Server
79	„Finger"-Server (Infos über Benutzer)
80	Web-Server (HTTP-Server)
6000	X11-Server

Eine umfangreiche Liste der (möglichen) Dienste und deren Port-Nummern finden Sie in der Datei „`/etc/services`".

Verbindung (Connection)

Unter „Verbindung" versteht man in der Netzwerkkommunikation nicht die *physikalische* Leitung zwischen zwei Computern (also das Kabel), sondern den *Datenpfad*, der beispielsweise zwischen einem Browser und einem Webserver aufgebaut wird.

Neben der *verbindungsorientierten* Kommunikation, wie sie in diesem Kapitel behandelt wird, gibt es auch *verbindungslosen* Datentransfer im Netzwerk. Ist z. B. einem Netzwerkteilnehmer die Adresse des *Domain Name Servers* (DNS) noch nicht bekannt, fordert er diese mit Hilfe einer *Broadcast*-Nachricht an, ohne zuvor eine Verbindung aufzubauen. Verbindungslose Kommunikation hat also nichts mit Wireless LAN zu tun.

7.1.2 Vorbereitung

Die Firewall

Auf aktuellen Linux-Systemen ist in der Regel eine *Firewall* installiert und auch aktiviert. Diese blockiert aus Sicherheitsgründen verschiedene Netzwerkaktivitäten, ohne dass Warn- oder Fehlermeldungen ausgegeben werden. Dadurch laufen einige der folgenden Beispielprogramme bei aktivierter Firewall nicht oder nicht korrekt. Aus diesem Grund sollten Sie zunächst prüfen, ob eine Firewall aktiv ist

und diese dann ggf. deaktivieren.[1] Bei SuSE-Linux geht das mit Hilfe des Konfigurationstools YAST. Wählen Sie dazu in der Rubrik „Sicherheit und Benutzer" den Punkt „Firewall" (Abbildung 7.2).

Abbildung 7.2: Konfiguration der Firewall mit YAST unter SuSE

Abbildung 7.3: Abschalten der Firewall

[1] Aus Sicherheitsgründen sollten Sie bei Versuchen mit eigenen Netzwerkprogrammen und abgeschalteter Firewall keine Verbindung mit dem Internet haben!

Es erscheint dann das Fenster zur Konfiguration, in dem Sie die Schaltfläche „Firewall nun stoppen“ wählen (Abbildung 7.3). Danach können Sie die Konfiguration mit der entsprechenden Schaltfläche abbrechen. Wenn Sie die Einstellung „Service starten: Bei Systemstart“ aktiviert lassen, wird die Firewall automatisch beim nächsten Neustart des Rechners wieder aktiv.

Netzwerkdienste aktivieren

Linux stellt bereits einige Netzwerkdienste zur Verfügung, von denen einige in dem „xinetd“ zusammengefasst sind. Dieser Dämon ist in der Grundkonfiguration häufig deaktiviert. Da die im xinetd enthaltenen Dienste aber gerade für erste Versuche mit Netzwerkprotokollen hilfreich sind, sollten sie den Dämon aktivieren (siehe Abbildung 7.4, „Netzwerkdienste (xinetd)“). Abbildung 7.5 zeigt einen Teil der Dienste, die vom xinetd bereitgestellt werden. Hier sollten Sie „finger“, „echo“ und „ftp“ für die Beispiele in den folgenden Abschnitten aktivieren.

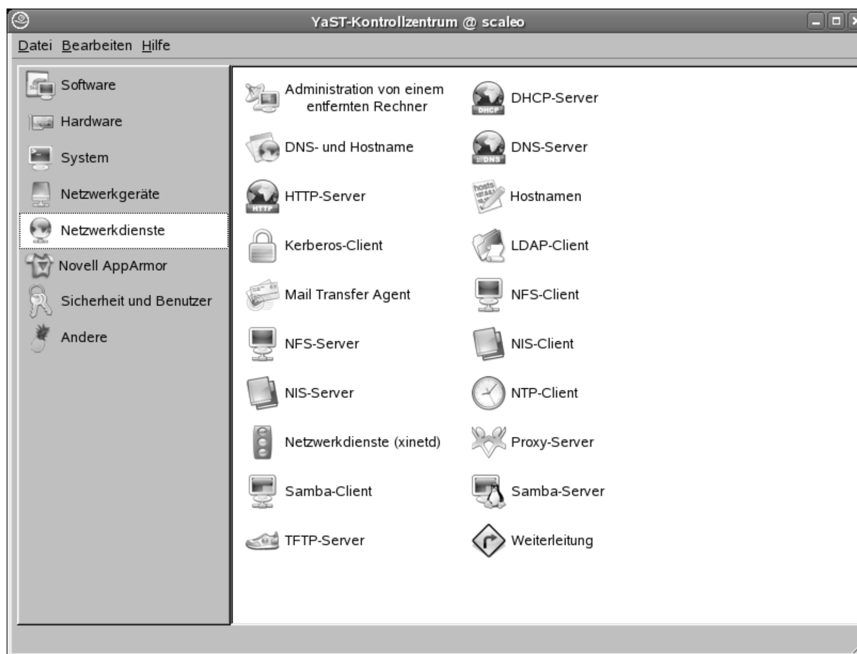


Abbildung 7.4: Konfiguration der Netzwerkdienste mit YaST

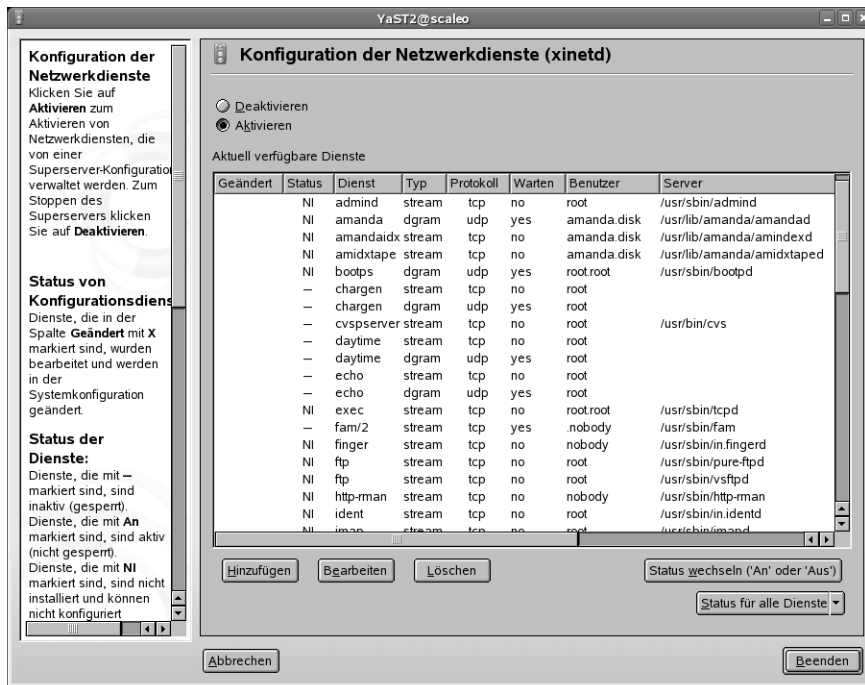


Abbildung 7.5: Aktivierung der „xindet"-Dienste

7.1.3 Das Client-Server-Prinzip

Bei der verbindungsorientierten Netzwerkkommunikation, wie sie in den folgenden Abschnitten beschrieben wird, tritt jeweils ein Teilnehmer als Server und der andere als Client auf. Die Abläufe (Funktionsaufrufe) zum Verbindungsaufbau sind bei Client und Server ganz unterschiedlich: Der Server meldet seinen Dienst in der Regel mit einer festen Port-Nummer an (z. B. belegt ein Webserver den Port Nr. 80). Der Client (Browser) baut eine Verbindung zum Server auf, indem er diesen mit der IP-Adresse und der Port-Nummer adressiert.

Sobald die Verbindung aufgebaut ist, wird dem Client automatisch ebenfalls eine Port-Nummer zugewiesen, die in der Regel unabhängig von der Port-Nummer des Servers ist (siehe Abbildung 7.6).

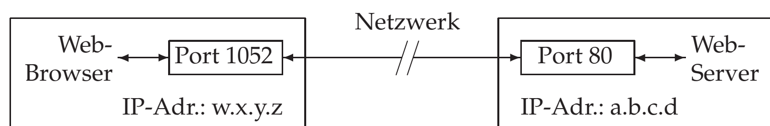


Abbildung 7.6: Verbindungsorientierte Kommunikation zwischen Client und Server

Damit können beide Applikationen, Client und Server, jeweils eindeutig über ihre IP-Adresse und Port-Nummer adressiert werden.

Übrigens: Client und Server können natürlich auch auf dem gleichen Computer laufen, also die gleiche IP-Adresse haben.

7.1.4 Sockets

Basis der Netzwerkkommunikation – ob verbindungsorientiert oder verbindungslos – bilden die *Sockets* (zu Deutsch: „Sockel“, „Steckdosen“). Sowohl Client als auch Server müssen zunächst einen Socket öffnen, bevor eine Verbindung aufgebaut werden kann. Zum Öffnen eines Sockets dient die gleichnamige Funktion

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

Dabei gibt der Parameter `domain` die *Protokollfamilie* für die Kommunikation über diesen Socket an. Für TCP/IP-Verbindungen ist das die Konstante `PF_INET`. Der zweite Parameter legt die Art der Kommunikation fest, für verbindungsorientierte Kommunikation ist hier die Konstante `SOCK_STREAM` zu wählen. Der dritte Parameter, der das zu benutzende Protokoll bestimmt, kann in der Regel auf 0 gesetzt werden, wodurch automatisch das zu Protokollfamilie und Kommunikationsart passende Protokoll gewählt wird.

Als Rückgabewert liefert `socket()` einen Dateideskriptor (oder —1 im Fehlerfall) – ein Socket ist also quasi eine Datei. Daher wird er wie eine Datei wieder geschlossen:

```
int close(int sock_fd);
```

Damit sieht das „Rahmenprogramm“ für die Netzwerkkommunikation wie folgt aus:

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

int main(int argc, char *argv[])
 {
  int sock_fd;
  ...
  sock_fd = socket(PF_INET, SOCK_STREAM, 0);
  if (sock_fd == -1)
    perror("socket() failed");
  ...
  close(sock_fd);
 }
```

Zum Lesen aus oder Schreiben in einen Socket können Sie – wie bei Devices – die Funktionen `read()` und `write()` verwenden. Darüber hinaus gibt es für

die Netzwerkkommunikation spezielle Funktionen, die zusätzliche Möglichkeiten bieten:

```
int send(int sock_fd, void *data, size_t len, int flags);
int recv(int sock_fd, void *buffer, size_t len, int flags);
```

Doch bevor wir überhaupt Daten über einen Socket lesen oder schreiben können, muss eine Verbindung zum Socket des Kommunikationspartners hergestellt werden. Weil dies bei Client und Server unterschiedlich ist, wird in den folgenden Abschnitten zunächst ein typischer Client und anschließend ein einfacher Server vorgestellt.

7.2 Der TCP/IP-Client

Für die Demonstration von Netzwerkprogrammierung benötigt man sowohl ein Client- wie auch ein Server-Programm. Glücklicherweise ist Linux von Haus aus mit einer Reihe von Server-Programmen auf TCP/IP-Basis ausgestattet, sodass wir uns zunächst auf das Programmieren eines TCP/IP-Clients konzentrieren können.

7.2.1 Aufbau einer Verbindung

Als erster Schritt muss, wie bereits erwähnt, ein Socket geöffnet werden. Danach kann das Client-Programm diesen Socket mit einem Socket des Server-Programms verbinden. Dies geschieht mit der Funktion `connect()`:

```
int connect(int sock_fd, struct sockaddr *serv_addr,
            socklen_t addrlen);
```

War der Verindungsaufbau erfolgreich, gibt `connect()` eine 0 zurück, andernfalls –1. Als ersten Parameter erwartet `connect()` den Dateideskriptor des Sockets, der verbunden werden soll. Danach folgen als zweiter und dritter Parameter die (Internet-)Adresse des zu kontaktierenden Servers und die Länge der Adresse. Als Adresse verwenden wir nicht die in der Deklaration von `connect()` angegebene, allgemein gehaltene Struktur `sockaddr`, sondern die speziell auf IP-Verbindungen abgestimmte Variante `sockaddr_in`:

```
struct sockaddr_in
 {
  sa_family_t        sin_family;  /* Adressfamilie */
  unsigned short int sin_port;    /* Port-Nummer */
  struct in_addr     sin_addr;    /* Internet-Adr. */
  unsigned char      __pad[8];    /* auffuellen */
 }
```

Vor dem Aufruf der Funktion `connect()` müssen in diese Struktur die IP-Adresse und die Port-Nummer des Servers eingetragen werden. Das erste Element gibt dabei die *Adressfamilie* an; für eine IP-Adresse ist das `AF_INET`. Als zweites Element enthält die Struktur die Port-Nummer des Servers. Diese besteht aus zwei Bytes (`unsigned short int`), deren Reihenfolge (High Byte, Low Byte) im Internet-Protokoll festgelegt ist und von der Architektur des Rechners abweichen kann. Um die Port-Nummer korrekt in die Struktur einzutragen, sollte unbedingt die Funktion `htons()` verwendet werden. Analog dazu sollte auch die IP-Adresse mit Hilfe der Funktion

```
int inet_aton(char *text, struct in_addr *addr);
```

in die Struktur kopiert werden, wobei die Zeichenkette `text` die IP-Adresse in der üblichen Schreibweise (z. B. „127.0.0.1“) enthält, die von `inet_aton()` in binäre Form umgewandelt wird. Ist die angegebene IP-Adresse fehlerhaft, liefert `inet_aton()` eine 0 als Rückgabewert.

Um dieser abstrakten, theoretischen Beschreibung etwas Anschauung zu verleihen, soll folgendes Beispiel den Verbindungsaufbau zu Port 80 der IP-Adresse 127.0.0.1 (localhost) demonstrieren:

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

int sock_fd;
struct sockaddr_in server_addr;

server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(80);
inet_aton("127.0.0.1", &(server_addr.sin_addr));

err = connect(sock_fd, (struct sockaddr *)&server_addr,
              sizeof(struct sockaddr_in));
if (err == -1)
  perror("connect() failed");
```

Mit dem „Rahmenprogramm“ von Seite 170 kombiniert, ergibt sich daraus ein vollständiges Programm zum Öffnen eines Sockets und Verbinden des Sockets mit einem Web-Server (falls eingerichtet). Da jedoch keine Daten mit dem Server ausgetauscht werden, ist dieses Programm nicht besonders nützlich – es kann allenfalls feststellen, ob ein entsprechendes Server-Programm eingerichtet und aktiviert wurde.

7.2.2 Ein „Universal“-Client

Das folgende Programm öffnet einen Socket und baut die Verbindung zu einem Server auf. Es erwartet als Kommandozeilenparameter die IP-Adresse und die Port-Nummer des Servers. Nach erfolgreichem Verbindungsaufbau arbeitet das Programm ähnlich einem Terminal-Programm: (Tastatur-)Eingaben werden an den Server geschickt; die Antwort des Servers wird im Shell-Fenster ausgegeben. Mit Ctrl-D kann die Eingabe – und somit auch das Programm – beendet werden.

```
1   /*
2       connect.c - einfacher Netzwerk-Client
3   */
4
5   # include <stdio.h>
6   # include <unistd.h>
7   # include <string.h>
8   # include <sys/types.h>
9   # include <sys/socket.h>
10  # include <netinet/in.h>
11  # include <arpa/inet.h>
12
13  int main(int argc, char *argv[])
14   {
15    static char buffer[256];
16    int sock_fd, err, length, port;
17    struct sockaddr_in server_addr;
18    fd_set input_fdset;
19
20    if (argc != 3)
21     {
22      fprintf(stderr, "Usage: connect ip-addr port\n");
23      return(1);
24     }
25
26    if (sscanf(argv[2], "%d", &port) != 1)
27     {
28      fprintf(stderr, "connect: bad argument '%s'\n",
29              argv[2]);
30      return(1);
31     }
32
33    sock_fd = socket(PF_INET, SOCK_STREAM, 0);
34    if (sock_fd == -1)
35     {
36      perror("connect: Can't create new socket");
```

```
37      return(1);
38     }
39
40    server_addr.sin_family = AF_INET;
41    server_addr.sin_port = htons(port);
42    err = inet_aton(argv[1], &(server_addr.sin_addr));
43    if (err == 0)
44     {
45      fprintf(stderr, "connect: Bad IP-Address '%s'\n",
46              argv[1]);
47      return(1);
48     }
49
50    err = connect(sock_fd, (struct sockaddr *)&server_addr,
51                  sizeof(struct sockaddr_in));
52    if (err == -1)
53     {
54      perror("connect: connect() failed");
55      return(1);
56     }
57
58    while (1)
59     {
60      FD_ZERO(&input_fdset);
61      FD_SET(STDIN_FILENO, &input_fdset);
62      FD_SET(sock_fd, &input_fdset);
63      if (select(sock_fd+1, &input_fdset, NULL, NULL, NULL)
64          == -1)
65        perror("connect: select() failed");
66      if (FD_ISSET(STDIN_FILENO, &input_fdset))
67       {
68        if (fgets(buffer, 256, stdin) == NULL)
69         {
70          printf("connect: Closing socket.\n");
71          break;
72         }
73        length = strlen(buffer);
74        send(sock_fd, buffer, length, 0);
75       }
76      else
77       {
78        length = recv(sock_fd, buffer, 256, 0);
79        if (length == 0)
80         {
```

```
81          printf("Connection closed by remote host.\n");
82          break;
83         }
84        write(STDOUT_FILENO, buffer, length);
85       }
86     }
87
88    close(sock_fd);
89    return(0);
90  }
```

Nach dem Auswerten der Kommandozeilenparameter in den Zeilen 20 bis 31 wird in Zeile 33 ein Socket geöffnet, wiederum mit der Protokollfamilie IP (`PF_INET`) und für verbindungsorientierte Kommunikation (`SOCK_STREAM`). In den Zeilen 40 bis 42 werden IP-Adresse und Port-Nummer des Servers in die Adress-Struktur für den Verbindungsaufbau eingetragen.

Die Verbindung zum Server-Programm erfolgt in den Zeilen 50 und 51 mit dem Aufruf der Funktion `connect()`. In der `while()`-Schleife (Zeile 58 bis 86) wird mit der Funktion `select()` auf Daten von `stdin` (Tastatur) und vom adressierten Server gewartet (siehe auch Seite 110). Die Tastatureingaben werden mit `fgets()` zeilenweise gelesen und mit `send()` zum Server-Programm übertragen (Zeile 68 bis 74). Daten, die der Socket empfängt, werden mit `recv()` eingelesen und mit `write()` ausgegeben (Zeile 78 bis 84).

Zum Test und zur Demonstration soll eine Verbindung zum FTP-Server aufgebaut und dessen Online-Hilfe aufgerufen werden. Die Benutzereingaben sind *schräg* dargestellt – vorausgesetzt, der FTP-Server ist wie in Abschnitt 7.1.2 beschrieben aktiviert (Benutzereingaben sind *schräg* dargestellt):

```
> connect 127.0.0.1 21
220 toshi.at-home FTP server (Version
6.2/OpenBSD/Linux-0.11)
help
214- The following commands are recognized (*
unimplemented).
   USER    PORT    STOR    MSAM*   RNTO    NLST    MKD     CDUP
   PASS    PASV    APPE    MRSQ*   ABOR    SITE    XMKD    XCUP
   ACCT*   TYPE    MLFL*   MRCP*   DELE    SYST    RMD     STOU
   SMNT*   STRU    MAIL*   ALLO    CWD     STAT    XRMD    SIZE
   REIN*   MODE    MSND*   REST    XCWD    HELP    PWD     MDTM
   QUIT    RETR    MSOM*   RNFR    LIST    NOOP    XPWD
214 Direct comments to ftp-bugs@toshi.at-home.
quit
221 Goodbye.
Connection closed by remote host.
```

Neben dem FTP-Server gibt es noch andere Dienste, mit denen Sie „Klartext“ sprechen können: Versuchen Sie einmal, den SMTP-Dienst (Port 25) oder den User-Informationsdienst (Port 79) zu kontaktieren. Bei Letzterem müssen Sie entweder einen Benutzernamen eingeben oder einfach RETURN drücken.

Je nach Übertragungsmedium (Ethernet oder analoges Modem) gibt es unterschiedliche Obergrenzen für die mit einem `send()`-Aufruf übertragenen Daten. Es kann also sein, dass `send()` nicht alle angegebenen Daten überträgt. Der Rückgabewert von `send()` liefert die Anzahl der tatsächlich gesendeten Bytes. Um sicherzugehen, dass alle Bytes zum Server übertragen werden, muss der Rückgabewert mit der zu sendenden Anzahl an Bytes verglichen und die Differenz ggf. mit einem zweiten `send()`-Aufruf übertragen werden.

7.2.3 Rechnernamen in IP-Adressen umwandeln

Bislang haben wir das Ziel immer in Form einer IP-Adresse angegeben, doch diese ist ja nur in seltenen Fällen bekannt. Meistens kennt man den Rechnernamen oder den Domain-Namen z. B. in der Form „`www.hanser.de`“.

Zur Auflösung des Domain-Namens in eine IP-Adresse – ggf. unter Zuhilfenahme eines Domain Name Servers – dient die Funktion `gethostbyname()`:

```
struct hostent *gethostbyname(char *name);
```

die als Rückgabewert einen Zeiger auf die Struktur `hostent` liefert:

```
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;   /* list of addresses */
 };
```

Neben den Elementen dieser Struktur ist in den Include-Dateien auch das Element `h_addr` definiert:

```
#define h_addr h_addr_list[0]
```

`h_addr` zeigt somit auf den ersten Eintrag der „list of addresses“ `h_addr_list`. Dieser ist in der Struktur zwar als Zeiger auf `char` deklariert, zeigt aber im Falle einer IP-Adresse auf eine Struktur vom Typ `struct in_addr`, wie sie als drittes Element in der bereits beschriebenen Struktur `sockaddr_in` enthalten ist.

Das folgende Programm erwartet als Kommandozeilenparameter einen Domain- oder Rechnernamen und ermittelt die dazugehörige IP-Adresse.

```
1  /*
2      ip-lookup.c - IP-Adresse einer Domain holen
3  */
4
5  # include <stdio.h>
6  # include <string.h>
7  # include <arpa/inet.h>
8  # include <netdb.h>
9
10 int main(int argc, char *argv[])
11  {
12   struct hostent *host;
13   struct in_addr *host_ip;
14
15   if ((argc != 2) || (strcmp(argv[1], "-h") == 0))
16    {
17     fprintf(stderr, "Usage: ip-lookup domain-name\n");
18     return(1);
19    }
20
21   host = gethostbyname(argv[1]);
22   if (host == NULL)
23    {
24     herror("connect2: Can't get IP-address");
25     return(1);
26    }
27   host_ip = (struct in_addr *) host->h_addr;
28
29   printf("Hostname:\t%s\n", host->h_name);
30   printf("IP-Address:\t%s\n", inet_ntoa(*host_ip));
31
32   return(0);
33  }
```

Das Programm übergibt den Kommandozeilenparameter (also den Domain-Namen) an die Funktion `gethostbyname()`, diese liefert dann die zugehörige `hostent`-Struktur (Zeile 21). In den Zeilen 29 und 30 wird dann der „offizielle“ Name der Domain sowie ihre IP-Adresse ausgegeben. Da die Struktur `hostent` die IP-Adresse nur in binärer Form enthält, wird diese mit Hilfe der Funktion `inet_ntoa()` zuvor in eine Zeichenkette umgewandelt.

Sollten Sie einen Internet-Zugang haben und gerade online sein, können Sie mit dem Programm `ip-lookup` die IP-Adressen beliebiger Domains im Internet erfragen:[1]

```
> ip-lookup www.hanser.de
Hostname:        www.hanser.de
IP-Address:      213.183.13.138
```

Bitte beachten Sie, dass die Struktur `hostent` bei weiteren Aufrufen von `gethostbyname()` unter Umständen überschrieben wird. Daher sollten die benötigten Daten (beispielsweise die IP-Adresse) für die weitere Verwendung – wie den Verbindungsaufbau mit `connect()` – in eine lokale Variable kopiert werden:

```
struct hostent *server;
struct in_addr *server_ip;
struct sockaddr_in server_addr;

server = gethostbyname("www.hanser.de");
server_ip = (struct in_addr *) server->h_addr;
server_addr.sin_addr.s_addr = server_ip->s_addr;
```

7.3 Server-Programme

Nachdem gezeigt wurde, wie ein TCP/IP-Client programmiert wird, soll in den folgenden Abschnitten die Arbeitsweise eines entsprechenden Server-Programms erläutert werden.

7.3.1 Die Funktionsweise eines Servers

Ebenso wie jeder Client benötigt auch ein Server-Programm zunächst einen Socket als Basis für die Netzwerkkommunikation. Die anschließenden Schritte sind jedoch ganz anders als bei einem Client-Programm; Abbildung 7.7 zeigt den prinzipiellen Ablauf mit den zugehörigen Funktionen.

Hier ist die Beschreibung der einzelnen Funktionsaufrufe im Detail:

```
int bind(int sock_fd, struct sockaddr *my_addr,
         socklen_t addrlen);
```

Der `bind()`-Aufruf ist ähnlich dem `connect()`-Aufruf eines Client-Programms, mit dem Unterschied, dass `bind()` als zweiten Parameter die *eigene* Adresse (IP-Adresse + Port-Nummer) erwartet. War der Aufruf erfolgreich, liefert die Funk-

[1] Es gibt natürlich bereits ein Tool unter Linux, das dies (und noch mehr) kann: `nslookup`.

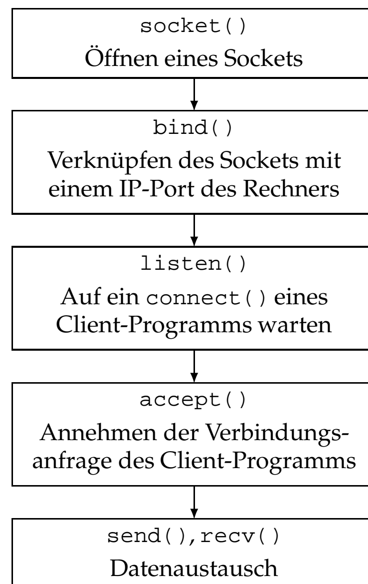


Abbildung 7.7: Prinzipieller Ablauf eines TCP/IP-Servers

tion eine 0 als Rückgabewert. Im Fehlerfall – beispielsweise wenn der Port bereits belegt ist – gibt `bind()` eine –1 zurück.

> Um einen Socket an einen Port zu „binden“, muss das Programm (je nach Portnummer) ggf. über root-Rechte verfügen!
> Nach Beendigung des Programms bleibt der Port noch eine Zeit lang „belegt“

Der nächste erforderliche Funktionsaufruf ist `listen()`:

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

Der Parameter `backlog` gibt an, wie lang die Warteschlange mit `connect()`-Anforderungen an diesem Port maximal werden kann. Auch die Funktion `listen()` gibt bei einem Fehler –1 zurück, sonst 0. Zur Annahme einer Verbindung von einem Client ist schließlich noch die Funktion `accept()` erforderlich:

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

Die Funktion `accept()` wartet, bis die `connect()`-Anforderung eines Client-Programms eintrifft. Dann trägt `accept()` die Adresse des „anklopfenden“ Cli-

ents in die Adress-Struktur ein, die als zweiter Parameter angegeben ist. Die Variable, auf die der dritte Parameter zeigt, muss vor dem `accept()`-Aufruf mit der Länge der Adress-Struktur initialisiert werden. Nach dem Funktionsaufruf enthält die Variable die tatsächliche Länge der Client-Adressinformation.

`accept()` liefert als Rückgabewert den Dateideskriptor eines neuen Sockets, bzw. –1 im Fehlerfall. Dieser neue Socket stellt den Kommunikationskanal zu dem Client dar, dessen Verbindungsanforderung angenommen wurde. Die Kommunikation mittels `send()` und `recv()` findet also nicht über den zunächst mit `socket()` geöffneten Socket statt, sondern über den von `accept()` gelieferten Socket. Indem für jede mit `accept()` angenommene Verbindung ein eigener Socket geöffnet wird, kann der Server mit mehreren Clients parallel kommunizieren – sonst wären Webserver ja undenkbar. Für gewöhnlich wird für jede dieser Verbindungen ein eigener Kind-Prozess gestartet, während der Eltern-Prozess erneut mit `accept()` in Bereitschaft geht, weitere Verbindungen aufzubauen.

7.3.2 Ein interaktiver TCP/IP-Server

Das folgende Server-Programm stellt das Pendant zu dem TCP/IP-Client aus Abschnitt 7.2.2 dar. Als Kommandozeilenparameter muss die Port-Nummer angegeben werden, unter der das Server-Programm seinen Dienst anbietet. Die Port-Nummer darf natürlich nicht bereits von einem anderen Dienst belegt sein.

```
1   /*
2       server.c - interaktiver Netzwerk-Server
3   */
4
5   # include <stdio.h>
6   # include <string.h>
7   # include <stdlib.h>
8   # include <unistd.h>
9   # include <sys/types.h>
10  # include <sys/socket.h>
11  # include <netinet/in.h>
12  # include <arpa/inet.h>
13
14  void err_exit(char *message)
15   {
16    perror(message);
17    exit(1);
18   }
19
20  int main(int argc, char *argv[])
21   {
22    static char buffer[256];
```

```
23      int sock_fd, client_fd, port, err, length;
24      socklen_t addr_size;
25      struct sockaddr_in my_addr, client_addr;
26      fd_set input_fdset;
27
28      if ((argc != 2) || (strcmp(argv[1], "-h") == 0))
29       {
30        fprintf(stderr, "Usage: server port\n");
31        return(1);
32       }
33
34      if (sscanf(argv[1], "%d", &port) != 1)
35       {
36        fprintf(stderr, "server: Bad port number.\n");
37        return(1);
38       }
39                                          /*--- socket() ---*/
40      sock_fd = socket(PF_INET, SOCK_STREAM, 0);
41      if (sock_fd == -1)
42        err_exit("server: Can't create new socket");
43
44      my_addr.sin_family = AF_INET;
45      my_addr.sin_port = htons(port);
46      my_addr.sin_addr.s_addr = INADDR_ANY;
47
48                                          /*--- bind() ---*/
49      err = bind(sock_fd, (struct sockaddr *)&my_addr,
50                 sizeof(struct sockaddr_in));
51      if (err == -1)
52        err_exit("server: bind() failed");
53
54                                          /*--- listen() ---*/
55      err = listen(sock_fd, 1);
56      if (err == -1)
57        err_exit("server: listen() failed");
58
59                                          /*--- accept() ---*/
60      addr_size = sizeof(struct sockaddr_in);
61      client_fd = accept(sock_fd,
62              (struct sockaddr *)&client_addr, &addr_size);
63      if (client_fd == -1)
64        err_exit("server: accept() failed");
65      printf("I'm connected from %s\n",
66             inet_ntoa(client_addr.sin_addr));
```

```
67
68     while (1)
69      {
70       FD_ZERO(&input_fdset);
71       FD_SET(STDIN_FILENO, &input_fdset);
72       FD_SET(client_fd, &input_fdset);
73       if (select(client_fd+1, &input_fdset, NULL, NULL,
74                  NULL) == -1)
75         err_exit("server: select() failed");
76       if (FD_ISSET(STDIN_FILENO, &input_fdset))
77        {
78         if (fgets(buffer, 256, stdin) == NULL)
79          {
80           printf("server: Closing socket.\n");
81           break;
82          }
83         length = strlen(buffer);
84         send(client_fd, buffer, length, 0);
85        }
86       else
87        {
88         length = recv(client_fd, buffer, 256, 0);
89         if (length == 0)
90          {
91           printf("Connection closed by remote host.\n");
92           break;
93          }
94         write(STDOUT_FILENO, buffer, length);
95        }
96      }
97     close(client_fd);
98     close(sock_fd);
99     return(0);
100   }
```

Nach Auswertung der Kommandozeilenparameter in Zeile 28 bis 38 wird in Zeile 40 zunächst ein Socket geöffnet. In den Zeilen 44 bis 46 wird dann die Adress-Struktur initialisiert. Durch die Angabe der Konstanten `INADDR_ANY` für die IP-Adresse wird der Socket beim folgenden `bind()`-Aufruf (Zeile 49) automatisch mit allen IP-Adressen des Computers[1] verknüpft. Ist dies nicht erwünscht, muss hier explizit die IP-Adresse angegeben werden, unter der der Dienst eingerichtet werden soll.

[1] Ist der Computer mit mehreren Netzwerk-Interfaces ausgestattet – beispielsweise eine Netzwerkkarte und ein WLAN-Interface –, haben diese in der Regel unterschiedliche IP-Adressen.

In den Zeilen 55 bis 64 folgen die Funktionsaufrufe `listen()` und `accept()`, mit denen die „Kontaktaufnahme“ durch einen Client vorbereitet wird. Nach einem `connect()` durch ein Client-Programm kehrt die Funktion `accept()` mit dem Dateideskriptor des neuen Sockets zurück, und die Struktur `client_addr` wurde mit der IP-Adresse und Port-Nummer des Client-Programms gefüllt. Die Funktion `inet_ntoa()` wandelt die binäre IP-Adresse in eine Zeichenkette um (Zeilen 65 und 66).

Wie bereits bei dem in Abschnitt 7.2.2 vorgestellten TCP/IP-Client ermöglicht auch hier die `while()`-Schleife in den Zeilen 68 bis 96 die bidirektionale Kommunikation über den Socket.

Das Belegen eines *reservierten* Ports[1] mit einem Dienst erfordert root-Rechte. Damit das Programm trotzdem von einem „normalen“ Benutzer ausgeführt werden kann, sind neben dem Kompilieren weitere Schritte erforderlich (vgl. auch Abschnitt 9.1.1):

```
> gcc server.c -o server
> su
Kennwort:
# chown root server
# chmod a+s server
# exit
```

Jetzt kann das Programm mit Angabe einer (freien) Port-Nummer gestartet werden. Zum Test soll hier ein HTTP-Server „vorgetäuscht“ werden – Voraussetzung ist, dass nicht bereits ein solcher Server (z. B. der Apache Webserver) läuft. Als Client-Programm dient ein Browser, z. B. Firefox:

```
> server 80
```

Jetzt „*firefox http://localhost/*“ in einem anderen Fenster aufrufen.

```
I'm connected from 127.0.0.1
GET / HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (X11; U; Linux i686 (x86_64); en-US;
  rv:1.7.10) Gecko/20050715 Firefox/1.0.6 SUSE/1.0.6-16
Accept: text/xml,application/xml,application/xhtml+xml,
text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: de-de,de;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

[1] Die Organisation IANA vergibt und registriert Portnummern für bestimmte Dienste wie z. B. Port 80 für HTTP. Unregistrierte Ports wie beispielsweise 30.000 können ohne besondere Benutzerrechte belegt werden.

An dieser Stelle können Sie das Programm mit Ctrl-D beenden oder eine HTTP-Antwort eingeben, was jedoch relativ mühsam ist und daher „echten“ Webservern überlassen werden sollte. Alternativ kann das Server-Programm natürlich auch mit dem interaktiven Client-Programm aus Abschnitt 7.2.2 kommunizieren. Auf diese Weise lässt sich eine „Chat“-Verbindung aufbauen, über die sich zwei Benutzer miteinander unterhalten können.

7.3.3 Ein kleiner Webserver

Der Nutzen eines interaktiven TCP/IP-Servers, wie er im vorigen Abschnitt dargestellt wurde, ist relativ eingeschränkt. Typische Server-Programme erlauben eine Kommunikation mit mehreren Client-Programmen gleichzeitig. Wie man dies realisiert, soll im Folgenden anhand eines kleinen HTTP-Servers demonstriert werden.

HTTP-Grundlagen

Eine typische HTTP-Anfrage eines Browsers (Firefox) war ja bereits im vorherigen Abschnitt zu sehen. Viele der Angaben, die der Browser seiner Anfrage mit auf den Weg gibt, werden wir nicht benötigen und auch nicht auswerten. Der grundsätzliche Aufbau einer HTTP-Anfrage und einer HTTP-Antwort sind in Abbildung 7.8 dargestellt.

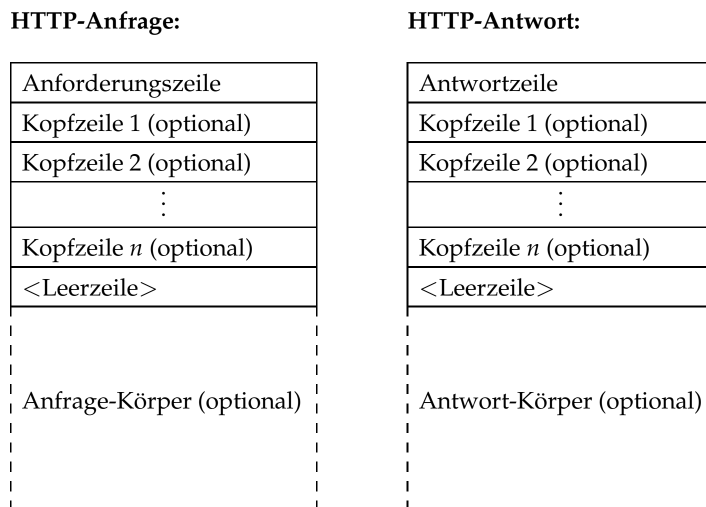


Abbildung 7.8: Prinzipieller Aufbau von HTTP-Anfragen und -Antworten

Eine HTTP-Anforderungszeile besteht dabei aus drei, durch Leerzeichen getrennte Elemente: Methode, Pfad, Protokoll. Beispiel:

```
GET /index.html HTTP/1.0
```

Mögliche *Methoden* sind:

GET – Daten (z. B. HTML-Datei) vom Server anfordern.

HEAD – Nur die Kopfzeilen vom Server anfordern. Damit kann beispielsweise die Länge der Daten und das Format erfragt werden, ohne die Daten selbst zu übertragen.

POST – Die Inhalte eines HTML-Formulars an den Server übertragen und die Antwort anfordern.

In der Regel enthält eine HTTP-Anfrage mindestens eine Kopfzeile der Form:

```
Host: www.hanser.de
```

Ab HTTP 1.1 ist diese Kopfzeile Pflicht. Sie ermöglicht es, mehrere WWW-Adressen auf *einem* Server (also mit gleicher IP-Adresse) zu verwalten, da der Server an der `Host`-Angabe erkennen kann, auf welche WWW-Adresse sich der in der Anforderungszeile angegebene Pfad bezieht.

Die Methoden `GET` und `HEAD` benötigen keinen Anfrage-Körper (*Body*), während er bei der Methode `POST` mit den Inhalten des HTML-Formulars gefüllt ist. In diesem Fall ist mindestens eine weitere Kopfzeile erforderlich, die die Länge des Anfrage-Körpers angibt.

Die Antwortzeile des Servers enthält – wie die Anforderungszeile – drei Elemente. Hier sind es: Protokoll, Status, Textmeldung. Beispiel:

```
HTTP/1.0 200 OK
```

oder

```
HTTP/1.0 404 Not Found
```

Die möglichen Statuswerte sind der detaillierten Protokollbeschreibung zu entnehmen, wie sie z. B. im Internet zu finden ist. Mit den beiden hier angegebenen Werten kennen Sie aber bereits die wichtigsten.

Ein Server, der etwas auf sich hält, sollte danach mindestens zwei Kopfzeilen ausgeben: die Art des Anwort-Körpers und dessen Länge, z. B.:

```
Content-type: text/html
Content-length: 1374
```

Nach einer Leerzeile (CR + LF!) folgt dann der eigentliche Körper, also beispielsweise die HTML-Datei, die JPEG-Grafik, ...

Hier das Ganze (HTTP-Anfrage und -Antwort) an einem Beispiel:

Browser:
```
GET /index.html HTTP/1.0
Host: www.kein-inhalt.de
```
Leerzeile

Server: `HTTP/1.0 200 OK`
`Content-type: text/html`
`Content-length: 38`
Leerzeile
`<html><body>Leere Seite.</body></html>`

Eine ausführlichere Beschreibung des HTTP-Standards finden Sie im Internet z. B. unter [14].

Das Programm

Die Beschreibung des HTTP war zwar sehr knapp gehalten und auf das Notwendigste beschränkt, diese Informationen reichen aber aus, um damit den folgenden kleinen HTTP-Server zu programmieren.

```
1    /*
2        webserver.c - minimalistischer HTTP-Server
3    */
4
5    # include <stdio.h>
6    # include <string.h>
7    # include <unistd.h>
8    # include <stdlib.h>
9    # include <sys/types.h>
10   # include <sys/socket.h>
11   # include <netinet/in.h>
12   # include <arpa/inet.h>
13   # include <sys/stat.h>
14   # include <signal.h>
15
16   # define MY_PORT 80
17   # define N_CONNECTIONS 20
18   # define HTML_PATH "."
19   # define DEFAULT_FILE "index.html"
20
21   void err_exit(char *message)
22    {
23     perror(message);
24     exit(1);
25    }
26
27   int get_line(int sock_fd, char *buffer, int length)
28    {
29     int i;
30
```

```
31     i = 0;
32     while ((i < length-1) &&
33            (recv(sock_fd, &(buffer[i]), 1, 0) == 1))
34       if (buffer[i] == '\n')
35         break;
36       else
37         i++;
38     if ((i > 0) && (buffer[i-1] == '\r'))
39       i--;
40     buffer[i] = '\0';
41     return(i);
42   }
43 
44 int is_html(char *filename)
45   {
46     if (strcmp(&(filename[strlen(filename)-5]), ".html")
47          == 0)
48       return(1);
49     if (strcmp(&(filename[strlen(filename)-4]), ".htm")
50         == 0)
51       return(1);
52     return(0);
53   }
54 
55 size_t file_size(char *filename)
56   {
57     struct stat file_info;
58 
59     if (stat(filename, &file_info) == -1)
60       return(0);
61     return(file_info.st_size);
62   }
63 
64 void http_service(int client_fd)
65   {
66     char buffer[256], cmd[8], url[128], *filename;
67     int length;
68     FILE *stream;
69 
70     if (get_line(client_fd, buffer, 256) == 0)
71       return;
72     if (sscanf(buffer, "%7s %127s", cmd, url) < 2)
73       return;
74     while (get_line(client_fd, buffer, 256) > 0);
```

```
75      if ((strcmp(cmd, "GET") != 0)
76          && (strcmp(cmd, "HEAD") != 0))
77        return;
78
79      filename = &(url[1]);
80      if (strlen(filename) == 0)
81        filename = DEFAULT_FILE;
82
83      if ((stream = fopen(filename, "r")) == NULL)
84       {
85        send(client_fd, "HTTP/1.0 404 Not Found\r\n"
86             "Content-type: text/html\r\n"
87             "Content-length: 91\r\n\r\n"
88             "<html><head><title>Error</title></head>"
89             "<body><hr><h2>File not found.</h2><hr>"
90             "</body></html>", 162, 0);
91        return;
92       }
93
94      send(client_fd, "HTTP/1.0 200 OK\r\n", 17, 0);
95      if (is_html(filename))
96        send(client_fd, "Content-type: text/html\r\n", 25, 0);
97      sprintf(buffer, "Content-length: %ld\r\n\r\n",
98              file_size(filename));
99      send(client_fd, buffer, strlen(buffer), 0);
100     if (strcmp(cmd, "GET") == 0)
101       while (!feof(stream))
102        {
103         length = fread(buffer, 1, 256, stream);
104         if (length > 0)
105           send(client_fd, buffer, length, 0);
106        }
107     fclose(stream);
108     return;
109    }
110
111   /*--------------- Hauptprogramm ---------------*/
112
113   int main()
114    {
115     int sock_fd, client_fd, err, pid;
116     struct sockaddr_in my_addr, client_addr;
117     socklen_t addr_size;
118
```

```
119    sock_fd = socket(PF_INET, SOCK_STREAM, 0);
120    if (sock_fd == -1)
121      err_exit("webserver: Can't create new socket");
122
123    my_addr.sin_family = AF_INET;
124    my_addr.sin_port = htons(MY_PORT);
125    my_addr.sin_addr.s_addr = INADDR_ANY;
126
127    err = bind(sock_fd, (struct sockaddr *)&my_addr,
128               sizeof(struct sockaddr_in));
129    if (err == -1)
130      err_exit("webserver: bind() failed");
131
132    setuid(getuid());
133
134    err = listen(sock_fd, N_CONNECTIONS);
135    if (err == -1)
136      err_exit("webserver: listen() failed");
137
138    if (chdir(HTML_PATH) != 0)
139      err_exit("webserver: Can't set HTML path");
140
141    signal(SIGCHLD, SIG_IGN);
142
143    printf("Type Ctrl-C to stop.\n");
144
145    while (1)
146     {
147      addr_size = sizeof(struct sockaddr_in);
148      client_fd = accept(sock_fd,
149              (struct sockaddr *)&client_addr, &addr_size);
150      if (client_fd == -1)
151        err_exit("webserver: accept() failed");
152
153      if ((pid = fork()) == -1)
154       {
155        fprintf(stderr, "webserver: fork() failed.\n");
156        return(1);
157       }
158      else if (pid == 0)          /* Kind-Prozess */
159       {
160        close(sock_fd);
161        http_service(client_fd);
162        shutdown(client_fd, SHUT_RDWR);
```

```
163         close(client_fd);
164         return(0);
165        }
166      close(client_fd);
167     }
168
169    return(0);              /* wird nie erreicht */
170   }
```

Um den Webserver als „normaler“ Benutzer starten zu können, muss auch hier wiederum als Besitzer des Programms „root“ eingestellt und das „s“-Bit gesetzt werden (vgl. Abschnitt 9.1.1). Danach kann man das Programm ohne Kommandozeilenparameter aufrufen. Der Server läuft nun so lange, bis der Prozess durch Eingabe von Ctrl-C oder durch einen `kill`-Befehl beendet wird. Um dem Webserver eine HTML-Seite zu entlocken, sollten Sie entweder eine HTML-Datei in das aktuelle Verzeichnis kopieren oder ein Verzeichnis, das HTML-Dateien enthält, in Zeile 18 als „`HTML_PATH`“ eintragen. Als URL geben Sie dann im Browser „`http://localhost/`*HTML-Datei*“ ein.

Betrachten wir zunächst das Hauptprogramm ab Zeile 113. Zu Beginn wird gemäß dem bereits vorgestellten Schema zunächst ein Socket geöffnet und anschließend mit `bind()` mit dem Port 80 (Konstante `MY_PORT`) verknüpft. In Zeile 132 erfolgt dann der Aufruf `setuid(getuid())`. Er bewirkt, dass das Programm nach der Verknüpfung mit dem Port die über das „s“-Bit verliehenen root-Rechte wieder abgibt. Dadurch könnte das Programm auch bei einer Fehlfunktion keinen ernsthaften Schaden mehr anrichten (siehe auch Abschnitt 7.5). Erst danach erfolgt der `listen()`-Aufruf, um den Port in Verbindungsbereitschaft zu bringen.

Mit Hilfe der Funktion `signal()` wird in Zeile 141 eingestellt, dass Kind-Prozesse nach Beendigung sofort aus dem Speicher und aus der Prozessliste entfernt werden, statt abzuwarten, bis der Eltern-Prozess den Exit-Status der Kind-Prozesse abfragt. Da bei unserem Webserver-Programm jede HTTP-Anfrage einen Kind-Prozess startet, würde andernfalls eine Unmenge so genannter *Zombie*-Prozesse[1] erzeugt.

In der `while()`-Schleife (Zeilen 145 bis 167) werden Verbindungsanforderungen von Client-Programmen (Browsern) mit `accept()` angenommen und jeweils in einem eigenen Kind-Prozess mit dem Unterprogramm `http_service()` abgearbeitet. Danach wird die Verbindung zum Client mit Hilfe der Funktion `shutdown()` beendet (Zeile 162) und anschließend der Socket geschlossen (Zeile 163).

Die Funktion `http_service()` ist in den Zeilen 64 bis 109 definiert. Sie liest zunächst die HTTP-Anforderung ein und prüft auf die Kommandos „GET“ und

[1] Das sind Prozesse, die eigentlich beendet sind, aber noch darauf warten, ihren Status zurückzumelden.

„HEAD“ (andere werden zur Zeit nicht unterstützt). Schlägt das Öffnen der angeforderten Datei in Zeile 83 fehl, so wird eine entsprechende HTTP-Fehlermeldung an den Client zurückgeschickt (Zeile 85 bis 91). Wenn die angegebene Datei existiert, wird eine positive Rückmeldung generiert (Zeile 94) und geprüft, ob es sich um eine HTML-Datei handelt (Zeile 95). Dies wird dem Client dann durch die Information „`Content-type: text/html`“ angezeigt. In jedem Fall wird die Länge der Datei mit Hilfe der Kopfzeile „`Content-length:`“ übermittelt. Handelte es sich bei der Anforderung um ein „HEAD“, ist die Kommunikation an dieser Stelle beendet. Bei einer „GET“-Anforderung folgt die Übertragung der eigentlichen Datei (Zeile 101 bis 106).

7.4 Das *User Datagram Protocol* (UDP)

In den Abschnitten 7.2 bis 7.3.3 haben wir *verbindungsorientierte* Sockets auf Basis des Protokolls TCP verwendet. Der Vorteil dabei ist, dass eine *gesicherte* Verbindung zwischen zwei Netzwerk-Ports hergestellt wird, bei der keine Daten „verloren gehen“ oder in falscher Reihenfolge ankommen können. Im Gegensatz dazu besteht beim Versenden von UDP-Paketen keine Garantie, dass diese wirklich ankommen. Dennoch bietet UDP-basierte Netzwerkkommunikation zwei Eigenschaften, die sehr nützlich oder sogar notwendig sein können:

- Der „Datenstrom“ wird nicht blockiert, falls ein Paket nicht oder nur fehlerhaft angekommen ist. Dies ist für zeitkritische Anwendungen wie Internettelefonie wichtig.
- UDP erlaubt es, Nachrichten an *mehrere* (Multicast) oder an *alle* (Broadcast) Netzwerkteilnehmer zu schicken. Das ist mit TCP nicht möglich!

Aus diesem Grund zeigen wir in den folgenden Abschnitten, wie die Kommunikation über UDP funktioniert und wie Broadcast- und Multicast-Nachrichten versendet werden.

7.4.1 UDP-Nachrichten senden

Bereits beim Öffnen des Sockets muss festgelegt werden, ob das Protokoll TCP oder UDP verwendet werden soll (vgl. Abschnitt 7.1.4). Für UDP muss man als Typ „`SOCK_DGRAM`“ angeben:

```
int sock_fd = socket(PF_INET, SOCK_DGRAM, 0);
```

Bei dem Protokoll UDP werden einzelne Pakete versendet, ohne dass zuvor eine „Verbindung“ zwischen Client und Server aufgebaut wird. Daher entfällt bei der UDP-Kommunikation der Funktionsaufruf `connect()`. Um das Ziel (IP-Adresse und Port) beim Versenden eines Pakets angeben zu können, müssen Sie bei UDP die Funktion

```
ssize_t sendto(int sockfd, void *buff, size_t n, int flags,
               struct sockaddr *addr, socklen_t addr_len);
```

verwenden, die im Gegensatz zu `send()` den Parameter `addr` für die Zieladresse enthält. Analog dazu gibt es auch eine Funktion zum Empfangen von UDP-Paketen:

```
ssize_t recvfrom(int sockfd, void *buff, size_t n, int flags,
                 struct sockaddr *addr, socklen_t addr_len);
```

Hier dient der Parameter `addr` als Zeiger auf einen „Platzhalter“, der mit den Adressinformationen des Absenders der Nachricht gefüllt wird. Das folgende Beispielprogramm „udp-client“ verwendet beide Funktionen, um eine Nachricht an einen UDP-Server zu schicken und eine Antwort vom Server zu empfangen.

```
1   /*
2       udp-client.c - Text an UDP-Server schicken
3   */
4
5   # include <stdio.h>
6   # include <unistd.h>
7   # include <string.h>
8   # include <sys/poll.h>
9   # include <sys/types.h>
10  # include <sys/socket.h>
11  # include <sys/time.h>
12  # include <netinet/in.h>
13  # include <arpa/inet.h>
14
15  # define BUF_SIZE 1000
16  # define TIMEOUT 1000   /* Millisekunden */
17
18  int main(int argc, char *argv[])
19   {
20    int sock_fd, port, length, err;
21    struct sockaddr_in server_addr, from_addr;
22    socklen_t addr_size;
23    struct pollfd pollfd;
24    static char buffer[BUF_SIZE];
25
26    if (argc != 4)
27     {
28      fprintf(stderr,
29              "Usage: udp-client ip-addr port message\n");
30      return(1);
```

```
31     }
32
33    if (sscanf(argv[2], "%d", &port) != 1)
34     {
35      fprintf(stderr, "udp-client: bad port number '%s'\n",
36              argv[2]);
37      return(1);
38     }
39                                  /*--- Socket öffnen ---*/
40    sock_fd = socket(PF_INET, SOCK_DGRAM, 0);
41    if (sock_fd == -1)
42     {
43      perror("udp-client: Can't create new socket");
44      return(1);
45     }
46                                  /*--- Zieladresse ---*/
47    server_addr.sin_family = AF_INET;
48    server_addr.sin_port = htons(port);
49    err = inet_aton(argv[1], &(server_addr.sin_addr));
50    if (err == 0)
51     {
52      fprintf(stderr, "udp-client: Bad IP-Address '%s'\n",
53              argv[1]);
54      return(1);
55     }
56                              /*--- Nachricht senden ---*/
57    length = sendto(sock_fd, argv[3], strlen(argv[3]), 0,
58                    (struct sockaddr *)&server_addr,
59                    sizeof(struct sockaddr));
60    if (length != strlen(argv[3]))
61      perror("udp-client: sendto() failed");
62
63                              /*--- auf Antwort warten ---*/
64    pollfd.fd = sock_fd;          /* Polling vorbereiten */
65    pollfd.events = POLLIN | POLLPRI;
66    err = poll(&pollfd, 1, TIMEOUT);
67    if (err < 0)
68      perror("udp-client: poll() failed");
69    else if (err == 0)
70      printf("<No answer received.>\n");
71    else
72     {
73      addr_size = sizeof(struct sockaddr_in);
74      length = recvfrom(sock_fd, buffer, BUF_SIZE-1, 0,
```

```
75                        (struct sockaddr *)&from_addr,
76                        &addr_size);
77        if (length == -1)
78          perror("udp-client: recvfrom() failed");
79        else
80         {
81          buffer[length] = '\0';
82          printf("Response from %s: %s\n",
83                 inet_ntoa(from_addr.sin_addr), buffer);
84         }
85      }
86      close(sock_fd);
87      return(0);
88    }
```

Nach dem Öffnen des Sockets (Zeile 40) trägt das Programm die als Kommandozeilenparameter angegebene IP-Adresse und Portnummer in die Struktur `server_addr` ein (Zeile 47 bis 49). An diese Adresse wird dann der als dritter Kommandozeilenparameter übergebene Text gesendet (Zeile 57 bis 59). Anschließend wartet das Programm maximal 1 Sekunde auf eine Antwort vom Server. Dies geschieht mit Hilfe der Funktion `poll()` in den Zeilen 64 bis 66. `poll()` arbeitet ähnlich wie die Funktion `select()`, die unter anderem in unseren TCP-Beispielen verwendet und auf Seite 110 beschrieben wurde.

Um das Programm zu testen, können Sie den Netzwerkdienst „Echo“ (UDP-Port 7) verwenden. Dieser Dienst ist im Dämon „xinetd“ enthalten und lässt sich, wie in Abbildung 7.9 gezeigt, aktivieren (siehe auch Seite 168).

Der „Echo“-Dienst sendet alle empfangenen Pakete an den jeweiligen Absender zurück. In Verbindung mit unserem Beispielprogramm sieht das dann so aus:

```
> gcc udp-client.c -o udp-client
> ./udp-client 127.0.0.1 7 "Dies ist ein Test."
Response from 127.0.0.1: Dies ist ein Test.
```

7.4.2 Der UDP-Server

Im Vergleich zu dem in Abbildung 7.7 dargestellten Ablauf von Funktionsaufrufen eines TCP-Servers entfallen beim UDP-Server die Funktionen `listen()` und `accept()`, die man nur für verbindungsorientierte Kommunikation benötigt. Außerdem werden zum Senden und Empfangen von Paketen die oben beschriebenen Funktionen `sendto()` und `recvfrom()` verwendet. Das folgende Programm realisiert einen UDP-Server, der alle empfangenen Pakete als Text ausgibt und einen Bestätigungstext an den Absender schickt. Empfängt der Server ein UDP-Paket mit dem Inhalt „quit“, wird das Programm beendet.

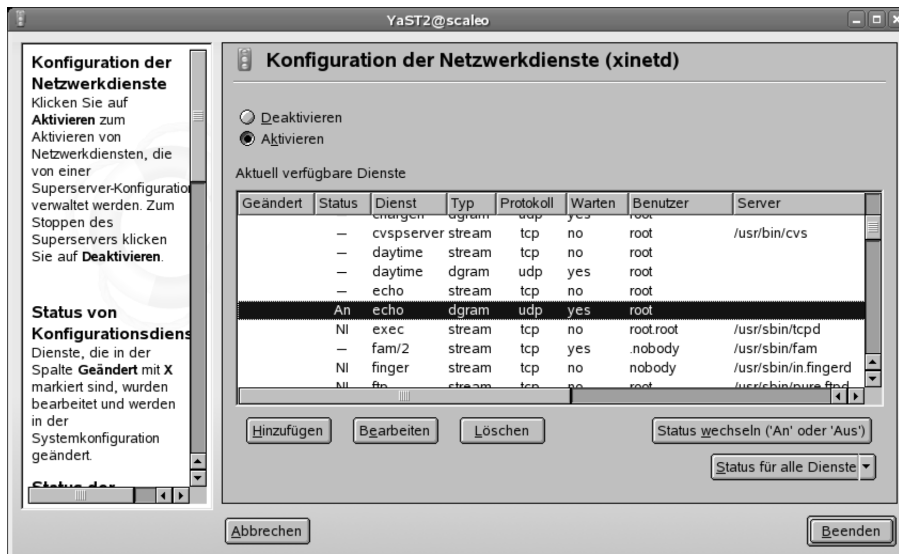


Abbildung 7.9: Aktivierung des „Echo"-Dienstes (UDP)

```
/*
    udp-server.c - Server für UDP-Datagramme
*/

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

# define BUF_SIZE 1000

int main(int argc, char *argv[])
 {
  int sock_fd, client_fd, port, err, length, stop;
  struct sockaddr_in my_addr, client_addr;
  socklen_t addr_size;
  static char buffer[BUF_SIZE];

  if ((argc != 2) || (strcmp(argv[1], "-h") == 0))
   {
```

```
24        fprintf(stderr, "Usage: udp-server port\n");
25        return(1);
26       }
27
28     if (sscanf(argv[1], "%d", &port) != 1)
29       {
30        fprintf(stderr, "udp-server: Bad port number '%s'.\n",
31                argv[1]);
32        return(1);
33       }
34                                 /*--- Socket öffnen ---*/
35     sock_fd = socket(PF_INET, SOCK_DGRAM, 0);
36     if (sock_fd == -1)
37       {
38        perror("udp-server: Can't create new socket");
39        return(1);
40       }
41                         /*--- Socket an Port binden ---*/
42     my_addr.sin_family = AF_INET;
43     my_addr.sin_port = htons(port);
44     my_addr.sin_addr.s_addr = INADDR_ANY;
45     err = bind(sock_fd, (struct sockaddr *)&my_addr,
46                 sizeof(struct sockaddr_in));
47     if (err == -1)
48       {
49        perror("udp-server: bind() failed");
50        return(1);
51       }
52
53     stop = 0;
54     while (!stop)       /*--- so lange, bis 'quit' empf. */
55       {                           /*--- Paket empfangen ---*/
56        addr_size = sizeof(struct sockaddr_in);
57        length = recvfrom(sock_fd, buffer, BUF_SIZE-1, 0,
58                          (struct sockaddr *)&client_addr,
59                          &addr_size);
60        if (length == -1)
61          perror("udp-server: recvfrom() failed");
62        else
63         {                          /*--- Paket ausgeben ---*/
64          buffer[length] = '\0';
65          printf("Datagram from %s:\n%s\n",
66                 inet_ntoa(client_addr.sin_addr), buffer);
67          if (strcmp(buffer, "quit") == 0)
```

```
68          {
69            strcpy(buffer, "Server stopped.");
70            stop = 1;
71          }
72        else
73            strcpy(buffer, "Message received.");
74                                 /*--- Antwort senden ---*/
75        length = sendto(sock_fd, buffer, strlen(buffer), 0,
76                        (struct sockaddr *)&client_addr,
77                        sizeof(struct sockaddr));
78        if (length < strlen(buffer))
79          perror("udp-server: sendto() failed");
80      }
81    }
82    close(client_fd);
83    close(sock_fd);
84    return(0);
85  }
```

Das Programm erwartet als Kommandozeilenparameter die Nummer des Ports, an dem der Server Pakete empfangen soll. Wenn Sie hier eine unregistrierte Portnummer (z. B. 30000) angeben, sollte das Programm auch ohne root-Rechte laufen:

```
> gcc udp-server.c -o udp-server
> ./udp-server 30000
```

Wenn Sie jetzt in einem anderen Terminal-Fenster das UDP-Client-Programm auf den Port 30000 anwenden, erhalten Sie die entsprechenden Empfangsbestätigungen:

```
> ./udp-client 127.0.0.1 30000 Test
Response from 127.0.0.1: Message received.
> ./udp-client 127.0.0.1 30000 quit
Response from 127.0.0.1: Server stopped.
```

7.4.3 Pakete an alle Teilnehmer senden: Broadcast

Wie bereits oben beschrieben, ist es mit UDP möglich, eine Nachricht an *alle* Teilnehmer eines lokalen Netzwerks zu schicken.[1] Dieses Verfahren wird beispielsweise für die Suche von Diensten im Netzwerk verwendet, wenn die IP-Adresse des Servers noch nicht bekannt ist.

Das Senden und Empfangen von Broadcast-Paketen geschieht mit den gleichen Funktionen wie das Übertragen von „normalen" Datagrammen, jedoch mit einer

[1] Damit solche Nachrichten nicht über das Internet an jeden Computer gehen, der gerade online ist, werden Broadcast-Pakete von Netzwerkroutern nicht weitergeleitet und bleiben daher auf das *Local Area Network* (LAN) beschränkt.

speziellen Zieladresse. Jeder Netzwerkschnittstelle, die eine gültige IP-Adresse besitzt, ist automatisch eine entsprechende Broadcast-Adresse zugeordnet. Die Schnittstelle empfängt UDP-Pakete, die an diese Adresse geschickt werden, so als ob sie an die eigentliche IP-Adresse geschickt wurden.

Eine Broadcast-Adresse ist dadurch gekennzeichnet, dass alle Bits des *Host*-Teils der IP-Adresse – also der Teil, der bei allen Teilnehmern *eines* Netzwerks unterschiedlich ist – auf 1 gesetzt sind. Beispiel: Haben die Teilnehmer eines lokalen Netzwerks Adressen der Form 192.168.0.xxx, so lautet die zugehörige Broadcast-Adresse 192.168.0.255. Mit dem Programm „ifconfig“ können Sie die IP-Adresse und die zugehörige Broadcast-Adresse aller Netzwerkschnittstellen anzeigen lassen:

```
> /sbin/ifconfig
eth0  Protokoll:Ethernet  Hardware Adresse 00:13:D4:85:4C:89
      inet Adresse:192.168.0.1 Bcast:192.168.0.255
      Maske:255.255.255.0
      inet6 Adresse: fe80::213:d4ff:fe85:4c89/64
      Gültigkeitsbereich:Verbindung
      UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
      RX packets:0 errors:0 dropped:0 overruns:0 frame:0
      TX packets:19 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 Sendewarteschlangenlänge:1000
      RX bytes:0 (0.0 b) TX bytes:1575 (1.5 Kb)
      Interrupt:225 Basisadresse:0x2000
```

Darüber hinaus gibt es eine besondere Broadcast-Adresse, die von der IP-Adresse der Schnittstelle unabhängig ist: 255.255.255.255.

Auch wenn das Senden von Broadcast-Paketen mit den bereits beschriebenen Funktionen für „normale“ Pakete geschieht, ist bei Linux eine zusätzliche Sicherung gegen das unbeabsichtigte Verschicken eines Broadcasts eingebaut: Zuerst muss der Socket für Broadcast-Adressen „freigeschaltet“ werden. Dies geschieht mit der Funktion `setsockopt()` und der Option `SO_BROADCAST`.

Um das Programm aus Abschnitt 7.4.1 in die Lage zu versetzen, Broadcast-Pakete zu verschicken, müssen Sie ab Zeile 47 folgenden Quelltext einfügen:

```
int i = 1;
if (setsockopt(sock_fd, SOL_SOCKET, SO_BROADCAST, &i,
               sizeof(i)) < 0)
  perror("udp-client: Can't set BROADCAST option.");
```

Wenn Ihr Rechner in einem lokalen Netzwerk z. B. die Adresse 192.168.0.1 hat, können Sie mit dem folgenden Aufruf den Echo-Dienst aller Teilnehmer des lokalen Netzwerks adressieren:

```
> udp-client 192.168.0.255 7 Hallo
Response from 192.168.0.1: Hallo
```

Sollen die Antworten *aller* Rechner im Netzwerk angezeigt werden, müssen der `poll()`- und der `recvfrom()`-Aufruf so lange wiederholt werden, bis keine Antwort mehr eintrifft, so dass die `poll()`-Funktion einen TIMEOUT liefert. Auf diese Weise erfahren Sie dann automatisch die IP-Adressen aller Rechner im lokalen Netzwerk, bei denen der Echo-Dienst aktiviert ist!

7.4.4 Multicast-Sockets

Bei den bisher vorgestellten Adressierungsarten des Internet Protokolls (IP) gab es entweder die Möglichkeit, eine Nachricht gezielt an einen Teilnehmer zu senden (*Unicast*), oder ein Paket an *alle* Netzwerkteilnehmer des lokalen Netzwerks zu schicken (Broadcast). Der aktuelle IP-Standard sieht eine weitere Möglichkeit der Adressierung vor, bei dem eine Nachricht an mehrere, aber nicht alle Teilnehmer gesendet wird: Multicast.

Multicast-Adressen

Multicast-Adressen liegen – anders als die Broadcast-Adressen – in einem speziellen IP-Adressbereich, der unabhängig von der Adresse der Netzwerkschnittstelle ist: 224.0.0.0 bis 239.255.255.255. Dies bedeutet, dass jeder Empfänger von Multicast-Paketen so eingerichtet sein muss, dass er neben seiner eigentlichen IP-Adresse auch Pakete annimmt, die an eine oder mehrere Multicast-Adressen gerichtet sind!

Wie Sockets für den Empfang von Multicast-Paketen eingerichtet werden, zeigt der nächste Abschnitt. Aber zunächst zum Senden von Multicast-Nachrichten: Für ausgehende Pakete muss in der Routing-Tabelle eingetragen sein, über welche Schnittstelle Multicast-Pakete gesendet werden, z. B. eth0:

```
> su
Kennwort:
# route add -net 224.0.0.0 netmask 240.0.0.0 dev eth0
```

Weitere Vorbereitungen sind für das Senden von Nachrichten an Multicast-Adressen nicht notwendig. Mit dem oben beschriebenen Programm „udp-client“ lassen sich nun Multicast-Pakete an das lokale Netzwerk verschicken.

Multicast-Sockets einrichten

Es sei zuerst erwähnt, dass der Linux-Kernel Multicast unterstützen muss, um einen Multicast-Socket einrichten zu können. Dies lässt sich u. a. wie folgt überprüfen:

```
> cat /proc/config.gz | gunzip | grep MULTICAST
CONFIG_IP_MULTICAST=y
```

Um ein UDP-Server-Programm Multicast-fähig zu machen, muss die gewünschte Multicast-Adresse mit Hilfe der Funktion `setsockopt()` mit dem Socket verknüpft werden, oder, anders ausgedrückt: Der Socket muss der Multicast-Gruppe hinzugefügt werden:

```
# define MCAST_ADDR 224.0.0.1

struct ip_mreq mreq;

mreq.imr_multiaddr.s_addr = inet_addr(MCAST_ADDR);
mreq.imr_interface.s_addr = htonl(INADDR_ANY);
if (setsockopt(sock_fd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
    &mreq, sizeof(mreq)) < 0)
  {
   perror("setsockopt() failed");
   return(1);
  }
```

7.4.5 UPnP – *Universal Plug And Play*

Es gibt ein internationales Gremium, das daran arbeitet, Netzwerke „Hot-Plug-And-Play"-fähig zu machen. So wie beim Anschließen einer USB-Maus an einen freien USB-Port automatisch die entsprechenden Treiber geladen werden und die Maus als Eingabegerät eingerichtet wird, soll das Gleiche mit Diensten in einem Netzwerk möglich sein. Der zu diesem Zweck entwickelte Standard heißt *Universal Plug And Play* (kurz UPnP) und basiert auf UDP-Multicast, HTTP und XML. Da auch Microsoft™ in diesem Gremium vertreten ist, bieten Windows™-Rechner bereits einige Dienste über UPnP an. Auch IP-Kameras verschiedener Hersteller (z. B. Axis oder Pelco) haben UPnP implementiert, um die Kameras und deren Funktionsumfang automatisch über das Netzwerk ermitteln zu können.

Um nach UPnP-Diensten im lokalen Netzwerk zu suchen, können Sie die folgende Sequenz an die Multicast-Adresse 239.255.255.250 und Port 1900 schicken:

```
M-SEARCH * HTTP/1.1
HOST: 239.255.255.250:1900
MAN: "ssdp:discover"
MX: 2
ST: ssdp:all
<Leerzeile>
```

Sie sehen, dass es sich dabei um eine HTTP-Anfrage handelt, doch ist als *Methode* nicht GET, sondern M-SEARCH angegeben. Die Kopfzeile „MX:" gibt die

Anzahl der Sekunden an, die der UPnP-Server maximal warten soll, bis er eine Antwort sendet. Damit soll verhindert werden, dass viele Server gleichzeitig antworten. Mit der Kopfzeile „ST:“ wird angegeben, nach welchem Dienst gesucht wird (*Search Target*). Der Wert „`ssdp:all`“ fordert eine Rückmeldung *aller* Dienste an, während „`upnp:rootdevice`“ nur *eine* Antwort pro UPnP-Host liefert.

Die im lokalen Netzwerk verfügbaren UPnP-Dienste reagieren auf diese Anforderung mit einer HTTP-Antwort entsprechend Abbildung 7.8, wobei die Antworten in der Regel nur aus den Kopfzeilen ohne *Body* bestehen. Die wichtigste Kopfzeile beginnt mit dem Schlüsselwort „Location:“, gefolgt von einem URL (*Uniform Resource Locator*) der Form „http://*IP-Adresse*:*Port*/*Pfad*“. Über diesen URL können Sie eine XML-Datei mit einer detaillierten Beschreibung des Dienstes abrufen. Das folgende Programm fragt auf diese Weise die aktiven UPnP-Dienste ab und gibt die Antworten aus. Es benötigt keine Kommandozeilenparameter.

```
1   /*
2       upnp-search.c - UPnP-Dienste abfragen
3   */
4
5   # include <stdio.h>
6   # include <unistd.h>
7   # include <string.h>
8   # include <sys/poll.h>
9   # include <sys/types.h>
10  # include <sys/socket.h>
11  # include <sys/time.h>
12  # include <netinet/in.h>
13  # include <arpa/inet.h>
14
15  # define BUF_SIZE 20000
16  # define TIMEOUT 3000   /* Millisekunden */
17
18  # define UPNP_ADDR "239.255.255.250"
19  # define UPNP_PORT 1900
20
21  int main()
22   {
23    int sock_fd, length, i, err;
24    struct sockaddr_in server_addr, from_addr;
25    socklen_t addr_size;
26    struct pollfd pollfd;
27    static char buffer[BUF_SIZE];
28
29                                   /*--- Socket öffnen ---*/
30    sock_fd = socket(PF_INET, SOCK_DGRAM, 0);
```

```
31    if (sock_fd == -1)
32     {
33      perror("upnp-search: Can't create new socket");
34      return(1);
35     }
36                                       /*--- Zieladresse ---*/
37    server_addr.sin_family = AF_INET;
38    server_addr.sin_port = htons(UPNP_PORT);
39    inet_aton(UPNP_ADDR, &(server_addr.sin_addr));
40
41                                    /*--- Anfrage senden ---*/
42    strcpy(buffer, "M-SEARCH * HTTP/1.1\r\n"
43                   "HOST: 239.255.255.250:1900\r\n"
44                   "MAN: \"ssdp:discover\"\r\n"
45                   "MX: 2\r\n"
46                   "ST: ssdp:all\r\n"
47                   "\r\n");
48
49    length = sendto(sock_fd, buffer, strlen(buffer), 0,
50                    (struct sockaddr *)&server_addr,
51                    sizeof(struct sockaddr));
52    if (length != strlen(buffer))
53     {
54      perror("upnp-search: sendto() failed");
55      return(1);
56     }
57                                /*--- auf Antwort warten ---*/
58    pollfd.fd = sock_fd;          /* Polling vorbereiten */
59    pollfd.events = POLLIN | POLLPRI;
60    i = 0;
61    while (1)
62     {
63      err = poll(&pollfd, 1, TIMEOUT);
64      if (err < 0)
65       {
66        perror("upnp-search: poll() failed");
67        break;
68       }
69      else if (err == 0)          /* Timeout erreicht */
70       {
71        if (i == 0)
72          printf("<No response received.>\n");
73        break;
74       }
```

```
75      else
76       {
77        addr_size = sizeof(struct sockaddr_in);
78        length = recvfrom(sock_fd, buffer, BUF_SIZE-1, 0,
79                          (struct sockaddr *)&from_addr,
80                          &addr_size);
81        if (length == -1)
82          perror("upnp-search: recvfrom() failed");
83        else
84         {
85          buffer[length] = '\0';
86          printf("\33[1m---- Response from %s:\33[0m\n%s\n",
87                 inet_ntoa(from_addr.sin_addr), buffer);
88         }
89        i = 1;
90       }
91     }
92    close(sock_fd);
93    return(0);
94   }
```

Wenn sich z. B. ein Windows™-Rechner als Internet-Gateway im lokalen Netzwerk befindet, werden Sie mehrere UPnP-Antworten ähnlich dem folgenden Beispiel erhalten:

```
> ./upnp-search
---- Response from 192.168.0.1:
HTTP/1.1 200 OK
ST:urn:schemas-upnp-org:device:InternetGatewayDevice:1
USN:uuid:e21500e7-3d2f-45a2-849c-d64707cb66b3::urn:schemas↩
  -upnp-org:device:InternetGatewayDevice:1
Location:http://192.168.0.1:2869/upnphost/udhisapi.dll?con↩
  tent=uuid:e21500e7-3d2f-45a2-849c-d64707cb66b3
Cache-Control:max-age=1800
Server:Microsoft-Windows-NT/5.1 UPnP/1.0
UPnP-Device-Host/1.0
Ext:
```

Aus der mit „ST:“ beginnenden Zeile können Sie entnehmen, dass es sich um ein „InternetGatewayDevice“ handelt. Den hinter „Location:“ angegebenen URL können Sie beispielsweise mit Firefox öffnen, um die XML-Beschreibungsdatei des Dienstes darzustellen.

Für weitere Informationen zu UPnP möchten wir auf die entsprechenden Spezifikationen [15] und [16] verweisen.

7.5 Noch ein Wort zur Sicherheit

Die beschriebenen Techniken und Programmbeispiele versetzen Sie in die Lage, über das Internet mit der „großen, weiten Welt“ zu kommunizieren. Unglücklicherweise sind es – neben anderen – genau diese Techniken, die Hackern, Viren und Würmern dabei helfen, ihre weniger harmlosen Absichten zu verfolgen. Sobald Sie einen Dienst über einen TCP/IP-Port einrichten, könnten ungebetene „Gäste“ versuchen, diesen als Hintertür zu Ihrem System zu missbrauchen. Dies gilt vor allem dann, wenn Sie ohne *Firewall* eine Verbindung zum Internet herstellen.

Am sichersten ist es daher, wenn Sie Client-Server-Programme zunächst auf Systemen entwickeln, die keine Verbindung „nach draußen“ haben. Außerdem sollten Sie beim Erstellen solcher Programme immer berücksichtigen, dass Ihr „Gegenüber“ möglicherweise andere Anfragen oder Antworten schickt als die von Ihrem Programm erwarteten. Ein beliebter Fehler ist beispielsweise ein nicht abgefangener Pufferüberlauf: Sendet der entfernte Rechner mehr Zeichen, als in den Empfangspuffer passen, können dadurch andere Bereiche des Programms mit Daten des fremden Rechners überschrieben werden.

Wenn Sie Programme zur Netzwerkkommunikation auf einem eigenen kleinen Rechnernetzwerk testen wollen, sollten Sie – falls nicht ohnehin schon geschehen – IP-Adressen verwenden, die speziell für private Netzwerke reserviert sind und in der Regel von einem *Router* nicht weitergeleitet werden. Dies sind die Adressen 192.168.0.1 bis 192.168.0.254. Weitere Informationen zu den verschiedenen IP-Adressbereichen finden Sie z. B. unter [17].