Netzwerkprogrammierung mit Sockets und C

Ulrich Vogel

Netzwerkprogrammierung mit Sockets und C

von Ulrich Vogel

Dieses Tutorial ist eine Einführung in die Netzwerkprogrammierung mit Hilfe von Sockets. Dabei wird auf Aspekte der Client- sowie der Serverprogrammierung eingegangen. Es empfiehlt sich der Einsatz eines Unix-Systems mit GNU C Compiler, jedoch kann auch Windows verwendet werden.

Versionsgeschichte

Version 1 03.10.2002
Die erste Version

Inhaltsverzeichnis

Beispiele

Kapitel 1. Vorwort

Dieses Tutorial ist eine Einführung in die Netzwerkprogrammierung mit Hilfe von Sockets und der Programmiersprache C. Als Vorwissen genügen C-Grundlagen, außerdem sollten Begriffe wie IP-Adresse, Port oder Server nicht völlig fremd sein.

Zum Aufbau dieses Dokuments: Ich habe es im wesentlichen eingeteilt in Client- und Server-Programmierung. Das heißt jedoch nicht, daß die beiden Teile unabhängig voneinander sind, vielmehr baut das Server- auf dem Client-Kapitel auf. Es empfiehlt sich daher, dieses Tutorial von vorne bis hinten zu lesen und nicht zu springen. Als Referenz ist es nicht geeignet.

Wer tiefer in die Materie einsteigen möchte, sollte sich im Netz nach weiteren Quellen umschauen, wovon es genug gibt. Ein sehr gutes englischsprachiges Tutorial ist Beej’s Guide to Network Programming(http://www.ecst.csuchico.edu/~beej/guide/net/). Was die verschiedenen Funktionen betrifft, um die es in diesem Text geht, lohnt sich ein Blick in die Man-Pages, die auf jedem Linux-Rechner installiert sein sollten. Ansonsten gibt es sie auch im Netz, z.B. unter http://www.rt.com/man/.

Kapitel 2. Hinweise zu den Codebeispielen

Die angegebenen Beispiele habe ich mit dem GNU C Compiler unter Linux getestet. Prinzipiell funktionieren sie aber auch unter Windows, und zwar mit der Winsock-Bibliothek. Abgesehen von den verschiedenen Header-Dateien, die eingebunden werden müssen, muß hierfür die `winsock32.lib` oder eine ähnlich lautende Datei mitgelinkt werden. Im Bloodshed Dev-C++ beispielsweise wählt man unter Project->Project options->Load object files die `libwsock32.a` aus dem Unterverzeichnis `lib` aus.

Möglicherweise sind nicht alle Header-Dateien, die ich hier angebe, für jedes Programm notwendig. In den Man-Pages steht, welche Funktion welche Bibliothek benötigt.

Beispiel 2-1. Einbinden der Header-Dateien

```
#include <stdio.h>
#include <errno.h>
#include <string>

/* Windows-System */
#ifdef _WIN32
#include <winsock.h>

/* Unix-System */
#else
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#endif
```

Außerdem muß unter Windows die Socket-Schnittstelle erst initialisiert werden, bevor man darauf zugreift. Vor dem Programmende sollte dann noch `WSACleanup()` aufgerufen werden.

Beispiel 2-2. Winsock initialisieren

```
#ifdef _WIN32
WSADATA wsaData;
if (WSAStartup (MAKEWORD(1, 1), &wsaData) != 0) {
    fprintf (stderr, "WSAStartup(): Kann Winsock nicht initialisieren.\n");
    exit (EXIT_FAILURE);
}
#endif

/* ... */

#ifdef _WIN32
WSACleanup();
#endif
```

Kapitel 3. Hintergrundwissen

3.1. TCP und UDP

Es gibt zwei wichtige Transportprotokolle, die auf dem IP-Protokoll aufsetzen: TCP (Transmission Control Protocol) und UDP (User Datagram Protocol). Beide befinden sich auf der Transport-Ebene des ISO/OSI-Schichtenmodells. TCP garantiert eine zuverlässige Datenübertragung, indem es sicherstellt, daß Datenpakete, die nach einer gewissen Zeit (Timeout) den Empfänger nicht erreicht haben, erneut geschickt werden. Kommen Pakete in einer anderen Reihenfolge an, als in derjenigen, in der sie losgeschickt wurden, sorgt TCP dafür, daß das empfangende Anwendungsprogramm sie dennoch in der korrekten Reihenfolge erhält. TCP ist ein verbindungsorientiertes Protokoll, d.h., es wird mit einem "Handshake" eine logische Verbindung zwischen den Partnern aufgebaut und nach der Übertragung wieder abgebaut. Die Datenübertragung wird als ein Datenstrom gesehen. UDP bietet diese Sicherheitsmechanismen nicht. Es ist unzuverlässig und verbindungslos. Es sieht die Datenübertragung als das Senden von Einzelpaketen ("Datagramme").

Da stellt sich natürlich angesichts der Nachteile von UDP die Frage, wieso nicht immer TCP eingesetzt wird. Der Grund ist, daß mit UDP durch den geringen Verwaltungsaufwand und den geringeren Overhead der Datenpakete ein höherer Datendurchsatz erreicht werden kann, der mit TCP gar nicht möglich ist. So bietet sich UDP z.B. für die Videoübertragung an. Dort ist es im übrigen nicht weiter tragisch, wenn gelegentlich Datenpakete - also einzelne Frames - verlorengehen. Selbstverständlich ist es auch möglich, auf der Anwendungsebene bestimmte Sicherungsmaßnahmen zu implementieren, die für die jeweilige Anwendung ausreichen.

3.2. Sockets

Was sind Sockets? Ein Socket ist eine Schnittstelle zwischen einem Programm - genauer: einem Prozeß - und einem Transportprotokoll. Letzteres kann z.B. TCP oder UDP sein. Das Socket-Prinzip entspricht dem von File-Deskriptoren. Dort repräsentiert nach dem Öffnen einer Datei ein Handle die Verbindung zu dieser Datei und unter Angabe des Handles ist der Lese- oder Schreibzugriff möglich. Bei Sockets geht es jedoch nicht um physikalische Dateien sondern um Kommunikationskanäle, über die Daten gesendet und empfangen werden können.

Kapitel 4. Client-Programmierung

Bevor wir loslegen, müssen wir zunächst unterscheiden, was in diesem Zusammenhang zwischen einem Client und einem Server zu verstehen ist. Auf der TCP/IP-Ebene zeichnet sich ein Client dadurch aus, daß er zu einem anderen Rechner eine Verbindung herstellt, also den Verbindungsaufbau initiiert. Ein Server auf der Gegenseite nimmt Verbindungsanfragen entgegen. Ist eine Verbindung zwischen Client und Server erst einmal hergestellt, entfällt diese Unterscheidung. Beide können gleichberechtigt Daten senden und empfangen sowie die Verbindung beenden. Auf der Anwendungsebene, einer höheren Abstraktionsebene, kann selbstverständlich weiterhin zwischen einem Client, der Dienste in Anspruch nimmt, und einem Server, der Dienste bereitstellt, differenziert werden. Für unseren Problembereich ist das jedoch nicht von Bedeutung.

In diesem Kapitel soll es nun darum gehen, einen Client zu schreiben. Ich werde hier die einzelnen Funktionen vorstellen, die dazu nötig sind. Ein einfacher HTTP-Client findet sich hierzu als vollständiges Beispielprogramm im Anhang.

4.1. socket() - Einen Socket anfordern

Egal ob Client oder Server - der Aufbau einer Kommunikationsverbindung beginnt stets damit, einen Socket vom Betriebssystem anzufordern. Dies geschieht über die Funktion `socket()` (**man**(http://www.rt.com/man/socket.2.html)). Hier spielt es noch keine Rolle, mit wem wir kommunizieren möchten, entscheidend ist wie.

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

Der Parameter *`domain`* gibt die Adressfamilie an, die wir verwenden wollen. Einige Protokolle, die unterstützt werden, habe ich hier aufgeführt. Die vollständige Liste befindet sich (auf meinem System) in der Datei `/usr/include/linux/socket.h`. Wir verwenden für unsere Zwecke `AF_INET`, das Internet Protocol in der Version 4.

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

`type` ist der nächste Parameter. Er gibt die Übertragungsart an, wobei für uns zwei mögliche Belegungen relevant sind: `SOCK_STREAM` - die Übertragung eines Datenstroms - und `SOCK_DGRAM` - Übertragung von Datagrammen.

Der Wert `0` für den Parameter *`protocol`* weist `socket()` an, das zur Übertragungsart standardmäßig verwendete Protokoll auszuwählen. Im Fall von `SOCK_STREAM` ist das TCP, bei `SOCK_DGRAM` ist es UDP.

`socket()` gibt uns die Nummer zurück, die den bereitgestellten Socket identifiziert und die wir in unseren weiteren Funktionsaufrufen verwenden werden - den Socket-Deskriptor. Falls kein Socket bereitgestellt werden kann, gibt `socket() -1` zurück. Der Aufruf sieht also mit Fehlerabfrage für eine Verbindung über TCP/IP folgendermaßen aus:

Beispiel 4-1. Socket anfordern

```
int sockfd = socket (AF_INET, SOCK_STREAM, 0);
if (sockfd == -1) {
    perror ("socket()");
}
```

4.2. connect() - Eine Verbindung aufbauen

Nachdem uns das System freundlicherweise einen Socket bereitgestellt hat, können wir denselben verwenden, um darüber eine Verbindung herzustellen. Hier trennen sich nun die Wege in der Programmierung von Client und Server. Als Client werden wir nun einen anderen Rechner, der über das Netzwerk erreichbar ist, kontaktieren. Dazu dient der Befehl `connect()` (**man**(http://www.rt.com/man/connect.2.html)).

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

`sockfd` ist der Socket, über den die Verbindung hergestellt werden soll. Hier geben wir den Wert an, den wir von `socket()` erhalten haben. *`serv_addr`* ist ein Zeiger auf eine Struktur, welche die Adressinformation zu demjenigen Prozeß bereitstellt, den wir kontaktieren möchten. `connect()` erwartet eine Struktur vom Typ sockaddr. Die ist jedoch so gehalten, daß sie unabhängig von der verwendeten Adreßfamilie und deswegen im konkreten Fall von Hand nur umständlich auszufüllen ist.

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

Für IP-Anwendungen gibt es daher eine spezielle Struktur, sockaddr_in, die es ermöglicht, die IP-Adresse sowie die Portnummer getrennt einzutragen. Im Speicher sind diese beiden Strukturen kompatibel, es reicht also eine einfache Typumwandlung, um `connect()` die gewünschten Informationen zu übergeben.

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

```
};
```

`addrlen` ist ganz einfach die Größe von sockaddr, also `sizeof(sockaddr)`.

Das Ausfüllen von sockaddr_in ist nicht ganz so trivial, wie es auf den ersten Blick vielleicht scheint. Der Hintergrund ist der, daß in einem heterogenen Netz Rechner verschiedener Architekturen aufeinandertreffen. Ein wesentliches Unterscheidungsmerkmal verbirgt sich hinter dem Begriff "Byteorder". Die Byteorder gibt an, wie Zahlenwerte im Speicher repräsentiert werden, genauer: welches Byte welche Wertigkeit besitzt. Bei der "Big-Endian" Byteorder ist das jeweils letzte Byte dasjenige mit dem höchsten Wert, bei "Little-Endian" entsprechend umgekehrt. Die Zahl 255 hätte (als short int) im ersten Fall in hexadezimaler Schreibweise die Darstellung `FF 00`, im zweiten Fall `00 FF`. Um dennoch zu gewährleisten, daß sich die Rechner untereinander "verstehen", hat man sich auf eine einheitliche Form für die Datenübertragung geeinigt - die Network Byteorder. Die ist übrigens Big-Endian.

Um Zahlen der Typen short int bzw. long int aus der Byteorder des eigenen Systems ("Host Byteorder") in Network Byteorder und umgekehrt zu konvertieren, werden 4 Funktionen bereitgestellt:

```
/* short integer from host to network byte order */
unsigned short int htons(unsigned short int hostshort);

/* long integer from host to network byte order */
unsigned long int htonl(unsigned long int hostlong);

/* short integer from network to host byte order */
unsigned short int ntohs(unsigned short int netshort);

/* long integer from network to host byte order */
unsigned long int ntohl(unsigned long int netlong);
```

So, jetzt aber genug der Theorie. Als erstes brauchen wir eine Variable vom Typ sockaddr_in, in die wir die Adressinformation über den gewünschten Verbindungspartner eintragen.

`sin_family` erhält den Wert `AF_INET`, den wir auch an `socket()` übergeben haben.

Die Portnummer - *`sin_port`* - können wir direkt angeben, z.B. `80` um einen Webserver zu erreichen, wobei wir aber nicht vergessen dürfen, den Wert in Network Byteorder anzugeben.

Auch die IP-Adresse *`sin_addr`* ist in Network Byteorder anzugeben. Hier hilft uns die Funktion `inet_addr()` (**man**(http://www.rt.com/man/inet_addr.3.html)), der wir die Adresse als Zeichenkette übergeben können.

Beispiel 4-2. Verbindungspartner festlegen

```
sockaddr_in serv_addr;
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(80);
```

```
serv_addr.sin_addr.s_addr = inet_addr("195.227.67.232");
```

Möglicherweise ist es eleganter, die Portnummer über die Funktion `getservbyname()` (**man**(http://www.rt.com/man/getservbyname.3.html)) zu ermitteln. Dabei wird in der Datei `/etc/services` (**man**(http://www.rt.com/man/services.5.html)) nach einem Eintrag für den übergebenen Namen eines Dienstes (http, ftp, pop3 etc) und eines Transportprotokolls (tcp oder udp) gesucht.

Beispiel 4-3. Portnummer des Dienstes ermitteln

```
servent* serviceinfo = getservbyname ("http", "tcp");
serv_addr.sin_port = serviceinfo->s_port;
```

Die Struktur ist nun mit allen Werten belegt - *`pad[8]`* dient lediglich dazu, sockaddr_in auf die Größe von sockaddr aufzufüllen - und wir können `connect()` aufrufen. Im Fehlerfall wird `-1` zurückgegeben, was in jedem Fall zu überprüfen ist. Ansonsten steht unsere Verbindung und wir können Daten senden und empfangen! (Korrekterweise darf man bei UDP nicht sagen, daß die Verbindung "steht", da UDP ein verbindungsloses Protokoll ist. Hier legt `connect()` lediglich den Verbindungspartner fest, ohne einen Handshake durchzuführen.)

Beispiel 4-4. Verbindung aufbauen

```
if (connect(sockfd, (sockaddr *) &serv_addr, sizeof(sockaddr)) == -1) {
    perror ("connect()");
}
```

4.3. send() und recv() - Senden und Empfangen mit TCP

Zum Datenaustausch gibt es für TCP und UDP jeweils ein Funktionspäärchen: Dies ist bei TCP `send()` (**man**(http://www.rt.com/man/send.2.html)) und `recv()` (**man**(http://www.rt.com/man/recv.2.html)). Zunächst zum Senden von Daten:

```
int send(int s, const void *msg, int len, unsigned int flags);
```

Der erste Parameter *`s`* bezeichnet - wie gehabt - unseren Socket. *`msg`* zeigt auf den Speicherbereich, in dem sich die Daten befinden, die wir senden wollen. *`len`* gibt die Größe dieses Speicherbereichs an. *`flags`* können wir für unsere Zwecke auf `0` setzen.

Einfach, oder? Einen Haken hat die Sache aber dann doch. Wir können uns nämlich nicht darauf verlassen, daß `send()` alle Daten auf einmal losschickt, sondern nur einen Teil davon. Daß auch der Rest geschickt wird, darum müssen wir uns selbst kümmern. Wir müssen daher den Rückgabewert von `send()` - die Anzahl der gesendeten Zeichen - mit dem Parameter *`len`* vergleichen. Ergibt sich daraus, daß ein Teil noch nicht gesendet wurde, müssen wir `send()` erneut für diesen Speicherbereich aufrufen. Falls `send()` `-1` zurückliefert, ist irgendetwas schiefgelaufen.

Beispiel 4-5. Daten senden

```
char *msg = "GET / HTTP/1.0\n\n";
int len = strlen (msg);
if (send (sockfd, msg, len, 0) == -1) {
    perror ("send()");
}
```

Die Funktion `recv()` zum Empfangen von Daten ist `send()` vom Aufbau sehr ähnlich.

```
int recv(int s, void *buf, int len, unsigned int flags);
```

Bevor wir `recv()` aufrufen, müssen wir einen Speicherbereich bereitstellen, in welchen die Funktion die empfangenen Daten hineinschreiben kann. Dies geschieht sehr einfach über ein Char-Array. Dessen Adresse geben wir in *`buf`* an, die Länge des reservierten Bereichs in *`len`*. *`flags`* belassen wir wieder bei `0`, *`s`* bezeichnet natürlich unseren Socket.

Analog zu `send()` gibt `recv()` die Anzahl der empfangenen Bytes zurück. Wird `0` zurückgegeben, so bedeutet das, daß der Verbindungspartner die Verbindung beendet hat, `-1` weist auf einen aufgetretenen Fehler hin.

Beispiel 4-6. Daten empfangen

```
char buf[1024];
if (recv (sock, buf, 1024, 0) == -1) {
    perror ("recv()");
}
```

4.4. sendto() und recvfrom() - Senden und Empfangen mit UDP

Die UDP-Pendants zu `send()` und `recv()` heißen `sendto()` (**man**(http://www.rt.com/man/sendto.2.html))

```
int sendto(int s, const void *msg, int len, unsigned int flags, const struct sockaddr *to,
```

und `recvfrom()` (**man**(http://www.rt.com/man/recvfrom.2.html)).

```
int recvfrom(int s, void *buf, int len, unsigned int flags struct sockaddr *from, int *from
```

Gegenüber den bereits bekannten Funktionen kommen jeweils zwei Parameter hinzu. Da UDP ein verbindungsloses Protokoll ist, muß jedesmal die Adresse des Verbindungsparters angegeben werden. *`to`* bzw. *`from`* entsprechen dem Parameter *`serv_addr`* von `connect()`, *`tolen`* bzw. *`fromlen`* erhalten wieder einmal die Größe der Struktur sockaddr.

Wenn wir UDP verwenden, können wir übrigens auf den Aufruf von `connect()` verzichten. Rufen wir `connect()` dennoch auf, können wir alternativ wieder - wie bei TCP - `send()` und `recv()` benutzen. Die fehlenden Adressierungsinformationen werden dann automatisch ergänzt.

4.5. close() - Socket freigeben

Genauso wie man eine gewöhnliche Datei nach dem Lesen oder Schreiben schließt, schließt man auch einen Socket - mit `close()` (**man**(http://www.rt.com/man/close.2.html)). Handelt es sich dabei um eine TCP-Verbindung, wird diese beendet. Unter Windows wird `closesocket()` anstelle von `close()` verwendet.

```
int close(int fd);
```

Kapitel 5. Server-Programmierung

Seitenwechsel. Im letzten Kapitel haben wir eine Verbindung zu einem bereits existierenden Server aufgebaut, jetzt basteln wir unser eigenes Serverprogramm. Der wesentliche Unterschied zur Programmierung eines Clients liegt lediglich in der Art, wie eine Verbindung zustande kommt. Da es Sache des Clients ist, die Verbindung zu initiieren, begeben wir uns hier in einen Wartezustand und nehmen die eingehenden Verbindungswünsche entgegen. Die Datenübertragung selbst funktioniert jedoch aus Serversicht genauso wie im Client-Kapitel beschrieben.

5.1. bind() - Den eigenen Port festlegen

Will jemand unseren Server kontaktieren, so muß er die IP-Adresse unseres Rechners sowie die Portnummer kennen, unter welcher der Serverprozeß erreichbar ist. Das setzt in der Regel voraus, daß der Server nicht auf einem willkürlich gewählten Port auf Anfragen wartet, sondern daß dieser Port immer derselbe ist. Wir müssen daher dem Betriebssystem beim Programmstart mittels `bind()` (**man**(http://www.rt.com/man/bind.2.html)) mitteilen, welchen Socket es mit einem bestimmten Port assoziieren soll. Geht ein Datenpaket ein, kann das System anhand der Zielportnummer feststellen, für welchen Socket das Paket gedacht ist.

Damit keine Mißverständnisse aufkommen: natürlich besitzt ein Socket auch dann einen Port, wenn wir - wie im letzten Kapitel - mit `connect()` selbst eine Verbindung aufbauen. In diesem Fall ist es jedoch nicht zwingend notwendig, `bind()` aufzurufen, da automatisch ein freier Port gewählt wird. Welcher dies ist, ist meistens egal.

Genug der Worte, so sieht `bind()` aus:

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

`sockfd` ist - was nicht weiter überraschen dürfte - der Socket, den wir zuvor mit `socket()` angefordert haben, um darüber Verbindungsanfragen entgegenzunehmen. *`my_addr`* teilt dem System mit, welche Datenpakete für diesen Socket bestimmt sind. Hier werden IP-Adresse und Portnummer eingetragen. *`addrlen`* setzen wir auf `sizeof(sockaddr);`

Die Struktur sockaddr bzw. sockaddr_in des Parameters *`my_addr`* ist bereits bekannt, dennoch ein paar Bemerkungen:

Beim Eintragen der IP-Adresse gilt es zu beachten, daß ein Rechner über verschiedene Netze und somit unter mehreren Adressen erreichbar sein kann, z.B. über das Internet und ein lokales Netzwerk. Wenn der Server über alle Netze Verbindungen aufnehmen soll, gibt man hier `INADDR_ANY` (in Network Byte Order) an. Andernfalls die entsprechende IP mit Hilfe von `inet_addr()`.

Beispiel 5-1. Eigenen Port festlegen

```
sockaddr_in my_addr;
my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(5000);
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);

if (bind(sock, (sockaddr *)&my_addr, sizeof(sockaddr)) == -1) {
    perror ("bind()");
}
```

In der Regel werden wir eine bestimmte Portadresse angeben wollen. Trotzdem ist es auch möglich, dem System zu überlassen, uns einen beliebigen, noch nicht belegten Port zuzuweisen. In diesem Fall geben wir für *`sin_port`* `0` an (korrekterweise in Network Byte Order, also `htons(0)`). Da wir in diesem nicht Fall nicht von vornherein wissen, welchen Port wir erhalten, ist es notwendig, dies nach dem Aufruf von `bind()` abzufragen - mit `getsockname()` (**man**(http://www.rt.com/man/getsockname.2.html)):

Beispiel 5-2. Zugewiesenen Port ermitteln

```
socklen_t len;
getsockname(sock, (sockaddr *) &my_addr, &len);
printf ("Port: %d\n", ntohs(my_addr.sin_port));
```

5.2. listen() - Auf eingehende Verbindungen warten

Nun ist alles soweit eingerichtet, daß wir nur noch darauf warten müssen, daß uns ein Client kontaktiert. `listen()` (**man**(http://www.rt.com/man/listen.2.html)) unterbricht die Programmausführung solange, bis eben dies eintritt.

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

`s` sollte klar sein, *`backlog`* gibt an, wieviele Verbindungsanfragen maximal in eine Warteschlange gestellt werden können, wenn gerade keine Verbindungen angenommen werden können.

Beispiel 5-3. Auf eingehende Verbindungen warten

```
if (listen (sock, 5) == -1) {
    perror ("listen()");
}
```

5.3. accept() - Die eingehende Verbindung annehmen

Während wir also mit `listen()` auf eine Verbindung warten, versucht hoffentlich jemand anders, eine solche zu uns aufzunehmen. Diesem Wunsch kommen wir mit `accept()` (**man**(http://www.rt.com/man/accept.2.html)) nach.

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

Damit wir wissen, mit wem wir es da eigentlich zu tun haben, wenn wir die Verbindung akzeptieren, müssen wir eine Struktur sockaddr_in bereitstellen, in die `accept()` Informationen über den Verbindungspartner - IP-Adresse und Port - hineinschreiben kann. *`addr`* ist ein Zeiger auf diese. *`addrlen`* ist wieder die Größe der Struktur sockaddr_in - aber Vorsicht: diesmal wird ein Zeiger erwartet.

Interessant ist der Rückgabewert der Funktion. `accept()` fordert nämlich einen neuen Socket an, über den die Datenübertragung dieser neuen Verbindung abgewickelt wird. Den Deskriptor dieses Sockets erhalten wir zurück. Über den "alten" Socket können dagegen weiterhin Verbindungswünsche entgegengenommen werden.

Beispiel 5-4. Eingehende Verbindung annehmen

```
socklen_t sin_size = sizeof (sockaddr_in);
sockaddr_in remote_host;
int sock2 = accept (sock, (sockaddr *) &remote_host, &sin_size);
if (sock2 == -1) {
    perror ("accept()");
}
```

Nun können wir über die neue Verbindung zum Client wie gehabt mit `send()` und `recv()` bzw. `sendto()` und `recvfrom()` Daten übertragen.

Kapitel 6. Beispiele

Zunächst nur ein ganz simpler HTTP-Client, der als Parameter den Webserver und eine Datei erwartet und letztere auf dem Bildschirm ausgibt.

6.1. Ein HTTP-Client

Beispiel 6-1. HTTP-Client

```
#include <stdio.h>
#include <errno.h>
#include <string>

/* Windows-System */
#ifdef _WIN32
#include <winsock.h>
#include <io.h>
/* Unix-System */
#else
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#endif

#define HTTP_PORT 80

int main(int argc, char **argv)
{
    int sock;
    struct sockaddr_in host_addr;
    struct hostent *hostinfo;
    char *host, *file;
    char command[1024];
    char buf[1024];
    unsigned int bytes_sent, bytes_recv;

    /* Ist der Aufruf korrekt? */
    if (argc != 3) {
        fprintf (stderr, "Aufruf: httprecv host file\n");
        exit (EXIT_FAILURE);
    }

    host = argv[1];
    file  = argv[2];

    /* ggf. Winsock initialisieren */
```

```
#ifdef _WIN32
WSADATA wsaData;
if (WSAStartup (MAKEWORD(1, 1), &wsaData) != 0) {
    fprintf (stderr, "WSAStartup(): Kann Winsock nicht initialisieren.\n");
    exit (EXIT_FAILURE);
}
#endif

/* Socket erzeugen */
sock = socket (AF_INET, SOCK_STREAM, 0);
if (sock == -1) {
    perror ("socket()");
    exit (EXIT_FAILURE);
}

/* Adresse des Servers festlegen */
memset( &host_addr, 0, sizeof (host_addr));
host_addr.sin_family = AF_INET;
host_addr.sin_port = htons (HTTP_PORT);

host_addr.sin_addr.s_addr = inet_addr (host);
if (host_addr.sin_addr.s_addr == INADDR_NONE) {
    /* Server wurde nicht mit IP sondern mit dem Namen angegeben */
    hostinfo = gethostbyname (host);
    if (hostinfo == NULL) {
        perror ("gethostbyname()");
        exit (EXIT_FAILURE);
    }
    memcpy((char*) &host_addr.sin_addr.s_addr, hostinfo->h_addr, hostinfo->h_length);
}

/* Verbindung aufbauen */
if (connect(sock, (struct sockaddr *) &host_addr, sizeof(struct sockaddr)) == -1) {
    perror ("connect()");
    exit (EXIT_FAILURE);
}

/* HTTP-GET-Befehl erzeugen */
sprintf (command, "GET %s HTTP/1.0\nHost: %s\n\n", file, host);

/* Befehl senden */
bytes_sent = send (sock, command, strlen (command), 0);
if (bytes_sent == -1) {
    perror ("send()");
    exit (EXIT_FAILURE);
}

// Antwort des Servers empfangen und ausgeben */
while ((bytes_recv = recv (sock, buf, sizeof(buf), 0)) > 0) {
    write (1, buf, bytes_recv);
}
if (bytes_recv == -1) {
    perror ("recv()");
```

```
            exit (EXIT_FAILURE);
        }

        printf ("\n");

        #ifdef _WIN32
        closesocket(sock);
        WSACleanup();
        #else
        close(sock);
        #endif

        return 0;

    }
```

Kapitel 7. In eigener Sache

Ich übernehme natürlich keine Garantie für die Richtigkeit der Informationen auf diesen Seiten. Aber das meiste dürfte wohl stimmen... Ich hoffe, daß ich das Tutorial von Zeit zu Zeit verbessern und erweitern kann.

Über (konstruktive) Kritik, Verbesserungsvorschläge, Korrekturen etc. würde ich mich freuen. Solches bitte an <mail@ulrich-vogel.de> schicken.

Dieses Dokument befindet sich im Original unter
http://www.ulrich-vogel.de/programmierung/sockdoc.html.