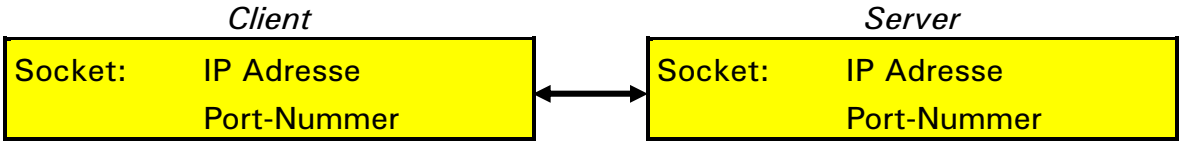
TCP/IP-Client-Server-Programme mit Multitasking

Als Beispiel für Programme, die nur bei Aufteilung auf mehrere Threads sinnvoll strukturiert werden können, soll ein einfaches Client-Server-Programm betrachtet werden, bei dem ein Chat über eine Internet-TCP/IP-Verbindung realisiert wird.

1.1 Grundprinzipien der TCP/IP-Kommunikation unter Windows

TCP/IP-Verbindungen arbeiten immer nach dem Client-Server-Schema. Der Client (aktive Rolle) verbindet sich mit einem Server (passive Rolle) und richtet Anfragen an den Server, die von diesem beantwortet werden.



Verbindungen werden auf der Seite des Clients und auf der Seite des Servers jeweils durch eine IP-Adresse und eine Port-Nummer eindeutig identifiziert. Die **IP-Adresse** ist eine weltweit eindeutige 32bit-Zahl, die die Netzwerkadresse des jeweiligen Rechners definiert. Um die Zahl für den Menschen lesbarer zu gestalten, wird die 32bit Zahl in der Regel als Gruppe von 4 Dezimalzahl mit dazwischenliegenden Punkten angeschrieben, z.B. statt 86 6C 22 03h = 134.108.034.003 (IP-Adresse als „Dotted Decimal"). Da auch diese Dezimalzahlen noch schwer zu merken sind, haben Rechner auch noch einen „Klartextnamen", z.B. ‚www.fht-esslingen.de'. Durch den „Domain Name Service", d.h. das Abfragen einer verteilten Namensdatenbank im Internet, kann die Zuordnung zwischen einer IP-Adresse und einem Klartextnamen oder umgekehrt hergestellt werden. Der Domain Name Service kann unter Windows NT/XP mit dem Programm nslookup abgefragt werden,

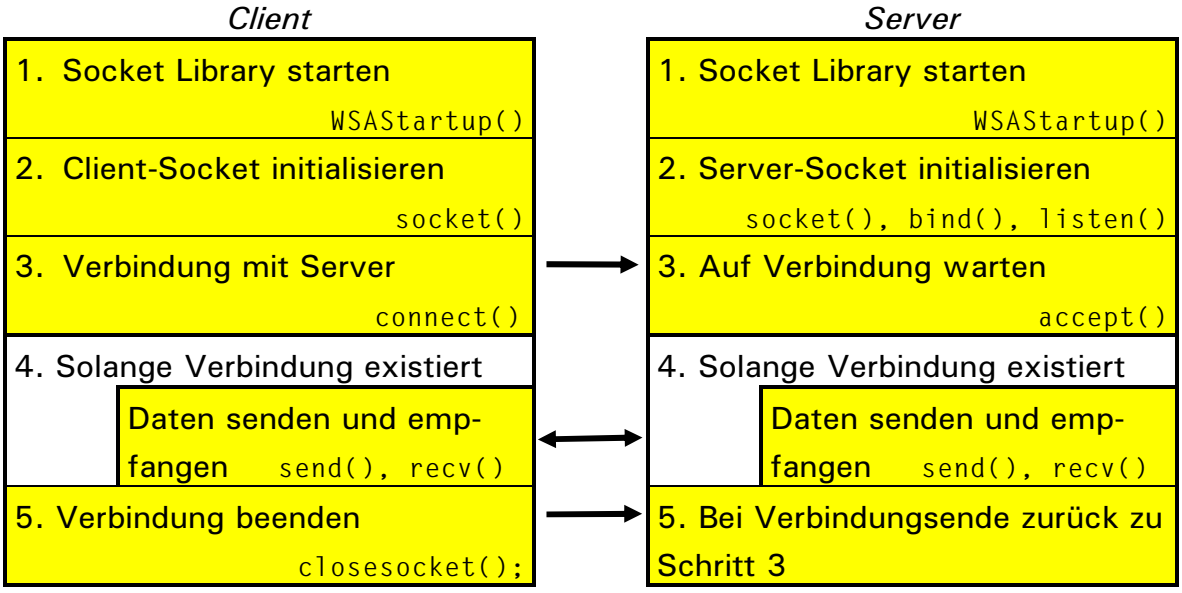
z.B. nslookup www.fht-esslingen.de → liefert 134.108.34.3
nslookup 134.108.34.3 → liefert www.fht-esslingen.de

Rechner, die gar nicht mit dem Internet verbunden sind, können TCP/IP-Verbindungen auch lokal, z.B. zwischen mehreren Programmen auf demselben Rechner nutzen. Der Default-Rechnername in derartigen Fällen ist „localhost", die zugehörige IP-Adresse „127.0.0.1". (Der DNS funktioniert auf einem lokalen PC i.a. nicht).

Während Clients eine beliebige Port-Nummer verwenden können, die in der Regel bei der Aufnahme der Verbindung einfach vom Betriebssystem zugewiesen wird, müssen Server definierte, dem Client bekannte Port-Nummern verwenden, damit eine Verbindung überhaupt möglich ist. HTTP-Webserver z.B. verwenden die Port-Nummer 80, FTP-Server 21 usw. („Well-known Port").

Um die Programmierung von TCP/IP-Verbindungen zu vereinfachen, wurden die Funktionen der sogenannte **Socket-API** definiert (Windows Socket Library).

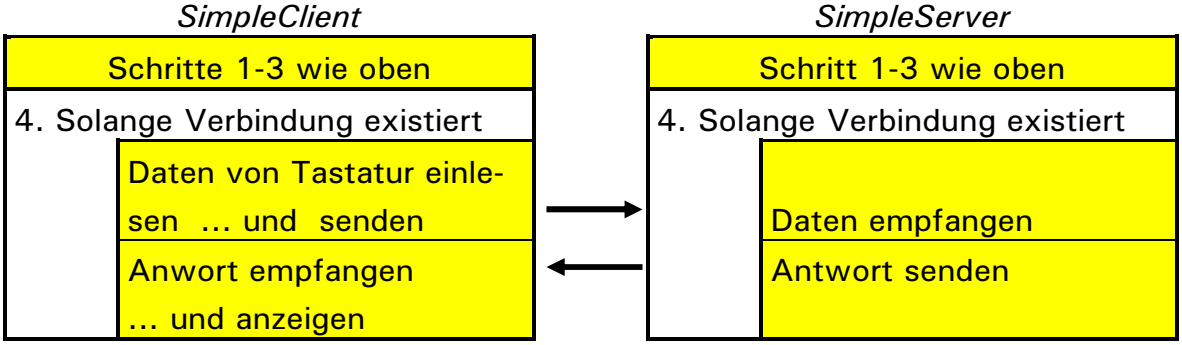
Der prinzipielle Aufbau eines Client- und eines Server-Programms unter Windows sieht folgendermaßen aus:



Zunächst muss mit der Funktion `WSAStartup()` die Windows-Socket-Library gestartet werden und mit `socket()` jeweils ein Socket initialisiert werden (auf der Serverseite zusätzlich mit `bind()` und `listen()`). Der Server wartet danach mit `accept()`, bis sich ein Client mit `connect()` verbindet. Solange die Verbindung existiert, können beide Seiten mit `send()` und `recv()` Daten senden und empfangen. Der Client beendet die Verbindung mit `closesocket()`, worauf der Server mit `accept()` auf eine neue Verbindung wartet.

1.2 Ein einfaches Beispiel: SimpleClient und SimpleServer

Auf den folgenden Seiten wird dieses Zusammenspiel an zwei einfachen Beispielprogrammen gezeigt. Dazu muss zusätzlich der Ablauf beim Senden und Empfangen definiert werden, solange die Verbindung existiert. Hierzu wird zunächst ein einfaches „Ping-Pong-Protokoll“ definiert. Das Client-Programm liest von der Tastatur und sendet die eingegebenen Zeichen an der Server, der Server wandelt sie in Grossbuchstaben und sendet sie zurück zum Client. Der Client zeigt die Antwort an und wartet auf die nächste Tastatureingabe:



Beispielprogramm Einfacher TCP/IP-Chatting-Client ***SimpleClient.cpp***

```
#include <windows.h>
#include <stdio.h>
#include <process.h>

typedef struct {SOCKET sock; SOCKADDR_IN addr; } SOCKDATA;

SOCKDATA sd;                                    //Socket-Datenstruktur

void OpenConnection(char *ServerIp, char *ServerPort)
{   char buffer[4096];                          //Empfangspuffer
    int i;

    //***** Windows Socket Library laden *****                          2

    WSAData wsaData;
    WSAStartup(0x0101, &wsaData);

    //***** Den Client-Socket initialisieren *****                       3

    if ((sd.sock=socket(AF_INET, SOCK_STREAM, 0)) == INVALID_SOCKET)
      return;                               //Socket erzeugen
    setsockopt(sd.sock, SOL_SOCKET, SO_REUSEADDR, "1", sizeof(char));

    //***** Verbindung mit Server herstellen *****                       4

    sd.addr.sin_family= AF_INET;            //Adressfamilie Internet (TCP/IP)
    sd.addr.sin_port = htons(atoi(ServerPort)); //Portadresse des Servers
    if ((sd.addr.sin_addr.S_un.S_addr = inet_addr(ServerIp)) == INADDR_NONE)
      return;                               //IP-Adresse des Servers
    if ((connect(sd.sock, (SOCKADDR*) &sd.addr,
                 sizeof(SOCKADDR)))==SOCKET_ERROR)       return;

    do //***** Schleife, solange Client mit Server verbunden ist **** 5
    { gets(buffer);                         //Zeichen von der Tastatur einlesen
      i=send(sd.sock, buffer, strlen(buffer), NULL);//... an Server senden
      if ((i==SOCKET_ERROR) || (i==0))      //Abbruch bei Fehler oder wenn der
          break;                            //Server die Verbindung beendet hat
      if (!strcmp(buffer,""))               //Abbruch, wenn eine leere Zeile
          break;                            //eingegeben wird

      i=recv(sd.sock, buffer, sizeof(buffer), 0);   //Antwort des Servers
      if ((i==SOCKET_ERROR) || (i==0))      //Abbruch bei Fehler oder wenn der
           break;                           //Server die Verbindung beendet hat
      buffer[i]=0;                          //Ende des Zeichenstrings markieren
      printf("Empfangen: %s\n", buffer); //Antwort des Servers anzeigen
    } while (TRUE);

    printf("Verbindung zum Server wird geschlossen\n");                  6

    closesocket(sd.sock);                   //Socket des Client schliessen
}
```



```
void main(int argc, char* argv[])                                    1
{   char IpNummer[64]   = "127.0.0.1";   //Default-Wert fuer die IP-Nummer
    char PortNummer[64] = "1234";        //Default-Wert fuer den IP-Port

    OpenConnection(IpNummer, PortNummer);     //TCP/IP-Client starten
}
```

1 Das Hauptprogramm des Clients geht davon aus, dass der Server auf demselben PC läuft (IP-Adresse 127.0.0.1 = localhost) und dass der Server unter der Port-Nummer 1234 auf eine Client-Verbindung wartet.

2 `WSAStartup()` lädt die Windows Socket-Library.

3 Mit `socket()` wird ein TCP/IP-Socket erzeugt und mit `setsockopt()` wird dafür gesorgt, dass die IP-Adresse und der Port wiederverwendet werden können, wenn der Client beendet wird. Ansonsten gibt Windows den Socket erst nach einigen Sekunden wieder frei, so dass eine anschließende Verbindungsaufnahme scheitern würde, wenn der Client sofort erneut aufgerufen wird.

4 Anschließend werden die IP-Adresse und die Portnummer des Servers, zu dem eine Verbindung hergestellt werden soll, in die Socket-Struktur `sd` eingetragen. Mit der Funktion `inet_addr()` wird die „Dotted Decimal"-IP-Adresse des Servers in einen 32bit-Hexadezimalwert umgewandelt, mit htons() wird die Portnummer in die im Netzwerk übliche Big-Endian-Darstellung konvertiert. Die eigentliche Verbindung zum Server wird dann durch die Funktion `connect()` hergestellt. Falls keine Verbindung zustande kommt, z.B. weil das Server-Programm nicht läuft, liefert `connect()` den Rückgabewert `SOCKET_ERROR`.

5 Das Senden und Empfangen der Daten während der Verbindung erfolgt mit den Funktionen `send()` und `recv()`. Bei beiden Funktionen muss der Socket der Verbindung angegeben werden. Bei `send()` außerdem der Buffer mit den zusendenden Daten sowie die Anzahl der zu sendenden Daten in Byte. Bei `recv()` wird der Buffer angegeben, in den die empfangenen Daten kopiert werden sowie dessen Länge in Byte. Es können beliebige Daten gesendet und empfangen werden, nicht nur ASCII-Zeichen. Beide Funktionen warten, bis Daten erfolgreich gesendet bzw. empfangen wurden und geben die Anzahl der gesendeten bzw. empfangenen Bytes zurück. Falls die Verbindung unterbrochen wurde, wird `i=0` bzw. `i=SOCKET_ERROR` zurückgegeben und die Sende-/Empfangsschleife beendet. Der Anwender kann die Schleife durch Eingabe einer leeren Textzeile verlassen und damit das Ende der Verbindung veranlassen.

6 Mit `closesocket()` beendet der Client die Verbindung zum Server.

Beispielprogramm Einfacher TCP/IP-Chatting-Server ***SimpleServer.cpp***

```
#include <windows.h>
#include <stdio.h>
#include <process.h>

typedef struct {SOCKET sock; SOCKADDR_IN addr; } SOCKDATA;

SOCKDATA sdSrv, sdCli;       //Socket-Datenstrukturen für Server und Client

void StartServer(char *localPort)
{   char buffer[4096];                  //Empfangspuffer
    int  addrlen = sizeof(SOCKADDR_IN), i;
    //***** Windows Socket Library laden *****                          2
    WSAData wsaData;    WSAStartup(0x0101, &wsaData);
    //***** Den Server-Socket initialisieren *****                       3
    if ((sdSrv.sock=socket(AF_INET, SOCK_STREAM, 0)) == INVALID_SOCKET)
      return;                           //Socket erzeugen
    setsockopt(sdSrv.sock, SOL_SOCKET, SO_REUSEADDR, "1", sizeof(char));

    sdSrv.addr.sin_family= AF_INET;     //Adressfamilie Internet(TCP/IP)
    sdSrv.addr.sin_port = htons(atoi(localPort));//Portadresse des Servers
    sdSrv.addr.sin_addr.S_un.S_addr = INADDR_ANY;//Verbindung mit bel.Client
    if (bind(sdSrv.sock, (SOCKADDR *) & sdSrv.addr, sizeof(SOCKADDR_IN)))
      return;                           //IP-Adresse und Port zuordnen
    if (listen(sdSrv.sock, 5))          //Socket als Server-Socket
      return;
    //***** Warten, bis sich ein Client mit dem Server verbindet ****  4
    while ((sdCli.sock=accept(sdSr.sock,(SOCKADDR *)
                              &sdCli.addr,&addrlen))!=INVALID_SOCKET)
    { printf("*** Client %s:%d hat eine Verbindung hergestellt\n",
           inet_ntoa(sdCli.addr.sin_addr), htons(sdCli.addr.sin_port));
      do    //***** Schleife, solange Client mit Server verbunden ist  5
      {   i=recv(sdCli.sock, buffer,sizeof(buffer),0);//Zeichen empfangen
          if ((i==SOCKET_ERROR) || (i==0))//Abbruch bei Fehler oder wenn der
            break;                      //Client die Verbindung beendet
          buffer[i]=0;                  //Ende des Strings mit markieren
          printf("Empfangen: %s\n", buffer);//Empfangene Zeichenanzeigen
          strupr(buffer);               //In Grossbuchstaben   umwandeln
          i=send(sdCli.sock,buffer,strlen(buffer),NULL);//… an zurücksenden
          if (i==SOCKET_ERROR)          //Abbruch bei Fehler, z.B. weil der
                break;                  //Client die Verbindung beendet hat
      } while (TRUE);

      printf("*** Client %s:%d hat die Verbindung beendet\n",
           inet_ntoa(sdCli.addr.sin_addr), htons(sdCli.addr.sin_port));
    }
 }
```

```
BOOL WINAPI closeServer(DWORD type)           //Callback-Funktion     6b
{   printf("*** Server wird beendet ***\n");  //... für STRG (CTRL)-C
    closesocket(sdSrv.sock);                  //Server-Socket schliessen
    return TRUE;
}

void main(int argc, char* argv[])                                       1
{   char PortNummer[64]="1234";               //Default-Wert für den IP-Port
    SetConsoleCtrlHandler(closeServer, TRUE); //Callback-Funktion     6a
    StartServer(PortNummer);                        //TCP/IP-Server starten
}
```

1 Das Hauptprogramm des Servers geht davon aus, dass der Server unter der Port-Nummer 1234 auf eine Client-Verbindung wartet. Die IP-Adresse muss im Server-Programm nicht angegeben werden, da sie durch die Netzwerkkonfiguration des PC-Betriebssystems, auf dem der Server läuft, bereits festgelegt ist.

2 `WSAStartup()` lädt die Windows Socket-Library.

3 Mit `socket()` wird der TCP/IP-Socket `sdSrv.sock` erzeugt, auf dem der Server auf eine Client-Verbindung warten soll. Die Port-Adresse des Servers wird in die Datenstruktur sdSrv.addr eingetragen und durch `INADDR_ANY` festgelegt, dass Clients mit beliebigen IP-Adressen eine Verbindung mit dem Server aufnehmen dürfen. Durch die Funktion `bind()` werden diese Werte dem Socket zugeordnet. Mit der Funktion `listen()` wird der Socket als passiver Socket definiert, d.h. als ein Server-Socket, der auf eine Verbindungsaufnahme durch einen Client wartet. (`bind()` und `listen()` sind nur bei Servern notwendig).

4 Mit der Funktion `accept()` wartet der Server, bis ein Client tatsächlich eine Verbindung zu ihm aufnimmt. Durch die Verbindungsaufnahme wird ein neuer, diesem Client eindeutig zugeordneter Socket `sdCli.sock` definiert. In die Datenstruktur `sdCli.addr` werden die IP-Adresse und Port-Nummer des Clients eingetragen, so dass der Server diesen eindeutig identifizieren kann. Die Funktion `inet_ntoa()` dient bei der Bildschirmausgabe dazu, die hexadezimale IP-Adresse des Clients aus dem 32bit-Wert in `sdCli.addr.sin_addr` in einen besser lesbaren "Dotted Decimal"-String umzuwandeln. Die Port-Nummer des Clients steht in `sdCli.addr.sin_port` und wird mit `htons()` konvertiert.

5 Das Senden und Empfangen der Daten während der Verbindung erfolgt mit den Funktionen `send()` und `recv()`. Bei beiden Funktionen muss der dem Client zugeordnete Socket `sdCli.sock` angegeben werden und nicht der Server-

Socket `sdSrv.sock` ! Wenn der Client die Verbindung beendet, geben die Funktionen `send()` bzw. `recv()` jeweils `i=0` bzw. `i=SOCKET_ERROR` zurück und die Sende-/Empfangsschleife `do {...} while (TRUE)` wird beendet. Anschließend wartet der Server wieder mit `accept()` auf eine neue Client-Verbindung. Durch die Trennung in einen einzigen Server-Socket `sdSrv.sock`, mit dem der Server auf eine Verbindung wartet, und einen jedem Client eindeutig zugeordneten Client-Socket `sdCli.sock` kann ein Server (mit einem entsprechenden Multithreading-Programm) mehrere Clients gleichzeitig bedienen. Diese Möglichkeit ist im vorliegenden einfachen Server-Programm aber noch nicht implementiert.

Unter normalen Bedingungen läuft der Server ständig, solange der TCP/IP-Stack des Betriebssystems fehlerfrei arbeitet. Er kann aber durch Betätigen der Tasten STRG (CTRL)-C über die Tastatur abgebrochen werden. Dazu wird mit `SetConsoleCtrlHandler()` die Callback-Funktion `closeServer()` installiert, die bei Betätigen dieser Tasten aufgerufen wird und den Server-Socket schliesst.

AUFGABE

Übersetzen Sie die beiden Beispielprogramme SimpleServer.cpp bzw. SimpleClient.cpp.

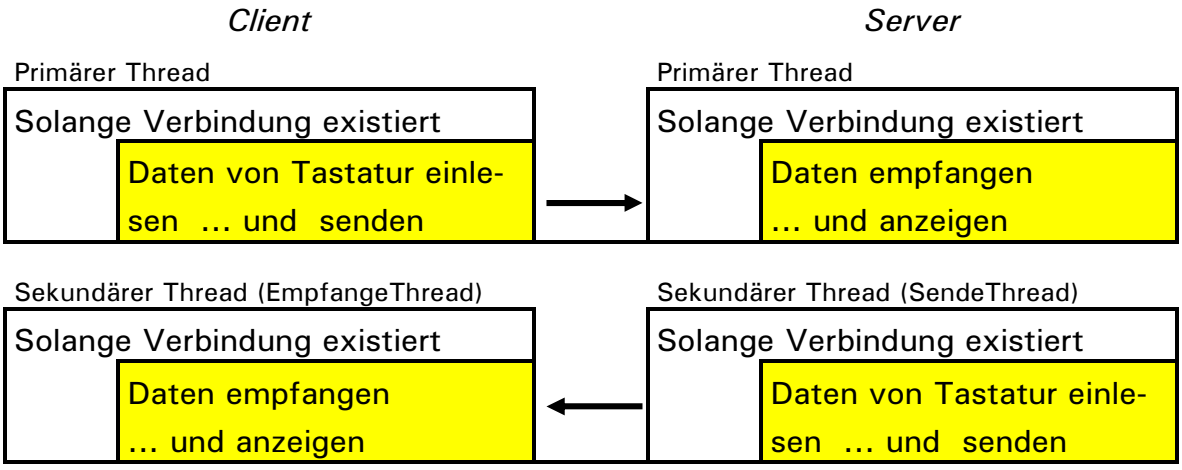
Testen Sie die Programme, indem Sie in einem Konsolenfenster zunächst den Server und dann in einem zweiten Konsolenfenster den Client starten. Im Konsolenfenster des Client können Sie dann Zeichen eingeben, die zum Server gesendet, dort in Grossbuchstaben umgewandelt, zum Client zurückgesendet und dort angezeigt werden. Durch Eingabe einer leeren Zeile (Eingabe-Taste am Zeilenanfang betätigen) können Sie den Client beenden. Wenn Sie den Client anschließend wieder starten, wird erneut eine Verbindung zum Server hergestellt.

Klären Sie mit Hilfe der Visual-C++-Dokumentation zu den Win32-API-Funktionen `socket(), bind(),`usw. sowie mit Hilfe der Unterlagen zur Vorlesung Kommunikationstechnik/Rechnernetze verbleibende Unklarheiten zur Funktion dieser beiden Programmbeispiele.

1.3 Client und Server mit getrennten Sende- und Empfangs-Threads

Weil die Funktionen `gets()` und `recv()` (und auch `send()`) blockieren, d.h. warten, bis Daten über die Tastatur eingegeben bzw. empfangen werden, ist der Ablauf der Kommunikation streng vorgegeben ("Ping Pong"-Protokoll), obwohl TCP/IP-Verbindungen eigentlich Voll-Duplex-Verbindungen sind, bei denen gleichzeitig in beiden Richtungen gesendet werden kann.

Die beiden Programme sollen daher so geändert werden, dass die Tastatureingabe mit Senden der eingegebenen Daten in einem Thread und das Empfangen mit Anzeigen der empfangenen Daten in einem zweiten Thread durchgeführt werden. Damit ist dann ein echtes Chatting möglich, bei dem sowohl beim Client als auch beim Server Texte über die Tastatur eingegeben und an den Partner gesendet und dort angezeigt werden können:



Ändern Sie das Programm **SimpleClient.cpp** folgendermaßen zum Programm **Client.cpp** ab:

- Erzeugen Sie einen sekundären Thread `EmpfangeThread`, wenn der Client eine Verbindung aufgenommen hat, d.h. nach erfolgreichem `connect()`. Der Empfangs-Thread soll als Parameter eine Referenz (C++-Pointer) auf die `SOCKDATA`-Struktur `sd` erhalten und ansonsten nur lokale Variable verwenden!
- Das Empfangen und Anzeigen der Daten

```
i=recv(sd.sock, ...);
. . .
printf("Empfangen: %s\n", buffer);
```

  soll aus dem primären Thread entfernt und in eine Endlosschleife in den Empfangs-Thread verlegt werden. Die Abbruchbedingungen (`i==0` bzw. `i==SOCKET_ERROR`) für die Endlosschleifen im primären und im sekundären Thread sowie die Möglichkeit, den Client durch Eingabe einer Leerzeile zu beenden, bleiben. Wenn der Empfangs-Thread gestartet und wenn er beendet wird, soll er jeweils eine Meldung auf dem Bildschirm ausgeben.

AUFGABE

Schreiben und testen Sie Ihr neues Programm Client.cpp mit dem 'alten' Server SimpleServer.cpp.

*Falls Sie beim Übersetzen Ihrer Programme mit der graphischen Oberfläche von Visual Studio arbeiten, vergessen Sie bitte nicht, die Multithread-Laufzeitbibliothek und die Windows-Socket-Library in den Projekt-Eigenschaften einzurichten (s.h. auch Seite **Fehler! Textmarke nicht definiert**.) :*

- *Falls Sie die **Fehlermeldung "Error C2065**: _beginthread unknown symbol" bzw. "... nicht deklarierter Bezeichner" erhalten, haben Sie diese Einstellung vergessen, da _beginthread() nur in der Multithread-Version der C/C++-Headerdateien definiert ist.*
- *Falls Sie die **Fehlermeldung "ERROR LNK2001**: Unresolved symbol: _socket, _recv, _send, ..." bzw. "... Nicht aufgelöstes Signal ..." erhalten, haben Sie diese Einstellung für die winsock-Bibliothek vergessen, da der Linker die TCP/IP-Funktionen sonst nicht findet.*

Ändern Sie das Programm **SimpleServer.cpp** folgendermaßen zum Programm **Server.cpp** ab:

- Erzeugen Sie einen sekundären Thread `SendeThread`, wenn der Client eine Verbindung aufgenommen hat, d.h. am Anfang der `while(accept(...))`-Schleife. Der Sende-Thread soll als Parameter eine Referenz (C++-Pointer) auf die `SOCKDATA`-Struktur `sdCli`, d.h. auf die dem Client zugeordneten Socket-Daten, erhalten und ansonsten möglichst nur lokale Variable verwenden!
- Entfernen Sie das Umwandeln der empfangenden Zeichen in Grossbuchstaben und das Zurücksenden an den Client aus dem primären Thread.
- Wenn der Client die Verbindung beendet, d.h. wenn die `do { } while(TRUE)`-Schleife im primären Thread verlassen wird, soll `sdCli.socket=NULL` gesetzt werden, bevor in `while(...accept(...))` auf die nächste Verbindung gewartet wird.
- Im Sende-Thread soll in einer `do { } while()`-Schleife abgefragt werden, ob eine Taste gedrückt wurde (Funktion `kbhit()`). Falls keine Taste betätigt wurde, soll bis zum nächsten Schleifendurchlauf 200ms gewartet werden (Funktion `Sleep()`). Falls eine Taste gedrückt wurde, soll ein Textstring von der Tastatur eingelesen und zum Client gesendet werden. In der Bedingung `while(...)` der Schleife soll überprüft werden, ob `sdCli.socket!=NULL` ist. Auf diese Weise kann die Sende-Schleife abgebrochen und der Sende-Thread beendet werden,

wenn der Client die Verbindung beendet und daher im primären Thread `sdCli.socket=NULL` gesetzt wurde. Wenn der Sende-Thread gestartet und wenn er beendet wird, soll er jeweils eine Meldung auf dem Bildschirm ausgeben.

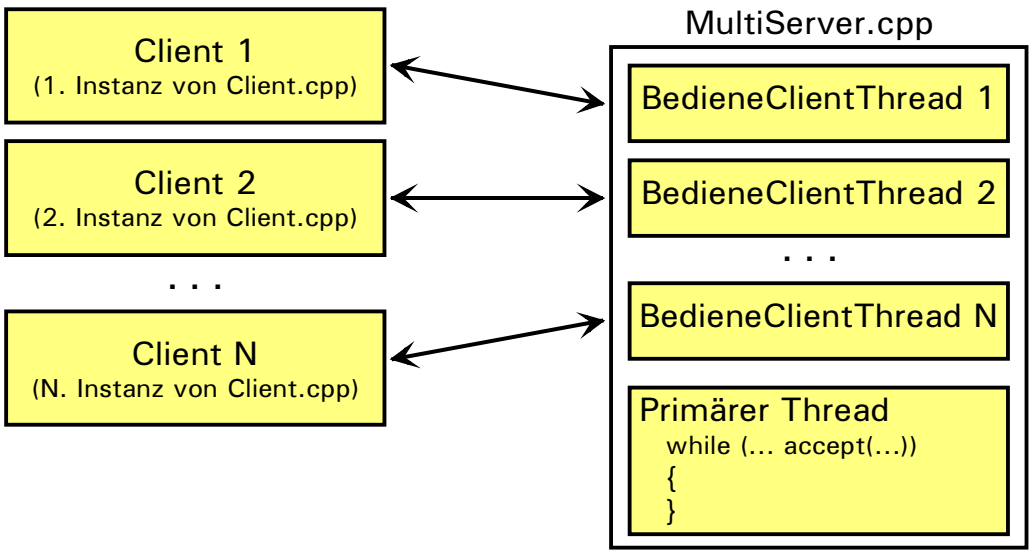
- Die Abbruchbedingungen (`i==0` bzw. `i==SOCKET_ERROR`) für die Sende- bzw. Empfangsschleifen im primären und im sekundären Thread sollen beibehalten werden.

AUFGABE

Schreiben und testen Sie Ihr neues Programm Server.cpp mit dem Client-Programm Client.cpp aus der vorigen Aufgabe.

2. Server-Programm für mehrere Clients

Basierend auf dem Server-Programm SimpleServer.cpp soll ein Chatting-Server entwickelt werden, der mehrere Clients gleichzeitig bedienen kann. Dabei sollen die Clients Textmitteilungen an den Server senden können, die der Server dann an alle anderen Clients weiterverteilt.



Das Programm **SimpleServer**.cpp wird dazu folgendermassen zu **MultiServer.cpp** abgeändert:

a) **Primärer Thread (Funktion** `StartServer()`**)**

- Zur Verwaltung aller Client-Verbindungen wird ein Array `SOCKDATA sdCli[ N ]` als globale Variable angelegt. Durch N, z.B. N = 16, wird die Anzahl der gleichzeitig möglichen Client-Verbindungen festgelegt.
- Alle `i=0...N–1` Elemente `sdCli[i].sock` dieses Arrays werden am Anfang des Programms mit `NULL` initialisiert. Der Wert `NULL` soll anzeigen, dass ein Socket frei ist, d.h. gerade nicht für eine Client-Verbindung verwendet wird.
- Eine lokale Variable `iFree` im primären Thread, die mit `iFree=0` initialisiert wird, zeigt auf die Nummer des nächsten freien Sockets in diesem Array.
- Beim Warten auf eine Verbindung mit `while(... accept(...))` wird `sdCli[iFree].sock` bzw. `sdCli[iFree].addr` verwendet, d.h. es wird der nächste freie Socket belegt.
- Sobald sich ein Client mit dem Server verbindet, wird ein sekundärer Thread `void BedieneClientThread(int k)` erzeugt, der sich um die Bedienung des Clients kümmert. Diesem Thread wird als Parameter `k` der Wert `iFree` übergeben, d.h. der Index des zu diesem Client-gehörenden Eintrags im Array `sdCli[]`.

- Der primäre Thread dagegen sucht im Array `sdCli[] sofort` den nächsten freien Socket, setzt den Index `iFree` auf diesen Socket und wartet anschließend mit `accept()` auf eine weitere Client-Verbindung. Die Suche soll dabei immer am Anfang des Arrays begonnen werden, da die Clients bei Ende einer Verbindung ihren Socket wieder freigeben (siehe unten). Falls kein freier Socket gefunden wird, soll der primäre Thread 1sec warten und dann erneut suchen.

Der primäre Thread ist damit ausschließlich für die Verbindungsaufnahme zu Clients zuständig, das Senden und Empfangen von Daten findet ausschließlich in den sekundären Threads statt. Entsprechend der Anzahl N der verfügbaren Socket-Datenstrukturen können bis zu N sekundäre Threads gleichzeitig aktiv sein.

b) Sekundäre Threads (Funktion `BedieneClientThread(int k)`)

- Innerhalb einer Endlosschleife soll der Thread auf empfangene Daten von "seinem" Client warten. Der eigene Client wird durch den Thread-Parameter `k` identifiziert, der als Index für das Array `sdCli[k].sock` dient.
- Die vom "eigenen" Client empfangenen Daten sollen innerhalb der Endlosschleife an alle anderen gerade mit dem Server verbundenen Clients gesendet werden, d.h. an alle `i=0...N–1` Clients `sdCli[i].sock` mit `i != k` und `sdCli[i] != NULL`. Dabei soll den gesendeten Daten die Zeile: "`*** Client k *** IP : Port ***: \n`" vorangestellt werden, wobei für k, IP (aus `inet_ntoa(sdCli[k].addr.sin_addr)`) und Port (aus `htons(sdCli[k].addr.sin_port)`) die Werte des Clients einzusetzen sind, von dem Daten empfangen wurden.
- Die Endlosschleife soll verlassen werden, wenn die Empfangsfunktion `i==0` bzw. `i==SOCKET_ERROR` zurückgibt, d.h. wenn der Client die Verbindung beendet. Danach soll dann `sdCli[k]=NULL` gesetzt, d.h. der Client-Socket als frei gekennzeichnet und der sekundäre Thread beendet werden.
- Beim Start des sekundären Threads und beim Beenden des sekundären Threads soll der Server an alle zu diesem Zeitpunkt verbundenen Clients eine Meldung "`*** Client k *** IP : Port *** hat eine Verbindung hergestellt`" bzw. "`*** Client k *** IP : Port *** hat die Verbindung beendet`" senden. Für k, IP und Port sind wieder die entsprechenden Werte einzusetzen (siehe oben).
- Alle empfangenen Daten und alle Meldungen sollen auch im Konsolenfenster des Servers angezeigt werden.

AUFGABE

Schreiben und testen Sie das Programm MultiServer zunächst mit mehreren Clients auf Ihrem eigenen PC; anschließend mit Clients, die auf anderen Rechnern laufen.

Um die IP-Adressen eines Rechners herauszufinden, können Sie z.B. folgendermaßen vorgehen:

- *Geben Sie in einem Konsolenfenster den Befehl 'ipconfig' ein, um die IP-Adresse des Rechners zu erfahren.*

Alternativ können Sie bei Rechnern, deren Adresse im Internet Domain Name Verzeichnis registriert ist, ‚nslookup <hostname>' dessen IP-Adresse abfragen. Dabei müssen Sie statt <hostname> den tatsächlichen Netzwerknamen des Rechners eingeben, z.B. ‚nslookup www.fht-esslingen.de'.

Die folgenden Aufgaben sind optional. Wählen Sie gegebenenfalls eine davon aus:

Chatrooms

„Richtige" Chatting-Server bieten verschiedene Chat-Räume an. Der Server soll so modifiziert werden, dass er gleichzeitig drei verschiedenen Chat-Räume A, B und C unterstützt. Die Botschaften sollen nur jeweils an die Teilnehmer desselben Chat-Raums verteilt werden:

- Wenn der Benutzer sich mit dem Server verbindet, soll er zuerst gefragt werden, welchen der drei Chat-Räume A, B oder C er „betreten" will.
- Zur Auswahl muss der Benutzer in seinem Client einen dieser drei Buchstaben als erstes Zeichen seiner Antwort senden. Alle übrigen Zeichen dieser ersten Antwort werden vom Server ignoriert.
- Danach erhält er vom Server eine Bestätigung für seine Auswahl.
- Anschließend kann der Benutzer mit den anderen Teilnehmern seines eigenen Chat-Raums wie gewohnt chatten.
- Der Server versendet empfangene Daten jeweils nur an diejenigen Clients, die im selben Chat-Raume angemeldet sind.

Aufgabe

Ändern Sie das das Server-Programm so ab, dass es die beschriebenen 3 Chat-Räume unterstützt. An den Client-Programmen sollen keine Änderungen durchgeführt werden. Ein möglicher Lösungsansatz besteht darin, dass Sie die Datenstruktur `SOCKDATA sdCli[ ]` *so erweitern, dass in einem weiteren Strukturelement die Information über den vom Client ausgewählten Chat-Raum mitgeführt wird. Diese Information wird mit der o.g. Abfrage nach dem Verbindungsaufbau initialisiert und dient beim Verteilen einer empfangenen Botschaft im Server als weitere Bedingung.*

Verschlüsselung

Bei diesem Chatting-Server werden die Daten im Klartext übertragen, so dass die übertragenen Daten sowohl von den Clients als auch auf dem Server mitgelesen werden können. Das Client-Programm soll jetzt so abgeändert werden, dass die Daten vor dem Versenden verschlüsselt und nach dem Empfangen wieder entschlüsselt werden. Als primitiver Verschlüsselungsalgorithmus (der in der Praxis relativ einfach zu „knacken" wäre) soll eine XOR-Verknüpfung der Sendedaten mit einem Schlüsselwort eingesetzt werden.

Aufgabe

Ändern Sie das Client-Programm so ab, dass beim Programmstart ein 8stelliger ASCII-String (=64bit) als Schlüsselwort über die Tastatur eingegeben werden muss. Die zu sendenden Daten werden vor dem Senden jeweils in Blöcken von 8 Byte mit diesem Schlüssel Exklusiv-ODER (XOR) verknüpft. Auf der Empfängerseite sollen die empfangenen Daten wiederum mit demselben 8 Byte Schlüssel Exklusiv-Oder verknüpft werden. Die Verschlüsselung und Entschlüsselung selbst erfolgt zeichenweise, d.h. das 1. Zeichen der Botschaft mit dem1. Zeichen des Schlüssels, das 2. Zeichen der Botschaft mit dem 2. Zeichen des Schlüssels, ..., das 9. Zeichen der Botschaft wieder mit dem 1. Zeichen des Schlüssels usw.
Beachten Sie dabei bitte folgendes:

- *Die C-Funktionen zur Ermittlung der Stringlänge strlen(), zum Kopieren oder Anhängen von Strings strcpy() bzw. strcat() und die Funktionen zur Bildschirmausgabe von Strings printf(), puts() usw. funktionieren nur mit unverschlüsselten Strings. Durch die Verschlüsselung können normale Textzeichen in 0h umgewandelt bzw. das 0h-Byte am Ende eines üblichen C-Strings in einen Wert ungleich 0h konvertiert werden, so dass die genannten C-Funktionen das Stringende nicht mehr korrekt erkennen. Ermitteln Sie daher die Länge des zu sendenden Textstrings mit strlen() vor der Verschlüsselung und verwenden Sie diesen Wert in der send()-Funktion bzw. den von recv() zurückgegebenen Wert beim Entschlüsseln.*
- *Der Server soll den Schlüssel der Client-Programme nicht kennen. Da er aber nicht nur die Botschaften zwischen den Client-Programmen weiterleitet, sondern auch eigene Zeichen, z.B. Informationen über An- und Abmelden der Clients sowie bei jeder Datenbotschaft Informationen über den Sender einfügt, werden diese Zeichen vom Server nicht verschlüsselt und dürfen im Client nicht entschlüsselt werden. Daher wird vereinbart, dass die Informationen im Server jeweils nur als allererste Zeile einer Botschaft übertragen werden und dass die Client-Daten erst ab der zweiten Zeile folgen. Die Entschlüsselung im Client soll so erfolgen, dass die Entschlüsselung erst ab der zweiten Zeile einer empfangenen Botschaft beginnt, während die erste Zeile direkt angezeigt wird.*

