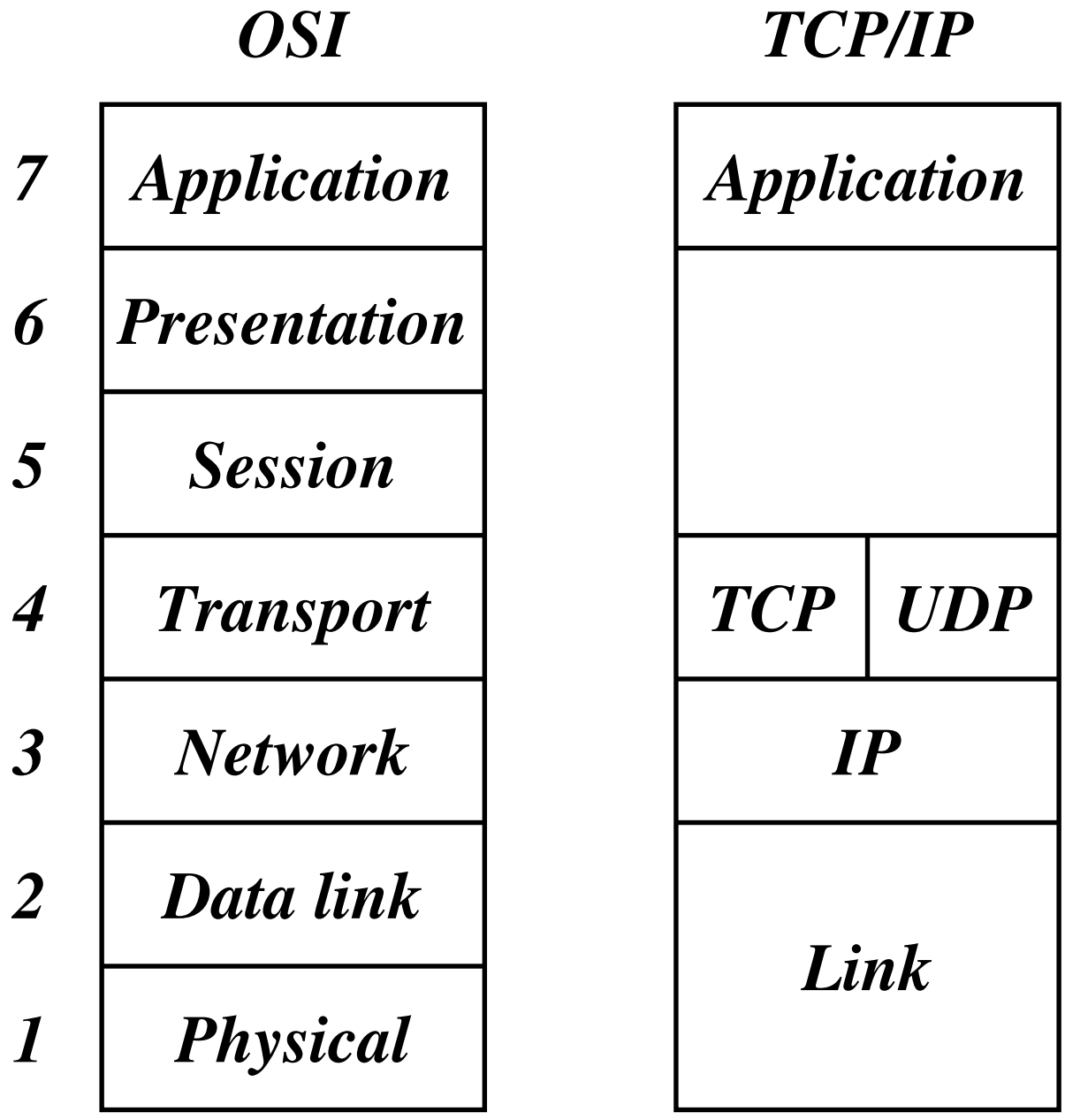
- IP-Adressen wie *134.60.66.7* werden nur auf einer abstrakten Ebene zur Verfügung gestellt.
- IP-Adressen werden auf der darunterliegenen physischen Ebene und denen damit verbundenen Protokollen nicht verstanden.
- So wird beispielsweise beim Ethernet, das bei uns weitgehend an der Universität zum Einsatz kommt, mit 6-Byte-Adressen gearbeitet.
- Die Theon hat beispielsweise die Ethernet-Adresse *90:1b:0e:ae:12:84* (Bytes werden hier in Form von Hexzahlen angegeben). Diese Adressen sind jedoch nur lokal auf einem Ethernet-Segment von Bedeutung.

- Aufbauend auf der Schicht mit IP-Adressen (IP-Protokoll) gibt es alternative Transport-Schichten, über die Pakete versendet werden können.
- Mittels UDP (*User Datagram Protocol*) können einzelne Pakete sehr effizient, aber unzuverlässig versendet werden.
- Im Gegensatz dazu gewährleistet TCP (*Transmission Control Protocol*) eine sichere Verbindung, die jedoch mehr Verwaltungsaufwand mit sich bringt.
- Parallel zu TCP/IP entstand 1983 das OSI-Referenz-Modell (*Open Systems Interconnection*), das eine feinere Schichtung vorsieht. Die Präsentations- oder Sitzungsebene fand jedoch nie ihren Weg in die Protokollhierarchie von TCP/IP.


OSI
TCP/IP
7
Application
6
Presentation
5
Session
4
Transport
3
Network
2
Data link
1
Physical
Application
TCP
UDP
IP
Link

- Für TCP/IP gibt es zwei Schnittstellen, die beide zum POSIX-Standard gehören:
- Die Berkeley Sockets wurden 1983 im Rahmen von BSD 4.2 eingeführt. Dies war die erste TCP/IP-Implementierung.
- Im Jahr 1987 kam durch UNIX System V Release 3.0 noch TLI (*Transport Layer Interface*) hinzu, die auf Streams basiert (einer anderen System-V-spezifischen Abstraktion).
- Die Berkeley-Socket-Schnittstelle hat sich weitgehend durchgesetzt. Wir werden uns daher nur mit dieser beschäftigen.

Die Entwickler der Berkeley-Sockets setzten sich folgende Ziele:

- **Transparenz**: Die Kommunikation zwischen zwei Prozessen soll nicht davon abhängen, ob sie auf dem gleichen Rechner laufen oder nicht.
- **Effizienz**: Zu Zeiten von BSD 4.2 (also 1983) war dies ein außerordentlich wichtiges Kriterium wegen der damals noch sehr geringen Rechenleistung. Aus diesem Grund werden insbesondere keine weiteren System-Prozesse zur Kommunikation eingesetzt, obwohl dies zu mehr Flexibilität und Modularität hätte führen können.
- **Kompatibilität**: Viele bestehende Applikationen und Bibliotheken wissen nichts von Netzwerken und sollen dennoch in einem verteilten Umfeld eingesetzt werden können. Dies wurde dadurch erreicht, dass nach einem erfolgten Verbindungsaufbau (der z.B. von einem anderen Prozess durchgeführt werden kann) Ein- und Ausgabe in gewohnter Weise (wie bei Dateien, Pipelines oder Terminal-Verbindungen) erfolgen können.

Die Semantik einer Kommunikation umschließt bei jeder Verbindung eine Teilmenge der folgenden Punkte:

1. Daten werden in der Reihenfolge empfangen, in der sie abgeschickt worden sind.
2. Daten kommen nicht doppelt an.
3. Daten werden zuverlässig übermittelt.
4. Einzelne Pakete kommen in der originalen Form an (d.h. sie werden weder zerstückelt noch mit anderen Paketen kombiniert).
5. Nachrichten außerhalb des normalen Kommunikationsstromes (*out-of-band messages*) werden unterstützt.
6. Die Kommunikation erfolgt verbindungs-orientiert, womit die Notwendigkeit entfällt, sich bei jedem Paket identifizieren zu müssen.

Die folgende Tabelle zeigt die Varianten, die von der Berkeley-Socket-Schnittstelle unterstützt werden:

Name	1	2	3	4	5	6
SOCK_STREAM	⋆	⋆	⋆		⋆	⋆
SOCK_DGRAM				⋆		
SOCK_SEQPACKET	⋆	⋆	⋆	⋆	⋆	⋆
SOCK_RDM	⋆	⋆	⋆	⋆		

(1: Reihenfolge korrekt; 2: nicht doppelt; 3: zuverlässige Übermittlung; 4: keine Stückelung; 5: *out-of-band*; 6: verbindungsorientiert.)

- *SOCK_STREAM* lässt sich ziemlich direkt auf TCP abbilden.
- *SOCK_STREAM* kommt den Pipelines am nächsten, wenn davon abgesehen wird, dass die Verbindungen bei Pipelines nur unidirektional sind.
- UDP wird ziemlich genau durch *SOCK_DGRAM* widergespiegelt.
- Die Varianten *SOCK_SEQPACKET* (TCP-basiert) und *SOCK_RDM* (UDP-basiert) fügen hier noch weitere Funktionalitäten hinzu. Allerdings fand *SOCK_RDM* nicht den Weg in den POSIX-Standard und wird auch von einigen Implementierungen nicht angeboten.
- Im weiteren Verlauf dieser Vorlesung werden wir uns nur mit *SOCK_STREAM*-Sockets beschäftigen.

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

- Bis zu einem gewissen Grad ist eine Betrachtung, die sich an unserem Telefonsystem orientiert, hilfreich.
- Bevor Sie Telefonanrufe entgegennehmen oder selbst anrufen können, benötigen Sie einen Telefonanschluss.
- Dieser Anschluss wird mit dem Systemaufruf *socket* erzeugt.
- Bei *domain* wird hier normalerweise *PF_INET* angegeben, um das IPv4-Protokoll auszuwählen. (Alternativ wäre etwa *PF_INET6* für IPv6 denkbar.)
- *PF* steht dabei für *protocol family*. Bei *type* kann eine der unterstützten Semantiken ausgewählt werden, also beispielsweise *SOCK_STREAM*.
- Der dritte Parameter *protocol* erlaubt in einigen Fällen eine weitere Selektion. Normalerweise wird hier schlicht 0 angegeben.

- Nachdem der Anschluss existiert, fehlt noch eine zugeordnete Telefonnummer. Um bei der Analogie zu bleiben, haben wir eine Vorwahl (IP-Adresse) und eine Durchwahl (Port-Nummer).
- Auf einem Rechner können mehrere IP-Adressen zur Verfügung stehen.
- Es ist dabei möglich, nur eine dieser IP-Adressen zu verwenden oder alle gleichzeitig, die zur Verfügung stehen.
- Bei den Port-Nummern ist eine automatische Zuteilung durch das Betriebssystem möglich.
- Alternativ ist es auch möglich, sich selbst eine Port-Nummer auszuwählen. Diese darf aber noch nicht vergeben sein und muss bei nicht-privilegierten Prozessen eine Nummer jenseits des Bereiches der wohldefinierten Port-Nummern sein, also typischerweise mindestens 1024 betragen.
- Die Verknüpfung eines Anschlusses mit einer vollständigen Adresse erfolgt mit dem Systemaufruf *bind*...

```
struct sockaddr_in address = {0};
address.sin_family = AF_INET;
address.sin_addr.s_addr = htonl(INADDR_ANY);
address.sin_port = htons(port);
bind(sfd, (struct sockaddr *) &address, sizeof address);
```

- Die Datenstruktur **struct** *sockaddr_in* repräsentiert Adressen für IPv4, die aus einer IP-Adresse und einer Port-Nummer bestehen.
- Das Feld *sin_family* legt den Adressraum fest. Hier gibt es passend zur Protokollfamilie *PF_INET* nur *AF_INET* (*AF* steht hier für *address family*).
- Bei dem Feld *sin_addr.s_addr* lässt sich die IP-Adresse angeben. Mit *INADDR_ANY* übernehmen wir alle IP-Adressen, die zum eigenen Rechner gehören.
- Das Feld *sin_port* spezifiziert die Port-Nummer.

```
struct sockaddr_in address = {0};
address.sin_family = AF_INET;
address.sin_addr.s_addr = htonl(INADDR_ANY);
address.sin_port = htons(port);
bind(sfd, (struct sockaddr *) &address, sizeof address);
```

- Da Netzwerkadressen grundsätzlich nicht von der Byte-Anordnung eines Rechners abhängen dürfen, wird mit *htonl* (*host to network long*) der 32-Bit-Wert der IP-Adresse in die standardisierte Form konvertiert. Analog konvertiert *htons*() (*host to network short*) den 16-Bit-Wert *port* in die standardisierte Byte-Reihenfolge.
- Wenn die Port-Nummer vom Betriebssystem zugeteilt werden soll, kann bei *sin_port* auch einfach 0 angegeben werden.

```
struct sockaddr_in address = {0};
address.sin_family = AF_INET;
address.sin_addr.s_addr = htonl(INADDR_ANY);
address.sin_port = htons(port);
bind(sfd, (struct sockaddr *) &address, sizeof address);
```

- Der Datentyp **struct** *sockaddr_in* ist eine spezielle Variante des Datentyps **struct** *sockaddr*. Letzterer sieht nur ein Feld *sin_family* vor und ein generelles Datenfeld *sa_data*, das umfangreich genug ist, um alle unterstützten Adressen unterzubringen.
- Bei *bind*() wird der von *socket*() erhaltene Deskriptor angegeben (hier *sfd*), ein Zeiger, der auf eine Adresse vom Typ **struct** *sockaddr* verweist, und die tatsächliche Länge der Adresse, die normalerweise kürzer ist als die des Typs **struct** *sockaddr*.
- Schön sind diese Konstruktionen nicht, aber C bietet eben keine objekt-orientierten Konzepte, wenngleich die Berkeley-Socket-Schnittstelle sehr wohl polymorph und damit objekt-orientiert ist.

```
listen(sfd, SOMAXCONN);
```

- Damit eingehende Verbindungen (oder Anrufe in unserer Telefon-Analogie) entgegengenommen werden können, muss *listen*() aufgerufen werden.
- Nach *listen*() kann der Anschluss „klingeln“, aber noch sind keine Vorbereitungen getroffen, das Klingeln zu hören oder den Hörer abzunehmen.
- Der zweite Parameter bei *listen*() gibt an, wieviele Kommunikationspartner es gleichzeitig klingeln lassen dürfen.
- *SOMAXCONN* ist hier das Maximum, das die jeweilige Implementierung erlaubt.

Anschlussdaten ermitteln

newsocket.c

```
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>

void print_ip_addr(in_addr_t ipaddr) {
   if (ipaddr == INADDR_ANY) {
      printf("INADDR_ANY");
   } else {
      uint32_t addr = ntohl(ipaddr);
      printf("%u.%u.%u.%u",
         addr>>24, (addr>>16)&0xff,
         (addr>>8)&0xff, addr&0xff);
   }
}

int main() {
   int sfd = socket(PF_INET, SOCK_STREAM, 0);
   if (sfd < 0) exit(1);
   if (listen(sfd, SOMAXCONN) < 0) exit(2);
   struct sockaddr address;
   socklen_t len = sizeof address;
   if (getsockname(sfd, &address, &len) < 0) exit(3);
   struct sockaddr_in * inaddr = (struct sockaddr_in *) &address;
   printf("This is the address of my new socket:\n");
   printf("IP address:  "); print_ip_addr(inaddr->sin_addr.s_addr);
   printf("\n");
   printf("Port number: %hu\n", ntohs(inaddr->sin_port));
}
```

```
struct sockaddr client_addr;
socklen_t client_addr_len = sizeof client_addr;
int fd = accept(sfd, &client_addr, &client_addr_len);
```

- Liegt noch kein Anruf vor, blockiert *accept*() bis zum nächsten Anruf.
- Wenn mit *accept*() ein Anruf eingeht, wird ein Dateideskriptor auf den bidirektionalen Verbindungskanal zurückgeliefert.
- Normalerweise speichert *accept*() die Adresse des Klienten beim angegebenen Zeiger ab. Wenn als Zeiger 0 angegeben wird, entfällt dies.

Ein einfacher Zeitdienst

timeserver.c

```
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <time.h>
#include <unistd.h>
#define PORT 11011
int main () {
   struct sockaddr_in address = {0};
   address.sin_family = AF_INET;
   address.sin_addr.s_addr = htonl(INADDR_ANY);
   address.sin_port = htons(PORT);
   int sfd = socket(PF_INET, SOCK_STREAM, 0);
   int optval = 1;
   if (sfd < 0 ||
        setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR,
                   &optval, sizeof optval) < 0 ||
        bind(sfd, (struct sockaddr *) &address,
             sizeof address) < 0 ||
        listen(sfd, SOMAXCONN) < 0) {
      perror("socket"); exit(1);
   }
   int fd;
   while ((fd = accept(sfd, 0, 0)) >= 0) {
      char timebuf[32]; time_t clock; time(&clock);
      ctime_r(&clock, timebuf);
      write(fd, timebuf, strlen(timebuf)); close(fd);
   }
}
```

timeserver.c

```
if (sfd < 0 ||
     setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR,
                &optval, sizeof optval) < 0 ||
     bind(sfd, (struct sockaddr *) &address,
          sizeof address) < 0 ||
     listen(sfd, SOMAXCONN) < 0) {
   perror("socket"); exit(1);
}
```

- Hier wird zusätzlich noch *setsockopt* aufgerufen, um die Option *SO_REUSEADDR* einzuschalten.
- Dies empfiehlt sich immer, wenn eine feste Port-Nummer verwendet wird.
- Fehlt diese Option, kann es passieren, dass bei einem Neustart des Dienstes die Port-Nummer nicht sofort wieder zur Verfügung steht, da noch alte Verbindungen nicht vollständig abgewickelt worden sind.

Ein Klient für den Zeitdienst

timeclient.c

```
#include <netdb.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>
#define PORT 11011
int main (int argc, char** argv) {
   char* cmdname = *argv++; --argc;
   if (argc != 1) {
      fprintf(stderr, "Usage: %s host\n", cmdname); exit(1);
   }
   char* hostname = *argv; struct hostent* hp;
   if ((hp = gethostbyname(hostname)) == 0) {
      fprintf(stderr, "unknown host: %s\n", hostname); exit(1);
   }
   char* hostaddr = hp->h_addr_list[0];
   struct sockaddr_in addr = {0}; addr.sin_family = AF_INET;
   memcpy((void *) &addr.sin_addr, (void *) hostaddr, hp->h_length);
   addr.sin_port = htons(PORT);
   int fd;
   if ((fd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
      perror("socket"); exit(1);
   }
   if (connect(fd, (struct sockaddr *) &addr, sizeof addr) < 0) {
      perror("connect"); exit(1);
   }
   char buffer[BUFSIZ]; ssize_t nbytes;
   while((nbytes = read(fd, buffer, sizeof buffer)) > 0 &&
         write(1, buffer, nbytes) == nbytes);
}
```

timeclient.c

```
char* hostname = *argv;
struct hostent* hp;
if ((hp = gethostbyname(hostname)) == 0) {
   fprintf(stderr, "unknown host: %s\n", hostname);
   exit(1);
}
char* hostaddr = hp->h_addr_list[0];
struct sockaddr_in addr = {0};
addr.sin_family = AF_INET;
memmove((void *) &addr.sin_addr, (void *) hostaddr, hp->h_length);
addr.sin_port = htons(PORT);
```

- Der Klient erhält über die Kommandozeile den Namen des Rechners, auf dem der Zeitdienst zur Verfügung steht.
- Für die Abbildung eines Rechnernamens in eine IP-Adresse wird die Funktion *gethostbyname*() benötigt, die im Erfolgsfalle eine oder mehrere IP-Adressen liefert, unter denen sich der Rechner erreichen lässt.
- Hier wird die erste IP-Adresse ausgewählt.

Für die Kombination aus Rechnernamen (oder alternativ einer IP-Adresse) und einer Portnummer gibt es mit RFC 2396 und RFC 2373 einen Standard:

$$
\begin{aligned}
\langle \text{hostport} \rangle &\longrightarrow \langle \text{host} \rangle \ [\ \text{„:“}\ \langle \text{port} \rangle\ ] \\
\langle \text{host} \rangle &\longrightarrow \langle \text{hostname} \rangle \\
&\longrightarrow \langle \text{IPv4address} \rangle \\
&\longrightarrow \langle \text{IPv6reference} \rangle \\
\langle \text{hostname} \rangle &\longrightarrow \{\ \langle \text{domainlabel} \rangle\ \text{„.“}\ \}\ \langle \text{toplabel} \rangle\ [\ \text{„.“}\ ] \\
\langle \text{domainlabel} \rangle &\longrightarrow \langle \text{alphanum} \rangle \\
&\longrightarrow \langle \text{alphanum} \rangle\ \{\ \langle \text{alphanum} \rangle\ \mid\ \text{„-“}\ \}\ \langle \text{alphanum} \rangle \\
\langle \text{toplabel} \rangle &\longrightarrow \langle \text{alpha} \rangle \\
&\longrightarrow \langle \text{alpha} \rangle\ \{\ \langle \text{alphanum} \rangle\ \mid\ \text{„-“}\ \}\ \langle \text{alphanum} \rangle \\
\langle \text{IPv4address} \rangle &\longrightarrow \{\ \langle \text{digit} \rangle\ \}\ \text{„.“}\ \{\ \langle \text{digit} \rangle\ \} \\
&\qquad \text{„.“}\ \{\ \langle \text{digit} \rangle\ \}\ \text{„.“}\ \{\ \langle \text{digit} \rangle\ \}
\end{aligned}
$$

$$
\begin{aligned}
\langle \text{IPv6reference} \rangle &\longrightarrow \text{„[“ } \langle \text{IPv6address} \rangle \text{ „]“} \\
\langle \text{IPv6address} \rangle &\longrightarrow \langle \text{hexpart} \rangle \text{ [„:“ } \langle \text{IPv4address} \rangle \text{]} \\
\langle \text{hexpart} \rangle &\longrightarrow \langle \text{hexseq} \rangle \\
&\longrightarrow \langle \text{hexseq} \rangle \text{ „::“ [} \langle \text{hexseq} \rangle \text{]} \\
&\longrightarrow \text{„::“ [} \langle \text{hexseq} \rangle \text{]} \\
\langle \text{hexseq} \rangle &\longrightarrow \{ \langle \text{hexdigit} \rangle \} \\
&\longrightarrow \langle \text{hexseq} \rangle \text{ „:“ } \{ \langle \text{hexdigit} \rangle \}
\end{aligned}
$$

Beispiele:

- 127.0.0.1:12345
- [::1]:12345
- 0:12345
- *theon.mathematik.uni−ulm.de*:12345

Datenstruktur für einen *hostport*

hostport.h

```
typedef struct hostport {
   /* parameters for socket() */
   int domain;
   int protocol;
   /* parameters for bind() or connect() */
   struct sockaddr_storage addr;
   socklen_t namelen;
} hostport;

bool parse_hostport(char* input, hostport* hp,
   in_port_t defaultport);
```

- In der Vorlesungsbibliothek gibt es eine Funkton *parse_hostport*, die eine Zeichenkette entsprechend der Syntax des RFC 2396 analysiert und in einer **struct** *hostport* ablegt.
- In der Datenstruktur *hostport* liegen alle Parameter, die für die Systemaufrufe *socket()*, *bind()* oder *connect()* benötigt werden.
- Mit so einer Schnittstelle lässt sich auch die Festlegung auf IPv4 oder IPv6 vermeiden.

timeclient2.c

```
hostport hp;
if (!parse_hostport(*argv, &hp, PORT)) {
   fprintf(stderr, "unknown hostport: %s\n", *argv); exit(1);
}
int fd;
if ((fd = socket(hp.domain, SOCK_STREAM, hp.protocol)) < 0) {
   perror("socket"); exit(1);
}
if (connect(fd, (struct sockaddr *) &hp.addr, hp.namelen) < 0) {
   perror("connect"); exit(1);
}
```

- Der im *hostport* verwendete Datentyp **struct** *sockaddr_storage* ist unabhängig von der Wahl eines bestimmten Netzwerks bzw. des zugehörigen Adressraums.
- Deswegen kann nicht mehr **sizeof** verwendet werden, da dieser jetzt eine Maximalgröße aufweist für alle denkbaren Varianten. Die zu verwendende Größe steht über *hp.namelen* zur Verfügung.

timeserver2.c

```
hostport hp;
if (!parse_hostport(*argv, &hp, PORT)) {
   fprintf(stderr, "unknown hostport: %s\n", *argv); exit(1);
}
int sfd;
if ((sfd = socket(hp.domain, SOCK_STREAM, hp.protocol)) < 0) {
   perror("socket"); exit(1);
}

int optval = 1;
if (setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR,
                &optval, sizeof optval) < 0 ||
     bind(sfd, (struct sockaddr *) &hp.addr, hp.namelen) < 0 ||
     listen(sfd, SOMAXCONN) < 0) {
   perror("socket"); exit(1);
}
```

- Analog kann *parse_hostport* auch in Verbindung mit *bind* verwendet werden.

Fragmentierung der Pakete bei Netzwerkverbindungen 213

- Die Ein- und Ausgabe über Netzwerkverbindungen bringt in Vergleich zur Behandlungen von Dateien und interaktiven Benutzern einige Veränderungen mit sich.
- Wenn eine Verbindung des Typs *SOCK_STREAM* zum Einsatz gelangt, so kommen die Daten zwar in der korrekten Reihenfolge an, jedoch nicht in der ursprünglichen Paketisierung.
- Als ursprüngliche Pakete werden hier die Daten betrachtet, die mit Hilfe eines einzigen Aufrufs von *write*() geschrieben werden:

```
const char greeting[] = "Hi, how are you?\r\n";
ssize_t nbytes = write(sfd, greeting, sizeof greeting);
```

Fragmentierung der Pakete bei Netzwerkverbindungen 214

- Wenn beispielsweise bei einer Netzwerkverbindung immer vollständige Zeilen mit *write*() geschrieben werden, so ist es möglich, dass die korrespondierende *read*()-Operation nur einen Teil einer Zeile zurückliefert oder auch ein Fragment, das sich über mehr als eine Zeile erstreckt.
- Diese Problematik legt es nahe, nur zeichenweise einzulesen, wenn genau eine einzelne Zeile eingelesen werden soll:

```
char ch;
stralloc line = {0};
while (read(fd, &ch, sizeof ch) == 1 && ch != '\n') {
   stralloc_append(&line, &ch);
}
```

Fragmentierung der Pakete bei Netzwerkverbindungen 215

- Diese Vorgehensweise ist jedoch außerordentlich ineffizient, weil Systemaufrufe wie *read*() zu einem Kontextwechsel zwischen dem aufrufenden Prozess und dem Betriebssystem führen.
- Wenn ein Kontextwechsel für jedes einzulesende Byte initiiert wird, dann ist der betroffene Rechner mehr mit Kontextwechseln als mit sinnvollen Tätigkeiten beschäftigt.
- Wenn jedoch in größeren Einheiten eingelesen wird, ist möglicherweise mehr als nur die gewünschte Zeile in *buf* zu finden. Oder auch nur ein Teil der Zeile:

```
char buf[512];
ssize_t nbytes = read(fd, buf, sizeof buf);
```

Entsprechend ist eine gepufferte Eingabe notwendig, bei der die Eingabe-Operationen aus einem Puffer versorgt werden, der, wenn er leer wird, mit Hilfe einer *read*()-Operation aufzufüllen ist.

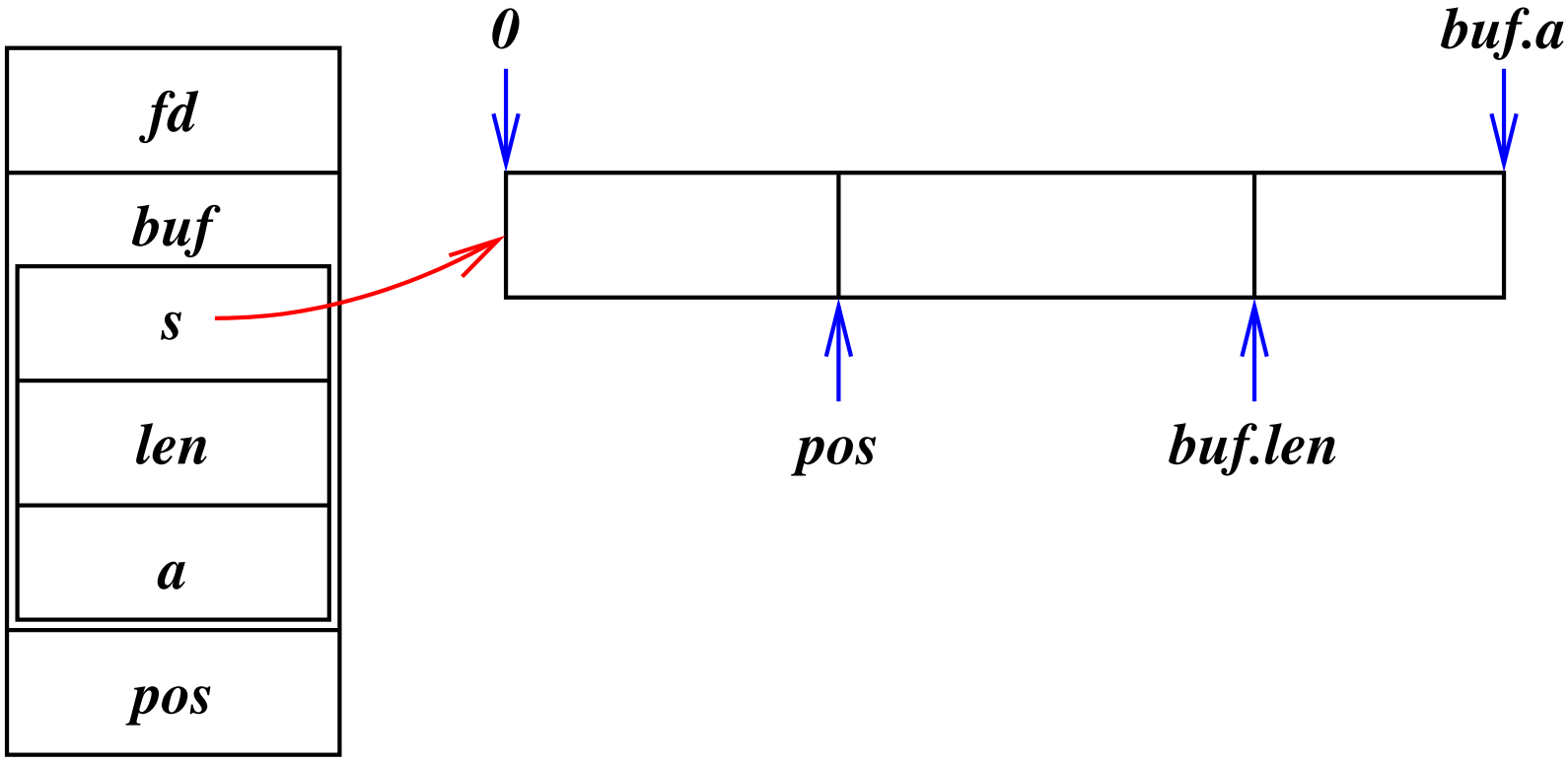
Diese Funktion kann die *stdio* jedoch nicht erfüllen:

- Es gibt keine Lesefunktion, die sich wie *read* auf den vorhandenen Buffer-Inhalt beschränkt. Dies ist jedoch für die Vermeidung ineffizienter Lese-Operationen unerlässlich.
- Systemaufrufe wie *poll* oder *select*, die für Netzwerkverbindungen häufig benötigt werden, funktionieren nur für Dateideskriptoren, nicht für *FILE*. Sie können nicht angepasst werden, da es keine Funktionen gibt, die den Füllungsstand des Buffers angibt.

- Entsprechend wird eine Alternative zur *stdio* benötigt, die für Netzwerkverbindungen geeignet ist.
- Die Datenstruktur für einen Eingabe-Puffer benötigt entsprechend einen Dateideskriptor, einen Puffer und einen Positionszeiger innerhalb des Puffers:

inbuf.h

```
typedef struct inbuf {
   int fd;
   stralloc buf;
   unsigned int pos;
} inbuf;
```


fd
buf
s
len
a
pos
0
buf.a
pos
buf.len

Schnittstelle für eine gepufferte Eingabe

inbuf.h

```
#ifndef AFBLIB_INBUF_H
#define AFBLIB_INBUF_H

#include <stdbool.h>
#include <stddef.h>
#include <stralloc.h>
#include <unistd.h>

typedef struct inbuf {
   int fd;
   stralloc buf;
   size_t pos;
} inbuf;

/* set size of input buffer */
bool inbuf_alloc(inbuf* ibuf, size_t size);

/* works like read(2) but from ibuf */
ssize_t inbuf_read(inbuf* ibuf, void* buf, size_t size);

/* works like fgetc but from ibuf */
int inbuf_getchar(inbuf* ibuf);

/* move backward one position */
bool inbuf_back(inbuf* ibuf);

/* release storage associated with ibuf */
void inbuf_free(inbuf* ibuf);

#endif
```

Implementierung für eine gepufferte Eingabe

inbuf.c

```
/* set size of input buffer */
bool inbuf_alloc(inbuf* ibuf, size_t size) {
   return stralloc_ready(&ibuf->buf, size);
}

/* works like read(2) but from ibuf */
ssize_t inbuf_read(inbuf* ibuf, void* buf, size_t size) {
   if (size == 0) return 0;
   if (ibuf->pos >= ibuf->buf.len) {
      if (ibuf->buf.a == 0 && !inbuf_alloc(ibuf, 512)) return -1;
      /* fill input buffer */
      ssize_t nbytes;
      do {
         errno = 0;
         nbytes = read(ibuf->fd, ibuf->buf.s, ibuf->buf.a);
      } while (nbytes < 0 && errno == EINTR);
      if (nbytes <= 0) return nbytes;
      ibuf->buf.len = nbytes;
      ibuf->pos = 0;
   }
   ssize_t nbytes = ibuf->buf.len - ibuf->pos;
   if (size < nbytes) nbytes = size;
   memcpy(buf, ibuf->buf.s + ibuf->pos, nbytes);
   ibuf->pos += nbytes;
   return nbytes;
}
```

inbuf.c

```
/* works like fgetc but from ibuf */
int inbuf_getchar(inbuf* ibuf) {
   char ch;
   ssize_t nbytes = inbuf_read(ibuf, &ch, sizeof ch);
   if (nbytes <= 0) return -1;
   return ch;
}

/* move backward one position */
bool inbuf_back(inbuf* ibuf) {
   if (ibuf->pos == 0) return false;
   ibuf->pos--;
   return true;
}

/* release storage associated with ibuf */
void inbuf_free(inbuf* ibuf) {
   stralloc_free(&ibuf->buf);
}
```

- Die Ausgabe sollte ebenfalls gepuffert erfolgen, um die Zahl der Systemaufrufe zu minimieren.
- Ein Positionszeiger ist nicht erforderlich, wenn Puffer grundsätzlich vollständig an *write*() übergeben werden.
- Hier ist das einzige Problem, dass die *write*()-Operation unter Umständen nicht den gesamten gewünschten Umfang akzeptiert und nur einen Teil der zu schreibenden Bytes akzeptiert und entsprechend eine geringere Quantität als Wert zurückgibt.

outbuf.h

```
typedef struct outbuf {
   int fd;
   stralloc buf;
} outbuf;
```

outbuf.h

```
#ifndef AFBLIB_OUTBUF_H
#define AFBLIB_OUTBUF_H

#include <stdbool.h>
#include <stddef.h>
#include <stralloc.h>
#include <unistd.h>

typedef struct outbuf {
   int fd;
   stralloc buf;
} outbuf;

/* works like write(2) but to obuf */
ssize_t outbuf_write(outbuf* obuf, void* buf, size_t size);

/* works like fputc but to obuf */
int outbuf_putchar(outbuf* obuf, char ch);

/* write contents of obuf to the associated fd */
bool outbuf_flush(outbuf* obuf);

/* release storage associated with obuf */
void outbuf_free(outbuf* obuf);

#endif
```

outbuf.c

```
/* works like write(2) but to obuf */
ssize_t outbuf_write(outbuf* obuf, void* buf, size_t size) {
   if (size == 0) return 0;
   if (!stralloc_readyplus(&obuf->buf, size)) return -1;
   memcpy(obuf->buf.s + obuf->buf.len, buf, size);
   obuf->buf.len += size;
   return size;
}

/* works like fputc but to obuf */
int outbuf_putchar(outbuf* obuf, char ch) {
   if (outbuf_write(obuf, &ch, sizeof ch) <= 0) return -1;
   return ch;
}
```

outbuf.c

```
/* write contents of obuf to the associated fd */
bool outbuf_flush(outbuf* obuf) {
   ssize_t left = obuf->buf.len; ssize_t written = 0;
   while (left > 0) {
      ssize_t nbytes;
      do {
         errno = 0;
         nbytes = write(obuf->fd, obuf->buf.s + written, left);
      } while (nbytes < 0 && errno == EINTR);
      if (nbytes <= 0) return false;
      left -= nbytes; written += nbytes;
   }
   obuf->buf.len = 0;
   return true;
}

/* release storage associated with obuf */
void outbuf_free(outbuf* obuf) {
   stralloc_free(&obuf->buf);
}
```

- Zwischen Dienste-Anbietern und ihren Klienten auf dem Netzwerk besteht häufig ein ähnliches Verhältnis wie zwischen einer Shell und dem zugehörigen Benutzer.
- Der Klient gibt ein Kommando, das typischerweise mit dem Zeilentrenner CR LF, beendet wird, und der Dienste-Anbieter sendet darauf eine Antwort zurück,
  - die zum Ausdruck bringt, ob das Kommando erfolgreich verlief oder fehlschlug, und
  - einen Antworttext über eine oder mehrere Zeilen bringt.
- Es gibt keine zwingende Notwendigkeit, bei einem Protokoll Zeilentrenner zu verwenden. Alternativ wäre es auch denkbar,
  - die Länge eines Pakets zu Beginn explizit zu deklarieren oder
  - Pakete fester Länge zu wählen.

```
theon$ telnet mail.rz.uni-ulm.de smtp
Trying 134.60.1.11...
Connected to mail.uni-ulm.de.
Escape character is '^]'.
220 mail.uni-ulm.de ESMTP Sendmail 8.15.2/8.15.2; Tue, 11 Jun 2019 13:24:51 +0200 (CEST)
help
214-2.0.0 This is sendmail version 8.15.2
214-2.0.0 Topics:
214-2.0.0       HELO    EHLO    MAIL    RCPT    DATA
214-2.0.0       RSET    NOOP    QUIT    HELP    VRFY
214-2.0.0       EXPN    VERB    ETRN    DSN     AUTH
214-2.0.0       STARTTLS
214-2.0.0 For more info use "HELP <topic>".
214-2.0.0 To report bugs in the implementation see
214-2.0.0       http://www.sendmail.org/email-addresses.html
214-2.0.0 For local information send email to Postmaster at your site.
214 2.0.0 End of HELP info
huhu
500 5.5.1 Command unrecognized: "huhu"
helo theon.mathematik.uni-ulm.de
250 mail.uni-ulm.de Hello theon.mathematik.uni-ulm.de [134.60.66.7], pleased to meet you
quit
221 2.0.0 mail.uni-ulm.de closing connection
Connection to mail.uni-ulm.de closed by foreign host.
theon$
```

```
theon$ telnet mail.rz.uni-ulm.de smtp
Trying 134.60.1.11...
Connected to mail.uni-ulm.de.
Escape character is '^]'.
220 mail.uni-ulm.de ESMTP Sendmail 8.15.2/8.15.2; Tue, 11 Jun 2019 13:24:51 +0
```

- Beim SMTP-Protokoll erfolgt zunächst eine Begrüßung des Dienste-Anbieters.
- Die Begrüßung oder auch eine andere Antwort des Anbieters besteht aus einer dreistelligen Nummer, einem Leerzeichen oder einem Minus und beliebigem Text, der durch CR LF abgeschlossen wird.
- Die erste Ziffer der dreistelligen Nummer legt hier fest, ob ein Erfolg oder ein Problem vorliegt. Die beiden weiteren Ziffern werden zur feineren Unterscheidung der Rückmeldung verwendet.
- Eine führende 2 bedeutet Erfolg, eine 4 signalisiert ein temporäres Problem und eine 5 signalisiert einen permanenten Fehler.

```
help
214-2.0.0 This is sendmail version 8.15.2
214-2.0.0 Topics:
214-2.0.0       HELO    EHLO    MAIL    RCPT    DATA
214-2.0.0       RSET    NOOP    QUIT    HELP    VRFY
214-2.0.0       EXPN    VERB    ETRN    DSN     AUTH
214-2.0.0       STARTTLS
214-2.0.0 For more info use "HELP <topic>".
214-2.0.0 To report bugs in the implementation see
214-2.0.0       http://www.sendmail.org/email-addresses.html
214-2.0.0 For local information send email to Postmaster at your site.
214 2.0.0 End of HELP info
```

- In der Beispielsitzung ist das erste Kommando ein „help“, gefolgt von CR LF.
- Da die Antwort sich über mehrere Zeilen erstreckt, werden alle Zeilen, hinter der noch mindestens eine folgt, mit einem Minuszeichen hinter der dreistelligen Zahl gekennzeichnet.

```
huhu
500 5.5.1 Command unrecognized: "huhu"
helo theon.mathematik.uni-ulm.de
250 mail.uni-ulm.de Hello theon.mathematik.uni-
ulm.de [134.60.66.7], pleased to meet you
quit
221 2.0.0 mail.uni-ulm.de closing connection
Connection to mail.uni-ulm.de closed by foreign host.
theon$
```

- Das unbekannte Kommando „huhu“ provoziert hier eine Fehlermeldung provoziert, die durch den Code 500 als solche kenntlich gemacht wird.
- Das SMTP-Protokoll erlaubt auch eine Fortsetzung des Dialogs nach Fehlern, so dass dann noch ein „helo“-Kommando akzeptiert wurde.
- Die Verbindung wurde mit dem „quit“-Befehl beendet.