W&T
www.WuT.de

Ready in 1 day forTCP/IP-Sockets

The WinSock Socket Interface

Sample code for linking Com-Servers

TCP/IP Basics

Introduction

In all likelihood you are a programmer. and you probably have experience already in linking peripherals – for example through the COM port – into your applications. And now you‘re interested in linking these devices directly over the network.

Once you have gotten over the initial unfamiliarity with this new material, you‘ll see that for a programmer the network approach is no more complicated than using the good old COM port.

You will encounter a few new terms, but on the other hand many of the peculiarities of the COM link are absent here. So let‘s just get started!

Ready for TCP/IP sockets in 1 day - here‘s how to proceed:

1. First read *Part1 – The Socket Interface*. This will not only give you a rapid overview, but also provide you with a lot of important detail information which will make things easier in the later steps.
2. Now you‘ll already need a „playground“ where you can test out your newly acquired knowledge. To avoid having to start out with two applications, we recommend you order a sample Com-Server from us as a test unit, which will then take over the tasks of the network communications partner (Socket client or server).

   (If you are really pressed for time, you can instead try to establish communication between two computers. This method is more challenging, since you have to get two programs running at the same time. Unless you have previous experience with such things, you should probably stay away from this approach.)
3. Now go to *Part 2 – Socket-API Programming Examples* and select the examples for your respective programming environment. You can also download the sources including all the necessary Includes and resource data as well as additional examples for Windows 9x/NT from our Web site at *http://www.WuT.de*.
4. By the time you have worked through the programming examples you will have come across new terms from the network world. But everything you actually need to know about this terminology is summarized in compact form in *Part 3 – TCP/IP-Basics*.

Between working with the programming examples and your own experimenting, you will quickly find that you have acquired all the knowledge you require.

If you still feel the need for a more extensive introduction, we have included a bibliography at the end of this document.

Contents

Part 1
The Socket Interface

1. The Socket Interface

The socket interface, a recent addition to the PC world, was actually developed more than 15 years ago as the „Berkley Socket Interface“ under BSD-UNIX 4.3. This interface provides relatively easy to use commands to access the functionality of TCP/IP; over the years it has been incorporated into many other UNIX systems.

In the meantime, *WinSock.DLL* and with it the functionality of the socket interface has become a standard component of Windows 9x. It‘s easy to understand the reason for this interface. Not only does it permit the new development of Internet applications, but also the moving of applications from UNIX to the PC, since it is for the most part compatible with Berkley Socket.

1.1 Client-Server principle

Internet applications are created according to the client-server principle. In most cases the client is here the user interface and makes use of services determined by the server. Based on previously defined events (e.g. starting of an Internet application by a user), it establishes the connection to the server and is in that sense the active partner.

The server then makes the desired service available. It must always be in a state in which it can accept connection requests from clients – which makes it the passive partner. A server can never request a service from the client.

Client and server have to speak the same language: in other words, they must adhere to a common protocol.

The different role of client and server does however result in a certain asymmetry which in turn means that different interface commands need to be used when implementing a client- or server application.

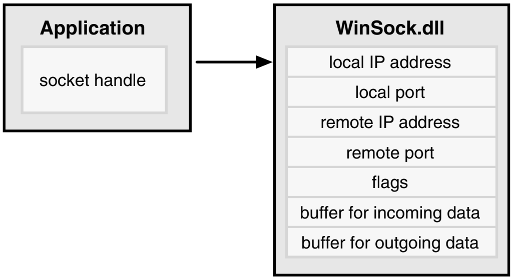
1.2 Linking socket functions in C

To integrate the functionality of the WinSock interface into your own application, include the file *winsock.h* into the C source code using *#include <winsock.h>* into the C source code.

The compiler and linker need the LIB file *winsock.lib* for 16-bit applications or *Wsock32.lib* for 32-bit applications in order to generate the program code. Therefore include one of the two files into your project.

1.3 The socket variable

The socket variable is of the integer type. It has nothing to do with the port number of the application, as is often assumed, but is simply a handle for a connection. Under this number various commands are used so that the driver provides all the connection information which belongs to this handle.



1.4 The main socket functions in C

The extent of WinSock functionalities can vary significantly depending on the reference source of the development tool. One often comes across a variety of undocumented functions that no one knows what to do with.

The essential functions of the WinSock interface can be found in the table below. This listing includes the basic functions which are represented in all versions.

basic functions	description
accept	permits server program to accept a new connection and returns a new socket for this connection
bind	associates a local Internet address and a port with a socket
closesocket	closes an existing socket
connect	establishes a connection to a specified socket
getpeername	retrieves the IP address of the peer to which the socket is connected
getsockname	retrieves the local name for a socket
getsockopt	retrieves a specified socket option
ioctlsocket	sets socket flags
listen	generates queue for incoming connections
recv	receives data from connected socket
recfrom	receives data and stores the source address
send	sends data on a connected socket
sendto	sends data to specified address
select	determines the status of one or more sockets
shutdown	disables sending or receiving of data on a socket
socket	creates a new socket

1.5 Network order or host order?

Anyone who has network application programming experience is familiar with that annoying problem of byte order. The cause of this lies in the system- or architecture-dependent interpretation of the memory if the latter is not processed byte-for-byte but rather as WORD or LONG. To make sure an application on an Intel PC can also make use of the services of an application on a Macintosh for example, a standard had to be found for sending WORDs and LONGs over the network.

Network order (also referred to as big endian) specifies that the highest byte is sent first and the lowest byte last.

storage address	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	

Since the memory on all Intel PCs is processed using host order (the low-value byte is ahead of the high-value byte in the memory), it is necessary to convert all LONG or WORD type values to network order before handing over to the driver. Likewise all received values must be converted to host order before they can be used. Caution should be used when making comparisons: You will not get the same result in both network and host order!

Here the socket interface provides a variety of conversoin functions. If you take a closer look at the table you will notice that there is always one function for each direction (network -> host order and host -> network order). This is actually a duplication, since no matter which variable you take, it will and must always give you the same result. The only advantage is in better readability of the program. Based on the command used you can tell whether the respective value is in host or network order.

converting functions	description
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	converts IP address in string format to numerical address *(long)*
inet_ntoa	converts numerical IP address *(long)* to string format

The functions *inet_addr()* and *inet_ntoa()* differ slightly from each other. They convert an Internet address which exists as a string in *Internet Standard Dotted Format* into a 32-bit value (*inet_addr()*) or vice-versa (*inet_ntoa()*).

1.6 Database functions

The purpose of these functions is to retrieve information about names, IP addresses, network addresses and other data from the driver. A user for example enters the target hose in the dialog box of an application not as an IP address, but rather enters the name of the station. It‘s impossible to remember all the IP addresses in the Internet. Since names are easier to recall than numbers, it is customary to use names here.

Database functions take care of things such as converting names into IP addresses and vice-versa. For this they make use of a domain name server or process local files.

data base functions	description
gethostbyaddr	retrieves host name for specified IP address
gethostbyname	retrieves IP address for specified host name
gethostname	retrieves name of local host

1.7 Blocking functions

All the functions in the socket interface are blocking functions. The blocking effect won‘t be noticed with database or conversion functions, since these functions always provide an immediate result. But if you invoke for example the *recv()* function to receive data from the socket, it will first return the check if there are actually data to receive.

To get around such blocking effects it is mandatory to invoke the *select()* function prior to each read or write action. This tells for a given number of sockets whether data can be sent or received.

Note also with regard to this problem the *WSAAsynSelect()* function, a Windows-specific versoin of *select()*.

1.8 Specific functions of the WinSock interface

The WinSock interface has several functions which are especially adapted to the Windows environment.

Before a socket function is first invoked, use of the *WinSock.DLL* must be initialized by the started process:

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

Only after initializing with *WSAStartup()* can other functions be successfully invoked. *WSAStartup()* allows the required version of the DLL to be specified and details about the implemented DLL stored in the *WSDATA* structure. Likewise, before ending the process you must terminate the work with *Winsock.DLL*. The last function invoked is threfore always *WSACleanup()*.

Windows-specific functions	description
WSAStartup	initialises Windows sockets
WSACleanup	frees socket bindings before application terminates
WSAAsyncSelect	variant of *select()* for sockets in asynchronous mode
WSAGetLastError	returns error code of last socket call that failed

The use of *WSAAsyncSelect()* – the asynchronous version of *select()* – offers many advantages. This function allows non-blocking work with sockets: it initializes application messaging using Windows Messages as soon as network events occur for this socket.

The function *WSAGetLastError()* provides the last error code when a socket is invoked. If a socket invoke returns the value *SOCKET_ERROR*, this function must be immediately invoked.

1.9 The main structures

The socket interface structures appear to be confusing at first glance. But upon closer inspection it becomes clear that all these structures refer to the same thing – just in a different form.

```
/*  WINSOCK.H */
struct sockaddr
{ u_short sa_family;          // Address-Family (always AF_INET)
  char sa_dat[14];            // Address
}
struct sockaddr_in
{ u_short sin_family;         // Address-Family (AF_INET)
  u_short sin_port;           // desired port
  struct in_addr sin_addr;    // the IP address
  char sin_zero[ 8 ];         // fills the structure
}
```

The structures *sockaddr* and *sockaddr_in* have the same contents: the address family and the address itself. Addresses from all the families can be entered in the array *sa_dat[14]* of the structure *sockaddr*. In the Internet the address family is always *AF_INET*. The structure *sockaddr_in* configures this array for Internet addressing format: for the port number and IP address. You only need 6 bytes for this – which is why you see the 8 unused bytes at *sin_zero[8]* at the end of the structure.

```
/*   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;
}
```

The structure *in_addr* contains nothing more than the IP address itself. It simply allows access to individual components of the IP address without having to form complicated casts. Depending on the IP network class, you can use this structure to access network and host ID separately.

```
/*  WINSOCK.H */
struct hostent
{ char FAR *h_name;              // String with official host name
  char FAR* FAR* h_aliases;      // pointer to alternate names
  short h_addrtype;              // always PF_INET
  short h_length;                // always 4 (IP address)
  char FAR* FAR* h_addr_list;    // pointer to array with IP addresses

  #define h_addr h_addr_list[0]  // for access to the first IP address
};
```

The structure *hostent* is important for database functions. Here you find all the supplementary information for a main information, such as name and additional alternate names for a given IP address, or all IP addresses associated with a name. Normally there is only one address for a name; only in the case of multi-homed hosts might you get back multiple IP addresses.

1.10 Streams and datagrams

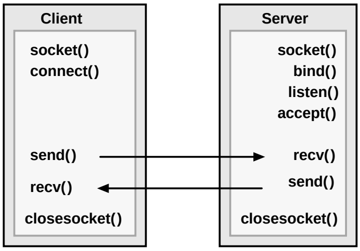
When creating a new socket you must decide whether a STREAM socket or a DATAGRAM socket will be initialized. Lurking behind this is the not insignificant decision between TCP and UDP. Both protocols have their pros and cons depending on the application.

The usual way is to initialize a STREAM socket, i.e. to use TCP. This relieves you of all the worries involved with securing and checking the data flow. Note however that if you have rapidly changing senders and receivers in this case, connections also have to be constantly made and terminated or many sockets have to be initialized – which costs time and administration overhead.

UDP is faster, but does not provide any security mechanisms. You will have to use other means of checking for data integrity.

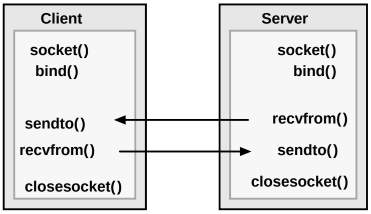
The two overviews on the following page each shows the initialization of the corresponding protocol using the *socket()* command along with the command sequence which has to be used when implementing client and server applications.

Stream clients and servers (TCP)



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

Datagram clients and servers (UDP protocol)



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

W&T
www.WuT.de

Part 2
Program Examples
Socket-API

DOS Applications in C

Windows 9x Applications in C

Visual Basic

Java

This brief programming reference is devoted specifically to TCP/IP protocol. Here you will find a short selection of sample programs for various environments (Windows95/C, DOS/C, JAVA and Visual Basic), which should get you started quicker on your own application. Have no fear: the socket interface is really quite easy to use. To experience your first success – in other words: to send or receive data to or from the Com-Server – all it takes is invoking of a few functions.

All these examples can also be found at our Web site *http://www.WuT.de* for downloading.

2.1 C: DOS environment

Description of the programming environment

The examples in this section are intended to show in brief form how to create applications for the Com-Server using the socket interface for the DOS operating system. The programs can be run under DOS or in the DOS box of Windows and were created for the Novell TCP/IP stack (LAN Work Place V4.1).

Programming environment of the example:

Programming language:	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 Programming Interface (API)

In these examples the module *LLIBSOCK.LIB* for Large-Model-DOS-Library-Functions was linked.

2.1.1 Program example: Socket Client

The program *tcpclnt.c* implements a TCP client. When invoking the program the IP address of the Com-Server is given in dot notation (e.g. 190.107.231.1) or the name of the Com-Server is given as an argument.

The program establishes a connection to port A of the desired Com-Server, displays all the received data on the monitor and sends all keyboard entries to Com-Server port A after the Enter key is pressed.

The *terminal()* function implements the functionality of a terminal (data in- and output).

```
/*************************************************
***   tcpclnt.c                                ***
***   TCP Client-Program: Terminal-Function    ***
***   Quit program with 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 descriptor
    struct sockaddr_in box;           //Com-Server address
    int portno = 8000;                //predefine Com-Server port A
    char *hostname;
    u_long remote_ip;
    if(!loaded())                    //TCP/IP-Stack installed?
      { printf("Kein TCP/IP protocol stack active\n");
        exit (1);
      }
    if(argc < 2)                     //get host name from argument
      { printf("No host name given\n");
        exit (1);
      }
    if(argc > 2)                     //port number may be 2nd parameter
      portno = atoi(argv[2]);

    bzero((char *)&box,sizeof(box));//delete address structure
    hostname = argv[1];              //rhost() expects (char**)-Argument!
    if((remote_ip = rhost(&hostname)) == -1)
      { printf("Unknown host name\" %s\"\n", argv[1]);
        exit(1);
      }

    //open handle for TCP-Transport
    if((sd = socket(PF_INET,SOCK_STREAM, 0)) < 0)
      { soperror("socket");
        exit(1);
      }

    box.sin_family = AF_INET;
    box.sin_port = htons(portno);             //destinnation port number
    box.sin_addr.s_addr = rhost(&hostname);   //destination IP address

    //open connection to COM-Server Port A
    if(connect (sd, (struct sockaddr*)&box,
      sizeof(box)) < 0)
      { soclose(sd);                 //close handle again
        soperror("connect");
        exit(1);
      }

    //receive and send data until ALT Q pressed
    printf("Linked to COM-Server %s:%d\n", inet_ntoa(box.sin_addr),
              htons(box.sin_port));

    terminal();
    soclose(sd);                     //close handle again
  }

void terminal(void)
  {
    fd_set rd_ready,wr_ready;        //bit fields per socket descriptor
    struct timeval maxwait;          //Max. wait time for select()
    int s_len = 0;
    int r_len = 0;
    int z_count = 0;
    char key;
    for(;;)
      {
        FD_ZERO(&rd_ready);          //The API should let us wait for max. 10µs.
        FD_SET (sd,&rd_ready);       //select() indicates the number of active
        FD_ZERO(&wr_ready);          //connections and sets for each active
        FD_SET (sd,&wr_ready);       //connection one bit in the transfered
        maxwait.tv_sec = 0;          //bit fields. Return 0: no data
        maxwait.tv_usec = 10;        //received and no send possible

        if(select(sd+1,&rd_ready,&wr_ready,
            (fd_set*)0,&maxwait) == 0) continue;
        if(FD_ISSET(sd, &rd_ready)) //data ready?
          { if((r_len=soread(sd, RecBuf, BUF_SIZE))>0)
            { *(RecBuf[r_len]) = 0   //mark string end
              printf("%s",RecBuf);   //output data to monitor
            }
            else if(r_len == 0)      //regular connection break
              { printf("\nCOM-Server has ended the connection\n");
                return;
              }
            else if(r_len <= 0)      //connection error
            { soperror("soread");
              return;
            }
          }

        if(kbhit())                  //read keyboard inputs
          { key = getch();
            if(!key)                 // special character
              { if(getch()==16)      // ALT Q -> quit terminal
                { printf("\n");
                  return;
                }
              }
            else
              { SendBuf[z_count++] = key;
                if(key==0x0D)        //ENTER->send line to Com-Server
                { SendBuf[z_count++] = 0x0A;
                  s_len = z_count;
                  z_count = 0;
                }
              }
          }
        //as soon as an input line is complete (and the API is ready),
        //send line:
        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 Program example: Socket Server

The program *tcpserv.c* implements a TCP server on Socket 2000. The Com-Server is also run here in „Socket Client Mode“: If there are data present on the serial interface, the Com-Server establishes a connection to a server program and gives it the data for processing.

This program outputs all received data on the monitor and sends all keyboard inputs to the corresponding port of the Com-Server after pressing the ENTER key.

The *terminal()* function for implementing the functionality of a terminal was already discussed in the preceding section.

```
/**************************************************
***   tcpserv.c                                ***
***   TCP Server-Program: Terminal Function    ***
***   To close a connection press ALT Q        ***
***   Quit Server Mode with 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 descriptors
    struct sockaddr_in box,loc;         //addresses of Com-Server and PC
    int box_size = sizeof(box);
    fd_set rd_ready;                    //flags
    struct timeval maxwait;             //max. wait time for select()
    bzero((char *)&loc, sizeof(loc));   //delete address structures
    bzero((char *)&box, sizeof(box));
    if(!loaded())                       //TCP/IP stack installed?
      { printf("No TCP/IP stack active\n");
        exit (1);
      }

    // open handle for 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" would also be permitted
  // link socket ss to a "Name" (IP address and port)
  if(bind (ss, (struct sockaddr*)&loc, sizeof(loc)) < 0)
    { soclose(ss);                  //close handle again
      soperror("bind()");
      exit(1);
    }

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

  for(;;)                           //wait for connections
    {
      if(kbhit() && getch()==27)
        break;                      //ESC quits server mode

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

      //ask API if data were received
      if(select(ss+1, &rd_ready, NULL, NULL, &maxwait) == 0)
        continue;
      //accept connection and store client address in the structure "box"
      if((sd = accept(ss, (struct sockaddr*)&box, &box_size)) < 0)
        { soclose(ss);
          soperror("accept()");
          exit(1);
        }

      printf("Connection from %s:%d accepted.\n",
              inet_ntoa(box.sin_addr),htons(box.sin_port));
      //invoke terminal function for data exchange until the client ends
      //the connection or ALT Q is pressed
      //(see "Terminal" function in example 2.2.2)
      terminal(sd);
      soclose(sd);                  //close connection to client
      printf("Socket closed\n");
    }                               //wait for the next server invoke
  soclose(ss);                      //disable server socket
}
```

2.1.3 Program example: UDP Server

The program *udpserv.c* implements a UDP server on Socket 2000. The Com-Server is configured in „UDP Mode“ and sends all serial data to this UDP server.

UDP provides no connection control. You should only work with UDP if the data between the serial terminal device and your final application have already been sent using a protocol which itself provides for error-free data transfer.

The program *udpserv.c* outputs all received data on the monitor, and after ENTER is pressed sends all keyboard inputs to the Com-Server port from which data were last received.

```
/*************************************************
***   udpserv.c                               ***
***   UDP Server-Programm: Terminal Function  ***
***   Quit program with 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 descriptor
    struct sockaddr_in box;               //structure Com-Server Port
    struct sockaddr_in loc;               //structur PC
    struct timeval maxwait;               //time to wait for API
    fd_set rd_ready,wr_ready;             //flags for select()
    int s_len = 0;
    int r_len = 0;
    int z_count = 0;
    int boxlen = sizeof(box);
    char key;

    bzero((char *)&loc, sizeof(loc));     //delete address structures
    bzero((char *)&box, sizeof(box));
    if(!loaded())                         //TCP/IP stack installed?
      { printf("No TCP/IP protocol stack active\n");
        exit (1);
      }
    //open handle for 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" would also be permitted
    box.sin_family = AF_INET;
```

```
//attach socket sd to a "Name"(IP address and port):
if(bind (sd, (struct sockaddr*)&loc, sizeof(loc)) < 0)
  { soclose(sd);                      //close handle again
    soperror("bind()");
    exit(1);
  }
printf("UDP-Server ready: %s:%d\n", inet_ntoa(loc.sin_addr),
         htons(loc.sin_port));
for(;;)                               //wait for data
  { 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;       //the API should block for a maximum of 10µs

    //select() indicates the number of active connections and sets for each
    //active connection a bit in the transfered bit fields. return value 0:
    //no data received and at the moment no data are being sent.

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

    //receive data and store sender in structure "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;        //mark end of string
            printf("Data from %s:%d: %s\n", inet_ntoa(box.sin_addr),
                     htons(box.sin_port), RecBuf);
          }
        else                          //connection error
          { soperror("recvfrom()");
            goto quit;
          }
      }

    if(kbhit())                       //read keyboard inputs
      { key = getch();
        if(!key)                      //special character
          { if((char)getch()==16)     //ALT Q -> quit Terminal
              goto quit;
          }
        else if(box.sin_port)         //Com-Server port known?
          { SendBuf[z_count++] = key;
              if(key==0x0D)           //ENTER -> send line
                { SendBuf[z_count++] = 0x0A;
                  s_len = z_count;
                  z_count = 0;
                }
          }
        else
          printf("Send to whom?\n");
      }
```

```
        //Were characters read from the keyboard and ist the API ready to send?

        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;
                                              //set character number to zero
          }
      }                                       //End for(;;)
    quit:
    soclose(sd);
    exit(1);
  }
```

2.2 C application environment: Windows 9x/NT

Description of programming environment

This application for Windows 9x or Windows NT implements a TCP client and enables data exchange with the Com-Server. This example shows handling of the socket interface using Windows Messages.

System requirements:

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

Programming environment for the example:

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

```
/*****************************************************************
*                                                                *
*       clnt_tcp.c    (Win32 Application)                        *
*                      Microsoft Visual C++ 5.0                  *
*                                                                *
******************************************************************
* TCP Client: The client opens the connection to the TCP         *
* server whose address or name is entered in the dialog box.     *
* All the data entered in the "Send" field are sent to the       *
* server, all received data aare output in the "Receive" window. *
* The status window displays those WinSock functions which       *
* were just carried out.                                         *
*****************************************************************/

#include <winsock.h>                //include also windows.h!
#include <stdio.h>

#include "resource.h"               //dialog box constants
#define WM_SOCKET (WM_USER + 1)     //private windows messages
#define SERVER_PORT 8000            //Com-Server port A

SOCKET iClient = INVALID_SOCKET;

/* log status messages*/
void ShowStatus(HWND hWnd, LPSTR lpMsg)
{
  int iEntries;
  //add new entry
  SendMessage(GetDlgItem(hWnd, IDC_STATUS),
              LB_ADDSTRING, (WPARAM)-1, (LPARAM)lpMsg);
  //show last entry
  iEntries = SendMessage(GetDlgItem(hWnd,IDC_STATUS), LB_GETCOUNT, 0, 0);
  SendMessage(GetDlgItem(hWnd, IDC_STATUS),  LB_SETTOPINDEX, iEntries-1, 0);
}
```

```
/* dialog procedure of main dialog*/

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

    case WM_SOCKET:                   //WINSOCK-Messages
    {
      switch(WSAGETSELECTEVENT(lP))
      {
        case FD_CONNECT:            //message from 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:               //receive data
        { char rd_data[255];
          int  iReadLen;            //read data
          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:              //connection break
          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:                      //button messages
  switch(LOWORD(wP))
  { case IDCANCEL:                    //close window box
      EndDialog(hWnd, 0);
    break;

    case IDC_SEND:                    //send data
      if(iClient != INVALID_SOCKET)
      {
        char Buffer[255];
        int iSendLen;
        ShowStatus(hWnd, "FD_WRITE"); //read server address
        iSendLen = GetWindowText(GetDlgItem (hWnd,IDC_SENDDATA),
                                 Buffer, sizeof(Buffer));
        ShowStatus(hWnd, "send()...");

        if(send(iClient, Buffer, iSendLen, 0) != SOCKET_ERROR)
        { ShowStatus(hWnd, "... ready");
          EnableWindow(GetDlgItem(hWnd, IDC_SEND), TRUE);
        }
        else
        { if(WSAGetLastError() == WSAEWOULDBLOCK)
            ShowStatus(hWnd, "... blocked");
          else
            ShowStatus(hWnd, "send error()");

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

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

      //read out destination address
      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 address-> dot-Notation
      ShowStatus(hWnd, "inet_addr()");
      sin.sin_addr.s_addr = inet_addr(remoteIP);
      //resolve address through DNS
      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, "Invalid Internet address");
          break;
        }
      }

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

      //create socket
      ShowStatus(hWnd, "socket()");
      iClient = socket(AF_INET, SOCK_STREAM, 0);
      if(iClient == INVALID_SOCKET)
      { ShowStatus(hWnd, "Error when allocating connect socket");
        ShowStatus(hWnd, "No connection can be established");
        break;
      }

      //activate asynchronous mode
      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, "Wait...");
            //deactivate "Send" button
            EnableWindow(GetDlgItem(hWnd, IDC_SEND),FALSE);
            break;
          }
        }
      }
      else
        ShowStatus(hWnd, "Error with 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:                  //close window
    if(iClient != INVALID_SOCKET)
      closesocket(iClient);         //close socket
  break;
  }
return FALSE;
}

/***************************************************
* WinMain: Main entry point                       *
****************************************************
* Parameters:Standard dialog parameters           *
* Return value:  0                                *
***************************************************/

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

if(WSAStartup(MAKEWORD(1,1), &wsadata) == 0)
{
  DialogBox(hInstance,              //show dialog
            MAKEINTRESOURCE(IDD_WSCLIENT),
            NULL,
            WSClientProc);
  WSACleanup();                     //clean up Winsock-DLL
}
else MessageBox(0,
        "Error when initializing WINSOCK.DLL",
        "WSClient.EXE", 0);
return 0;
}
```

2.3 The Visual Basic environment

Description of the programming environment

This example shows how you can create a TCP-Client application for the Com Server using a Socket interface in Visual Basic. The MS Winsock control element is used, which is included standard with Visual Basic Version 5.

System requirements:

- Microsoft Windows 95, 98 or NT 4.0
- Visual Basic 5.0 or higher
- Microsoft Windows TCP/IP-Stack

Programming environment for the example:

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

2.3.1 Integrating Winsock Control with Visual Basic project

The MS Winsock Control enables communication through TCP/IP or UDP/IP, whereby TCP allows both client as well as server applications.

The Winsock Control must first be added to the VB project you wish to create as a new component.. Go to *Projects->Components* ... to find the selection of all optional control elements.. Select the item „Microsoft Winsock Control 5.0“ and confirm your selection with *OK*. Now you can add the control element to the project using the Winsock icon in the Tools collection. By default the names *Winsock1, Winsock2* etc. are assigned.

After completing this step the communication path through TCP/IP protocol is available to the new program. The necessary steps and functions for connection and disconnection as well as for sending and receiving payload data are explained using the following short TCP client program as an example..

A description of all properties, methods and events related to the Winsock Control is available using the online help function of Visual Basic. Simply select the Control in your project and press *F1*.

2.3.2 Explanation of the sample program (TCP Socket Client)

The program implements a TCP client which establishes a connection with the TCP server indicated in the text fields. Then all entered characters are sent to the server while incoming characters from the network are represented in the text window.

Note*: The following demo program is intended only to clarify the basic structure of TCP client applications (establishing the connection -> data exchange -> terminating the connection). Your own programs will have to be expanded as necessary especially with respect to controlled data transfer using the "Send Complete" event as well as error handling.*

"Connect" button, establishing and terminating the connection

```
Private Sub connect_Click()
  If Winsock1.State = sckClosed Then          'If there is no connection ...
    Winsock1.RemoteHost = IP_Nr.Text          'determine target IP address
    Winsock1.RemotePort = Val(Port_Nr.Text)   'determine target port no.
    Connect.Enabled = False                   'deactivate Connect button
    TCPSocketCLIENT.MousePointer = 11         'mouse pointer = sandglass
    Winsock1.Connect                          'Open the connection
  Else                                        'If already open,
    Winsock1.Clos                             'close the connection
    Winsock1.LocalPort = 0                    'set local port no. to 0
    Connect.Caption = "Connect"
  End If
End Sub
```

"Connect" event, connection to server was successfully established

```
Private Sub Winsock1_Connect()        'Connection successfully established
Terminal.SetFocus
  Connect.Caption = "Disconnect"
  Connect.Enabled = True              'activate Connect button
  TCPSocketCLIENT.MousePointer = 0    'Set mouse pointer to standard
End  Sub
```

Send characters over the network to the TCP server

```
Private Sub terminal_KeyPress(KeyAscii As Integer)
  If Winsock1.State = sckConnected Then  'If there is a connection...
    Winsock1.SendData Chr$(KeyAscii)     'Send characters
    KeyAscii = 0
  End If
End Sub
```

Receive characters over the network from the TCP server

```
Private Sub Winsock1_DataArrival(ByVal bytesTotal As Long)
                                      'Event"Receive data"
  Winsock1.GetData strData$           'Read into string variable
  variableTerminal.Text = Terminal.Text + strData$
                                      'Display in terminal window
End Sub
```

Error handling and outputting returned error text

```
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                  'Display message box with error string
  Winsock1.Close                      'Close TCP connection
  Connect.Enabled = True              'Activate Connect button
  TCPSocketCLIENT.MousePointer = 0    'Set mouse pointer to standard
End Sub
```

2.4 Java application environment

Description of programming environment

This example shows how a simple Java application is constructed for the COM server. The program can be run under Windows in the DOS box using a Java interpreter. The requirement is that the MS Windows TCP/IP stack is installed.

Programming environment for the example:

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

```
/*********************************************************************/
/*  ComportTcp.java         Win32 Application                        */
/*                          Borland JBuilder 4.0                     */
/*********************************************************************/
/*  Sample program for a client                                      */
/*  The IP address for the server must be transmitted as             */
/*  a parameter at program start.                                    */
/*  - Entries must be confirmed with <RETURN>.                       */
/*  - Received data are output in bytes.                             */
/*  - Receipt is processed by a separate thread.                     */
/*  - The program is ended with x+<ENTER>.                           */
/*********************************************************************/

//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();       //Get characters from stream
            if( i != 0x78 )
              { outgoing.write( i );        //and send to COM port
              }
            else
              { more = false;               //End with ESC
              }
          }
        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();       // receive bytes from the stream
          System.out.write( b, 0, 1 );      // send bytes to the standard output
        }
        incoming.close();
      }
      catch( Exception e )
      {
      }
    }
  }
```

Part 3
TCP/IP Basics

3.1 IP – Internet Protocol

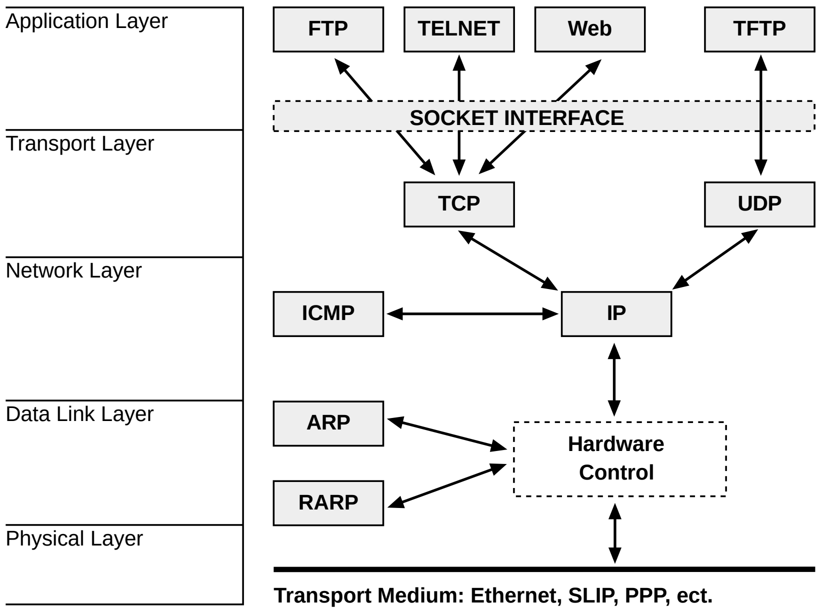
The Internet Protocol defines the basis of data communication on the lowest level. It allows, regardless of the physical medium used, various network and hardware architectures to be merged into a uniform network.

The Internet Protocol handles data transmission by means of a connectionless, non-secure transport medium. Security mechanisms are the responsibility of higher-order protocols such as TCP.

Basic elements for cross-network communication:

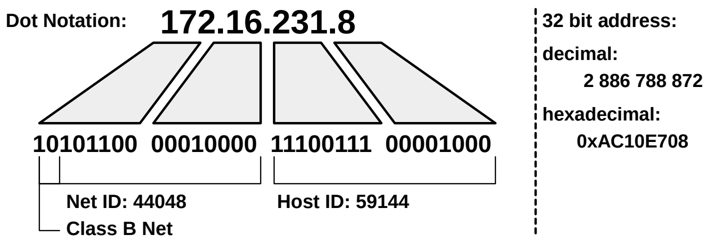
- Addressing mechanism for giving sender and receiver an unique identity
- Concept for transporting data packets through nodes (routing)
- Format for data exchange (defined header with important information)

3.1.1 The protocol layers of the Internet

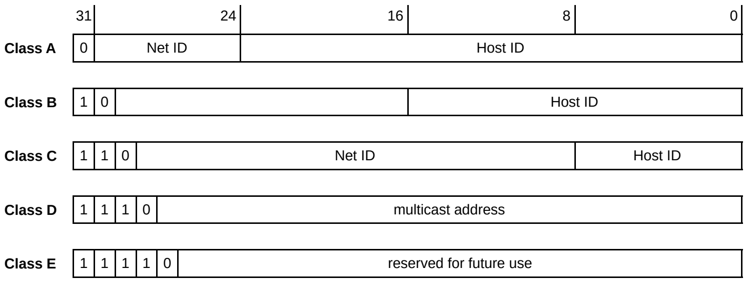


3.1.2 Internet addresses

Each host in the Internet has an address which is unique in the world. This IP address is a 32-bit value which for ease of reading is generally represented in dot notation, i.e., bytes separated by periods.



The IP address is divided into the network and the host ID. How many bits are used respectively for the network ID and the host ID depends on the class of the IP network. This network class can – as shown in the table below – be read off from the highest address bits:



The following address spaces are derived according to the definition of the network classes:

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

Generally however only IP addresses for Classes A through C are assigned. You will likely never come into contact with Classes D and E: Class D includes networks for multicasting, and Class E is reserved for research purposes.

The following Internet addresses have a special meaning and are not allowed to be assigned as an Internet host address:

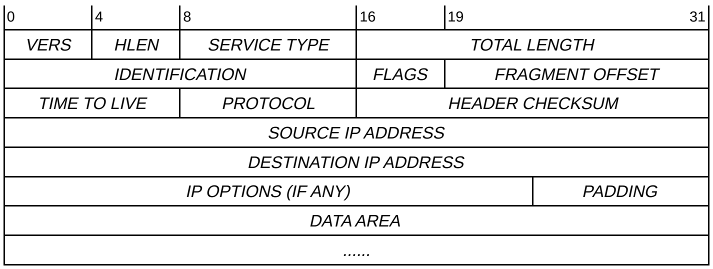
all bits 0		° addresses present host with network ID and host ID	only for startup (no valid Internet addresses)
all bits 0	host ID	° addresses host with this host ID in present network	
all bits 1		° broadcast in local network	
net ID	all bits 1	° broadcast in network given by net ID	
01111111	all bits 1	° loopback within TCP/IP protocol software (for testing purposes)	

3.1.3 The packete format of IP

A datagram consists of a packet head (Header) and the Data Area. The Header contains information about the datagram; here you will find for example the addresses of the sender and receiver, routing information, the number of the higher-order protocol for passing on the datagram, as well as special options.

datagram header	*datagram data area*

format of IP datagram header:



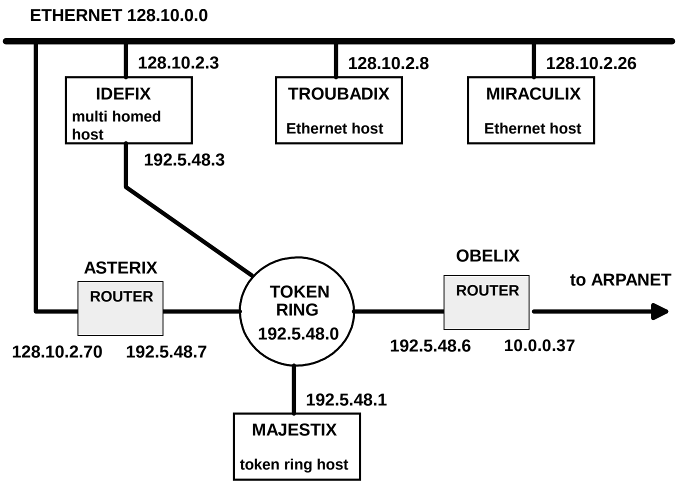
version:	Binary coded version of the IP protocol (currently 4.0)
hlen:	Length of the header in DWORDs (32-bit)
service type:	Priority of a packet and features of the desired transmission path
total length:	Total length of the IP packet including header and actual data in bytes (8-bit)
identification:	Value set by sender for identifying the individual fragments
flags (3-bit) :	Bit 2: Fragmenting allowed 0=yes, 1=no; Bit 3: 0=last fragment, 1=more fragments to follow

time to live:	Counter which is decremented at each router. Once the value 0 is reached the packet is rejected.
protocol:	No. of the higher-order protocol (e.g. TCP=6, UDP=17, ...)
header checksum:	Just what it says
source IP address:	IP address of the sender
destination IP addr.:	IP address of the receiver
IP options (variab.):	IP options if needed
padding:	Filler bytes for bringing the header length up to a multiple of DWORDs

3.2 Routing IP packets

Routing is the transport of a datagram from the sender to the receiver. A distinction is made between direct and indirect routing. Direct routing takes place within a local network, whereby no router is needed. Indirect routing takes place between two statoins in different networks, with the sender passing the IP packet on to the next router.

Whether the packet has to be routed directly or indirectly is easy to decide: The software compares the net ID of the destination with the „real“ net ID; if they are not identical, the packet is handed off to the router.



The illustration above shows an example of a network containing hosts and routers. The host IDEFIX is a „muti-homed host“: it has access to multiple networks (for example through two Ethernet cards) but does not have any router software.

The hosts IDEFIX, TROUBADIX and MIRACULIX belong to a Class B network (129.10.0.0). The token ring network is a Class C network (192.5.48.0) which is linked to the Arpanet (Class A network 10.0.0.0) thorugh the router OBELIX.

3.3 Subnets

If a local network is insufficient or too cumbersome due to its size (such as Class A networks having over 16 million hosts), it is subdivided into smaller networks, so-called subnets. Different network technologies in the individual departments, limitations with respect to cable length and the number of connected stations, as well as performance optimization are other reasons for subdividing networks into smaller segments.

Since the structure of the IP address does not allow for fitting this additional coding into the address itself, the subnet mask had to be created. It defines which bits of the host ID are to be used for coding the subnet ID and which define the host ID.

The subnet mask is determined by the administrator and represented in dot notation (e.g. 255.255.255.128) just like the ID address.

When subnets are formed the routing algorithm has to be expanded, since the net ID of the receiver can be identical to that of the current host, even though each resides in a different local network.

Binary operations with the Subnet Mask:

Host-ID =	IP-Address AND(NOT(Subnet-Mask))
Net-IDS =	IP-Address AND Subnet-Mask (combination of Net- and Subnet-ID)
Subnet-ID:	Set the Net ID in the Net IDS to 0

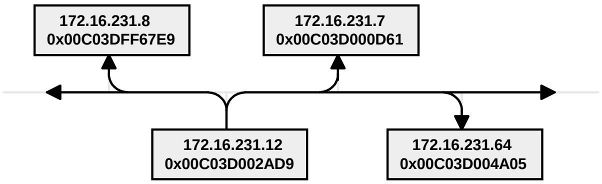
Example: values of a class B net IP address

IP address:	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 and RARP

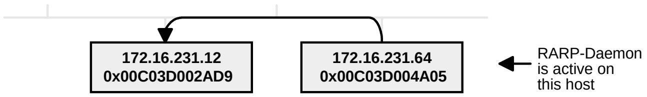
ARP and RARP (the latter is used only under UNIX) provide mechanisms for mapping IP addresses to the physical network addresses which one needs for direct routing. Each hardware-based protocol (Ethernet, X.25,k ISDN ...) has its own address format and doesn‘t understand IP addresses. If the target does not reside in the local network, you need the physical address of the router which will hand the packet over to another network.

REQUEST to all



ARP Request:	**To whom belongs the IP address 172.16.231.64?**	
Example:	Sender hardware address:	0x00C03D002AD9
	Sender IP address:	172.16.231.12
	Target hardware address:	0xFFFFFFFF
	Target IP address:	172.16.231.64
RARP-Request:	**I give my physical address. Who knows my IP address?**	
Beispiel:	Sender hardware address:	0x00C03D002AD9
	Sender IP address:	0.0.0.0
	Target hardware address:	0xFFFFFFFF
	Target IP address:	255.255.255.255

RESPONSE to sender:



ARP/RARP response:

Example:	Sender hardware address:	0x00C03D004A05
	Sender IP address:	172.16.231.64
	Target hardware address:	0x00C03D2AD9
	Target IP address:	172.16.231.12

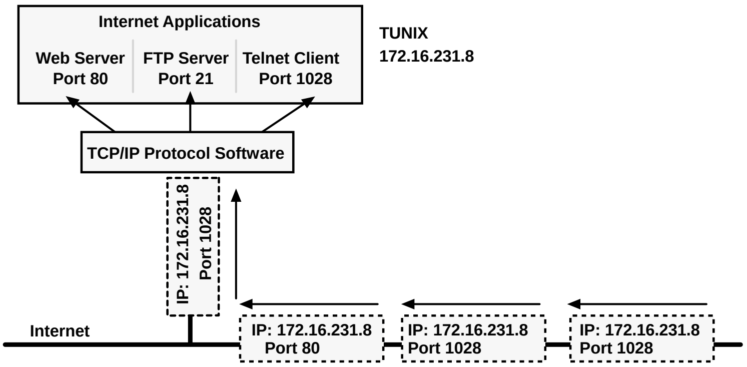
The Ethernet address and IP address assignment is stored in a table and only deleted after a timeout.

Note: *If you change this assignment (such as by assigning the same IP address to interchange device), you may no longer have a connection to the target. If the „arp“ command is not available, the only remedy is to reboot the computer or to assign a new IP address.*

3.5 Transport Layer

3.5.1 Addressing the applications with port numbers

The IP address addresses the host and only the host. But each host can have more than one application running at the same time, such as a Web browser, a Telnet client and so on. The necessary mechanisms for addressing the applications are provided by the TCP and UDP protocols.



Known applications have fixed ports assigned to which each applications can refer when establishing a connection. The range from 0 to 1023 therefore contains reserved port numbers. Under no circumstances can these be used for your own applications. The complete list of „assigned numbers“ can be found in RFC 1700 (1994).

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

Here is a short list of applications and their port numbers:

Application	Port	Protocol	Description
ftp	21	udp/tcp	File Transfer Protocol
telnet	23	udp/tcp	Teletype Network
smtp	25	udp/tcp	Simple Mail Transfer Protocol
domain	53	udp/tcp	Domain Name Server
tftp	69	udp/tcp	Trivial File Transfer Protocol
http	80	udp/tcp	HyperText Transfer Protocol
sftp	115	udp/tcp	Simple File Transfer Protocol
snmp	161	udp/tcp	Simple Network Management Protocol
...			

3.5.2 UDP format

UDP‘s features are limited to separating communications channels of the applications. UDP does not provide the service of dividing a message into datagrams and reassembling it at the other end. Specifically, UDP doesn‘t provide sequencing of the packets that the data arrives in.

This means an Internet application which uses UDP must be able to make sure that the entire message has arrived and is in the right order. UDP does save „protocol overhead“ and therefore offers higher transmission speeds than TCP. In addition there are no mechanisms for establishing and releasing a connection.

0 16 31

UDP SOURCE PORT	*UDP DESTINATION PORT*
UDP MESSAGE LENGTH	*UDP CHECKSUM*
DATA	
. . .	

Format of an UDP datagram header

Source Port:	The sender‘s port; needed for relating reply packets back to the correct connection.
Destination Port:	Port to which the source is to send the packet.
Length:	Size of the UDP datagram in bytes (header and data).
Checksum:	Checksum on the UDP datagram, used only optionally (if not used a „0“ appears in this field).

3.5.3 TCP – Transport Control Protocol

TCP frees the Internet application from the need for security mechanisms and in contrast to UDP implements a secure communications channel. This is why virtually all important Internet applications (HTTP, e-mail etc.) are based on TCP.

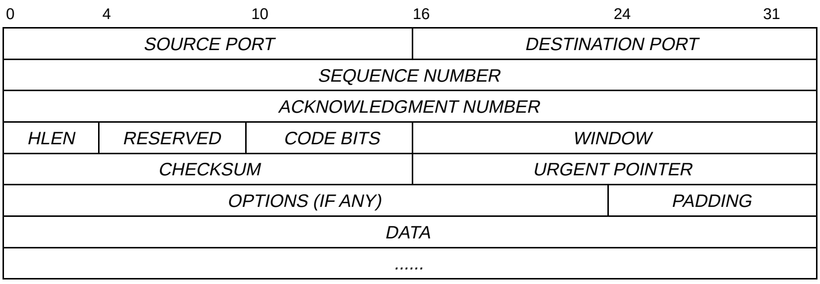
The endpoints of a TCP connection form two rows consisting of IP address and port number. A virtual connection is established between the two endpoints.

Communication is *full duplex*, which means that both communication partners can send and receive at the same time.

The protocol is transparent to the application - data brought to the TCP interface also arrive at the destination unchanged.

Packet sizes are freely selectable. As long as there are no hardware restrictions, anything from a single byte to several megabytes can be sent.

The format of a TCP packet:



Source Port:	Port number for the source application
Destination Port:	Port number for the destination application
Sequence No:	Usually specifies the number assigned to the first byte of data in the current message (guarantees proper sequencing)
Acknowl. No:	Contains the sequence number of the next byte of data the sender of the packet expects to receive. (ACK for the received bytes)
HLEN:	Size of the TCP header in DWORDs (32 Bit), start of the data area
Code Bits:	Indicate the purpose and contents of the packet:

Bit 1:	URG	segment contains urgent data, see field *Urgent Pointer* SOCKET-INTERFACE: out of band data
2:	ACK	segment contains an Acknowledge
3:	PSH	push received data immediately
4:	RST	reset connection
5:	SYN	open connection and synchronize sequence numbers
6:	FIN	no further data from sender, close connection

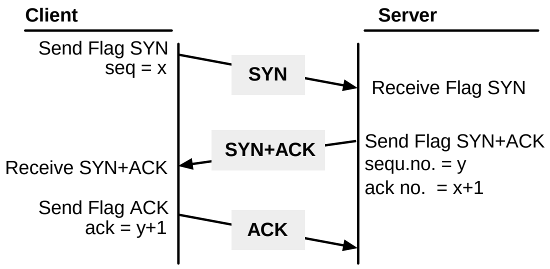
Window:	Specifies the size of the sender‘s receive window (that is, the buffer space for incoming data, starting from the byte shown in the Acknowl. No. field)
Checksum:	Checksum on the TCP datagram and a pseudo-header (indicates whether the header was damaged in transit)
Urgent Pointer:	Points to the first urgent data byte in the packet
Options:	Specifies various TCP options (most important: maximum segment size)

Establishing and closing a connection

TCP uses fixed mechanisms for establishing a connection between client and server. Establishing a connection also serves to synchronize both ends to make sure each is ready to transmit data and knows that the other side is ready to transmit as well. Another important point in this step is the negotiation of transmission parameters such as packet length and buffer size.

- **Establish:**

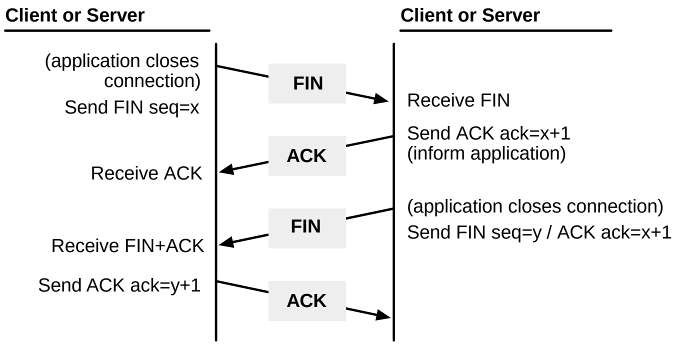
  The client sends a packet with the initial sequence number (X) and SYN bit set to indicate a connection request. Each side synchronizes itself to the *Sequence-No.* of the other station.



  In the option *Maximum Segment Size* each side can specify how many bytes it can receive in a segment following the TCP header.

- **Close:**

  Either the client or the server can initiate closing of a connection. To do this the FIN flag is set. Only when both sides have set this flag is the connection considered closed.



Flow control

TCP has various mechanisms for ensuring secure and efficient data transmission. Here are a few of the most important rules:

- The sender must keep all data available until the data have been acknowledged by the receiver.
- In the case of defective packets (such as packets with an erroneous checksum) the receiver sends back the last Acknowledgement number, whereupon the sender repeats the packet.
- If packets are lost, after a timeout the sender resends all the packets which followed the last received acknowledgement.
- The receiver uses the *Window* field to indicate for each packet how much buffer space it has left. If this field contains 0, the sender stops transmitting until it gets a packet from the receiver showing a *Window* value greater than zero.
- Since the sender is constantly updated as to the current buffer size of the destination, it doesn‘t need to wait for the acknowledgement of each individual packet, but rather can keep sending data until the buffer is full. The receiver is then acknowledging only a part of the byte-stream and not the individual packets. This means multiple bytes or packets can be sent without waiting for an acknowledgement. This method is called *sliding window* operation.

Bibliography

- **INERNET intern**
  Tischer und Jennrich
  Publisher: DATA Becker
  ISBN 3-8158-1160-0
  (in German, out of print)

- **Inside Visual C++**
  David J. Kruglinski
  Publisher: Microsoft Press
  ISBN 3-86063-394-5

- **Internetworking with TCP/IP**
  Publisher: 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