

## Letzte Woche

- Protokolle
- ISO/OSI Schichtenmodell
- Aufgaben der einzelnen Schichten
- Internet-Protokoll, IPv4
- IP-Adressen und Adressklassen

## Klassenloses IP

Probleme mit Klassen:

- Große Routingtabellen
- IP-Adressen werden knapp

Deshalb: *Classless InterDomain Routing (CIDR)*

- Betrifft zuerst nur Class-C-Netze
- Aneinander grenzende Class-C-Netze statt Class-B-Netz
- Adressblöcke für regionale Zonen
- Router erkennen Adressblöcke

## Nahe Verwandte

Wichtig für das Internet, aber nicht im Fokus:

- *Internet Control Message Protocol (ICMP)*  
Fehlerbehandlung, Steuerung innerhalb von IP
- *Address Resolution Protocol (ARP)*  
Adressabbildung IP-Adressen auf Schicht-2-Adressen
- *Domain Name Service (DNS)*  
Bildet Namen auf IP-Adressen ab

## IPv6

IP-Adressen (32 Bit) werden knapp.

- Nachfolger für das IP-Protokoll: IPv6
- Adressen mit Länge 128 Bit
- Vereinfachtes Headerformat
- Funktionalität zur sicheren Kommunikation

## TCP und UDP

Implementieren Schicht 4 (transport layer):

- *Transmission Control Protocol (TCP)*
- *User Datagram Protocol (UDP)*

IP-Adressen identifizieren einen Host.

- Es laufen mehrere Prozesse auf einem Host
- Zusätzliche Adressierung für Prozesse: Ports
- 16 Bit Ganzzahl

## well-known ports

Ports ermöglichen Prozess-zu-Prozess-Kommunikation.

*well-known ports:*

- Bestimmte Dienste auf bestimmten Port-Nummern
- Z. B. Port 80 für HTTP, 25 für SMTP (Mail)...
- Über diese Ports beginnt die Kommunikation

## Kurze Zusammenfassung

Eine Verbindung ist gekennzeichnet durch:

- Das Protokoll (TCP, UDP)
- IP-Adresse des lokalen Hosts
- Die lokale Port-Nummer
- IP-Adresse des entfernten Hosts
- Port-Nummer des entfernten Hosts

## UDP

- Verbindungsloses Protokoll
- Nicht zuverlässig
- Einziges Feature: optionale Checksumme für das Paket
- Einsatzgebiet: Übertragung von Multimedia-Daten

## TCP Eigenschaften (1)

- Verbindungsorientiert:  
Einmal aufgebaute Verbindung besteht, bis sie beendet wird
- Korrektheit der Übertragung wird mit Prüfsummen geprüft
- Automatische Wiederholung der Übertragung im Fehlerfall

## TCP Eigenschaften (2)

- Empfangene Nachrichten werden bestätigt. Es gehen keine Daten unbemerkt verloren.
- Erhaltung der Sende-Reihenfolge
- Stromorientiert:  
Keine maximale Nachrichtenlänge
- Flußkontrolle:  
Automatische Anpassung der Übertragungsgeschwindigkeit

TCP ist das wichtigste Schicht-4-Protokoll im Internet

## Vergleich UDP/TCP

	UDP	TCP
Verbindungsorientiert?	nein	ja
Längenbeschränkungen?	ja	nein
Checksumme	optional	ja
Empfangsbestätigung?	nein	ja
Timeout mit erneuter Übertragung	nein	ja
Erkennung von Duplikaten?	nein	ja
Reihenfolge?	nein	ja
Flußkontrolle?	nein	ja

***Mit Sockets programmieren***

## Sockets

Sockets:

- Programmiermodell für Netzwerk-Programmierung
- Standardisierte Schnittstelle (POSIX)
- Socket als Endpunkt einer Verbindung

Jetzt: Kurzer Überblick über die Schnittstelle

Mehr Details im Skript und auf den `man`-pages:

Beispiel: `man 2 socket` oder `man -s 3socket socket`

## Neuen Socket erzeugen

Die Funktion `socket()` erzeugt einen Socket:

```
#include <sys/types.h>
#include <sys/socket.h>

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

Für TCP-Verbindung:

```
sd = socket (AF_INET, SOCK_STREAM, 0);
```

Für UDP-Verbindung:

```
sd = socket (AF_INET, SOCK_DGRAM, 0);
```

## Lokale Adresse assoziieren (1)

`bind()` assoziiert lokale Adresse mit einem Socket:

```
#include <sys/types.h>
#include <sys/socket.h>
```

```
int bind(int sd, struct sockaddr *addr,
        int addrlen);
```

Platzhalter `struct sockaddr`:

```
struct sockaddr {
    unsigned short sa_family;
    char sa_data[14];
}
```

Wird durch protokoll-spezifisches Struct ersetzt.

## Lokale Adresse assoziieren (2)

Für IP-Adressen wird `struct sockaddr_in` verwendet:

```
#include <netinet/in.h>
```

```
struct in_addr {
    unsigned long s_addr;
}
```

```
struct sockaddr_in {
    short sin_family;
    unsigned short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
}
```

## Lokale Adresse assoziieren (3)

Bei der Verwendung von `bind()` zu beachten:

- Aufruf im Client optional
- Structs mit `bzero()` initialisieren
- Portnummer in Network-Byte-Order (Konvertierung mit `htonl()`)

IP-Adressen von/zu Strings konvertieren:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int inet_aton(char *str, struct in_addr *ip);

char* inet_ntoa(struct in_addr in);
```

## Server: Vorbereitungen

Server macht sich mit `listen()` bereit für eingehende Verbindungen:

```
#include <sys/types.h>
#include <sys/socket.h>

int listen(int sd, int backlog)
```

- `listen()` kehrt nach Aufruf sofort zurück
- `backlog`: Länge Warteschlange ( `SOMAXCONN` )

## Verbindungsaufbau und Kommunikation

Verbindung mit `connect()` aufbauen:

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sd, struct sockaddr *addr,
            int addrlen)
```

- Adresse des Kommunikationspartners in `addr`
- Sockets lesen und schreiben: `read()` und `write()`

## Server: Warten

Server wartet mit `accept()` auf eingehende Verbindungen:

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int sd, struct sockaddr *addr,
           int *addrlen)
```

- Blockiert bis eingehende Verbindung vorhanden
- Gibt Socket für Kommunikation mit Client zurück
- Adresse des Clients ist in `addr`

## Ablauf der Kommunikation

Client	Server
<code>socket()</code>	<code>socket()</code>
	<code>bind()</code>
	<code>listen()</code>
	<code>accept()</code>
<code>connect()</code>	
<code>read/write</code>	<code>read/write</code>
<code>close()</code>	<code>close</code>

### *Client mit TCP in C*

## Fehlerbehandlung

Fehlerbehandlung bei erwähnten Funktionen:

- Bei Fehler ist der Rückgabewert negative Ganzzahl
- Fehlercode in globaler Variable `errno`
- Erläuterungen zu Fehlern auf `man-page`

Fehlerbeschreibung zu einem `errno`-Code als Text:

```
#include <stdio.h>
```

```
char* strerror(int errnum)
```

## socket\_client.c (1)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
```

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
int panic(char *where)
{
    fprintf(stderr, "error at %s '%s'\n", where, strerror(errno));
    exit(127);
}
```

## socket\_client.c (2)

```
int main(void) {
    int sd;
    struct sockaddr_in remote_addr;
    int addrlen = sizeof(remote_addr);

    bzero((void *) &remote_addr, addrlen);
    remote_addr.sin_family = AF_INET;
    remote_addr.sin_port = htons(8005);
    remote_addr.sin_addr.s_addr = inet_addr("134.2.12.120");

    if ((sd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        panic("socket");

    if (connect(sd, (struct sockaddr*) &remote_addr, addrlen) < 0)
        panic("connect");

    printf("Connected to server\n");
    close (sd);
    return 0;
}
```

*Server mit TCP in C*

## socket\_server.c (1)

```
int main(void) {

    int sd;
    struct sockaddr_in local_addr;
    int local_addr_len = sizeof(local_addr);

    if ((sd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        panic("socket");

    bzero((void *) &local_addr, local_addr_len);
    local_addr.sin_family = AF_INET;
    local_addr.sin_port = htons(8005);
    local_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
```

## socket\_server.c (2)

```
    if (bind(sd, (struct sockaddr *) &local_addr,
             local_addr_len) < 0)
        panic("bind");

    if (listen(sd, SOMAXCONN) < 0)
        panic("listen");
```

## socket\_server.c (3)

```
while (1) {
    int new_sd;
    struct sockaddr_in client_addr;
    unsigned int client_addr_len = sizeof(struct sockaddr);

    new_sd = accept(sd, (struct sockaddr*) &client_addr,
                   &client_addr_len);

    if (new_sd < 0)
        panic("accept");

    printf("New connection  %d\n", new_sd);
    close (new_sd);
}

close(sd);
return 0;
}
```

## Hypertext Transfer Protocol (HTTP)

## Adressen im World-Wide-Web

Syntax für Adressen im World-Wide-Web:  
*Uniform Resource Locator (URL)*

Allgemeinere Fassung *Uniform Resource Identifier (URI)*:

```
URI = scheme ":" hier-part [ "?" query ] [ "#" fragment ]
```

Beispiele:

```
ftp://ftp.informatik.uni-tuebingen.de/pub/wsi.ps.gz
http://www.ietf.org/rfc/rfc3986.txt
http://www.beispiel.de/cgi/;sid=12/test?c=%2A#(complex)
mailto:knauel@informatik.uni-tuebingen.de
tel:+49-7071-2970500
```

## Bestandteile von URIs

Interpretation von URIs:

- **authority**: Adresse des Servers
- **hier-part**: identifiziert Dokument oder Programm auf dem Server  
( *Resource* )
- **query**: Argumente für Programme
- **fragment** und abgetrennte Bestandteile des Pfades: selten verwendet

## Syntax für URIs (vereinfacht)

Server und Pfad auf dem Server:

```
hier-part      = "//" authority path-abempty
```

Pfade:

```
path-abempty  = ( "/" segment )*
```

Bestandteile eines Pfades:

```
segment       = pchar*
pchar         = unreserved | pct-encoded | sub-delims | ":" | "@"
pct-encoded   = "%" HEXDIG HEXDIG
unreserved   = ALPHA | DIGIT | "-" | "." | "_" | "~"
reserved     = gen-delims | sub-delims
gen-delims   = ":" | "/" | "?" | "#" | "[" | "]" | "@"
sub-delims   = "!" | "$" | "&" | "'" | "(" | ")"
              | "*" | "+" | "," | ";" | "="
```

## HTTP-Nachrichten

Es gibt zwei gleich strukturierte HTTP-Nachrichten:

```
HTTP-message = Request | Response
```

Aufbau einer HTTP-Nachricht:

```
generic-message = start-line
                  (message-header CRLF)*
                  CRLF
                  [ message-body ]

start-line       = Request-Line | Status-Line
```

## Ablauf Kommunikation HTTP

Ablauf einer Kommunikation über HTTP:

- Client baut Verbindung auf
- Client sendet **Request**
- Server sendet **Response**
- Verbindung wird beendet

*persistente Verbindungen*: optionales Feature bei HTTP 1.1. Mehrere **Requests** pro Verbindung

## HTTP-Anfragen

Anfragen werden durch **Request-Line** eingeleitet:

```
Request-Line   = Method " " Request-URI " " HTTP-Version CRLF
```

**Method**: Operation, die auf einer Resource ausgeführt werden soll.

Einige der möglichen Methoden:

```
Method         = "GET"
                | "HEAD"
                | "POST"
                | "PUT"
                | "DELETE"
                | ...
                | extension-method
extension-method = token
```

Für unsere Zwecke: **HTTP-Version** = "HTTP/1.1"

## Header einer HTTP-Message

Syntax von HTTP-Headern:

```
message-header = field-name ":" [ field-value ]
field-name     = <printable ASCII characters>
field-value    = (field-content | LWS)*
field-content  = <any ASCII characters other than CR or LF>
LWS           = [CRLF] ( " " | ASCII 9)
```

Im Allgemeinen: Erlaubte Header sind abhängig von verwendeter Methode

## Die GET-Methode

**GET** ist der häufigste Typ von Anfragen. Liest Inhalt des Dokumentes an der URI aus.

- Anfrage enthält keinen Rumpf
- URI-Pfad wird z. B. als Pfad im Dateisystem des Webserver interpretiert
- Zugehörige **Response** enthält Inhalt der Datei
- Ausführung der Anfrage kann an Bedingungen geknüpft werden

Die **HEAD**-Methode funktioniert genauso. Die Response enthält allerdings keinen Rumpf.