FACHHOCHSCHULE TECHNIKUM WIEN

Verteilte Systeme

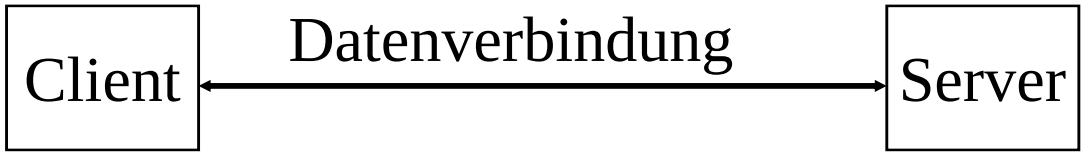
Netzwerkprogrammierung unter Linux

Einführung

Einführung

FACHHOCHSCHULE TECHNIKUM WIEN

- Client-Server Architektur



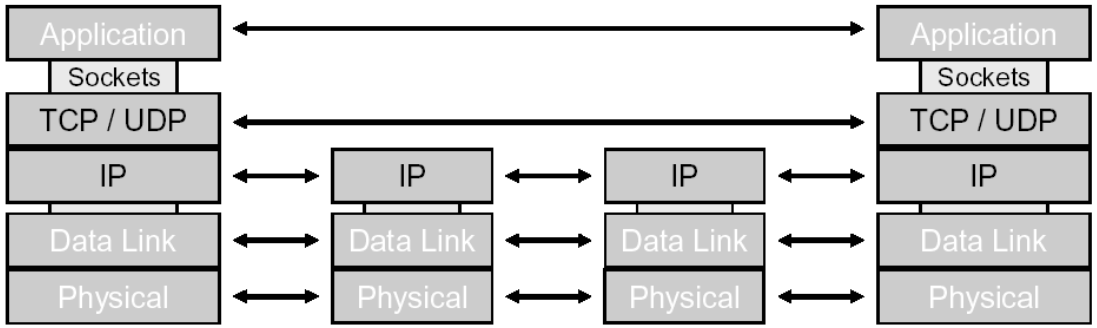
- Kommunikation über TCP/IP Protokoll
  - ✓ verbindungsorientiert: TCP
  - ✓ verbindungslos: UDP
  - ✓ Grundlagen zum TCP/IP Protokoll siehe LV Netzwerke Grundlagen

- Socket Programmierung
  - ✓ Verwendung der Socket API

Sockets

FACHHOCHSCHULE TECHNIKUM WIEN

- Socket
  - ✓ wörtlich Buchse oder Steckdose
  - ✓ Endpunkt einer 2-Wege Kommunikation zwischen 2 Prozessen im Netzwerk
  - ✓ entwickelt unter BSD Unix
    - verfügbar unter Unix/Linux
    - auch Implementierungen für Windows (Winsock)



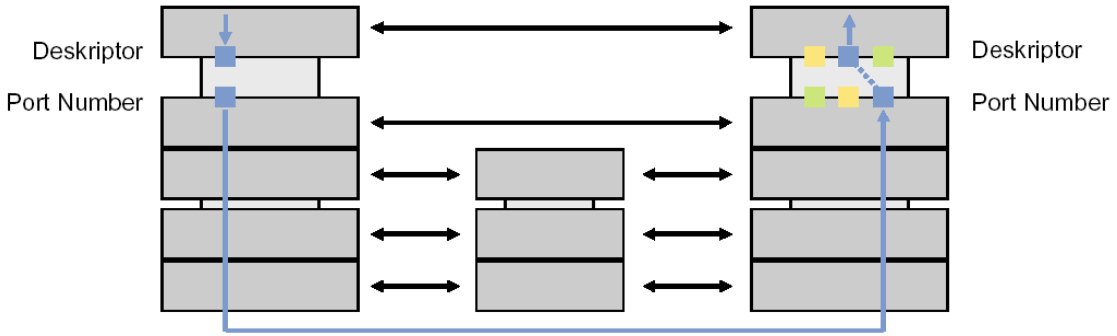
Sockets

FACHHOCHSCHULE TECHNIKUM WIEN

- Eigenschaften
  - ✓ werden von Netzwerk-Applikationen erzeugt, genutzt und freigegeben
  - ✓ liefern Adresse auf Anwendungsebene –Port Number
  - ✓ ermöglichen Lesen und Schreiben von Daten
    - » intern werden Sockets über Deskriptoren adressiert
    - » Ablauf analog zu einem einfachen Dateizugriff mit Standard I/O Operationen
  - ✓ ermöglichen Pufferung
  - ✓ Implementierung unter Linux im Kernel (Zugriff durch Systemaufrufe)

Ports

- Wozu werden Port Numbers benötigt?
  - ✓ auf einem Host können mehrere Netzwerk-Applikationen gleichzeitig aktiv sein
  - ✓ jeder Dienst nutzt einen Socket für die Verbindung zum TCP/IP-Stack
  - ✓ extern werden Sockets über Port Numbers adressiert
  - ✓ auf Hosts existiert eine eindeutige Zuordnung zwischen Deskriptor und Port Number



Ports

FACHHOCHSCHULE TECHNIKUM WIEN

- Eigenschaften
  - ✓ 16 bit number
  - ✓ well known ports (< 1024) nur durch privileged users nutzbar (root)

- eine Verbindung wird durch fünf Angaben eindeutig identifiziert:
  - ✓ Quelle IP Adresse und Port Number
  - ✓ Ziel IP Adresse und Port Number
  - ✓ Transport protocoll (TCP oder UDP)

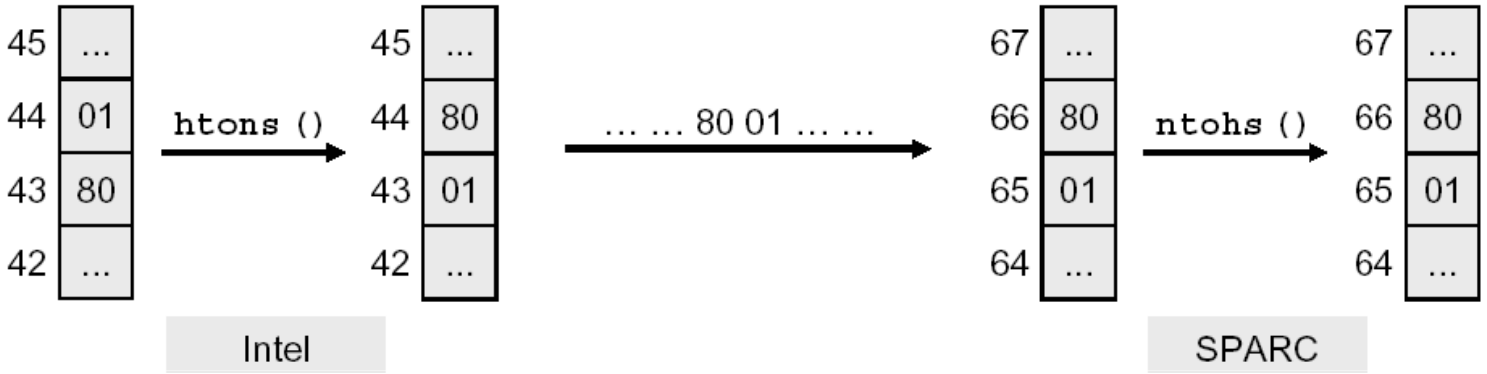
Datenaustausch

FACHHOCHSCHULE TECHNIKUM WIEN

- Wie werden Daten zwischen Systemen mit unterschiedlichen Datenrepräsentationen ausgetauscht?
  - ✓ big endian – werthöchstes Byte zuerst (SPARC, Motorola)
  - ✓ little endian – wertniedrigstes Byte zuerst (Intel)
- Lösung: Network Byte Order
  - ✓ Festlegung der Darstellung der Daten im Internet (big endian)
  - ✓ Konvertierung muss auf Anwendungsebene erfolgen

Network Byte Order

Beispiel: Übertragung des Wertes 384_{10} zwischen big und little endian



- Funktionen zur Umwandlung
  - ✓ `#include <netinet/in.h>`
  - ✓ `htons()` host-to-network für short
  - ✓ `ntohs()` network-to-host für short
  - ✓ `htonl()` host-to-network für long
  - ✓ `ntohl()` network-to-host für long

Socket API

FACHHOCHSCHULE TECHNIKUM WIEN

- Notwendige Includes für Socket Programmierung

```
#include <sys/socket.h>    // socket(), bind(), ...
#include <netinet/in.h>    // struct sockaddr_in
#include <arpa/inet.h>     // inet_ntoa(), ...
#include <unistd.h>        // read(), write(), close()
#include <errno.h>         // global var errno
```

Socket API
Übersicht

FACHHOCHSCHULE TECHNIKUM WIEN

- Funktionen zum Anlegen von Sockets und zum Aufbau der Verbindung:

- *socket(domain, type, protocol)*
  - ✓ Anlegen eines Socket
- *bind(socket, addr, addrlen)*
  - ✓ Zuweisen einer Adresse
- *connect(socket, addr, addrlen)*
  - ✓ *Aufbau einer Verbindung*
- *listen(socket, backlog)*
  - ✓ Zulassen eines Verbindungsaufbaues
- *accept(socket, addr, addrlen)*
  - ✓ Akzeptieren eines Verbindungsaufbaues

Socket API
Datenstrukturen

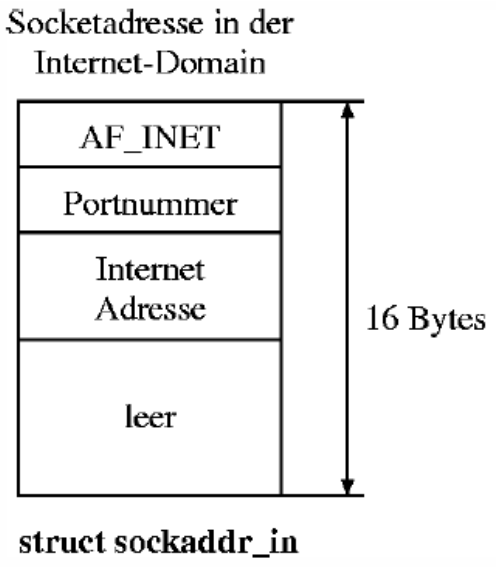
FACHHOCHSCHULE TECHNIKUM WIEN

- Datenstruktur für Adressierung (IPv4)

```
struct sockaddr_in{
   /* Länge der Struktur (16 Byte) */
   sa_family_t sin_family; /* nur AF_INET möglich(IPv4)*/
   in_port_t sin_port; /* 16-Bit TCP-oder UDP-Portnummer*/
   struct in_addr sin_addr; /* 32-Bit IPv4-Adresse */
   char sin_zero[8]; /* nicht belegt */
};

struct in_addr
{
   unsigned long s_addr;
};
```

  - ✓ sin_port und sin_addr in Network Byte Order!



Socket API
socket()

FACHHOCHSCHULE TECHNIKUM WIEN

- Anlegen

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

- Parameter
  - ✓ family
    - AF_INET für IPv4
    - AF_INET6 für IPv6
    - AF_LOCAL für UNIX domain sockets
  - ✓ type
    - SOCK_STREAM für Stream Sockets (TCP)
    - SOCK_DGRAM für Datagram Sockets (UDP)
    - SOCK_RAW für Raw Sockets direkt auf IP
  - ✓ protocol
    - In der Regel 0, nur für RAW Sockets verwendet

Socket API
bind()

FACHHOCHSCHULE TECHNIKUM WIEN

- Zuweisen einer Adresse

```
int bind (int sd, const struct sockaddr *myaddr,
          socklen_t myaddr_len)
```

- Parameter
  - ✓ sd — Socket Deskriptor (Ergebnis von socket())
  - ✓ *myaddr — Pointer auf Adressstruktur
  - ✓ myaddr_len — Größe der Adressstruktur in Bytes
- Anmerkungen
  - ✓ bind() assoziert Deskriptor mit IP-Adresse und Port und macht den Socket von außen sichtbar
  - ✓ wird beim Server verwendet

Socket API
Anlegen und Adressierung Beispiel

FACHHOCHSCHULE TECHNIKUM WIEN

```
int socket_fd;
struct sockaddr_in my_addr;

socket_fd = socket (AF_INET, SOCK_DGRAM, 0)// IPv4 und UDP
memset(&my_addr,0,sizeof(my_addr)); //Adressstrukur mit 0
initialisieren
my_addr.sin_family = AF_INET; // IPv4
my_addr.sin_port = htons (6000 ); // Port 6000
my_addr.sin_addr.s_addr = htonl (INADDR_ANY); // eigene IP-
Adresse

if ( bind( socket_fd,(struct sockaddr *) &my_addr,
          sizeof (my_addr)) == -1) {
      perror("bind error");
      exit(1);
}
```

Socket API
Adressumwandlung

FACHHOCHSCHULE TECHNIKUM WIEN

- Wandelt String mit IP-Adresse in Adressstruktur um

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

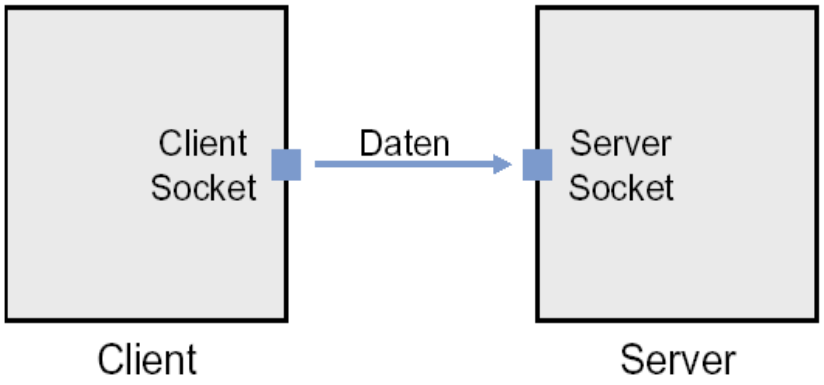
- Parameter
  - ✓ *cp — String der die IP-Adresse beinhaltet
  - ✓ *inp — Pointer auf Adressstruktur
- Anmerkungen
  - ✓ Adresse wird bereits in Network Byte Order umgewandelt
  - ✓ Funktion inet_ntoa() liefert wieder String
- Beispiele
  - ✓ `inet_aton("10.128.1.2",&(my_addr.sin_addr));`
  - ✓ `printf("%s", inet_ntoa(my_addr.sin_addr));`

Socket API
Vergleich UDP/TCP

FACHHOCHSCHULE TECHNIKUM WIEN

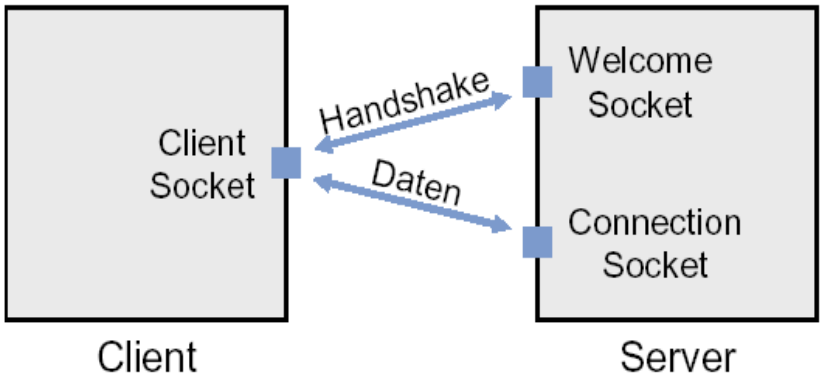
UDP

- ein Socket auf Client
- ein Socket auf Server
- kein Verbindungsaufbau
- unidirektionale Datenübertragung

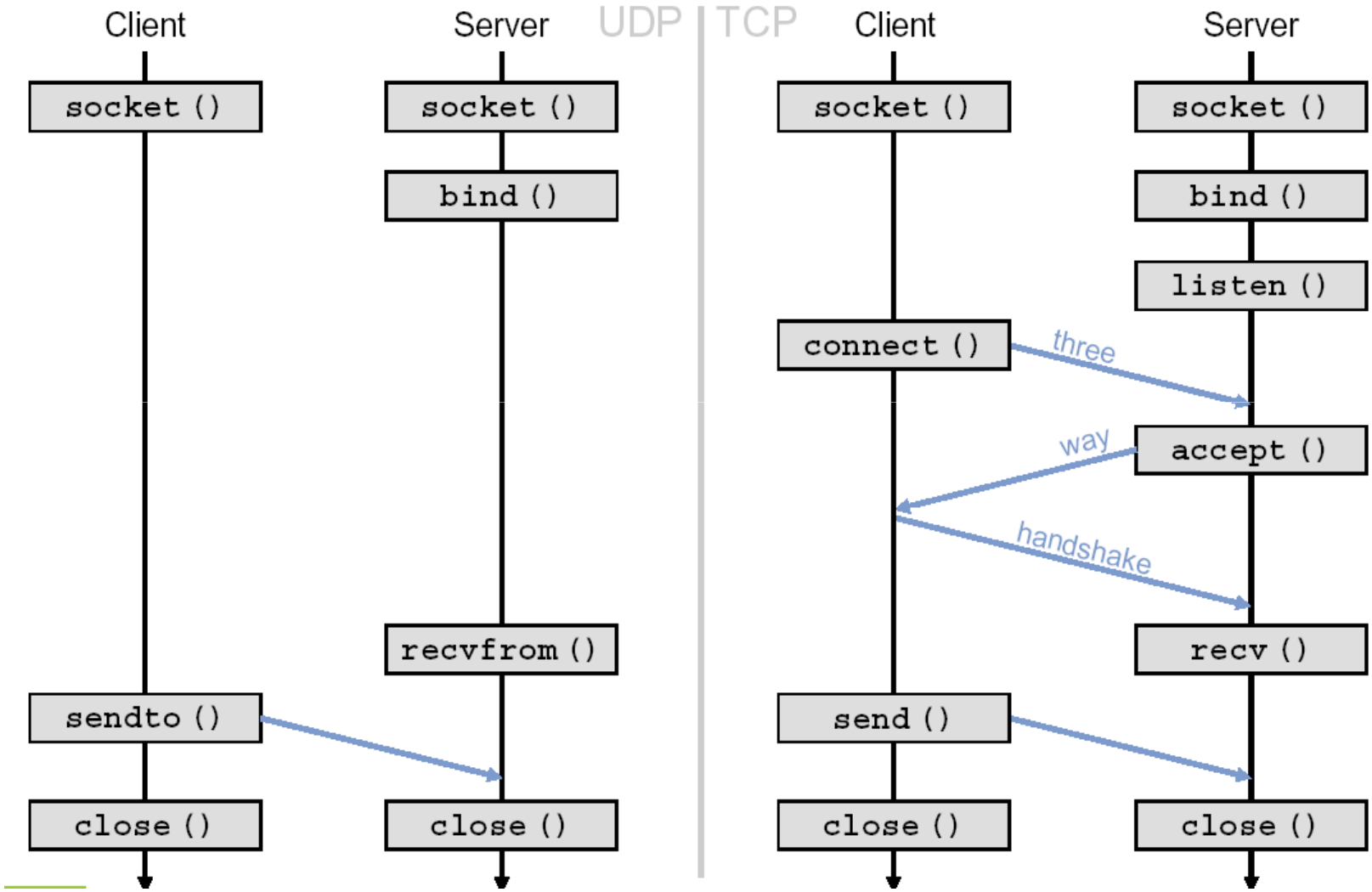


TCP

- ein Socket auf Client
- zwei Sockets auf Server
  - Welcome Socket (auch Listen Socket) für Verbindungsaufbau (Handshake)
  - Connection Socket (auch Connecting Socket) für Datenübertragung
- bidirektionale Datenübertragung



Socket API
Vergleich Ablauf UDP/TCP



Socket API
connect()

FACHHOCHSCHULE TECHNIKUM WIEN

- Aufbau einer Verbindung am Client

```
int connect (int sd, const struct sockaddr *servaddr,
             socklen_t addrlen)
```

- Parameter
  - ✓ sd — Socket Deskriptor (Ergebnis von socket())
  - ✓ *servaddr — Pointer auf Adressstruktur mit Daten des Empfängers (Servers)
  - ✓ addrlen — Größe der Adressstruktur in Bytes
- Anmerkungen
  - ✓ connect() baut eine Verbindung über TCP zum Server auf
  - ✓ löst den three way handshake aus

Socket API
listen()

FACHHOCHSCHULE TECHNIKUM WIEN

- Warten auf Verbindungsanforderung am Server

```
int listen (int sd, int backlog)
```

- Parameter
  - ✓ sd — Socket Deskriptor (Ergebnis von socket())
  - ✓ backlog — Anzahl der Verbindungen die in einer Queue auf ein accept() warten dürfen, ist die Queue voll bekommt der Client „Connection refused“

Socket API
accept()

FACHHOCHSCHULE TECHNIKUM WIEN

- Akzeptieren einer Verbindung am Server

```
new_sd = accept (int sd, struct sockaddr *cliaddr,
                 socklen_t *addrlen)
```

- Parameter
  - ✓ sd — Socket Deskriptor (Ergebnis von socket())
  - ✓ *cliaddr — Pointer auf Adressstruktur für Daten des Clients
  - ✓ addrlen — Größe der Adressstruktur in Bytes
- Anmerkungen
  - ✓ accept() liefert einen neuen Socket Deskriptor new_sd für die weitere Kommunikation mit dem Client
  - ✓ accept() blockiert bis eine Verbindung aufgebaut ist

Socket API
close()

FACHHOCHSCHULE TECHNIKUM WIEN

- Schließen eines Sockets

```
int close (int sd)
```

- Parameter
  - ✓ sd — Socket Deskriptor (Ergebnis von socket() oder von accept())
- Anmerkungen
  - ✓ weder Senden noch Empfangen nach close() möglich
  - ✓ gibt Deskriptor sd auch frei

Socket API
Datenaustausch

FACHHOCHSCHULE TECHNIKUM WIEN

- Funktionen zum Senden und Empfangen von Daten

- *send(…), recv(…)*
  - ✓ Senden/Empfangen von Daten
- *sendto(…), recvfrom(…)*
  - ✓ Senden/Empfangen mit Adressangabe für Datagram Sockets

- Die Low-Level I/O Funktionen read() und write() können ebenfalls verwendet werden

Socket API
send()

FACHHOCHSCHULE TECHNIKUM WIEN

- Senden in Stream Sockets

```
int send(int sd, const void *msg, int len, int flags);
```

- Parameter
  - ✓ sd — Socket Deskriptor (Ergebnis von socket())
  - ✓ *msg — Pointer auf Daten, die gesendet werden sollen
  - ✓ len — Länge der Daten in Bytes
  - ✓ flags — im Normalfall 0
- Anmerkungen
  - ✓ send() retourniert Anzahl der gesendeten Bytes oder -1 im Fehlerfall
  - ✓ Achtung: Anzahl kann kleiner als len sein, daher entweder kleine Pakete schicken (z.B. 1K) oder send() in einer Schleife aufrufen
- Beispiel

```
const char *string = “Hello!“;
int len = strlen(string) + 1;
if (send(sd, string, len, 0) == -1)
{
   /* error */
}
```

Socket API
send()

FACHHOCHSCHULE TECHNIKUM WIEN

- Beispiel für einen eigenen send() wrapper

```
int sendall(int sd, char *buf, int *len)
{
   int total = 0; // how many bytes we've sent
   int bytesleft = *len; // how many we have left to send
   int n;

   while(total < *len) {
      n = send(sd, buf+total, bytesleft, 0);
      if (n == -1) { break; }
      total += n;
      bytesleft -= n;
   }
   *len = total; // return number actually sent here
   return n==-1?-1:0; // return -1 on failure, 0 on success
}
```

Socket API
recv()

FACHHOCHSCHULE TECHNIKUM WIEN

- Empfangen in Stream Sockets

```
int recv(int sd, void *msg, int len, int flags);
```

- Parameter
  - ✓ sd — Socket Deskriptor (Ergebnis von socket())
  - ✓ *msg — Pointer auf Empfangsbuffer für Daten
  - ✓ len — Größe der Empfangsbuffers in Bytes
  - ✓ flags — Empfangsmodus, im Normalfall 0
- Anmerkungen
  - ✓ recv() retourniert Anzahl der empfangenen Bytes oder -1 im Fehlerfall bzw. 0 wenn Remote Socket geschlossen wurde
  - ✓ Achtung: Anzahl kann kleiner als len sein
  - ✓ recv() blockiert standardmäßig bis Daten empfangen werden
- Beispiel

```
char string[LEN];
int len=0;

if ((len=recv(sd, string, LEN-1, 0)) == -1){  /* error */}
string[len] = '\0';
```

Socket API
read() und write()

FACHHOCHSCHULE TECHNIKUM WIEN

- Statt send() und recv() können auch die normalen Low-Level I/O Funktionen read() und write() verwendet werden, da Sockets ja genauso wie Dateien mit Deskriptoren ansprechbar sind
  - ✓ gleiche Parameter und Syntax bis auf flags

```
ssize_t read(int fd, void *puffer, size_t anzahl_bytes);

ssize_t write(int fd, void *puffer, size_t anzahl_bytes);
```

Socket API
sendto()

FACHHOCHSCHULE TECHNIKUM WIEN

- Senden in Datagram Sockets

```
int sendto(int sd, const void *msg, int len, int flags,
           const struct sockaddr *to, int tolen);
```

- Parameter
  - ✓ sd — Socket Deskriptor (Ergebnis von socket())
  - ✓ *msg — Pointer auf Daten, die gesendet werden sollen
  - ✓ len — Länge der Daten in Bytes
  - ✓ flags — im Normalfall 0
  - ✓ *to — Zeiger auf Empfängeradresse
  - ✓ tolen — Länge der Empfängeradresse in Bytes
- Anmerkungen
  - ✓ sendto() retourniert Anzahl der gesendeten Bytes oder -1 im Fehlerfall

Socket API
recvfrom()

FACHHOCHSCHULE TECHNIKUM WIEN

- Empfangen in Datagram Sockets

```
int recvfrom(int sd, void *msg, int len, int flags,
             struct sockaddr *from, int fromlen);
```

- Parameter
  - ✓ sd — Socket Deskriptor (Ergebnis von socket())
  - ✓ *msg — Pointer auf Empfangsbuffer für Daten
  - ✓ len — Größe der Empfangsbuffers in Bytes
  - ✓ flags — Empfangsmodus, im Normalfall 0
  - ✓ *from — Pointer auf Buffer für Senderadresse
  - ✓ fromlen — Größe des Adressbuffers in Bytes
- Anmerkungen
  - ✓ recvfrom() retourniert Anzahl der empfangenen Bytes oder -1 im Fehlerfall

Socket API

FACHHOCHSCHULE TECHNIKUM WIEN

- weitere nützliche Befehle
  - ✓ Adresse des anderen Ende(Peers) eines stream Sockets

```
int getpeername(int sockfd, struct sockaddr *addr,
                int *addrlen);
```

  - ✓ Name des eigenen Hosts

```
int gethostname(char *hostname,size_t size);
```

  - ✓ DNS Abfrage

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

- nützliche Systembefehle
  - ✓ netstat, tcpdump
  - ✓ lsof -i

Literatur

- Beej's Guide to Network Programming
  - ✓ pdf im Literatur Verzeichnis
- Linux-Unix Programmierung Open Book
  - ✓ Kapitel Netzwerkprogrammierung
  - ✓ http://openbook.galileocomputing.de/c_von_a_bis_z/025_c_netzwerkprogrammierung_001.htm
- Manual Pages

FACHHOCHSCHULE TECHNIKUM WIEN

Verteilte Systeme

Netzwerkprogrammierung
Client Server Architekturen

Design von Server-Applikationen

FACHHOCHSCHULE TECHNIKUM WIEN

Merkmale

- ✓ stateful oder stateless
- ✓ connectionless oder connection oriented
- ✓ iterative oder concurrent

Stateful Server

- ✓ Zustandsinformationen werden über mehrere Übertragungsvorgänge gespeichert
- ✓ Probleme:
  - » Clients können zu unerwarteten Zeitpunkten aus- und eingeschaltet werden
  - » durch Fehler im Netz können zu unerwarteten Zeitpunkten Daten verloren oder vervielfacht werden
- ✓ erfordern also eine äußerst sorgfältige Implementierung

Design von Server-Applikationen

FACHHOCHSCHULE TECHNIKUM WIEN

- Stateless Server
  - ✓ die meisten Server-Applikationen im Internet sind stateless
  - ✓ jeder Übertragungsvorgang wird von Verbindungsauf- und Abbau begrenzt
    - dabei ist es egal, ob ein einzelnes Objekt oder eine Reihe von Objekten (z. B. Persistent Connections über HTTP) übertragen wird
  - ✓ Erweiterung zur Speicherung eines Status erfordert i. d. R. hohen zusätzlichen Aufwand
    - Beispiel Cookies im WWW

Design von Server-Applikationen

FACHHOCHSCHULE TECHNIKUM WIEN

Connectionless Server

- ✓ typisch im Internet über UDP
- ✓ geringer Overhead, d. h. hohe Performance
- ✓ keine Sicherheit über die übertragenen Daten
- ✓ ggf. aufwändige Implementierung, um Zuverlässigkeit zu erreichen
  - » d. h. gesicherte Übertragung erst auf Application Layer, z. B. TFTP
- ✓ theoretisch keine Begrenzung der Anzahl gleichzeitiger Clients
  - » alle Daten werden über denselben Socket-Deskriptor empfangen
  - » eine Trennung der Clients muss gezielt auf Application Layer erfolgen

Design von Server-Applikationen

FACHHOCHSCHULE TECHNIKUM WIEN

- Connection oriented Server
  - ✓ typisch im Internet über TCP
  - ✓ Verbindungs-Management wird vom Betriebssystem übernommen
  - ✓ durch notwendigen Overhead geringere Performance
  - ✓ hohe Übertragungssicherheit
  - ✓ einfache Implementierung
  - ✓ jede Verbindung benötigt einen eigenen Socket, d. h. ist über Deskriptor identifizierbar
  - ✓ daraus folgt limitierte Anzahl gleichzeitig zu bearbeitender Clients

Design von Server-Applikationen

FACHHOCHSCHULE TECHNIKUM WIEN

Iterative Server

- ✓ Server-Applikation kann nur jeweils eine Verbindung bearbeiten
- ✓ typisch für einfache TCP-Server
- ✓ empfangene Requests müssen warten, bis aktuelle Verbindung abgebaut wird
- ✓ blockierender Server
- ✓ ggf. werden eingehende Requests abgewiesen

Design von Server-Applikationen

FACHHOCHSCHULE TECHNIKUM WIEN

Concurrent Server

- ✓ nebenläufiger Server
- ✓ der eigentliche Server-Prozess nimmt lediglich Requests entgegen
- ✓ der Server-Prozess startet für jede Verbindung einen Kind-Prozess
- ✓ Request wird an Kind-Prozess übergeben und dort verarbeitet
- ✓ nach Abbau der Verbindung wird Kind-Prozess beendet
- ✓ hohe Performance durch parallele Bearbeitung mehrerer Verbindungen
- ✓ typisch für TCP-Server mit mehreren gleichzeitigen Verbindungen
  - WWW, FTP, …
- ✓ jeder Kind Prozess belegt zusätzliche System Ressourcen

Concurrent Server Implementierung

FACHHOCHSCHULE TECHNIKUM WIEN

- Implementierung Concurrent Server

- forking server
  - ✓ klassische Variante unter UNIX
  - ✓ durch fork() wird Kind-Prozess gestartet
  - ✓ in TCP-basiertem Server wird fork() nach accept() aufgerufen
  - ✓ in UDP-basiertem Server wird fork() nach recvfrom() aufgerufen
  - ✓ fork() ist nicht geeignet für hohe Performance

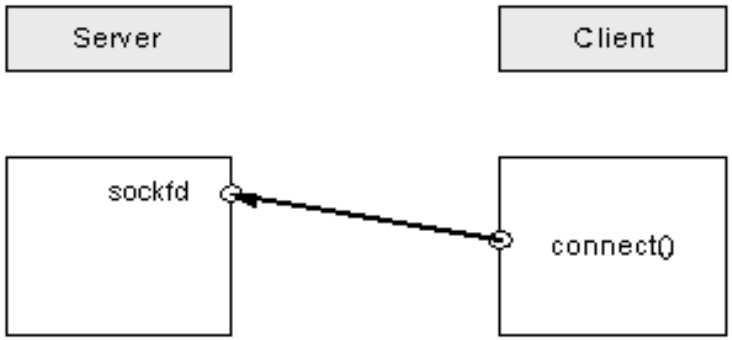
forking Server Implementierung

FACHHOCHSCHULE TECHNIKUM WIEN

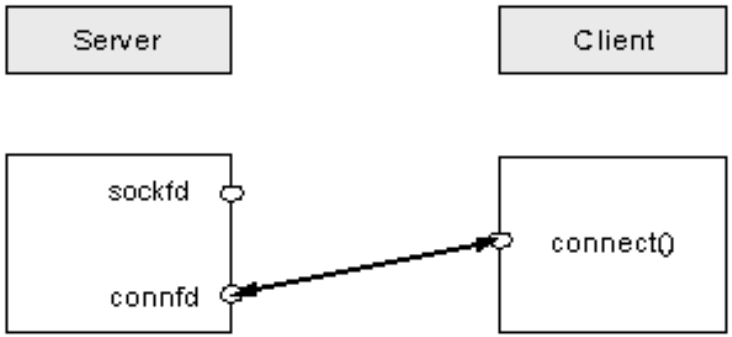
```
while(1) {
   connfd = accept( sockfd,(struct sockaddr *)&adresse, &adrlaenge);
   if( connfd < 0 ) {
      if( errno == EINTR ) continue;
      else { printf("Fehler bei accept() ...\n");
             exit(EXIT_FAILURE);
           }
   }
   printf (" ...Daten empfangen\n");
   /* Neuen Kind-Prozess-Server starten */
   if ((pid = fork ()) == 0) {
      close (sockfd);
      /* Arbeit des Kindprozesses steht hier */
      close(connfd);
      exit(EXIT_SUCCESS); /* Ende Kindprozess */
   }
   /* Elternprozess */
   close (connfd);
 }
```

forking Server Implementierung

- Aufbau der Verbindung durch den Client

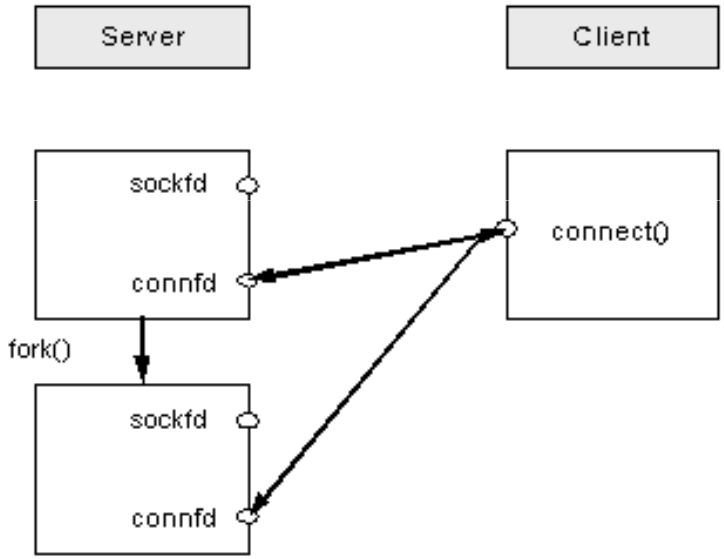


- Nach 3way Handshake, Rückkehr von accept()



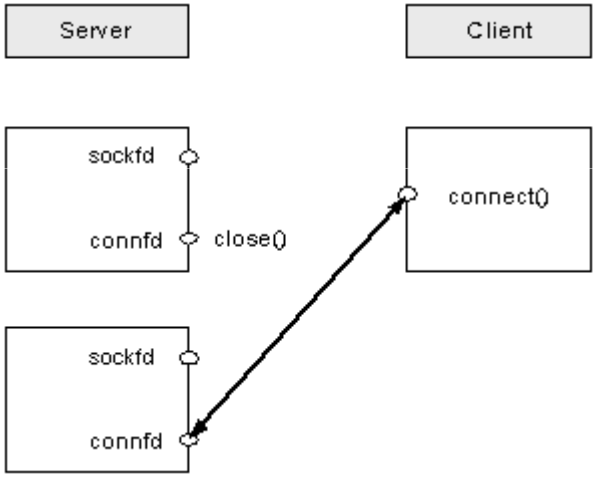
forking Server Implementierung

- Nach fork()
  - ✓ Kindprozess erbt offene Socket Deskriptoren



forking Server Implementierung

- ✓ Nicht benötigte Socket Enden müssen geschlossen werden
  - » Kindprozess schließt Listening Socket
  - » Elternprozess schließt connection Socket



  - » Wird die Verbindung beendet, schließt der Kindprozess den connection Socket und terminiert
  - » Elternprozess muss auf die Terminierung reagieren, sonst entsteht ein Zombie Prozess!!!

forking Server Implementierung

FACHHOCHSCHULE TECHNIKUM WIEN

- Signalbehandlung im Elternprozess, um Zombie Prozesse zu verhindern
  - ✓ Abfangen des Signals SIGCHLD

```
void no_zombie (int signr) {
   pid_t pid;
   int ret;
   while ((pid = waitpid (-1, &ret, WNOHANG)) > 0)
      printf ("Child-Server mit pid=%d hat sich beendet\n",pid);
   return;
}

...
/* in main() */
(void) signal (SIGCHLD, no_zombie);
```

Concurrent Server Implementierung

FACHHOCHSCHULE TECHNIKUM WIEN

- preforked Server
  - ✓ Server erzeugt zu Beginn eine gewisse Anzahl an Kind-Prozessen, die alle über den selben listening socket verfügen
  - ✓ Jeder Kindprozess blockiert im accept() Call
  - ✓ eingehende Requests werden an Kind-Prozesse verteilt
  - ✓ Kind-Prozesse werden nach Verbindungsabbau nicht beendet
  - ✓ bei Bedarf werden zusätzliche Kind-Prozesse gestartet
  - ✓ Beispiel: WWW-Server Apache Version 1.x
  - ✓ Performance Steigerung im Vergleich zum forking Server

Concurrent Server Implementierung

FACHHOCHSCHULE TECHNIKUM WIEN

- Alternative: Threads
  - ✓ für jede Verbindung wird ein Thread gestartet
  - ✓ schneller als fork(), geringerer Verbrauch von Ressourcen
  - ✓ einfacher Austausch von Informationen zwischen verschiedenen Threads
  - ✓ auch „prethreaded“ Server möglich
  - ✓ maximale Anzahl an Threads pro Prozess ist begrenzt, daher auch Kombination von forking Server und Threads möglich

  - ✓ Threads Einführung siehe GPR2 Folien bzw.
  - ✓ http://www.llnl.gov/computing/tutorials/pthreads/

Threads
Implementierung

```
while(1) {
   connfd = accept( sockfd,(struct sockaddr *)&adresse,
&adrlaenge);
   if( connfd < 0 ) {
      if( errno == EINTR ) continue;
      else { printf("Fehler bei accept() ...\n");
             exit(EXIT_FAILURE);
           }
   }
   printf (" ...Daten empfangen\n");

   pthread_create(&th,NULL,threading_socket,(void *)&connfd);
}
```

Threads
Implementierung

FACHHOCHSCHULE TECHNIKUM WIEN

```
void * threading_socket (void *arg)
{
        int clientfd = *((int *)arg);
        pthread_detach (pthread_self ());

         /* Kommunikation mit Client steht hier */

        close (clientfd);
        return NULL;
}
```