Kapitel 4

Einführung in Netzwerkdienste und TCP/IP

4.1 Netzwerkdienste

Ein Netzwerkdienst ist ein Prozess, der unter einer Netzwerkadresse einen Dienst anbietet. Ein Klient, der die Netzwerkadresse kennt, kann einen bidirektionalen Kommunikationskanal zu dem Netzwerkdienst eröffnen und über diesen mit dem Dienst kommunizieren, beispielsweise indem Kommandos auf dem Hinweg übermittelt werden und der Dienst auf dem Rückweg des Kommunikationskanals die Antwort überträgt.

Im Unterschied zu Pipelines müssen die beiden Kommunikationspartner nicht miteinander verwandt sein. Sie müssen nicht einmal auf dem gleichen Rechner laufen. Da der Kommunikationskanal bidirektional ist, wird ein echter Dialog zwischen den beiden Prozessen möglich. Der Aufbau einer Verbindung ist jedoch schwieriger, da die Netzwerkadresse des gewünschten Partners ermittelt werden muss.

Wenn Dienste über das Netzwerk angeboten und in Anspruch genommen werden, ergeben sich viele Vorteile:

- Der Dienst kann allen offenstehen, und ein direkter Zugang zu dem Rechner, auf dem der Dienst angeboten wird, ist nicht notwendig.
- Viele Parteien können in kooperativer Weise einen Dienst gleichzeitig nutzen.
- Der Dienste-Anbieter hat weniger Last, da die Benutzerschnittstelle auf anderen Rechnern laufen kann.

Andererseits bergen Netzwerkdienste eine Reihe von Risiken und Problematiken:

- Der Kreis derjenigen, die auf einen Netzwerkdienst zugreifen können, ist möglicherweise ziemlich umfangreich (normalerweise das gesamte Internet).
- Somit muss jeder Netzwerkdienst Zugriffsberechtigungen einführen und überprüfen und kann sich dabei nicht wie traditionelle Applikationen auf die des Betriebssystems verlassen.
- Dienste, die gleichzeitig von vielen genutzt werden können, haben vielerlei zusätzliche Konsistenz- und Synchronisierungsprobleme, für die nicht jede Art von Datenhaltung geeignet ist.
- Netzwerke bringen neue Arten von Ausfällen mit sich, wenn eine Netzwerkverbindung zusammenbricht oder es zu längeren „Hängern“ kommt.

Es gibt eine unüberschaubare Vielfalt an Netzwerk-Hardware, Transport-Protokollen und Schnittstellen zu deren Benutzung. Einen exzellenten Überblick hierfür bietet das Standardwerk *Computer Networks* von Andrew S. Tanenbaum (Prentice Hall, Third Edition, 1996, ISBN 0-13-349945-6). Im weiteren konzentrieren wir uns jedoch auf die Transport-Protokolle des Internets (TCP/IP).

4.2 IP-Adressen

Hier ist ein triviales Beispiel für den Aufbau einer Netzwerkverbindung. Als Klient wird hier *telnet* verwendet, das eine interaktive Nutzung von Netzwerkdiensten ermöglicht. Als trivialer Netzwerkdienst kommt hier *daytime* zum Einsatz, das die Eingabe auf dem Kommunikationskanal ignoriert und nur die aktuelle Zeit analog zum *date*-Kommando zurückgibt:

```
clonard$ telnet 134.60.54.12 13
Trying 134.60.54.12...
Connected to 134.60.54.12.
Escape character is '^]'.
Thu May 29 14:59:31 2008
Connection to 134.60.54.12 closed by foreign host.
clonard$
```

In diesem Beispiel besteht die Netzwerkadresse aus *134.60.54.12* (einer sogenannten IP-Adresse) und 13 (der Port-Nummer). IP-Adressen dienen dazu, einen Rechner zu adressieren, während Port-Nummern einem Dienst, der auf einem Rechner läuft, zugeordnet werden. In diesem Beispiel adressiert *134.60.54.12* unsere Theseus und die Port-Nummer 13 üblicherweise den *daytime*-Dienst.

Beim Internet gibt es für die Adressierung von Rechnern zwei Adressräume. IPv4 arbeitet mit 32-Bit-Adressen, ist seit dem 1. Januar 1983 in Benutzung und dominiert bis heute. Daneben gibt es auch IPv6, das mit 128-Bit-Adressen arbeitet, um einen Ausweg zu bieten für die drohende Knappheit mit Adressen bei IPv4. Im weiteren beschäftigen wir uns hier allerdings nur mit IPv4-Adressen, die kurz als IP-Adressen bezeichnet werden.

Die sogenannte *dottet-decimal*-Notation für IP-Adressen spezifiziert eine 32-Bit-Adresse als Folge von vier dezimalen Werten, die jeweils ein Byte angeben. *134.60.54.12* ist somit eine etwas übersichtlichere Schreibweise als 2252092940. Port-Nummern liegen im Bereich von 1 bis 65535, wobei der Bereich von 1 bis 1023 typischerweise für wohldefinierte Dienste reserviert ist. Diese Zuordnungen werden von der IANA (*Internet Assigned Numbers Authority*) vergeben und sind unter *http://www.iana.org/assignments/port-numbers* zu finden.

Beide Adressräume werden in hierarchisierter Form verwaltet. Ganz oben steht IANA, das große Teile des Adressraumes an regionale Institutionen weitergibt (ARIN für Amerika, RIPE für Europa, den Mittleren Osten und Zentralasien, APNIC für Asien, Australien und Ozeanien, AfriNIC für Afrika und LACNIC für Lateinamerika einschließlich Teile der Karibik). Die Universität Ulm hat hat seit 1989 den Adressbereich *134.60.0.0/16*[1] (damals noch von ARIN) zugeteilt bekommen.

Die Abbildung von Rechnernamen wie theseus.mathematik.uni-ulm.de in IP-Adressen wie *134.60.54.12* erfolgt dabei durch sogenannte Domain-Server, die ebenfalls hierarchisch

[1] Diese Schreibweise wird als CIDR-Notation bezeichnet, wobei CIDR für *Classless Inter-Domain Routing* steht. Die Zahl hinter dem Schrägstrich gibt dabei die Zahl der führenden Bits an, die den Netzwerkteil ausmachen. Bei 16 sind das die ersten beiden Bytes, also *134.60*. Alle Adressen in der Form *134.60.*.** gehören somit zur Universität Ulm.

organisiert sind. Wenn jemand von außerhalb den Namen theseus.mathematik.uni-ulm.de angibt, wird zunächst untersucht, wer für die Domain „de“ zuständig ist. Dies geschieht durch eine Anfrage an einen der sogenannten Root-Server, beispielsweise *198.41.0.4*, der selbst den Namen a.root-servers.net trägt. Dieser liefert unter anderem den Name-Server *194.0.0.53* mit dem Namen a.nic.de. Wenn a.nic.de nach uni-ulm.de gefragt wird, wird u.a. auf *134.60.1.111* mit dem Namen dns1.uni-ulm.de verwiesen. Letzterer Name-Server kann dann endlich auch theseus.mathematik.uni-ulm.de in *134.60.54.12* abbilden.

IP-Adressen wie *134.60.54.12* werden nur auf einer abstrakten Ebene zur Verfügung gestellt. IP-Adressen werden auf der darunterliegenen physischen Ebene und denen damit verbundenen Protokollen nicht verstanden. So wird beispielsweise beim Ethernet, das bei uns weitgehend an der Universität zum Einsatz kommt, mit 6-Byte-Adressen gearbeitet. Die Theseus hat beispielsweise die Ethernet-Adresse *0:14:4f:3e:a1:f0* (Bytes werden hier in Form von Hexzahlen angegeben). Diese Adressen sind jedoch nur lokal auf einem Ethernet-Segment von Bedeutung.

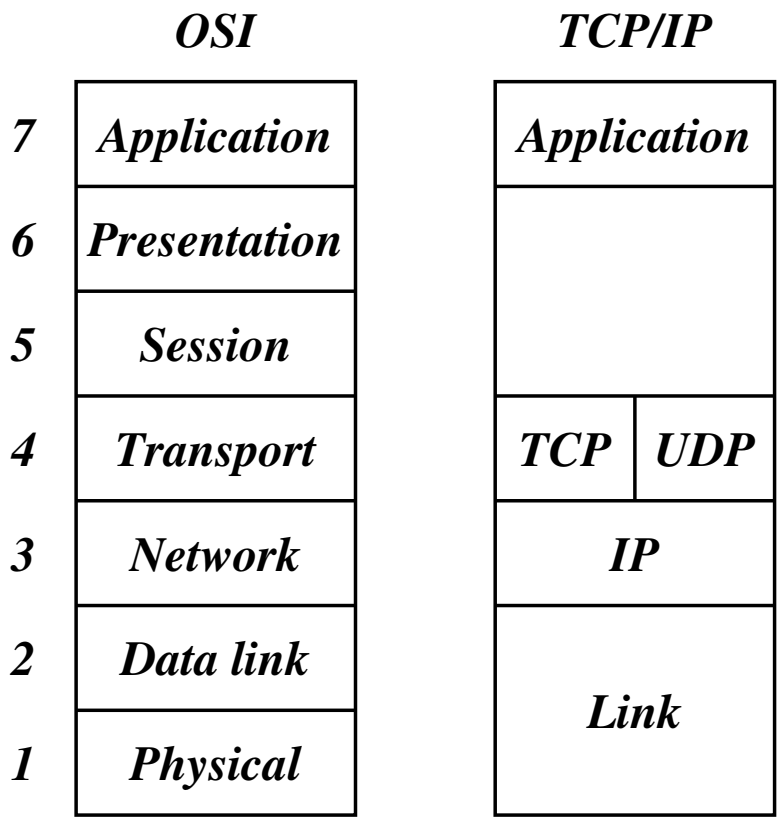

Abbildung 4.1: Schichtenmodell von OSI und TCP/IP

Wir haben also eine Schicht mit IP-Adressen und darunter eine Schicht, die von der verwendeten Netzwerk-Hardware abhängt. Aufbauend auf der Schicht mit IP-Adressen (IP-Protokoll) gibt es alternative Transport-Schichten, über die Pakete versendet werden können: Mittels UDP (*User Datagram Protocol*) können einzelne Pakete sehr effizient, aber unzuverlässig versendet werden, während TCP (*Transmission Control Protocol*) eine sichere Verbindung gewährleistet, die jedoch weniger effizient ist. Abbildung 4.1 zeigt auf der rechten Seite die vier Schichten, die beim Internet eine Rolle spielen: Anwendungen (wie etwa *telnet* und *daytime* im obigen Beispiel), Transport (UDP oder TCP), IP und die phy-

sische Ebene. Parallel dazu entstand 1983 das OSI-Referenz-Modell[2] der ISO (*International Standards Organization*), das eine etwas feinere Schichtung vorsieht. Die Präsentations- oder Sitzungs-Ebene fand jedoch nie ihren Weg in die Protokollhierarchie von TCP/IP.

4.3 Berkeley Sockets

Für TCP/IP gibt es zwei Schnittstellen, die beide zum POSIX-Standard gehören: Die Berkeley Sockets, die mit BSD 4.2 im Jahr 1983 eingeführt worden sind, und TLI (*Transport Layer Interface*), das auf Streams basiert und zuerst durch UNIX System V Release 3.0 im Jahr 1987 veröffentlicht wurde. Die Berkeley-Socket-Schnittstelle hat sich weitgehend durchgesetzt, da sie bei einigen UNIX-Systemen die einzige zur Verfügung stehende Schnittstelle ist.

Das Standardwerk über BSD 4.3 (*The Design and Implementation of the 4.3 BSD UNIX Operating System*, Samuel J. Leffler et al, Addison Wesley, 1989, ISBN 0-201-06196-1) nennt folgende Ziele:

- **Transparenz**: Die Kommunikation zwischen zwei Prozessen soll nicht davon abhängen, ob sie auf dem gleichen Rechner laufen oder nicht.
- **Effizienz**: Zu Zeiten von BSD 4.2 (also 1983) war dies ein außerordentlich wichtiges Kriterium wegen der damals noch sehr geringen Rechenleistung. Aus diesem Grund werden insbesondere keine weiteren System-Prozesse zur Kommunikation eingesetzt, obwohl dies zu mehr Flexibilität und Modularität hätte führen können.
- **Kompatibilität**: Viele bestehende Applikationen und Bibliotheken wissen nichts von Netzwerken und sollen dennoch in einem verteilten Umfeld eingesetzt werden können. Dies wurde dadurch erreicht, dass nach einem erfolgten Verbindungsaufbau (der z.B. von einem anderen Prozess durchgeführt werden kann) Ein- und Ausgabe in gewohnter Weise (wie bei Dateien, Pipelines oder Terminal-Verbindungen) erfolgen können.

Um den Anforderungen gerecht zu werden, wurden folgende Abstraktionen entwickelt:

- Die Abstraktion eines Kommunikationsbereiches macht es möglich, nicht nur TCP/IP zu unterstützen, sondern auch viele weitere Netzwerke (z.B. Appletalk, DECnet, IPX von Novell). Zu jedem Kommunikationsbereich gibt es unterschiedliche Namen (bzw. Adressen) für Kommunikationsendpunkte. Bei TCP/IP sind das die bekannten 32 Bit langen IP-Adressen (z.B. 134.60.166.2 für die Theseus) kombiniert mit der Port-Nummer des einzelnen Dienstes (z.B. 13 für *daytime*).
- Die Abstraktion eines Kommunikationsendpunktes (daher der Name „socket“), der mit der eines Dateideskriptors verbunden wird und über den eine bidirektionale Kommunikation möglich ist.
- Die Semantik einer Kommunikation.

Die Semantik einer Kommunikation umschließt bei jeder Verbindung eine Teilmenge der folgenden Punkte:

1. Daten werden in der Reihenfolge empfangen, in der sie abgeschickt worden sind.
2. Daten kommen nicht doppelt an.
3. Daten werden zuverlässig übermittelt.

[2] OSI steht für *Open Systems Interconnection*.

4. Einzelne Pakete kommen in der originalen Form an (d.h. sie werden weder zerstückelt noch mit anderen Paketen kombiniert).
5. Nachrichten außerhalb des normalen Kommunikationsstromes (*out-of-band messages*) werden unterstützt.
6. Die Kommunikation erfolgt verbindungs-orientiert, womit die Notwendigkeit entfällt, sich bei jedem Paket identifizieren zu müssen.

Die folgende Tabelle zeigt die Varianten, die von der Berkeley-Socket-Schnittstelle unterstützt werden:

Name	1	2	3	4	5	6
SOCK_STREAM	⋆	⋆	⋆		⋆	⋆
SOCK_DGRAM				⋆		
SOCK_SEQPACKET	⋆	⋆	⋆	⋆	⋆	⋆
SOCK_RDM	⋆	⋆	⋆	⋆		

Von diesen Varianten kommt *SOCK_STREAM* den Pipelines am nächsten, wenn davon abgesehen wird, dass die Verbindungen bei Pipelines nur unidirektional sind. Die Variante *SOCK_STREAM* lässt sich ziemlich direkt auf TCP abbilden, während UDP ziemlich genau durch *SOCK_DGRAM* widergespiegelt wird. Die Varianten *SOCK_SEQPACKET* (TCP-basiert) und *SOCK_RDM* (UDP-basiert) fügen hier noch weitere Funktionalitäten hinzu. Allerdings fand *SOCK_RDM* nicht den Weg in den POSIX-Standard und wird auch von einigen Implementierungen nicht angeboten. Im weiteren Verlauf werden wir uns nur mit *SOCK_STREAM*-Sockets beschäftigen.

4.4 Aufbau einer Netzwerk-Verbindung

Bis zu einem gewissen Grad mag hier zur Einführung die Analogie mit unserem Telefonsystem hilfreich sein. Bevor Sie Telefonanrufe entgegennehmen oder selbst anrufen können, benötigen Sie einen Telefonanschluss. Dieser Anschluss wird mit dem Systemaufruf *socket*() erzeugt:

int *sfd* = *socket*(*domain*, *type*, *protocol*);

Bei *domain* wird hier normalerweise *PF_INET* angegeben, um das IPv4-Protokoll auszuwählen. *PF* steht dabei für *protocol family*. Bei *type* kann eine der unterstützten Semantiken ausgewählt werden, also beispielsweise *SOCK_STREAM*. Der dritte Parameter *protocol* erlaubt in einigen Fällen eine weitere Selektion. Normalerweise wird hier schlicht 0 angegeben.

Nachdem der Anschluss existiert, fehlt noch eine zugeordnete Telefonnummer. Um bei der Analogie zu bleiben, haben wir eine Vorwahl (IP-Adresse) und eine Durchwahl (Port-Nummer). Auf einem Rechner können mehrere IP-Adressen zur Verfügung stehen. Es ist dabei möglich, nur eine dieser IP-Adressen zu verwenden oder alle, die zur Verfügung stehen. Bei den Port-Nummern ist eine automatische Zuteilung durch das Betriebssystem möglich. Alternativ ist es auch möglich, sich selbst eine Port-Nummer auszuwählen. Diese darf aber noch nicht vergeben sein und muss bei nicht-privilegierten Prozessen eine Nummer jenseits des Bereiches der wohldefinierten Port-Nummern sein, also typischerweise mindestens 1024 betragen. Die Verknüpfung eines Anschlusses mit einer vollständigen Adresse erfolgt mit dem Systemaufruf *bind*():

struct *sockaddr_in address* = {0};
address.sin_family = *AF_INET*;
address.sin_addr.s_addr = *htonl*(*INADDR_ANY*);
address.sin_port = *htons*(*port*);
bind(*sfd*, (**struct** *sockaddr* ∗) &*address*, **sizeof** *address*);

Die Datenstruktur **struct** *sockaddr_in* repräsentiert Adressen für IPv4, die aus einer IP-Adresse und einer Port-Nummer bestehen. Das Feld *sin_family* legt den Adressraum fest. Hier gibt es passend zur Protokollfamilie *PF_INET* nur *AF_INET* (*AF* steht hier für *address family*). Bei dem Feld *sin_addr.s_addr* lässt sich die IP-Adresse angeben. Mit *INADDR_ANY* übernehmen wir alle IP-Adressen, die zum eigenen Rechner gehören. Das Feld *sin_port* spezifiziert die Port-Nummer. Da Netzwerkadressen grundsätzlich nicht von der Byte-Anordnung eines Rechners abhängen dürfen, wird mit *htonl* (*host to network long*) der 32-Bit-Wert der IP-Adresse in die standardisierte Form konvertiert. Analog konvertiert *htons*() (*host to network short*) den 16-Bit-Wert *port* in die standardisierte Byte-Reihenfolge. Wenn die Port-Nummer vom Betriebssystem zugeteilt werden soll, kann bei *sin_port* auch einfach 0 angegeben werden. Der Datentyp **struct** *sockaddr_in* ist eine spezielle Variante des Datentyps **struct** *sockaddr*. Letzterer sieht nur ein Feld *sin_family* vor und ein generelles Datenfeld *sa_data*, das umfangreich genug ist, um alle unterstützten Adressen unterzubringen. Bei *bind*() wird der von *socket*() erhaltene Deskriptor angegeben (hier *sfd*), ein Zeiger, der auf eine Adresse vom Typ **struct** *sockaddr* verweist, und die tatsächliche Länge der Adresse, die normalerweise kürzer ist als die des Typs **struct** *sockaddr*. Schön sind diese Konstruktionen nicht, aber C bietet eben keine objekt-orientierten Konzepte, wenngleich die Berkeley-Socket-Schnittstelle sehr wohl polymorph und damit objekt-orientiert ist.

Grundsätzlich kann auf *bind*() auch verzichtet werden. Wenn dies geschieht, entspricht dies der Angabe von *INADDR_ANY* bei dem Feld *sin_addr.s_addr* und einer 0 bei *sin_port*. Das bedeutet dann, dass die Adresse vollständig von dem Betriebssystem vergeben wird. Die auf diese Weise erhaltene Adresse lässt sich mit der Funktion *getsockname*() ermitteln, sobald eine Verbindung eröffnet wird oder eingehende Verbindungen angenommen werden können.

Damit eingehende Verbindungen (oder Anrufe in unserer Telefon-Analogie) entgegengenommen werden können, muss *listen*() aufgerufen werden:

listen(*sfd*, *SOMAXCONN*);

Wohlgemerkt, nach *listen*() kann der Anschluss „klingeln“, aber noch sind keine Vorbereitungen getroffen, das Klingeln zu hören oder den Hörer abzunehmen. Der zweite Parameter bei *listen*() gibt an, wieviele Kommunikationspartner es gleichzeitig klingeln lassen dürfen. *SOMAXCONN* ist hier das Maximum, das die jeweilige Implementierung erlaubt. Der folgende Programmtext zeigt, wie mit *socket*() und *listen*() ein Anschluss angelegt werden kann:

Programm 4.1: Einrichtung eines Anschlusses (*newsocket.c*)

```
1  #include <sys/socket.h>
2  #include <netinet/in.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  void print_ip_addr(in_addr_t ipaddr) {
7      if (ipaddr == INADDR_ANY) {
8          printf("INADDR_ANY");
9      } else {
10         uint32_t addr = ntohl(ipaddr);
11         printf("%d.%d.%d.%d",
12             addr>>24, (addr>>16)&0xff,
13             (addr>>8)&0xff, addr&0xff);
14     }
15 }
16
```

```
17 int main() {
18     int sfd = socket(PF_INET, SOCK_STREAM, 0);
19     if (sfd < 0) exit(1);
20     if (listen(sfd, SOMAXCONN) < 0) exit(2);
21     struct sockaddr address;
22     socklen_t len = sizeof address;
23     if (getsockname(sfd, &address, &len) < 0) exit(3);
24     struct sockaddr_in * inaddr = (struct sockaddr_in *) &address;
25     printf("This␣is␣the␣address␣of␣my␣new␣socket:\n");
26     printf("IP␣Address:␣␣␣"); print_ip_addr(inaddr->sin_addr.s_addr);
27     printf("\n");
28     printf("Port␣Number:␣%d\n", (int) ntohs(inaddr->sin_port));
29 }
```

Die zugeteilte Port-Nummer wird dann mit *getsockname*() ermittelt. Interessanterweise gibt es keine von POSIX unterstützte Methode, alle verfügbaren IP-Adressen zu ermitteln. Dies ist allerdings kein gravierendes Problem, da typischerweise *INADDR_ANY* oder eine explizit (z.B. über die Kommandozeile) angegebene IP-Adresse verwendet wird. Die Entgegennahme eines Anrufes erfolgt mit *accept*():

```
struct sockaddr client_addr;
socklen_t client_addr_len = sizeof client_addr;
int fd = accept(sfd, &client_addr, &client_addr_len);
```

Liegt noch kein Anruf vor, blockiert *accept*() bis zum nächsten Anruf. Wenn mit *accept*() ein Anruf eingeht, wird ein Dateideskriptor auf den bidirektionalen Verbindungskanal zurückgeliefert. Normalerweise speichert *accept*() die Adresse des Klienten beim angegebenen Zeiger ab. Wenn als Zeiger 0 angegeben wird, entfällt dies.

Das folgende Beispiel zeigt einen einfachen Zeitdienst analog zu *daytime*, der die Port-Nummer 11011 verwendet:

Programm 4.2: Ein einfacher Zeitdienst (*timeserver.c*)

```
1  #include <netinet/in.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <sys/socket.h>
6  #include <sys/time.h>
7  #include <time.h>
8  #include <unistd.h>
9
10 #define PORT 11011
11
12 int main () {
13     struct sockaddr_in address = {0};
14     address.sin_family = AF_INET;
15     address.sin_addr.s_addr = htonl(INADDR_ANY);
16     address.sin_port = htons(PORT);
17
18     int sfd = socket(PF_INET, SOCK_STREAM, 0);
19     int optval = 1;
20     if (sfd < 0 ||
21         setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR,
22                    &optval, sizeof optval) < 0 ||
```

```
23              bind(sfd, (struct sockaddr *) &address,
24                      sizeof address) < 0 ||
25              listen(sfd, SOMAXCONN) < 0) {
26          perror("socket"); exit(1);
27      }
28
29      int fd;
30      while ((fd = accept(sfd, 0, 0)) >= 0) {
31          char timebuf[32]; time_t clock; time(&clock);
32          ctime_r(&clock, timebuf);
33          write(fd, timebuf, strlen(timebuf)); close(fd);
34      }
35  }
```

Zusätzlich kommt hier noch *setsockopt*() in den Zeilen 21 und 22 zum Einsatz, um die Option *SO_REUSEADDR* einzuschalten. Dies empfiehlt sich immer, wenn eine feste Port-Nummer verwendet wird. Fehlt diese Option, kann es passieren, dass bei einem Neustart des Dienstes die Port-Nummer nicht sofort wieder zur Verfügung steht, da noch alte Verbindungen nicht vollständig abgewickelt worden sind. In den Zeilen 31 bis 33 wird jeweils die aktuelle Zeitangabe in eine Zeichenkette konvertiert und mit Hilfe von *write*() an die zuvor geöffnete Verbindung verschickt. Mit *close*() wird die Verbindung dann auf der eigenen Seite aufgehängt.

Die Verbindungsaufnahme erfolgt mit *connect*(), wobei der bei *socket*() zurückgelieferte Deskriptor anzugeben ist und die Adresse des Kommunikationspartners:

```
connect(fd, (struct sockaddr *) &addr, sizeof addr);
```

Der folgende Programmtext zeigt einen Klienten, der den vorgestellten Zeitdienst kontaktiert, die Zeitangabe entgegennimmt und sie ausgibt:

Programm 4.3: Ein Klient des Zeitdienstes (*timeclient.c*)

```
1   #include <netdb.h>
2   #include <netinet/in.h>
3   #include <stdio.h>
4   #include <stdlib.h>
5   #include <string.h>
6   #include <sys/socket.h>
7   #include <unistd.h>
8
9   #define PORT 11011
10
11  int main (int argc, char** argv) {
12      char* cmdname = *argv++; --argc;
13      if (argc != 1) {
14          fprintf(stderr, "Usage: %s host\n", cmdname);
15          exit(1);
16      }
17      char* hostname = *argv;
18      struct hostent* hp;
19      if ((hp = gethostbyname(hostname)) == 0) {
20          fprintf(stderr, "unknown host: %s\n", hostname);
21          exit(1);
22      }
23      char* hostaddr = hp->h_addr_list[0];
```

```
24      struct sockaddr_in addr = {0};
25      addr.sin_family = AF_INET;
26      memmove((void *) &addr.sin_addr, (void *) hostaddr, hp->h_length);
27      addr.sin_port = htons(PORT);
28
29      int fd;
30      if ((fd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
31          perror("socket"); exit(1);
32      }
33      if (connect(fd, (struct sockaddr *) &addr, sizeof addr) < 0) {
34          perror("connect"); exit(1);
35      }
36
37      char buffer[BUFSIZ]; ssize_t nbytes;
38      while((nbytes = read(fd, buffer, sizeof buffer)) > 0 &&
39              write(1, buffer, nbytes) == nbytes);
40  }
```

timeclient
Der Klient erhält über die Kommandozeile den Namen des Rechners, auf dem der Zeitdienst zur Verfügung steht. Für die Abbildung eines Rechnernamens in eine IP-Adresse wird die Funktion *gethostbyname*() benötigt, die im Erfolgsfalle eine oder mehrere IP-Adressen liefert, unter denen sich der Rechner erreichen lässt. In Zeile 23 wird die erste IP-Adresse ausgewählt, die bereits in der richtigen Byte-Anordnung vorliegt. Mit Hilfe von *memmove*() wird sie zu *addr.sin_addr* in Zeile 26 kopiert. Danach kann in Zeile 33 mit der zuvor erzeugten Socket die gewünschte Verbindung mit Hilfe von *connect*() aufgebaut werden. Die **while**-Schleife in Zeile 38 gibt dann alles aus, was sich über die Netzwerkverbindung einlesen ließ.

4.5 Gepufferte Ein- und Ausgabe für Netzwerkverbindungen

Die Ein- und Ausgabe über Netzwerkverbindungen bringt in Vergleich zur Behandlungen von Dateien und interaktiven Benutzern einige Veränderungen mit sich. Wenn eine Verbindung des Typs *SOCK_STREAM* zum Einsatz gelangt, so kommen die Daten zwar in der korrekten Reihenfolge an, jedoch nicht in der ursprünglichen Paketisierung. Als ursprüngliche Pakete werden hier die Daten betrachtet, die mit Hilfe eines einzigen Aufrufs von *write*() geschrieben werden:

```
const char greeting[] = "Hi, how are you?\r\n";
ssize_t nbytes = write(sfd, greeting, sizeof greeting);
```

Wenn beispielsweise bei einer Netzwerkverbindung immer vollständige Zeilen mit *write*() geschrieben werden, so ist es möglich, dass die korrespondierende *read*()-Operation nur einen Teil einer Zeile zurückliefert oder auch ein Fragment, das sich über mehr als eine Zeile erstreckt. Diese Problematik legt es nahe, nur zeichenweise einzulesen, wenn genau eine einzelne Zeile eingelesen werden soll:

```
char ch;
stralloc line = {0};
while (read(fd, &ch, sizeof ch) == 1 && ch != '\n') {
    stralloc_append(&line, &ch);
}
```

Diese Vorgehensweise ist jedoch außerordentlich ineffizient, weil Systemaufrufe wie *read* () zu einem Kontextwechsel zwischen dem aufrufenden Prozess und dem Betriebssystem führen. Wenn ein Kontextwechsel für jedes einzulesende Byte initiiert wird, dann ist der betroffene Rechner mehr mit Kontextwechseln als mit sinnvollen Tätigkeiten beschäftigt. Wenn jedoch mit

char *buf*[512];
ssize_t nbytes = *read*(*fd*, *buf*, **sizeof** *buf*)

eingelesen wird, ist möglicherweise mehr als nur die gewünschte Zeile in *buf* zu finden. Oder auch nur ein Teil der Zeile.

Entsprechend ist eine gepufferte Eingabe notwendig, bei der die Eingabe-Operationen aus einem Puffer versorgt werden, der, wenn er leer wird, mit Hilfe einer *read*()-Operation aufzufüllen ist. Die Datenstruktur für einen Eingabe-Puffer benötigt entsprechend einen Dateideskriptor, einen Puffer und einen Positionszeiger innerhalb des Puffers:

typedef struct *inbuf* {
    int *fd*;
    stralloc buf;
    unsigned int *pos*;
} *inbuf*;

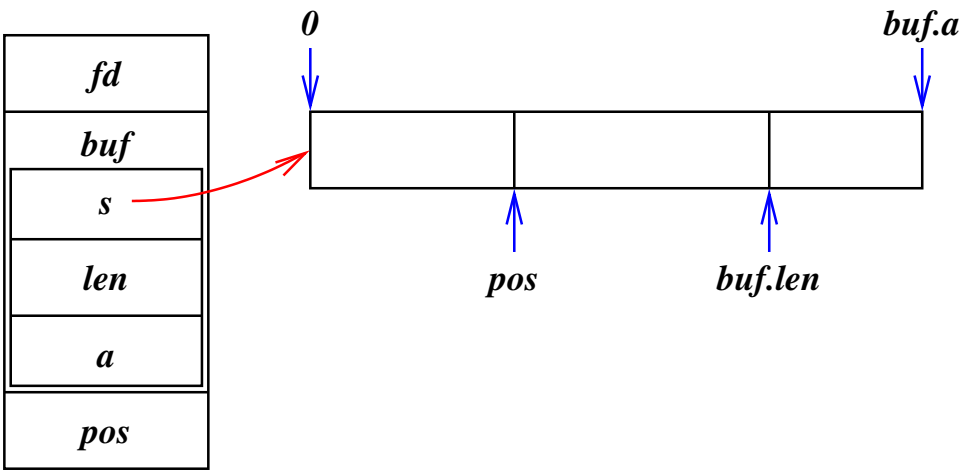

Abbildung 4.2: Struktur eines Eingabe-Puffers

Abbildung 4.2 dient der Illustrierung des Eingabe-Puffers. Als Puffer-Bereich dient *buf.s* mit einem zulässigen Indexbereich von 0 bis *buf.a*−1. Entsprechend dem letzten erfolgreichen Aufruf von *read*() ergibt sich ein Füllgrad des Puffers, der von *buf.len* repräsentiert wird. Der Positionszeiger *pos* begann unmittelbar nach der *read*()-Operation auf Position 0 und wandert bei jeder Einlese-Operation aus dem Puffer der Grenze von *buf.len* entgegen. Wird die Grenze erreicht, so ist die nächste *read*()-Operation fällig.

Der folgende Programmtext zeigt die entsprechende Schnittstelle:

Programm 4.4: Schnittstelle für den Eingabe-Puffer (*inbuf.h*)

```
1 #ifndef INBUF_H
2 #define INBUF_H
3
4 #include <stralloc.h>
5 #include <unistd.h>
```

```
6
7  typedef struct inbuf {
8      int fd;
9      stralloc buf;
10     unsigned int pos;
11 } inbuf;
12
13 /* set size of input buffer */
14 int inbuf_alloc(inbuf* ibuf, unsigned int size);
15
16 /* works like read(2) but from ibuf */
17 ssize_t inbuf_read(inbuf* ibuf, void* buf, size_t size);
18
19 /* works like fgetc but from ibuf */
20 int inbuf_getchar(inbuf* ibuf);
21
22 /* move backward one position */
23 int inbuf_back(inbuf* ibuf);
24
25 /* release storage associated with ibuf */
26 void inbuf_free(inbuf* ibuf);
27
28 #endif
```

Die Funktion *inbuf_alloc*() dient dazu, die Größe des Puffers einzurichten, wobei eine sinnvolle Voreinstellung automatisch gewählt wird, wenn der Aufruf dieser Funktion unterbleibt. Als Einlese-Operationen vom Puffer dienen *inbuf_read*() und *inbuf_getchar*(), die sich in ihrer Aufrufsemantik an *read*() bzw. *fgetc*() orientieren. Ein zuviel gelesenes Zeichen kann mit *inbuf_back*() wieder zum erneuten Einlesen zur Verfügung gestellt werden. Mit *inbuf_free*() wird der Puffer deallokiert.

Programm 4.5: Implementierung des Eingabe-Puffers (*inbuf.c*)

```
1  #include <errno.h>
2  #include <string.h>
3  #include <unistd.h>
4  #include "inbuf.h"
5
6  /* set size of input buffer */
7  int inbuf_alloc(inbuf* ibuf, unsigned int size) {
8      return stralloc_ready(&ibuf->buf, size);
9  }
10
11 /* works like read(2) but from ibuf */
12 ssize_t inbuf_read(inbuf* ibuf, void* buf, size_t size) {
13     if (size == 0) return 0;
14     if (ibuf->pos >= ibuf->buf.len) {
15         if (ibuf->buf.a == 0 && !inbuf_alloc(ibuf, 512)) return -1;
16         /* fill input buffer */
17         ssize_t nbytes;
18         do {
19             errno = 0;
20             nbytes = read(ibuf->fd, ibuf->buf.s, ibuf->buf.a);
21         } while (nbytes < 0 && errno == EINTR);
```

```
22          if (nbytes <= 0) return nbytes;
23          ibuf->buf.len = nbytes;
24          ibuf->pos = 0;
25      }
26      ssize_t nbytes = ibuf->buf.len - ibuf->pos;
27      if (size < nbytes) nbytes = size;
28      memcpy(buf, ibuf->buf.s + ibuf->pos, nbytes);
29      ibuf->pos += nbytes;
30      return nbytes;
31  }
32
33  /* works like fgetc but from ibuf */
34  int inbuf_getchar(inbuf* ibuf) {
35      char ch;
36      ssize_t nbytes = inbuf_read(ibuf, &ch, sizeof ch);
37      if (nbytes <= 0) return -1;
38      return ch;
39  }
40
41  /* move backward one position */
42  int inbuf_back(inbuf* ibuf) {
43      if (ibuf->pos == 0) return 0;
44      ibuf->pos--;
45      return 1;
46  }
47
48  /* release storage associated with ibuf */
49  void inbuf_free(inbuf* ibuf) {
50      stralloc_free(&ibuf->buf);
51  }
```

Der vorstehende Programmtext zeigt die zugehörige Implementierung, wobei hier insbesondere *inbuf_read*() interessant ist. In Zeile 14 wird untersucht, ob der Puffer bereits geleert ist, d.h. ob *pos* bereits *buf.len* erreicht hat. Falls ja, wird in den Zeilen 15 bis 24 der Puffer neu gefüllt. In Zeile 15 wird zunächst untersucht, ob der Puffer bereits allokiert worden ist. Falls nicht, wird dies mit der Standardgröße von 512 Bytes versucht. In Zeile 20 erfolgt die *read*()-Operation, bei der grundsätzlich versucht wird, den gesamten Puffer zu füllen. Allerdings ist damit zu rechnen, dass die Zahl der tatsächlich gelesenen Bytes *nbytes* niedriger als *buf.a* ist. Dies liegt daran, dass das Betriebssystem uns bereits vorhandene Daten sofort zur Verfügung stellt, selbst wenn es sich um eine geringere Quantität als angefordert handelt. Dies stellt sicher, dass effizientes Einlesen mit größen Puffergrößen ohne unnötiges Blockieren möglich ist. Die *read*()-Operation selbst ist in eine Schleife in den Zeilen 18 bis 21 eingebettet, die sicherstellt, dass es zu einem erneuten Versuch kommt, falls *read*() wegen einer Signalunterbrechung nicht erfolgreich sein konnte.

Sobald sichergestellt ist, dass mindestens ein Byte in dem Puffer verfügbar ist, wird *nbytes* in Zeile 26 auf die maximal mögliche Rückgabequantität gesetzt. Wurden weniger verlangt, so wird *nbytes* in Zeile 27 entsprechend zurückgesetzt. In Zeile 28 wird die zurückzuliefernde Quantität an Bytes aus dem Puffer in *buf* mit Hilfe von *memcpy*() kopiert. Der erste Parameter (*buf*) zeigt dabei auf das Ziel, der zweite Parameter (*buf.s + pos*) auf die Quelle und der dritte Parameter gibt die Zahl der zu kopierenden Bytes an (*nbytes*). Nach der Kopieraktion wird *pos* entsprechend aktualisiert.

Die Ausgabe sollte ebenfalls gepuffert erfolgen, um die Zahl der Systemaufrufe zu minimieren. Ein Positionszeiger ist nicht erforderlich, wenn Puffer grundsätzlich vollständig

an *write*() übergeben werden. Hier ist das einzige Problem, dass die *write*()-Operation unter Umständen nicht den gesamten gewünschten Umfang akzeptiert und nur einen Teil der zu schreibenden Bytes akzeptiert und entsprechend eine geringere Quantität als Wert zurückgibt.

Der folgende Programmtext zeigt die Schnittstelle für den Ausgabe-Puffer:

Programm 4.6: Schnittstelle für den Ausgabe-Puffer (*outbuf.h*)

```
1  #ifndef OUTBUF_H
2  #define OUTBUF_H
3
4  #include <stralloc.h>
5  #include <unistd.h>
6
7  typedef struct outbuf {
8      int fd;
9      stralloc buf;
10 } outbuf;
11
12 /* works like write(2) but to obuf */
13 ssize_t outbuf_write(outbuf* obuf, void* buf, size_t size);
14
15 /* works like fputc but to obuf */
16 int outbuf_putchar(outbuf* obuf, char ch);
17
18 /* write contents of obuf to the associated fd */
19 int outbuf_flush(outbuf* obuf);
20
21 /* release storage associated with obuf */
22 void outbuf_free(outbuf* obuf);
23
24 #endif
```

Die Funktion *outbuf_write*() schreibt in den gegebenen Puffer und entspricht ansonsten dem Systemaufruf *write*(). Mit Hilfe von *outbuf_putchar*() können bequem einzelne Zeichen in den Puffer ausgegeben werden. Beide Schreiboperationen führen nur zur Verlängerung des Pufferinhalts, ohne dass dieser mit Hilfe einer *write*()-Operation geleert wird. Letzteres ist nur durch den Aufruf von *outbuf_flush*() möglich. Wenn der Puffer nicht mehr benötigt wird, kann er durch *outbuf_free*() freigegeben werden. Es folgt die zugehörige Implementierung:

Programm 4.7: Implementierung des Ausgabe-Puffers (*outbuf.c*)

```
1  #include <errno.h>
2  #include <stralloc.h>
3  #include <string.h>
4  #include "outbuf.h"
5
6  /* works like write(2) but to obuf */
7  ssize_t outbuf_write(outbuf* obuf, void* buf, size_t size) {
8      if (size == 0) return 0;
9      if (!stralloc_readyplus(&obuf->buf, size)) return -1;
10     memcpy(obuf->buf.s + obuf->buf.len, buf, size);
11     obuf->buf.len += size;
12     return size;
```

```
13 }
14
15 /* works like fputc but to obuf */
16 int outbuf_putchar(outbuf* obuf, char ch) {
17     if (outbuf_write(obuf, &ch, sizeof ch) <= 0) return -1;
18     return ch;
19 }
20
21 /* write contents of obuf to the associated fd */
22 int outbuf_flush(outbuf* obuf) {
23     ssize_t left = obuf->buf.len; ssize_t written = 0;
24     while (left > 0) {
25         ssize_t nbytes;
26         do {
27             errno = 0;
28             nbytes = write(obuf->fd, obuf->buf.s + written, left);
29         } while (nbytes < 0 && errno == EINTR);
30         if (nbytes <= 0) return 0;
31         left -= nbytes; written += nbytes;
32     }
33     obuf->buf.len = 0;
34     return 1;
35 }
36
37 /* release storage associated with obuf */
38 void outbuf_free(outbuf* obuf) {
39     stralloc_free(&obuf->buf);
40 }
```

In *outbuf_write*() wird in Zeile 9 darauf geachtet, dass der Puffer genügend Platz für den aufzunehmenden Inhalt aufweist, wonach in Zeile 10 der Kopiervorgang mit Hilfe von *memcpy*() durchgeführt werden kann. Danach muss nur noch *buf.len* in Zeile 11 angepasst werden. In der Funktion *outbuf_flush*() gibt es zwei Schleifen. Die äußere Schleife in den Zeilen 24 bis 32 sorgt dafür, dass der gesamte Puffer-Inhalt geschrieben wird, da einzelne *write*()-Operationen die Freiheit haben, nur einen Teil umzusetzen. Mit Hilfe der Variablen *left* und *written* wird vermerkt, wieviel noch zu schreiben ist bzw. wieviel bereits geschrieben wurde. Die innere Schleife in den Zeilen 26 bis 29 wiederholt die *write*()-Operation im Falle von Unterbrechungen.