2.4.1 Verfahren zur Interprozeßkommunikation

- Signale
- Pipes
- Gemeinsamer Speicher
- Nachrichten: Beispiel Socket-Schnittstelle

2.4.2 Übersicht über die Socket-Schnittstelle

1. Signale

Ziel: schnelle Reaktion auf bestimmte vordefinierte Ereignisse durch Schicken eines Signals
nicht: Datenaustausch

Beispiele für Anwendung:

- Prozeßkontrolle (Job control), z. B. Beenden eines (fehlerhaften) Programms (Ctrl-c)
- Fehlersuche (Debugging)
  `> man ptrace`

Beispiele für in 4.4BSD definierte Signale

Name	Default Action	Description
SIGKILL	terminate process	kill program
SIGINT	terminate process	interrupt program
SIGHUP	terminate process	terminal line hangup
SIGTSTP	stop process	stop signal generated from keyboa
SIGQUIT	create core image	quit program
SIGBUS	create core image	bus error
SIGFPE	create core image	floating-point exception
SIGSYS	create core image	non-existent system call invoked
SIGCONT	discard signal	continue after stop
SIGCHLD	discard signal	child status has changed

Eigenschaften:

- Signale unterbrechen die aktuelle Arbeit des Prozesses und erlauben daher eine extrem schnelle Reaktion auf ein Ereignis.
- Signale erlauben es nicht, direkt Daten auszutauschen.
- Signale sind auf das lokale System begrenzt.
- Prozesse mit gleicher effektiver UserID (wird vom Elternprozess geerbt) können sich Signale schicken.

  Ausnahme: Root-Prozesse können Signale an alle Prozesse schicken.
- Signale werden gemerkt, d.h. ein Signal erreicht auch einen schlafenden/blockierten Prozeß nach dem Aufwachen.

Beispiel: `>kill pid`
Empfängerprozess hat keine CPU-Zeit, wenn das Signal gesendet wird.
Wie erhält der Empfängerprozess trotzdem das Signal?

Realisierung:

- Signale werden vom Betriebssystem als **asynchrone Softwareunterbrechungen** realisiert.
- Jeder Prozeß besitzt eine **Bit-Map** für Signale:
  $\Longrightarrow$ Für ein gesendetes Signal wird das entsprechende Bit in der Bit-Map des Empfängers gesetzt.
- Bevor der Dispatcher einen Prozeß rechnend setzt, wird die Signal-Bitmap geprüft.

In welcher Datenstruktur des Betriebssystems wird die Bit-Map gespeichert? $\Longrightarrow$ Im Prozeßkontrollblock

Jedem Signal ist eine Signalbehandlungsroutine action zugeordnet:

1. Default action
2. benutzerdefinierte Signal-Behandlungsroutinen sind möglich: Benutzer kann einen signal handler installieren. Dieser fängt das Signal ab (catching a signal) und ruft die vom Benutzer spezifizierte Behandlungsroutine auf.
3. Ein Signal kann ignoriert werden.

Achtung: Nicht alle Signale können abgefangen bzw. ignoriert werden. Ausnahmen: SIGKILL, SIGSTOP

Weitere Infos:

- `> man sigaction`
- in der Übung

Unix-Philosopie

Doug McIlroy, Erfinder der Unix-Pipe:

"This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface."

Quelle: https://en.wikipedia.org/wiki/Unix_philosophy

- Unidirektionaler Datenfluß von einem Daten erzeugenden Prozeß zu einem Daten verarbeitenden Prozeß.
- Daten werden gepuffert in FIFO-Reihenfolge zugestellt.
- Keine Unterstützung zur Strukturierung von Daten. (Empfänger muß den Bytestrom oftmals wieder mit einem Scanner/Parser in Datenblöcke zerlegen.)
- Pipes werden beispielsweise zwischen Eltern/Kind-Prozessen eingesetzt.
- Sogenannte *named pipes* erlauben beliebigen Prozessen an einer Pipe teilzunehmen.
- Pipes sind auf das lokale System beschränkt.

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

DESCRIPTION

`pipe()` creates a pair of file descriptors, pointing to a pipe inode, and places them in the array pointed to by `filedes`.
`filedes[0]` is for reading, `filedes[1]` is for writing.

Beipiel: Verwaltung von I/O-Kanälen unter UNIX

Ein I/O - Kanal wird durch einen sogenannten Descriptor beschrieben.

Ein Descriptor wird als Integerwert realisiert und ist der Index für den Tabelleneintrag in der sogenannten Descriptor-Tabelle.

Die Descriptor-Tabelle enthält für jeden geöffneten I/O-Kanal einen Eintrag:

Datei:	File-Descriptor: Rückgabewert vom `open()`
Socket:	Socket-Descriptor: Rückgabewert vom `socket()`
Pipe:	2 Pipe-Descriptoren: im `pipe()` gesetzt

Wo wird die Descriptor-Tabelle gespeichert?
$\Longrightarrow$ im Prozeßkontrollblock

Die Tabelleneinträge der Descriptor-Tabelle bestehen aus:

Datei:	Verweis auf *inode*, aktuelle Position in der Datei, ...
Pipe:	aktuelle Position für Lesen bzw. Schreiben
Socket:	Speicherplatzadresse für Daten, Verwaltungsinfos wie z.B. benutztes Transportprotokoll (UDP, TCP), ...

Demo-Beispiel zur Pipe

If all file descriptors referring to the write end of a pipe have been closed, then an attempt to read(2) from the pipe will see end-of-file (read(2) will return 0).

If all file descriptors referring to the read end of a pipe have been closed, then a write(2) will cause a SIGPIPE signal to be generated for the calling process. If the calling process is ignoring this signal, then write(2) fails with the error EPIPE.

Quelle: >man 7 pipe

3. Gemeinsamer Speicher (Shared Memory)

- Mehrere Prozesse teilen sich einen gemeinsamen Speicherbereich.
  (UNIX: `shmget`, `shmat`, `shmctl`)
- Ermöglicht schnelle Kommunikation, da gemeinsame Daten direkt gelesen werden können.
- Programmieraufwand zur korrekten Synchronisation der Prozesse erforderlich.
- Verwaltung dynamischer, komplexer Datenstrukturen erzwingt häufig durch die erforderliche Synchronisation eine Serialisierung der beteiligten Prozesse.
- Gemeinsamer Speicher ist auf das lokale System begrenzt.
  Es existieren Vorschläge zur verteilten Realisierung von gemeinsamem Speicher (Distributed Shared Memory), wobei aber der Vorteil von Shared Memory (schnelle Kommunikation) verloren geht.

Der lokale Endpunkt einer (Netz-)Kommunikation wird Socket genannt (engl. Steckdose).

- Allgemeine Schnittstelle zur Interprozeßkommunikation, die **nicht auf lokale Prozesse** beschränkt ist.
- Entwickelt und zuerst implementiert im 4.2BSD UNIX (1984). Mittlerweile auf allen wichtigen Betriebssystemen verfügbar.
- In vielen Betriebssystemen als Teil des Betriebssystemkerns implementiert.
- Sockets realisieren einen abstrakten, protokollunabhängigen Zugriff auf die Dienste, die in der Regel von einem Transportprotokoll bereitgestellt werden.

Ziele und Eigenschaften:

- Transparenz: Die Kommunikation soll davon unabhängig sein, ob sich die Kommunikationspartner auf einem lokalen System befinden oder räumlich getrennt sind.
- Die Schnittstelle muß unabhängig von konkreten Protokollen für lokale Kommunikation bzw. Netzkommunikation sein. $\Longrightarrow$ Man unterscheidet Protokollfamilien (*Communication Domains*).
  Beispiel: Internet Domain, UNIX Domain
- Prozesse müssen in der Lage sein,
  andere Kommunikationsendpunkte (Empfänger) zu lokalisieren.
  $\Longrightarrow$ Socket-Adressen (spezifisch für die einzelnen Domänen)

Beispiele:

1. Internetanwendungen sind socket-basiert.
2. Message Passing Interface (MPI):
   MPI spezifiziert Funktionen zur Kommunikation zwischen Prozessen einer parallelen Anwendung.
   Die MPI Bibliothek wird auf Ethernet mittels der Socket API umgesetzt.

"Be liberal in what you accept, and
conservative in what you send"

Source: RFC 1122, section 1.2.2 [Braden, 1989a]

Overview

A seasoned network programmer appreciates the many complexities and pitfalls associated with meeting the requirements of robustness, scalability, performance, portability, and simplicity for an application that may be deployed in a heterogeneous environment and a wide range of network configurations. The TCP/IP programmer controls only the end-points of the network connection, but must provide for all contingencies, both predictable and unpredictable. Therefore, an extensive knowledge base is required. The TCP/IP programmer must understand the relationship among network API calls, protocol exchange, performance, system and network configuration, and security.

Matt Muggeridge: *Programming with TCP/IP*

2.4.2 Übersicht über die Socket-Schnittstelle

Erzeugen eines Kommunikationsendpunkts/Sockets:

int `socket(domain, type, protocol)`

- Die Protokollfamilie (domain) legt die erlaubten Protokolle und das Format der Adressen fest: `AF_UNIX` (lokal), `AF_INET` (Internetprotokolle), `AF_INET6`, `AF_ISO` (ISO-Protokolle), ...
- Der Sockettyp (type) legt den *Verbindungstyp* der Kommunikation fest: `SOCK_DGRAM`, `SOCK_STREAM`, `SOCK_RAW`, `...`

Typ des Sockets:

SOCK_DGRAM Ein Datagramm-Socket stellt einen Endpunkt für unstrukturierte unzuverlässige Kommunikation bereit (verbindungslos, Nachrichten besitzen maximale Größe). Die Kommunikation kann mit mehreren Partnern erfolgen.

SOCK_STREAM Endpunkt für einen geordneten Paketstrom (zuverlässig, verbindungsorientiert).
Die Anwendung kann auf den Socket zugreifen, als ob es sich um eine Datei (Stream) handelt.

SOCK_RAW Endpunkt für die Kommunikation ohne die lokale Beteiligung der Kommunikationsschicht (TCP/IP, UDP/IP), benötigt Root-Rechte. RAW Sockets erlauben das Erstellen eigener TCP- und UDP-Header (beispielsweise zum Testen).

...

Erzeugen eines Kommunikationsendpunkts/Sockets:

int `socket(domain, type, protocol)`

- Die Protokollfamilie (domain) legt die erlaubten Protokolle und das Format der Adressen fest: `AF_UNIX` (lokal), `AF_INET` (Internetprotokolle), `AF_INET6`, `AF_ISO` (ISO-Protokolle), ...
- Der Sockettyp (type) legt den *Verbindungstyp* der Kommunikation fest: `SOCK_DGRAM`, `SOCK_STREAM`, `SOCK_RAW`, `...`
- Der Parameter protocol definiert, welches von mehreren möglichen Protokollen benutzt werden soll. Allerdings ist bei gegebener Protokollfamilie und gegebenem Sockettyp meist nur ein Protokoll möglich.

protocol $= 0 \Longrightarrow$ Das Betriebssystem bestimmt das passende Kommunikationsprotokoll.

Socket-Adressen

- Prozesse müssen in der Lage sein, andere Kommunikationsendpunkte zu lokalisieren. $\Longrightarrow$ Socket-Adressen (spezifisch für die einzelnen Domänen)
- Adressen werden in einer allgemeinen Struktur angegeben, die aus einem Feld für die Protokollfamilie, einer Portnummer und einer für diese Protokollfamilie spezifischen Adresse besteht:

```
struct sockaddr_in {
    sa_family_t     sin_family; /* address family: AF_INET */
    u_int16_t       sin_port;   /* port in network byte order */
    struct in_addr  sin_addr;   /* AF_INET: internet address */
};
```

- Für die Internet-Protokollfamilie (`AF_INET`) besteht der spezifische Teil aus einer 32 Bit IPv4-Adresse und einer 16 Bit Portnummer.

```
/* Internet address. */
struct in_addr {
    u_int32_t   s_addr; /* address in network byte order */
};
```

Well-known port numbers für bestimmte Dienste:

```
# WELL KNOWN PORT NUMBERS
#


...
ftp-data      20/tcp    #File Transfer [Default Data]
ftp           21/tcp    #File Transfer [Control]
ssh           22/tcp    #Secure Shell
telnet        23/tcp    #
smtp          24/tcp    #Simple Mail Transfer
....
http          80/tcp    #Hypertext Transfer Protocol
....
SIP         5060/UDP    #Session Initiation Protocol (VoIP)
```

Erzeugen eines Kommunikationsendpunkts/Sockets:

int `socket(domain, type, protocol)`

socket() creates an endpoint for communication and ***returns a socket descriptor.***

$\Longrightarrow$ Das Socket hat noch keine Socket-Adresse zugewiesen bekommen!
$\Longrightarrow$ Man nennt das Zuweisen der Socket-Adresse (z.B. Internet-Adresse) an den lokalen Endpunkt Binden.

Übersicht über die Socket-Schnittstelle (1)

- `bind(socketfd, address, length)`: Bindet eine Socket-Adresse an den lokalen Endpunkt, den der `socketfd` referenziert.
- `connect(socketfd, address, length)`: Herstellen einer Verbindung zu einem zweiten Socket, der durch die Adresse bestimmt ist.
- `listen(socketfd, backlog)`: Anzeigen, daß eingehende Verbindungswünsche akzeptiert werden.
- `accept(socketfd, address, length)`: Annahme einer Verbindung.
- `write(...)`, `send(...)`, `sendto(...)`: Übertragen einer Bytefolge über einen Socket.
- `read(...)`, `recv (...)`, `recvfrom(...)`: Empfangen einer Bytefolge von einem Socket.
- `close(socket)`: Terminieren einer Verbindung und Freigabe des Kommunikationsendpunkts.

Socket-Aufrufe für verbindungsloses Protokoll

- Beim verbindungslosen Protokoll werden auf beiden Seiten Sockets geöffnet und an eine Adresse gebunden.
- Der `recvfrom()`-Aufruf **blockiert** den Prozeß, bis eine Nachricht eingetroffen ist. Die Adresse des sendenden Prozesses wird dem empfangenden Prozeß mitgeteilt.
- Der `sendto()`-Aufruf sendet die Daten an die angegebene Adresse.
- Beim Schließen eines Endpunkts ist keine Interaktion erforderlich.

Prozeß A	Prozeß B
`socket()`	`socket()`
`bind()`	`bind()`
`sendto()`	`recvfrom()`
`close()`	`close()`

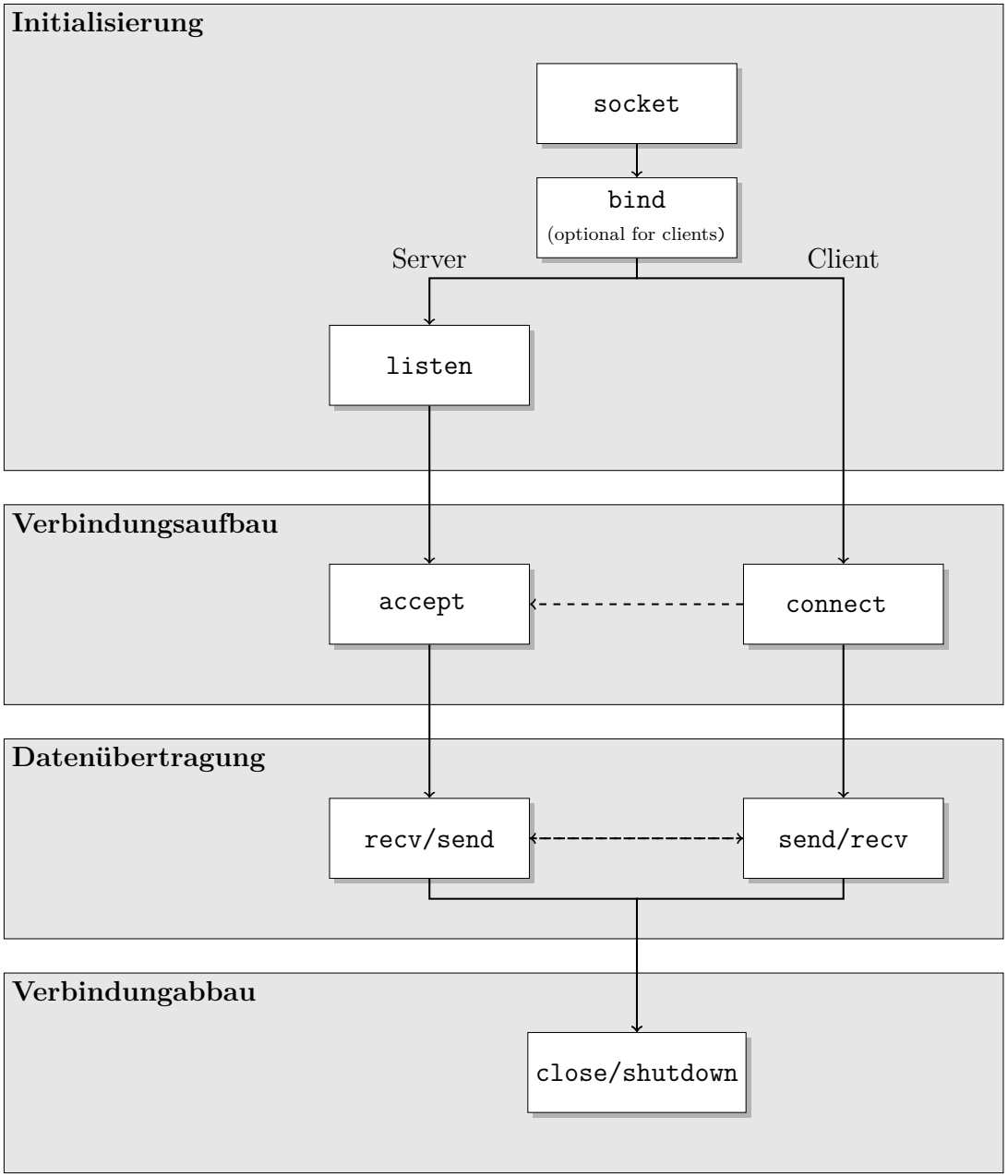
```
> man sendto

...
If space is not available at the sending socket to hold the
message to be transmitted and the socket file descriptor
does not have O_NONBLOCK set, sendto() shall block until
space is available.
If space is not available at the sending socket to hold the
message to be transmitted and the socket file descriptor
does have O_NONBLOCK set, sendto() shall fail.
...
```

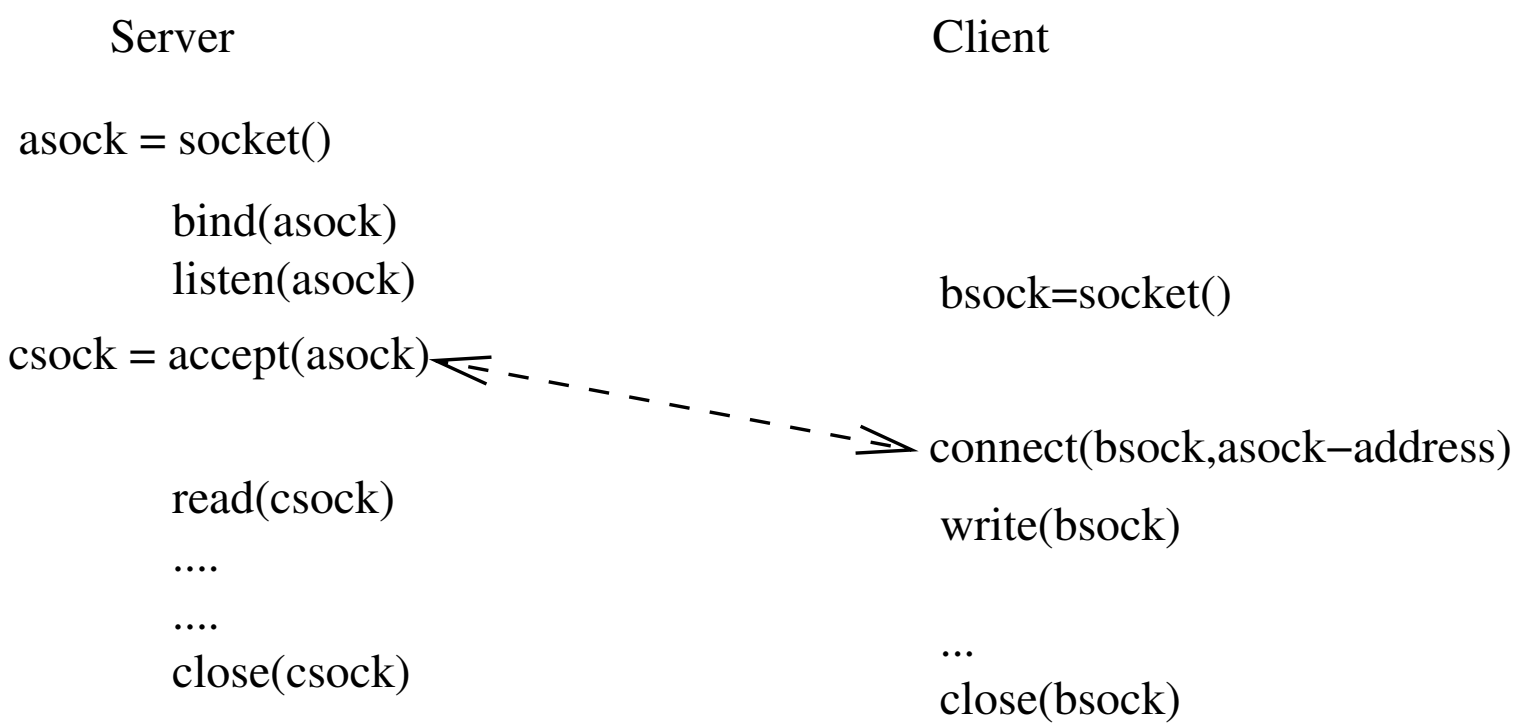
Socket-Aufrufe für verbindungsorientiertes Protokoll

- Beim verbindungsorientierten Protokoll wird zunächst von einer Seite ein Socket geöffnet, über den Verbindungswünsche entgegen genommen werden.
- Der `listen()`-Aufruf markiert das Socket als *passives* Socket, d.h. es wird nur dazu benutzt ankommende Nachrichten anzunehmen.
- Der `accept()`-Aufruf **blockiert** den Prozeß, bis eine Verbindung etabliert ist und liefert einen **neuen** Socket für diese Verbindung.
- Die `read()` und `write()` Aufrufe sind **blockierend**.
- Nach der Auflösung der Verbindung kann mit einem erneuten Aufruf von `accept()` eine weitere Verbindung entgegen genommen werden.

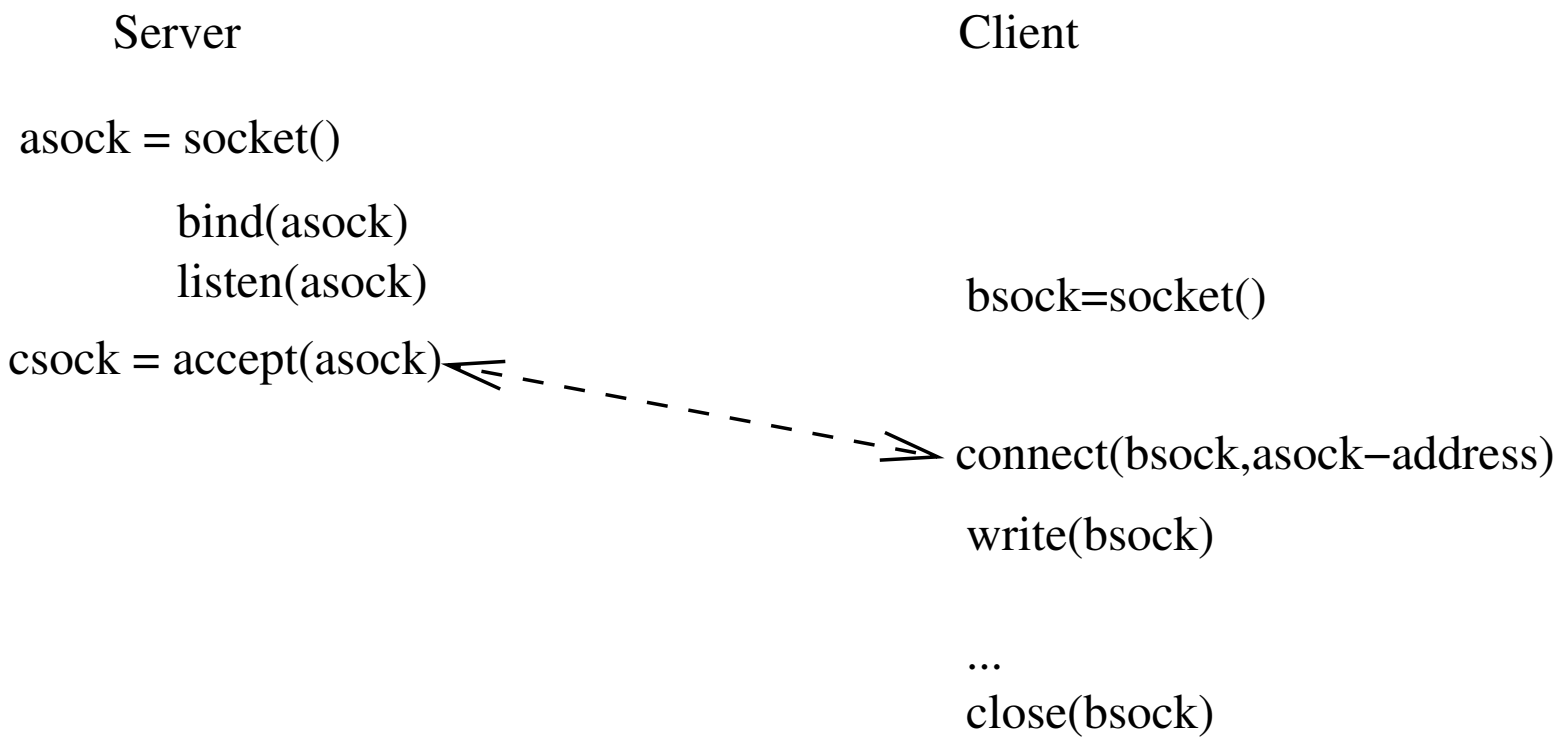
Ablauf der **verbindungsorientierten** Socket-Kommunikation



Beispiel: File Deskriptoren `asock`, `bsock` und `csock`



Beispiel: Der Server erzeugt für jede Verbindung einen eigenen Kindprozeß



Übersicht über die Socket-Schnittstelle (2)

- `shutdown(socket, how)`: (Teilweise) Beendigung einer Verbindung.
- `setsockopt()`, `getsockopt()`: Setzen und Lesen von Socket-Optionen.
- `getsockname()`, `getpeername()`: Erfragen der lokalen bzw. entfernten Adresse.
- Eine Reihe von Hilfsfunktionen zur Umwandlung von Namen in Adressen und zur Konvertierung von Zahlen in eine einheitliche Byteordnung, die sogenannte Network Byte Order.

  Eindeutige Darstellung von (unsigned) 16- bzw. 32-Bit Integer (short/long):
  `htons()`, `htonl()`: host to network order
  `ntohs()`, `ntohl()`: network to host order

Problem: Eindeutige Datendarstellung

Computers store the bytes that form a multibyte value in individual memory locations. A 16-bit integer (2 bytes), may be stored in two different ways:

Big Endian: (*as you may write*)

- The high-order byte is stored on address n
- The low-order byte is stored on address $n + 1$

Little Endian:

- The high-order byte is stored on address $n + 1$
- The low-order byte is stored on address n

Beispiel: Ein 16-Bit short Integer 1111111100000000 wird wie folgt gespeichert:

	Adresse n	Adresse $n + 1$
Big Endian	11111111	00000000
Little Endian	00000000	11111111

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

The sockfd argument is a file descriptor that refers to a socket of type SOCK_STREAM[17].

The backlog argument defines the maximum length to which the queue of pending connections for sockfd may grow.

Speicher ist eine beschränkte Ressource $\Longrightarrow$ Es gibt eine OS-spezifische Obergrenze.

- Die Länge der Backlog-Queue ist das Minimum aus dem angegebenen Backlog-Wert `backlog` und dem (OS-spezifischen) gesetzten Limit.
- Falls der Backlog-Wert ≤ 0 ist, wird die Länge der Backlog-Queue auf die OS-spezifische Obergrenze gesetzt.

[17] eher selten SOCK_SEQPACKET: connectionless reliable datagrams

Hinweis für Linux:

- The behavior of the backlog argument on TCP sockets changed with Linux 2.2. Now it specifies the queue length for completely established sockets waiting to be accepted, instead of the number of incomplete connection requests.
- The maximum length of the queue for incomplete sockets can be set using `/proc/sys/net/ipv4/tcp_max_syn_backlog`. When syncookies[18] are enabled there is no logical maximum length and this setting is ignored.

[18] Mechanismus, um DOS-Attacken zu vermeiden.

Auszug aus > `man getaddrinfo`:
`getaddrinfo` allows programs to eliminate IPv4-versus-IPv6 dependencies.

```
int getaddrinfo(const char *node, const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);
```

Beispiel: TCP Client, der dem Heise-Webserver die Nachricht `GET / HTTP/1.0` schickt.

Die Zieladresse wird durch `node` (hier: www.heise.de) und `service` (hier: Port 80) beschrieben. `getaddrinfo` übernimmt die Adressauflösung und liefert eine Liste (`**res`) aller möglichen Internetadressen (IPv4 und/oder IPv6) zurück. In `hints` können die gewünschte Domain, Socket-Typ und Protokoll spezifiziert werden.

Blockiertes Warten auf mehrere I/O-Kanäle

Beispiel: Ursprung im 4.2 BSD:

```
select(width, readfds, writefds, exceptfds, timeout)
```

Der `select`-Aufruf **blockiert**, bis einer der File-Descriptoren in `readfds`, `writefds` oder `exceptfds` *bereit* ist, oder bis das Timeout-Intervall abgelaufen ist.

```
On exit, the sets are modified in place to indicate
which file descriptors actually changed status.
Each of the three file descriptor sets may be specified
as NULL if no file descriptors are to be watched for the
corresponding class of events.
```

Four macros are provided to manipulate the file descriptor sets.

FD_ZERO() clears a set.

FD_SET() and FD_CLR() respectively add and remove a given file descriptor from a set.

FD_ISSET() tests to see if a file descriptor is part of the set; this is useful after select() returns.

Beispiel: System V unterstützte den Aufruf:

```
poll(fds, nfds, timeout)
```

Der `poll`-Aufruf blockiert, bis ein File-Deskriptor in `fds` bereit ist, oder das Timeout-Intervall abgelaufen ist.

Weitere Varianten:

1. FreeBSD: `kqueue`
2. Linux: `epoll`

"epoll is a new system call introduced in Linux 2.6. It is designed to replace the deprecated select (and also poll). Unlike these earlier system calls, which are O(n), epoll is an O(1) algorithm - this means that it scales well as the number of watched file descriptors increase. select uses a linear search through the list of watched file descriptors, which causes its O(n) behaviour, whereas epoll uses callbacks in the kernel file structure."

Quelle: Oleksky Kovyrin, `http://kovyrin.net/2006/04/13/epoll-asynchronous-network-programming/`

siehe auch: **libevent - an event notification library**[19]
https://libevent.org/ :

The libevent API provides a mechanism to execute a callback function when a specific event occurs on a file descriptor or after a timeout has been reached. Furthermore, libevent also support callbacks due to signals or regular timeouts.
libevent is meant to replace the event loop found in event driven network servers. An application just needs to call `event_dispatch()` *and then add or remove events dynamically without having to change the event loop.*
Currently, libevent supports `/dev/poll`, `kqueue(2)`, `event ports`, `POSIX select(2)`, `Windows select()`, `poll(2)`, `and epoll(4)`.

[19] Urprünglich von Niels Provos entwickelt, der auch das Sicherheitswerkzeug `honeyd` entworfen hat.

1506 Seiten Wissen: Michael Kerrisk[20]: The LINUX Programming Interface - A Linux and UNIX System Programming Handbook, No Starch Press.

[20] seit 2004 Maintainer of the Linux man-pages project, co-authored about a third of the man pages