Hawkeyes Tipps zur Socket-Programmierung

[PDF-Version]

Ich hielt es mal für nötig ein paar Tipps für alle die loszulassen, die mittels Sockets, sei es unter Linux oder Solaris, unter Windows 9x oder Windows NT/2000, Server und Clients schreiben wollen. Da dies mein erster Versuch einer Sammlung von Tipps zur Programmierung ist, bitte ich um rege Beteiligung am Feedback damit ich in Zukunft weiterhin den Geschmack meiner Leser treffe ;-)

0. Inhalt

Ich denke ich sollte zuerst einen Überblick geben, welche Themen auf dieser Seite behandelt werden sollen. Ich weiss noch nicht, ob mir so knackige Titel einfallen, dass jeder weiterlesen wird, aber ich kann es nur empfehlen :->

1. Voraussetzungen
2. die Grundbefehle
3. Buffer und warum manchmal Schrott drinsteht
4. sprintf() und andere ANSI-Freunde
5. Grundstruktur eines Clients
6. Grundstruktur eines Servers
7. Tricks mit select()
8. verkettete Listen für sparsame Aufgaben
9. mehrere Prozesse für anstrengede Aufgaben (derzeit nur für Unix)
10. abschliessende Worte und eigener Senf für die Welt ;-)

Ich hoffe, dass dieser Inhalt den einen oder anderen dazu bewegen weiterzulesen - wenn nicht ist auch nicht schlimm, jeder hat das Recht dumm zu sterben *lol*

1. Voraussetzungen

Die Voraussetzungen sind ganz einfach: ein Betriebssystem, das Netzwerkprogramme mit Sockets unterstützt, eine Programmiersprache dies Sockets unterstützt (und die man logischerweise einigermassen beherrschen sollte) sowie etwas Geduld am Anfang und wirkliches Interesse an diesem Teilbereich der Programmierung.

Zu den **Betriebssystemen**:
Unterstützt werden Sockets unter Linux, fast allen neueren Unixen (= nach 1980 :->), Windows 95, 98 sowie Windows NT und Windows 2000. Windows Millenium und die ganzen Teile die noch kommen sollen (und natürlich vieeeel besser sein werden als alles dagewesene ...) können auch nicht darauf verzichten. Windows 3.x jedoch bleibt von Haus aus aussen vor, da TCP/IP nicht unterstützt wird (jedenfalls nicht ohne Zusatzprogramme). Wie es mit OS/2 oder dem Mac steht weiss ich nicht und bin für Informationen hierzu jederzeit dankbar.

Die **Sprache**, mit der man nun die Sockets programmieren will, ist natürlich auch von enormer Wichtigkeit. Mit Basic wird es vermutlich nicht klappen, anders ist es mit allen C-Abkömmlingen. Hierauf ist auch die Unterstützung durch die API abgestimmt. Unter Windows stehen Delphi ebenfalls Sockets zur Verfügung, jedoch werde ich dazu nichts weiteres sagen (wenn die Standard API greift kann man natürlich auch Delphi nehmen, doch wenn das in irgendwelchen nervtötenden Objekte gekapselt ist ... viel Glück ;-). Ich persönlich ziehe C vor (siehe dazu (1) im Anhang), doch kann ich mich (ja, es kostet Überwindung dies zuzugeben) auch mit Visual C++ anfreunden (hab ich das wirklich gesagt!?). Unter Unix ist C natürlich prädestiniert dafür, C++ soll natürlich auch recht sein. Falls noch jemand mit DOS das lesen sollte: wie zum Henker bist Du an dieses Dokument gelangt? Hat es jemand für Dich gesaugt und in ASCII konvertiert ;-) ?

Zur **Geduld** sag ich jetzt nichts :->

Die **Verwendung der Sockets in C** respektive seiner Abkömmlinge ist verhältnismässig einfach.

Unter **Windows** muss die Header-Datei winsock.h eingebunden werden, sowie beim Compilerlauf die Bilbiothek wsock32.lib. Ausserdem müssen die Sockets (und das ist wichtig, weil sonst absolut nichts geht - ich werde im Folgenden auch nicht mehr darauf hinweisen, da es eine Windows-Spezialität ist und bei Unix nicht nötig ist) "angeschaltet" werden. Dies erledigt WSAStartup(). Am einfachsten macht man dies, indem man den folgenden Codeausschnitt einfügt:

```
/* initialize windows sockets */
{
        WSADATA wsa;
        if (WSAStartup(MAKEWORD(1, 1), &wsa))
        {
                printf("WSAStartup() failed, %lu\n", (unsigned long)GetLastError());
                return EXIT_FAILURE;
        }
}
```

wobei dies am Günstigsten gleich zu Beginn in main() erledigt wird, bevor es noch vergessen geht. Ausserdem verwendet Windows den Typ SOCKET statt int für Sockets sowie SOCKET_ERROR statt -1 bei Fehlern von socket(). Dies ist jedoch nur der Form halber, da int auch funktioniert ;-)

Unter **Unix** respektive Linux müssen je nach Verwendung mehrere Header-Dateien eingebunden werden. Die Sockets benötigen netdb.h, die Struktur sockaddr_in die häufig benötigt wird, findet sich in netinet/in.h. Zusätzliche Bibliotheken oder Befehle zur Initialisierung werden nicht benötigt.

2. die Grundbefehle

Die Grunbefehle die man benötigt sind leicht zu überblicken. Ich werde zuersteinmal die wichtigsten aufzählen und dann später genauer auf sie eingehen.

- socket()
- connect()
- bind()

- listen()
- accept()
- select()
- close()
- send()
- recv()
- htons()
- ntohs()
- htonl()
- ntohl()
- inet_addr()
- inet_aton()
- inet_ntoa()
- gethostbyname()
- gethostbyaddr()
- getservbyname()
- getservbyport()

nur für Unix:

- fcntl()

Hehe, sieht so aus als würde das ein langes Kapitel werden ...

socket()

Dieser Befehl lässt schon vermuten, dass er was mit Sockets zu tun hat ;-). Die Funktionsdeklaration ist

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

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

socket() erstellt einen neuen Socket der für eigene zwecke verwendet werden kann. Der Rückgabewert ist der Filedeskiptor (eine kleine nichtnegative Zahl) anhand dessen der Socket von nun an identifiziert werden kann. Falls kein Socket erstellt werden konnte, liefert socket() den Rückgabewert -1. Ausserdem wird die globale Variable errno gesetzt, die z.B. mit perror() ausgewertet werden kann. Ein häufiger Codeausschnitt, den man bei vielen Programmen antreffen wird, ist

```
s = socket(AF_INET, SOCK_STRAM, 0);
if (s == -1)
{
        perror("socket() failed");
        return 1;
}
```

und um nun nicht noch lange um den heissen Brei herumzureden kommen wir jetzt zu den Parametern:

int domain
Dieser Parameter gibt den Bereich an, für den dieser Socket verwendet werden soll. Die Familien sind (unter Unix) in <sys/socket.h> definiert. Gültige Werte sind

```
AF_UNIX
AF_INET
AF_ISO
AF_NS
AF_IMPLINK
```

wobei AF_INET die für uns wohl am interessante Familie bezeichnet: Die ARPA Internet protocols.

int type
Dieser Parameter bestimmt den Typ der Sockets und somit die Semantik der Kommunikation. Hier gibt es folgende gültige Werte:

```
SOCK_STREAM
SOCK_DGRAM
SOCK_RAW
SOCK_SEQPACKET
SOCK_RDM
```

Ich werde hier nur auf SOCK_STREAM eingehen, weil dieser Typ die Kommunikation mit verbindungsorietnierten TCP beschreibt. Die anderen benötigt man für andere Protokolle (z.B. UDP) oder wenn man die IP-Header manuell verändern will (SOCK_RAW). Im allgemeinen geben wir also als zweiten Parameter SOCK_STREAM an.

int protocol
Der dritte Paramter von socket() gibt das zu verwendende Protokoll an. Man kann dieses entweder explizit angeben, oder einfach mit 0 das Standard-Protokoll für diesen Socket-Typ verwenden. Wir ziehen letztere Methode vor.

Man sollte bedenken, dass die Zahl der Sockets nicht unbegrenzt (wenn auch hoch) ist. Nichtmehr benötigte Sockets sollten mit close() freigegeben werden (dies geschieht übrigens automatisch, wenn das Programm beendet wird).

connect()

Wie der Name schon vermuten lässt wird mit connect() eine Verbindung zu einem Server aufgebaut. Die Deklaration des Befehls ist

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

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

Connect liefert als Rückgabewert 0 wenn der Vorgang geklappt hat und -1 wenn die Verbindung fehlgeschlagen ist. Wie immer wird errno gesetzt und liefert nähere Informationen (wie z.B. "connection refused" falls kein Server gefunden wurde). Natürlich muss connect() wissen, mit welchem Server man sich verbinden möchte. Dies geschieht über die Parameter:

int sockfd
Dieser Parameter gibt den Socket an, der verwendet werden soll.

struct sockaddr *serv_addr
Dieser Parameter gibt die Informationen der Verbindung an, wie zum Beispiel die Zieladresse, der Port sowie die verwendete Socket-Familie. Die Struktur sockaddr_in, die hier Verwendung findet, ist wie folgt deklariert:

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

struct sockaddr_in {
    short int           sin_family;  /* AF_INET     */
    unsigned short int  sin_port;    /* Port-Nummer */
    struct in_addr      sin_addr;    /* IP-Adresse  */
};
```

int addrlen
Dieser Parameter ist die Länge (also Grösse der Struktur) der Adresse. Hier gibt man am besten den Wert direkt mit sizeof() an.

Ein somit häufig anzutreffender Codeausschnitt ist

```
int s;
struct sockaddr_in addr;
...                             /* s = socket (...); */
addr.sin_addr = ...             /* z.B. inet_addr("127.0.0.1"); */
addr.sin_port = ...             /* z.B. htons(80);              */
addr.sin_family = AF_INET;

if (connect(s, &addr, sizeof(addr)) == -1)
{
        perror("connect() failed");
        return 2;
}
```

Damit solte die Verwendung von connect() klar sein. Ansonsten einfach nachschlagen (2).

bind()

Mit bind() wird ein Socket mit einer lokalen Adresse verbunden. Dies findet bei Servern Anwendung. Bind() ist folgendermaßen deklariert:

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

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

Auch hier ist der Rückgabewert bei Erfolg 0 bzw. bei Fehlschlagen -1. Ebenfalls wird errno gesetzt und kann weitere Informationen liefern ("address already in use" zum Beispiel). Die Parameter der Funktion geben die Adresse an, mit der der Socket verbunden werden soll.

int sockfd
Dieser Parameter ist der zu verbindende Socket.

struct sockaddr *my_addr
Dieser Parameter gibt die Adresse an. Man verwendet hier die Struktur sockaddr_in (siehe connect()). Für den Wert sin_addr.s_addr gibt man bei einem Server in der Regel INADDR_ANY an, das dafür sorgt dass von jeder beliebigen Adresse eine Verbindung eingehen kann.

int addrlen
Hier ist wieder die Grösse der Struktur mit der Adresse gemeint, also sizeof(my_addr) in diesem Fall.

Zu diesem Befehl sieht das häufig auftauchende Codefragment so aus:

```
int s;
struct sockaddr_in addr;
...                             /* s = socket (...); */
addr.sin_addr = ...             /* z.B. inet_addr("127.0.0.1"); */
addr.sin_port = ...             /* z.B. htons(80);              */
addr.sin_family = AF_INET;

if (bind(s, &addr, sizeof(addr)) == -1)
{
        perror("connect() failed");
        return 2;
}
```

listen()

Dieser Befehl versetzt den Socket in den Lausch-Modus, so dass sich ein Client mit ihm verbinden kann. Dies ist eine Funktion, die von einem Server verwendet wird (wer hätte das gedacht ;-). Die Deklaration des Befehls listen() sieht folgendermaßen aus:

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

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

Bei dieser Funktion ist der Rückgabewert ebenfalls 0 bei Erfolg und -1 bei Misserfolg, errno wird auch gesetzt und liefert weitere Informationen.

int s
Dies ist der Socket der in den Lausch-Modus versetzt werden soll.

int backlog
Dieser Parameter gibt die maximale Anzahl der Verbindungen, die in der Warteschlagen gehalten werden sollen. Ist die Warteschlange voll (weil die Clients nicht mit accpet() abgeholt werden), so wird der Fehler "connection refused" an den Client zurückgegeben. Wenn man portable Programme schreiben will, sollte man den Wert 5 nicht überschreiten, in der Regel gibt man 3 an (scheint sich bewährt zu haben).

Unser typisches Codefragment sieht bei bind() ganz einfach aus:

```
if (bind(s, 3) == -1)
{
        perror("bind () failed");
        return 3;
}
```

accept()

Dieser Befehle ist für Server wichtig: er holt die wartenden Clients die sich verbinden wollen aus der Warteschlange ab. Der Befehl ist wie folgt deklariert:

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

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

Der Rückgabewert ist diesmal zwar -1 bei einem Fehler, bei Erfolg jedoch der neue Socket, der den Client beschreibt. Dies ist besonders wichtig, weil der Socket s in unserer Deklaration weiterhin für eingehende Verbindungen zur Verfügung steht. Die Parameter fangen hier die Informationen des Clients auf:

int s
Dies ist der Socket auf dem die Verbindungen eingehen.

struct sockaddr *addr
In diese Struktur vom Typ sockaddr_in werden die Daten des Clients gespeichert (also Adresse, Port sowie Familie).

int *addrlen
Dies ist die Adresse der Variable in die die Länge der Struktur die die Daten enthält gespeichert wird. Bitte zur Kenntnis nehmen, dass dies nicht wie bei connect ein int ist, sondern ein Zeiger auf int!

Das Codefragment auf das die Welt jetzt wartet:

```
struct sockaddr_in cli;
int cli_size;

c = accept(s, &cli, &cli_size);
```

wobei es für Server in der Regel sinnvoll ist hier eine Endlosschleife zu verwenden, damit der Server nicht nach einer Verbindung abbricht:

```
struct sockaddr_in cli;
int cli_size;

for(;;)
{
        c = accept(s, &cli, &cli_size);

        printf("Verbindung von %s\n", inet_ntoa(cli.sin_addr));
        client_behandlung(c);

        close(c);
}
```

Zu inet_ntoa() später mehr.

select()

Der Befehle select() ist für alle Programme interessant, die ein dynamisches Protokoll implementieren, das nicht immer nach dem Schema senden-empfangen-senden-empfangen läuft, sondern erkennen muss, ob Daten zu lesen oder zu schreiben sind. Ausserdem kann select() verwendet werden, wenn ein Server als einzelner Prozess mehrere Clients bedienen soll, da hier der Server erkennen kann, auf welchem Socket etwas gesendet / empfangen werden soll. Dies ist notwendig, da der Aufruf von recv() so lange wartet, bis etwas empfangen wurde (er ist also blockierend). Der Server würde nun stehen bleiben, und das beim ersten Socket den er überprüft. Eine weitere Möglichkeit wäre nichtblockierende Ein-/Ausgabe (siehe fcntl()). Dies ist jedoch ressourcenunfreundlicher als select, denn select() wartet bis etwas auf einem Socket aus der Socket-Liste ankommt bzw. gesendet werden kann. Ausserdem kann man select() verwenden, um den Programmfluss für eine bestimmte Zeit zu unterbrechen (wie sleep() respektive usleep()). Doch nun zur Deklaration von select():

```
#include time.h>
#include types.h>
#include
int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeo

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

Dies sieht auf den ersten Blick etwas kompliziert aus, doch das legt sich nach der Erklärung (hoffe ich ;-) Der Rückgabewert ist die Anzahl der Deskriptoren für die die geforderten Bedingungen zutreffen. Dies kann auch 0 sein, wenn der Timeout abgelaufen ist, ohne dass eine Verbindung eingegangen ist. Bei einem Fehler wird -1 zurückgegeben.

int n
Dies ist der höchste Deskriptor plus 1. Wenn also der Deskriptor für s überprüft werden muss, gilt n = s + 1.

fd_set *readfds
Dies ist die Adresse des Deskriptor-Sets, das die Deskriptoren enthält die auf eine mögliche Leseaktion überwacht werden sollen. Siehe dazu FD_... weiter unten.

fd_set *writefds
Analog zu readfds, bloss eben die Deskriptoren auf denen geschrieben werden können soll.

fd_set *exceptfds
Auf diesen Deskriptoren treten Exceptions auf. Hierauf wird nicht weiter eingegangen (d.h. ich weiss es selbst nich genau *lol*)

struct timeval *timeout
Dieser Parameter gibt die Adresse eines struct timeval an, in dem der Timeout gespeichert wird, den select() verstreichen lassen soll bevor es mit 0 zurückkehrt. Bei manchen Implementationen wird hier die Restzeit gespeichert wenn vor dem Ablaufen auf einem Deskriptor die geforderten Bedingungen zutreffen. Man sollte sich nicht darauf verlassen, jedoch ist es für portable Programme unbedingt notwendig, dass der Timeout vor einem erneuten Aufruf von select() wieder gesetzt wird, da er eventuell doch verändert worden sein kann.

Die Struktur timeval ist in <sys/time.h> wie folgt deklariert:

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

Ich gebe zu diesem Befehl ein komplettes Beispiel aus der Manpage select(2) von Linux an:

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

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

        /* Watch stdin (fd 0) to see when it has input. */
        FD_ZERO(&rfds);
        FD_SET(0, &rfds);
        /* Wait up to five seconds. */
        tv.tv_sec = 5;
        tv.tv_usec = 0;

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

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

        exit(0);
}
```

Hier wird nochmals verdeutlicht, dass man nach dem Aufruf von select() nicht mehr auf den Wert der Struktur timeval verlassen kann.

close()

Der Befehl close() ist vielleicht einigen schon von der Verwendung von Filedeskriptoren bekannt. Da sich diese genauso verhalten wie Sockets, werden auch Sockets mit close() geschlossen. Unter Win32 wird closesocket() statt close() verwendet, das jedoch die selbe Deklaration hat, nämlich:

```
#include <unistd.h>

int close(int fd);
```

Der Rückgabewert ist 0 bei Erfolg und -1 bei dem Auftreten eines Fehlers, errno wird gesetzt.

int fd

Dies ist der Socket (bzw. allgemein der Filedeskriptor) der geschlossen werden soll

Das typische Codefragment dürfte wohl überflüssig sein :->

send()

Nun wird es interessant! Der Befehl send() wird in der Socket-Programmierung verwendet, um Daten zu versenden. Hierbei wird ein Block von Daten versendet, dessen Inhalt nicht beachtet wird (also keine Überprüfung auf \0 als Ende!). Dieser Block wird in der Regel als ein Paket versendet, es sei denn es wird unterwegs fragmentiert, oder wenn es schlicht und einfach zu gross ist. Dabei ist nicht garantiert, dass auch alles mit einem Aufruf weg ist, doch kann man meistens damit rechnen, da es dann im TCP/IP-Stack des Betriebssystem wartet. Zur Sicherheit gibt send() die anzahl der tatsächlich gesendeten Bytes zurück, oder aber -1 bei einem Fehler. Die Deklaration von send() sieht folgendermassen aus:

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

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

int s
Dieser Parameter bezeichnet den Socket, auf dem die Daten gesendet werden sollen.

const void *msg
Dies ist ein Zeiger auf die Daten, die gesendet werden sollen. In der Regel ist dies ein Buffer, der aus einem Array aus char besteht, jedoch ist dies nicht vorgeschrieben.

int len
Dieser Parameter gibt die Länge des Bereiches an, der mit *msg startet. Im Beispiel eines Buffers ist dies dann die Länge des Buffers (bei binären Daten) oder bei Text die Länge des Strings.

unsigned int flags
Dieser Parameter gibt eventuelle Flags an (in der Regel verwenden wir 0 für "keine Flags"). Ich gebe hier die Erklärung der Manpage send(2) von Linux an:

```
The flags parameter may include one or more of the following:

#define MSG_OOB 0x1 /* process out-of-band data */
#define MSG_DONTROUTE 0x4 /* bypass routing, use direct interface */

The flag MSG_OOB is used to send out-of-band data on sockets that
support this notion (e.g. SOCK_STREAM); the underlying protocol
must also support out-of-band data. MSG_DONTROUTE is usually used
only by diagnostic or routing programs.
```

Als typischen Code-Abschnitt gebe ich einen Abschnitt an, der eine Willkommensnachricht für einen Server ausgibt:

```
int willkommen(int s /* der Socket, wird vom Hauptprogramm übergeben */)
{
        int bytes;
        char buffer[] = "Willkommen zu dem Test-Server\r\n";
        bytes = send(s, buffer, strlen(buffer), 0);
        if (bytes == -1)
        {
                perror("send() in \"willkommen()\" fehlgeschlagen");
                return -1;
```

```
        }

        return 0,
}
```

Auf das "\r\n" gehe ich im Abschnitt über Buffer näher ein.

recv()

Diese Funktion ist wie man schon erahnen kann das Gegenstück zu send(). Recv() empfängt einen Block von Daten (nicht unbedingt der Block der woanders losgeschickt wurde, denn hier wird gelesen was im TCP/IP-Stack steht - wenn das Paket unterwegs fragmentiert wurde kann hier unter Umständen nur ein Teil stehen!) und gibt als Rückgabewert die Zahl der empfangenen Bytes an. Die Deklaration von recv() ist hier zu entnehmen:

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

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

int s
Genau, dies ist der Socket von dem gelesen werden soll.

void *buf
Dies ist die Adresse des Buffers in den der empfangene Block geschrieben werden soll.

int len
Ganz wichtig: dies ist die Grösse des Buffers bzw. die Anzahl der Bytes die man maximal empfangen möchte. Wird diese Zahl falsch gewählt kann und wird es zu Buffer Overflows kommen!

unsigned int flags
Ich verweise auf die Manpage recv(2):

```
The flags argument to a recv call is formed by or'ing one or more of the values:

MSG_OOB process out-of-band data

MSG_PEEK
peek at incoming message
MSG_WAITALL
wait for full request or error
```

Wir verwenden hierbei immer 0 für "keine Flags".

Das Codefragment hierzu stellt die Funktion dar, die die Meldung vom send()-Beispiel aufnimmt und auf den Bildschirm schreibt:

```
#define BUFFER_SIZE 1024        /* ein guter Wert, meiner Meinung nach */
...
int banner_empfangen(int s)
{
        char buffer[BUFFER_SIZE];
        int bytes;

        bytes = recv(s, buffer, sizeof(buffer), 0);
```

```
        if (bytes == -1)
        {
                perror("recv() in \"banner_empfangen()\" fehlgeschlagen");
                return -1;
        }
        buffer[bytes] = '\0';
        printf("Server: %s", buffer);

        return 0;
}
```

Hier ist ebenfalls ein häufiges Phänomen zu beobachten: buffer[bytes] = '\0';. Hierzu ebenfalls mehr im Buffer-Abschnitt dieser Seite.

htons(), ntohs(), htonl(), ntohl()

Ich fasse diese Befehle hier zusammen, da sie im prinzip fast identisch sind. Sie wandeln Zahlen von der Host Byte Order in die Network Byte Order um. Dies hat historische Gründe, da verschiedene Rechner-Architekturen verschieden Anordnungen der Zahlen im Speicher verwenden. Man hat sich im Bereich der Netzwerktechnik auf eine Anordnung geeinigt. Da es aber systemabhängig ist, ob die Zahlen nun umgewandelt werden müssen oder nicht, gibt es diese Funktionen. Die Deklarationen sind folgende:

```
#include <netinet/in.h>

unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

Aus reiner Bequemlichkeit (ich will ehrlich sein ;-) gebe ich hier einfach einen Auszug aus der Manpage an:

```
The htonl() function converts the long integer hostlong from
host byte order to network byte order.

The htons() function converts the short integer hostshort from
host byte order to network byte order.

The ntohl() function converts the long integer netlong from
network byte order to host byte order.

The ntohs() function converts the short integer netshort from
network byte order to host byte order.

On the i80x86 the host byte order is Least Significant Byte first,
whereas the network byte order, as used on the Internet, is Most
Significant Byte first.
```

Benötigt werden diese Funktionen beispielsweise um die Portnummer 80 in die entsprechende Zahl nach Network Byte Order umzuwandeln, die connect() erwartet. Dazu das folgende Code-Beispiel:

```
int main(int argc, char *argv[])
{
        struct sockaddr_in *srv;
        ...
        srv.sin_addr.s_addr = inet_addr(argv[1]);
        srv.sin_family = AF_INET;
        srv.sin_port = htons (atoi(argv[2]));
        ...
}
```

Hier gibt man beim Aufruf des Programms zwei Parameter auf der Kommandozeile mit: der erste ist die IP-Adresse des Hosts mit dem man sich verbinden will, der zweite der Port (in Host Byte Order, also in der "normalen" Schreibweise). Dies muss für das Programm dann in Network Byte Order übertragen werden, damit es auch funktioniert.

Ich hoffe hiermit wurde das hinreichend erklärt. Falls es undeutlich sein sollte bitte ich um Feedback!

inet_addr()

Dieser Befehl wandelt eine IP-Adresse in der "dotted"-Schreibweise, also beispielsweise 127.0.0.1, in eine Adresse um, mit der das Programm etwas anfangen kann: eine 32-bittige Zahl ohne Vorzeichen. Der Befehl inet_addr() sollte nur angewandt werden, wenn man Folgendes beachtet: der Rückgabewert ist die Adresse als unsigned long int, falls die "dotted"-Adresse jedoch nicht umgewandelt werden kann, wird -1 zurückgegeben. Das Problem dabei ist, dass -1 eine gültige Adresse ist, nämlich 255.255.255.255. Man sollte deshalb inet_aton() verwenden, denn laut Manpage ist inet_addr() eine überflüssige Schnittstelle dazu. Ich weiss nicht, aber ich mag inet_addr() trotzdem lieber ;-) Die Deklaration der Funktion ist:

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

unsigned long int inet_addr(const char *cp);
```

Der Parameter **const char *cp** ist hierbei die Zeichenkette, die die IP-Nummer in ihrer "dotted"-Schreibweise enthält.

inet_aton()

Die Funktion inet_aton() wandelt ebenfalls eine Zeichenkette mit einer IP-Nummer in der "dotted"-Schreibweise in eine 32-bit Zahl um. Die Deklaration der Funktion ist folgende:

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

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

Die Struktur in_addr ist in netinet/in.h wie folgt deklariert:

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

Die Parameter der Funktion inet_ntoa() sind:

const char *cp
Die Zeichenkette, die die IP-Adresse in ihrer "dotted"-Schreibweise enthält.

struct in_addr *inp
Ein Zeiger auf die Struktur vom Typ in_addr, die die Adresse aufnehmen soll. Sie ist danach als unsigned long int in inp.s_addr verfügbar, wobei die Struktur in_addr häufig direkt Anwendung findet.

inet_ntoa()

Die Funktion inet_ntoa() ist quasi die Umkehrung von inet_aton(). Sie sorgt dafür, dass die IP-Adresse als 32-bit Zahl wieder in die für uns leichter lesbare "dotted"-Schreibweise konvertiert wird. Die Deklaration von inet_ntoa() sieht folgendermaßen aus:

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

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

struct in_addr in
Dieser Parameter gibt die IP-Adresse als 32-bit Zahl an. Diese kommt zum Beispiel in der Struktur sockaddr_in.sin_addr vor.

Als Beispiel empfehle ich das Beispiel von accept().

gehostbyname(), gethostbyaddr()

Die Funktionen gethostbyname() und gethostbyaddr() lösen Hostnamen in IP-Adresse auf, bzw gehen diesen Weg in die andere Richtung. Ich werde hier nur genauer auf gethostbyname() eingehen, da ich gethostbyaddr() irgendwie nie ganz verstanden habe :->

Die Deklarationen sind wie folgt:

```
#include <netdb.h>
#include <sys/socket.h> /* for AF_INET */

struct hostent *gethostbyname(const char *name);
struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

Der Rückgabewert ist ein Zeiger auf die Struktur hostent, die in netdb.h wie folgt deklariert ist:

```
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 from name server.  */
    #define h_addr h_addr_list[0]       /* Address, for backward compatibility.  */
};
```

const char *name
Der Hostname als Zeichnkette. Beispiel: "home.netscape.com".

Um an die Adresse zu kommen bedarf es eines etwas merkwürdigen Casts. Ich muss selbst regelmässig grübeln bis ich ihn wieder vor Augen habe, deswegen gebe ich ihn hier an:

```
struct sockaddr_in in;

in.sin_addr = *(struct in_addr*) host->h_addr;
```

Also halt: was ist passiert? Nun, host->h_addr ist ein Zeiger auf eine Adresse des Servers. Vom Typ char*, weil dies der Standard-Typ für Zeiger war bevor void* eingeführt wurde. Dieser Zeiger muss nun auf einen Zeiger auf struct in_addr gecastet werden, weil es ja eine Adresse als 32-bit Zahl ist. Danach muss man mittels * auf den Wert des Zeigers zugreifen, da man sonst die Speicher-Adresse bekommen würde. Durch *(struct in_addr*)host->h_addr kriegt man also eine Adresse vom Typ struct in_addr raus, deren Element .s_addr einem unsigned long int entspricht. Puh, das wäre geschafft ;-)

getservbyname(), getservbyport()

Diese Funktionen liefern analog zu gethostbyname() den Service der sich hinter einem Namen (beispielsweise "ftp") verbirgt, oder aber liefern den Service anhand siner Portnummer (z.B. 21). Die Deklarationen der beiden Funktionen sind folgende:

```
#include <netdb.h>

struct servent *getservbyname(const char *name, const char *proto);
struct servent *getservbyport(int port, const char *proto);
```

Der Rückgabewert ist ein Zeiger auf die Struktur servent, die in netdb.h wie folgt deklariert ist:

```
struct servent
{
    char *s_name;               /* Official service name.  */
    char **s_aliases;           /* Alias list.  */
    int s_port;                 /* Port number.  */
    char *s_proto;              /* Protocol to use.  */
};
```

Dabei ist zu beachten: s_port ist in Network Byte Order!
Die Parameter der Funktionen getservbyname() und getservbyport sind:

const char *name
Die Zeichenkette die den Namen des Services enthält (z.B. "ftp")

const char *proto
Diese Zeichenkette enthält den Namen des Protokolls (z.B. "tcp")

int port
Die Portnummer des Services dessen Informationen man einholen will (z.B. htons(21)). Achtung: es wird Network Byte Order verlangt!

Nur für Unix:

fcntl()

Diese Funktion wird verwendet um die Eigenschaften eines Filedeskriptors respektive eines Sockets festzulegen. Ich habe es hier nur aufgeführt, weil man damit Socket nicht-blockierend machen kann, das heisst dass ein recv() beispielsweise sofort zurückkehrt, auch wenn nichts zu lesen ist (dann eben mit 0 als Anzahl der gelesenen Bytes). Man kann damit einen Socket z.B. mit einem Timer jede Sekunde lesen ohne zu wissen ob inzwischen etwas angekommen ist. Man kann diese Aufgabe zwar auch mit select() lösen (dies wäre dann auch portabel für Win32), jedoch braucht man manchmal nichtblockierende Ein-/Ausgabe und mit fcntl() kriegt man sie. Fcntl() kann noch viel mehr, doch gehe ich hier im Rahmen der Socket-Programmierung nicht genauer darauf ein. Wer interessiert ist kann auch in der Manpage fcntl(2) die weiteren Funktionen nachlesen. Die Deklaration von fcntl() ist folgende:

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
```

Wir benötigen dabei die untere.

int fd
Dies ist der Filedeskriptor respektive Socket, dessen Optionen verändert werden sollen.

int cmd
Dieser Parameter beschreibt, welche Funktion ausgeführt werden soll. Um einen Socket nicht-blockierend zu machen setzt man hier F_SETFL ein.

long arg
Dieser Parameter gibt an, welche Option gesetzt werden soll. Wir setzen für unseren Zweck an dieser Stelle O_NONBLOCK ein.

Damit sind wir am Ende der Grundbefehle, also der Socket-API (ja, das war alles wirklich wichtige), angelangt und sind um einiges schlauer, oder? (Feedback)

3. Buffer und warum manchmal Schrott drinsteht

In diesem Abschnitt möchte ich etwas über die Buffer bei der Socket-Programmierung erzählen und auf häufige Fehlerquellen hinweisen, die man mit etwas Sorgfalt erfolgreich vermeiden kann. Beginnen werde ich mit der Antwort auf:

Was ist ein Buffer?
Als Buffer bezeichnet man einen Speicherbereich oder eine Variable, in der man Daten unterbringt bevor man sie weiterverarbeitet. Dies kann entweder bei der Ein-/Ausgabe mit Dateien sein, dass man nicht Zeichenweise von einem Gerät liest (bei einer Festplatte wäre das zum beispiel verschenkte Performance), sondern gleich einen ganzen Block in einen Puffer liest und dann programmintern auswertet. Bei den Standard-Befehlen zur Ein-/Ausagabe (fgets, fputs, fread, fwrite) übernimmt das System die Arbeit einen Puffer anzulegen und diesen zu überwachen. Bei den elementaren Befehlen zur Ein-/Ausgabe (read, write) trägt der Programmierer selbst die Pflicht dafür zu sorgen. Dies kann Vorteile (z.B. kann man die Grösse des Puffers auf die Aufgabe abstimmen - dies kann erhebliche Verbesserungen der Performance bieten), aber auch Nachteile (Puffer läuft über, Puffergrösse ist ineffektiv usw.) haben. Für Sockets gelten in der Regel auch die elementaren Befehle (bzw. recv und sent statt read und write), das heisst auch hier muss man für die Pufferung selbst Sorge tragen.

Um die Bedeutung der Grösse deutlich zu machen, habe ich mal testhalber ein Programm geschrieben um Dateien zu kopieren. Während ich im lokalen Netzwerk (hier 10 MBit, also maximal 1.25 MB/s) mit einer Puffergrösse von 1 (also zeichenweise) kaum Geschwindigkeiten die grösser als 35 KB/s waren, erreichen konnte, so hat sich die Geschwindigkeit auf etwa 950 KB/s erhöht nachdem ich die Puffergrösse auf 1024 erhöht habe (also pro Paket immer 1 KB verschickt habe). Die Erklärung dafür ist recht einfach: ein Paket bei der Übertragung mit TCP/IP besteht nicht nur aus Nutzlast, sondern auch aus administrativen Daten (IP-Header). Diese Daten sind etwa 40 Bytes gross (hab ich mir sagen lassen), also machen sie bei byteweiser Übertragung rund das 40fache der Nutzlast aus. Überträgt man jedoch nun 1024-Byte-Pakete beträgt der IP-Header nur noch etwa ein 25stel der Nuetzlast. Dieser enorme Gewinn an Geschwindigkeit sollte einem lehren sich genau zu überlegen wie man Daten übertragen möchte. Zu grosse Puffer bringen jedoch keinen Vorteil mit sich, da die Pakete unterwegs fragmentiert werden und somit eher noch mehr Arbeit beim wieder zusammensetzen bzw. der Verwaltung im Empfänger-Programm entsteht. Ich habe gute Erfahrungen mit Buffern von 1024 gemacht.

Die Praxis
In der Socket-Programmierung ist ein Buffer meist ein Array aus char. Strings in C haben die Eigenschaft durch ein \0 begrenzt zu sein (null-terminierte Strings). Wenn man nun binäre Daten (beispielsweise Programmcode) übertragen will, kann man sich natürlich nicht danach richten. Hier muss man beim Versenden als Länge (3. Argument bei send() bzw. recv) immer die Anzahl der gelesenen bzw. zu lesenden Bytes angeben. Will man jedoch Text übertragen, so kann man bei send() mit strlen() arbeiten. Dann muss man im Gegenzug jedoch bei recv() auch den Buffer an der Stelle, wo der Text aufhört, mit einem \0 terminieren (wird oft vergessen!). Somit ist ein typisches Codefragment

```
bytes = recv(s, buffer, sizeof(buffer), 0);
if (bytes > 0)
        buffer[bytes] = '\0';
```

Um das nochmal zu verdeutlichen ein Beispiel. Angenommen unser Text ist "Hallo Welt!\r\n", so

sieht es folgendermaßen aus:

```
/* Beim Sender: */
send(s, buffer, strlen(buffer), 0);

buffer:
/----|----|----|----|----|----|----|----|----|----|----|----|----|----\
| H  | a  | l  | l  | o  |    | W  | e  | l  | t  | !  | \r | \n | \0 |
\----|----|----|----|----|----|----|----|----|----|----|----|----|----/

/* Beim Empfänger: */
bytes = recv(s, buffer, sizeof(buffer), 0);

Buffer:
/----|----|----|----|----|----|----|----|----|----|----|----|----|-     -|----\
| H  | a  | l  | l  | o  |    | W  | e  | l  | t  | !  | \r | \n |  ...  |    |
\----|----|----|----|----|----|----|----|----|----|----|----|----|-     -|----/
wobei sich bei ... beliebige Daten befinden können (= Schrott).

/* Deswegen: */
buffer[bytes] = '\0';

/* bytes = 13 (inklusive \r \n) */

Somit ist buffer[bytes] das Feld 14 (Arrays fangen ja bei 0 an zu zählen).

Danach sieht der Buffer wieder so aus:
/----|----|----|----|----|----|----|----|----|----|----|----|----|----\
| H  | a  | l  | l  | o  |    | W  | e  | l  | t  | !  | \r | \n | \0 |
\----|----|----|----|----|----|----|----|----|----|----|----|----|----/

also ist die Datenübertragung erfolgreich verlaufen, der String ist terminiert.
```

Wenn man sich also daran hält bei ASCII-Text immer den Buffer mit \0 zu terminieren, kann eigentlich nichts mehr schiefgehen. Das \r\n wird übrigens verwendet, weil damit plattformunabhängig sichergestellt wird, dass das selbe gemeint ist: Carriage Return + New Line. Bei Unix nämlich ist das mit \n schon erledigt, bei anderen Systemen (beispielsweise DOS oder Windows) ist damit nur New Line gemeint. Ein Beispiel:

```
|-----------------------------------------------|
| Unter Unix (mit \n am Ende):                  |
| Dies ist ein Text                             |
| der normalerweise                             |
| so aussieht                                   |
|-----------------------------------------------|
| Unter Windows (ebenfalls \n am Ende):         |
| Dies ist ein Text                             |
|                  der normalerweise            |
|                                    so aussieht|
|===============================================|
| Unter Unix (mit \n\r am Ende):                |
| Dies ist ein Text                             |
| der normalerweise                             |
| so aussieht                                   |
|-----------------------------------------------|
| Unter Windows (ebenfalls \n\r am Ende):       |
| Dies ist ein Text                             |
| der normalerweise                             |
| so aussieht                                   |
|-----------------------------------------------|
```

Somit wäre bei der Methode \r\n zu verwenden eine gleiche Interpretation auf allen Systemen sichergestellt. Auch Telnet verwendet aus diesem Grund *immer* \r\n.

Noch ein Wort zum Schrott im Buffer
Buffer sollten immer lokale Variablen sein und grunsätzlich sollte man immer davon ausgehen, dass in einem Buffer Zeug steht, mit dem man nichts anfangen kann. Falls zufälligerweise das richtige drinsteht darf man sich auf keinen Fall darauf verlassen, das dies immer der Fall ist. Beim nächsten Systemstart oder auf einer anderen Plattform, ja sogar wenn die Systemzeit die CPU-Geschwindigkeit im Quadrat durch den freien Speicher geteilt überschreitet, kann es völlig anders sein (letzteres soll verdeutlichen, dass "cosmic rays" durchaus ihre Berechtigung haben ;-).
Fazit: traue nur dem, das Du reingeschrieben hast und terminiere mit \0 bzw. weiss genau wie viel Du reingeschrieben hast!

4. sprintf() und andere ANSI-Freunde

Auch wenn man Visual C++ verwendet sind die guten alten Befehle aus der stdio.h, stdlib.h oder string.h nicht tabu. Im Gegenteil: gerade für die Manipulation von Zeichenketten sind sie manchmal unverzichtbar. Zwar kann man in Visual C++ auch einfach durch a += b zwei Strings aneinanderhängen (also strcat() resp. strncat() ersetzen), doch ist sprintf() ein Freund, den man nicht gerne missen möchte. Angenommen (Visual C++ Szenario) in der Variablen m_user steht die Anzahl der User die gerade eingeloggt sind, und in m_hostname steht der Name des Hosts auf dem sie eingeloggt sind, so kann man wählen zwischen

```
C:    sprintf(buffer, "Es sind %i User auf %s eingeloggt", m_user, m_hostname);
VC++: buffer = "Es sind " + m_user + " User auf " + m_hostname + " eingeloggt";
```

(sofern das Konvertieren bei VC++ automatisch geht, bin mir nicht sicher).

Trotzdem kann man mit sprintf() leichter den Inhalt eines Arrays (angenommen dort sind alle Benutzer drin) verarbeiten:

```
for (i = 0; i < user; i++)
{
        sprintf(buffer, "%s, %s", old_buffer, usernames[i]);
        strcpy(old_buffer, buffer);
}
```

bzw. mit strcat() und strncat() geht es noch leichter. Ich will mich jedoch nicht damit verzetteln und sämtliche ANSI-Funktionen rechtfertigen, sondern einfach nur als Anregung geben: vergesst die ANSI-Funktionen nicht, denn sie sind oftmals angenehm und schnell zur Hand.

5. Grundstruktur eines Clients

Ein Client hat zwar je nach Anwendung sehr verschiedene Arbeiten zu verrichten, jedoch kann man die Grundstruktur deutlich ausmachen. Alle Clients haben mindestens zwei Sachen gemeinsam: socket() und connect(). Die Struktur aller Clients ist:

```
/-----------------\
| Socket()        |
|-----------------|
| Connect()       |
|-----------------|
| spezieller Teil |
|-----------------|
| Close()         |
\-----------------/
```

Wobei Close() automatisch erfolgt wenn das Programm beendet wird (es zeugt jedoch von einem schöneren Prgorammierstil wenn es vorkommt). Somit sieht die Grundstruktur als Code-Fragment folgendermaßen aus:

```
/* prg1.c
 * Beispiel für einen Client für Unix
 * (für Win32 geringe Änderungen notwendig)
 */
#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>

#define BUFFER_SIZE 1024

int handling(int sock)
{
        char buffer[BUFFER_SIZE];
        int bytes;

        bytes = recv(sock, buffer, sizeof(buffer), 0);
        if (bytes == -1)
                return -1;
        buffer[bytes] = '\0';

        printf("%s", buffer);

        return 0;
}

int main(int argc, char *argv[])
{
        int s;
        struct sockaddr_in srv;

        if (argc != 3)
        {
                fprintf(stderr, "usage: %s host port\n", argv[0]);
                return 1;
        }

        s = socket(AF_INET, SOCK_STREAM, 0);
        if (s == -1)
        {
                perror("socket failed()");
```

```
                return 2;
        }

        srv.sin_addr.s_addr = inet_addr(argv[1]);
        srv.sin_port = htons( (unsigned short int) atol(argv[2]));
        srv.sin_family = AF_INET;

        if (connect(s, &srv, sizeof(srv)) == -1)
        {
                perror("connect failed()");
                return 3;
        }

        if (handling(s) == -1)
        {
                fprintf(stderr, "%s: error in handling()\n", argv[0]);
                return 4;
        }

        close(s);

        return 0;
}
```

Wobei dann die Ausgabe des Programms folgendermaßen aussieht:

```
felix@murphy:~ > prg1
usage: prg1 host port
felix@murphy:~ > prg1 192.168.1.2 21
220 titania.fun FTP server (Version 6.2/OpenBSD/Linux-0.10) ready.
felix@murphy:~ > prg1 192.168.1.2 15
connect failed(): Connection refused
```

Ersetzt man nun den speziellen Teil unter handling() durch einen anderen Code, so hat man einen anderen Client für einen anderen Zweck. Ich denke hiermit ist die Grundstruktur eines Clients deutlich genug. Man kann sie natürlich noch verbessern, z.B. als host auch Hostnamen zulassen und diese dann mittels gethostbyname() auflösen, oder als Port auch die Namen wie "ftp" verarbeiten. Ausserdem ist es bei grösseren Projekten der Übersicht sehr zuträglich, wenn an den Code auf mehrere Quelldateien aufteilt und ein Makefile erstellt. Doch das kann zur Not so lange warten bis der Lieblingseditor an der 32769ten Zeile streikt ;-)

6. Grundstruktur eines Servers

Bei einem Server ist es ähnlich wie bei einem Client: die Grundstruktur gleicht sich noch, doch wenn es ins Spezielle geht ist natürlich für jeden Zweck ein eigener Server nötig. Die Grundstruktur kann man wie folgt wiedergeben:

```
/-----------------\
| Socket()        |
|-----------------|
| bind()          |
|-----------------|
| listen()        |
|-----------------|
```

```
| accept()        |
|-----------------|
| spezieller Teil |
|-----------------|
| Close()         |
\-----------------/
```

Es empfhielt sich jedoch eine Endlosschleife um das accept() zu basteln, damit der Server nicht nach einem Durchlauf fertig ist. Eine Grundstruktur als Code-Fragment sieht folgendermaßen aus:

```
/* prg2.c
 * Beispiel für einen Server für Unix
 * (für Win32 geringe Änderungen notwendig)
 */
#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>

#define BUFFER_SIZE 1024

int handling(int c)
{
        char buffer[BUFFER_SIZE], name[BUFFER_SIZE];
        int bytes;

        strcpy(buffer, "My name is: ");
        bytes = send(c, buffer, strlen(buffer), 0);
        if (bytes == -1)
                return -1;

        bytes = recv(c, name, sizeof(name), 0);
        if (bytes == -1)
                return -1;
        name[bytes] = '\0';

        sprintf(buffer, "Hello %s, nice to meet you!\r\n", name);
        bytes = send(c, buffer, strlen(buffer), 0);
        if (bytes == -1)
                return -1;

        return 0;
}

int main(int argc, char *argv[])
{
        int s, c, cli_size;
        struct sockaddr_in srv, cli;

        if (argc != 2)
        {
                fprintf(stderr, "usage: %s port\n", argv[0]);
                return 1;
        }

        s = socket(AF_INET, SOCK_STREAM, 0);
        if (s == -1)
        {
                perror("socket() failed");
                return 2;
        }
```

```
        srv.sin_addr.s_addr = INADDR_ANY;
        srv.sin_port = htons( (unsigned short int) atol(argv[1]));
        srv.sin_family = AF_INET;

        if (bind(s, &srv, sizeof(srv)) == -1)
        {
                perror("bind() failed");
                return 3;
        }

        if (listen(s, 3) == -1)
        {
                perror("listen() failed");
                return 4;
        }

        for(;;)
        {
                c = accept(s, &cli, &cli_size);
                if (c == -1)
                {
                        perror("accept() failed");
                        return 5;
                }

                printf("client from %s", inet_ntoa(cli.sin_addr));
                if (handling(c) == -1)
                        fprintf(stderr, "%s: handling() failed", argv[0]);

                /* hier empfiehlt sich kein return mehr, weil sonst der
                 * ganze Server beendet wird wenn ein Client wegstirbt. Das ist
                 * natürlich nicht sinnvoll.
                 */

                 close(c);
        }

        return 0;
}
```

Wobei dann die Ausgabe des Programms folgendermaßen aussieht:

```
Auf der Konsole des Servers:

felix@murphy:~ > prg2
usage: prg2 port
felix@murphy:~ > prg2 2000
(läuft hier immer weiter bis mit Strg + C abgebrochen wird)
felix@murphy:~ >

Auf der Konsole des Clients:

felix@murphy:~ > telnet localhost 2000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
My name is: Felix
Hello Felix
, nice to meet you!
Connection closed by foreign host.
```

Die sonderbare Ausgabe des Servers an den Client (hier telnet) kommt wie erwartet dadurch zustande, dass der ankommende Buffer (hier name[]) einfach weiterverwendet wird, obwohl ja noch die unsichtbaren Sonderzeichen \r\n enthalten sind. Würde man diese abschneiden, so würde die Ausgabe richtig aussehen. Doch dies ist ja nicht mehr die Grundstruktur der Servers, sondern eine genaue Implementierung eines bestimmten Zwecks und nicht Ziel dieses Kapitels. Analog zum Client ist das "Herz" des Servers ebenfalls in handling().

Dieser Server kann zwar nacheinander beliebig viele Clients empfangen, jedoch nur einen auf einmal. Um mehrere Benutzer zu verwalten gibt es verschiedene Ansätze: mit fork() mehrere Prozesse kreieren, oder mit select() mehrere Clients parallel verwalten. Beide Ansätze werden in den weiteren Kapiteln noch besprochen.

7. Tricks mit select()

Select() ermöglicht es mehrere Sockets zu überwachen und dann gezielt auf einzelne zu reagieren. Dies ermöglicht viele Sachen, beispielsweise ein Programm das auf einem Port wartet und alle Anfragen die eingehen 1:1 weitersendet an einen anderen Port, der auch auf einem anderen Server sein kann. Gemeint ist ein Proxy-Server. Angenommen man hat einen Rechner der mit einem Modem eine Verbindung zum Internet aufgebaut hat. Er fungiert als Gateway für ein lokales Netz mittels IP-Masquerading. Nun ist ein Nebeneffekt davon, dass nur der Gateway von aussen (also vom Internet aus) sichtbar ist, die Rechner des lokalen Netzes jedoch dahinter versteckt sind. Nun nehmen wir weiterhin an, dass ein Rechner im lokalen Netz den Web-Server laufen hat, und man aber Aussenstehenden die Daten vermitteln möchte. Man braucht also ein Programm, das die Anfragen die an den Gateway, Port 80, kommen weitergereicht werden zu dem Web-Server auf einem Host der eine IP-Adresse aus dem 192.168.1.x Bereich hat und von aussen nicht bekannt ist (was einen einfachen Portforwarder aus dem Rennen wirft). Um das jetzt zu verwirklichen braucht man also ein Programm, das zwei Sockets offen hat: einen zum Benutzer ausserhalb des Netzes, auf dem die Anfragen reinlaufen und die Daten wieder rausgehen, und einen zweiten der zu dem Web-Server geht, auf dem ebenfalls Anfragen in die eine Richtung und Antworten in die andere laufen. Das Programm muss nun erkennen auf welchem Socket gerade etwas ankommt und diese Daten dann über den anderen Socket wegschicken. Und genau hier kommt select() zum Einsatz.

Ich werde im Folgenden die Implementation eines solchen Proxy-Servers mit select() verdeutlichen.

Allgemeine Teile wie das Starten des Servers, die Schleife sowie das Erzeugen eines eigenen Prozesses mit fork() werde ich nicht angeben (Server mit mehreren Prozessen werden im nächsten Kapitel behandelt).

Dieses Programmfragment stellt die eigentliche Funktion dar. Sie enthält ausserdem noch ein paar Zeilen Code dies ermöglichen zeichenweise hereinkommende Daten zu ganzen Zeilen zusammenzusetzen. Dank dieser Operation kann man auch mit dem zeichenweise arbeitenden Telnet-Client von Windows zeilenbasierende Server wie den Beispielserver in Kapitel 6 bedienen.

Der Code stammt aus einem Programm, das für Interessierte als komplettes Paket unter http://home.t-online.de/home/felix.opatz/sources/bcmp/bcmp_gw-0.8.tar.gz downgeloadet werden kann.

```
/* forward.c
 * Der BCMP Gateway
 * Version 0.8
 *
 * Dieses Programm steht unter GNU Public License. Sie können
 * diese Lizenzbestimmungen unter http://www.gnu.org/copyleft
 * nachlesen. Es besteht KEINERLEI GARANTIE auf die Funktions-
 * weise dieses Programms!
 *
 * Für Fragen bzgl. des Programms wenden Sie sich bitte an
 * Hawkeye  */

#include <fcntl.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>
#include <netinet/in.h>

#include "bcmp_gw.h"

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

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

        fcntl(src, F_SETFL, O_NONBLOCK);
        fcntl(dest, F_SETFL, O_NONBLOCK);

        tv.tv_sec = 600;
        tv.tv_usec = 0;


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

        total = 0;

        for (;;)
        {
                FD_SET(src, &rfds);
                FD_SET(dest, &rfds);
                select(max + 1, &rfds, NULL, NULL, &tv);

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

                if (src_recvd > 0)
```

```
                {
                        write_log(LOG_FWD, "Paket:\tQuelle -> Ziel");
                        if (LINEBUFFERING == 0)
                                send(dest, buffer, src_recvd, 0);
                        else
                        {

                                buffer[src_recvd] = '\0';
                                strcat(linebuffer, buffer);
                                if (linebuffer[strlen(linebuffer)-1] == '\n')
                                {
                                        write_log(LOG_FWD, "Zeile:\tQuelle -> Ziel");
                                        send(dest, linebuffer, strlen(linebuffer), 0);
                                        linebuffer[0] = '\0';
                                }
                        }

                }

                if (dest_recvd > 0)
                {
                        write_log(LOG_FWD, "Paket:\tZiel   -> Quelle");
                        send(src, buffer, dest_recvd, 0);
                }

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

        return 0;
}
```

Mit etwas Abstand zu der Programmierung dieses Programms fällt mir noch ein Fehler auf, der eine Version 0.8.1 rechtfertigen würde. Der Aufruf von select() erfolgt in der Endlosschleife. Davor wird immer wieder das Deskriptor-Set gesetzt, jedoch wird der Timeout nicht wieder erneut auf 600 Sekunden festgelegt. Wie ich bei der Beschreibung von slelect() jedoch betont habe kann man sich nicht auf den Wert den Timeout nach dem Aufruf enthält verlassen kann. Vermutlich wird dieses Programm (unter Linux jedenfalls, denn das verändert den Timeout hierbei) ohne Problem laufen, bis ein Client kommt und für insgesamt mehr als 600 Sekunden keine Daten sendet. Das heisst nach etwa 10 Minuten wird der Server ein Problem kriegen: er wird bei select() nicht mehr anhalten, sondern gleich weitermachen. Dadurch wird der Verbrauch an Systemressourcen auf nahezu 100 % hochschnellen und das System ist matt gesetzt. Wenn der Timeout jedoch 0 ist, wartet Select unbegrenzt ... ob sich das ausgleicht? Ob der Server noch mal mit einem blauen Auge davon kommt? Das einfachste wäre es das ganze einfach mal auszuprobieren ...

Worauf ich aber mit diesem Beispiel aber hinaus wollte war die Implementierung eines Proxy-Servers. Wie man sieht wartet Select() darauf, dass von einem der beiden Sockets gelesen werden kann. Ist dies der Fall, wird davon gelesen (man hätte auch mit FD_ISSET() testen können von welchem Socket gelesen werden kann, doch so haben wir gleich noch ein Beispiel für nicht-blockierende Ein-/Ausgabe). Dadurch, dass die Sockets durch den Aufruf von fcntl() am Anfang mit dem Attribut O_NONBLOCK versehen wurden, blockiert ein recv() nicht so lange, bis etwas zu Lesen da ist, sondern kommt sofort zurück, eben mit 0 wenn nichts gelesen wurde. Die beiden if-Abfragen danach überprüfen, von welchem der Sockets etwas gelesen wurde (nämlich welcher Wert > 0 ist). Ist von dem Socket an dem Benutzer ausserhalb hängt gelesen worden, so wird erst eine Zeile zusammengesetzt (sofern mit #define LINEBUFFERING 1

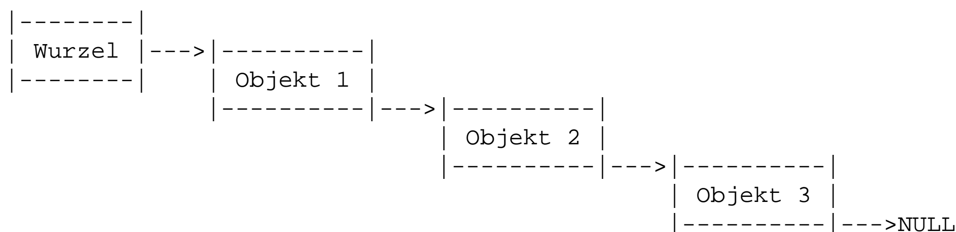
übersetzt wurde). Wird von der anderen Seite (bei uns der Web-Server) gelesen, so wird das Paket sofort weitergeschickt. Falls von beiden Sockets gleichzeitig nichts gelesen wird, so beendet sich der Prozess. Spätestens hier wird unser Server ein Problem kriegen, da der Timeout ja fest gesetzt war und nicht unendlich ist. Man kann also sagen: ein Server-Prozess hat sofern nicht ständig reger Verkehr herrscht eine Lebenszeit von maximal 600 Sekunden. Falls beim Ablauf der Timeouts nichts gesendet wurde ist Schluss.

Wie dieses Beispiel zeigt kann man mit select() interessante Probleme lösen, doch wenn man nicht aufpasst kann man auch flott einen Fehler einbauen, der sich irgendwann als Zeitbombe herausstellen kann. Wenn man bei select() jedoch auf alles achtet hat man ein gutes Werkzeug an der Hand, um komplizierte Probleme zu lösen.

8. verkettete Listen für sparsame Aufgaben

Wenn man einen Server schreiben möchte, der zwar mehrere Benutzer bedienen soll, diese Benutzer jedoch miteinander interagieren sollen (bei unserem Beispiel miteinander chatten) kann man nicht mehrere Prozesse verwenden, da hier die Interprozesskommunikation (IPC) zu kompliziert ausarten würde. Für solche Server verwendet man eine andere Methode:

Wir haben einen Server, der wie gewohnt auf einem Socket auf neue Benutzer lauscht. Jedoch passiert dies in einer etwas abgewandelten Schleife, mit einem select() davor. Wenn nun auf dem Lausch-Socket gelesen werden kann (ist der Fall wenn sich jemand einloggt) wird die Prozedur mit accept() abgefahren. Ansonsten wird jedoch wenn auf einem anderen Socket (der dann einem der Clients gehört) etwas ankommt dieses gelesen und an alle Sockets ausser dem Lausch-Socket gesendet. Somit bekommt jeder das mit, das einer geschrieben hat. Man kann dafür entweder ein statisches Array nehmen, das die einzelnen Sockets aufnimmt, doch begrenzt dies dann die maximale Anzahl der User. Eine andere Möglichkeit sind verkettete Listen. Hierbei repräsentiert ein Objekt einen Benutzer. Dies ist ein struct mit den Elementen die die Verwaltung braucht (hier beispielsweise den Socket) sowie ein Element das ein Zeiger auf ein struct ist und auf das nächste Element zeigt (oder auf NULL wenn das Element das Ende der Kette ist). Dadurch ergibt sich folgende Konstruktion:



Wichtig für das Programm ist dabei, dass bei select() als erster Parameter wirklich der höchste Socket + 1 steht (hierfür muss man die gesamte Liste durchlaufen und den höchsten Socket ermitteln. Dann einfach noch eins dazu zählen und das war's). Das Versenden an alle Benutzer läuft ähnlich einfach: man durchläuft die gesamte Liste und schreib in jeden Socket die Nachricht rein. Wenn -1 als Rückgabewert entsteht (also der Client nicht erreichbar ist) wird er einfach aus der Liste genommen und das "Loch" geflickt, indem man den Zeiger vom vorhergehenden Objekt auf das nächste setzt, also das rausgeworfene übergeht. Ausserdem braucht man Hilfsfunktionen, die neue Objekte einfügen. Ein solches Programm, das alle diese Sachen

erledigt, habe ich unter dem Namen BCMP_chat in dem /sources/bcmp-Verzeichnis auf meiner Homepage. Einfach mal vorbeschauen unter http://home.t-online.de/home/felix.opatz/sources/bcmp/.

Diese Technik hat jedoch auch ein paar Probleme: sofern nicht die Anzahl der Benutzer begrenzt wird, kann es passieren dass die Ressourcen ausgehen. Ausserdem kann der Server leicht blockiert werden: wenn ein neuer Benutzer kommt wird er zuerst nach einem Nickname gefragt. Diese Abfrage ist in Version 0.3.1 des Servers noch blockierend und muss noch mal überdacht werden, denn wenn hier nichts eingegeben wird bleibt die gesamte Verbindung hängen. Eine einfache Lösung wäre vor dem recv() eine select() zu setzen, dass einen Timeout von maximal 5 Sekunden zulässt. Wenn in der Zeit nichts kommt wird der Client einfach verworfen. Dar der Chat-Client in der Regel gleich den Nickname sendet und eine Verspätung von 5 Sekunden schon wirklich viel ist, sollte es hier für reguläre Benutzer keine ernsthaften Probleme geben. Man kann zwar immer noch alle 5 Sekunden den Server von neuem blocken, aber man könnte sich diese Block-Versuche ja notieren und wenn ein Client 3 davon versucht hat wird er in Zukunft einfach ignoriert ;-) Na gut, mit IP-Spoofing kann man ja ... das ist bekannt, jedoch kann man Server auch mit einem SYN-Flooder lahmlegen, und das egal wie gut sie programmiert sind. Das fällt nicht in den Bereich von Programmierer für einfache Server, sondern da hat eine ausgeklügelte Spoofing-Protection ihre Berechtigung. Wir schweifen vom Thema ab :-o

Zusammenfassend kann man sagen: die Technik mit select() mehrere Sockets zu verwalten stösst an ihre Grenzen, wenn der Service sehr Übertragungsintensiv ist, denn dann bleibt der Server irgendwann auf der Strecke mit seinen send() und recv() (wobei die Schwachstelle wohl eher die Bandbreite sein wird, da die ausgehenden Daten vom send() gleich in den TCP/IP-Stack wandern). Hat man Übertragungsintensive Programme (z.B. Filetransfers oder ähnliches) sollte man die Methode verwenden, die im nächsten Kapitel behandelt wird: Server mit mehreren Prozessen.

9. mehrere Prozesse für anstrengede Aufgaben

Wenn ein Server Arbeiten verrichten soll, die ihn fast ständig in Aspruch nehmen, er aber nicht oder nur wenig mit anderen Benutzern kommunizieren muss, so bietet es sich an einen Server zu schreiben, der für jeden Benutzer einen eigenen Prozess startet. Dies geschieht unter Unix mit fork(). Unter Windows stehen dafür andere Methoden zur Verfügung, auf die ich hier nicht eingehen werde. Die Deklaration von fork() ist die folgende:

```
#include <unistd.h>

pid_t fork(void);
```

pid_t ist ein primitiver Datentyp und nimmt die PID (Process ID) des Prozesses auf. Der Rückgabewert ist etwas knifflig, da es zwei gibt: fork() verdoppelt beim Aufrufen den aktuellen Prozess und jedes Exmeplar bekommt einen Wert zurück. Dabei bekommt der Elternprozess die PID des erzeugten Kindprozesses, während der Kindprozess 0 bekommt. Wenn fork() fehlschlägt, so liefert es -1 an den Aufrufer (da es ja nur einen gibt, wenn es fehlgeschlagen hat). Ein somit häufig erscheinendes Codefragment ist:

```
{
        ...
        pid = fork();
        if (pid == -1)
        {
                perror("fork() failed");
                return -1;
        }

        if (pid == 0)
        {
                /* Kindprozess */
                ...
                exit(0);
        }

        /* Elternprozess;
        ...
        return 0;
}
```

Wobei für die "..." natürlich beliebiger Code folgen kann. Fork() ist ausserdem interessant für Programme, die als Daemon weiterlaufen sollen (mit TSRs für DOS vergleichbar). Hierbei soll sich der Elternprozess beenden. Da der Kindprozess dann verwaist ist, wird seine PPID (Parent Process IF) zu 1 (der PID des init-Prozessed, der niemals stirbt [there can be only one ;-]). Das Programm läuft dann weiter und die Konsole ist wieder frei. Eine solche Funktion sieht oft folgendermassen aus:

```
int daemonize(void)
{
        pid_t pid;

        pid = fork();
        if (pid == -1)
        {
                perror("fork() failed");
                return -1;
        }

        if (pid == 0)
                return 0;       /* Kindprozess */

        exit(0);                /* Elternprozess */
}
```

Ein Problem mit Kindern ist, dass sie zu Zombies werden wenn sie sterben (der Satz klingt makaber ;-). Zombies entstehen, wenn Kinder sterben und es die Eltern "nicht interessiert". Ein richter Elternprozess wartet wenn ein Kind stirbt bis es tot ist (wird immer schöner ;-). Dies geschieht mit wait(). Doch ist wait() blockierend, das heisst dass der Server so lange warten würde, bis das Kind tot ist. Also lässt er wieder nur einen Benutzer zu. Doch glücklicherweise sendet ein totes Kind ein Signal aus, nämlich SIGCHLD. Nun gibt es die Funktion signal(), die einen Signalhandler installiert und eine Funktion immer dann ausführt, wenn ein bestimmtes Signal eintrifft. Das passende Codefragment ist das folgende:

```
void kind_tot(void)
{
        wait();
}
```

```
int main(int argc, char *argv[])
{
        ...
        signal(SIGCHLD, (void*)kind_tot);
        ...

        return 0;
}
```

Genial einfach, einfach genial ;-)

Ich denke damit dürften die Grundzüge für Server mit mehreren Prozessen klar sein. Zwar war dieses Kapitel recht knapp, doch denke ich Leute die sich dafür interessieren werden noch weitere Informationen dazu finden, zum Beispiel in den entsprechenden Manpages signal(2), wait(2), fork(2) und kill(2).

10. abschliessende Worte und eigener Senf für die Welt ;-)

Nun, wir sind am Ende meiner Tipps für Socket-Programmierer und solche die es werden wollen angelangt. Ich hoffe dass dieser Text dem/der einen oder anderen als Sprungbrett in die Netzwerkprogrammierung hilft, oder einfach nur einen interessanten Einblick in die Welt derSocket-Programmierung gegeben hat. Falls noch Fragen bestehen oder Ihr Anregungen habt, wie ich diese Seite noch verbessern kann, einfach mal eine E-Mail an gallenstein@web.de schicken. Vielleicht kommen ja noch ein paar neue Ideen zusammen und es reicht um einen weiteren Ausflug in die Programmierung zu unternehmen. Vieleicht Systemprogrammierung allgemein? Mehr über Prozesse und Interprozesskommunikation (IPC)? Ich bin der Meinung dass ich mit dieser Seite gut gebrüllt habe und warte einfach mal auf das Echo.

(1) Einen guten C-Compiler der die komplette Win32 API ausschöpft gibt es unter http://www.cs.virginia.edu/~lcc-win32/

(2) Die Unix-API findet man in der Regel in den Manpages, ich kann aber das Buch unter (3) wärmstens empfehlen. Für Win32 existiert auch ein Verzeichnis der kompletten API und ist bei Visual C++ dabei, oder auch im Internet downzuloaden (Dateiname: win32.hlp und win32s.hlp. Einfach mal mit einer Suchmaschine suchen).

(3) Linux / Unix Systemprogrammierung, ISBN 3-8273-1512-3, Addison-Wesley Verlag.

---

geschrieben von Hawkeye <gallenstein@web.de>, Juli 2000
Dieses Dokument stammt von http://home.t-online.de/home/felix.opatz/socket-tipps.html
Weitergabe und Vervielfältigung **erwünscht**, solange die Urheberrechte gewahrt bleiben
Ich garantiere nicht für die Richtigkeit der Informationen auf dieser Seite
(insbesondere nicht für die Rechtschreibung ;-)