

# System-Programmierung

## 9: Sockets

CC BY-SA 4.0, T. Amberg, FHNW  
(Soweit nicht anders vermerkt)

Slides: [tmb.gr/syspr-9](https://tmb.gr/syspr-9)

# Überblick

Heute geht's um *UNIX Domain* und *Internet Sockets*.

Welche Varianten der Datenübertragung es gibt.

Wie man die Socket Schnittstelle benutzt.

# Sockets

*Sockets* sind ein IPC Mechanismus um zwischen zwei Programmen Daten auszutauschen, die beide auf dem selben Host oder durch ein Netzwerk verbunden sind.

Die erste Implementierung des *Socket API* erschien 1983 mit 4.2BSD, deshalb auch "Berkeley Sockets".

Heute wird diese Schnittstelle für UNIX und Internet Sockets auf fast allen Betriebssystemen unterstützt.

# Socket Verwendung

In einem typischen *Client-Server* Szenario nutzen Programme bzw. Anwendungen Sockets wie folgt:

Beide, Client und Server, kreieren einen Socket.

Der Server bindet seinen Socket auf eine wohlbekannte Adresse, so dass der Client ihn findet.

Kommunikation erfolgt uni- oder bidirektional.

# Socket Domänen

Die *Domäne* (communication domain) eines Sockets bestimmt, wie eine Socket Adresse aussieht, und ob lokal oder über ein Netzwerk kommuniziert wird.

Heutige Betriebssysteme unterstützen mindestens die UNIX (*AF\_UNIX* bzw. *AF\_LOCAL*) Domäne auf dem Host, sowie die Domänen IPv4 (*AF\_INET*) und IPv6 (*AF\_INET6*) für *Internet Protocol* (IP) Netzwerke.

# Stream Sockets

*Stream Sockets (SOCK\_STREAM)* sind zuverlässige, bidirektionale, verbindungsorientierte Byte Streams.

*Zuverlässig:* Bytes kommen entweder genau so an wie gesendet, oder Sender erhält eine Fehler-Notifikation.

*Bidirektional:* Datenübertragung in beide Richtungen, wie zwei Pipes, aber über ein Netzwerk. Deshalb auch *verbindungsorientiert:* verbunden mit einem *Peer*.

# Datagram Sockets

*Datagram Sockets (SOCK\_DGRAM)* sind Message-basiert, verbindungslos und unzuverlässig.

*Verbindungslos* bedeutet, dass einzelne Messages verschickt werden, ohne dass eine Verbindung da ist.

*Unzuverlässig* heisst, Übertragung und Reihenfolge sind nicht garantiert, Mehrfachübertragung möglich.

# Socket System Calls

Der *socket()* System Call kreiert einen neuen Socket.

Mit *bind()* binden Server ein Socket an eine Adresse.

Mit *listen()* hört ein Server auf neue Verbindungen.

Mit *accept()* wird eine Verbindung angenommen.

Der *connect()* System Call erstellt eine Verbindung.



# Socket kreieren mit *socket()*

Socket kreieren mit Domäne *domain* und Typ *type*:

```
int socket( // liefert einen File Deskriptor
    int domain, // AF_UNIX oder AF_INET, AF_INET6
    int type, // SOCK_STREAM oder SOCK_DGRAM
    int protocol); // immer 0 für diese Typen
```

Im Fehlerfall liefert *socket()* *-1* und setzt *errno*.

# Socket an Adresse binden mit *bind()*

Socket *sock\_fd* an die Adresse *sock\_addr* binden:

```
int bind( // 0 bei Erfolg, sonst -1 und errno  
         int sock_fd, // von socket() erstellt  
         const struct sockaddr *sock_addr,  
         socklen_t sock_addr_len);
```

Die Adresse hat je nach Domain einen anderen Typ,  
UNIX Domain Sockets verwenden einen Pfadnamen,  
Internet Sockets eine IP Adresse und einen Port.

# Socket Adressen

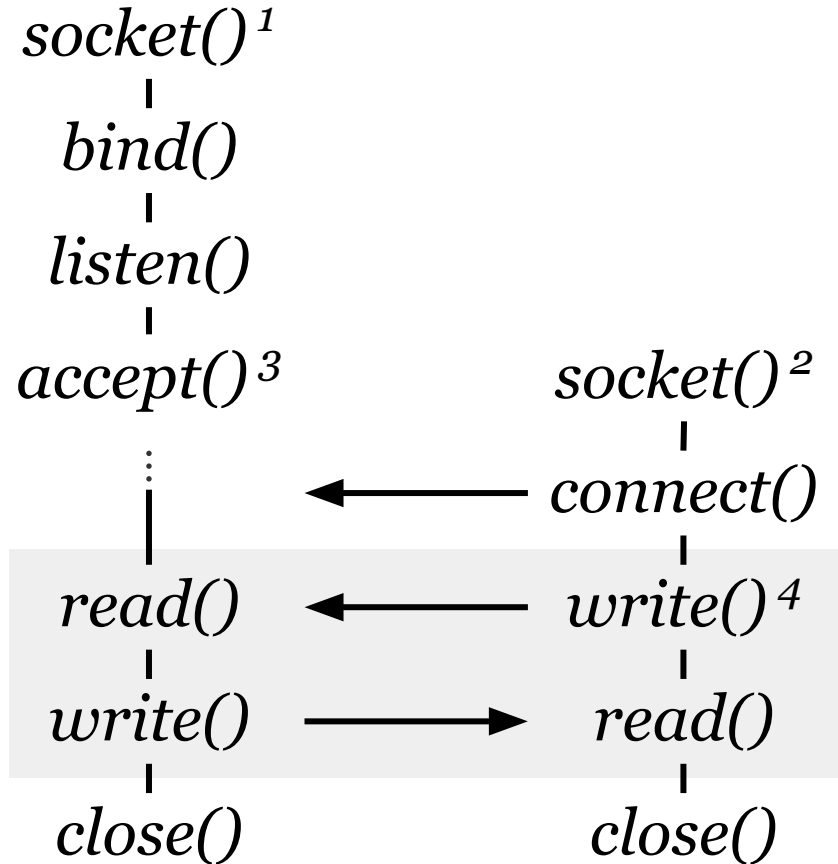
Der Struct *sockaddr* ist ein generischer Platzhalter:

```
struct sockaddr {  
    sa_family_t sa_family; // AF_ Konstante  
    char sa_data[14]; // Länge variiert  
}
```

Der *sa\_family* Wert genügt, um *sa\_data* zu parsen.

Der *sockaddr* Typ wird nur für Type-casts benutzt.

# Stream Sockets Ablauf



<sup>1</sup> Server, passiver Socket.

<sup>2</sup> Client, aktiver Socket.

<sup>3</sup> Accept blockiert, bis zum Connect.

<sup>4</sup> Write kann von beiden Seiten initiiert werden, auch mehrfach.

# Auf Connections hören mit *listen()*

Auf eingehende Connections hören mit *listen()*:

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

Muss vor *accept()* und *connect()* aufgerufen werden.

Der *backlog* Parameter bestimmt die Anzahl *pending* Connections, die von *accept()* angenommen werden.

Im Fehlerfall liefert *listen()* *-1* und setzt *errno*.

# Connections annehmen mit *accept()*

Eingehende Connections annehmen mit *accept()*:

```
int accept( // remote Socket fd, od. -1, errno  
    int sock_fd, // lokaler Socket File Deskr.  
    struct sockaddr *addr, // remote Adresse  
    socklen_t *addr_len); // Struct Grösse
```

Kreiert einen neuen Socket, der mit dem remote Peer / Client verbunden ist, der *connect()* aufgerufen hat.

Der Server Socket *sock\_fd* wird weiter verwendet. 14

# Socket verbinden mit *connect()*

*connect()* verbindet zu einem Server bzw. Peer Socket:

```
int connect( // 0 bei Erfolg, oder -1, errno  
            int sock_fd, // lokaler Socket File Deskript.  
            const struct sockaddr *addr, // remote Adr.  
            socklen_t addr_len); // Struct Grösse
```

Falls *connect()* einen Fehler liefert, Socket schliessen mit *close()* und neuen Socket kreieren mit *socket()*.

# Lesen und Schreiben mit *read()/write()*

Sockets sind bidirektional, beide Seiten können mit *read()/write()* oder *send()/recv()* lesen/schreiben.

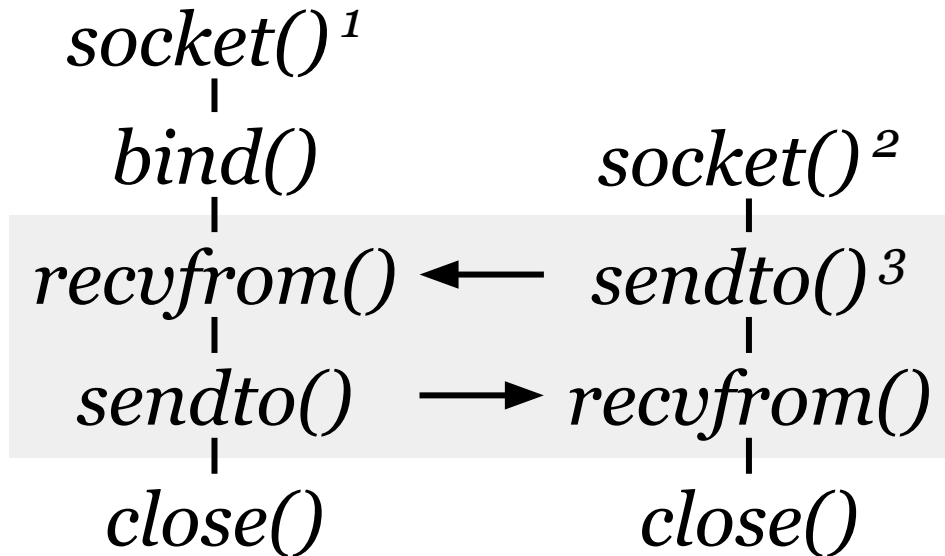
Das Verhalten ist vergleichbar mit dem von Pipes, falls ein Ende geschlossen wird, kommt am anderen *EOF* raus bei *read()*, bzw. *EPIPE* bei *write()*, wenn zuvor das *SIGPIPE* Signal ignoriert worden ist.

Mit *close()* schliesst man eine Verbindung.



# Datagram Sockets Ablauf

Bei Datagram Sockets entfällt *listen()* und *accept()*, sowie *connect()*, da diese verbindungslos sind.



<sup>1</sup> Server, passiver Socket.

<sup>2</sup> Client, aktiver Socket.

<sup>3</sup> Auch mehrfach und in beide Richtungen, weil *recvfrom()* die Adresse des Absenders liefert.

# Datagram empfangen mit *recvfrom()*

Datagram empfangen, blockierend, mit *recvfrom()*:

```
ssize_t recvfrom( // Resultat wie bei read()
    int socket_fd, // Socket FD wie bei read()
    void *buf, // wie bei read()
    size_t buf_len, // wie bei read()
    int flags, // 0, oder Socket spezifisch
    struct sockaddr *source_addr,
    socklen_t *source_addr_len);
```

# Datagram senden mit *sendto()*

Datagram senden an *dest\_addr* mit *sendto()*:

```
ssize_t sendto( // Resultat wie bei write()
    int sock_fd, // Socket FD wie bei write()
    const void *buf, // wie bei write()
    size_t buf_len, // wie bei write(), 0 ist OK
    int flags, // 0, oder Socket spezifisch
    const struct sockaddr *dest_addr,
    socklen_t dest_addr_len);
```

# UNIX Domain Sockets

UNIX Domain Sockets erlauben die Kommunikation zwischen zwei Prozessen auf demselben Hostsystem.

UNIX Domain Sockets nutzen File-Pfade als Adresse.

Der Zugriff darauf ist über File Permissions geregelt.

Es gibt sowohl Stream als auch Datagram Sockets.

# UNIX Domain Datagram Sockets

UNIX Domain Datagram Sockets übertragen Daten-Pakete zuverlässig, sequentiell und ohne Duplikate, im Gegensatz zu *Internet Domain* Datagram Sockets.

Pakete die grösser sind, als der bei *recvfrom()* mitgegebene Buffer werden abgeschnitten empfangen.

# UNIX Domain Socket Permissions

File Permissions bestimmen, wer Lese- oder Schreib-Zugriff auf UNIX Domain Sockets bekommen kann.

*bind()* erzeugt einen Socket Eintrag im File-System, inklusive Permissions für *owner*, *group* und *other*.

Für *connect()* und *sendto()* ist Schreibzugriff nötig, zudem braucht es *execute* (Such-) Rechte.

# UNIX Domain Socket Adressen

Struct für Socket Adresse in der UNIX Domain:

```
struct sockaddr_un {  
    sa_family_t sun_family; // Immer AF_UNIX  
    char sun_path[108]; // Null-terminierter  
};                        // Socket File-Pfad
```

Die max. Länge von *sun\_path* ist Plattform-abhängig.

Deshalb beim Zuweisen *strncpy()* verwenden.

# UNIX Domain Socket binden mit *bind()*

Socket *sock\_fd* an die Adresse *addr* binden:

```
struct sockaddr_un addr;  
memset(&addr, 0, sizeof(struct sockaddr_un));  
addr.sun_family = AF_UNIX;  
strncpy(addr.sun_path, "/tmp/mysock",  
        sizeof(addr.sun_path) - 1);  
int sock_fd = socket(AF_UNIX, SOCK_STREAM, 0);  
bind(sock_fd, (struct sockaddr *) &addr,  
      sizeof(struct sockaddr_un));
```



# UNIX Domain Socket *bind()* Details

Der File-Pfad *addr.sun\_path* muss schreibbar sein.

UNIX Domain Sockets sind im RAM, nicht auf Disk.

Bestehenden Pfad erneut binden gibt *EADDRINUSE*.

File *open()* funktioniert nicht auf Socket File-Pfad.

Unbenutzte Sockets mit *remove()* entfernen.

# Hands-on, 15': UNIX Domain Sockets

Analysieren Sie diese Socket Beispiele bestehend aus:

Header `us_xfr.h`<sup>TLPI</sup>,      Header `ud_ucase.h`<sup>TLPI</sup>,  
Server `us_xfr_sv.c`<sup>TLPI</sup>,      Server `ud_ucase_sv.c`<sup>TLPI</sup>,  
Client `us_xfr_cl.c`<sup>TLPI</sup>.      Client `ud_ucase_cl.c`<sup>TLPI</sup>.

Builden Sie die Programme, und lassen Sie sie laufen.

Zeichnen Sie **Sequenzdiagramme** mit User, Client, Server, das den Ablauf / übertragene Daten zeigt.

# Socket Paar kreieren mit *socketpair()*

Unbenanntes Socket Paar kreieren mit *socketpair()*:

```
int socketpair( // 0 oder -1, errno
               int domain, // nur für UNIX Domain AF_UNIX
               int type, // SOCK_DGRAM oder SOCK_STREAM
               int protocol, // 0
               int sock_fd[2]); // zwei verbundene Sockets
```

Typischerweise gefolgt von *fork()*, wie bei *pipe()*.

Kein File-Pfad => "unsichtbar", bessere Security.

# Internet Domain Sockets

Internet Domain *Stream Sockets* basieren auf dem *TCP* Protokoll. Sie bieten zuverlässige, bidirektionale Kommunikation mit Byte Stream Semantik.

Internet Domain *Datagram Sockets* basieren auf dem *UDP* Protokoll. Im Unterschied zu der UNIX Variante sind UDP Sockets nicht zuverlässig, garantieren keine Ordnung, es gibt Duplikate und "dropped packets".

# Netzwerk Byte Reihenfolge

Die *Network Byte Order* ist eine Konvention wie man Integer Werte in Bytes zerlegt und zwar "Big Endian".

Bei *Big Endian* schreibt man das *MSB* vor dem *LSB*:



Library Funktionen die IP Adressen ausgeben, liefern Resultate immer in Network Byte Order. Konstanten wie *INADDR\_ANY* müssen konvertiert werden.

# Byte Reihenfolge konvertieren

Konvertieren von Netzwerk zu Host Byte Order:

```
uint32_t ntohs(uint32_t netlong);  
uint16_t ntohs(uint16_t netshort);
```

Konvertieren von Host zu Netzwerk Byte Order:

```
uint32_t htonl(uint32_t hostlong);  
uint16_t htons(uint16_t hostshort);
```

Die Host Byte Reihenfolge kann je nach Hardware Plattform entweder Big oder Little Endian sein.

# Repräsentation von Daten

Nicht nur bei Adressen, auch bei allen anderen via ein Netzwerk gesendeten Daten ist das *Encoding* wichtig.

Bei TCP und UDP legt das die Anwendungsebene fest.

*HTTP* fordert z.B. *US-ASCII* für den Message Header, und via *Content-Type* beliebige Content Encodings.

Content vom Typ *application/json* würde z.B. gemäss JSON Standard mit UTF-8 Encoding übertragen.

# IPv4 Internet Socket Adressen

IPv4 Internet Socket Adresse, z.B. 192.168.0.42

```
struct in_addr {  
    uint32_t s_addr; // Network Byte Order  
};  
  
struct sockaddr_in {  
    sa_family_t sin_family; // AF_INET  
    in_port_t sin_port; // Network Byte Order  
    struct in_addr sin_addr; // Internet Adresse  
};
```



# IPv6 Internet Socket Adressen

```
struct in6_addr {  
    unsigned char s6_addr[16]; // IPv6 address  
};
```

```
struct sockaddr_in6 {  
    sa_family_t sin6_family; // AF_INET6  
    in_port_t sin6_port; // Port Nummer  
    uint32_t sin6_flowinfo; // IPv6 Flow Info  
    struct in6_addr sin6_addr; // IPv6 Adresse  
    uint32_t sin6_scope_id; // Scope ID  
};
```

# Loopback und Wildcard Adressen

IPv4 Loopback 127.0.0.1 und Wildcard 0.0.0.0 Adr.:  
INADDR\_LOOPBACK, INADDR\_ANY

IPv6 Loopback (::1) und Wildcard (::) Adresse:  
in6addr\_loopback bzw. IN6ADDR\_LOOPBACK\_INIT,  
in6addr\_any bzw. IN6ADDR\_ANY\_INIT

# Internet Socket Adressen Konvertieren

Von Punkt-Notation zu Binärformat konvertieren:

```
int inet_pton( // Erfolg: 1, Fehler: 0 od. -1
    int addr_family, // AF_INET, AF_INET6
    const char *src, // IP Adr. in Punkt-Notation
    void *dst); // IP Adresse im Binärformat
```

Von Binärformat zu Punkt-Notation konvertieren:

```
const char *inet_ntop( // dst od. NULL, errno
    int addr_family, // AF_INET, AF_INET6
    const void *src, // IP Adresse im Binärformat
    char *dst, socklen_t size); // IP String 35
```

# Host Lookup mit *getaddrinfo()*

Lookup von *host* und *service* (mit *hints*) liefert *result*:

```
int getaddrinfo( // 0 bei Erfolg, sonst != 0
    const char *host, // Hostname od. IP Adresse
    const char *service, // Name od. Port Nummer
    const struct addrinfo *hints, // Bsp. unten
    struct addrinfo **result); // Liste, ai_next
```

Nach Gebrauch, *addrinfo* Struct *result* freigeben:

```
void freeaddrinfo(struct addrinfo *result)
```

# Struct *addrinfo*

```
struct addrinfo { // hint* u. result, Rest = 0
    int ai_flags*; // Siehe Doku für AI_... Flags
    int ai_family*; // AF_UNSPEC, AF_INET(6)
    int ai_socktype*; // SOCK_STREAM, SOCK_DGRAM
    int ai_protocol*; // 0
    socklen_t ai_addrlen; // IP Adress-Länge
    struct sockaddr *ai_addr; // IP Adress-Struct
    char *ai_canonname; // Kanonischer Name
    struct addrinfo *ai_next; // "next" od. NULL
};
```

# Hands-on, 15': Internet Domain Sockets

Analysieren Sie dieses Socket Beispiel bestehend aus:

Header `i6d_ucose.h`<sup>TLPI</sup>,

Server `i6d_ucose_sv.c`<sup>TLPI</sup>,

Client `i6d_ucose_cl.c`<sup>TLPI</sup>.

Builden Sie die Programme, und lassen Sie sie laufen:

```
$ ./i6d_ucose_sv &
```

```
$ ./i6d_ucose_cl ::1 hello
```

# Challenge: Web Client und Server

Lösen Sie die beiden folgenden Hands-on Aufgaben.

Eventuell hilft dabei auch die HTTP Spezifikation

<https://tools.ietf.org/html/rfc2616>, Kapitel 4-7.

# Hands-on: Web Client

http\_client.!c

Schreiben Sie einen Web Client *my\_http\_client.c*, der folgenden HTTP Request an den Host *tmb.gr*, Port 80 sendet, die Antwort liest, und auf *stdout* ausgibt:

```
"GET /syspr HTTP/1.1\r\n"
```

```
"Host: tmb.gr\r\n"
```

```
"\r\n"
```

Hinweis: HTTP nutzt TCP als Transport-Protokoll.  
Länge der Antwort ist im *Content-Length* Header.



# Hands-on: Web Server

`http_server.!c`

Schreiben Sie einen Web Server *my\_http\_server.c*, der einkommende HTTP Requests auf Port 8080 liest und folgende Antwort zum Client / Browser sendet:

```
"HTTP/1.1 200 OK\r\n"
```

```
"Connection: close\r\n"
```

```
"Content-Length: 5\r\n"
```

```
"\r\n"
```

```
"hello"
```

# Feedback oder Fragen?

Gerne in Teams, oder per Email an

[thomas.amberg@fhnw.ch](mailto:thomas.amberg@fhnw.ch)

Danke für Ihre Zeit.