

# Systemprogrammierung

## Foliensatz G

- Projekt: Web- und FTP-Proxy
- TCP/IP-Grundlagen

Prof. Dr. Hans-Georg Eßer

Wintersemester 2023/24

v1.0 – 17.01.2024

## Programmier-Projekt

- Implementierung eines Web-Proxys mit folgenden Features
  - HTTP Proxy (GET, HEAD, POST)
  - Keep-Alive (persistente Verbindung)
  - Caching im RAM plus HEAD-Anfrage
  - Frei wählbare Ports für Proxy-Dienst und Steuerung / Konfiguration
  - beliebig viele parallele Verbindungen, über Threads realisiert
  - Server beendet sich/gibt Statusinfo, wenn das Dokument /exit bzw. /status angefordert wird
  - Zugriff auf FTP-Server

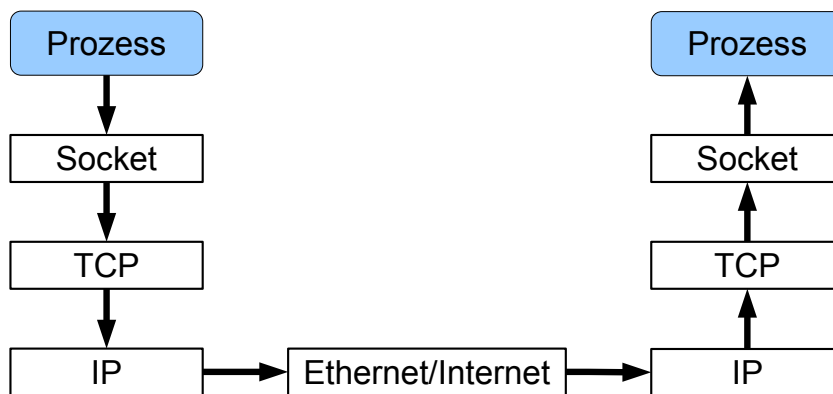
## Projekt: Grundlagen

### Übersicht

- Datenstrukturen für Sockets
- `socket()`, `bind()`, `listen()`, `accept()`
- `htons()`, `ntohs()`, `inet_ntoa()`, `inet_aton()`
- Socket-Deskriptoren als File-Deskriptoren
- Basics zu HTTP

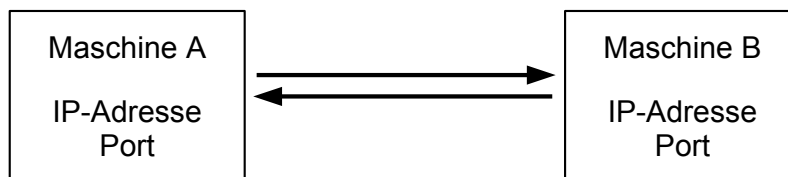
## TCP, IP & Co. (1)

- Hier keine Theorie zu Netzwerken, Layers etc.
- Sockets erlauben Kommunikation über TCP/IP



## TCP, IP & Co. (2)

- TCP (Transmission Control Protocol):
  - verbindungsorientiert, zuverlässig
  - nutzt IP (Internet Protocol)
- TCP-Verbindung: (IP<sub>1</sub>, Port<sub>1</sub>, IP<sub>2</sub>, Port<sub>2</sub>)



(Port nötig, da mehrere Verbindungen möglich)

## TCP, IP & Co. (3)

- netstat zeigt (u. a.) offene TCP-Verbindungen an, bekannte Ports erscheinen mit Namen

```
[esser@macbookpro:C-Projekte]$ netstat
Active Internet connections (w/o servers)
Proto Rec-Q Snd-Q Local Address Foreign Address State
tcp4 0 0 macbookpro.59433 fau1s219.inform.https ESTABLISHED
tcp4 0 0 macbookpro.59432 channelproxy-shv.https ESTABLISHED
tcp4 0 0 macbookpro.59428 edge-star-shv-07.https ESTABLISHED
tcp4 0 0 macbookpro.59348 my.ohmportal.de.imaps ESTABLISHED
tcp4 0 0 macbookpro.59347 imap.1und1.de.imap2 ESTABLISHED
tcp4 0 0 macbookpro.59241 173.194.116.149.https ESTABLISHED
[...]
[esser@macbookpro:~]$ egrep "^imap|^https" /etc/services | grep tcp
imap2 143/tcp # Internet Message Access Protocol
https 443/tcp # http protocol over TLS/SSL
imaps 993/tcp # imap4 protocol over TLS/SSL
```

- `netstat -a` zeigt zusätzlich Server an, d. h., Sockets, die auf Verbindungsanfragen warten:

```
[esser@macbookpro:C-Projekte]$ netstat -a
Active Internet connections (servers and established)
Proto Rec-Q Snd-Q Local Address Foreign Address State
tcp4 0 0 macbookpro.59433 faui1s219.inform.https ESTABLISHED
tcp4 0 0 *.http-alt *.* LISTEN
tcp4 0 0 macbookpro.59432 channelproxy-shv.https ESTABLISHED
[...]
```

- Ubuntu: `netstat` in Paket `net-tools` enthalten;  
`sudo apt install net-tools`

## Datenstrukturen

- Socket-Deskriptoren
  - einfache Integers, wie File-Deskriptoren
  - `socket()` & `accept()` geben Socket-Deskriptoren zurück

```
struct sockaddr_in {
    short          sin_family;   // Typ, z. B. AF_INET
    unsigned short sin_port;     // Port, z. B. htons (8080)
    struct in_addr sin_addr;     // kodierte Adresse
    char           sin_zero[8];  // freier Platz
};
```

- IP-Adressen

```
struct in_addr {
    unsigned int s_addr;         // mit inet_aton() füllen
                                // oder auf INADDR_ANY setzen
};                               // 32 Bit für IP-Adresse
```

## Sockets erzeugen

### Zwei Arten von Sockets (für unsere Zwecke)

- generischer Socket, der mit `socket()` erzeugt wird  
→ kann verwendet werden,
  - um einen TCP-Port zu binden; bleibt dann dauerhaft „in Betrieb“ (Server)
  - um eine Verbindung zu einem Server aufzubauen (Client)
- Verbindungs-Socket, der mit `accept()` erzeugt wird  
→ ist nur für eine konkrete Verbindung (mit einem Client) zuständig

## Sockets erzeugen

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

int
connect(int socket, const struct sockaddr *address,
         socklen_t address_len);           Client

int
bind(int socket, const struct sockaddr *address,
      socklen_t address_len);

int
listen(int socket, int backlog);           Server

int
accept(int socket, struct sockaddr *restrict address,
       socklen_t *restrict address_len);
```

## Sockets erzeugen

- **Server**
  - sd = socket ()
  - Server-Adresse konfigurieren
  - bind (sd, Server-Adresse)
  - listen (sd)
  - conn = accept (sd, &Client-Adresse)  
(zweiter Socket!)
- **Client**
  - sd = socket ()
  - Ziel-Adresse konfigurieren
  - connect (sd, &Ziel-Adresse)

## TCP-Server erzeugen (1)

```
int sd; // socket descriptor for server
struct sockaddr_in server;

sd = socket (PF_INET, SOCK_STREAM, 0);
// PF_INET: Protocol Family, Internet, SOCK_STREAM: TCP
server.sin_port = htons (PORT);
server.sin_addr.s_addr = INADDR_ANY;
server.sin_family = AF_INET; // Address Family, Internet

int conn; // socket descriptor for connection
#define SOCKADDR_SIZE sizeof(struct sockaddr_in)
int clilen = SOCKADDR_SIZE;
struct sockaddr_in client;

bind (sd, (struct sockaddr *)&server, SOCKADDR_SIZE);
listen (sd, 0);
```

## TCP-Server erzeugen (2)

```
conn = accept (sd, (struct sockaddr *) &client, &clilen);

char buf[bufsize];

// lesen: wie aus Datei
readbytes = read (conn, &buf, bufsize);

// schreiben: wie in Datei
write (conn, &buf, n);

// diese Verbindung schließen
shutdown (conn, SHUT_RDWR);
close (conn);
```

## TCP-Client erzeugen

```
int sd; // socket descriptor for server
struct sockaddr_in client;
sd = socket (PF_INET, SOCK_STREAM, 0);

client.sin_port = htons(PORT);
client.sin_addr.s_addr = inet_addr("192.1.2.3");
client.sin_family = AF_INET; // Address Family, Internet

int res = connect (sd, (struct sockaddr *)&client, sizeof(client));
write (sd, request, strlen(request));
int n = read (sd, response, sizeof(response));
```

## send(), recv()

- Statt write und read besser send und recv verwenden  
→ speziell für Sockets gedacht
- write (sd, buffer, length);  
→  
send (sd, buffer, length, flags);
- read (sd, buffer, length);  
→  
recv (sd, buffer, length, flags);
- jeweils mit flags = 0 (siehe Manpages, führt zu send = write bzw. recv = read)

## htons(), ntohs()

- TCP/IP gibt Standard-Byte-Order vor (Network Byte Order, Big-Endian, most-significant byte first)
- Linux-PC verwendet Little-Endian (least-significant byte first)
- htons() und ntohs() konvertieren Portnummern:  
**host to network** bzw. **network to host**
- darum:  

```
server.sin_port = htons(PORT);
```

## inet\_ntoa()

- Aus Adress-Eintrag IP-Adresse des Clients mit  
inet\_ntoa() auslesen:  

```
struct sockaddr_in client;  
accept (sd, (struct sockaddr *)&client, &clilen);  
printf ("Client-Adresse: %s \n",  
        inet_ntoa (client.sin_addr));
```
- wertet client.sin\_addr.s\_addr aus

```
struct sockaddr_in {  
    sa_family_t    sin_family; /* AF_INET */  
    in_port_t      sin_port;   /* Port number */  
    struct in_addr  sin_addr;   /* IPv4 address */  
};  
  
struct in_addr {  
    in_addr_t s_addr;  
};  
  
typedef uint32_t in_addr_t;
```

## Demo: Echo-Server

- TCP Echo Server
- Quelle:  
<https://github.com/mafintosh/echo-servers.c/blob/master/tcp-echo-server.c>

# HTTP

- TCP-Verbindung über Sockets, ASCII-Protokoll

- Client → Server:

```
GET /index.html HTTP/1.1
Host: domainname.top
User-Agent: Mozilla/5.0 (...)
```

- Server → Client:

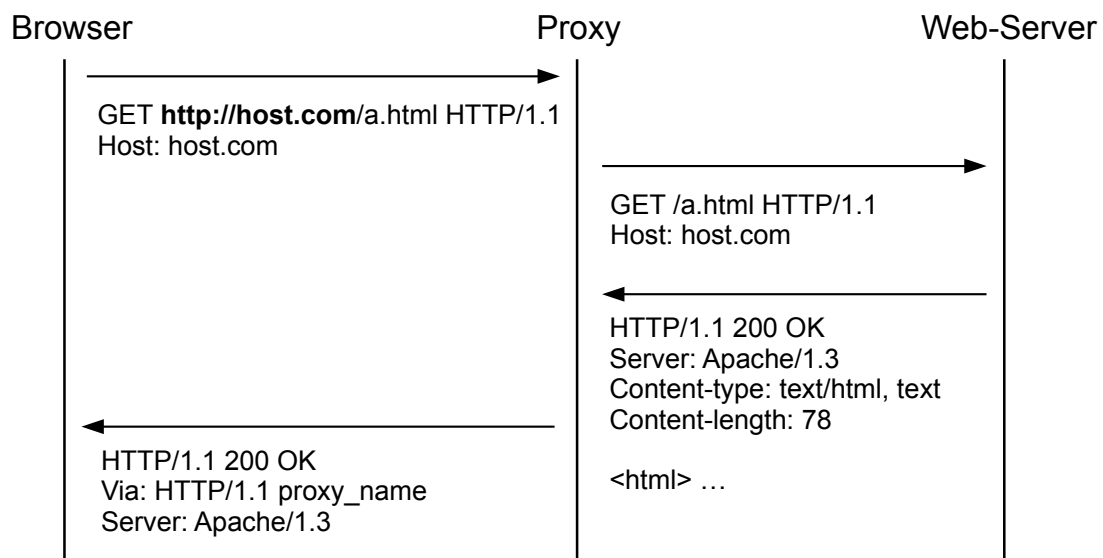
```
HTTP/1.1 201 OK
Content-Type: text/html
```

```
<html>
...
```

```
HTTP/1.1 404 Not found
Content-Type: text/html
```

```
<html>
...
```

## Proxy-Grundlagen



## Proxy-Grundlagen

- Browser schickt an Proxy-Server eine Anfrage der folgenden Form:

```
GET http://google.de/ HTTP/1.1
Host: google.de
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:40.0)
Gecko/20100101 Firefox/40.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: de,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
Cookie: SID=DQAAAF [...]
Connection: keep-alive
```

- Anders als bei normaler HTTP-Anfrage:  
enthält Domain (`http://google.de/` statt `/`)

# Schneller Test mit Squid in Docker-Container

- Container mit Squid:

```
$ docker run -d --name squid-container -e TZ=UTC -p 3128:3128 \
  ubuntu/squid:5.2-22.04_beta
```

- zweifacher Test-Download mit wget:

```
$ export http_proxy=localhost:3128
$ wget swf.hgesser.de/index.html
$ wget swf.hgesser.de/index.html
```

- Log im Container:

```
$ docker exec -it squid-container bash
# tail /var/log/squid/access.log
1705472535.805      31 172.17.0.1 TCP_MISS/200 390
  GET http://swf.hgesser.de/test.txt -
  HIER_DIRECT/217.160.135.96 text/plain
1705472541.534     14 172.17.0.1 TCP_REFRESH_UNMODIFIED/200 396
  GET http://swf.hgesser.de/test.txt -
  HIER_DIRECT/217.160.135.96 text/plain
```

The screenshot shows a Wireshark interface capturing traffic on the loopback interface lo0 (port 3128). The packet list pane shows several packets, with packet 8 selected, which is an HTTP GET request. The packet details pane shows the structure of the request, including the request line, headers, and body. The packet bytes pane shows the raw data of the request.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	:::1	:::1	TCP	88	60128 → 3128 [SYN] Seq=0 Win=65535 Len=0 MSS=16324 WS=64 TSval=792241715 TSecr=0
2	0.000074	:::1	:::1	TCP	88	3128 → 60128 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16324 WS=64 TSval=276524527 TSecr=0
3	0.000085	:::1	:::1	TCP	76	60128 → 3128 [ACK] Seq=1 Ack=1 Win=407744 Len=0 TSval=792241715 TSecr=276524527
4	0.000092	:::1	:::1	TCP	76	[TCP Window Update] 3128 → 60128 [ACK] Seq=1 Ack=1 Win=407744 Len=0 TSval=276524527 TSecr=0
5	0.000127	:::1	:::1	HTTP	266	GET http://swf.hgesser.de/index.html HTTP/1.1
6	0.000150	:::1	:::1	TCP	76	3128 → 60128 [ACK] Seq=1 Ack=191 Win=407552 Len=0 TSval=276524527 TSecr=792241715
7	0.017136	:::1	:::1	TCP	8582	3128 → 60128 [PSH, ACK] Seq=1 Ack=191 Win=407552 Len=8506 TSval=276524545 TSecr=792241715
8	0.017150	:::1	:::1	HTTP	2951	HTTP/1.1 200 OK (text/html)
9	0.017158	:::1	:::1	TCP	76	60128 → 3128 [ACK] Seq=191 Ack=8507 Win=399232 Len=0 TSval=792241733 TSecr=276524527
10	0.017167	:::1	:::1	TCP	76	60128 → 3128 [ACK] Seq=191 Ack=11382 Win=396416 Len=0 TSval=792241733 TSecr=276524527
11	0.018376	:::1	:::1	TCP	76	60128 → 3128 [FIN, ACK] Seq=191 Ack=11382 Win=396416 Len=0 TSval=792241734 TSecr=276524527
12	0.018398	:::1	:::1	TCP	76	3128 → 60128 [ACK] Seq=11382 Ack=192 Win=407552 Len=0 TSval=276524546 TSecr=792241734
13	0.019203	:::1	:::1	TCP	76	3128 → 60128 [FIN, ACK] Seq=11382 Ack=192 Win=407552 Len=0 TSval=276524546 TSecr=792241734
14	0.019229	:::1	:::1	TCP	76	60128 → 3128 [ACK] Seq=192 Ack=11383 Win=396416 Len=0 TSval=792241734 TSecr=276524546

Frame 5: 266 bytes on wire (2128 bits), 266 bytes captured (2128 bits) on interface 0  
Null/Loopback  
> Internet Protocol Version 6, Src: :::1, Dst: :::1  
> Transmission Control Protocol, Src Port: 60128, Dst Port: 3128, Seq: 1, Ack: 1, Len: 190  
Hypertext Transfer Protocol  
GET http://swf.hgesser.de/index.html HTTP/1.1\r\n  
[Expert Info (Chat/Sequence): GET http://swf.hgesser.de/index.html HTTP/1.1\r\n] [GET http://swf.hgesser.de/index.html HTTP/1.1\r\n] [Severity level: Chat] [Group: Sequence]  
Request Method: GET  
Request URI: http://swf.hgesser.de/index.html  
Request Version: HTTP/1.1  
Host: swf.hgesser.de\r\nUser-Agent: Wget/1.21.4\r\nAccept: \*/\*\r\nAccept-Encoding: identity\r\nConnection: Keep-Alive\r\nProxy-Connection: Keep-Alive\r\n  
0000 1e 00 00 00 60 0f 00 00 00 de 06 40 00 00 00 00 .....@.....  
0010 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00 .....  
0020 00 00 00 00 00 00 00 00 00 00 00 01 ea e0 0c 38 .....8

Loopback: lo0: <live capture in progress> Packets: 50 - Displayed: 50 (100.0%) Profile: Default