Sockets in LINUX – Grundlagen (1)

- **Allgemeines**

  ◇ Linux stellt auch die **BSD Socket API** zur Verfügung.
  Diese kann als die heutige **Standard-Schnittstelle** für die **Programmierung** von **Netzwerkverbindungen** betrachtet werden. ⇒ Sockets ermöglichen eine **Interprozeßkommunukation** (IPC) **über Rechnergrenzen hinweg**.
  Sie lassen sich aber **auch** als **lokaler IPC-Mechanismus** innerhalb eines Rechners einsetzen.

  ◇ Die Socket API realisiert eine **einheitliche Schnittstelle** für **unterschiedliche Netzwerkprotokolle**. Die Schnittstelle ist so konzipiert, daß jederzeit neue Protokolle ohne Änderung der Schnittstelle hinzugefügt werden können.
  Netzwerkprotokolle werden in Abhängigkeit von ihren Kommunikationseigenschaften in verschiedene **Protokoll-Typen** unterteilt (z.B. Datagramm-Protokolle, Stream-Protokolle usw) und in **Protokollfamilien** (Kommunikations-Domänen, *communiction domains*) zusammengefaßt (z.B. TCP/IP-Protokoll-Familie, Novell IPX-Protokoll usw)

  ◇ Sockets ermöglichen eine **bidirektionale Kommunikation.**
  Ein **Socket** bildet einen **Kommunikations-Endpunkt** → Jeder der beiden an einer Kommunikation beteiligten Prozesse benötigt einen Socket.

  ◇ Ein Socket wird – wie eine Datei – über einen **File Deskriptor** referiert
  → für jeden Socket existiert eine **`struct file`**-Struktur.
  Damit läßt sich eine Socket-Kommunikation mittels der **Dateibearbeitungs-Funktionen** (System Calls) durchführen.

- **Realisierung von Sockets**

  ◇ Sockets werden durch **spezielle Datenstrukturen** und durch **spezielle Funktionen** realisiert.

  Im virtuellen Dateisystem (*Virtual File System*) werden sie durch **spezielle Inodes** (VFS-Inodes) repräsentiert.
  Diese besitzen eine **socket-spezifische Komponente** vom Typ **`struct socket`**

  ◇ Datenstruktur **`struct socket`** (definiert in *"linux/include/linux/net.h"*)

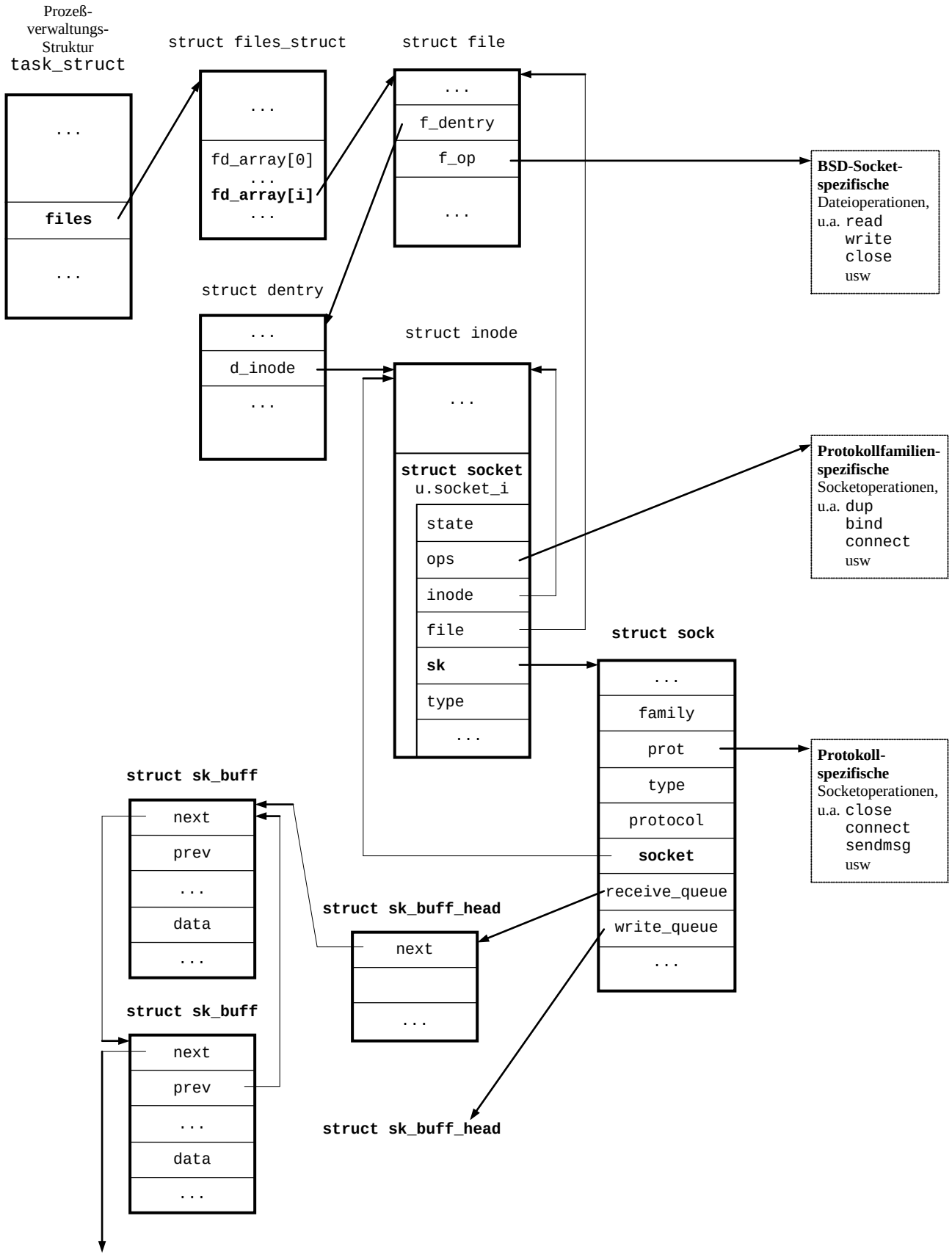
```
typedef enum {
  SS_FREE = 0,          /* not allocated          */
  SS_UNCONNECTED,       /* unconnected to any socket     */
  SS_CONNECTING,        /* in process of connecting      */
  SS_CONNECTED,         /* connected to socket           */
  SS_DISCONNECTING      /* in process of disconnecting   */
} socket_state;


struct socket
{
  socket_state          state;
  unsigned long         flags;
  struct proto_ops*     ops;
  struct inode*         inode;
  struct fasync_struct* fasync_list;  /* Asynchronous wake up list    */
  struct file*          file;         /* File back pointer for gc     */
  struct sock*          sk;
  struct wait_queue*    wait;
  short                 type;
  unsigned char         passcred;
  unsigned char         tli;
};
```

Bedeutung einiger Komponenten :

`ops`	Pointer auf Struktur mit Pointern auf Funktionen für protokollfamilienspezifische Socket-Operationen
`inode`	Pointer auf Beginn des Inodes zu dem die `struct socket` Struktur gehört
`sk`	Pointer auf eine Unterstruktur mit protokollspezifischen Informationen
`type`	Socket-Typ (entsprechend dem Protokoll-Typ)

Socketrelevante Verwaltungsstrukturen in LINUX

- **Überblick :**

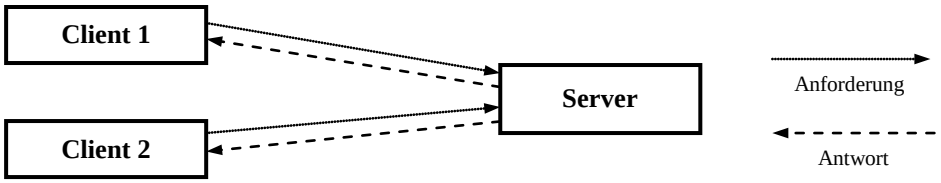


Sockets in LINUX – Grundlagen (2)

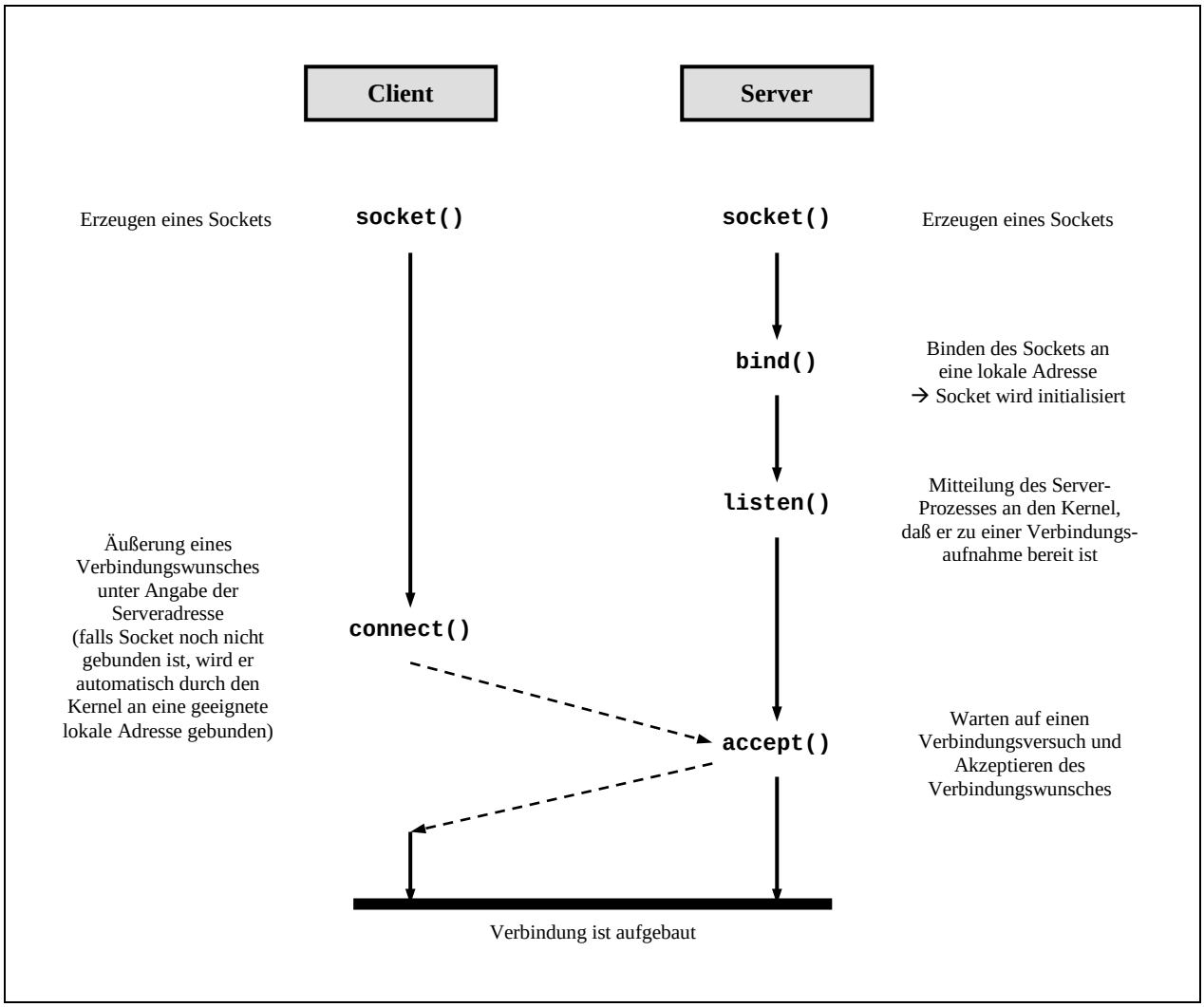
- **Komunikationsprinzip**

Die Kommunikation über Sockets erfolgt nach dem **Client-Server-Modell** :
Ein **Server-Prozeß** stellt **Dienstleistungen** über eine definierte Schnittstelle zur Verfügung, die von **Client-Prozessen** genutzt werden können.
Nach dem Aufbau einer Kommunikationsverbindung kann eine **bidirektionale Kommunikation** stattfinden.
Diese erfolgt i.a. **asymmetrisch** auf der Basis eines festgelegten **Protokolls** nach dem **Request-Response-Prinzip** :
Der **Client** richtet zu einem beliebigen Zeitpunkt, also **asynchron**, **Anforderungen** an den **Server**. Dieser **antwortet** i.a. innerhalb einer bestimmten Zeit, also **synchron**.

Ein Server kann häufig zu **mehreren Clients** eine Kommunikationsverbindung unterhalten.



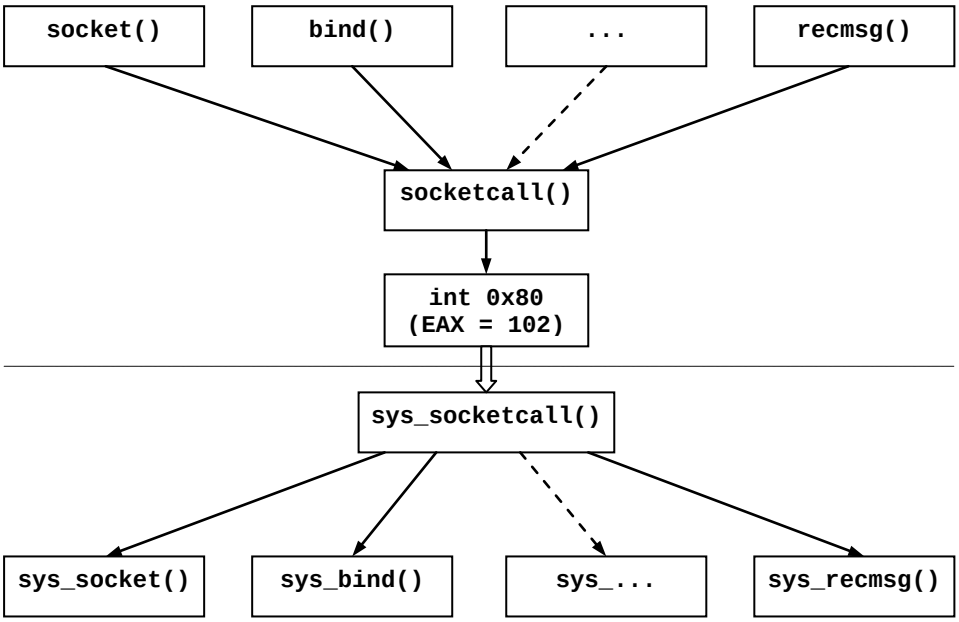
- **Ablauf eines Verbindungsaufbaus**



Socket-API in LINUX – Überblick (1)

- **Prinzipielle Realisierung**

Das **Socket-API** besteht aus einer **Reihe von Betriebssystemfunktionen**, die alle als **Unterfunktionen** des System Calls **`socketcall()`** (Funktions-Nr. **`102`**) realisiert sind.
Die einzelnen API-Funktionen sind durch eine **Unterfunktions-Nr**. gekennzeichnet, die im Register **`EBX`** übergeben wird.



- **Socket-API-Funktionen und Unterfunktions-Nummer**

Die Unterfunktions-Nummern für die Socket-API-Funktionen sind in `<linux/net.h>` definiert.

```
#define SYS_SOCKET        1       /* sys_socket(2)          */
#define SYS_BIND          2       /* sys_bind(2)            */
#define SYS_CONNECT       3       /* sys_connect(2)         */
#define SYS_LISTEN        4       /* sys_listen(2)          */
#define SYS_ACCEPT        5       /* sys_accept(2)          */
#define SYS_GETSOCKNAME   6       /* sys_getsockname(2)     */
#define SYS_GETPEERNAME   7       /* sys_getpeername(2)     */
#define SYS_SOCKETPAIR    8       /* sys_socketpair(2)      */
#define SYS_SEND          9       /* sys_send(2)            */
#define SYS_RECV          10      /* sys_recv(2)            */
#define SYS_SENDTO        11      /* sys_sendto(2)          */
#define SYS_RECVFROM      12      /* sys_recvfrom(2)        */
#define SYS_SHUTDOWN      13      /* sys_shutdown(2)        */
#define SYS_SETSOCKOPT    14      /* sys_setsockopt(2)      */
#define SYS_GETSOCKOPT    15      /* sys_getsockopt(2)      */
#define SYS_SENDMSG       16      /* sys_sendmsg(2)         */
#define SYS_RECVMSG       17      /* sys_recvmsg(2)         */
```

Socket-API in LINUX – Überblick (2)

- **Adreß-Familien**

Sockets werden an eine **Adresse** gebunden (→ Socket-Adresse). Diese Adresse referiert den jeweiligen Kommunikationsteilnehmer eindeutig (z.B. im Netzwerk : Netzwerk-Adresse).
Der genaue Aufbau einer derartigen Adresse hängt von der Protokoll-Familie, für den der Socket eingesetzt wird, ab.
→ Jeder **Protokoll-Familie** entpricht eine **Adreß-Familie**.

Protokoll-Familie und Adreß-Familie werden durch die gleiche ganzzahlige Konstante (Familien-Nr) referiert.
Ihnen sind **symbolische Namen** zugeordnet, die in der durch **`<sys/socket.h>`** eingebundenen Headerdatei `<bits/socket.h>` definiert sind.

Die **wichtigsten** von LINUX unterstützten **Protokoll- und Adreß-Familien** sind :

Familien-Nr	Protokoll-Familie	Adreß-Familie	"Kommunikations-Domäne"
1	**PF_UNIX, PF_LOCAL**	**AF_UNIX, AF_LOCAL**	UNIX-Domain-Sockets (lokale Kommunikation)
2	**PF_INET**	**AF_INET**	IPv4 Internet-Protokoll-Familie (TCP/IP, Version 4)
3	**PF_AX25**	**AF_AX25**	Amateur-Radio AX.25 Protokoll
4	**PF_IPX**	**AF_IPX**	Novell Internet-Protokolle (IPX)
5	**PF_APPLETALK**	**AF_APPLETALK**	Appletalk DDP
9	**PF_X25**	**AF_X25**	ITU-T X.25 / ISO-8208 Protokoll
10	**PF_INET6**	**AF_INET6**	IPv6 Internet-Protokoll-Familie (TCP/IP, Version 6)
16	**PF_NETLINK**	**AF_NETLINK**	"Kernel User Interface Device"
17	**PF_PACKET**	**AF_PACKET**	"Low Level Packet Interface"

- **Generischer Socket-Adreß-Typ**

Das Socket-API **abstrahiert** die unterschiedlichen Adressen durch einen **generischen Socket-Adreß-Typ**.
In diversen API-Funktionen wird ein Pointer auf diesen Typ als Parameter verwendet.
Beim Aufruf dieser Funktionen muß der Pointer auf den tatsächlich verwendeten protokollfamilien-spezifischen Adreßtyp in einen Pointer auf den generischen Adreßtyp gecastet werden.
Zusätzlich ist in einem gesonderten Parameter die Länge des tatsächlich verwendeten Adreßtyps zu übergeben.

Der generische Socket-Adreß-Typ **`struct sockaddr`** ist in der durch **`<sys/socket.h>`** eingebundenen Headerdatei `<bits/socket.h>` wie folgt definiert :

```
 struct sockaddr {
   unsigned short  sa_family;     /* Address family, AF_xxx */
   char            sa_data[14];   /* Address Data           */
};
```

LINUX Socket-API-Funktion `socket`

- **Funktionalität :** **Erzeugung** .eines **Sockets.**
  Rückgabe eines den Socket referierenden **File Deskriptors**.

- **Interface :**

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

  - **Header-Datei :** **`<sys/socket.h>`**

  - **Parameter :**

`domain` **Protokoll-Familie**.
Die wichtigsten zulässigen Werte sind :

`PF_UNIX, PF_LOCAL`	UNIX-Domain-Sockets (lokale Kommunikation)
`PF_INET`	IPv4 Internet-Protokoll-Familie (TCP/IP, Version 4)
`PF_INET_6`	IPv6 Internet-Protokoll-Familie (TCP/IP, Version 6)
`PF_IPX`	Novell Internet-Protokolle (IPX)
`PF_NETLINK`	"Kernel User Interface Device"
`PF_X25`	ITU-T X.25 / ISO-8208 Protokoll
`PF_AX25`	Amateur-Radio AX.25 Protokoll
`PF_APPLETALK`	Appletalk DDP
`PF_PACKET`	"Low Level Packet Interface"

`type` **Protokoll-Typ** (Socket-Typ)
Zulässig sind die folgenden Werte :

`SOCK_STREAM`	verbindungsorientiertes Protokoll, streambasiert, mit Einhaltung der Datenreihenfolge (Sequencing), mit Fehlerkontrolle
`SOCK_DGRAM`	verbindungsloses Protokoll, paketbasiert, ohne Sequencing und ohne Fehlerkontrolle
`SOCK_SEQPACKET`	verbindungsorientiertes Protokoll, paketbasiert, mit Sequencing und Fehlerkontrolle
`SOCK_RDM`	verbindungsorientiertes Protokoll, paketbasiert, ohne Sequencing, mit Fehlerkontrolle
`SOCK_RAW`	direkter Zugriff zur Netzwerkschicht (z.B. IP) ermöglicht die Implementierung neuer Transportschicht-protokolle im User-Bereich (nur für Prozesse mit `EUID==0`)

`protocol` zu verwendendes **Protokoll**.
Meist kann der Wert **`0`** übergeben werden → der Kernel wählt das **Standard-Protokoll** des angegebenen Typs der Protokoll-Familie aus.
(Normalerweise existiert innerhalb einer Protokoll-Familie nur **ein Protokoll** eines **bestimmten Typs**.)

  - **Rückgabewert :** - **File Deskriptor**, über den der Socket referiert wird, bei **Erfolg**
    - **`-1`** im **Fehlerfall**, `errno` wird entsprechend gesetzt

- **Implementierung :** mittels System Call Nr. **`102 (socketcall())`**
  → `sys_socketcall(...)` (in *`net/socket.c`*)
  → `sys_socket(...)` (in *`net/socket.c`*)

- **Anmerkung :** Der erzeugte Socket ist **nicht initialisiert**, d.h. an keine Resource (Adresse) gebunden.
  → Seine Verwendung (Lesen und Schreiben) ist noch nicht möglich.

LINUX Socket-API-Funktion `bind`

- **Funktionalität :** **Binden** .eines **Sockets** an eine lokale **Adresse.**
  ("Zuordnung eines Namens" zum Socket)

- **Interface :**

```
int bind(int sockfd, struct sockadr* addr, socklen_t addrlen);
```

  - **Header-Datei :** **`<sys/socket.h>`**

  - **Parameter :**
    - *`sockfd`* File-Deskriptor , der Socket referiert
    - *`addr`* Pointer auf lokale Adresse, an die der Socket gebunden werden soll., Art und Aufbau der Adresse sind abhängig von der Protokoll-Familie (Adreß-Familie) des Sockets. Der Pointer auf die tatsächliche Adresse muß in `struct sockaddr*` gecastet werden.
    - *`addrlen`* Länge der durch *`addr`* referierten Adresse in Bytes

  - **Rückgabewert :**
    - **`0`** bei **Erfolg**
    - **`-1`** im **Fehlerfall**, `errno` wird entsprechend gesetzt

- **Implementierung :** mittels System Call Nr. **`102 (socketcall())`**
  → `sys_socketcall(...)` (in *`net/socket.c`*)
  → `sys_bind(...)` (in *`net/socket.c`*)

- **Anmerkungen:**
  1. Der Datentyp **`socklen_t`** ist in der Headerdatei `<sys/socket.h>` definiert als **`unsigned int`**.
  2. Üblicherweise verzichtet der Client-Prozeß auf ein explizites Binden des Sockets an eine lokale Adresse.
     Stattdessen überläßt er es dem Kernel, den Socket – bei der Äußerung eines Verbindungswunsches - an eine geeignete Adresse zu binden.

LINUX Socket-API-Funktion `listen`

- **Funktionalität :** **Mitteilung** des (Server-)Prozesses an den Kernel, daß er für eine **Verbindungsaufnahme** über den Socket durch andere Prozesse (Client-Prozesse) **bereit** ist.

- **Interface :**

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

  - **Header-Datei :** **`<sys/socket.h>`**

  - **Parameter :** *`sockfd`* File-Deskriptor , der Socket referiert

    `backlog` Festlegung der maximal erlaubten Anzahl wartender Verbindungswünsche (*"pending connections"*).
    Verbindungswunsch :Aufruf von `connect()` durch einen Client-Prozeß.
    Stehen bereits *`backlog`* Verbindungswünsche an, schlagen weitere Aufrufe von `connect()` fehl.
    BSD-UNIX sieht **`5`** als maximale Größe von *`backlog`* vor
    → portable Programme sollten diesen Wert einhalten.

  - **Rückgabewert :** - **`0`** bei **Erfolg**
    - **`-1`** im **Fehlerfall**, `errno` wird entsprechend gesetzt

- **Implementierung :** mittels System Call Nr. **`102 (socketcall())`**
  → `sys_socketcall(...)` (in *`net/socket.c`*)
  → `sys_listen(...)` (in *`net/socket.c`*)

LINUX Socket-API-Funktion `accept`

- **Funktionalität :** **Akzeptieren** eines **Verbindungswunsches** zu einem Socket durch einen Serverprozeß
  Anwendung nur für verbindungsorientierte Protokolle.
  Die Funktion wandelt den nächsten anstehenden Verbindungswunsch (aus der gegebenenfalls vorhandenen Warteschlange) in eine akzeptierte Verbindung um.
  Hierfür wird ein **neuer Socket** erzeugt, über den eine sich anschließende Kommunikation abgewickelt werden kann. Der ursprüngliche Socket wird nicht verändert. Er steht für eine Kommunikation nicht zur Verfügung, sondern nur für das Warten auf – weitere – Verbindungswünsche.
  Der neu erzeugte Socket hat die gleichen Eigenschaften wie der ursprüngliche Socket.
  Rückgabe eines den neu erzeugten Socket referierenden **File-Deskriptors**.
  Steht kein Verbindungswunsch an, **blockiert** die Funktion solange bis einer auftritt. Es sei denn der Socket ist mit `fcntl()` als `NON_BLOCKING` markiert worden. In diesem Fall kehrt die Funktion sofort mit dem Fehler `EAGAIN` zurück.

- **Interface :**

```
int accept(int sockfd, struct sockadr* addr, socklen_t* paddrlen);
```

  - **Header-Datei :** **`<sys/socket.h>`**

  - **Parameter :**

    `sockfd` File-Deskriptor , der Socket referiert, zu dem der Verbindungswunsch ansteht.

    `addr` Pointer auf Adreß-Struktur, in die der Kernel die Socket-Adresse des Prozesses, der den Verbindungswunsch äußert (Client-Prozeß), ablegt
    Art und Aufbau dieser Struktur sind abhängig von der Protokoll-Familie (Adreß-Familie) des Sockets.
    Der Pointers auf die tatsächliche Adreß-Struktur muß gecastet werden in `struct sockaddr*`
    Wird der `NULL`-Pointer übergeben, erfolgt keine Eintragung durch den Kernel.

    `paddrlen` Pointer auf Variable, in die der Kernel die Adreßlänge des Prozesses, der den Verbindungswunsch geäußert hat, ablegt.
    Beim Aufruf sollte *`*paddrlen`* auf die Größe der durch *`addr`* referierten Adreß-Struktur gesetzt werden.

  - **Rückgabewert :**
    - **File Deskriptor**, über den der neu erzeugte Socket referiert wird, bei **Erfolg**
    - **`-1`** im **Fehlerfall**, `errno` wird entsprechend gesetzt

- **Implementierung :** mittels System Call Nr. **`102 (socketcall())`**
  → `sys_socketcall(...)` (in *`net/socket.c`*)
  → `sys_accept(...)` (in *`net/socket.c`*)

- **Anmerkungen :**
  1. Der Datentyp **`socklen_t`** ist in der Headerdatei `<sys/socket.h>` definiert als **`unsigned int`**.
  2. Die – bidirektional mögliche – Kommunikation über den neu erzeugten Socket-File-Deskriptor kann mittels der System Calls `read()` und `write()` abgewickelt werden. Nach Beendigung der Kommunikation ist der Socket-File-Deskriptor mittels `close()` zu schließen.

LINUX Socket-API-Funktion `connect`

- **Funktionalität :** **Initiierung** eines **Verbindungswunsches** zu einem Socket durch einen Client.
  Wenn der Verbindungswunsch durch den Serverprozeß akzeptiert wird, kehrt die Funktion erfolgreich zurück

- **Interface :**

```
int connect(int sockfd, struct sockadr* servaddr, socklen_t addrlen);
```

  - **Header-Datei :** **`<sys/socket.h>`**

  - **Parameter :**
    - *`sockfd`* File-Deskriptor , der den Socket des Cloient-Prozesses referiert
    - *`servaddr`* Pointer auf die Adresse, an die der Socket des Ziel-Serverprozesses (zu dem eine Verbindung aufgebaut werden soll) gebunden ist
      Art und Aufbau der Adresse sind abhängig von der Protokoll-Familie (Adreß-Familie) des Sockets.
      Der Pointer auf die tatsächliche Adresse muß in `struct sockaddr*` gecastet werden.
    - *`addrlen`* Länge der durch *`servaddr`* referierten Adresse in Bytes

  - **Rückgabewert :**
    - **`0`** bei **Erfolg**
    - **`-1`** im **Fehlerfall**, `errno` wird entsprechend gesetzt

- **Implementierung :** mittels System Call Nr. **`102 (socketcall())`**
  → `sys_socketcall(...)` (in *`net/socket.c`*)
  → `sys_connect(...)` (in *`net/socket.c`*)

- **Anmerkung :** Der Datentyp **`socklen_t`** ist in der Headerdatei `<sys/socket.h>` definiert als **`unsigned int`**.

LINUX Socket-API-Funktion `setsockopt`

- **Funktionalität :** **Setzen** von **Optionen** für Sockets

- **Interface :**

```
int setsockopt( int sock, int level, int option, const void* val,
                socklen_t* vallen);
```

  - **Header-Datei :** **`<sys/socket.h>`**

  - **Parameter :**

    sock — File-Deskriptor des Sockets

    level — Typ der Option
    Beispiele : - **`SOL_SOCKET`** generische Socket-Option (auf Socket-Ebene interpretiert) (definiert in `<asm/socket.h>`, - indirekt – eingebunden durch `<sys/socket.h>`)
    - Protokoll-Nummer des die Option interpretierenden Protokolls (Protokoll-Nummern sind u.a. in */etc/protocols* enthalten)

    option — zu setzende Option, Angabe durch symbolischen Namen (mögliche Werte sind definiert in der durch `<sys/socket.h>` - indirekt – eingebundenen Headerdatei `<asm/socket.h>`)
    Beispiel : - **`SO_REUSEADDR`** Adresse, an die Socket gebunden ist, kann in sehr kurzer Zeit wieder verwendet werden (sonst i.a. erst nach längerer Wartezeit, z.B. bei TCP-Ports 2 min)

    val — Pointer auf Speicherbereich (Variable), der den Wert der zu setzenden Option enthält.
    Die meisten generischen Socket-Optionen verwenden hierfür eine int-Variable (!=0 : Setzen der Option, ==0 : Rücksetzen der Option)

    vallen — Größe des Speicherbereichs (in Bytes), auf den *val* zeigt.

  - **Rückgabewert :** - **`0`** bei Erfolg
    - **`-1`** im Fehlerfall, `errno` wird entsprechend gesetzt

- **Implementierung :** mittels System Call Nr. **`102`** (**`socketcall()`**)
  → `sys_socketcall(...)` (in *net/socket.c*)
  → `sys_setsockopt(...)` (in *net/socket.c*)

- **Beispiel für Anwendung :**

  Bei INET-Socket-Kommunikation : Aufhebung der Beschränkung, daß das Serverprogramm sein lokales Port erst nach einer längeren Wartezeit erneut benutzen kann. → mehrere Clients können nacheinander innerhalb kurzer Zeit mit dem Server in Verbindung treten.

```
int fdSock;          /*File Deskriptor des Sockets */
...
int iOpt=1;
setsockopt(fdSock, SOL_SOCKET, SO_REUSEADDR, &iOpt, sizeof(iOpt));
```