Kapitel 6

client server

Bisher wurden Methoden besprochen, die auf einem Rechner funktionieren. Dabei wurde bei threads bereits wirklich parallel gerechnet, sofern mehrere Prozessoren vorhanden waren. Die nächste Stufe ist Parallelität innerhalb eines Netzwerkes, und dies ist dann ein Netz aus Rechnern, die mit eigener CPU parallel rechnen können. Die heutzutage hierbei meist verwendete Vorgehensweise ist die client-server-Methode. Ein Prozess pro Rechner, genannt Server stellt seine Leistung innerhalb des Netzes zur Verfügung. Ein client kann nun an den Server auf einen Rechner herantreten und um die zu Verfügungstellung der entsprechenden Dienstleistung nachfragen. Ein verbreitetes Beispiel ist der ftp Dienst. Durch den Aufruf

```
$ftp ftp.uni-bayreuth.de
```

starte ich einen ftp client auf meinen Rechner, der an den ftp server des Rechners ftp.uni-bayreuth.de herantritt und um die Ausführung des file-transfer-protokoll bittet. Die client server Methode ist eine Art der Netzwerk Kommunikation. Hierfür gibt es verschiedene Standards (socket, TLI (transport layer interface), STREAMS). Es gibt keinen POSIX Standard. Es wird ein eigenes Interface verwendet (UICI = universal internet communication interface) anhand dessen die client server Methode vorgestellt wird. Die nötigen Routinen werden mittels aller drei Netzwerk Kommunikation Interfaces implementiert. Für sockets und TLI wird der Spec 1170 Standard befolgt.

6.1 Verbindungsorientiert, Verbindunslos

Bei dem zeitlichen Ablauf der Kommunikation zwischen client und server unterscheidet man zwischen verbindungsorientierten Protokoll und verbindunslosen Protokoll. Bei dem verbindungsorientierten Protokoll wartet der Server auf einen client, der eine Verbindung wünscht. Dann kann ein filedescriptor erzeugt werden, und mittels diesen wird kommuniziert. Die andere Variante ist verbindunslos, hier sendet der client einen request und wartet anschliessend auf die Antwort. Hier muss natürlich in der Antwort eine Identifikation enthalten sein. Dafür ist der Overhead vorher geringer. Im Weiteren

wird der Schwerpunkt auf ein verbindungsorientiertes Protokoll gelegt. Steht erstmal ein Kommunikationskanal zur Verfügung kann der request mit verschiedenen Strategien gehandelt werden:

- serial server: Der request wird abgearbeitet, bis der nächste behandelt wird. Dies ist nicht geschickt bei servern, die typischerweise viele und lange requests haben (z.B ftp)
- parent server: Der request wird angenommen, und mittels fork an einen Sohn übertragen.
- threaded server: Besser, vorausgesetzt keine Probleme durch den gemeinsamen Adressraum. (thread safe)

6.2 UICI

Es werden die nötigen Routinen um ein client server Modell zu implementieren vorgestellt. Eine konkrete Implementierung wird später besprochen.

UICI Prototyp	Beschreibung
int u_open(u_port_t port)	öffnet file descriptor für einen port
int u_listen(int fd, char *hostname)	wartet auf ein request bei fd, liefert communication fd und hostname
int u_connect(u_port_t port, char *hostname)	baut verbindung zum server am port auf, liefert communication fd
int u_close(fd)	
ssize_t u_read(int fd, char *buf size_t nbyte)	liest bis zu nbyte vom fdin buf, Rückgabe Anzahl der gelesenen Bytes
ssize_t u_write(int fd, char *buf size_t nbyte)	schreibt nbyte vom buf auf fd, Rückgabe Anzahl der geschriebenen Bytes
void u_error(char *errmsg)	Ausgabe von errmsg+UICI Fehler Meldung
int u_sync(int fd)	

6.2.1 UICI Server

Der Server wird wie folgt programmiert:

- Mit u_open wird ein listening fd (=filedescriptor) an einem festen Port erzeugt
- Mit u_listen wird nun auf ein request gewartet (blocking). Ist ein request gekommen erhält man den Namen des anfragenden Rechners und ein fd für die Kommunikation mit dem client. Dies ist eine bidirectionale connection spezifische Verbindung.
- Mit u_read und u_write mit dem client bezüglich der Anfrage kommunizieren
- Mit u_close die Kommunikations file descriptoren schliessen und weiter mit u_listen warten.

Als Beispiel wollen wir einen Server schreiben der aus dem Netz liest und dieses auf stdout schreibt. Dies soll durch die folgende Hilfsroutine gemacht werden.

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include "uici.h"
#define BLKSIZE 1024
int copy_from_network_to_file(int communfd, int filefd)
{
int bytes_read;
int bytes_written;
int bytes_to_write;
int total_bytes = 0;
char buf[BLKSIZE];
char *bufp;

for ( ; ; ) {
   if ((bytes_read = u_read(communfd, buf, BLKSIZE)) <
0) {
       u_error("Server read error");
       break;
   } else if (bytes_read == 0) {
       fprintf(stderr, "Network end-of-file\n");
       break;
   } else { /* allow for interruption of write by signal
*/
       for (bufp = buf, bytes_to_write = bytes_read;
            bytes_to_write > 0;
            bufp += bytes_written, bytes_to_write -=
bytes_written){
       bytes_written = write(filefd, bufp,
bytes_to_write);
       if ((bytes_written) == -1 && (errno != EINTR)){
          perror("Server write error");
          break;
       } else if (bytes_written == -1)
          bytes_written = 0;
       total_bytes += bytes_written;
       }/*for*/
       if (bytes_written == -1)
       break;
       }/*else*/
   } /*for;;;*/
return total_bytes;
}
```

Es wird bis EOF gelesen, dazu wird die UICI Routine u_read verwendet. Der zugehörige Server schaut so aus:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>
#include "uici.h"
int copy_from_network_to_file(int communfd, int filefd);
/*
* UICI Server
* Open a UICI port specified as a command-line argument
* and listen for a request. When a request
* arrives, use the provided communication file
descriptor to
* read from the UICI connection and echo to standard
output
* until the connection is terminated. Then the server
resumes
* listening for additional requests.
*/
void main(int argc, char *argv[])
{
u_port_t portnumber;
int listenfd;
int communfd;
char client[MAX_CANON];
int bytes_copied;

if (argc != 2) {
fprintf(stderr, "Usage: %s port\n", argv[0]);
exit(1);
}
portnumber = (u_port_t) atoi(argv[1]);
if ((listenfd = u_open(portnumber)) == -1) {
u_error("Unable to establish a port connection");
exit(1);
}
while ((communfd = u_listen(listenfd, client)) != -1) {
  fprintf(stderr, "A connection has been made to
%s\n",client);
  bytes_copied = copy_from_network_to_file(communfd,
STDOUT_FILENO);
  fprintf(stderr, "Bytes transferred = %d\n",
bytes_copied);
  u_close(communfd);
}
exit(0);
}
```

Das Programm wird mit `a.out portnumber` gestartet. Es wurde die einfachere serial server Strategie gewählt.

6.2.2 UICI client

Der Client wird wie folgt programmiert:

- Mit u_connect an den bekannten festen Port am bekannten Rechner wenden, man erhält den fd für die bidirektionale Kommunikation.
- Mit u_read und u_write mit dem server bezüglich der Anfrage kommunizieren
- Mit u_close die Kommunikations file descriptoren schliessen

Für unser Beispiel der source code für den client:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include "uici.h"
#define BLKSIZE 1024
/*
* UICI Client
* Make a UICI connection request to the host and port
specified
* as command-line arguments. Read from standard input
and
* write to the UICI communication file descriptor until
* end-of-file.
*/
void main(int argc, char *argv[])
{
u_port_t portnumber;
int communfd;
ssize_t bytesread;
char buf[BLKSIZE];
if (argc != 3) {
fprintf(stderr, "Usage: %s host port\n", argv[0]);
exit(1);
}
portnumber = (u_port_t)atoi(argv[2]);
if ((communfd = u_connect(portnumber, argv[1])) < 0) {
u_error("Unable to establish an Internet connection");
exit(1);
}
fprintf(stderr, "A connection has been made to
%s\n",argv[1]);
for ( ; ; ) {
   if ((bytesread = read(STDIN_FILENO, buf, BLKSIZE)) <
0) {
       perror("Client read error");
       break;
   } else if (bytesread == 0) {
       fprintf(stderr, "Client detected end-of-file on
input\n");
       break;
   } else if (bytesread !=
              u_write(communfd, buf, (size_t)bytesread))
{
       u_error("Client write_error");
       break;
   }
   }/*for*/
u_close(communfd);
exit(0);
}
```

Um den client zu starten muss man a.out hostname portnummer eingeben. Nachfolgend erscheint die Eingabe des clients beim server auf stdout. Greifen mehrere clients zu, so werden Sie vom vorgestellten (serial) Server nacheinander abgearbeitet.

6.2.3 parent server

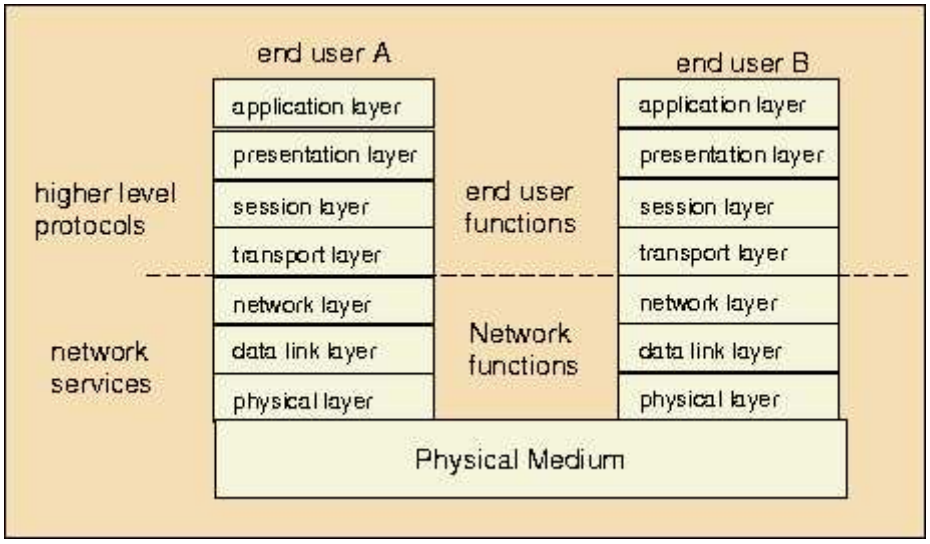
Die Tatsache, dass die Anfragen der clients nacheinander abgearbeitet werden ist von Nachteil bei vielen Anfragen, das kann zu langen Wartezeiten führen, dies wird durch die vorgestellte parent server Strategie umgangen:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>
#include <sys/wait.h>
#include "uici.h"
int copy_from_network_to_file(int communfd, int filefd);
/*
* UICI Server
* Open a UICI port specified as a command-line argument
* and listen for requests. When a request
* arrives, fork a child to handle the communication and
* listen again.
*/
void main(int argc, char *argv[])
{ u_port_t portnumber;
int listenfd;
int communfd;
char client[MAX_CANON];
int bytes_copied;
int child;
if (argc != 2) {
fprintf(stderr, "Usage: %s port\n", argv[0]);
exit(1);
}
portnumber = (u_port_t) atoi(argv[1]);
if ((listenfd = u_open(portnumber)) == -1) {
  u_error("Unable to establish a port connection");
  exit(1);
}
while ((communfd = u_listen(listenfd, client)) != -1) {
  fprintf(stderr, "[%ld]: A connection has been made to %s\n",
     (long) getpid(), client);
  if ((child = fork()) == -1) {
    fprintf(stderr, "Could not fork a child\n");break;}
if (child == 0) { /* child code */
  close(listenfd);
  fprintf(stderr, "[%ld]: A connection has been made to %s\n",
     (long) getpid(), client);
  bytes_copied =
     copy_from_network_to_file(communfd, STDOUT_FILENO);
  close(communfd);
  fprintf(stderr, "[%ld]:Bytes transferred = %d\n",
    (long) getpid(), bytes_copied);
  exit(0);
} else { /* parent code */
  u_close(communfd);
  while (waitpid(-1, NULL, WNOHANG) > 0);
  }
}
exit(0);}
```

Der Aufruf `while (waitpid(-1, NULL, WNOHANG) > 0);` hat zur Folge, dass zombies eingesammelt werden, d.h. beendete server instanzen. Der 1. Parameter -1 bedeuet alle möglichen Sohnprozesse. Das WNOHANG bedeutet es wird nicht gewartet, d.h. wenn kein Sohnprozess da ist, wird das waitpid sofort beendet. Man will ja auch wieder weiter warten auf neue Anfragen.

6.3 7 Schichten Modell

Durch die ISO (international standards organization) wurde ein Standard für Netzwerkanwendungen geschaffen. Dies ist das sog. 7 Schichten Modell



Jede dieser 7 Schichten enthält Funktionen um eine Kommunikation zu erlauben. Ausserdem kommunizieren Funktionen aus einem Layer nur mit Funktionen in der gleichen Schicht oder einer direkt benachbarten Schicht. Je tiefer die Schicht desto hardware naher ist die Kommunikation. Physical layer und data link layer behandeln den Daten transport zwischen zwei Netzwerkknoten. Ein weit verbreitetes Protokol ist das Ethernet Protokol, hier wird ein Netzwerkknoten durch eine 6 Byte Ethernet Adresse bezeichnet, diese ist fest auf dem Netzwerkinterface eingetragen. Die führenden Bytes bezeichnen dabei den Hersteller. Andere Protokolle auf dieser Ebene sind ATM, token ring, FDDI. Die nächste Schicht, das transport layer behandelt das routing etc, um von einem Netzwerk zum nächsten zu kommen. Hier ist das am weitest verbreitete Protokoll, das IP, das internet protokoll. Auf dieser Ebene werden die Netzwerkknoten durch 4 Byte, der sog. Internet Adresse identifiziert. (z.B. 198.86.40.81) Alternativ werden auch Namen verwendet, die mittels dem Systemaufruf gethostbyaddr in die IP Adresse umgewandelt werden. Bei den IP Adressen zeigen die führenden Bytes sog. domains an, z.B. 132.180 = Uni Bayreuth. Auf der Ebene des transport layers geht es um die Kommunikation zwischen den Knoten, die beiden verbreitesten Protokolle in der UNIX Welt sind TCP (transmisson control protocol), welches verbindungs orientiert ist und UDP (user datagram protocol) welches verbindungslos ist und auch nicht gewährleistet, dass die Daten wirklich entgegenommen wurden. Da mehr als ein Datenaustausch zwischen zwei Knoten passieren kann verwenden sowohl TCP als auch UDP sog. ports, die sind 2 Byte zahlen die für einzelne services zur Verfügung gestellt werden. Dabei sind einige üblicherweise reserviert (/etc/services) ftp=21, telnet=23, tftp=69. Verwendet man eigene ports, so sollten Zahlen >7000 verwendet werden. Auf der nächsten Ebene (session layer) werden interfaces zur darunter liegenden Schicht zur Verfügung gestellt. Zwei verbreitete sind (Berkeley) sockets und TLI (transport

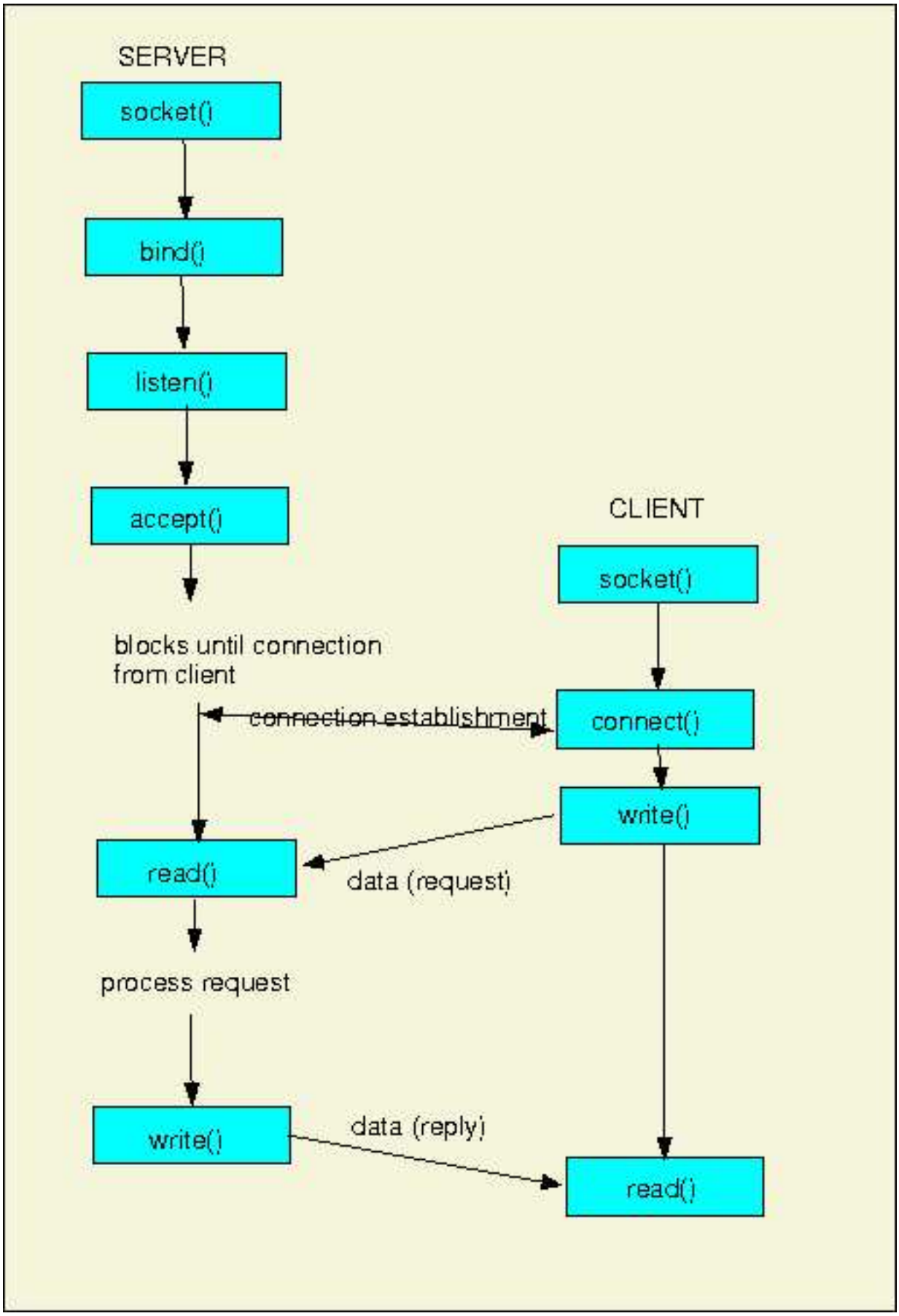
layer interface). Darüber liegende Schichten sind dann schon Anwendungsabhängig (z.B Komprimierung).

6.4 Socket Implementierung von UICI

Sockets wurden bereits Anfang der 80er Jahre mit BSD4.1c eingeführt, und in Spec 1170 wurde die BSD4.3 Version standardisiert. Seit 2001 sind die sockets auch in POSIX standardisiert. Folgende Tabelle zeit die Socket Aufrufe, die nötig sind um UICI zu implementieren.

UICI	Sockets
u_open	socket bind listen
u_listen	accept
u_connect	socket connect
u_read	read
u_write	write
u_sync	

Nachfolgend ein Bild zu einer Verbindungs orientierten Socket Kommunikation:


SERVER
socket()
bind()
listen()
accept()
blocks until connection
from client
CLIENT
socket()
connection establishment
connect()
write()
read()
data (request)
process request
write()
data (reply)
read()

Zuerst wird beim Server mit socket ein handle erzeugt, welches mit bind an einen Netzwerkknoten angebunden wird. Mit listen wird eine Queue erzeugt, dann wartet der server mit accept. Der client erzeugt auch ein handle und kann dann mit connect an den server gehen. Die beiden handles, werden manchmal Übertragungsendpunkte genannt. Steht die Verbindung kann mit dem Systemaufruf read/write gearbeitet werden. socket erzeugt die Endpunkte und liefert file descriptoren:

```
NAME
socket - create an endpoint for communication
SYNOPSIS
#include <sys/types.h>
#include <sys/socket.h>
int socket(int family, int type, int protocol);
```

Der erste parameter gibt die Protokolfamilie an, dies sind

- AF_UNIX Für rechnerinterne Verbindungen, zur interprocess Kommunikation
- AF_INET Für Internet Protokolle, dies erlaubt eine Verbindung zwischen Rechnern.

AF bedeutet dabei adress family. Der zweite Parameter ist für die Socket Typen:

- SOCK_STREAM Für eine verbindungsorientierte zuverlässige bidirektionale Verbindung, typischerweise TCP.
- SOCK_DGRAM Für eine verbindungslose Verbindung mit fester Datengrösse, üblicherweise UDP.

Der dritte Parameter legt eigentlich das Protokoll fest, aber es gibt für die Typen meist nur ein Protokoll, daher wird hier 0=default verwendet. Der nächste Aufruf auf server-Seite ist bind:

```
NAME
bind - bind a name to a socket
SYNOPSIS
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *my_addr,
        int addrlen);
```

Der erste Parameter ist der fd, den man aus socket bekommen hat. Der dritte Parameter ist die Grösse in Bytes der Struktur sockaddr, die zweiter Parameter ist. Sie enthält eine family Namen und Protokoll spezifische Informationen. Die wirkliche Adress Struktur ist Protokoll abhängig, für das Internet:

```
/*
* Socket address, internet style.
*/
struct sockaddr_in {
sa_family_t sin_family;
in_port_t sin_port;
struct in_addr sin_addr;
char sin_zero[8];
};
```

Dabei ist sin_family immer AF_INET. Unter sin_port wird die Port Nummer gespeichert, sin_addr steuert die Zugriffsrechte, mit INADDR_ANY kann jeder Internet Rechner eine Verbindung zu dem Server aufbauen. Die restlichen Bytes sind Füllbytes um zu erreichen, dass struct sockaddr für alle Adress Familien gleich gross ist. Der letzte Aufruf um ein u_open zu bekommen ist listen:

```
NAME
listen - listen for connections on a socket
SYNOPSIS
#include <sys/types.h>
#include <sys/socket.h>
int listen(int s, int backlog);
```

damit wird eingestellt wieviele Versuche eines Verbindungsaufbaus vom server gebuffert werden. Der server kann keine neue Verbindung behandeln, solange er eine weitere behandelt. Wenn der buffer voll ist wird die Fehlermeldung errno=ECONNREFUSED an den client gemeldet. Der erste Parameter ist der fd aus socket, der zweite die Grösse des buffers= Anzahl der requests. Hier eine Implementierung von u_open:

```
int u_open(u_port_t port)
{
int sock;
struct sockaddr_in server;
if ( (u_ignore_sigpipe() != 0) ||
      ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) )
  return -1;
server.sin_family = AF_INET;
server.sin_addr.s_addr = INADDR_ANY;
server.sin_port = htons((short)port);
if ( (bind(sock, (struct sockaddr *)&server,
sizeof(server)) < 0) ||
       (listen(sock, MAXBACKLOG) < 0) )
  return -1;
return sock;
}
```

htons (host to networkshort) sorgt dafür, dass eventuell byte order Probleme berücksichtigt werden. u_ignore_sigpipe dient dazu das SIG_PIPE abzufangen, welches erzeugt wird, wenn in ein socket(pipe) ohne Lese-ende geschrieben wird. Besser ist es hier errno=EPIPE zu setzen.

6.4.1 u_listen

Dies ist die Funktion, die auf Server Seite wartet, zur Implementierung mit sockets verwendet man die Funktion accept:

```
NAME
accept - accept a connection on a socket
SYNOPSIS
#include <sys/types.h>
#include <sys/socket.h>
int accept(int s, struct sockaddr *addr, int *addrlen);
```

Wenn accept erfolgreich war enthält addr Information über den angeschlossenen Client. Der Rückgabewert ist ein fd um mit dem Client zu kommunizieren. Um den Namen zu bekommen verwendet man gethostbyaddr:

```
NAME
gethostbyadd - get network host entry
SYNOPSIS
#include <netdb.h>
struct hostent *gethostbyaddr(const char *addr, int len,
           int type);
```

Die zurück gegebene Struktur enthält eine Komponente h_name mit dem Namen. Nun kann u_listen implementiert werden:

```
int u_listen(int fd, char *hostn)
{
struct sockaddr_in net_client;
int len = sizeof(struct sockaddr);
int retval;
struct hostent *hostptr;
while ( ((retval =
  accept(fd, (struct sockaddr *)(&net_client), &len)) ==
-1) &&
  (errno == EINTR) )
;
if (retval == -1)
  return retval;
hostptr =
gethostbyaddr((char *)&(net_client.sin_addr.s_addr), 4,
AF_INET);
if (hostptr == NULL)
  strcpy(hostn, "unknown");
else
  strcpy(hostn, (*hostptr).h_name);
return retval;
}
```

6.4.2 u_connect

Die Implementierung des Client codes ist einfacher. Mit dem Aufruf von connect:

```
NAME
connect - Verbindungsaufbau zu einem "Socket"
ÜBERSICHT
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, struct sockaddr *serv_addr, int
          addrlen );
```

Die Parameter sind wie bei bind, sockfd hat man durch den Aufruf von socket bekommen. Der Rückgabewert ist 0, falls alles OK ist, dann kann sockfd als normaler filedescriptor für read/write verwendet werden. Da bei connect die IP Adresse verwendet werden muss, wird der Name mit dem Aufruf gethostbyname umgewandelt.

```
NAME
gethostbyname - get network host entry
SYNOPSIS
#include <netdb.h>
struct hostent *gethostbyname(const char *name);
```

Der Rückgabewert enthält die Komonente h_addr_list mit eventuell mehreren IP Adressen, es wird die erste verwendet:

```
int u_connect(u_port_t port, char *hostn)
{
struct sockaddr_in server;
struct hostent *hp;
int sock;
int retval;
if ( (u_ignore_sigpipe() != 0) ||
    !(hp = gethostbyname(hostn)) ||
    ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
   )
   return -1;

memcpy((char *)&server.sin_addr, hp->h_addr_list[0],
hp->h_length);
server.sin_port = htons((short)port);

server.sin_family = AF_INET;
while ( ((retval =
connect(sock, (struct sockaddr *)&server,
sizeof(server))) == -1)
&& (errno == EINTR) )
;
if (retval == -1) {
         close(sock);
         return -1;
         }
return sock;
}
```