C9 Programmierung der Datenkommunikation, Sockets

Bei der Datenkommunikation findet ein Datenaustausch zwischen zwei, evtl. auf verschiedenen Rechnern ablaufenden Programmen statt, dies nennt man auch **Interprozesskommunikation**.

Diese Interprozesskommunikation wird benötigt bei der Programmierung von Client-Serversystemen aber auch bei der Realisierung von **Pipes.**

Mit dem Befehl `prog >x1.dat` kann das Programm `prog` seine Standardausgabe auf eine Datei `x1.dat` statt auf die Konsole schreiben. Mit `prog <x1.dat` kann man statt von der Tastatur über die Standardeingabe von der Datei `x1.dat` lesen.
Mit `prog <x1.dat >x2.dat` kann man beides auch kombinieren.
Um dies zu realisieren muss die Eingabe von der Tastatur bzw. die Ausgabe auf die Konsole mit den gleichen Prozeduraufrufen möglich sein, wie die Ein- bzw. Ausgabe auf Dateien.

Mit dem Befehl `prog1 | prog2` kann man erreichen, dass die Standardausgabe von `prog1` die Standardeingabe von `prog2` wird. Dies nennt man eine **Pipe**. Unter DOS wurde dies erreicht, indem `prog1` auf eine Datei geschrieben hat und `prog2` hat diese Datei gelesen. Dabei musste aber das Programm `prog1` zunächst vollständig abgearbeitet werden, bevor `prog2` begonnen wurde und evtl. eine Ausgabe von `prog2` am Bildschirm erschien.

Mit Hilfe der **Sockets** kann man erreichen, dass `prog1` und `prog2` parallel laufen können. Eine Eingabe bei `prog1`, die eine Ausgabe bewirkt, kann sofort über die Pipe in `prog2` weiter verarbeitet werden und so eine sofortige Ausgabe auf dem Bildschirm bewirken. Dies erfordert aber, dass die Systemaufrufe für die Ausgabe auf eine Pipe mit Hilfe der Sockets die gleichen sein können, wie die Aufrufe zur Ausgabe auf eine Datei. Für die Eingabe muss das gleiche gelten. Dem wird die Socket- Bibliothek von UNIX gerecht.

socket()

```
#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 Filedeskriptor (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_STREAM, 0);
if (s == -1)
{
        perror("socket() failed");
        return 1;
}
```

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  Interprozesskommunikation auf einem System
```

```
AF_INET  Kommunikation über TCP/IP bzw. UDP/IP
AF_ISO
AF_NS
AF_IMPLINK
```

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
```

SOCK_STREAM für die Kommunikation mit verbindungsorientierten TCP.
SOCK_DGRAM für die Kommunikation mit verbindungslosem UDP.

int protocol
Der dritte Parameter 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.

Die Zahl der Sockets ist nicht unbegrenzt. Nichtmehr benötigte Sockets sollten mit close() freigegeben werden (dies geschieht übrigens automatisch, wenn das Programm beendet wird).

Im Client wird der so erzeugte Socket mit einem Server verbunden.

connect()

Mit connect() wird eine Verbindung von einem Client 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. Für eine Verbindung ins Internet wird die Struktur sockaddr_in, verwendet, die wie folgt deklariert ist:

```
#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  */
};
```

Für die Interprozesskommunikation (AF_UNIX) wird der Pfad des Serverprogramms angegeben.

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

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

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

Auf der Serverseite muss zunächst festgelegt werden, welchen Dienst der Socket bedienen soll. Bei AF_INET wird der Dienst beschrieben durch die IP- Adresse und die Portnummer.

bind()

Mit bind() wird ein Socket mit einer 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 gleiche Struktur sockaddr_in wie in 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.

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

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

listen()

Dieser Befehl versetzt den Socket in den Lausch-Modus, so dass sich ein Client mit ihm verbinden kann.

```
#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

Mit dem Befehl accept holt der Server die Information aus der Warteschlange ab.

accept()

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

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

Der Rückgabewert ist bei Erfolg jedoch der neue Socket, der den Client beschreibt, bzw. den man für die Eingabe benutzen kann, im Fehlerfall -1. 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.

```
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);
}
```

Mit den Befehlen send und recv wird nun die Information geschrieben bzw. gelesen.

send()

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").

Als typischen Code-Abschnitt ein Abschnitt, 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,
}
```

recv()

Diese Funktion ist 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.

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

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

int s
der Socket von dem gelesen werden soll.

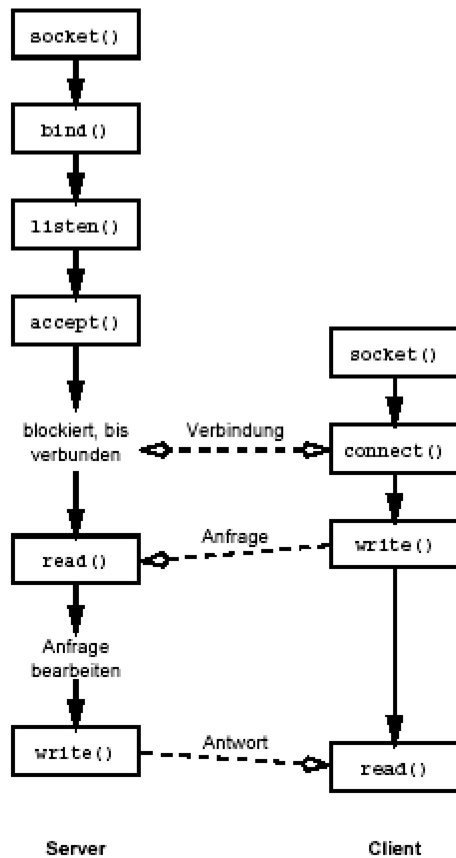
void *buf
die Adresse des Buffers in den der empfangene Block geschrieben werden soll.

int len
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

Statt send() und recv() können auch die Standard Ein- Ausgabefunktionen read() und Write() benutzt werden. Der Dateihandle wird ersetzt durch den Socketidentifier, der auch bei send() bzw. recv() benutzt wird.

Insgesamt stellt sich die Kommunikation zwischen Client und Server bei einer verbindungsorientierten Kommunikation wie folgt dar:



Das Codefragment 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;
}
```

Beispiel für einen Server:

```
/* 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(;;)
          {
          cli_size = sizeof(cli);
          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;
}
```

Will man eine verbindungslose Kommunikation durchführen kann man die Funktionen sendto() und recvfrom() benutzen.

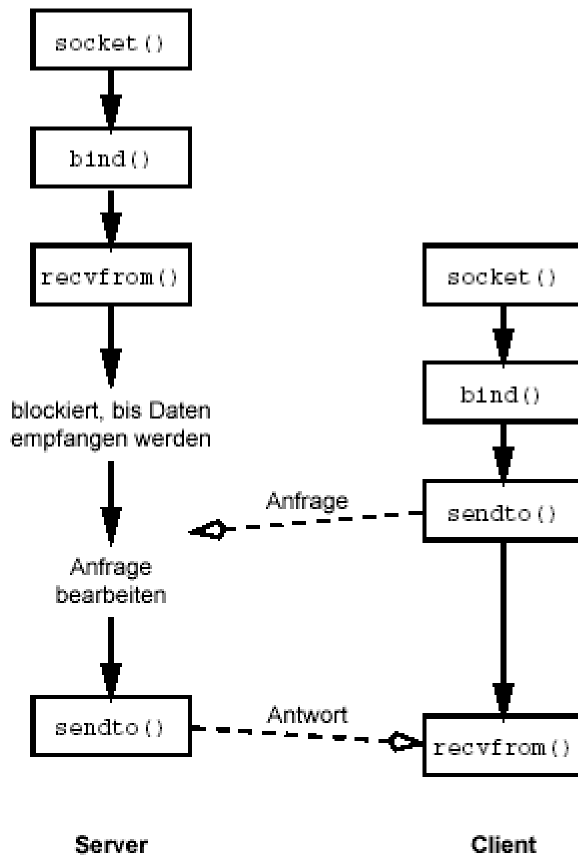
```
cc = sendto ( s, buf, len, flags, to, tolen );
int cc, s;
char *buf;
int len, flags;
struct sockaddr *to;
int tolen;
```

Der Parameter `to` beschreibt das Ausgabeziel. Die Angaben sind die gleichen, wie bei der Funktion `connect()`.

```
cc = recvfrom ( s, buf, len, flags, from, fromlen );
int cc, s;
char *buf;
int len, flags;
struct sockaddr *from;
int fromlen;
```

Hier werden von der durch `from` beschriebenen Quelle maximal `len` Bytes gelesen.

Insgesamt stellt sich die Kommunikation zwischen Client und Server bei einer verbindungslosen Kommunikation wie folgt dar:



Pipes

Mit der Funktion
`int pipe(int filedes[2]);`

erhält man 2 Filedescriptoren in dem `filedes` Array.
`filedes[0]` ist das Leseende
`filedes[1]` ist das Schreibende einer Socketverbindung.

Bei dem Befehl `prog1 | prog2` wird bei Programm `prog1` der Standardausgabe der Filedeskriptor `filedes[0]` und bei `prog2` der Standardeingabe der Filedeskriptor `filedes[1]` zugeordnet.

Wenn von einer pipe gelesen wird:

- read() gibt 0 (end of file) wenn das Schreibende der pipe geschlossen ist.
- Wenn das Schreibende noch offen ist und keine Daten da sind, schläft prog2 bis Daten da sind oder das Schreibende geschlossen wird.
- Wenn read() versucht mehr Daten zu lesen, als z.Z. in der Pipe, read() liefert nur die Anzahl Bytes, die tatsächlich gelesen wurden. Nachfolgende read() bleiben stehen, bis wieder Daten da sind.

Wenn beim Schreiben auf eine Pipe, das Leseende geschlossen ist, wird das Signal SIGPIPE erzeugt. Die Standardbehandlung bricht das Programm ab.