W&T
www.WuT.de

Fit in 1 Tag
fürTCP/IP-Sockets

die Socket-Schnittstelle

WinSock

Beispielcode zur Einbindung von Com-Servern

TCP/IP-Grundlagen

Einführung

Vermutlich sind Sie Programmierer oder Programmiererin. Vermutlich haben Sie schon früher Peripheriegeräte – etwa über den COM-Port – in Ihre Anwendungen eingebunden. Und nun haben Sie Lust bekommen, diese Geräte direkt über das Netzwerk einzubinden.

Nachdem Sie die erste Scheu vor der neuen Materie überwunden haben, werden Sie merken, daß der Weg über das Netzwerk für Programmierer gar nicht komplizierter ist als der gute alte COM-Port.

Zwar kommen ein paar neue Begriffe hinzu, dafür fällt aber auch manche Ungereimtheit der COM-Anbindung weg. Also fangen wir doch gleich an!

Fit für TCP/IP-Sockets in 1 Tag – so können Sie vorgehen:

1. Lesen Sie zuerst *Teil 1 – Die Socket-Schnittstelle* So verschaffen Sie sich nicht allein schnelle Übersicht, sondern finden auch bereits manche wichtige Detailinformation, die Ihnen später ein zielgerichtetes Arbeiten ermöglicht.
2. Nun brauchen sie eigentlich schon eine „Spielwiese“, auf der Sie Ihr neu erworbenes Wissen erproben können. Um nicht gleich mit zwei Anwendungen beginnen müssen, empfehlen wir Ihnen, zu diesem Zweck bei uns einen Com-Server als Testgerät zu bestellen, der dann die Aufgaben des Netzwerk-Kommunikationspartners (Socket-Client oder -Server) übernimmt.

   (Falls Sie besonders eilig sind, können Sie natürlich auch versuchen, eine Kommunikation zwischen zwei Rechnern aufzubauen. Dieser Weg ist anspruchsvoller, da Sie zwei Programme gleichzeitig zum Laufen bringen müssen. Ohne einschlägige Vorerfahrung sollten Sie also lieber von dieser Idee Abstand nehmen.)
3. Nun wählen Sie aus *Teil 2 – Programmierbeispiele Socket-API* die Beispiele für Ihre jeweilige Programmierumgebung aus. Die Quellen einschließlich aller notwendigen Includes und Resource-Dateien sowie weitere Beispiele für Windows 9x/NT können Sie auch unter *http://www.WuT.de* herunterladen.
4. Spätestens beim Durcharbeiten der Programmbeispiele werden Sie auf neue Begriffe aus der Netzwerkwelt stoßen. Doch was Sie zum Verständnis dieser Begriffe brauchen, ist sehr kompakt in *Teil 3 – TCP/IP-Grundlagen* zusammengestellt.

Im Zusammenspiel mit den Programmbeispielen und Ihren eigenen Experimenten sollte sich Ihnen das erforderliche Verständnis bald erschließen.

Wer trotzdem noch eine ausführlichere Einführung wünscht, sei auf die Literaturangaben im Anhang verwiesen.

Inhalt

Teil 1 Die Socket-Schnittstelle

Teil 2 Programmbeispiele

Teil 3 TCP/IP-Grundlagen

Teil 1
Die Socket-Schnittstelle

Client-Server-Prinzip
die Socket-Variable
Socket-Funktionen
Host- und Network-Order
WinSock-spezifische Funktionen
Streams und Datagramme
Client- und Server-Applikationen

1. Die Socket-Schnittstelle

Das Socket-Interface, das unlängst als Neuerung Einzug in die PC-Welt hielt, wurde vor mehr als 15 Jahren als „Berkley Socket Interface“ unter BSD-UNIX 4.3 entwickelt. Diese Schnittstelle ermöglicht mittels weniger einfach zu handhabender Befehle den Zugriff auf die Funktionalität von TCP/IP; im Laufe der Jahre wurde sie von vielen anderen UNIX-Systemen aufgegriffen.

Auch unter Windows 9x gehört die *WinSock.DLL* und damit die Funktionalität der Socket-Schnittstelle mittlerweile zum Lieferumfang. Die Entscheidung für diese Schnittstelle ist leicht nachzuvollziehen. Sie ermöglicht neben der Neuentwicklung von Internetanwendungen auch das Portieren der Anwendungen von UNIX auf den PC, da sie größtenteils kompatibel zum Berkley Socket ist.

1.1 Client-Server-Prinzip

Internetanwendungen werden nach dem Client-Server-Prinzip erstellt. Der Client ist hier in den meisten Fällen das Benutzer-Interface und nimmt vom Server bestimmte Dienste in Anspruch. Er baut in Abhängigkeit vorher definierter Ereignisse (z.B. dem Starten einer Internetanwendung durch einen Anwender) die Verbindung zum Server auf und ist somit der aktive Teil.

Der Server stellt nun den gewünschten Dienst zur Verfügung. Er muß sich ständig in einem Zustand befinden, in dem er Verbindungsaufforderungen von Clients entgegennehmen kann – er ist der passive Teil. Ein Server darf niemals einen Dienst vom Client anfordern.

Client und Server müssen die gleiche Sprache sprechen: Sie müssen sich also an einem gemeinsamen Protokoll orientieren.

Das unterschiedliche Verhalten von Client und Server läßt allerdings eine Asymetrie entstehen, die sich in der Verwendung unterschiedlicher Schnittstellen-Befehle bei der Realisierung einer Client- oder Serverapplikation niederschlägt.

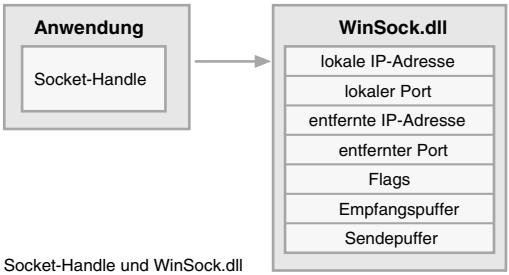
1.2 Einbindung der Socket-Funktionen in C

Um die Funktionalität der WinSock-Schnittstelle in die eigene Applikation zu integrieren, binden Sie das File *winsock.h* mit *#include <winsock.h>* in den C-Quellcode ein. Es enthält alle Deklarationen der Datentypen, Konstanten und Funktionen des Interfaces.

Compiler und Linker benötigen die LIB-Datei *winsock.lib* für 16-Bit-Anwendungen oder *Wsock32.lib* für 32-Bit-Anwendungen zur Erzeugung des Programmcodes. Binden Sie also eine der beiden Dateien in ihr Projekt ein.

1.3 Die Socket-Variable

Die Socket-Variable ist vom Typ Integer. Sie hat nichts mit der Portnummer der Applikation zu tun, wie oft angenommen wird, sondern ist lediglich ein Handle für eine Verbindung. Unter dieser Nummer erhält man vom Treiber mittels verschiedener Befehle alle Informationen zu der Verbindung, die zu diesem Handle gehört.



Socket-Handle und WinSock.dll

1.4 Die wichtigsten WinSock-Funktionen

Der Umfang an WinSock-Funktionalitäten kann je nach Bezugsquelle des Entwicklungstools recht unterschiedlich ausfallen. Oft findet man eine Vielzahl undokumentierter Funktionen vor, mit denen niemand so recht etwas anzufangen weiß.

Die wichtigsten Funktionen der WinSock-Schittstelle können Sie der folgenden Tabelle entnehmen. Bei dieser Aufstellung handelt es sich um die Basisfunktionen, die in allen Versionen vertreten sind.

Basisfunktionen	Beschreibung
accept	nimmt in einem Serverprogramm eine neue Verbindung an und liefert einen neuen Socket auf diese Verbindung
bind	verknüpft einen Socket mit einer lokalen Internet-Adresse und einem Port
closesocket	schließt einen Socket
connect	baut eine Verbindung zwischen zwei Sockets auf
getpeername	liefert die IP-Adresse der Gegenstation aus einem Socket
getsockname	liefert die eigene IP-Adresse aus einem Socket
getsockopt	liest die Flags eines Sockets aus
ioctlsocket	Einstellung von Socket-Flags
listen	erzeugt eine Warteschlange für einkommende Verbindungen
recv	Daten über eine Socketverbindung empfangen
recfrom	Daten empfangen und Absender auslesen
send	Daten über eine bestehende Socketverbindung senden
sendto	Daten an eine gegebene Adresse senden
select	überprüft den Status einer gegebenen Menge von Sockets
shutdown	unterdrückt wahlweise das Senden und Empfangen von Daten über einen Socket
socket	legt einen neuen Socket an

1.5 Network-Order oder Host-Order?

Wer schon einmal eine Netzwerkapplikation programmiert hat, kennt das leidige Problem mit der Byteorder. Es hat seine Ursache in der system- oder architekturabhängigen Interpretation des Speichers, wenn dieser nicht Byte für Byte sondern z.B. als WORD oder LONG ausgewertet wird. Damit eine Applikation auf einem Intel-PC auch die Dienste einer Applikation z.B. auf einem Macintosh in Anspruch nehmen kann, mußte eine Normierung für die Übertragung von WORDs und LONGs über das Netzwerk gefunden werden.

Die Network-Order (auch Big-Endian genannt) besagt, daß das höchste Byte zuerst, das niedrigste Byte zuletzt übertragen wird.

Speicher-adresse	Little-Endian (Host-Order)	Big-Endian (Net-Order)	Little-Endian (Host-Order)	Big-Endian (Net-Order)
n+3			31 ... 24	7 ... 0
n+2			23 ... 16	15 ... 8
n+1	15 ... 8	7 ... 0	15 ... 8	23 ... 16
n	7 ... 0	15 ... 8	7 ... 0	31 ... 24
	16 Bit WORD		32-Bit DWORD	

Da auf allen Intel-PCs der Speicher nach der Host-Order ausgewertet wird (das niederwertige Byte also vor dem höherwertigen im Speicher steht), ist es notwendig, alle Werte mit dem Type LONG oder WORD vor der Übergabe an den Treiber in die Network-Order zu konvertieren. Analog müssen alle Werte nach Erhalt vom Treiber in die Host-Order konvertiert werden, bevor mit ihnen gerechnet werden kann. Vorsicht ist auch bei Vergleichen geboten: Sie liefern in Network- und Host-Order nicht das gleiche Ergebnis!

Das Socket-Interface stellt hier eine Reihe von Konvertierungs-Funktionen zur Verfügung. Wer sich die Tabelle etwas genauer anschaut, wird feststellen, daß es für jede Richtung (Network -> Host-Order und Host -> Network-Order) jeweils eine Funktion gibt. Eigentlich ist dies eine Doppelung, denn welche Variante man nimmt – sie wird und muß das gleiche Ergebnis liefern. Der einzige Vorteil liegt in der besseren Lesbarkeit des Programms. Anhand des benutzten Befehls kann man erkennen, ob der jeweilige Wert anschließend in Host- oder Network-Order vorliegt.

Konvertierungs-funktionen	Beschreibung
htonl	32-Bit-Integer: Host -> Network Byte Order
ntohl	32-Bit-Integer: Network -> Host Byte Order
htons	16-Bit-Integer: Host -> Network Byte Order
ntohs	16-Bit-Integer: Network -> Host Byte Order
inet_addr	wandelt IP-Adresse im Stringformat in numerische Adresse als *long* um
inet_ntoa	wandelt eine IP-Adresse als *long* in den korrespondierenden String um

Die Funktionen *inet_addr()* und *inet_ntoa()* weichen etwas von den anderen ab. Sie wandeln eine Internetadresse, die als String im *Internet Standard Dotted Format* vorliegt, in einen 32-Bit-Wert (*inet_addr()*) oder umgekehrt (*inet_ntoa()*).

1.6 Datenbankfunktionen

Diese Funktionen dienen dazu, Informationen über Namen, IP-Adressen, Netzwerkadressen und andere Daten vom Treiber zu beziehen. Ein User wird z.B. im Dialogfeld einer Applikation den Zielhost nicht als IP-Adresse eingeben, sondern den Namen der Station eintragen. Niemand könnte sich die vielen IP-Adressen im Internet merken. Da sich Namen viel besser einprägen als Nummern, ist es üblich, auch hier mit Namen zu arbeiten.

Datenbankfunktionen sorgen z.B. für die Konvertierung von Namen in IP-Adressen und umgekehrt. Sie nehmen dafür die Dienste eines Domain Name Server in Anspruch oder werten lokale Files aus.

Datenbank-funktionen	Beschreibung
gethostbyaddr	Liefert den Namen eines Hosts, dessen IP-Adresse gegeben ist.
gethostbyname	Liefert die IP-Adresse eines Hosts, dessen Name gegeben ist.
gethostname	Liefert den Namen der lokalen Station.

1.7 Blockierende Funktionen

Alle Funktionen der Socket-Schnittstelle sind blockierende Funktionen. Bei Datenbank- oder Konvertierungsfunktionen wird man die blockierende Wirkung nicht spüren, da diese Funktionen immer sofort ein Ergebnis liefern. Ruft man aber z.B. die Funktion *recv()* auf, um Daten vom Socket zu empfangen, wird sie die Kontrolle erst zurückgeben, wenn tatsächlich Daten zum Empfang vorhanden sind.

Um solche blockierenden Wirkungen zu umgehen, ist es unerläßlich, vor jeder Lese- oder Schreibaktion die Funktion *select()* aufzurufen. Sie gibt für eine gegebene Menge von Sockets Auskunft, ob Daten versandt oder empfangen werden können.

Beachten Sie zu diesem Problem auch die Funktion *WSAAsynSelect()*, eine Windows-spezifische Version von *select()*.

1.8 Spezifische Funktionen der WinSock-Schnittstelle

Die WinSock-Schnittstelle hat einige Funktionen, die speziell an die Umgebung von Windows angepaßt sind.

Bevor der erste Aufruf einer Socket-Funktion erfolgt, muß die Nutzung der *WinSock.DLL* durch den gestarteten Prozeß initialisiert werden:

```
WSADATA wsadata;
WSAStartup(MAKEWORD(1,1), &wsadata);  //Version 1.1 required
```

Erst nach der Initialisierung mit *WSAStartup()* können andere Funktionen erfolgreich aufgerufen werden. *WSAStartup()* erlaubt es, die erforderliche Version der DLL zu spezifizieren und Details über die implementierte DLL in der Struktur *WSDATA* abzulegen. Analog muß vor dem Beenden des Prozesses auch die Arbeit mit der *WinSock.DLL* abgeschlossen werden. Die letzte aufzurufende Funktion ist also grundsätzlich *WSACleanup()*.

WinSock-spezif. Funktionen	Beschreibung
WSAStartup	Initialisierung der Windows-Sockets.
WSACleanup	Beenden der Arbeit mit den Sockets durch den aktuellen Prozeß.
WSAAsyncSelect	Asynchrone Version von *select ( )*, Socket im asynchronen Betrieb, Festlegen der Windows-Benachrichtigungen.
WSAGetLastError	Liefert den Fehlercode des letzten fehlerhaften Socket-Aufrufs.

Die Verwendung von *WSAAsyncSelect()* – der asynchronen Version von *select()* – bringt viele Vorteile. Diese Funktion erlaubt eine nichtblockierende Arbeit mit Sockets: Sie initialisiert die Benachrichtigung der Applikation über Windows-Messages, sobald Netzwerkereignisse für diesen Socket eintreten.

Die Funktion *WSAGetLastError()* liefert den letzten Fehlercode eines Socketaufrufs. Gibt ein Socket-Aufruf den Wert *SOCKET_ERROR* zurück, sollte diese Funktion unbedingt sofort aufgerufen werden.

1.9 Die wichtigsten Strukturen

Die Strukturen der Socket-Schnittstelle sehen auf den ersten Blick recht verwirrend aus. Bei genauerem Hinsehen wird allerdings klar, daß all diese Strukturen das gleiche bezeichnen – nur eben in unterschiedlicher Form.

```
/*  WINSOCK.H */
struct sockaddr
{ u_short sa_family;       // Address-Family (immer AF_INET)
  char sa_dat[14];         // Address
}
struct sockaddr_in
{ u_short sin_family;      // Address-Family (AF_INET)
  u_short sin_port;        // gewünschter Port
  struct in_addr sin_addr; // die IP-Adresse
  char sin_zero[ 8 ];      // Auffüllen der Struktur
}
```

Die Strukturen *sockaddr* und *sockaddr_in* haben den gleichen Inhalt: die Adressfamilie und die Adresse selbst. In das Array *sa_dat[14]* der Struktur *sockaddr* können Adressen aller Familien eingetragen werden. Die Adressfamilie ist im Internet immer *AF_INET*. Die Struktur *sockaddr_in* konkretisiert dieses Array auf das Format der Adressierung im Internet: auf die Portnummer und die IP-Adresse. Dafür benötigt man nur 6 Bytes – deshalb auch die 8 ungenutzten Bytes von *sin_zero[8]* am Ende der Struktur.

```
/*  WINSOCK.H */
struct in_addr
{ union
  { struct  { u_char  s_b1, s_b2, s_b3, s_b4; }
       S_un_b;
    struct  { u_short s_w1, s_w2;  } S_un_w;
    u_long S_addr;
  } S_un;
}
```

Die Struktur *in_addr* beinhaltet nichts weiter als die IP-Adresse selbst. Sie ermöglicht lediglich den Zugriff auf einzelne Bestandteile der IP-Adresse, ohne dabei komplizierte Casts

bilden zu müssen. Je nach Klasse des IP-Netzwerkes kann man mittels dieser Struktur auf Network- und Host-ID getrennt zugreifen.

```
/*  WINSOCK.H */
struct hostent
{ char FAR *h_name;             // String mit offiziellen Hostnamen
  char FAR* FAR* h_aliases;     // Zeiger auf Alternativnamen
  short h_addrtype;             // immer PF_INET
  short h_length;               // immer 4 (IP-Adresse)
  char FAR* FAR* h_addr_list;   // Zeiger auf Array mit IP-Adressen

  #define h_addr h_addr_list[0] // zum Zugriff auf die erste IP-Adresse
};
```

Die Struktur *hostent* ist wichtig für Datenbankfunktionen. In dieser Struktur findet man alle Zusatzinformationen zu einer übergebenen Information – z.B. Namen und weitere Alternativnamen zu einer gegebenen IP-Adresse oder alle IP-Adressen zu einem Namen. Im Normalfall gibt es nur eine Adresse zu einem Namen, lediglich bei Multi-Homed-Hosts kann man mehrere IP-Adressen zurückbekommen.

1.10 Streams und Datagramme

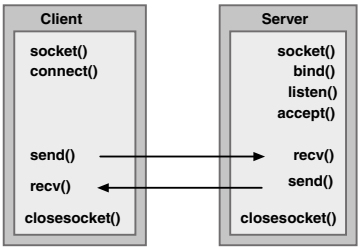
Beim Anlegen eines neuen Sockets muß man sich entscheiden, ob ein STREAM-Socket oder einen DATAGRAM-Socket initialisiert werden soll. Dahinter verbirgt sich die nicht ganz unerhebliche Entscheidung zwischen TCP und UDP. Je nach Applikation haben beide Protokolle ihre Vor- und Nachteile.

Der übliche Weg ist die Initialisierung eines STREAM-Sockets, d.h. die Verwendung von TCP. Damit ist man hinsichtlich der Sicherung und Kontrolle des Datenstroms alle Sorgen los. Bei schnell wechselnden Sendern und Empfängern müssen in diesem Fall allerdings auch ständig Verbindungen auf- und abgebaut werden oder viele Sockets initialisiert werden – was Zeit- und Verwaltungsaufwand bedeutet.

UDP ist schneller, bietet dafür aber keinerlei Sicherungsmechanismen. Für die Datenintegrität muß hier also anders gesorgt werden.

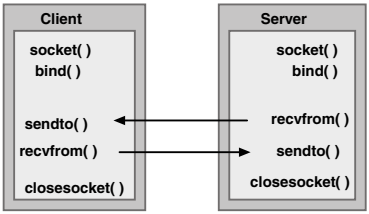
Die beiden Übersichten auf der folgenden Seite zeigen jeweils die Initialisierung des entsprechenden Protokolls mit dem Befehl *socket()* sowie die Befehlsfolge, die bei der Realisierung von Client- und Server-Applikationen verwendet werden muß.

Stream-Clients und -Server (TCP)




```
SOCKET iClient;
iClient = socket(AF_INET, SOCK_STREAM, 0);
if(iClient == INVALID_SOCKET)
    {   // Fehler
        int errcode = WSAGetLastError();
    }
```

Datagramm-Clients und -Server (Protocol UDP)




```
SOCKET iClient;
iClient = socket(AF_INET, SOCK_DGRAM, 0);
if(iClient == INVALID_SOCKET)
  {   // Fehler
        int errcode = WSAGetLastError();
  }
```

W&T
www.WuT.de

Teil 2
Programmbeispiele Socket-API

DOS-Applikationen in C

Windows-95-Applikationen in C

Visual Basic

Java

Diese kleine Programmier-Referenz widmet sich speziell dem Protokoll TCP/IP. Sie finden hier eine kleine Auswahl an Beispiel-Programmen für verschiedene Umgebungen (Windows95/ C, DOS/C, JAVA und VISUAL-Basic), die beim Erstellen einer eigenen Applikation den Start erleichtern sollen. Sie werden sehen: die Socket-Schnittstelle ist wirklich leicht zu handhaben. Um einen ersten Erfolg zu erzielen – um also Daten zum Com-Server zu senden oder von dort zu empfangen – bedarf es nur weniger Funktions-aufrufe.

Alle Beispiele finden Sie im Internet unter *http://www.WuT.de* als Download.

2.1 Anwendungsumfeld C: DOS

Beschreibung der Programmierumgebung

Die Beispiele dieses Anleitungsteils sollen knapp darstellen, wie man Anwendungen für den Com-Server mit Hilfe der Socket-Schnittstelle für das Betriebssystem DOS erzeugt. Die Programme sind unter DOS oder in der DOS-Box von Windows ausführbar und wurden für den TCP/IP-Stack von Novell (LAN Work Place V4.1) erstellt.

Programmierumgebung des Beispiels:

Programmiersprache:	C
Compiler:	Microsoft C/C++ Compiler Version 8.0
Linker:	Microsoft Segmented Executable Linker Version 5.50
Socket API:	Novell's LAN WorkPlace Windows Sockets Application Rrogramming Interface (API)

In den Beispielen wurde das Modul *LLIBSOCK.LIB* für Large-Model-DOS-Library-Funktionen eingebunden.

2.1.1 Programmbeispiel: Socket Client

Das Programm *tcpclnt.c* realisiert einen TCP-Client. Beim Aufruf des Programms wird die IP-Adresse des Com-Servers in Dot-Notation (z.B. 190.107.231.1) oder der Name des Com-Servers als Argument angegeben.

Das Programm baut eine Verbindung zu Port A des gewünschten Com-Servers auf, gibt alle empfangenen Daten auf dem Bildschirm aus und sendet alle Tastatureingaben nach dem Drücken der Entertaste an den Com-Server Port A.

Die Funktion *terminal()* realisiert die Funktionalität eines Terminals (Dateneingabe und -ausgabe).

```
/***********************************************
***   tcpclnt.c                                 ***
***   TCP Client Programm: Terminal Funktion    ***
***   Beenden des Programms mit ALT Q           ***
***********************************************/
#include <stdio.h>
#include <conio.h>
#include <nw/socket.h>
#define TCP_PORT_A 8000
#define BUF_SIZE 512
char SendBuf[BUF_SIZE];
char RecBuf [BUF_SIZE];

void terminal(int);

void main (int argc,char **argv)
  {
    int sd;                          //Socket-Deskriptor
    struct sockaddr_in box;          //Adresse des Com-Servers
    int portnr = 8000;               //Com-Server-Port A vordefinieren
    char *hostname;
```

```
    u_long remote_ip;

    if(!loaded())                    //TCP/IP-Stack installiert?
      { printf("Kein TCP/IP Protokollstack aktiv\n");
        exit (1);
      }
    if(argc < 2)                     //Hostname aus Argument übernehmen
      { printf("Kein Hostname angegeben\n");
        exit (1);
      }
    if(argc > 2)                     //evtl. Portnummer als 2. Parameter
      portnr = atoi(argv[2]);

    bzero((char *)&box,sizeof(box)); //Adreßstruktur löschen
    hostname = argv[1];              //rhost() erwartet (char**)-Argument!
    if((remote_ip = rhost(&hostname)) == -1)
      { printf("Unbekannter Hostname\" %s\"\n", argv[1]);
        exit(1);
      }

    //öffne Handle für TCP-Transport
    if((sd = socket(PF_INET,SOCK_STREAM, 0)) < 0)
      { soperror("socket");
        exit(1);
      }

    box.sin_family = AF_INET;
    box.sin_port = htons(portnr);           //Ziel Port-Nummer
    box.sin_addr.s_addr = rhost(&hostname); //Ziel IP-Adresse

    //öffne Verbindung zum COM-Server Port A
    if(connect (sd, (struct sockaddr*)&box,
      sizeof(box)) < 0)
      { soclose(sd);                 //schließe Handle wieder
        soperror("connect");
        exit(1);
      }

    //empfange u. sende Daten,bis ALT Q gedrückt wird
    printf("Verbunden mit COM-Server %s:%d\n", inet_ntoa(box.sin_addr),
             htons(box.sin_port));

    terminal();
    soclose(sd);                     //schließe Handle wieder
  }

void terminal(void)
  {
    fd_set rd_ready,wr_ready;        //Bitfelder pro Socket-Deskriptor
    struct timeval maxwait;          //Max. Wartezeit für select()
    int s_len = 0;
    int r_len = 0;
    int z_count = 0;
    char key;
    for(;;)
```

```
  {
    FD_ZERO(&rd_ready);           //das API soll uns max.10µs warten lassen.
    FD_SET (sd,&rd_ready);        //select() liefert die Anzahl aktiver
    FD_ZERO(&wr_ready);           //Verbindungen und setzt für jede aktive
    FD_SET (sd,&wr_ready);        //Verbindung ein Bit in den übergebenen
    maxwait.tv_sec = 0;           //Bitfeldern. Return 0: keine Daten
    maxwait.tv_usec = 10;         //empfangen und kein Senden möglich

    if(select(sd+1,&rd_ready,&wr_ready,
        (fd_set*)0,&maxwait) == 0) continue;
    if(FD_ISSET(sd, &rd_ready))  //liegen Daten vor?
      { if((r_len=soread(sd, RecBuf, BUF_SIZE))>0)
        { *(RecBuf[r_len]) = 0   //markiere Stringende
          printf("%s",RecBuf);   //gebe Daten auf Bildschirm aus
        }
        else if(r_len == 0)      //regulärer Verbindungsabbau
          { printf("\nCOM-Server hat die Verbindung beendet\n");
            return;
          }
        else if(r_len <= 0)      //Verbindungsfehler
        { soperror("soread");
          return;
        }
      }

    if(kbhit())                  //Tastatureingaben lesen
      { key = getch();
        if(!key)                 // Sonderzeichen
          { if(getch()==16)      // ALT Q -> beende Terminal
            { printf("\n");
              return;
            }
          }
        else
          { SendBuf[z_count++] = key;
            if(key==0x0D)        //ENTER->sende Zeile an Com-Server
            { SendBuf[z_count++] = 0x0A;
              s_len = z_count;
              z_count = 0;
            }
          }
      }
    //sobald eine Eingabezeile komplett ist (und  das API bereit ist),
    //Zeile absenden:
    if(s_len > 0 && FD_ISSET(sd, &wr_ready))
  { if(sowrite(sd, SendBuf, s_len) < 0)
    { soperror("sowrite()");
      return;
    }
    s_len = 0;
  }
 }
}
```

2.1.2 Programmbeispiel: Socket Server

Das Programm *tcpserv.c* realisiert einen TCP-Server auf dem Socket 2000. Der Com-Server wird hier im „Socket Client Mode“ betrieben: Liegen an der seriellen Schnittstelle Daten an, baut der Com-Server eine Verbindung zu einem Server-Programm auf und übermittelt ihm die Daten zur Verarbeitung.

Dieses Programm gibt alle empfangenen Daten auf dem Bildschirm aus und sendet alle Tastatureingaben nach dem Drücken der Entertaste an den entsprechenden Port des Com-Servers.

Die Funktion *terminal()* zur Realisation der Funktionalität eines Terminals wurde bereits im vorangegangenen Abschnitt dargestellt.

```
/***********************************************
***   tcpserv.c                                 ***
***   TCP Server-Programm: Terminal-Funktion    ***
***   Schließen einer Verbindung mit ALT Q      ***
***   Beenden des Server-Modes mit ESC          ***
***********************************************/

#include    <stdio.h>
#include    <conio.h>
#include    <nw/socket.h>
#define SERV_SOCKET 2000                //Server Port
#define BUF_SIZE 512

char SendBuf[BUF_SIZE];
char RecBuf [BUF_SIZE];
void terminal(int sd);

void main (void)
  {
    int sd,ss;                          //Socket-Deskriptoren
    struct sockaddr_in box,loc;         //Adressen von Com-Server und PC
    int box_size = sizeof(box);
    fd_set rd_ready;                    //Flags
    struct timeval maxwait;             //max. Wartezeit für select()
    bzero((char *)&loc, sizeof(loc));   //Adreßstrukturen löschen
    bzero((char *)&box, sizeof(box));
    if(!loaded())                       //TCP/IP-Stack installiert?
      { printf("Kein TCP/IP Protokoll-Stack aktiv\n");
        exit (1);
      }

    // öffne Handle für TCP-Transport
    if((ss = socket(PF_INET, SOCK_STREAM, 0)) < 0)
      { soperror("socket()");
         exit(1);
      }
    loc.sin_family = AF_INET;
    loc.sin_port = htons(SERV_SOCKET);
    loc.sin_addr.s_addr = getmyipaddr();  //"0" wäre auch zulässig
```

```
  // Socket ss an einen "Namen" binden (IP-Adresse und Port)
  if(bind (ss, (struct sockaddr*)&loc, sizeof(loc)) < 0)
    { soclose(ss);                  //Handle wieder schließen
      soperror("bind()");
      exit(1);
    }

  if(listen (ss, 1) < 0)            //akzeptiere Verbindungsanfragen,
    {                               //queue limit=1
      soclose(ss);
      soperror("listen()");
      exit(1);
    }
  printf("Server bereit: %s:%d\n",
              inet_ntoa(loc.sin_addr),
              htons(loc.sin_port));

  for(;;)                           //warte auf Verbindungen
    {
      if(kbhit() && getch()==27)
        break;                      //ESC beendet den Server-Mode

      FD_ZERO(&rd_ready); FD_SET(ss, &rd_ready);
      maxwait.tv_sec = 0; maxwait.tv_usec = 10;

      //beim API nachfragen, ob Daten empfangen wurden
      if(select(ss+1, &rd_ready, NULL, NULL, &maxwait) == 0)
        continue;
      //Akzeptiere Verbindung und speichere Client-Adresse in der Struktur "box"
      if((sd = accept(ss, (struct sockaddr*)&box, &box_size)) < 0)
        { soclose(ss);
          soperror("accept()");
          exit(1);
        }

      printf("Verbindung von %s:%d angenommen.\n",
             inet_ntoa(box.sin_addr),htons(box.sin_port));
      //rufe Terminal-Funktion zum Datenaustausch, bis der Client die Verbindung
      //beendet oder 'ALT Q' gedrückt wird (-> Funktion "Terminal" in Bsp. 2.2.2)
      terminal(sd);
      soclose(sd);                  //schließe Verbindung zum Client
      printf("Socket geschlossen\n");
    }                               //warte auf den nächsten Serveraufruf
  soclose(ss);                      //disable Server-Socket
}
```

2.1.3 Programmbeispiel: UDP-Server

Das Programm *udpserv.c* realisiert einen UDP-Server auf dem Socket 2000. Der Com-Server wird im „UDP Mode“ konfiguriert und sendet alle seriellen Daten an diesen UDP-Server.

UDP bietet keinerlei Verbindungskontrolle. Mit UDP sollte man demnach nur arbeiten, wenn die Daten zwischen dem seriellen Endgerät und Ihrer Endanwendung bereits mit einem Protokoll übertragen werden, das seinerseits einen fehlerfreien Datentransfer gewährleistet.

Das Programm *udpserv.c* gibt alle empfangenen Daten auf dem Bildschirm aus und sendet sämtliche Tastatureingaben nach dem Drücken der Entertaste an den Com-Server-Port, von dem zuletzt Daten empfangen wurden.

```
/*********************************************
***   udpserv.c                               ***
***   UDP Server-Programm: Terminal-Funktion  ***
***   Beenden des Programms mit ALT Q         ***
*********************************************/
#include    <stdio.h>
#include    <conio.h>
#include    <nw/socket.h>
#define SERV_SOCKET 2000                //Server-Port
#define BUF_SIZE 512

char SendBuf[BUF_SIZE];
char RecBuf [BUF_SIZE];

void main (void)
  {
    int sd;                             //Socket-Deskriptor
    struct sockaddr_in box;             //Struktur Com-Server-Port
    struct sockaddr_in loc;             //Struktur PC
    struct timeval maxwait;             //Wartezeit auf API
    fd_set rd_ready,wr_ready;           //Flags für select()
    int s_len = 0;
    int r_len = 0;
    int z_count = 0;
    int boxlen = sizeof(box);
    char key;

    bzero((char *)&loc, sizeof(loc));   //Adreßstrukturen löschen
    bzero((char *)&box, sizeof(box));
    if(!loaded())                       //TCP/IP-Stack installiert?
      { printf("Kein TCP/IP Protokoll-Stack aktiv\n");
        exit (1);
      }
    //öffne Handle für UDP-Transport
    if((sd = socket(PF_INET, SOCK_DGRAM, 0)) < 0)
      { soperror("socket()");
        exit(1);
      }
    loc.sin_family = AF_INET;
    loc.sin_port = htons(SERV_SOCKET);
```

```
loc.sin_addr.s_addr = getmyipaddr();  //"0" wäre auch zulässig
box.sin_family = AF_INET;
//Socket sd an einen "Namen" binden (IP-Adresse und Port):
if(bind (sd, (struct sockaddr*)&loc, sizeof(loc)) < 0)
  { soclose(sd);                      //Handle wieder schließen
    soperror("bind()");
    exit(1);
  }
printf("UDP-Server bereit: %s:%d\n", inet_ntoa(loc.sin_addr),
       htons(loc.sin_port));
for(;;)                               //warte auf Daten
  { FD_ZERO(&rd_ready); FD_SET(sd, &rd_ready);
    FD_ZERO(&wr_ready); FD_SET(sd, &wr_ready);
    maxwait.tv_sec = 0;
    maxwait.tv_usec = 10;             //das API soll max. 10µs blockieren

    //select() liefert die Anzahl aktiver Verbindungen und setzt für jede
    //aktive Verbindung ein Bit in den übergebenen Bitfeldern.
    //Rückgabewert 0: keine Daten empfangen, und es können zur Zeit auch keine
    //Daten gesendet werden.

    if(select(sd+1, &rd_ready, &wr_ready, NULL, &maxwait) == 0)
    continue;

    //empfange Daten und speichere Absender in Structur "box"
    if(FD_ISSET(sd, &rd_ready))
      { if((r_len = recvfrom(sd, RecBuf, BUF_SIZE, 0, (struct sockaddr*)&box,
            &boxlen)) > 0)
          { RecBuf[r_len] = 0;        //markiere Stringende
            printf("Daten von %s:%d: %s\n", inet_ntoa(box.sin_addr),
                    htons(box.sin_port), RecBuf);
          }
        else                          //Verbindungsfehler
          { soperror("recvfrom()");
            goto quit;
          }
      }

    if(kbhit())                       //lese Tastatureingaben
      { key = getch();
        if(!key)                      //Sonderzeichen
          { if((char)getch()==16)     //ALT Q -> beende Terminal
              goto quit;
          }
        else if(box.sin_port)         //Com-Server-Port bekannt?
          { SendBuf[z_count++] = key;
              if(key==0x0D)           //ENTER -> sende Zeile
                { SendBuf[z_count++] = 0x0A;
                  s_len = z_count;
                  z_count = 0;
                }
          }
        else
          printf("An wen senden?\n");
      }
```

```
      //Wurden Zeichen von der Tastatur gelesen und ist das API zum Senden bereit?

      if(s_len > 0 && FD_ISSET(sd, &wr_ready))
        { if(sendto(sd, SendBuf, s_len, 0,
              (struct sockaddr*)&box, sizeof(box)) < 0)
            { soperror("sendto()");
              goto quit;
            }
            s_len = 0;
                                          //setze Zeichenanzahl auf Null
        }
    }                                     //Ende for(;;)
  quit:
  soclose(sd);
  exit(1);
}
```

2.2 Anwendungsumfeld C:Windows 9x/NT

Beschreibung der Programmierumgebung

Diese Applikation für Windows 9x oder Windows NT realisiert einen TCP-Client und ermöglicht den Datenaustausch mit dem Com-Server. Dieses Beispiel zeigt das Handling der Socket-Schnittstelle mit Hilfe von Windows-Messages.

Systemvoraussetzungen:

- Microsoft Windows 9x oder Windows NT
- Microsoft Visual C++ 5.0 oder höher
- Microsoft Windows TCP/IP-Stack (32 bit)

Programmierumgebung des Beispiels:

Programmiersprache: C

Compiler: 32 bit edition of Visual C/C++ 5.0

```
/*************************************************************
*                                                            *
*       clnt_tcp.c   (Win32 Application)                     *
*                    Microsoft Visual C++ 5.0                *
*                                                            *
*************************************************************
* TCP-Client: Der Client öffnet die Verbindung zum TCP-Server,  *
* dessen Adresse oder Name in der Dialogbox eingetragen wird.   *
* Alle im Feld ´Senden´ eingegebenen Daten werden zum Server    *
* übertragen, alle empfangenen Daten im Fenster ´Empfang´       *
* ausgegeben. Im Statusfenster werden die gerade ausgeführten   *
* WINSOCK-Funktionen angezeigt.                                 *
*****************************************************************/

#include <winsock.h>                //bindet auch windows.h ein!
#include <stdio.h>

#include "resource.h"               //Konstanten der Dialogfelder
#define WM_SOCKET (WM_USER + 1)     //private Windows-Nachrichten
#define SERVER_PORT 8000            //Com-Server-Port A

SOCKET iClient = INVALID_SOCKET;

/* Statusmeldungen protokollieren */
void ShowStatus(HWND hWnd, LPSTR lpMsg)
{
  int iEntries;
  //Neuen Eintrag hinzufügen
  SendMessage(GetDlgItem(hWnd, IDC_STATUS),
              LB_ADDSTRING, (WPARAM)-1, (LPARAM)lpMsg);
  //letzten Eintrag anzeigen
  iEntries = SendMessage(GetDlgItem(hWnd,IDC_STATUS), LB_GETCOUNT, 0, 0);
  SendMessage(GetDlgItem(hWnd, IDC_STATUS), LB_SETTOPINDEX, iEntries-1, 0);
}
```

```
/* Dialogprozedur des Hauptdialogs */

BOOL WINAPI WSClientProc(HWND hWnd, UINT msg, WPARAM wP, LPARAM lP)
{
  switch(msg)
  {
    case WM_INITDIALOG:               //Dialogfenster initialisiert
    SetWindowText(GetDlgItem(hWnd, IDC_DESTADDRESS), "box");
    break;

    case WM_SOCKET:                   //WINSOCK-Messages
    {
      switch(WSAGETSELECTEVENT(lP))
      {
        case FD_CONNECT:              //Nachricht von connect()
          ShowStatus(hWnd, "FD_CONNECT");
          if(WSAGETSELECTERROR(lP) == 0)
          {
            EnableWindow(GetDlgItem(hWnd, IDC_CLOSE), TRUE);
            EnableWindow(GetDlgItem(hWnd, IDC_CONNECT), FALSE);
            EnableWindow(GetDlgItem(hWnd, IDC_SEND), TRUE);
          }
          else
          { closesocket(iClient);
            iClient = INVALID_SOCKET;
            ShowStatus(hWnd, "Kein Server!");
            EnableWindow(GetDlgItem(hWnd, IDC_CONNECT), TRUE);
            EnableWindow(GetDlgItem(hWnd, IDC_CLOSE), FALSE);
            EnableWindow(GetDlgItem(hWnd, IDC_SEND), FALSE);
          }
        break;

        case FD_READ:                 //Daten empfangen
        { char rd_data[255];
          int  iReadLen;              //Daten einlesen
          ShowStatus(hWnd, "FD_READ");
          iReadLen = recv(iClient, rd_data, sizeof(rd_data)-1, 0);
          if(iReadLen > 0)
          { rd_data[iReadLen] = 0;
            SendMessage(GetDlgItem(hWnd, IDC_RECEIVE), LB_ADDSTRING,
                        (WPARAM)-1, (LPARAM)rd_data);
          }
        }
        break;

        case FD_CLOSE:                //Verbindungsabbruch
          ShowStatus(hWnd, "FD_CLOSE");
          EnableWindow(GetDlgItem(hWnd, IDC_SEND), FALSE);
          EnableWindow(GetDlgItem(hWnd, IDC_CONNECT), TRUE);
          EnableWindow(GetDlgItem(hWnd, IDC_CLOSE), FALSE);
          closesocket(iClient);
          iClient = INVALID_SOCKET;
        break;
      }
    }
    break;
```

```
case WM_COMMAND:                        //Schaltflächen-Nachrichten
  switch(LOWORD(wP))
  { case IDCANCEL:                      //CloseBox des Fensters
      EndDialog(hWnd, 0);
    break;

    case IDC_SEND:                      //Daten senden
      if(iClient != INVALID_SOCKET)
      {
        char Buffer[255];
        int iSendLen;
        ShowStatus(hWnd, "FD_WRITE"); //Serveradresse einlesen
        iSendLen = GetWindowText(GetDlgItem (hWnd,IDC_SENDDATA),
                                 Buffer, sizeof(Buffer));
        ShowStatus(hWnd, "send()...");

        if(send(iClient, Buffer, iSendLen,0) != SOCKET_ERROR)
        { ShowStatus(hWnd, "... fertig");
          EnableWindow(GetDlgItem(hWnd, IDC_SEND), TRUE);
        }
        else
        { if(WSAGetLastError() == WSAEWOULDBLOCK)
            ShowStatus(hWnd, "... blockiert");
          else
            ShowStatus(hWnd, "Fehler bei send()");

          ShowStatus(hWnd, "closesocket()");
          EnableWindow(GetDlgItem(hWnd, IDC_SEND), TRUE);
          closesocket(iClient);
          iClient = INVALID_SOCKET;
          ShowStatus(hWnd, "Abbruch");
      }
    }
    break;

    case IDC_CONNECT:
    { SOCKADDR_IN sin;
      char remoteIP[64];
      char Buffer[80];

      //Zieladresse auslesen
      GetWindowText(GetDlgItem(hWnd, IDC_DESTADDRESS),
                    remoteIP, sizeof(remoteIP));
      memset(&sin, 0, sizeof(sin));
      sin.sin_family = AF_INET;
      sin.sin_port = htons(SERVER_PORT);
      //IP-Adresse -> Dot-Notation
      ShowStatus(hWnd, "inet_addr()");
      sin.sin_addr.s_addr = inet_addr(remoteIP);
      //Adresse über DNS auflösen
      if(sin.sin_addr.s_addr == INADDR_NONE)
      { HOSTENT *he;
        ShowStatus(hWnd, "gethostbyname()");
        he = gethostbyname(remoteIP);
        if(he)
          sin.sin_addr.s_addr = *((DWORD*)he->h_addr);
```

```
      else
      { ShowStatus(hWnd, "ungülitige Internet-Adresse");
        break;
      }
    }

    //Zieladresse protokollieren
    wsprintf(Buffer, "Adresse: 0x%08lx", ntohl(sin.sin_addr.s_addr));
    ShowStatus(hWnd, Buffer);

    //Socket erzeugen
    ShowStatus(hWnd, "socket()");
    iClient = socket(AF_INET, SOCK_STREAM, 0);
    if(iClient == INVALID_SOCKET)
    { ShowStatus(hWnd, "Fehler beim Allokieren des Connect-Sockets");
      ShowStatus(hWnd, "Kein Verbindungsaufbau möglich");
      break;
    }

    //asynchronen Mode aktivieren
    ShowStatus(hWnd, "WSAAsyncSelect()");
    if(WSAAsyncSelect(iClient,
              hWnd,
              WM_SOCKET,
              FD_CONNECT |
              FD_READ |
              FD_CLOSE) == 0)

    { ShowStatus(hWnd, "connect()");
      if(connect(iClient,(SOCKADDR*)&sin,sizeof(sin)) == SOCKET_ERROR)
      { if(WSAGetLastError() == WSAEWOULDBLOCK)
        { ShowStatus(hWnd, "Warte...");
          //Senden-Schalfläche deaktivieren
          EnableWindow(GetDlgItem(hWnd, IDC_SEND),FALSE);
          break;
        }
      }
    }
    else
      ShowStatus(hWnd, "Fehler bei WSAAsyncSelect()");
    ShowStatus(hWnd,"closesocket()");
    closesocket(iClient);
    iClient = INVALID_SOCKET;
  }
  break;

  case IDC_CLOSE:
    ShowStatus(hWnd,"closesocket()");
    closesocket(iClient);
    iClient = INVALID_SOCKET;
    EnableWindow(GetDlgItem(hWnd, IDC_CLOSE), FALSE);
    EnableWindow(GetDlgItem(hWnd, IDC_CONNECT), TRUE);
    EnableWindow(GetDlgItem(hWnd, IDC_SEND), FALSE);
    break;
}
break;                          //end "case WM_COMMAND"
```

```
    case WM_DESTROY:                    //Fenster schließen
      if(iClient != INVALID_SOCKET)
        closesocket(iClient);           //Socket schließen
    break;
    }
  return FALSE;
}

/***********************************************
* WinMain:  Haupteinsprungspunkt               *
***********************************************
* Parameter:  Standarddialogparameter          *
* Rückgabewert: 0                              *
***********************************************/

int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nShowCmd)
{
  WSADATA wsadata;                      //Version 1.1 der Winsock-DLL

  if(WSAStartup(MAKEWORD(1,1), &wsadata) == 0)
  {
    DialogBox(hInstance,                //Dialog anzeigen
              MAKEINTRESOURCE(IDD_WSCLIENT),
              NULL,
              WSClientProc);
    WSACleanup();                       //Winsock-DLL aufräumen
  }
  else MessageBox(0,
         "Fehler beim Initialisieren der WINSOCK.DLL",
         "WSClient.EXE", 0);
  return 0;
}
```

2.3 Anwendungsumfeld Visual-Basic

Beschreibung der Programmierumgebung

Dieses Beispiel zeigt, wie man eine TCP-Client-Anwendung für den Com-Server mit Hilfe der Socket-Schnittstelle in Visual Basic erstellt. Es wird das Winsock-Steuerelement von Microsoft verwendet, daß ab der Visual Basic Version 5 zum Standardlieferumfang gehört.

Systemvoraussetzungen:

- Microsoft Windows 95, 98 oder NT 4.0
- Visual Basic 5.0 oder höher
- Microsoft Windows TCP/IP-Stack

Programmierumgebung des Beispiels:

Programmiersprache:	Visual Basic
Compiler:	Visual Basic 5.0 (32 bit)
TCP/IP-Control:	Microsoft Winsock Control 5.0

2.3.1 Einbindung des Winsock Controls in Visual Basic Projekt

Das Microsoft Winsock Control ermöglicht die Kommunikation über die Protokolle TCP/IP bzw. UDP/IP, wobei unter TCP sowohl Client- als auch Server-Applikationen möglich sind.

Das Winsock Control muß dem zu erstellenden VB-Projekt zunächst als neue Komponente zugefügt werden. Über *Projekte->Komponenten* ... gelangt man in die Auswahl aller optional zufügbaren Steuerelemente. Markieren Sie hier den Eintrag „Microsoft Winsock Control 5.0“ und schließen Sie die Auswahl mit *OK*. Mit dem in der Werkzeugsammlung zur Verfügung stehenden Winsock-Symbol kann das Steuerelement jetzt dem Projekt hinzugefügt werden. Als Defaultnamen werden *Winsock1, Winsock2* usw. vergeben.

Nach diesem Schritt steht der Kommunikationsweg über das TCP/IP Protokoll dem neuen Programm zur Verfügung. Anhand des folgenden kleinen TCP-Client Programms werden die grundsätzlich notwendigen Schritte und Funktionen zum Verbindungsauf- und abbau sowie zum Senden und Empfangen von Nutzdaten erläutert.

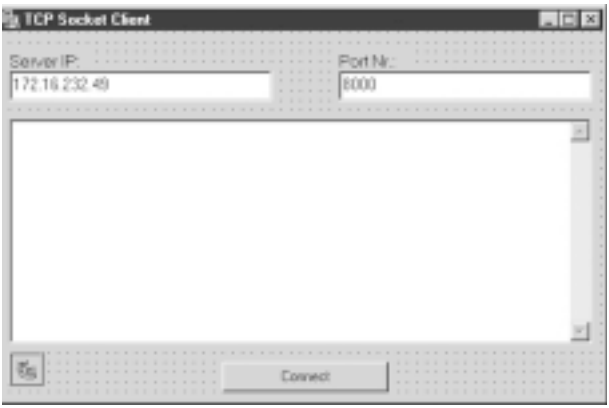
Eine Beschreibung aller Eigenschaften, Methoden und Ereignisse des Winsock-Controls ist über die Online-Hilfe von Visual Basic verfügbar. Markieren Sie hierzu das Control in Ihrem Projekt und betätigen Sie die Taste *F1*.

2.3.2 Erläuterung des Programmbeispiels (TCP Socket Client)

Das Programm realisiert einen TCP-Client der eine Verbindung zu dem in den Textfeldern angebenen TCP-Server aufbaut. Anschließend werden alle eingegebenen Zeichen an den Server gesendet, bzw. vom Netzwerk eingehende Zeichen im Textfenster dargestellt.

***Hinweis**: Das nachfolgende Demoprogramm dient lediglich zur Verdeutlichung der grundsätzlichen Struktur von TCP-Client-Anwendungen (Verbindungsaufbau -> Datenaustausch -> Verbindungsabbau). Besonders hinsichtlich eines kontrollierten Datenversandes über*

das Ereignis „Send-Complete“ sowie der Fehlerbehandlung müssen eigene Programme ggf. erweitert werden.



Button-Auswertung „Connect“ und Verbindungsauf-/abbau

```
Private Sub connect_Click()
  If Winsock1.State = sckClosed Then        'Wenn keine Verbindung besteht ...
    Winsock1.RemoteHost = IP_Nr.Text        'festlegen der Ziel IP-Adresse
    Winsock1.RemotePort = Val(Port_Nr.Text) 'festlegen der Ziel Port-Nr.
    Connect.Enabled = False                 'deaktivieren des Connect-Buttons
    TCPSocketCLIENT.MousePointer = 11       'Mauszeiger = Sanduhr
    Winsock1.Connect                        'Öffnen der Verbindung
  Else                                      'Wenn schon eine Verbindung besteht,
Winsock1.Close                              'schließen der Verbindung
    Winsock1.LocalPort = 0                  'lokale Port-Nr. auf 0 setzen
    Connect.Caption = „Connect“
  End If
End Sub
```

Ereignis „Connect“, Verbindung zum Server wurde erfolgreich aufgebaut

```
Private Sub Winsock1_Connect()      'Verbindung wurde erfolgreich aufgebaut
Terminal.SetFocus
  Connect.Caption = „Disconnect“
  Connect.Enabled = True            'aktivieren des Connect-Button
  TCPSocketCLIENT.MousePointer = 0  'Mauszeiger auf Standard setzen
End  Sub
```

Zeichen über das Netzwerk an TCP-Server senden

```
Private Sub terminal_KeyPress(KeyAscii As Integer)
  If Winsock1.State = sckConnected Then   'Wenn eine Verbindung besteht ...
    Winsock1.SendData Chr$(KeyAscii)      'Zeichen senden
    KeyAscii = 0
  End If
End Sub
```

Zeichen über das Netzwerk vom TCP-Server empfangen

```
Private Sub Winsock1_DataArrival(ByVal bytesTotal As Long)
                                    'Ereignis „Daten empfangen“
  Winsock1.GetData strData$         'Einlesen in String-Variable
  variableTerminal.Text = Terminal.Text + strData$
                                    'Anzeige im Terminalfenster
End Sub
```

Fehlerbehandlung und Ausgabe des zurückgegebenen Fehlertextes

```
Private Sub Winsock1_Error(ByVal Number As Integer, Description As String, _
                           ByVal Scode As Long, ByVal Source As String, _
                           ByVal HelpFile As String, ByVal HelpContext As Long, _
                           CancelDisplay As Boolean)
  MsgBox Description                'Anzeige Message-Box mit Fehlerstring
  Winsock1.Close                    'TCP-Verbindung schließen
  Connect.Enabled = True            'Aktivieren des Connect-Buttons
  TCPSocketCLIENT.MousePointer = 0  'Mauszeiger auf Standard setzen
End Sub
```

2.4 Anwendungsumfeld Java

Beschreibung der Programmierumgebung

Dieses Beispiel zeigt, wie eine einfache Java-Anwendung für den Com-Server aufgebaut ist. Das Programm ist unter Windows in der DOS-Box mit Hilfe eines Java-Interpreters ausführbar. Voraussetzung ist, daß der Microsoft Windows TCP/IP- Stack installiert ist.

Programmierumgebung des Beispiels:

System:	Windows 9x
TCP/IP-Stack:	Microsoft Windows TCPIP-Stack
Programmiersprache:	Java 1.3
Compiler:	Borland JBuilder 4.0

```
/*****************************************************************/
/*  ComportTcp.java                 Win32 Application            */
/*                                  Borland JBuilder 4.0         */
/*****************************************************************/
/*  Beispielprogramm für einen Client                            */
/*  Die IP-Adresse des Servers muß beim Start des Programms als  */
/*  Parameter übergeben werden.                                  */
/*  - Eingaben müssen mit <RETURN> beendet werden.               */
/*  - Empfangene Daten werden byte-weise ausgegeben.             */
/*  - Der Empfang wird von einem eigenen Thread bearbeitet.      */
/*  - Das Programm wird mit x+<ENTER> beendet.                   */
/*****************************************************************/

//Java Library Packages
import java.io.*;      //classes for file I/O
import java.net.*;     //classes to perform low-level Internet I/O

class ComportTcp
  { public static void main( String[] args )
    { try
      { if( args.length < 1 )
        { System.out.println( "Call: ComportTcp <IP-Address>" );
          return;
        }
        String strAddress = args[0];     // COM-Server IP-Address
        int iPort = 8000;                // COM-Server Port A (TCP)

        System.out.println( "Server IP-Address: " + strAddress );

        Socket         socket   = new Socket( strAddress, iPort );
        DataInputStream  incoming = new DataInputStream( socket.getInputStream() );
        DataOutputStream outgoing = new DataOutputStream( socket.getOutputStream() );

        System.out.println( "" );
        System.out.println( "Start receive thread!     End Session: x+<ENTER>" );
        System.out.println( "-------------------------------------------------" );
```

```
        new ThreadedReadStreamHandler( incoming ).start();

        boolean more = true;
        while( more )
          { int i = System.in.read();       //Zeichen von Stdeing. holen
            if( i != 0x78 )
              { outgoing.write( i );        //und zum ComPort senden
              }
            else
              { more = false;               //Bei ESC beenden
              }
          }
        socket.close();
        outgoing.close();
        System.out.println( "Exit" );
      }
      catch( IOException e )
      { System.out.println( "Error" + e );
      }
    }
  }

class ThreadedReadStreamHandler extends Thread
  { DataInputStream incoming;
    boolean more = true;
    ThreadedReadStreamHandler( DataInputStream i )
    { incoming = i;
    }

    public void run()
    { try
      { byte[] b = new byte[1];
        while( more )
        { b[0] = incoming.readByte();       // empfange Byte aus dem Stream
          System.out.write( b, 0, 1 );      // sende Byte an die Standardausgabe
        }
        incoming.close();
      }
      catch( Exception e )
      {
      }
    }
  }
```

Teil 3
TCP/IP-Grundlagen

Das Internet-Protokoll
Die IP-Adresse
Netzwerkklassen
Routing
Subnets
Die Protokolle UDP und TCP

3.1 IP – Internet Protocol

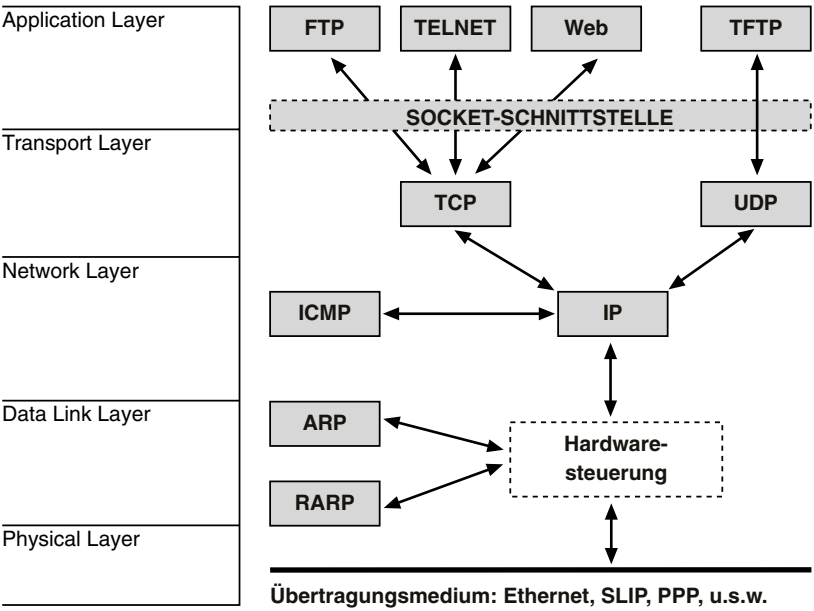
Das Internet-Protokoll definiert die Grundlage der Datenkommunikation auf der untersten Ebene. Es ermöglicht unabhängig von den verwendeten physikalischen Medien das Zusammenfügen vieler unterschiedlicher Netzwerk- und Hardwarearchitekturen zu einem einheitlichen Netz.

Das Internet-Protokoll gewährleistet die Übertragung der Daten durch einen verbindungslosen, nicht abgesicherten Transport. Für Sicherheitsmechanismen sind übergeordnet Protokolle wie TCP verantwortlich.

Grundlagen für netzübergreifende Verständigung:

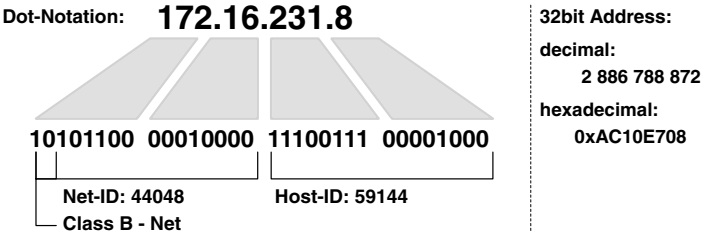
- Adressierungsmechanismus, um Sender und Empfänger eindeutig zu benennen
- Konzept zum Transport der Datenpakete über Knotenpunkte (Routing)
- Format für Datenaustausch (definierter Header mit wichtigen Informationen)

3.1.1 Die Protokollschichten des Internet

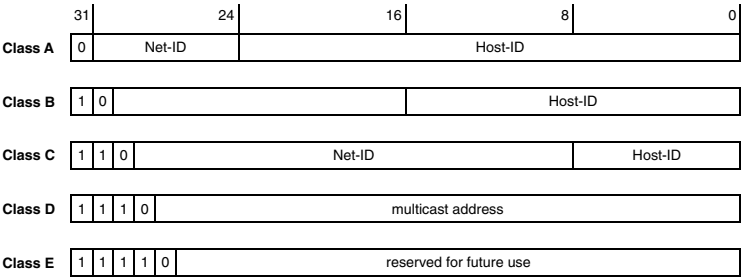


3.1.2 Internetadressen

Jeder Host im Internet hat eine weltweit einmalige Adresse. Diese IP-Adresse ist ein 32-Bit-Wert, der üblicherweise zur besseren Lesbarkeit in Dot-Notation – also byteweise durch Punkte getrennt – angegeben wird.



Die IP-Adresse ist in die Netzwerk- und die Host-ID unterteilt. Wieviel Bits jeweils für Netzwerk- und Host-ID verwendet werden, hängt von der Klasse des IP- Netzwerks ab. Diese Netzwerkklasse kann – wie in der unteren Tabelle gezeigt – an den höchsten Bits der Adresse abgelesen werden:

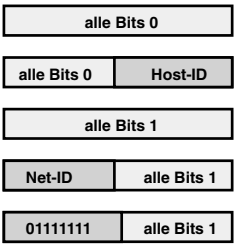


Aus der Definition der Netzwerkklassen ergeben sich damit die folgenden Adreßräume:

Class	Lowest Net-ID	Highest Net-ID
A	0.1.0.0	126.0.0.0
B	128.0.0.0	191.255.0.0
C	192.0.1.0	223.255.255.0
D	224.0.0.0	239.255.255.255
E	240.0.0.0	247.255.255.255

Üblicherweise werden jedoch nur IP-Adressen der Klassen A bis C vergeben. Mit den Klassen D und E kommt man nicht in Berührung: Die Klasse D umfaßt Netze für Multicasting und die Klasse E ist für Forschungszwecke reserviert.

Die folgenden Internetadressen haben eine besondere Bedeutung und dürfen nicht als Adresse an einen Internet-Host vergeben werden:



° meint aktuellen Host mit Netzwerk- und Host-ID
° meint Host mit dieser Host-ID im aktuellen Netz

nur für den Startup erlaubt (keine gültige Internet-Adresse)

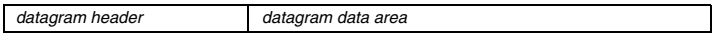
° Broadcast im lokalen Netz

° Broadcast in dem angegebenen Netz

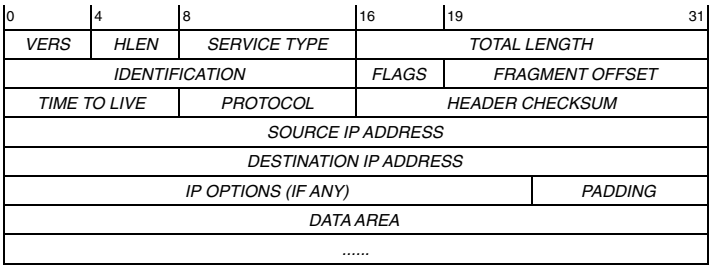
° Loopback innerhalb der TCP/IP Protokollsoftware (für Testzwecke)

3.1.3 Das Paketformat von IP

Ein Datagramm setzt sich aus einem Paketkopf (Header) und dem Datenbereich (Data Area) zusammen. Der Header enthält Informationen über das Datagramm; Hier finden sich u.a. die Adressen von Sender und Empfänger, Routing-Informationen, die Nummer des übergeordneten Protokolls zur Weiterleitung des Datagramms sowie spezielle Optionen.



Format IP-Datagram-Header:



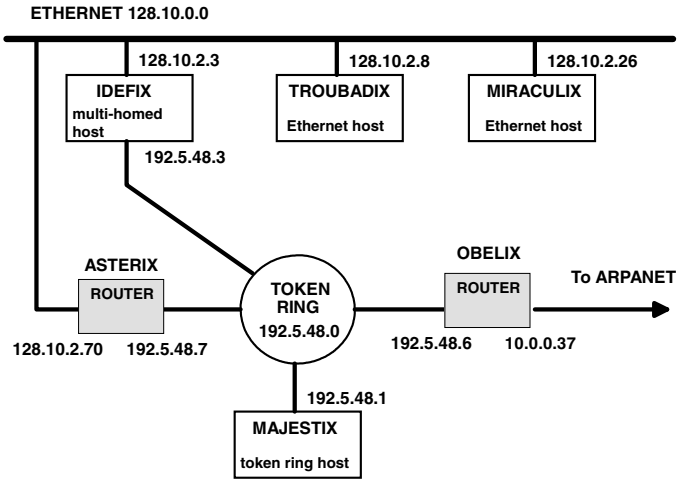
version:	binär kodierte Version des IP-Protokolls (aktuell 4.0)
hlen:	Länge des IP-Headers in DWORDs (32 Bit)
service type:	Priorität eines Pakets und Merkmale des gewünschten Übertragungswegs
total length:	Gesamtlänge des IP-Pakets aus Kopf und Nutzdaten in Bytes (8 Bit)
identification:	Vom Sender gesetzter Wert zur Identifizierung der einzelnen Fragmente
flags (3bit) :	Bit 2: Fragmentierung erlaubt 0=ja, 1=nein; Bit 3: 0=letztes Fragment, 1=weitere Fragmente folgen

time to live:	Zähler, der bei jedem Router dekrementiert wird. Wird der Wert 0 erreicht, wird das Paket verworfen.
protocol:	Nr. des übergeordneten Protokolls (z.B. TCP=6, UDP=17, ...)
header checksum:	Checksumme über den Header
source IP address:	IP-Adresse des Absenders
destination IP addr.:	IP-Adresse des Empfängers
IP options (variab.):	IP-Optionen, sofern benötigt
padding:	Füllbytes, um die Headerlänge auf ein Vielfaches von DWORDs zu bringen

3.2 Routing von IP-Paketen

Routing ist der Transport eines Datagramms vom Sender zum Empfänger. Wir unterscheiden zwischen direktem und indirektem Routing. Das direkte Routing erfolgt innerhalb eines lokalen Netzes, wobei kein Router benötigt wird. Indirektes Routing erfolgt zwischen zwei Stationen in unterschiedlichen Netzen, wobei der Sender das IP- Paket dem nächsten Router übergibt.

Ob das Paket direkt oder indirekt geroutet werden muß, ist leicht zu entscheiden: Die Software vergleicht die Net-ID des Empfängers mit der aktuellen Net-ID; sind sie nicht identisch, wird das Paket dem Router übergeben.



Die obenstehende Abbildung zeigt das Beispiel eines Netzwerks mit Hosts und Routern. Der Host IDEFIX ist ein „multi-homed Host“: Er hat Zugang zu mehreren Netzwerken (z.B. über zwei Ethernetkarten), verfügt aber über keine Routersoftware.

Die Hosts IDEFIX, TROUBADIX und MIRACULIX gehören einem Class-B- Netz (128.10.0.0) an. Das Token-Ring-Netz ist ein Class-C-Netz (192.5.48.0), welches durch den Router OBELIX mit dem Arpanet (Class-A-Netz 10.0.0.0) verbunden ist.

3.3 Subnets

Wenn ein lokales Netz nicht ausreicht oder wegen seiner Größe (z.B. bei Class-A-Netzen mit über 16 Millionen Hosts) zu unhandlich ist, wird es in weitere Teilnetzwerke – sogenannte Subnets – untergliedert. Unterschiedliche Netztechnologien in den einzelnen Abteilungen, Beschränkungen hinsichtlich der Kabellänge und der Anzahl der angeschlossenen Stationen sowie Performance-Optimierung sind weitere Gründe, Netzwerke in kleine Segmente aufzuteilen.

Da die Struktur der IP-Adresse keine Möglichkeit mehr bot, diese zusätzliche Kodierung in der Adresse selbst unterzubringen, wurde die Subnet-Mask eingeführt. Sie definiert, welche Bits der Host-ID dazu verwendet werden, die Subnet-ID zu kodieren und welche die ID des Hosts bezeichnen.

Die Subnet Mask wird vom Administrator festgelegt und wie die IP-Adresse in Dot-Notation (z.B. 255.255.255.128) dargestellt.

Werden Subnets gebildet, muß der beschriebene Routing-Algorithmus erweitert werden, da die Net-ID des Empfängers mit der des aktuellen Hosts identisch sein kann, obwohl sich beide in unterschiedlichen lokalen Netzen befinden. Deshalb müssen zusätzlich auch die Subnet-IDs von Empfänger und aktuellem Host verglichen werden. Sind diese ebenfalls identisch, kann direkt geroutet werden.

Binäroperationen mit der Subnet Mask:

Host-ID = IP-Address AND(NOT(Subnet-Mask))

Net-IDS = IP-Address AND Subnet-Mask
(Kombination aus Net- und Subnet-ID)

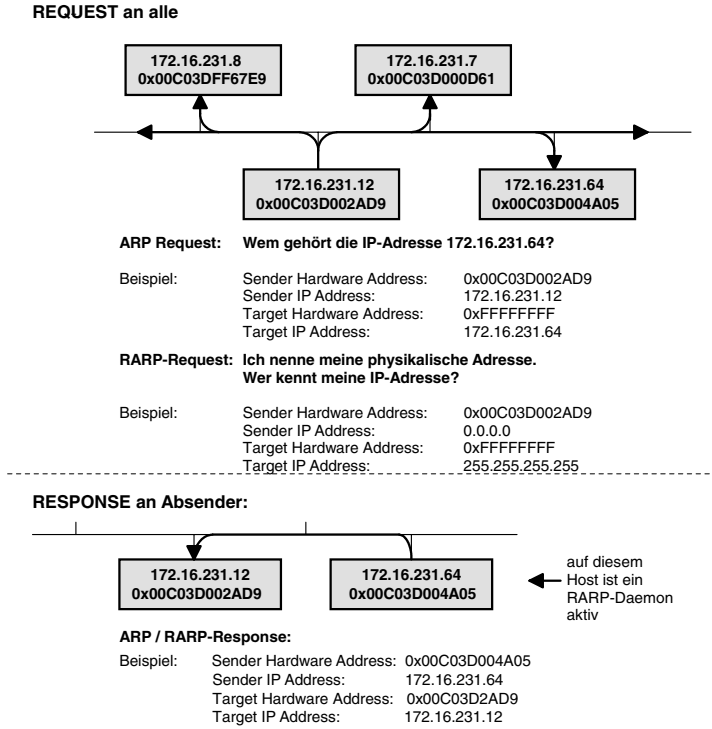
Subnet-ID: setzen Sie in der Net-IDS die Net-ID gleich 0

Beispiel für eine IP-Adresse aus einem Class B - Netz:

IP-Adresse:	172.16.233.200	10101100	00010000	11101001	11001000
Subnet-Mask:	255.255.255.128	11111111	11111111	11111111	10000000
Host-ID:	72	00000000	00000000	00000000	01001000
Net-ID:	172.16.0.0	10101100	00010000	00000000	00000000
Net-IDS:	172.16.233.128	10101100	00010000	11101001	10000000
Subnet-ID:	0.0.233.128	00000000	00000000	11111111	10000000

3.4 ARP und RARP

ARP und RARP (letzteres findet sich nur unter UNIX) liefern Mechanismen, mit denen sich IP-Adressen auf die physikalischen Netzadressen abbilden lassen, die man beim direkten Routing benötigt. Jedes hardwarenahe Protokoll (Ethernet, X.25, ISDN ...) hat sein eigenes Adreßformat und versteht keine IP-Adressen. Befindet sich der Empfänger nicht im lokalen Netz, benötigt man die physikalische Adresse des Routers, der das Paket in ein anderes Netz weiterreicht.



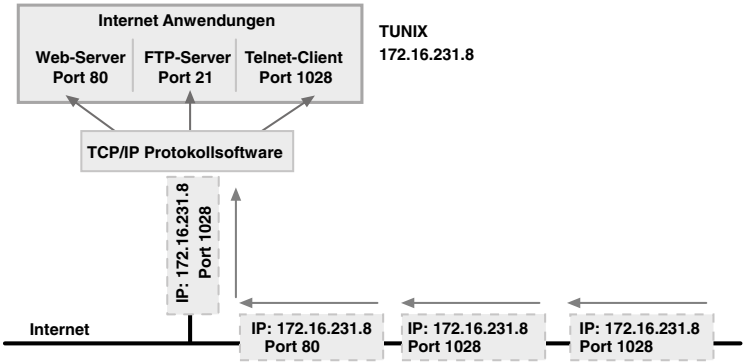
Die Zuordnung von Ethernet-Adresse und IP-Adresse wird in einer Tabelle gespeichert und erst nach einem Timeout gelöscht.

Achtung: *Bei Änderung dieser Zuordnung (z.B. Zuweisen der gleichen IP-Adresse an ein Austauschgerät) werden Sie u.U. keine Verbindung zum Empfänger erhalten. Steht der Befehl „arp" nicht zur Verfügung, schafft hier nur Warten, ein Neustart des Rechners oder die Vergabe einer neuen IP-Adresse Abhilfe.*

3.5 Transport Layer

3.5.1 Adressierung der Applikationen mit Portnummern

Die IP-Adresse adressiert einzig und allein den Host. Auf jedem Host können jedoch gleichzeitig mehrere Applikationen aktiv sein, z.B. ein Web-Browser, ein Telnet Client und andere. Die notwendigen Mechanismen zur Adressierung der Applikationen bieten die Protokolle TCP und UDP.



Für bekannte Anwendungen werden feste Ports vergeben, auf die sich jede Anwendung beim Verbindungsaufbau beziehen kann. Der Bereich von 0 bis 1023 enthält deshalb reservierte Portnummern. Diese dürfen unter keinen Umständen für eigene Anwendungen verwendet werden. Die komplette Liste der „normierten Dienste“ finden Sie im RFC 1700 (1994).

ftp://ftp.isi.edu/in-notes/iana/assignments/port-numbers // Ports
http://sunsite.auc.dk/RFC // RFCs

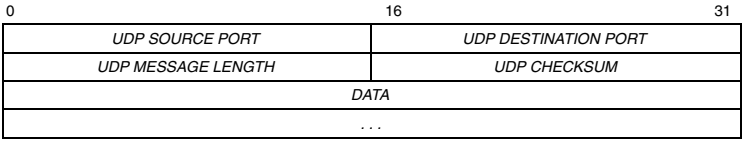
Hier eine kurze Aufstellung von Applikationen und deren Port-Nummern:

Anwendung	Port	Protokoll	Beschreibung
ftp	21	udp / tcp	File-Transfer
telnet	23	udp / tcp	Telnet
smtp	25	udp / tcp	Simple Mail Transfer
domain	53	udp / tcp	Domain Name Server
tftp	69	udp / tcp	Trivial File-Transfer
www-http	80	udp / tcp	World Wide Web HTTP
sftp	115	udp / tcp	Simple File Transfer Protocol
snmp	161	udp / tcp	SNMP
...			

3.5.2 Das Format von UDP

Die Leistungsmerkmale von UDP beschränken sich auf die Trennung der Kommunikationskanäle der Applikationen. Die Zustellung der Datagramme im Netz ist nicht abgesichert. Das Protokoll bietet keine Gewähr für die Einhaltung der Reihenfolge.

Eine Internetanwendung, die auf UDP aufsetzt, muß selbst für die Absicherung der Datenübertragung sorgen. UDP ist deshalb für Applikationen mit eigenen Sicherungsmechanismen geeignet. Es spart „Protokoll-Overhead“ und bietet dadurch höhere Übertragungsraten als TCP. Zusätzlich entfallen alle Mechanismen für Verbindungsaufbau und -abbau.



Format eines UDP-Datagram-Headers

Source Port: Port des Absenders; wird für das Rücksenden von Daten durch den Empfänger benötigt.

Destination Port: Ziel-Port, an den das Datenpaket beim Empfänger übertragen werden soll.

Länge: Größe des UDP-Datagramms in Byte (Header und Daten).

Checksumme: Checksumme über das UDP-Datagramm, jedoch optional (wird sie nicht genutzt, erscheint in diesem Feld eine 0).

3.5.3 TCP – Transport Control Protocol

TCP befreit die Internetanwendung von Sicherungsmechanismen und realisiert im Gegensatz zu UDP einen sicheren Kommunikationskanal. Deshalb basieren nahezu alle wichtigen Internetanwendungen (HTTP, E-Mail usw.) auf TCP.

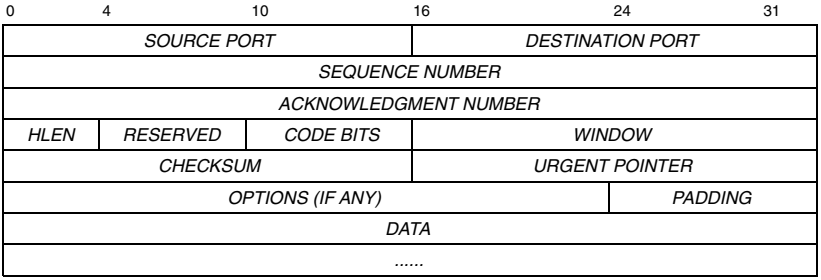
Die Endpunkte einer TCP-Verbindung bilden zwei Tupel aus IP-Adresse und Portnummer. Es wird eine virtuelle Verbindung zwischen den beiden Endpunkten aufgebaut.

Die Kommunikation ist *Vollduplex*, d.h. beide Kommunikationspartner können gleichzeitig Daten senden und empfangen.

Das Protokoll ist für die Applikation transparent – Daten, die an die TCP- Schnittstelle übergeben werden, kommen beim Empfänger auch so an.

Die Paketgrößen sind frei wählbar. Sofern es keine Hardwareeinschränkungen gibt, ist hier von einem Byte bis hin zu mehreren Megabyte alles erlaubt.

Das Format eines TCP-Pakets:



Source Port: Portnummer der Applikation des Absenders

Destination Port: Portnummer der Applikation des Empfängers

Sequence No: Offset des ersten Datenbytes relativ zum Anfang des TCP-Stroms (garantiert die Einhaltung der Reihenfolge)

Acknowl. No: im nächsten TCP-Paket erwartete Sequence-No. (ACK für die erhaltenen Bytes)

HLEN: Größe des TCP-Headers in DWORDs (32 Bit), Beginn des Datenbereichs

Code Bits: bezeichnen Aufgabe und Inhalt des Pakets:

Bit 1:	URG	das Segment enthält dringende Daten, Feld Urgent Pointer beachten SOCKET-INTERFACE: Out Of Band Data
2:	ACK	das Segment enthält ein Acknowledge
3:	PSH	empfangene Daten sofort weitergeben (push)
4:	RST	Verbindung zurücksetzen (reset)
5:	SYN	Verbindungsaufbau und Synchronisation der Sequence Numbers
6:	FIN	keine weiteren Daten vom Sender, Verbindung schließen

Window: Anzahl der Bytes (ausgehend von dem im Feld Acknowledgement-No. angezeigten Byte), für deren Entgegennahme der Empfänger bereit ist

Checksum: Checksumme über das TCP-Datagramm und einen Pseudoheader

Urgent Pointer: wenn das URG-Flag gesetzt ist, gibt dieser Wert den Offset des ersten Bytes hinter den dringenden Daten am Anfang der Segmentdaten an

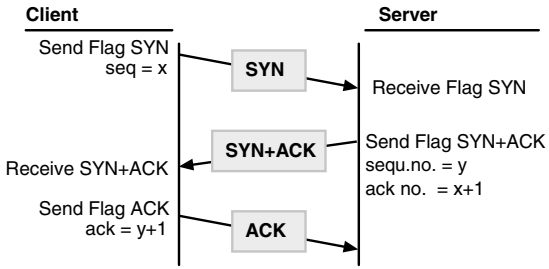
Options: Optionen (die wichtigste Option: maximale Segmentgröße)

Auf- und Abbau einer TCP-Verbindung

TCP verfügt über feste Mechanismen zur Einrichtung einer Verbindung zwischen Client und Server. Der Aufbau einer Verbindung dient unter anderem dazu, beide Teilnehmer auf den zu übertragenden Datenstrom zu synchronisieren und Übertragungsparameter wie Paketlänge und Empfangsspeichergröße auszutauschen.

- **Aufbau:**

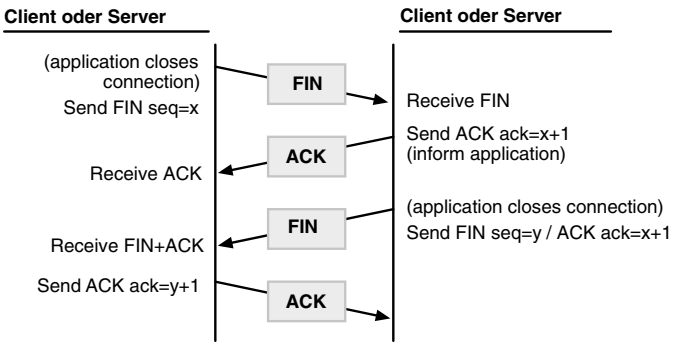
  Der Client sendet ein Paket, in dem in den *Code-Bits* das Flag SYN gesetzt ist. Der Server bestätigt den Empfang mit dem Flag ACK und setzt seinerseits das Flag SYN. Beide synchronisieren sich auf die *Sequence-No.* der Gegenstation.



  In der Option *Maximum Segment Size* kann jede Seite festlegen, wieviel Byte sie maximal in einem Segment nach dem TCP-Header empfangen kann.

- **Abbau:**

  Das Schließen einer Verbindung kann vom Client oder vom Server veranlaßt werden. Hierzu wird in den *Code-Bits* das Flag FIN gesetzt. Erst wenn beide Seiten dieses Flag gesetzt haben, gilt die Verbindung als geschlossen.



Flußsteuerung

TCP verfügt über mehrere Mechanismen, zur sicheren und effizienten Übertragung von Daten. Hier ein paar der wichtigsten Regeln:

- Der Sender muß alle Daten solange bereithalten, bis sie vom Empfänger bestätigt wurden.
- Bei fehlerhaften Paketen (z.B. Paketen mit fehlerhafter Checksumme) gibt der Empfänger die letzte Acknowledgment-Number zurück, woraufhin der Sender das Paket wiederholt.
- Gehen Pakete verloren, wiederholt der Sender nach Ablaufen eines Timeouts alle Pakete nach dem zuletzt empfangenen Acknowledgment.
- Mit dem Feld *Windows* teilt der Empfänger in jedem Paket mit, wieviel Empfangsspeicher er noch hat. Enthält das Feld den Wert 0, stellt der Sender die Übermittlung ein, bis er vom Empfänger ein Paket mit dem Wert *Windows* ungleich Null erhält.
- Da der Sender ständig über die aktuelle Buffergröße des Empfängers informiert ist, muß er nicht auf die Bestätigung jedes einzelnen Paketes warten, sondern kann so viele Daten senden, bis der Buffer des Empfängers gefüllt ist. Der Empfänger bestätigt dabei immer einen Teil des Empfangs-Bytestroms und nicht die einzelnen Pakete. Dadurch können Teile von Paketen oder auch mehrere Pakete auf einmal bestätigt werden. Diese Methode nennt man *Windowing*.

Literaturverzeichnis

- **INERNET intern**
  Tischer und Jennrich
  Verlag: DATA Becker
  ISBN 3-8158-1160-0
- **Inside Visual C++**
  David J. Kruglinski
  Verlag: Microsoft Press
  ISBN 3-86063-394-5
- **Internetworking with TCP/IP**
  Verlag: PRENTICE HALL
  - **Volume I: Principles, Protocols and Architecture**
    Douglas E. Comer
    ISBN: 0-13-216987-8
  - **Volume II: Design, Implementation and Internals**
    Douglas E. Comer, David L. Stevens
    ISBN: 0-13-125527-4
  - **Volume III: Client-Server Programming and Applications**
    Douglas E. Comer, David L. Stevens
    ISBN: 0-13-260969-X