

Verteilte Systeme

Betriebssysteme II

Kapitel 4: Elementare Programmiermodelle

Kapitel 4.1: Sockets

Prof. Dr. Wolfgang Küchlin

Dipl.-Inform., Dr. sc. techn. (ETH)

Arbeitsbereich Symbolisches Rechnen
Wilhelm-Schickard-Institut für Informatik
Fakultät für Informations- und Kognitionswissenschaften

Universität Tübingen

Steinbeis Transferzentrum
Objekt- und Internet-Technologien (OIT)

Wolfgang.Kuechlin@uni-tuebingen.de
<http://www.sr.informatik.uni-tuebingen.de>



Überblick

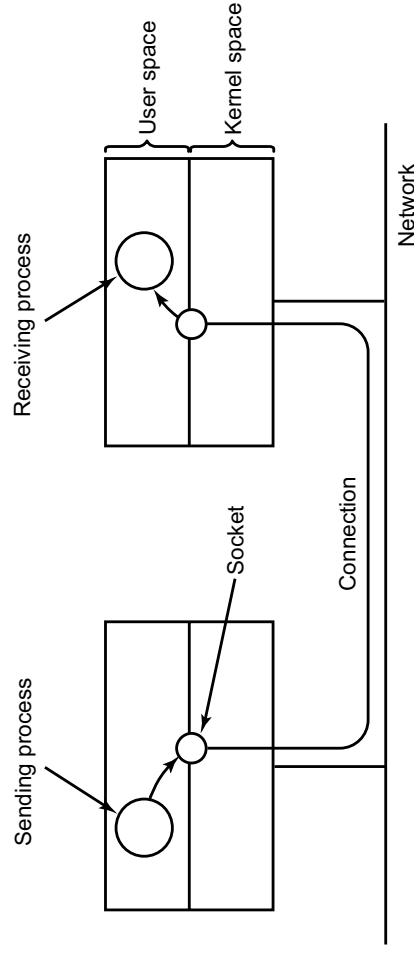
- UNIX Sockets
- Pipes und Socketpairs
- Prozessverbindung durch Sockets
 - Stream Sockets und Datagram Sockets
- Sockets in Java

2

Wolfgang Küchlin, WSI und STZ OIT, Uni Tübingen 08.06.2009



The use of sockets for networking



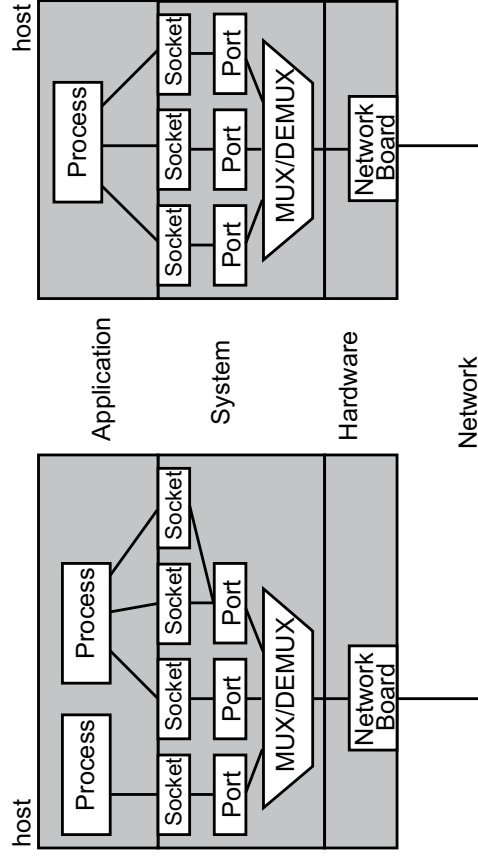
Tanenbaum, „Modern Operating Systems“, Abb. 10-19

Wolfgang Küchlin, WSI und STZ OIT, Uni Tübingen 08.06.2009

3



Abbildung von Sockets auf Ports



Wolfgang Küchlin, WSI und STZ OIT, Uni Tübingen 08.06.2009

4



UNIX Sockets

- TCP/IP Realisierung der OSI TSAPs (Transport Service Access Point).
 - Kommunikationsschnittstelle zwischen Anwendungsprogrammen und den TCP/IP Protokollen
- 1982/83 mit 4.2 BSD in UNIX eingeführt
- Stellen jeweils das Ende einer Prozess-Prozess-Verbindung dar.
- Grundidee:
 - Netzverbindung bezüglich Schreiben und Lesen wie eine Datei zu behandeln
 - Analog zu *file descriptors* jetzt *socket descriptors*
- Uniformität:
 - socket-Verbindungen sowohl zwischen Prozessen auf demselben Rechner als auch zwischen Prozessen, die durch das Internet verbunden sind
 - *UNIX-Domain* und *Internet-Domain*
 - *Domain* = Adressraum, wo eine bestimmte Art von Adressen gültig ist
 - Innerhalb der UNIX Domain sind socket-Verbindungen über das Dateisystem implementiert.



Pipes und Socketpairs

- In 4 BSD:
 - Pipes werden über sockets implementiert.
 - Pipe Kommunikation unidirektional, socket Kommunikation ist bidirektional
- Beispiel:

```
int endpoint[2];
pipe (endpoint);
```

 - pipe mit zwei Deskriptoren.
 - Von Deskriptor endpoint[0] lesen und auf Deskriptor endpoint[1] schreiben
- Pipes sind *Fluß-(stream)-Verbindungen*
 - ununterbrochener Fluss von Bytes



Pipes und Socketpairs

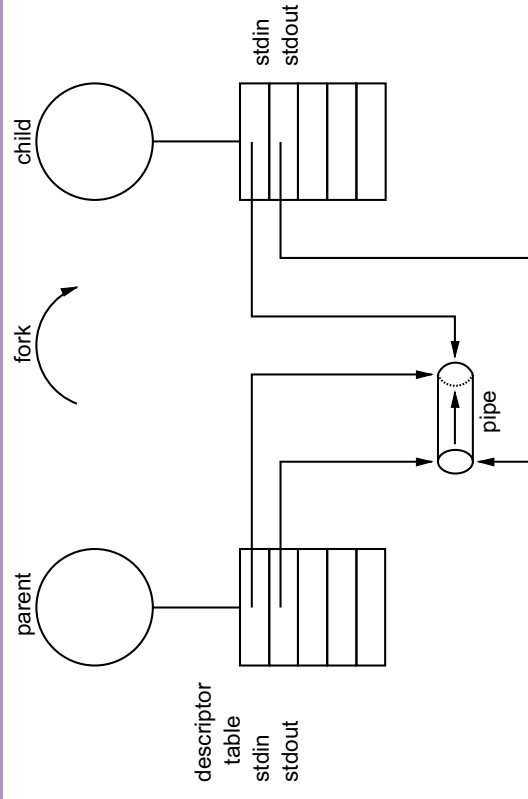
- Eintrag in die ersten freien Plätze der Deskriptortabelle des Prozesses
- Beispiel:

```
close(stdin);
close(stdout);
pipe(endpoint);
```

 - Pipe von stdout zu stdin
 - Nach Erzeugen der Pipe: `fork()` Aufruf → Duplizierung der Deskriptoren und somit der Pipe
 - Vater und Kind können über die Pipe kommunizieren



Pipes und Socketpairs



Socketpairs

- Ein verbundenes Paar aus Sockets stellt eine Verallgemeinerung einer Pipe dar, da der Datenfluss bidirektional ist.

- Aufruf:

```
#include <sys/types.h>
#include <sys/socket.h>
int sockets[2], errorcode;
errorcode = socketpair(domain,type,protocol,sockets);
ASSERT(errorcode == 0);
```

- Liefert ein Paar Socketdeskriptoren im Array sockets.

- Einschränkung:

```
domain = AF_UNIX
type = SOCK_STREAM
protocol = 0 /* ein passendes (TCP) wird gewählt */
```



Socketpairs

- Zugriff:

```
char buf [1024];
#define DATA "Once upon a time ..."
read (sockets[1], buf, 1024);
write (sockets[1], DATA, sizeof(DATA));
```

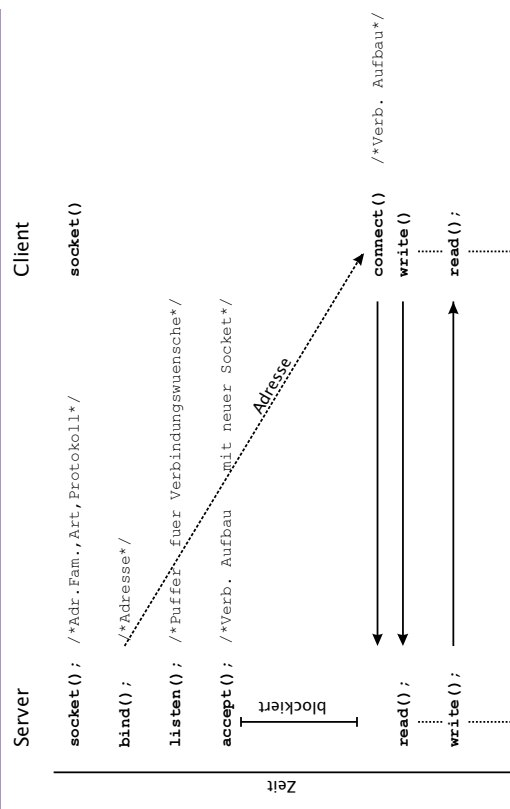


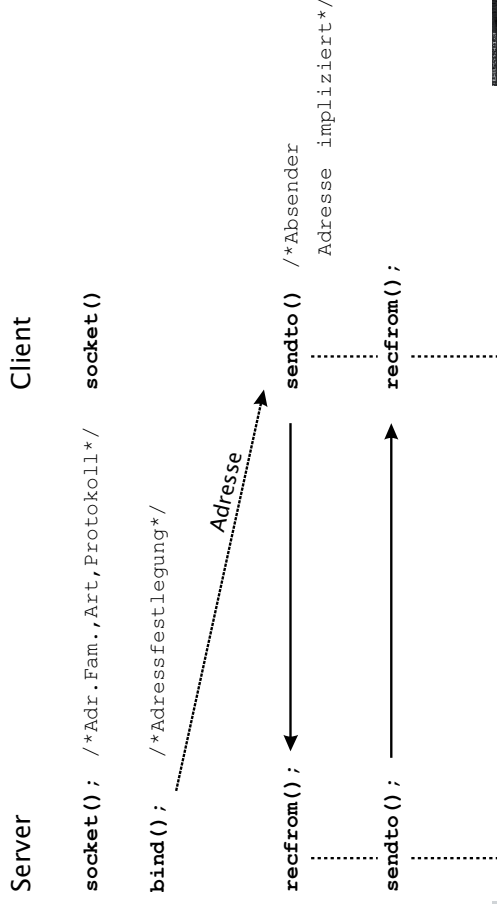
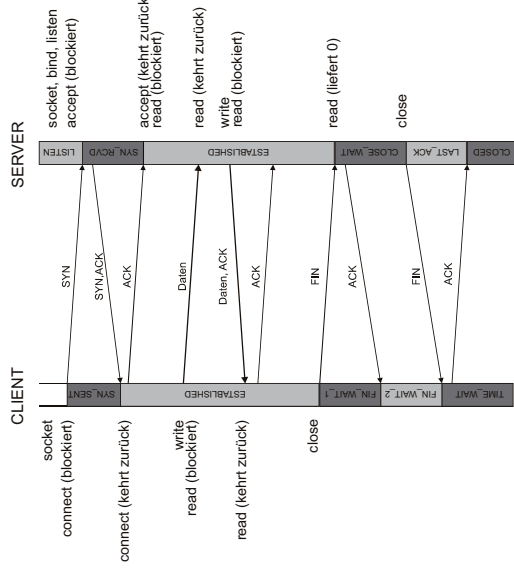
Prozessverbindungen durch Sockets

- fünf Komponenten einer Prozessverbindung
(protocol, local host, local port, remote host, remote port)
- Bei einem verbindungsorientierten Protokoll (Bsp.: TCP)
 - Adressen Teil der **Verbindung**.
 - Klient: connect() und Server: accept()
 - Rendezvous, bei dem Verbindung aufgebaut wird
 - Nachfolgendes Lesen und Schreiben unter Angabe der Verbindung
 - ohne weitere Angabe der Adressen
- Bei einem Datagramm (Brief-) orientierten Protokoll (Bsp.: UDP)
 - Zunächst: Endpunkte eines Kommunikationsweges werden aufgebaut
 - adressierbare Briefkästen
 - Jede Nachrichtenübermittlung als unabhängiges Paket unter Angabe der Zieladresse über lokalen Endpunkt



Aufreifequenz für Stream Sockets





- `socket()`
 - `domain`: Adressfamilie; `type`: Art der Komm. (Datagram od. Stream);
 - Resultat: `socket-Deskriptor` = Index in die Deskriptorentabelle des Prozesses
- heutige UNIX-Systeme: Normalerweise nur jeweils eine Protokollimplementierung für jede (`domain, type`) Kombination vorhanden
 - Theoretisch könnte es mehrere Implementierungen geben
- Daher: Nutzung der Standardimplementierung durch Angabe von:
 - wobei
 - `protocol = PF_UNSPEC /* Protocol family */`
 - `sys/socket.h`
 - `# define PF_UNSPEC 0`



- relevante Fälle:
 - `domain = AF_UNIX // Unix domain`
 - `AF_INET // Internet domain`
 - `type = SOCK_DGRAM // Datagram Connection`
 - `SOCK_STREAM // Stream Connection`
- Zuordnung zu einer Adresse
 - allgemeiner Adressrahmen:

```
struct sockaddr {
    u_short sa_family; /* address family AF_xxx */
    char sa_data[14]; /* up to 14 bytes of direct address */
}
```
 - Je nach Adressraum werden die 14 Bytes unterschiedlich genutzt und interpretiert.



Adressfamilien

```

0  /* unspecified */
1  #define AF_UNSPEC
2  /* local to host (pipes, portals) */
3  #define AF_INET
4  /* internet: UDP, TCP, etc. */
5  #define AF_INET6
6  /* arpanet imp addresses */
7  #define AF_PUP
8  /* pup protocols: e.g. BSP */
9  #define AF_CHAOS
10 /* mit CHAOS protocols */
11 #define AF_NS
12 /* nbs protocols */
13 #define AF_NBS
14 /* european computer manufacturers */
15 #define AF_ECMA
16 /* datakit protocols */
17 #define AF_CCITT
18 /* IBM SNA */
19 #define AF_SNA
20 /* DECnet */
21 #define AF_DLI
22 /* Direct data link interface */
23 #define AF_LAT
24 /* LAT */
25 #define AF_HYLINK
26 /* NSC Hyperchannel */
27 #define AF_APPLETALK
28 /* Apple Talk */
29 #define AF_NIT
30 /* Network Interface Tap */
31 #define AF_ISO
32 /* IEEE 802.2, also ISO 8802 */
33 #define AF_802
34 /* umbrella for all families used */
35 #define AF_X25
36 /* CCITT X.25 in particular */
37 #define AF_X25
38 /* AFI = 47, IDI = 4 */
39 #define AF_OSI
40 /* U.S. Government OSI */
41 #define AF_GOSIP
42 /* Novell Internet Protocol */
43 #define AF_IPX
44 /* Internal Routing Protocol */
45 #define AF_ROUTE
46 /* Link-layer interface */

```



Stream-Socket Verbindungen

- Aufbau einer Stream-Socket Verbindung ist asymmetrisch
 - anders als bei einer Datagramm Verbindung
- Ein Prozess übernimmt die Rolle des Servers, der andere die des Clients.
 - Server richtet Socket ein (socket(), bind(), listen()) und bietet über diese seine Dienste an
 - Mit connect-Befehl kann Client (via seiner eigenen Socket) Verbindung initiieren, um Anfrage zu starten



Stream-Socket Verbindungen – Beispiel

Unix-Domain	Internet-Domain
<pre> #include <sys/socket.h> #include <sys/un.h> #include <sys/types.h> int sd, nlen; struct sockaddr_un sock; sd = socket(AF_UNIX, SOCK_STREAM, 0); sock.sun_family = AF_UNIX; strcpy(sock.sun_path, <Socket-Name>); nlen = sizeof(struct sockaddr_un); connect(sd, (struct sockaddr) &sock, nlen) </pre>	<pre> #include <sys/socket.h> #include <netinet/in.h> #include <sys/types.h> #include <netdb.h> int sd, nlen; struct sockaddr_in sock; sd = socket(AF_INET, SOCK_STREAM, 0); sock.sin_family = AF_INET; sock.sin_port = htons(<Portno>); bcopy(<Host-Name>, sock.sin_addr, sizeof(sockaddr_in)); nlen = sizeof(struct sockaddr_in); </pre>



Stream-Socket Verbindungen

- Bei Stream Sockets ist Verbindung fest
 - Auflösen durch close() einer der beteiligten Sockets
- Server richtet durch listen(sd, MAX) eine Queue ein
 - anstehende Verbindungen werden in Queue aufgelistet, bis sie zur Abarbeitung akzeptiert worden sind
 - Queue hat normalerweise Länge von höchstens MAX=5
- Nach Einrichtung: Server kann Verbindungen aus dieser Queue akzeptieren



Stream-Socket Verbindungen

Unix-Domain	Internet-Domain
<pre>int newsock; struct sockaddr_un from; nlen = sizeof(struct sockaddr_un);</pre>	<pre>int newsock; struct sockaddr_in from; nlen = sizeof(struct sockaddr_in);</pre>
<pre>newsock = accept (sd, (struct sockaddr_un) &from, &nlen)</pre>	

- *accept* erzeugt Kopie der eingerichteten Socket
 - über diese neue Socket wird kommuniziert, Port bleibt gleich!
- Nach *accept*-Aufruf steht in *from* die Socket Adresse des Client.
- *accept* blockiert normalerweise den Server-Prozess.
 - Problem, wenn mit mehreren Clients gleichzeitig kommuniziert werden soll
- Alternativ: *select*-Befehl



Stream-Socket Verbindungen – Beispiel

```
#include <sys/time.h>
#include <sys/types.h>
fd_set rt, wt;
struct timeval wait;
int nb;

for(;;) {
    wait.tv_sec = 5;
    wait.tv_usec = 0;

    FD_ZERO(&rt);
    FD_ZERO(&wt);
    FD_SET(sd1, &rt);
    FD_SET(sd2, &rt);

    nb = select(FD_SETSIZE, &rt, &wt, (fd_set *) 0, &wait);

    if (FD_ISSET(sd1, &rt)) { ... } /* s1 is ready to be read from */
    if (FD_ISSET(sd2, &rt)) { ... } /* s2 is ready to be read from */
}
```



Stream-Socket Verbindungen – Beispiel

- Über die so eingerichteten Sockets kann kommuniziert werden mit:


```
int read( sd, &buf, sizeof(buf));
int write( sd, DATA, sizeof(DATA));
int recv( sd, &buf, sizeof(buf), flag);
int send( sd, DATA, sizeof(DATA), flag);
```
- Als flags stehen unter anderem zur Verfügung:


```
#include <sys/socket.h>
MSG_OOB      // send/receive out of band data
MSG_PEEK     // treat read data as still unread
```
- Eilige Sendungen
 - send(..., MSG_OOB)
- gelesene Message nicht aus der Socket löschen
 - recv(..., MSG_PEEK)
- entblocktes Lesen mit *fcntl* (→ Lesen von Files)


```
#include <fcntl.h>
fcntl(sd, FSETFL, FNDELAY);
```



OO-Kapselung von Socket-Schnittstellen

- ohne OO-Kapselung
 - Belastung der Programmierer mit „unnötigen Details“
- Konzept wird mit OO-Kapselung oftmals klarer
- 2 Beispiele:
 - Sockets in ACE (C++ Library)
 - Java Sockets Library

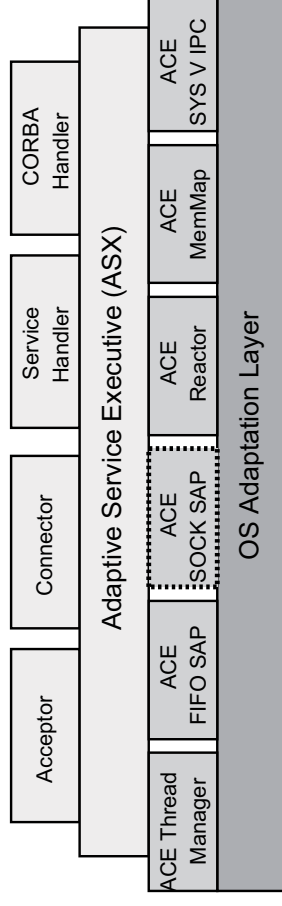


ACE: Adaptive Communication Environment

- objekt-orientierte System-Plattform für Entwicklung von Kommunikationssoftware
- Implementierungen für:
 - UNIX Systeme z.B. SunOS 4.x, Solaris 2.x, AIX, SGI IRIX, u.a.)
 - Windows NT (Win32)
 - MVS OpenEdition
- frei erhältlich
- Umfang: ca. 85000 LOC (Multi-Plattform Code)



ACE – Überblick

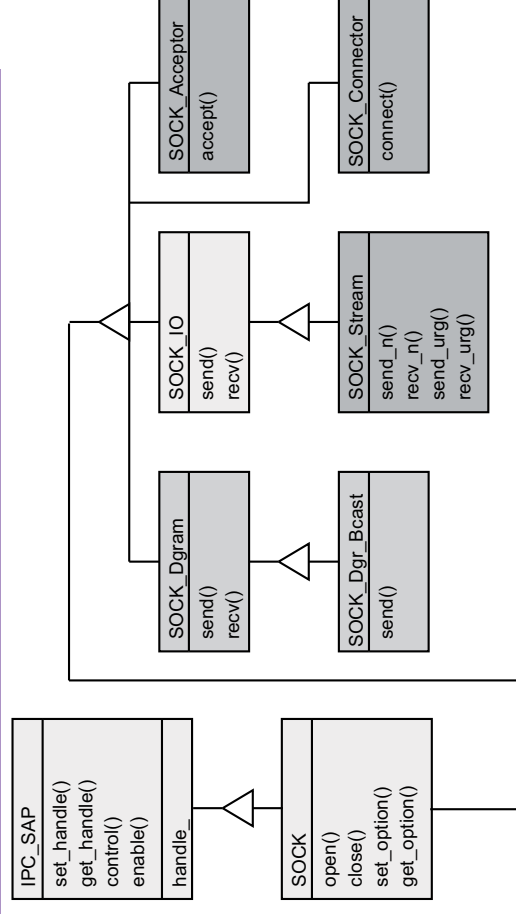


ACE – Überblick

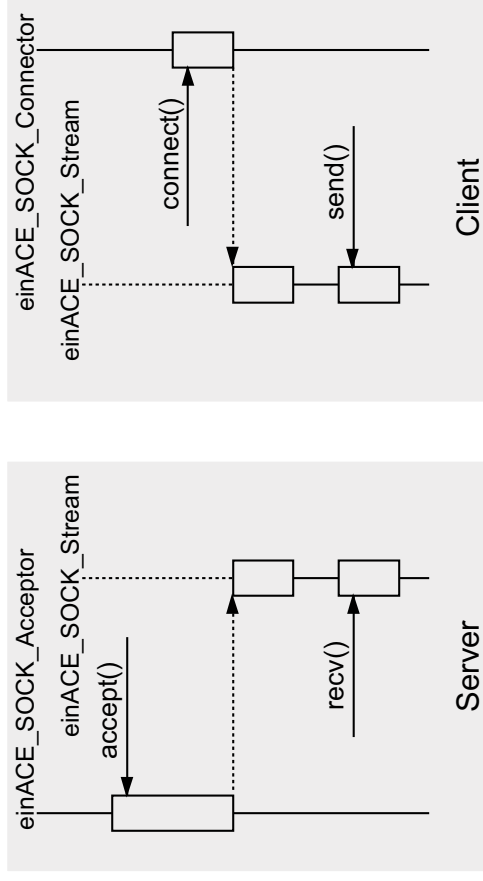
- **OS Adaptation Layer**
 - Abbildung Betriebssystem spezifischer APIs auf einheitliche (ACE) API
- **C++ Wrapper**
 - Reihe von typischeren C++ Klassenschnittstellen zu dem im OS Adaptation Layer definierten API
 - Beispiele:
 - ACE SOCK/TLI Service Access Point (SAP)
 - Kapselung des Socket API und des TLI (Transport Layer Interface)
 - ACE FIFO/SPIPE Service Access Point
 - Klassenschnittstelle zu den Named Pipe und Stream Pipe Konzepten
 - ACE Threads
- **Frameworks**
 - wieder verwendbare Software-Komponenten, die die Funktionalität der C++ Wrapper erweitern



ACE – Socket Service Access Point



ACE – Socket Service Access Point



ACE – Socket Service Access Point

Server	Client
ACE_INET_Addr addr(PORT);	ACE_INET_Addr server_addr(ADDR, PORT);
//socket, bind, listen	
ACE_SOCK_Acceptor acceptor(addr);	ACE_SOCK_Connector connector;
ACE_SOCK_Stream stream;	ACE_SOCK_Stream stream;
acceptor.accept(stream);	connector.connect(stream, server_addr);
stream.recv_n(msg, sizeof(msg));	stream.send_n(msg, sizeof(msg));



ACE – Vorteile & Nachteile

Vorteile

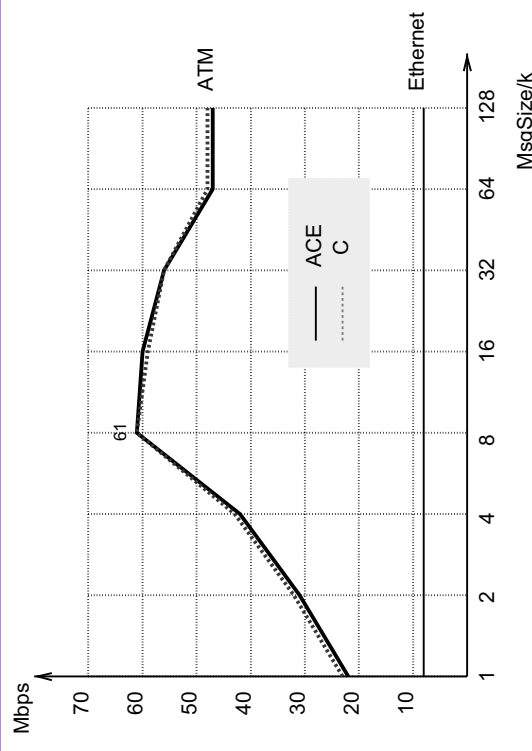
- Plattformunabhängigkeit
- Klassenkapselung → Typsicherheit
 - Fehlererkennung zur Übersetzungszeit
- Methoden der C++ Wrapper-Klassen fassen mehrere Systemaufrufe zu einem Aufruf zusammen
 - korrekte Aufrufreihenfolge
 - kürzerer und verständlicher Programmcode
- Verwendung von Konstruktoren und Default-Argumenten stellt korrekte Initialisierung sicher

Nachteile

- Ungenügende Dokumentation
- Kein allgemein verwendbares Interface zum fcntl() Systemaufruf



ACE – Performance Betrachtung



Sockets in Java

- Sockets in `java.net`
- Datagram-Sockets: `DatagramSocket`
 - `MulticastSocket` unterstützt IP Multicasting.
- TCP-Stream Sockets:
 - `ServerSocket` für wartende Sockets auf Serverseite.
 - `Socket` für Sockets zur Verbindung auf Server- und Clientseite.
- Verbindung erfolgt über `InputStream` und `OutputStream` Objekte auf Client- und Serverseite
 - Sockets sind *bidirektional*!



Vorteile der OO Kapselung

- Typsicherheit wird verbessert
 - Fehler werden bereits beim Übersetzen erkannt
 - Z.B. Falsche Verwendung von Deskriptoren
- Kombination mehrerer Systemaufrufe
 - Kompaktere Anwendungsprogramme
 - Vermeidung von Fehlern
- Korrekte Initialisierung durch Konstruktoren



Beispiel: Stream-Sockets in Java (Serverseite)

```
//hoere auf port 59000 auf lokalem host
ServerSocket s = new ServerSocket(59000);

while (true) {
    //warte auf connection request eines clients
    Socket clientConn = s.accept();

    InputStream in = clientConn.getInputStream();
    OutputStream out = clientConn.getOutputStream();
    // Datentransfer kann nun ueber in und out von
    // und zum Client erfolgen
    //...
}
```



Beispiel: Stream-Sockets in Java (Clientseite)

```
InetAddress serverAddr = InetAddress.getByName(
    "chaq.informatik.uni-tuebingen.de");

Socket s = new Socket(serverAddr, 59000);

InputStream in = s.getInputStream();
OutputStream out = s.getOutputStream();
// Datentransfer von und zum Server kann nun
// ebenfalls ueber in und out erfolgen
```



Datagram-Sockets in Java

- Schaffung einer Datagramm-Socket mit spezifischer Port-Nummer:


```
DatagramSocket udpSocket =
    new DatagramSocket(59000);
```
- Ohne Vorgabe eines spezifischen Ports:


```
DatagramSocket udpSocket =
    new DatagramSocket();
    int portNo = udpSocket.getLocalPort();
```
- Versenden von Datagrammen mittels Objekten vom Typ DatagramPacket.



Beispiel: Datagram-Sockets in Java

*// Nach Einrichten eines Sockets-Objekts udpSocket kann das
// Versenden von Datagrammen wie folgt geschehen.*

```
byte[] payload = {'h', 'a', 'l', 'l', 'o'};
```

```
InetAddress serverAddr = InetAddress.getByName(
    "chaq.informatik.uni-tuebingen.de");
```

```
DatagramPacket p = new DatagramPacket(payload,
    payload.length,
    serverAddr,
    59000);
```

```
udpSocket.send(p);
```

