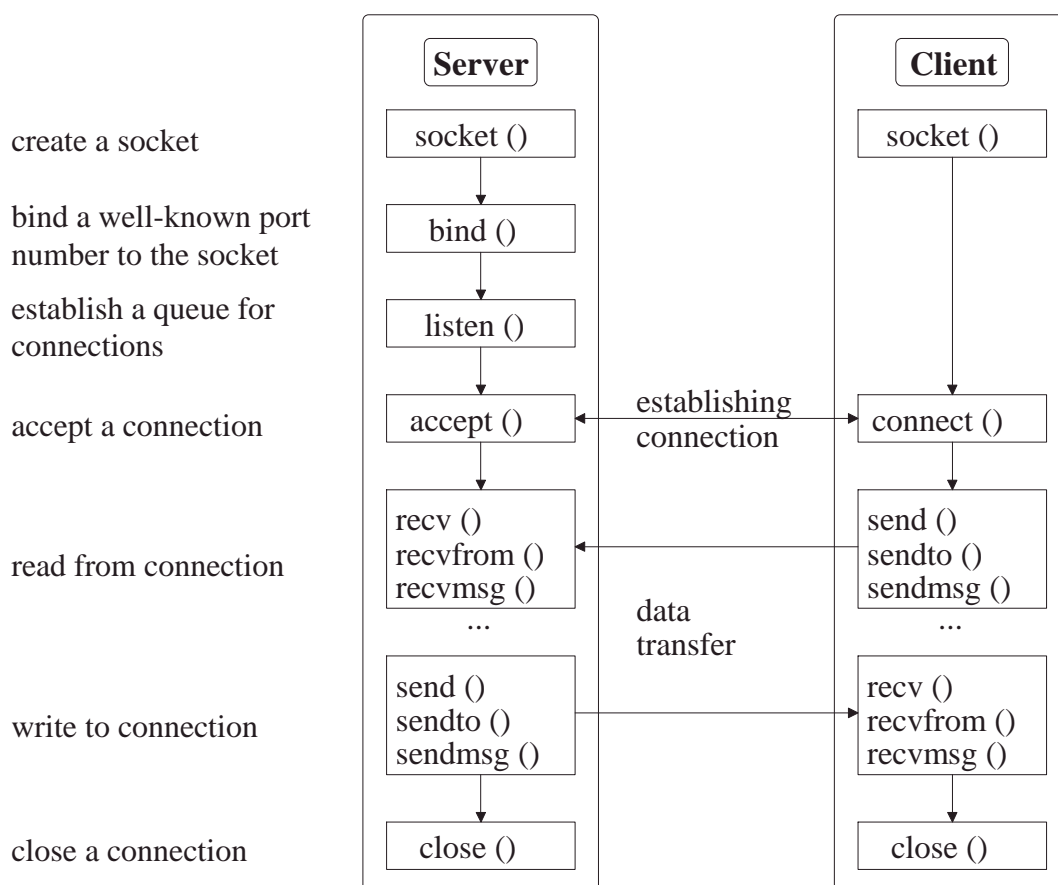


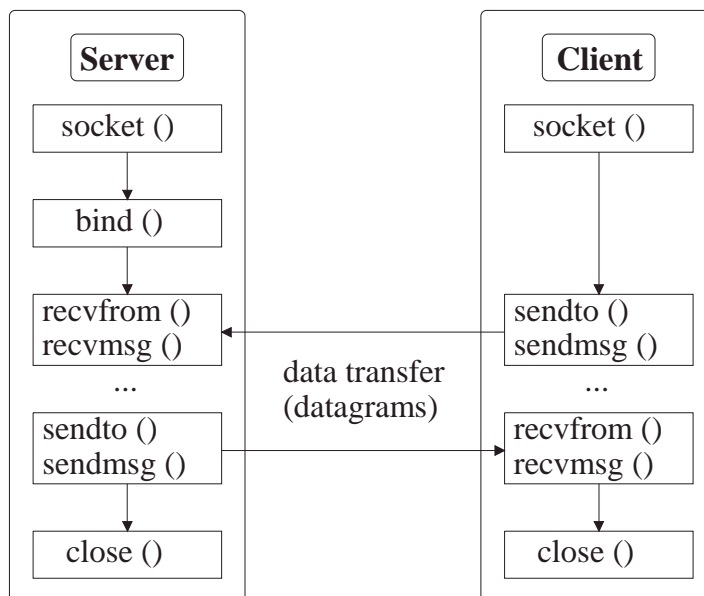
4 Die *Socket*-Schnittstelle

4.1 *Sockets* in der Programmiersprache C

- *Sockets* wurden zum ersten Mal im Jahr 1982 mit BSD-UNIX 4.1c für VAX-Rechner der ehemaligen Firma *Digital Equipment* (dann *Compaq* und jetzt HP) zur Verfügung gestellt
- *Socket*-Schnittstelle wurde mehrfach überarbeitet
- *Socket*-Systemaufrufe bei verbindungsorientiertem Protokoll (TCP/IP)



- *Socket-Systemaufrufe bei verbindungslosem Protokoll (UDP/IP)*



- *einige Datenstrukturen*

```

struct sockaddr                                /* common socket address      */
{
    u_short sa_family;                          /* address family: AF_XXX value */
    char    sa_data[14];                       /* up to 14 bytes of protocol   */
};                                              /*   specific address          */

```

```

struct in_addr                                /* internet address          */
{
    u_long s_addr;                             /* network byte ordered netid/hostid */
};

```

falls *s_addr* der Wert *INADDR_ANY* zugewiesen wird, werden Verbindungen über jede Netzwerkschnittstelle akzeptiert (im Allgemeinen hat eine Maschine nur eine Schnittstelle, wenn sie nicht als *Gateway* konfiguriert ist)

```

struct sockaddr_in                            /* internet style socket address */
{
    short    sin_family;                       /* AF_INET                    */
    u_short  sin_port;                         /* network byte ordered port number */
    struct in_addr sin_addr;                   /* network byte ordered netid/hostid */
    char     sin_zero[8];                     /* unused (padding to 14 bytes)   */
};

```

- einige Funktionen

- einen Kommunikationsendpunkt einrichten

int **socket** (int family, int type, int protocol);

family: AF_INET

...

type SOCK_STREAM
 SOCK_DGRAM

...

protocol im Allgemeinen "0"

(Der Wert kann ungleich Null sein, wenn ein beliebiges Protokoll mit sogenannten "*raw sockets*" (SOCK_RAW) benutzt wird, z. B. ICMP. "Raw sockets" benötigen Administratorrechte ("root"-Rechte).)

- ◆ legt das Kommunikationsprotokoll fest
 - ◆ liefert einen *Socket*-Deskriptor zurück
- einem unbekanntem *Socket* eine Adresse zuweisen

int **bind** (int sockfd, struct sockaddr *myaddr, int addrlen);

sockfd *Socket*-Deskriptor

myaddr protokollspezifische Adresse

addrlen protokollspezifische Größe der Adressstruktur

- ◆ die Adresse besteht aus einer IP-Adresse und einer *Port*-Nummer

- eine Warteschlange für Verbindungswünsche einrichten

int **listen** (int sockfd, int backlog);

sockfd *Socket*-Deskriptor

backlog max. Anzahl Verbindungsanforderungen, die in die Warteschlange eingestellt werden dürfen

- ◆ signalisiert die Empfangsbereitschaft des *Servers*
 - ◆ Warteschlange notwendig, falls *Server* gerade beschäftigt ist
(blockierender *Server*: bearbeiten der Anforderung, ...
nicht-blockierender *Server*: erzeugen eines Sohnprozesses, ...)
 - ◆ Verbindungswünsche werden abgelehnt, wenn die maximale Anzahl wartender Verbindungsanforderungen erreicht ist
- auf Verbindungswünsche von *Clients* warten

int **accept** (int sockfd, struct sockaddr *peer, int *addrlen);

sockfd *Socket*-Deskriptor

peer protokollspezifische Adresse (des rufenden *Clients*)

addrlen protokollspezifische Größe der Adressstruktur

- ◆ die erste Anforderung wird aus der Warteschlange genommen
- ◆ es wird ein neuer *Socket* mit denselben Eigenschaften wie *sockfd* erzeugt und als Rückgabewert geliefert
- ◆ liefert die Adresse des *Clients*

- eine Verbindung zu einem *Server-Port* aufbauen

int **connect** (int sockfd, struct sockaddr *servaddr, int addrlen);

sockfd *Socket*-Deskriptor

servaddr protokollspezifische Adresse des *Servers*

addrlen protokollspezifische Größe der Adressstruktur

- ◆ es wird versucht, eine Verbindung zu einem anderen *Socket* aufzubauen
 - ◆ Kommunikationsparameter (z. B. Puffergröße, ...) müssen zwischen *Client* und *Server* abgestimmt werden
- Daten lesen/schreiben

int **recv** (int sockfd, char *buffer, int nbytes, int flags);

int **recvfrom** (int sockfd, char *buffer, int nbytes, int flags,
 struct sockaddr *from, int *fromlen);

int **send** (int sockfd, const char *buffer, int nbytes, int flags);

int **sendto** (int sockfd, const char *buffer, int nbytes, int flags,
 const struct sockaddr *to, int tolen);

sockfd *Socket*-Deskriptor

buffer Adresse des Empfangs-/Sendepuffers

nbytes Empfangspuffergröße/Anzahl zu sendender Bytes

flags	im Allgemeinen "0" (Ungleich Null für „out-of-band“-Daten oder ähnliches. „Out-of-band“ ist im Prinzip eine „Überholspur“ für spezielle Daten, z. B. „<Strg-c>“ „<Strg-s>“, „<Strg-q>“ usw.)
from/to	protokollspezifische Adresse des Absenders/Empfängers (falls <i>from</i> ungleich <i>NULL</i> ist, wird in die Datenstruktur die Absenderadresse der Nachricht eingetragen)
fromlen	enthält beim Aufruf die Puffergröße für die Adresse und bei der Rückkehr die Größe der Absenderadresse
toalen	protokollspezifische Größe der Adressstruktur

- ◆ *send/recv* können nur bei verbindungsorientierten Protokollen benutzt werden
- ◆ *sendto/recvfrom* können bei verbindungsorientierten und verbindungslosen Protokollen benutzt werden
- ◆ alle Funktionen geben die Anzahl der gesendeten/empfangenen Bytes zurück

- Byte-Ordnungsroutinen

```
u_short htons (u_short hostshort);      /* byte order: host -> network */
u_long  htonl  (u_long  hostlong);
u_short ntohs (u_short netshort);      /* byte order: network -> host */
u_long  ntohl  (u_long  netlong);
```

- Adressumwandlungen

```
unsigned long inet_addr (char *);        /* IP-Adresse -> intern */
char          *inet_ntoa (struct in_addr); /* intern -> IP-Adresse */
```

- **Beispiel:** spezielle Festlegungen für die *Socket*-Programme (tcp_cliserv.h)

```

...
#ifndef PARSEPV
#define PARSEPV      1          /* 0: blocking server      */
#endif              /* 1: parallel server      */

#define DEFIPADDR    "127.0.0.1" /* default IP address      */
#define DEFPORT      8888        /* default port             */
#define BUFLLEN      80         /* buffer size              */
#define ADDRLEN      16         /* IP Addr. length (+1 for \0) */
#define MAXQUE       3         /* max. # of waiting processes */

/* operating system dependent type (special problem with Linux and
 * Solaris x86 because the type depends on the release: "socklen_t"
 * isn't defined in older versions)
 *
 * Linux up to 2.0.x and Solaris x86 up to 2.5.1:
 * #if defined(SunOS) && defined(sparc)
 * remark: results in warnings on newer systems but is nevertheless the
 * "secure" way for old and new systems
 *
 * starting with Linux 2.2.x and Solaris x86 2.7:
 * #if defined(SunOS) || defined(Linux)
 * remark: results in errors on older systems
 */
#if defined(SunOS) || defined(Linux) || defined(Darwin)
#define SOCKLEN_T socklen_t *
#else
#define SOCKLEN_T int *
#endif

/* operating system dependent function */
#ifdef Win32
#ifndef WIN32
#define WIN32
#endif
#define CLOSE      closesocket
#else
#define CLOSE      close
#endif

#ifdef Win32
#define WMAJOR      2          /* WinSocket version      */
#define WMINOR      2          /* WinSocket version      */
#undef  PARSEPV      /* only the blocking server */
#define PARSEPV      0          /* has been implemented    */
#endif

/* return values ("exval") for macro "TestMinusOne" */
#define ESOCK      -1          /* error calling "socket ()" */
...

/* evaluate the return value of a function */
#define TestMinusOne(val,line,function,exval) \
    if (val == -1) { fprintf (stderr, "line %d: \"%s ()\" failed: %s\n", \
        line, function, strerror (errno)); exit (exval); }

```

- **Beispiel:** verbindungsorientierter Echo-Server in C
 - Makro *TestMinusOne* wurde hier weggelassen
 - für nicht-blockierenden Server werden Sohnprozesse erzeugt

```

...
#include "tcp_cliserv.h"
#if defined(Linux) || defined(SunOS) || defined(Cygwin) || ...
    #include <sys/socket.h>
    ...
#endif
#ifdef Win32
    #include <winsock2.h>
    #include <process.h>
#endif

int listenfd;                                /* listen() socket descriptor */
/* signal handler for <Ctrl-c> or <Strg-c> resp. to close "listenfd" */
void MySigInt (int sig);

int main (int argc, char *argv[])
{
    int                connfd,                /* connect() socket descriptor */
                cliLen,                    /* address length */
    ...
    unsigned short    serv_port;            /* server port */
    struct sockaddr_in cli_addr,            /* client/server address */
                serv_addr;

    #if PARSESERV == 1
        int            finish;                /* terminate connection server */
        pid_t          childpid;            /* process ID */
    #endif
    #ifdef Win32
        WORD            WinSockVers;        /* socket version */
        WSADATA         wsaData;            /* for WinSockets */
    #endif
    ...

    #ifdef Win32
        WinSockVers = MAKEWORD (WMAJOR, WMINOR);
        ret = WSASStartup (WinSockVers, &wsaData);
        if (ret != 0)
        {
            fprintf (stderr, "line: %d: \"WSASStartup ()\" failed: Couldn't "
                "find library.\n", __LINE__);
            exit (EWINSOCK);
        }
        if ((LOBYTE (wsaData.wVersion) != WMAJOR) ||
            (HIBYTE (wsaData.wVersion) != WMINOR))
        {
            fprintf (stderr, "line: %d: Library doesn't support "
                "WinSock %d.%d\n", __LINE__, WMAJOR, WMINOR);
            WSACleanup ();
            exit (EWINSOCK);
        }
    #endif

    ...
    /* open socket */
    listenfd = socket (AF_INET, SOCK_STREAM, 0);
    /* install signal handler */
    ...

```

```

memset ((char *) buffer, 0, sizeof (buffer));
memset ((char *) &serv_addr, 0, sizeof (serv_addr));
/* bind port to socket and wait for connections */
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = htonl (INADDR_ANY);
serv_addr.sin_port = htons (serv_port);
ret = bind (listenfd, (struct sockaddr *) &serv_addr,
           sizeof (serv_addr));
printf ("TCP server is ready and waiting for connections ...");
ret = listen (listenfd, MAXQUE);
#if PARSEPV == 1
  finish = 0;
  while (finish == 0)
  {
    clilen = sizeof (cli_addr);
    connfd = accept (listenfd, (struct sockaddr *) &cli_addr,
                   (SOCKLEN_T) &clilen);

    childpid = fork();
    switch (childpid)
    { ...
      case 0: /* child process */
        { ...
          ret = CLOSE (listenfd);
          ...
          printf ("New server process for client request ...");
          more_to_do = 1;
          while (more_to_do == 1)
          {
            len = recv (connfd, buffer, BUFLen, 0);
            ...
            for (i = 0; i < len; ++i)
            {
              buffer[i] = (char) toupper (buffer[i]);
            }
            if (send (connfd, buffer, len, 0) == -1)
            {
              fprintf (stderr, "line %d: failure in ...");
            }
            if ((strcmp (buffer, "QUIT\n") == 0) ||
                (strcmp (buffer, "QUIT") == 0))
            {
              ret = CLOSE (connfd);
              printf ("TCP server %ld: " ...);
              more_to_do = 0; /* terminate process */
            }
            memset ((char *) buffer, 0, sizeof (buffer));
          }
          finish = 1;
          break;
        }
      default: /* parent process */
        ret = CLOSE (connfd); /* accept another connection */
    }
  }
  printf ("TCP server %ld has finished\n", (long) getpid ());
#else
  ...
#endif
return EXIT_SUCCESS;
}
...

```

- **Beispiel:** verbindungsorientierter *Client*

```

...
int main (int argc, char *argv[])
{
    ...

    /* initialize "serv_addr" */
    memset ((char *) &serv_addr, 0, sizeof (serv_addr));
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr (serv_ip_addr);
    serv_addr.sin_port        = htons (serv_port);

    /* open socket */
    connfd = socket (AF_INET, SOCK_STREAM, 0);

    /* connect to server */
    ret = connect (connfd, (struct sockaddr *) &serv_addr,
                  sizeof (serv_addr));
    printf ("connected to server at %s on port %d\n", serv_ip_addr,
           serv_port);

    more_to_do = 1;
    while (more_to_do == 1)
    {
        printf ("string to send (\"quit\" terminates connection): ");
        fgets (buffer, BUFLLEN, stdin);
        ...
        if (len > 0)
        {
            if (send (connfd, buffer, len, 0) == -1)
            {
                fprintf (stderr, "line %d: failure in \"send ()\": %s\n",
                        __LINE__, strerror (errno));
            }
            if (recv (connfd, buffer, BUFLLEN, 0) == -1)
            {
                fprintf (stderr, "line %d: failure in \"recv ()\": %s\n",
                        __LINE__, strerror (errno));
            }
            else
            {
                printf ("Received from echo server at %s: %s\n\n",
                        inet_ntoa (serv_addr.sin_addr), buffer);
            }
            if ((strcmp (buffer, "QUIT\n") == 0) || /* terminate process ? */
                (strcmp (buffer, "QUIT") == 0))
            {
                more_to_do = 0; /* yes */
            }
        }
        else
        {
            printf (" !!! I can't send an empty string ...");
        }
    }
    ret = CLOSE (connfd);
    return EXIT_SUCCESS;
}

```

- Übersetzung der Programme unter Windows NT (Microsoft C/C++)

```
cl /W3 /DWin32 sockserv.c /link wsock32.lib
cl /W3 /DWin32 sockcli.c /link wsock32.lib
```

- Übersetzung der Programme unter UNIX

```
cc -fd -fast -xtarget=generic -v -Xc -DSunOS sockserv.c -lnsl -lsocket
cc -fd -fast -xtarget=generic -v -Xc -DSunOS sockcli.c -lnsl -lsocket
```

```
gcc -Wall -Wstrict-prototypes -Wmissing-prototypes -std=c99 -pedantic -DLinux sockserv.c
gcc -Wall -Wstrict-prototypes -Wmissing-prototypes -std=c99 -pedantic -DLinux sockcli.c
```

```
gcc -Wall -Wstrict-prototypes -Wmissing-prototypes -std=c99 -pedantic -DDarwin sockserv.c
gcc -Wall -Wstrict-prototypes -Wmissing-prototypes -std=c99 -pedantic -DDarwin sockcli.c
```

- ein nicht-blockierender *Server* kann viele Verbindungen gleichzeitig verwalten

- der obige nicht-blockierende *Server* ist mangelhaft implementiert

(das Problem tritt nicht auf, wenn (*detached*) *Threads* anstelle der parallelen Prozesse benutzt werden)

- einige Verbindungen erzeugen und beenden
- z. B. "ps ax | grep sockserv" auf einem *Linux-Server* ausführen

```
196  1 S   0:00 sockserv
210  1 Z   0:00 (sockserv <zombie>)
235  1 Z   0:00 (sockserv <zombie>)
239  1 Z   0:00 (sockserv <zombie>)
```

⇒ die *Zombie*-Prozesse können nur vernichtet werden, wenn der *Server* beendet und neu gestartet wird

⇒ die *Zombie*-Prozesse füllen alle Einträge der Prozesstabelle

⇒ der *Server* muss auf das Ende seiner Sohnprozesse warten

- verhindern von *Zombie*-Prozessen

- SIGCHLD meldet jede Zustandsänderung von Sohnprozessen

⇒ kann benutzt werden, um auf Prozessende zu warten

```

...
void sigchld_handler (int sig); /* new signal handler */
...

int main (void)
{
    ...
    if (signal (SIGCHLD, sigchld_handler) == SIG_ERR)
        ...
}

void sigchld_handler (int sig)
{
    while (waitpid ((pid_t) -1, NULL, WNOHANG) > 0)
    {
        ; /* pick up childs */
    }
    if (signal (SIGCHLD, sigchld_handler) == SIG_ERR)
    {
        perror ("sigchld_handler");
        exit (-1);
    }
}

```

- hat einen **Seiteneffekt**: einige Systemfunktionen können durch Signale unterbrochen werden

⇒ diese Situation benötigt eine spezielle Behandlung

⇒ solche Aufrufe müssen z. B. in einer *while*-Schleife erfolgen

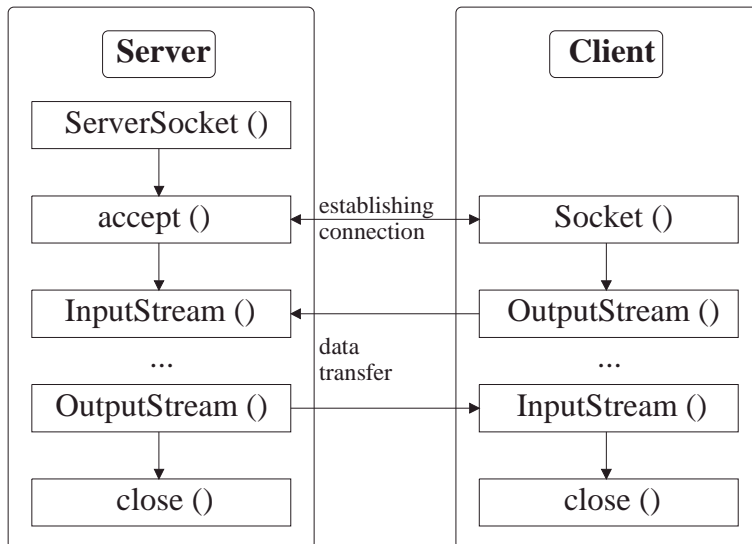
```

while ((connfd = accept (...)) < 0)
{
    if (errno != EINTR)
    {
        perror ("..."); /* an error has occurred */
        exit (...);
    }
}

```

4.2 Sockets in der Programmiersprache Java

- Sockets in Java mit verbindungsorientiertem Protokoll (TCP/IP)



- vereinfachte Programmierung gegenüber C
- `ServerSocket ()` führt automatisch die Funktionen `socket ()`, `bind()` und `listen ()` aus
- `Socket ()` führt die Funktionen `socket ()` und `connect ()` aus

- **Konstruktoren für die Klasse *ServerSocket***
 - ***ServerSocket* (int port) throws IOException;**
(der Parameter 0 erzeugt einen *Socket* auf irgendeinem freien *Port*; *port* muss zwischen 0 und 65535 liegen)
 - ***ServerSocket* (int port, int backlog) throws IOException;**
(*backlog* gibt die max. Länge der Warteschlange für Verbindungsanforderungen an)
 - ***ServerSocket* (int port, int backlog, InetAddress bindAddr) throws IOException**
(*bindAddr* kann auf einem Rechner mit mehreren Netzadressen (*multi-homed host*) benutzt werden, wenn der *Server* nur Anfragen an eine Adresse akzeptieren soll; falls *bindAddr* den Wert *NULL* hat, werden Anfragen über alle Adressen akzeptiert)

- **einige Methoden der Klasse *ServerSocket***

Socket <i>accept</i> ()	wartet auf einen Verbindungswunsch; liefert einen <i>Socket</i>
void <i>close</i> ()	schließt den <i>ServerSocket</i>
InetAddress <i>getInetAddress</i> ()	liefert die IP-Adresse oder <i>null</i>
int <i>getLocalPort</i> ()	liefert die <i>Port</i> -Nummer, auf der der <i>Server</i> Verbindungswünsche erwartet
String <i>toString</i> ()	liefert eine Zeichenfolge, die diesen <i>Socket</i> beschreibt

- einige Konstruktoren für die Klasse *Socket*
 - **Socket** (InetAddress addr, int port) throws IOException;
(erzeugt einen *StreamSocket* und verbindet ihn mit *addr* und *port* des *Servers*)
 - **Socket** (InetAddress addr, int port, InetAddress localAddr, int localPort) throws IOException;
(erzeugt einen *StreamSocket* und verbindet ihn mit *addr* und *port* des *Servers* sowie mit *localAddr* und *localPort* des *Clients*)
 - **Socket** (String host, int port) throws UnknownHostException, IOException;
(analog, wobei anstelle der IP-Adresse der Name des *Servers* angegeben wird)
 - **Socket** (String host, int port, InetAddress localAddr, int localPort) throws IOException;

- einige Methoden der Klasse *Socket*

<code>void close ()</code>	schließt den <i>Socket</i>
<code>InetAddress getInetAddress ()</code>	liefert die IP-Adresse des <i>Servers</i>
<code>InetAddress getLocalAddress ()</code>	liefert die IP-Adresse des <i>Clients</i>
<code>int getPort ()</code>	liefert die <i>Port</i> -Nummer des <i>Servers</i>
<code>int getLocalPort ()</code>	liefert die <i>Port</i> -Nummer des <i>Clients</i>
<code>InputStream getInputStream ()</code>	liefert einen Datenstrom, um über den <i>Socket</i> Bytes zu lesen

OutputStream `getOutputStream ()` liefert einen Datenstrom, über den Bytes gesendet werden können

int `getReceiveBufferSize ()` liefert die Größe des Empfangspuffers

int `getSendBufferSize ()` liefert die Größe des Sendepuffers

String `toString ()` liefert eine Zeichenfolge, die diesen *Socket* beschreibt

- einige Methoden der Klasse *InetAddress*

byte[] `getAddress ()` liefert die IP-Adresse als Feld von vier Bytes, wobei das höchst wertige Byte im ersten Feldelement liegt

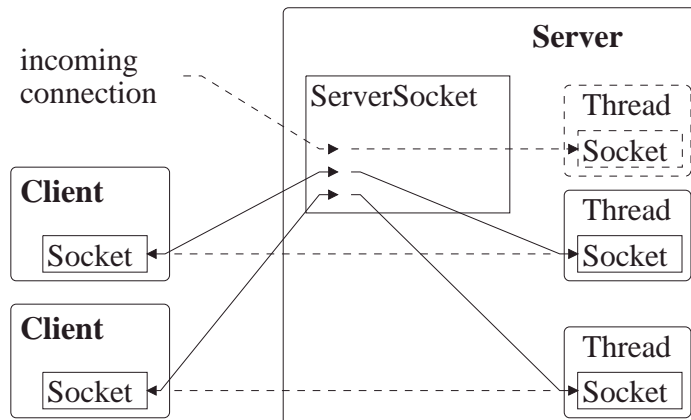
InetAddress `getByName (String host)` liefert ein *InetAddress*-Objekt für den Rechner *host*; der Parameter *null* liefert die Adresse des lokalen Rechners

String `getHostAddress ()` liefert die IP-Adresse in der Form "%d.%d.%d.%d"

String `getHostName ()` liefert den zu dieser Adresse gehörigen Namen oder *null*

String `toString ()` wandelt die IP-Adresse in eine Zeichenfolge um

- **Beispiel:** nicht-blockierender *Echo-Server*



```

/* File: ServerMain.java                               Author: S. Gross          */
public class ServerMain
{
    public static void main (String args[])
    {
        final int DEFPORT = 8888;                        /* default port          */
        final int MAXQUE  = 3;                          /* max. # of waiting processes */
        int      servPort,                               /* server port           */
              thrID;                                     /* Thread ID             */

        if (args.length != 1)
        {
            servPort = DEFPORT;
        }
        else
        {
            try
            {
                servPort = Integer.parseInt (args[0]);
            } catch (NumberFormatException e)
            {
                ...
            }
        }
        thrID = 0;
        ...
        try
        {
            /* open socket */
            ServerSocket listenfd = new ServerSocket (servPort, MAXQUE);
            while (true)
            {
                /* wait for connection requests */
                Socket connfd = listenfd.accept ();
                (new ConnectionServer (++thrID, connfd)).start ();
            }
        } catch (IOException e1)
        {
            ...
        }
    }
}

```

```

/* File: ConnectionServer.java          Author: S. Gross          */
class ConnectionServer extends Thread
{
    final int    BUFLLEN = 80;          /* buffer size          */
    private int  thrID;                 /* Thread ID           */
    private Socket connfd;              /* connection socket descriptor */

    public ConnectionServer (int thrID, Socket connfd)
    {
        ...
    }

    public void run ()
    {
        byte    buffer[] = new byte[BUFLLEN]; /* request from client */
        byte    tmp[];           /* temporary buffer     */
        boolean noErrors = true;
        int     len;              /* # of chars in buffer */
        String  msg               /* received message     */
        String  cli_addr = "";    /* IP address of client */
        String  name              /* getName ()           */

        try
        {
            InputStream in = connfd.getInputStream ();
            OutputStream out = connfd.getOutputStream ();
            cli_addr = connfd.getInetAddress ().toString ();
            System.out.println (name + ": new connection with " + cli_addr);
            while ((msg.compareTo ("QUIT\n") != 0) &&
                (msg.compareTo ("QUIT") != 0) && noErrors)
            {
                if ((len = in.read (buffer, 0, BUFLLEN)) != -1)
                {
                    msg = new String (buffer, 0, len);
                    System.out.println (name + ": received: " + msg);
                    msg = msg.toUpperCase ();
                    /* sometimes toUpperCase converts a german sharp s to SS */
                    if ((len = msg.length ()) > BUFLLEN)
                    {
                        len = BUFLLEN;
                        System.err.println (name + ": buffer too small. " +
                            "Message cut to buffer space.");
                    }
                    /* "buffer" may be too small if the message contains a german
                     * sharp s -> use temporary buffer of appropriate size
                     */
                    tmp = msg.getBytes ();
                    out.write (tmp, 0, len);
                }
                else
                {
                    System.out.println (...);
                    noErrors = false;
                }
            }
            connfd.close ();
            System.out.println (name + ": terminated");
        } catch (IOException e)
        {
            System.err.println (e.toString ());
        }
    }
    ...
}

```

- **Beispiel:** das zugehörige *Client*-Programm

```

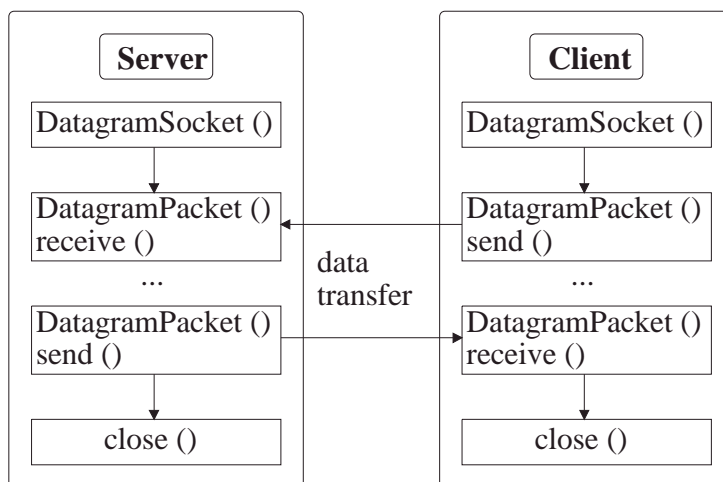
/* File: ClientMain.java                               Author: S. Gross          */
public class ClientMain
{
    public static void main (String args[])
    {
        final int     DEFPORT    = 8888;                /* default port          */
        final String  DEFIPADDR  = "127.0.0.1";        /* default IP address   */
        final int     BUFLLEN    = 80;                /* buffer size          */
        byte[]        buffer     = new byte[BUFLLEN]; /* reply from server    */
        String        msg        = "";                /* keyboard message from user */
        String        hostAddr;   /* name or IP address   */
        int           servPort;   /* server port          */
        int           len;        /* # of chars in buffer */

        switch (args.length)
        {
            case 1:
                ...
        }
        try
        {
            /* open socket */
            Socket connfd = new Socket (hostAddr, servPort);
            InputStream in = connfd.getInputStream ();
            OutputStream out = connfd.getOutputStream ();
            BufferedReader con = new BufferedReader (
                new InputStreamReader (System.in));
            System.out.println (...);

            while ((msg.compareTo ("QUIT\n") != 0) &&
                (msg.compareTo ("QUIT") != 0))
            {
                System.out.print (...);
                msg = con.readLine ();                /* keyboard input      */
                if (msg.length () > 0)
                {
                    if ((len = msg.length ()) > BUFLLEN)
                    {
                        len = BUFLLEN;
                        System.err.println (...);
                    }
                    out.write (msg.getBytes (), 0, len); /* send to server      */
                    len = in.read (buffer, 0, BUFLLEN); /* reply from server   */
                    msg = new String (buffer, 0, len);
                    System.out.println ("Received from echo server " ...);
                }
                else
                {
                    System.out.println ("Empty input line, i.e. nothing to do.");
                }
            }
            connfd.close ();
        } catch (IOException e1)
        {
            ...
        }
    }
}

```

- *Sockets* in Java mit verbindungslosem Protokoll (UDP/IP)



- einfache Netzwerkverbindung mit wenig Verwaltungsaufwand
- unzuverlässig (u. U.: Verlust, Änderung der Paketreihenfolge)
- **Senden:** Datagramm-Paket erstellen mit
 - ◆ Daten
 - ◆ Länge des Datenfeldes
 - ◆ Adresse des Empfängers
 - ◆ *Port*-Nummer des Empfängers
- **Empfangen:** Datagramm-Paket erstellen mit
 - ◆ Empfangspuffer
 - ◆ Größe des Empfangspuffers

⇒ nach dem Empfang enthält der Puffer die Daten sowie die Adresse und *Port*-Nummer des Absenders
- **DatagramSocket** ist wiederverwendbar für beliebige Pakete an beliebige Empfänger

- Nutzung eines *Sockets* ansehen (nur unter Linux)

```
gross@kea:/home/gross/socket > tcp_sockserv
I'm a non-blocking echo server. PID: 444
I stop my work, if you type <Ctrl-c> or <Strg-c>.

gross@kea:/home/gross/socket > socklist
type  port      inode      uid      pid      fd      name
tcp   8888      60046     500      444      3      tcp_sockserv
tcp   6000      38283     500      0        0
...

gross@kea:/home/gross/socket > java ServerMain
I'm a non-blocking echo server listening at port 8888
I stop my work, if you type <Ctrl-c> or <Strg-c>.

gross@kea:/home/gross/socket > socklist
type  port      inode      uid      pid      fd      name
tcp   8888      67177     500      467      4      java
tcp   6000      38283     500      0        0
...
```

- "bind" meldet einen Fehler

```
gross@kea:/home/gross/socket > tcp_sockserv
I'm a non-blocking echo server. PID: 493
I stop my work, if you type <Ctrl-c> or <Strg-c>.

line 144: "bind ()" failed: Address already in use

gross@kea:/home/gross/socket > socklist
type  port      inode      uid      pid      fd      name
tcp   8888      0          0        0        0
tcp   6000      38283     500      0        0
```

- dem *Socket* ist kein Prozess zugeordnet
- kommt manchmal vor, wenn *Server* mit <Ctrl-c> bzw. <Strg-c> beendet und gleich wieder gestartet wird
- nach einer kurzen Wartezeit (ca. eine Minute) steht der *Socket* wieder zur Verfügung

Aufgabe 4-1:

Realisieren Sie die *Echo-Client/Server*-Programme in der Programmiersprache C mit Hilfe von *POSIX-Threads* anstelle von Sohnprozessen.

Aufgabe 4-2:

Realisieren Sie die *Echo-Client/Server*-Programme in der Programmiersprache C mit Hilfe des verbindungslosen *Socket*-Protokolls.