System-Programmierung 9: Sockets

n|w

Ablauf heute

⅔ Vorlesung,

⅓ Hands-on,

Feedback.

Slides, Code & Hands-on: tmb.gr/syspr-9

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 +/- allen Betriebssystemen unterstützt.

n|w

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 wohl-bekannte Adresse, so dass der Client ihn findet.

Kommunikation erfolgt uni- oder bidirektional.

n|w

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.

n|w

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*.

n|w

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.

n|w

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 mit einem anderen Socket. (*Linux: Library Calls.)

n|w

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*.

n|w

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.

n|w

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.

n|w

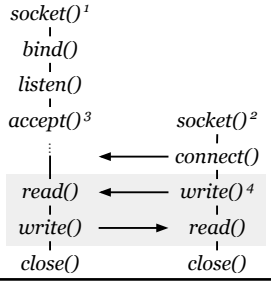
Stream Sockets "Telefon" Analogie

socket() "installiert einen Telefonanschluss",
bind() "löst eine Nummer", macht adressierbar,
listen() "schaltet das Telefon ein", macht anrufbar.

connect() "ruft eine Nummer an", bzw. Adresse,
accept() "nimmt einen eingehenden Anruf an",
send() & *recv()* ist "reden & zuhören", bidirektional,
close() "beide hängen auf am Ende des Anrufs".

n|w

Stream Sockets Ablauf



[1] Server, passiver Socket.
[2] Client, aktiver Socket.
[3] Accept blockiert, bis zum Connect.
[4] Write kann von beiden Seiten initiiert werden, auch mehrfach.

n|w

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*.

n|w

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. n|w

Socket verbinden mit *connect()*

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

```
int connect( // remote Socket fd, od. -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()*.

n|w

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.

n|w

Datagram Socket "Paketpost" Analogie

socket() "installiert einen Briefkasten",
bind() "weist dem Briefkasten eine Adresse zu".

sendto() "schickt ein Paket an einen Empfänger",
recvfrom() "wartet auf ein Paket, sieht Absender".

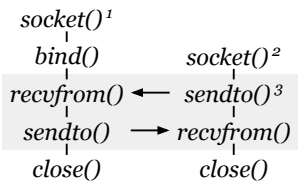
Die Pakete kommen in beliebiger Reihenfolge an.

close() "gibt den Briefkasten wieder frei".

n|w

Datagram Sockets Ablauf

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



[1] Server, passiver Socket.
[2] Client, aktiver Socket.
[3] Auch mehrfach und in beide Richtungen, weil *recvfrom()* die Adresse des Absenders liefert.

n|w

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 *restrict buf, // wie bei read()
  size_t buf_len, // wie bei read()
  int flags, // 0, oder Socket spezifisch
  struct sockaddr *restrict source_addr,
  socklen_t *restrict source_addr_len);
```

n|w

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);
```

n|w

UNIX Domain Sockets

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

Es gibt es sowohl Stream als auch Datagram Sockets.

Der Zugriff darauf ist über File Permissions geregelt.

socketpair() kreiert ein UNIX Domain Socket Paar.

Linux bietet einen abstrakten Socket Namespace.

n|w

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.

n|w

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));
```

n|w

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*.

Sockets werden an genau einen File-Pfad gebunden.

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

Unbenutzte Sockets mit *remove()* entfernen.

n|w

Hands-on, 30': UNIX Domain Sockets

Analysieren Sie diese Socket Beispiele bestehend aus:

Header us_xfr.h[TLPI],	Header ud_ucase.h[TLPI],
Server us_xfr_sv.c[TLPI],	Server ud_ucase_sv.c[TLPI],
Client us_xfr_cl.c[TLPI].	Client ud_ucase_cl.c[TLPI].

Builden Sie die Programme, und lassen Sie sie laufen.

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

n|w

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, mit allen Permissions für *owner*, *group* und *other*.

Für *connect()* und *sendto()* ist Schreibzugriff nötig, zudem braucht es *execute* (Such-) Rechte in jedem Directory des Socket File-Pfads.

n|w

UNIX Domain Datagram Sockets

UNIX Domain Datagram Sockets nutzen wie Stream Sockets File-Pfade als Adresse, z.B. */tmp/mysocket*.

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.

n|w

UNIX Domain Datagram Paket-Grösse

Die maximale Grösse hängt von der Konfiguration ab.

Der Wert kann über die Socket Option *SO_SNDBUF* mit *set-* / *getsockopt()* gesetzt bzw. abgefragt werden.

Beim Setzen eines Werts wird dieser verdoppelt (!), um Platz für die interne "Buchhaltung" zu schaffen:

```
setsockopt(fd, …, SO_SNDBUF, …, n);
getsockopt(fd, …, SO_SNDBUF, …, &m); // 2*n
```

n|w

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.

n|w

Linux Abstract Socket Namespace

Der *abstract* Namespace ist ein Linux-spezifisches Feature um UNIX Domain Sockets an einen Namen zu binden, der nicht im File-System kreiert wird.

Ohne Pfadname keine Kollisionen, kein *remove()*.

Um einen Namen im *abstract* Namespace zu kreieren, setzt man in *addr.sun_path* den ersten *char* auf '\0'.
(Kein gutes API Design, ging wohl nicht anders.)

n|w

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".

n|w

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*:

addr:	3 (MSB)	2	1	0 (LSB)

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

n|w

Byte Reihenfolge konvertieren

Konvertieren von Netzwerk zu Host Byte Order:

```
uint32_t ntohl(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.

n|w

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.

n|w

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
};
```

n|w

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
```


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_ucase.h[TLPI],
Server i6d_ucase_sv.c[TLPI],
Client i6d_ucase_cl.c[TLPI].

Builden Sie die Programme, und lassen Sie sie laufen:

```
$ ./i6d_ucase_sv &
$ ./i6d_ucase_cl ::1 hello
```

Hausaufgabe, 3h: Web Client und Server

Lesen Sie die Kapitel 4 bis 7 der HTTP Spezifikation
https://tools.ietf.org/html/rfc2616

Lösen Sie die beiden folgenden Hands-on Aufgaben.

n|w

Hands-on, 1h: 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.

n|w

Hands-on, 1h: 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"
```

n|w

Feedback?

Gerne im Slack oder an thomas.amberg@fhnw.ch

Programmierfragen am besten schriftlich.

Sprechstunde auf Voranmeldung :)

Slides, Code & Hands-on: tmb.gr/syspr-9