Programmentwicklung II

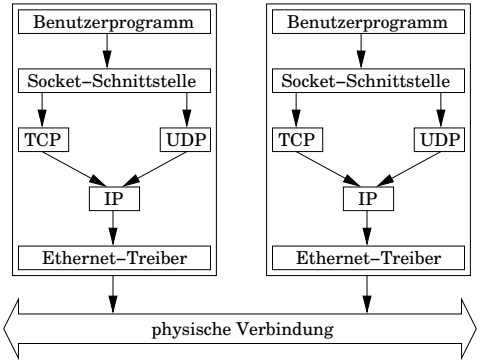
Bachelor of Science

Prof. Dr. Rethmann / Prof. Dr. Brandt

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

Sommersemester 2021

Prozesse verschiedener Rechner kommunizieren über *Sockets*. Die Zugangspunkte für die Netzwerkverbindung heißen *Ports*.



- Socket-Schnittstellen (API) bietet jedes Betriebssystem an.
- Verbindungsdetails übernehmen Kernel und Treiber.
- Austausch von kleinen Paketen auf unterer Ebene.
- In dieser Vorlesung werden nur Programme für Linux besprochen.

Anlegen von Sockets

Ein Socket ist eine interne Datenstruktur des Betriebssystems zur Abarbeitung der Kommunikation eines Prozesses, definiert in der Bibliothek `sys/socket.h`:

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

- Erzeugt einen Socket-Deskriptor und liefert einen Handle für zukünftige Kommunikationsoperationen.
- `domain` definiert globale Einstellungen der Communication Domain. Wir nutzen die Konstante `AF_INET` für die Internet-Adressfamilie.
- `type` legt die Art der Verbindung fest:
  - `SOCK_STREAM` (TCP: Transmission Control Protocol)
  - `SOCK_DGRAM` (UDP: User Datagram Protocol)
  - `SOCK_RAW` (Roh-Daten, Umgehen von Netzwerk-Protokollen)

  Verschiedene Kommunikationstypen unterscheiden sich insbesondere in der Zuverlässigkeit der Kommunikation.
- Sind in einer Domäne mehrere Protokolle erlaubt, kann mittels `protocol` eins ausgewählt werden. Normalerweise 0.

Socket-Adressen

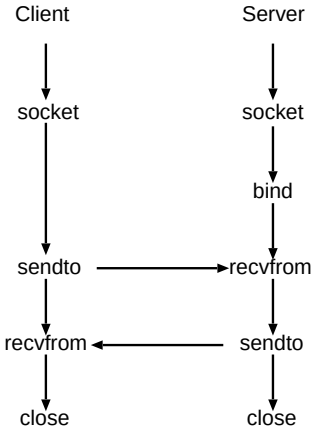
Sockets definieren einen Zugangspunkt für Netzwerkverbindungen.

Die Socket-Adressen werden in einem C++/C-Programm mittels einer allgemeinen Struktur `sockaddr` dargestellt.

Darauf aufbauend werden spezielle Strukturen definiert, wovon für uns nur die Struktur `sockaddr_in` für Internet-Sockets wichtig ist:

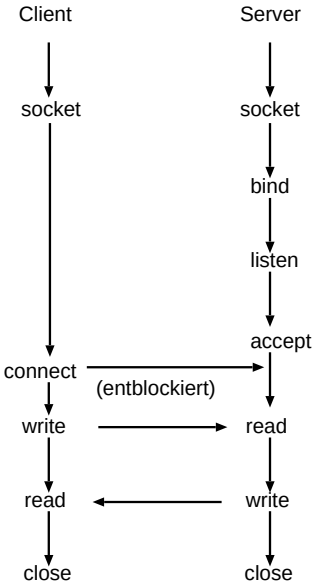
```
struct sockaddr_in {
    short sin_family;           // Domain AF_INET
    unsigned short sin_port;    // Port-Nummer
    struct in_addr sin_addr;    // IP(v4)-Adresse
    unsigned char __pad[8];     // dummy
}
```

Verbindungslose Socket-Kommunikation



- Der Typ ist `SOCK_DGRAM`.
- `bind` bindet einen Socket an eine Adresse.
- Abschicken bzw. empfangen von Daten mittels `sendto` bzw. `recvfrom`.
- `close` schließt den Socket.
- Wird auf Grund der Unzuverlässigkeit nicht in Client-Server-Systemen verwendet.

Verbindungsorientierte Socket-Kommunikation



- Der Typ ist `SOCK_STREAM`.
- `bind` bindet einen Socket an eine Adresse.
- `listen` bereitet den Server-Socket für ankommende Anfragen vor und legt die Größe der Warteschlange fest.
- `accept` blockiert den Server-Prozess und wartet auf ankommende Nachrichten.
- Der Client baut eine permanente Verbindung mittels `connect` auf.
- Abschicken bzw. empfangen von Daten mittels `write` bzw. `read`

Nachrichten dürfen nicht beliebig groß sein. Der Aufruf `man -S7 socket` der Manual-Seiten unter Linux liefert:

The core socket networking parameters can be accessed via files in the directory `/proc/sys/net/core/`.

- `rmem_default` contains the default setting in bytes of the socket receive buffer.
- `rmem_max` contains the maximum socket receive buffer size in bytes which a user may set by using the `SO_RCVBUF` socket option.
- `wmem_default` contains the default setting in bytes of the socket send buffer.
- `wmem_max` contains the maximum socket send buffer size in bytes which a user may set by using the `SO_SNDBUF` socket option.

Mit dem folgenden C-Programm werden die Größe des Receive- und des Send-Buffers gelesen und angezeigt.

Datenaustausch über Sockets

```c
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <sys/socket.h>

int main(void) {
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock == -1) {
        printf("cannot open socket: %s\n", strerror(errno));
        return 1;
    }

    int rcv = 0, snd = 0;
    socklen_t len = sizeof(int);
    getsockopt(sock, SOL_SOCKET, SO_RCVBUF, &rcv, &len);
    getsockopt(sock, SOL_SOCKET, SO_SNDBUF, &snd, &len);
    printf("SO_RCVBUF: %d, SO_SNDBUF: %d\n", rcv, snd);
    return 0;
}
```

```
int send(int sfd, const void *msg, size_t len, int flags)
```

- Schickt Daten der Länge `len` Bytes ab der Adresse `msg` über den Socket `sfd`.
- Rückgabe: Anzahl geschriebener Bytes.
- Die `send`-Methode muss ggf. die Nachricht zerstückeln und mittels mehrerer `send`-Aufrufe verschicken. Dazu ist es notwendig, den Rückgabewert von `send` zu überprüfen. Der eventuelle Rest, der nicht gesendet werden konnte, muss durch weitere `send`-Aufrufe versendet werden.
- Die Funktion `write` entspricht `send`, aber ohne den Parameter `flags`.

```
int recv(int sfd, void *buf, size_t len, int flags)
```

- Empfängt maximal `len` viele Bytes an Daten über den Socket `sfd` und schreibt sie nach `buf`.
- Es liefert die Anzahl empfangener Bytes. Um das Ende einer Nachricht zu erkennen, wird oft die Anzahl der Bytes in einem Header verschickt.
- Der `recv`-Aufruf blockiert, wenn keine Daten vorliegen. Aber mittels der Funktion `fcntl` (steht für File Control) kann ein abweichendes Verhalten definiert werden.
- Liegen Daten von mehreren `send`-Aufrufen vor, können diese ggf. mit einem einzigen `recv`-Aufruf gelesen werden.
- Wird der Flag auf `MSG_PEEK` gesetzt, werden die Daten gelesen ohne diese aus dem Socket zu entfernen.
- Die Funktion `read` entspricht `recv`, aber ohne den Parameter `flags`.

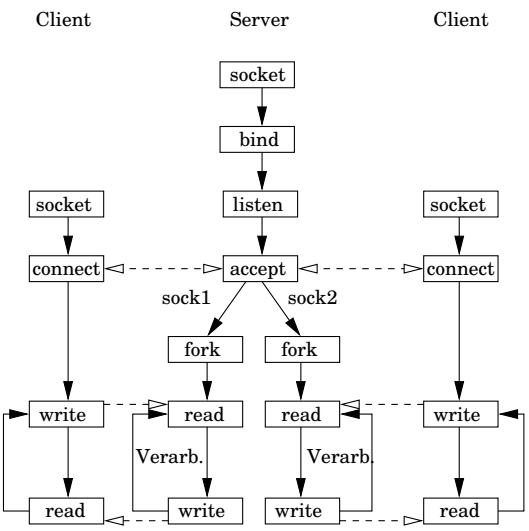
```
int bind(int sockfd, struct sockaddr *addr, int len)
```

- Bindet den Socket `fd` an die lokale Adresse `addr`. Die Größe der Struktur wird in `len` angegeben.
- Mittels `bind` informieren wir das System über unseren Server, sodass alle an dem angegebenen Port eingehenden Nachrichten an das Programm weitergereicht werden sollen.

Auszug aus der Linux Manual-Page:

> When a socket is created with socket(2), it exists in a name space (address family) but has no address assigned to it. bind() assigns the address specified by addr to the socket referred to by the file descriptor sockfd. [...] Traditionally, this operation is called „assigning a name to a socket“. [...] When INADDR_ANY is specified in the bind call, the socket will be bound to all local interfaces.

Server-Sockets



- Ein Client nimmt über den Socket des Servers Verbindung zum Server auf.
- Damit der Server sofort für Anfragen weiterer Clients bereit ist, wird die Abarbeitung der Anfrage in einem Thread oder einem mittels `fork` erzeugtem Prozess durchgeführt.
- Damit die Threads oder Prozesse nicht alle über denselben Socket kommunizieren, liefert `accept` einen neuen Socket, über den die weitere Kommunikation mit einem Client erfolgt.

Services

Jeder Dienst/Service läuft auf einem bestimmten Port. In der Datei `/etc/services` sind die Ports aufgelistet. Auszug:

ftp	21/tcp	pop3	110/tcp
ssh	22/tcp	sunrpc	111/tcp
telnet	23/tcp	ntp	123/tcp
smtp	25/tcp	imap3	220/tcp
http	80/tcp	talk	517/udp

Innerhalb eines Dienstes gibt es oft verschiedene Aktionen bzw. Methoden, die ausgeführt werden können.

Bei HTTP sind dies unter anderem `GET`, `HEAD`, `POST`, `PUT`, `DELETE` und `TRACE`.

Jede Programmiersprache stellt eine API zur Verfügung, um auf diese Daten zugreifen zu können. Auszug der Linux Manual Page:

- The `getservbyname(const char *name, const char *proto)` function returns a `servent` structure for the entry from the database that matches the service `name` using protocol `proto`.

  If `proto` is NULL, any protocol will be matched.
- The `getservbyport(int port, const char *proto)` function returns a `servent` structure for the entry from the database that matches the port `port` (given in network byte order) using protocol `proto`.

  If `proto` is NULL, any protocol will be matched.
- The `getservent(void)` function reads the next entry from the services database and returns a `servent` structure containing the broken-out fields from the entry.

The servent structure is defined in <netdb.h> as follows:

```
struct servent {
    char  *s_name;      // official service name
    char **s_aliases;   // alias list
    int    s_port;      // port number
    char  *s_proto;     // protocol to use
}
```

The members of the servent structure are:

- s_name The official name of the service.
- s_aliases A NULL-terminated list of alternative names for the service.
- s_port The port number for the service given in network byte order.
- s_proto The name of the protocol to use with this service.

```c
#include <stdio.h>
#include <netdb.h>

int main(void) {
    struct servent *svc = getservbyname("ftp", "tcp");
    printf("ftp/tcp: %d\n", ntohs(svc->s_port));

    svc = getservbyport(htons(517), "udp");
    printf("517/udp: %s\n", svc->s_name);

    for (int i = 0; i < 20; i++) {
        svc = getservent();
        printf("%3d: %10s %s\n", ntohs(svc->s_port),
               svc->s_name, svc->s_proto);
    }
    return 0;
}
```

Resolver: Einfach aufgebaute Software-Module des DNS (domain name service), die Informationen vom Nameserver abrufen können. Sie bilden die Schnittstelle zwischen Anwendung und Nameserver[1].

```
struct hostent * gethostbyname(const char *name)
```

- The function returns a structure of type `hostent` for the given host `name`.
- Here `name` is either a hostname or an IPv4 address in standard dot notation.

```
struct hostent {
    char  *h_name;              // official name of host
    char **h_aliases;           // alias list
    int    h_addrtype;          // host address type
    int    h_length;            // length of address
    char **h_addr_list;         // list of addresses
}
```

[1] https://de.wikipedia.org/wiki/Domain_Name_System#Resolver

Resolver

```c
#include <stdio.h>        // printf()
#include <netdb.h>        // gethostbyname()
#include <arpa/inet.h>    // inet_ntoa()

int main(void) {
    struct hostent *host;
    char **alias;

    host = gethostbyname("www.hsnr.de");

    printf("Official name: %s\n", host->h_name);
    for (alias = host->h_aliases; *alias != 0; alias++) {
        printf("alternative name: %s\n", *alias);
    }
```

```
    if (host->h_addrtype == AF_INET) {
        int i = 0;
        while (host->h_addr_list[i] != 0) {
            struct in_addr addr;

            addr.s_addr = *(u_long *) host->h_addr_list[i++];
            printf("IPv4 Address: %s\n", inet_ntoa(addr));
        }
    }

    return 0;
}
```

Die Adressen werden in der Struktur `in_addr` in einer binären Form abgelegt. Die Funktion `inet_ntoa` wandelt die binäre Form in die bekannte Numbers-And-Dots-Notation wie bei `192.168.3.1` um.

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

- Converts the Internet host address `cp` from the IPv4 numbers-and-dots notation into binary data and stores it in the structure that `inp` points to.
- Returns nonzero if the address is valid, zero if not.

`char * inet_ntoa(struct in_addr in)`

- Converts the Internet host address `in`, given in network byte order, to a string in IPv4 numbers-and-dots notation.
- The string is returned in a statically allocated buffer, which subsequent calls will overwrite.

`in_addr_t inet_addr(const char *cp)`

- Converts the Internet host address `cp` from IPv4 numbers-and-dots notation into binary data in network byte order.
- If the input is invalid, `INADDR_NONE` is returned. This is an obsolete interface to `inet_aton`.

Network Byte Order

Zahlen werden auf den Rechnern in binärer Form gespeichert, leider nicht einheitlich:

- Little Endian (least significant byte first): Einsatz bei dem legendären Prozessor 6502 des C64 Home-Computers, der NEC-V800-Reihe oder den Intel-x86-Prozessoren.
- Big-Endian (most significant byte first): Einsatz bei der Motorola-68000-Familie, den Prozessoren von IBM der System-z-Reihe, den SPARC-CPUs von SUN und dem PowerPC.

Computer verschiedener Plattformen können nur dann fehlerfrei kommunizieren, wenn bei den Netzwerkprotokollen die Byte-Reihenfolge festgeschrieben ist. Diese *Network Byte Order* ist bei TCP-IP festgelegt auf das Big-Endian-Format.

Die Byte-Reihenfolge des Systems wird als *Host Byte Order* bezeichnet. Da die x86-Prozessoren von Intel mit dem Little-Endian-Format arbeiten, muss bei den Rechnern in unseren Arbeitsräumen diese im Anwendungsprogramm ggf. umgewandelt werden.

Network Byte Order

Die Bibliothek `arpa/inet.h` definiert dazu einige Funktionen:

- `uint32_t htonl(uint32_t hlong)` converts the unsigned integer `hlong` from host to network byte order.
- `uint16_t htons(uint16_t hshort)` converts the unsigned short integer `hshort` from host to network byte order.
- `uint32_t ntohl(uint32_t nlong)` converts the unsigned integer `nlong` from network to host byte order.
- `uint16_t ntohs(uint16_t nshort)` converts the unsigned short integer `nshort` from network to host byte order.

Auf dem Client:

`int connect(int fd, struct sockaddr *adr, int l)` öffnet eine Verbindung vom Socket `fd` zu einem passenden Socket auf dem Server mit der Adresse `adr`. Da unterschiedliche Typen von Strukturen bei `adr` angegeben werden können, muss die Größe der Struktur in `l` übergeben werden.

Auf dem Server:

`int listen(int sfd, int backlog)` definiert für den Socket `sfd` die Länge der Warteschlange für eingehende Verbindungen.

`int accept(int s, struct sockaddr *addr, int len)` akzeptiert Verbindung über Server-Socket `s`, erzeugt passenden Accept-Socket und gibt dessen Deskriptor zurück. `len` muss Länge der Adressstruktur enthalten. Füllt `addr` mit Informationen über anfragenden Client.

Das folgende Programm protokolliert alle Anfragen, die auf Port 8080 (HTTP) eintreffen und schickt als Antwort `I am alive` an den Client. Zum Testen des Programms im Browser die URL `http://localhost:8080` aufrufen.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>

#define LEN     1024
#define PORT    8080

int _cnt = 0;
int _aSocket;
```

HTTP-Listener

```
void stopServer(int sig) {
    printf("shutdown server ...\n");
    close(_aSocket);
    exit(0);
}

int main(int argc, char *argv[]) {
    // Signal-Handler für CTRL-C registrieren
    signal(SIGINT, stopServer);

    // create a server socket
    _aSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (_aSocket == -1) {
        perror("cannot open socket ");
        exit(1);
    }
```

HTTP-Listener

```
// bind server port
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY;
addr.sin_port = htons(PORT);

int r = bind(_aSocket, (struct sockaddr *)&addr,
             sizeof(addr));

if (r == -1) {
    perror("cannot bind socket ");
    exit(2);
}
```

HTTP-Listener

```
// listen for incoming requests
listen(_aSocket, 3);
int addrlen = sizeof(struct sockaddr_in);

while (1) {          // server loops forever

    // waiting for incoming requests
    int conn = accept(_aSocket, (struct sockaddr *)&addr,
            &addrlen);

    if (conn == 0)
        continue;

    _cnt += 1;
    printf("client %s is connected...\n",
           inet_ntoa(addr.sin_addr));
```

```
        char buf[LEN];                          // read the request
        do {
            int len = recv(conn, buf, LEN, 0);
            buf[len] = '\0';
            printf("%d: %s\n", len, buf);
        } while (!strstr(buf, "\r\n\r\n") &&
                 !strstr(buf, "\n\n"));
        // Header ist vom Body durch eine Leerzeile getrennt

        if (strstr(buf, "GET / HTTP"))     // handle the request
            sprintf(buf, "HTTP/1.1 200 OK\r\n\r\n"
                         "I am alive: %d", _cnt);
        else sprintf(buf, "HTTP/1.1 404 Not Found\r\n");

        send(conn, buf, strlen(buf), 0);
        close(conn);
    }
    return 0;
}
```

HTTP ist die gemeinsame Sprache von Web-Browser und -Server:

- `GET` weist den Web-Server an, den Inhalt der spezifizierten Datei an den anfordernden Web-Browser zu schicken. Als Parameter wird zum einen die gewünschte Datei, zum anderen das verwendete Protokoll angegeben.
- `HEAD` ist analog zu dem Befehl GET. Der Web-Server liefert aber nicht den Inhalt der Datei aus, sondern schickt nur den Header zurück.
- Mittels `POST` kann der Browser dem Server Informationen übermitteln, die bspw. mittels eines HTML-Formulars vom Benutzer eingegeben wurden.
- `PUT` legt Dateien auf dem Web-Server ab.
- `DELETE` löscht Dateien vom Server.
- `TRACE` liefert genaue Anfrage des Clients an den Web-Server zurück an den Client.

Der Antwort des Web-Servers geht immer ein Header voraus, wo sich Angaben wie

- der Typ des Inhalts (mime type: text/html, text/plain, image/gif, usw.),
- die Größe der Datei, Status-Codes und ähnliches finden lassen.

Der Header ist vom Body durch eine Leerzeile getrennt.

Lassen wir auf dem Client einen Packet-Sniffer wie `ngrep -d lo` auf dem loopback-Device mitlaufen, so können wir uns die gesamte Anfrage des Web-Browsers ansehen:

```
GET / HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:72.0)
    Gecko/20100101 Firefox/72.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,
    image/webp,*/*;q=0.8
Accept-Language: de,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
DNT: 1
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

Auf die gleiche Art erhalten wir die Antwort des Web-Servers:

```
HTTP/1.1 200 OK.
Date: Sat, 08 Feb 2020 18:28:57 GMT.
Server: Apache/2.2.22.
X-Powered-By: PHP/5.4.38-0+deb7u1.
Vary: Accept-Encoding.
Content-Encoding: gzip.
Content-Length: 123                <--------- !!!!!
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html
                                   <--------- !!!!!
<html>
<head>
.....
```

Entwurfsmuster Wrapper Facade

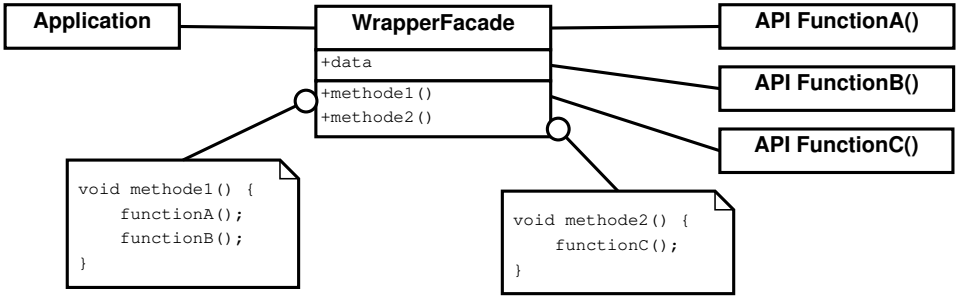
Zweck: Encapsulating Low-level Operating System APIs

Motivation:

- Software, die direkt Funktionen des Betriebssystems aufruft, ist nicht plattformunabhängig und nur schwer portierbar.
- Wir führen daher eine Zwischenschicht ein, die der Applikation eine definierte Schnittstelle bereitstellt.
- Die Implementierung der Schnittstelle kann dann für jedes Betriebssystem, unabhängig von der Anwendung, erstellt werden.

Entwurfsmuster aus dem Bereich Service Access & Configuration

Entwurfsmuster Wrapper Facade



Teilnehmer:

- API Funktionen stellen in der Regel einen Dienst mittels einer wohldefinierten Schnittstelle zur Verfügung, wie bspw. die Socket-API in C.
- Die WrapperFacade ist eine Menge von Klassen, die bestehende Funktionen und Daten kapselt, um einen direkten Aufruf von Betriebssystem-APIs oder Bibliotheken zu vermeiden.

Socket-Wrapper in C++

socket.hpp

```
#include <string>
#include <cerrno>
#include <cstring>
#include <netinet/in.h>

class Socket {
private:
    sockaddr_in _address;
    int _sfd;

public:
    Socket(std::string ip, int port);
    Socket(int socket);

    void send(std::string msg);
    std::string recv(void);
    void close(void);
};
```

socket.cpp

```
#include ....

Socket::Socket(string ip, int port) {
    _sfd = ::socket(AF_INET, SOCK_STREAM, 0);
    if (_sfd == -1)
        throw SocketException(strerror(errno));

    _address.sin_family = AF_INET;
    _address.sin_addr.s_addr = ::inet_addr(ip.c_str());
    _address.sin_port = ::htons(port);
    int len = sizeof(_address);

    int flag = ::connect(_sfd,
                         (struct sockaddr *) &_address, len);
    if (flag == -1)
        throw SocketException(strerror(errno));
}
```

```
Socket::Socket(int socket) : _sfd(socket) {
}

void Socket::send(string msg) {
    // siehe Übung ...
}

string Socket::recv() {
    // siehe Übung ...
}

void Socket::close(void) {
    ::close(_sfd);
}
```

Achtung: Die Anzahl der `write`-Operationen muss nicht identisch sein zur Anzahl `read`-Operationen, um die Daten zu empfangen. Die Daten mehrerer `write`-Aufrufe werden evtl. mit einer einzigen `read`-Anweisung gelesen. Jeder `recv`-Aufruf soll aber genau eine Nachricht liefern.

servSocket.hpp

```
#include <string>
#include <cerrno>
#include <cstring>
#include <netinet/in.h>
#include "socket.hpp"

class ServerSocket {
private:
    sockaddr_in _address;
    int _sfd;
    socklen_t _addrlen;
public:
    ServerSocket(int port, int queueSize);
    ~ServerSocket(void);

    Socket accept(void);
    void send(std::string msg);
    std::string recv(void);
};
```

servSocket.cpp

```
#include ....

ServerSocket::ServerSocket(int port, int qSize){
    // create a server-socket
    _sfd = ::socket(AF_INET, SOCK_STREAM, 0);
    if (_sfd == -1)
        throw SocketException(strerror(errno));

    // Fehler "cannot bind socket: Address
    // already in use" abfangen
    int i = 1;
    ::setsockopt(_sfd, SOL_SOCKET, SO_REUSEADDR,
                 &i, sizeof(i));

    // bind server port
    _address.sin_family = AF_INET;
    _address.sin_addr.s_addr = INADDR_ANY;
    _address.sin_port = ::htons(port);
```

Server-Socket-Wrapper in C++

```
    int r = ::bind(_sfd,
                   (struct sockaddr *) &_address,
                   sizeof(_address));
    if (r == -1)
        throw SocketException(strerror(errno));

    // listen for incoming requests
    ::listen(_sfd, qSize);
    _addrlen = sizeof(struct sockaddr_in);
}

ServerSocket::~ServerSocket() {
    ::close(_sfd);
}
```

Server-Socket-Wrapper in C++

```
Socket ServerSocket::accept(void) {
    // waiting for incoming requests
    cout << "waiting for incoming requests ..." << endl;
    int conn = ::accept(_sfd,
                        (struct sockaddr *) &_address,
                        &_addrlen);

    if (conn == -1)
        throw SocketException(strerror(errno));

    cout << "----> accept socket: " << conn << endl;
    return Socket(conn);
}
```

ECHO-Server in C++

Der ECHO-Server schickt die erhaltene Anfrage einfach zurück an den Client. Eine solche Funktionalität ist oft hilfreich zur Fehlersuche.

server.cpp

```
#include "socket.hpp"
#include "servSocket.hpp"
using namespace std;

int main(int argc, char **argv) {
    ServerSocket server(6200, 10);

    while (1) {
        Socket acceptSocket = server.accept();
        string req = acceptSocket.recv();
        acceptSocket.send("ECHO REPLY: " + req);
        acceptSocket.close();
    }
}
```

client.cpp

```
#include "socket.hpp"
#include <iostream>
#include <string>
using namespace std;

string host = "127.0.0.1";
int port = 6200;

void work(string req) {
    try {
        Socket sock(host, port);

        sock.send(req);
        string msg = sock.recv();
        cout << "reveived from server: " << msg << endl;
    } catch (SocketException e) {
        cout << e.getError() << endl;
    }
}
```

Client in C++

```
int main(int argc, char **argv) {
    string req[7] = {"ECHO:Hallo, Welt!",
                     "DATE:now",
                     "TIME:now",
                     "RANDOM:20000",
                     "PRIME:1234567899",
                     "GGT:12345678, 23456789",
                     "blubb"};

    for (int i = 0; i < 20; i++)
        work(req[i%7]);
}
```