- Ein Netzwerkdienst ist ein Prozess, der unter einer Netzwerkadresse einen Dienst anbietet.
- Ein Klient, der die Netzwerkadresse kennt, kann einen bidirektionalen Kommunikationskanal zu dem Netzwerkdienst eröffnen und über diesen mit dem Dienst kommunizieren.
- Die Kommunikation wird durch ein Protokoll strukturiert, bei dem typischerweise Anfragen oder Kommandos auf dem Hinweg übermittelt werden und auf dem Rückweg des Kommunikationskanals die zugehörigen Antworten kommen.
- Wenn erst die Antwort gelesen werden muss, bevor die nächste Anfrage gestellt werden darf, wird von einem *synchronen* Protokoll gesprochen.
- Wenn mehrere Anfragen unmittelbar hintereinander gestellt werden dürfen, ohne dass erst die Antworten abgewartet werden, wird von *Pipelining* gesprochen. (Das hat nichts mit den Pipes aus dem vorherigen Kapitel zu tun.)

- Die beiden Kommunikationspartner müssen nicht miteinander verwandt sein.
- Sie müssen nicht einmal auf dem gleichen Rechner laufen.
- Da der Kommunikationskanal bidirektional ist, wird ein echter Dialog zwischen den beiden Prozessen möglich.
- Der Aufbau einer Verbindung ist jedoch schwieriger, da zunächst die Netzwerkadresse des gewünschten Partners ermittelt werden muss.

Wenn Dienste über das Netzwerk angeboten und in Anspruch genommen werden, ergeben sich viele Vorteile:

- Der Dienst kann allen offenstehen, und ein direkter Zugang zu dem Rechner, auf dem der Dienst angeboten wird, ist nicht notwendig.
- Viele Parteien können in kooperativer Weise einen Dienst gleichzeitig nutzen.
- Der Dienste-Anbieter hat weniger Last, da die Benutzerschnittstelle auf anderen Rechnern laufen kann.

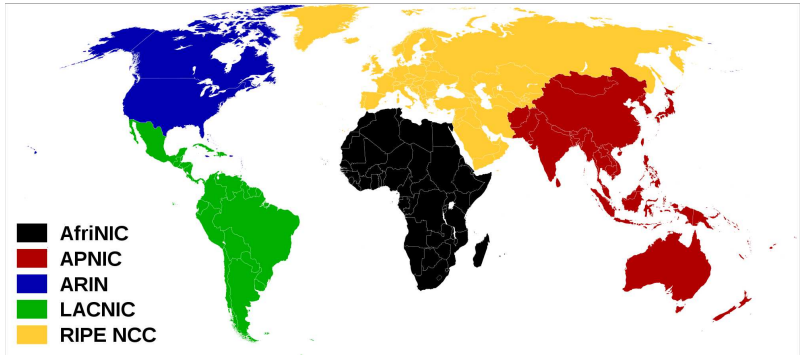
- Der Kreis derjenigen, die auf einen Netzwerkdienst zugreifen können, ist möglicherweise ziemlich umfangreich (normalerweise das gesamte Internet).
- Somit muss jeder Netzwerkdienst Zugriffsberechtigungen einführen und überprüfen und kann sich dabei nicht wie traditionelle Applikationen auf die des Betriebssystems verlassen.
- Dienste, die gleichzeitig von vielen genutzt werden können, haben vielerlei zusätzliche Konsistenz- und Synchronisierungsprobleme, für die nicht jede Art von Datenhaltung geeignet ist.
- Netzwerke bringen neue Arten von Ausfällen mit sich, wenn eine Netzwerkverbindung zusammenbricht oder es zu längeren „Hängern“ kommt.

- Im Rahmen dieser Vorlesung beschäftigen wir uns nur mit TCP/IP, also den verbindungsorientiertem Protokoll des Internets. (Mehr zur Semantik später.)
- Im Internet gibt es zwei etablierte Räume für Netzwerkadressen: IPv4 und IPv6.
- IPv4 arbeitet mit 32-Bit-Adressen und ist seit dem 1. Januar 1983 in Benutzung.
- Da der Adressraum bei IPv4 auszugehen droht, gibt es als Alternative IPv6, das mit 128-Bit-Adressen arbeitet.
- Im Rahmen dieser Vorlesung beschäftigen wir uns nur mit IPv4.
- Eine IPv4-Adresse (das gilt auch für IPv6) adressiert nur den Rechner, auf dem der Dienst läuft. Der Dienst selbst wird über eine Portnummer (16 Bit) ausgewählt.
- Ein Netzwerkdienst wird also z.B. über eine IPv4-Adresse und eine Port-Nummer adressiert.

```
clonard$ telnet 134.60.54.12 13
Trying 134.60.54.12...
Connected to 134.60.54.12.
Escape character is '^]'.
Mon Jun 14 11:03:16 2010
Connection to 134.60.54.12 closed by foreign host.
clonard$
```

- 134.60.54.12 ist eine IPv4-Adresse in der sogenannten *dotted-decimal*-Notation, bei der durch Punkte getrennt jedes der vier Bytes der Adresse einzeln dezimal spezifiziert wird.
- 134.60.54.12 ist also eine lesbarere Form für 2252092940.
- 13 ist die Port-Nummer des *daytime*-Dienstes.
- Die Port-Nummer ist nicht zufällig. Die 13 ist explizit von der IANA (*Internet Assigned Numbers Authority*) dem *daytime*-Dienst zugewiesen worden.

Aufteilung des Netzwerkraums



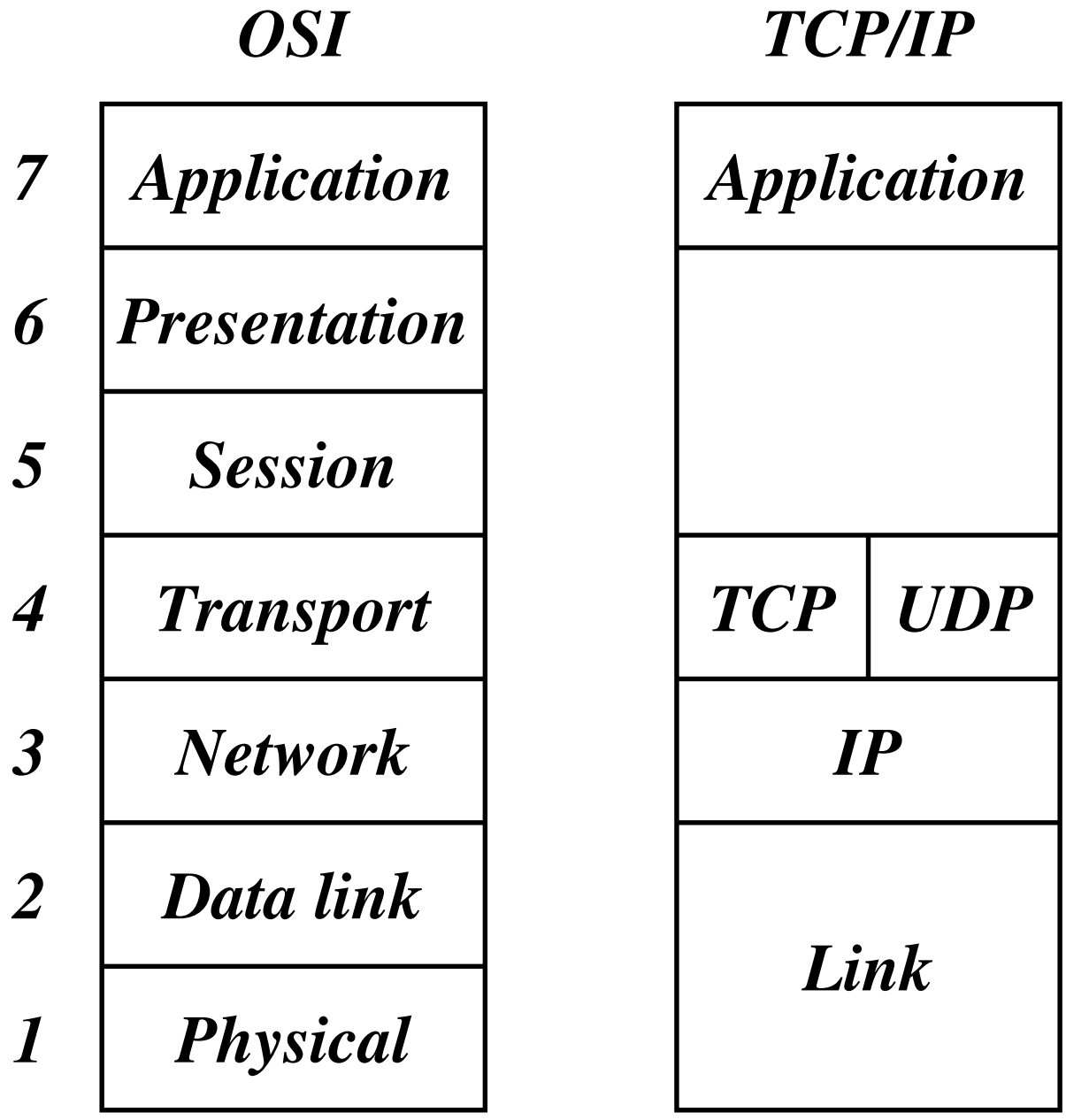
- Die IANA teilt den globalen IPv4-Adressraum auf einzelne lokale Institutionen, den sogenannten *Regional Internet Registries*.
- ARIN ist zuständig für Amerika, RIPE für Europa, den Mittleren Osten und Zentralasien, APNIC für Asien, Australien und Ozeanien, AfriNIC für Afrika und LACNIC für Lateinamerika einschließlich Teile der Karibik.
- Die Universität Ulm hat seit 1989 den Adressbereich *134.60.0.0/16*.

Wer ist zuständig für die 134.60.54.12? 171

```
theseus$ wget -O - -q http://www.iana.org/assignments/ipv4-address-space/ipv4-address-space.txt | \
> sed 's/  */ /g' | grep '^ 134'
 134/8 Administered by ARIN 1993-05 whois.arin.net LEGACY
theseus$ whois -h whois.arin.net 134.60.54.12
[...]
NetRange:   134.58.0.0 - 134.61.255.255
CIDR:       134.58.0.0/15, 134.60.0.0/15
NetName:    RIPE-ERX-134-58-0-0
NetHandle:  NET-134-58-0-0-1
Parent:     NET-134-0-0-0-0
NetType:    Early Registrations, Transferred to RIPE NCC
Comment:    These addresses have been further assigned to users in
Comment:    the RIPE NCC region.  Contact information can be found in
Comment:    the RIPE database at http://www.ripe.net/whois
[...]
theseus$ whois -h whois.ripe.net 134.60.54.12
[...]
inetnum:        134.60.0.0 - 134.60.255.255
netname:        UDN
descr:          Universitaet Ulm
descr:          Ulm, Germany
country:        DE
[...]
% Information related to '134.60.0.0/16AS553'

route:        134.60.0.0/16
descr:        UNI-ULM
origin:       AS553
mnt-by:       BELWUE-MNT
source:       RIPE # Filtered
[...]
```

- Für Rechnernamen wie *theseus.mathematik.uni-ulm.de* können über hierarchisierte Domain-Server die zugehörigen IP-Adressen abgefragt werden.
- Die Abfrage beginnt zuerst bei einem der 13 sogenannten Root-Server, die weltweit verteilt sind und deren IP-Adressen jedem Domain-Server bekannt sind.
- Einer davon ist *198.41.0.4*. Dieser verrät, welche Nameserver für die Top-Level-Domain *de* zuständig ist.
- Einer davon ist *194.0.0.53*. Dieser verrät, welche Nameserver für *uni-ulm.de* zuständig sind.
- Einer davon ist *134.60.1.111*, der sogleich in der Lage ist, diesen Namen vollständig aufzulösen und die *134.60.54.12* zurückzuliefern.


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

- IP-Adressen wie *134.60.54.12* 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 Theseus hat beispielsweise die Ethernet-Adresse *0:14:4f:3e:a1:f0* (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 weniger effizient ist.
- 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.

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

void print_ip_addr(in_addr_t ipaddr) {
   if (ipaddr == INADDR_ANY) {
      printf("INADDR_ANY");
   } else {
      uint32_t addr = ntohl(ipaddr);
      printf("%d.%d.%d.%d",
         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: %d\n", (int) 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.

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, sizeof 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;
   memmove((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.

Fragmentierung der Pakete bei Netzwerkverbindungen 193

- 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 194

- 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 195

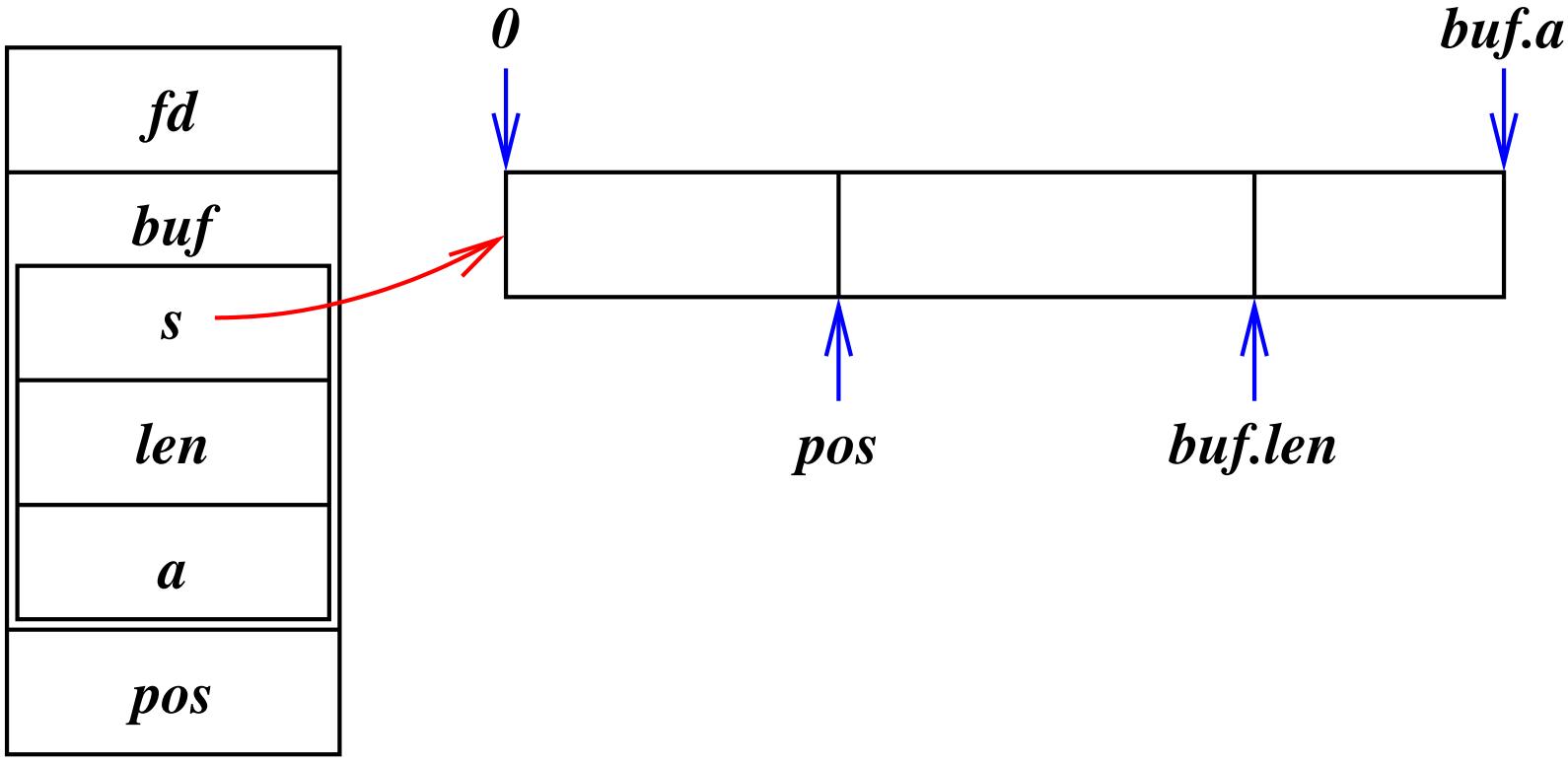
- Diese Vorgehensweise ist jedoch außerordentlich ineffizient, weil Systemaufrufe wie *read*() zu einem Kontextwechsel zwischen dem aufrufenden Prozess und dem Betriebssystem führen.
- Wenn ein Kontextwechsel für jedes einzulesende Byte initiiert wird, dann ist der betroffene Rechner mehr mit Kontextwechseln 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.
- 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 INBUF_H
#define INBUF_H

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

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

/* set size of input buffer */
int inbuf_alloc(inbuf* ibuf, unsigned int 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 */
int inbuf_back(inbuf* ibuf);

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

#endif
```

inbuf.c

```
/* set size of input buffer */
int inbuf_alloc(inbuf* ibuf, unsigned int 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;
}
```

Implementierung für eine gepufferte Eingabe

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 */
int inbuf_back(inbuf* ibuf) {
   if (ibuf->pos == 0) return 0;
   ibuf->pos--;
   return 1;
}

/* 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;
```

Schnittstelle für eine gepufferte Ausgabe

outbuf.h

```
#ifndef OUTBUF_H
#define OUTBUF_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 */
int 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 */
int 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 0;
      left -= nbytes; written += nbytes;
   }
   obuf->buf.len = 0;
   return 1;
}

/* 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.

Protokoll-Syntax am Beispiel von SMTP


```
clonard$ telnet mail.rz.uni-ulm.de smtp
Trying 134.60.1.11...
Connected to mail.rz.uni-ulm.de.
Escape character is '^]'.
220 mail.uni-ulm.de ESMTP Sendmail 8.14.2/8.14.2; Mon, 2 Jun 2008 10:18:51 +02
help
214-2.0.0 This is sendmail version 8.14.2
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 clonard.mathematik.uni-ulm.de
250 mail.uni-ulm.de Hello borchert@clonard.mathematik.uni-ulm.de [134.60.66.13
quit
221 2.0.0 mail.uni-ulm.de closing connection
Connection to mail.rz.uni-ulm.de closed by foreign host.
clonard$
```

```
clonard$ telnet mail.rz.uni-ulm.de smtp
Trying 134.60.1.11...
Connected to mail.rz.uni-ulm.de.
Escape character is '^]'.
220 mail.uni-ulm.de ESMTP Sendmail 8.14.2/8.14.2; Mon, 2 Jun 2008 10:18:51 +02
```

- 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 8.14.2
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 clonard.mathematik.uni-ulm.de
250 mail.uni-ulm.de Hello borchert@clonard.mathematik.uni-ulm.de [134.60.66.13
quit
221 2.0.0 mail.uni-ulm.de closing connection
Connection to mail.rz.uni-ulm.de closed by foreign host.
clonard$
```

- 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.

- Semaphore als Instrument zur Synchronisierung von Prozessen gehen auf den niederländischen Informatiker Edsger Dijkstra zurück, der diese Kontrollstruktur Anfang der 60er-Jahre entwickelte.
- Eine Semaphore wird irgendeiner Ressource zugeordnet, auf die zu einem gegebenen Zeitpunkt nur ein Prozess zugreifen darf, d.h. Zugriffe müssen exklusiv erfolgen.
- Damit sich konkurrierende Prozesse beim Zugriff auf die Ressource nicht ins Gehege kommen, erfolgt die Synchronisierung über Semaphore, die folgende Operationen anbieten:

  P Der Aufrufer wird blockiert, bis die Ressource frei ist. Danach ist ein Zugriff möglich.

  V Gib die Ressource wieder frei.

```
P(sema); // warte, bis die Semaphore fuer uns reserviert ist
// ... Kritischer Bereich, in dem wir exklusiven Zugang
// zu der mit sema verbundenen Ressource haben ...
V(sema); // Freigabe der Semaphore
```

- Semaphores werden so verwendet, dass jeder exklusive Zugriff auf eine Ressource in die Operationen P und V geklammert wird.
- Intern werden typischerweise Semaphore repräsentiert durch eine Datenstruktur mit einer ganzen Zahl und einer Warteschlange. Wenn die ganze Zahl positiv ist, dann ist die Semaphore frei. Ist sie 0, dann ist sie belegt, aber niemand sonst wartet darauf. Ist sie negativ, dann entspricht der Betrag der Länge der Warteschlange.
- Bei P wird entsprechend der Zähler heruntergezählt und, falls der Zähler negativ wurde, der Aufrufer in die Warteschlange befördert. Ansonsten erhält er sofort Zugang zur Ressource.
- Bei V wird der Zähler hochgezählt und, falls der Zähler noch nicht positiv ist, das am längsten wartende Mitglied der Warteschlange daraus entfernt und aufgeweckt.

Anmerkungen zu den Namen P und V, die beide auf Edsger Dijkstra zurückgehen:

- P steht für „Prolaag“ und V für „Verhoog“.
- „Verhoog“ ist niederländisch und bedeutet übersetzt „hochzählen“.
- Da das niederländische Gegenstück „verlaag“ (übersetzt: „herunterzählen“) ebenfalls mit einem „v“ beginnt, schuf Dijkstra das Kunstwort „prolaag“.
- Die erste Notiz, in der Dijkstra diese Operationen und die Namen P und V definierte, findet sich hier:
  http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD74.PDF
  Eine genaue Datierung liegt nicht vor, aber die Notiz muss wohl 1963 oder 1964 entstanden sein.
- 1968 erfolgte die erste Veröffentlichung in seinem Beitrag *Cooperating sequential processes* zur NATO-Konferenz über Programmiersprachen.

- Das *Mutual Exclusion Protocol* (MXP) sei ein Protokoll, das die Synchronisation einander fremder Prozesse über Semaphore erlaubt, die durch einen Netzwerkdienst verwaltet werden.
- Der Netzwerkdienst (in diesem Beispiel *mutexd* genannt) erlaubt beliebig viele Klienten, die sich jeweils namentlich identifizieren müssen.
- Jede der Klienten kann dann die bekannten P- und V-Operationen für beliebige Semaphoren absetzen oder den aktuellen Status einer Semaphore überprüfen.

- Das Protokoll sieht Anfragen (von einem Klienten an den Dienst) und Antworten (von dem Dienst an den Klienten) vor.
- Anfragen bestehen immer aus genau einer Zeile, die mit CR LF terminiert wird.
- Antworten bestehen aus einer oder mehrerer Zeilen, die ebenfalls mit CR LF terminiert werden.
- Die letzte Zeile einer Antwort beginnt immer mit dem Buchstaben „S“ oder „F“. „S“ steht für eine erfolgreich durchgeführte Operation, „F“ für eine fehlgeschlagene Operation.
- Wenn eine Antwort aus mehreren Zeilen besteht, dann beginnen alle Antwortzeilen mit Ausnahme der letzten Zeile mit dem Buchstaben „C“.

- Anfragen beginnen mit einer Folge von Kleinbuchstaben (dem Kommando), einem Leerzeichen und einem Parameter. Parameter sind beliebige Folgen von 8-Bit-Zeichen, die weder CR, LF noch Nullbytes enthalten dürfen.
- Antwortzeilen bestehen aus dem Statusbuchstaben („S“, „F“ oder „C“) und einer beliebigen Folge von 8-Bit-Zeichen, die weder CR, LF noch Nullbytes enthalten dürfen.

Folgende Anfragen werden unterstützt:

id *login*	Anmelden mit eindeutigem Namen. Dies muss als erstes erfolgen.
stat *sema*	Liefert den Status der genannten Semaphore. Wenn die Semaphore frei ist, wird „Sfree“ als Antwort zurückgeliefert. Ansonsten eine C-Zeile mit dem Namen desjenigen, der sie gerade reserviert hat, gefolgt von „Sheld“.
lock *sema*	Wartet, bis die Semaphore frei wird, und blockiert sie dann für den Aufrufer. Falls gewartet werden muss, gibt es sofort eine Antwortzeile „Cwaiting“. Sobald die Semaphore für den Aufrufer reserviert ist, folgt die Antwortzeile „Slocked“.
release *sema*	Gibt eine reservierte Semaphore wieder frei. Antwort ist ein einfaches „S“.

```
←  S
→  id alice
←  Swelcome
→  stat beer
←  Sfree
→  stat wine
←  Cbob
←  Sheld
→  lock beer
←  Slocked
→  lock wine
←  Cwaiting
←  Slocked
→  release wine
←  S
→  release cake
←  F
→  release beer
←  S
```

mxprequest.h

```
#ifndef MXP_REQUEST_H
#define MXP_REQUEST_H

#include <stdbool.h>
#include <stralloc.h>
#include <afblib/inbuf.h>
#include <afblib/outbuf.h>

typedef struct mxp_request {
   stralloc keyword;
   stralloc parameter;
} mxp_request;

/* read one request from the given input buffer */
bool read_mxp_request(inbuf* ibuf, mxp_request* request);

/* write one request to the given outbuf buffer */
bool write_mxp_request(outbuf* obuf, mxp_request* request);

/* release resources associated with request */
void free_mxp_request(mxp_request* request);

#endif
```

mxprequest.c

```
/* read one request from the given input buffer */
bool read_mxp_request(inbuf* ibuf, mxp_request* request) {
   return
      inbuf_scan(ibuf, "([a-z]+) ([^\r\n]*)\r\n",
         &request->keyword, &request->parameter) == 2;
}

/* write one request to the given outbuf buffer */
bool write_mxp_request(outbuf* obuf, mxp_request* request) {
   return
      outbuf_printf(obuf, "%.*s %.*s\r\n",
         request->keyword.len, request->keyword.s,
         request->parameter.len, request->parameter.s) > 0;
}

/* release resources associated with request */
void free_mxp_request(mxp_request* request) {
   stralloc_free(&request->keyword);
   stralloc_free(&request->parameter);
}
```

MXP-Pakete: Antworten

mxpresponse.h

```
#ifndef MXP_RESPONSE_H
#define MXP_RESPONSE_H

#include <stdbool.h>
#include <afblib/inbuf.h>
#include <afblib/outbuf.h>

typedef enum mxp_status {
   MXP_SUCCESS = 'S',
   MXP_FAILURE = 'F',
   MXP_CONTINUATION = 'C',
} mxp_status;

typedef struct mxp_response {
   mxp_status status;
   stralloc message;
} mxp_response;

/* write one (possibly partial) response to the given output buffer */
bool write_mxp_response(outbuf* obuf, mxp_response* response);

/* read one (possibly partial) response from the given input buffer */
bool read_mxp_response(inbuf* ibuf, mxp_response* response);

void free_mxp_response(mxp_response* response);

#endif
```

mxpresponse.c

```
bool read_mxp_response(inbuf* ibuf, mxp_response* response) {
   int ch = inbuf_getchar(ibuf);
   switch (ch) {
      case MXP_SUCCESS:
      case MXP_FAILURE:
      case MXP_CONTINUATION:
         response->status = ch;
         break;
      default:
         return false;
   }
   return inbuf_scan(ibuf, "([^\r\n]*)\r\n", &response->message) == 1;
}

bool write_mxp_response(outbuf* obuf, mxp_response* response) {
   return outbuf_printf(obuf, "%c%.*s\r\n", response->status,
      response->message.len, response->message.s) > 0;
}

void free_mxp_response(mxp_response* response) {
   stralloc_free(&response->message);
}
```

Es gibt vier Ansätze, um parallele Sitzungen zu ermöglichen:

- Für jede neue Sitzung wird mit Hilfe von *fork*() ein neuer Prozess erzeugt, der sich um die Verbindung zu genau einem Klienten kümmert.
- Für jede neue Sitzung wird ein neuer Thread gestartet.
- Sämtliche Ein- und Ausgabe-Operationen werden asynchron abgewickelt mit Hilfe von *aio_read*, *aio_write* und dem *SIGIO*-Signal.
- Sämtliche Ein- und Ausgabe-Operationen werden in eine Menge zu erledigender Operationen gesammelt, die dann mit Hilfe von *poll* oder *select* ereignis-gesteuert abgearbeitet wird.

Im Rahmen dieser Vorlesung betrachten wir nur die erste und die letzte Variante.

- Diese Variante ist am einfachsten umzusetzen und von genießt daher eine gewisse Popularität.
- Beispiele sind etwa der Apache-Webserver, der jede HTTP-Sitzung in einem separaten Prozess abhandelt, oder verschiedene SMTP-Server, die für jede eingehende E-Mail einen separaten Prozess erzeugen.
- Es gibt fertige Werkzeuge wie etwa *tcpserver* von Dan Bernstein, die die Socket-Operationen übernehmen und für jede Sitzung ein angegebenes Kommando starten, das mit der Netzwerkverbindung über die Standardein- und ausgabe verbunden ist.
- Es ist auch sinnvoll, das in Form einer kleinen Bibliotheksfunktion zu verpacken.

service.h

```
#ifndef AFBLIB_SERVICE_H
#define AFBLIB_SERVICE_H

#include <afblib/hostport.h>

typedef void (*session_handler)(int fd, int argc, char** argv);

/*
 * listen on the given port and invoke the handler for each
 * incoming connection
 */
void run_service(hostport* hp, session_handler handler,
   int argc, char** argv);

#endif
```

- *run_service* eröffnet eine Socket mit der über den Hostport spezifizierten Adresse und startet *handler* in einem separaten Prozess für jede neu eröffnete Sitzung. Diese Funktion läuft permanent und hört nur im Fehlerfalle auf.
- Wenn der *handler* beendet ist, terminiert der entsprechende Prozess.

- Problem: Wir haben konkurrierende Prozesse (für jede Sitzung einen), die eine gemeinsame Menge von Semaphoren verwalten.
- Prinzipiell könnten die das über ein Protokoll untereinander regeln oder den Systemaufrufen für Semaphore (die es auch gibt).
- In diesem Fallbeispiel wird eine primitive und uralte Technik eingesetzt:
  - Für jede Sitzung wird eine Datei angelegt, die nach dem jeweiligen Benutzer benannt wird.
  - Wer eine Semaphore reservieren möchte, versucht mit dem Systemaufruf *link* einen harten Link von der Datei zum Namen der Semaphore zu erzeugen. Da der Systemaufruf fehlschlägt, wenn der Zielname (der neue Link) bereits existiert, kann das maximal nur einem Prozess gelingen. Der hat dann den gewünschten exklusiven Zugriff.
  - Die anderen Prozesse verharren in einer Warteschleife und hoffen, dass irgendwann einmal die Semaphore wegfällt. Die primitive Lösung verwaltet keine Warteschlange.

Schnittstelle für die Semaphore

lockmanager.h

```
typedef struct lockset {
   char* dirname;
   char* myname;
   stralloc myfile;
   strhash locks;
} lockset;

/*
 * initialize lock set
 */
int lm_init(lockset* set, char* dirname, char* myname);

/* release all locks associated with set and allocated storage */
void lm_free(lockset* set);

/*
 * check status of the given lock and return
 * the name of the holder in holder if it's held
 * and an empty string if the lock is free
 */
int lm_stat(lockset* set, char* lockname, stralloc* holder);

/* block until 'lockname' is locked */
int lm_lock(lockset* set, char* lockname);

/* attempt to lock 'lockname' but do not block */
int lm_nonblocking_lock(lockset* set, char* lockname);

/* release 'lockname' */
int lm_release(lockset* set, char* lockname);
```

service.c

```
void run_service(hostport* hp, session_handler handler,
      int argc, char** argv) {
   int sfd = socket(hp->domain, SOCK_STREAM, hp->protocol);
   int optval = 1;
   if (sfd < 0 ||
        setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR,
                   &optval, sizeof optval) < 0 ||
        bind(sfd, (struct sockaddr *) &hp->addr,
             hp->namelen) < 0 ||
        listen(sfd, SOMAXCONN) < 0) {
      return;
   }

   /* our childs shall not become zombies */
   struct sigaction action = {
      .sa_handler = SIG_IGN,
      .sa_flags = SA_NOCLDWAIT,
   };
   if (sigaction(SIGCHLD, &action, 0) < 0) return;

   /* ... accept incoming connections ... */
}
```

service.c

```
int fd;
while ((fd = accept(sfd, 0, 0)) >= 0) {
   pid_t child = fork();
   if (child == 0) {
      close(sfd);
      handler(fd, argc, argv);
      exit(0);
   }
   close(fd);
}
```

- Der übergeordnete Prozess wartet mit *accept* auf die jeweils nächste eingehende Netzwerkverbindung.
- Sobald eine neue Verbindung da ist, wird diese mit *fork* an einen neuen Prozess übergeben, der dann *handler* aufruft. Diese Funktion kümmert sich dann nur noch um eine einzelne Sitzung.

mutexd.c

```
#include <stdio.h>
#include <stdlib.h>
#include <afblib/hostport.h>
#include <afblib/service.h>
#include "mxpsession.h"

int main (int argc, char** argv) {
   char* cmdname = *argv++; --argc;
   if (argc != 2) {
      fprintf(stderr, "Usage: %s hostport lockdir\n", cmdname);
      exit(1);
   }
   char* hostport_string = *argv++; --argc;
   hostport hp;
   if (!parse_hostport(hostport_string, &hp, 21021)) {
      fprintf(stderr, "%s: hostport in conformance to RFC 2396 expected\n",
         cmdname);
      exit(1);
   }

   /* pass lockdir argument to the service */
   run_service(&hp, mxp_session, argc, argv);
}
```

MXP-Dienst auf Basis von *run_service*

mxpsession.c

```
#define EQUAL(sa,str) (strncmp((sa.s),(str),(sa.len)) == 0)

void mxp_session(int fd, int argc, char** argv) {
   if (argc != 1) return;
   char* lockdir = argv[0];

   inbuf ibuf = {fd};
   outbuf obuf = {fd};
   lockset locks = {0};

   /* send greeting */
   mxp_response greeting = {MXP_SUCCESS};
   if (!write_mxp_response(&obuf, &greeting)) return;
   if (!outbuf_flush(&obuf)) return;

   /* ... rest of the session ... */

   /* release all locks */
   lm_free(&locks);
   /* free allocated memory */
   free_mxp_response(&response);
   stralloc_free(&myname);
}
```

mxpsession.c

```
/* receive identification */
mxp_request id = {{0}};
if (!read_mxp_request(&ibuf, &id)) return;
if (!EQUAL(id.keyword, "id")) return;
stralloc myname = {0};
stralloc_copy(&myname, &id.parameter);
stralloc_0(&myname);
int ok = lm_init(&locks, lockdir, myname.s);

/* send response to identification */
mxp_response response = {MXP_SUCCESS};
stralloc_copys(&response.message, "welcome");
if (!ok) response.status = MXP_FAILURE;
if (!write_mxp_response(&obuf, &response)) return;
if (!outbuf_flush(&obuf)) return;
if (!ok) return;
```

mxpsession.c

```
/* process regular requests */
mxp_request request = {{0}};
while (read_mxp_request(&ibuf, &request)) {
   stralloc lockname = {0};
   stralloc_copy(&lockname, &request.parameter);
   stralloc_0(&lockname);

   if (EQUAL(request.keyword, "stat")) {
      /* ... handling of stat ... */
   } else if (EQUAL(request.keyword, "lock")) {
      /* ... handling of lock ... */
   } else if (EQUAL(request.keyword, "release")) {
      /* ... handling of release */
   } else {
      response.status = MXP_FAILURE;
      stralloc_copys(&response.message, "unknown command");
   }
   if (!write_mxp_response(&obuf, &response)) break;
   if (!outbuf_flush(&obuf)) break;
}
```

mxpsession.c

```
      if (EQUAL(request.keyword, "stat")) {
         mxp_response info = {MXP_CONTINUATION};
         if (lm_stat(&locks, lockname.s, &info.message)) {
            response.status = MXP_SUCCESS;
            if (info.message.len == 0) {
               stralloc_copys(&response.message, "free");
            } else {
               if (!write_mxp_response(&obuf, &info)) break;
               stralloc_copys(&response.message, "held");
            }
         } else {
            response.status = MXP_FAILURE;
            stralloc_copys(&response.message,
               "unable to check lock status");
         }
         free_mxp_response(&info);
```

mxpsession.c

```
    } else if (EQUAL(request.keyword, "lock")) {
       if (lm_nonblocking_lock(&locks, lockname.s)) {
          response.status = MXP_SUCCESS;
          stralloc_copys(&response.message, "locked");
       } else {
          mxp_response notification = {MXP_CONTINUATION};
          stralloc_copys(&notification.message, "waiting");
          if (!write_mxp_response(&obuf, &notification)) break;
          if (!outbuf_flush(&obuf)) break;
          if (lm_lock(&locks, lockname.s)) {
             response.status = MXP_SUCCESS;
             stralloc_copys(&response.message, "locked");
          } else {
             response.status = MXP_FAILURE;
             stralloc_copys(&response.message, "");
          }
       }
    } else if (EQUAL(request.keyword, "release")) {
       stralloc_copys(&response.message, "");
       if (lm_release(&locks, lockname.s)) {
          response.status = MXP_SUCCESS;
       } else {
          response.status = MXP_FAILURE;
       }
```

- Wenn es um sehr schnelle Reaktionen auf eingehende Verbindungen ankommt, erscheint u.U. die Sequenz von *accept* und *fork* zu langsam.
- Alternativ ist es auch denkbar, den Netzwerkdienst zuerst mit *socket*, *bind* und *listen* aufzusetzen und dann mehrere Prozesse im Voraus mit *fork* zu erzeugen, die alle die Socket erben.
- Dann kann jeder dieser Prozesse konkurrierend *accept* aufrufen. Wenn dann eine Netzwerkverbindung durch einen Klienten eröffnet wird, dann ist genau einer der *accept*-Aufrufe erfolgreich. Die anderen Prozesse warten weiter auf andere Klienten.
- Das Modell ist insbesondere durch den Apache-Webserver bekannt geworden.

- Die Zahl der Prozesse, die mit dem Prefork-Modell erzeugt worden ist, begrenzt zunächst die Zahl der parallelen Sitzungen. Das ist nicht befriedrigend.
- Es müssen also bei Bedarf weitere Prozesse erzeugt werden. Aber wie bekommt der Hauptprozess mit, wieviele Prozesse noch frei sind, um eine Verbindung entgegenzunehmen?
- Signale sind ungeeignet, da die sich gegenseitig auslöschen können. Es wird also irgendeine Interprozesskommunikation benötigt. Hierfür bieten sich u.a. Pipelines an, da die leicht vererbt werden können.
- Das bedeutet aber, dass der Hauptprozess mehrere Pipelines unter Beobachtung halten muss. Das ist mit *poll* denkbar.
- Wie können die Prozesse alle abgebaut werden? Wenn der Hauptprozess mit *SIGTERM* terminiert wird, sollten die anderen Prozesse, die nur auf Sitzungen warten, folgen. Bestehende Sitzungen sollten aber nicht unterbrochen werden.

- Dieses Modell kommt noch ohne *poll* aus.
- Zu Beginn wird die gewünschte Zahl von Prozessen erzeugt.
- Jeder der erzeugten Prozesse (Kind-Prozess) legt eine Pipeline an und erzeugt einen weiteren Prozess (Enkel-Prozess), der die Pipeline zum Schreiben offenlässt, während der Erzeuger aus der Pipeline nur liest.
- Der Enkel-Prozess ruft dann *accept* auf, um auf eine eingehende Verbindung zu warten. Sobald *accept* erfolgreich ist, wird die Pipeline geschlossen und die Sitzung gestartet.
- Der Kind-Prozess liest aus der Pipeline und wird damit blockiert, bis der Enkel-Prozess die Pipeline schließt. Danach kann ein neuer Enkel-Prozess erzeugt werden.
- Sollte einer der Kind-Prozesse terminieren, wird vom Hauptprozess ein Nachfolger erzeugt.
- Vorteil: Es sind immer n Prozesse bereit, eine Sitzung entgegenzunehmen. Nachteil: Wir benötigen insgesamt $2n + 1$ Prozesse.

preforked_service.c

```
void run_preforked_service(hostport* hp, session_handler handler,
      unsigned int number_of_processes, int argc, char** argv) {
   assert(number_of_processes > 0);
   int sfd = socket(hp->domain, SOCK_STREAM, hp->protocol);
   int optval = 1;
   if (sfd < 0 ||
        setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR,
                   &optval, sizeof optval) < 0 ||
        bind(sfd, (struct sockaddr *) &hp->addr, hp->namelen) < 0 ||
        listen(sfd, SOMAXCONN) < 0) {
      close(sfd);
      return;
   }

   /* ... setup termination handler ... */
   /* ... create preforked processes ... */
   /* ... start a new preforked process for every one terminating ... */
   /* ... terminate everything ... */
}
```

preforked_service.c

```
/* setup termination handler */
struct sigaction action = {
   .sa_handler = termination_handler,
};
if (sigaction(SIGTERM, &action, 0) != 0) {
   return;
}

/* create preforked processes */
pid_t child_pid[number_of_processes];
for (int i = 0; i < number_of_processes; ++i) {
   pid_t pid = spawn_preforked_process(sfd, handler, argc, argv);
   if (pid < 0) return;
   child_pid[i] = pid;
}
```

preforked_service.c

```
/* start a new preforked process for every one terminating */
while (!terminate) {
   pid_t child; int wstat;
   if ((child = wait(&wstat)) > 0) {
      int index;
      for (index = 0; index < number_of_processes; ++index) {
         if (child_pid[index] == child) break;
      }
      if (index < number_of_processes) {
         child = spawn_preforked_process(sfd, handler, argc, argv);
         child_pid[index] = child;
         if (child < 0) break;
      }
   }
}

/* terminate everything */
for (int i = 0; i < number_of_processes; ++i) {
   if (child_pid[i] > 0) {
      kill(child_pid[i], SIGTERM);
   }
}
```

preforked_service.c

```
static pid_t spawn_preforked_process(int sfd, session_handler handler,
      int argc, char** argv) {
   pid_t child = fork();
   if (child) return child;

   /* our childs shall not become zombies */
   struct sigaction action = {
      .sa_handler = SIG_IGN,
      .sa_flags = SA_NOCLDWAIT,
   };
   if (sigaction(SIGCHLD, &action, 0) < 0) exit(1);

   while (!terminate) {
      /* ... */
   }
   exit(0);
}
```

Einfaches Prefork-Modell

preforked_service.c

```
while (!terminate) {
   /* now create another process and share a pipeline with it */
   int pipe_fds[2];
   if (pipe(pipe_fds) < 0) exit(1);
   pid_t pid = fork();
   if (pid < 0) exit(1);
   if (pid == 0) {
      /* grandchild of the original process */
      close(pipe_fds[0]); /* close reading side of pipe */
      int fd = accept(sfd, 0, 0);
      close(sfd);
      if (fd < 0) exit(1);
      /* now close the writing side of the pipe to indicate that
         we are busy with running a session */
      close(pipe_fds[1]);
      /* run the session and exit */
      handler(fd, argc, argv);
      exit(0);
   }
   close(pipe_fds[1]); /* close writing side of the pipe */
   /* now wait for the child process to accept a connection;
      we get notified by the closure of the pipe */
   char ch;
   if (read(pipe_fds[0], &ch, 1) < 0 && errno == EINTR && terminate) {
      kill(pid, SIGTERM); /* propagate termination */
   }
   close(pipe_fds[0]);
}
```

- In manchen Fällen ist es vorteilhaft, wenn alle Sitzungen einen gemeinsamen Adressraum verwenden, damit sitzungsübergreifende Datenstrukturen leichter verwaltet werden können.
- Prinzipiell lässt sich das mit Hilfe des Systemaufrufs *poll* erreichen, mit dem auf das Eintreten eines Ein- oder Ausgabe-Ereignisses gewartet werden kann.
- Dies führt zu einem grundlegend anderen Programmierstil, bei dem Ein- und Ausgaben ereignisgesteuert abgewickelt werden.
- Da bei jedem Ereignis entsprechende Behandler neu aufgerufen werden, kann der Sitzungskontext nicht in lokalen Variablen verwaltet werden. Stattdessen sind dafür dynamische Datenstrukturen zu verwenden, die bei jedem Aufruf erst lokalisiert werden müssen.

multiplexor.c

```
    if (poll(mpx.pollfds, count, -1) <= 0) return;
```

- *poll* erhält drei Parameter:
  - Einen Zeiger auf ein Array mit Einträgen des Datentyps **struct** *pollfd*,
  - einer natürlichen Zahl, die die Länge des Arrays angibt, und
  - einer zeitlichen Beschränkung in Millisekunden. (Hier wird -1 angegeben, wenn keine Befristung gewünscht wird.)
- Der Datentyp **struct** *pollfd* umfasst folgende Felder:

fd	Dateideskriptor
events	Menge der Ereignisse, auf die gewartet wird
revents	Menge der Ereignisse, die eingetreten sind

- Im Erfolgsfalle liefert *poll* die Zahl der eingetretenen Ereignisse zurück. Falls die zeitliche Beschränkung erreicht wurde, ohne dass eines der Ereignisse eintrat, wird 0 zurückgeliefert. Im Falle von Fehlern wird -1 zurückgegeben.

- Relevant sind nur *POLLIN* und *POLLOUT*. Prinzipiell kann *poll* noch Unterscheidungen treffen, ob priorisierte Pakete über die Netzwerkverbindung ankamen, aber das wird normalerweise nicht verwendet.
- Das Ereignis *POLLIN* bedeutet, dass ein *read*-Systemaufruf für den Dateideskriptor abgesetzt werden kann, ohne dass der Prozess blockiert wird.
- Analog bedeutet *POLLOUT*, dass ein *write*-Systemaufruf ohne die Gefahr eines Blocks abgesetzt werden kann.
- Bei mit *listen* vorbereiteten Sockets kann ebenfalls *POLLIN* verwendet werden. Das Ereignis tritt dann ein, sobald sich eine neue Netzwerkverbindung anbahnt und *accept* blockierungsfrei aufgerufen werden kann.

Verpackung von *poll* in eine Bibliothek

multiplexor.h

```
typedef void (*multiplexor_handler)(connection* link);
void run_multiplexor(int socket, multiplexor_handler open_handler,
   multiplexor_handler input_handler, multiplexor_handler close_handler,
   void* mpx_handle);
bool write_to_link(connection* link, char* buf, unsigned int len);
ssize_t read_from_link(connection* link, char* buf, unsigned int len);
void close_link(connection* link);
```

- Es ist sinnvoll, die Verwendung von *poll* in eine geeignete Bibliothek zu verpacken.
- Die Funktion *run_multiplexor* läuft dann permanent und übernimmt somit die vollständige Kontrolle des Programms. Es werden nur noch Behandler aufgerufen, wenn
  - neue Netzwerkverbindungen eröffnet werden,
  - neue Eingaben vorliegen oder
  - eine Verbindung beendet wird.
- Eine Rückkehr von *run_multiplexor* gibt es nur im Fehlerfalle.

multiplexor.h

```
typedef void (*multiplexor_handler)(connection* link);
void run_multiplexor(int socket, multiplexor_handler open_handler,
   multiplexor_handler input_handler, multiplexor_handler close_handler,
   void* mpx_handle);
bool write_to_link(connection* link, char* buf, unsigned int len);
ssize_t read_from_link(connection* link, char* buf, unsigned int len);
void close_link(connection* link);
```

- Konkret ruft *run_multiplexor* den Behandler *open_handler* für neue Verbindungen, *input_handler* für neue Eingaben und *close_handler* für beendete Verbindungen auf.
- Die Behandler dürfen selbst nichts direkt auf eine Netzwerkverbindung ausgeben, da dies zu längeren Blockaden führen könnte. Stattdessen muss dies durch *write_to_link* erfolgen, das dafür Warteschlangen unterhält.
- Der Parameter *mpx_handle* dient als Zeiger auf eine eigene Datenstruktur, die den Behandlern unter *connection−>mpx_handle* zur Verfügung gestellt wird.

Datenstruktur für eine Netzwerkverbindung

multiplexor.h

```
typedef struct connection {
   int fd;
   void* handle; /* may be freely used by the application */
   void* mpx_handle; /* corresponding parameter from run_multiplexor */
   bool eof;
   struct output_queue_member* oqhead;
   struct output_queue_member* oqtail;
   struct connection* next;
   struct connection* prev;
} connection;
```

- Für jede Netzwerkverbindung gibt es eine zugehörige Datenstruktur.
- Neben der Netzwerkverbindung *fd* und den beiden benutzerdefinierten Zeigern *handle* und *mpx_handle*, kommen noch folgende Felder hinzu:

eof	wird auf *true* gesetzt, sobald ein Eingabeende erkannt wurde
oqhead und *oqtail*	Zeiger auf das erste und letzte Element der Warteschlange mit den auszugebenden Puffern
next und *prev*	doppelt verkettete Liste aller Netzwerkverbindungen

multiplexor.c

```
typedef struct output_queue_member {
   char* buf;
   unsigned int len;
   unsigned int pos;
   struct output_queue_member* next;
} output_queue_member;
// ...
int write_to_link(connection* link, char* buf, unsigned int len);
```

- Jedes Element der Warteschlange weist auf einen Puffer.
- Zu Beginn ist die Position *pos* gleich 0 und *len* entspricht der Länge, die an *write_to_link* übergeben worden ist.
- Wenn jedoch der entsprechende Aufruf von *write* nicht vollständig umgesetzt werden kann, dann wird *pos* um die übertragene Quantität erhöht und *len* entsprechend gesenkt.
- Sobald die Schreiboperation abgeschlossen ist, wird nicht nur das Warteschlangen-Element, sondern auch der Puffer freigegeben.

multiplexor.c

```
typedef struct multiplexor {
   /* parameters passed to run_multiplexor */
   int socket;
   multiplexor_handler ohandler, ihandler, chandler;
   void* mpx_handle;
   /* additional administrative fields */
   bool socketok; /* becomes false when accept() fails */
   connection* head; /* double-linked linear list of connections */
   connection* tail; /* its last element */
   int count; /* number of connections */
   struct pollfd* pollfds; /* parameter for poll() */
   unsigned int pollfdslen; /* allocated len of pollfds */
   connection** pollcs; /* of the same len as pollfds */
} multiplexor;
```

- Es gibt nur ein Objekt dieser Datenstruktur, das von *run_multiplexor* zu Beginn angelegt wird.
- Neben den Parametern von *run_multiplexor* werden in der doppelt verketteten Liste mit *head* und *tail* alle offenen Verbindungen verwaltet. In *count* findet sich deren Zahl.
- *pollfds* zeigt auf ein dynamisch belegtes Feld mit *pollfdslen* Elementen. Dies dient der Verwaltung der *poll* zu übergebenden Datenstruktur.

Vorbereitung der Datenstruktur für *poll*

multiplexor.c

```
/* prepare fields pollfds and pollfdslen in mpx in
   dependence of the current set of connections */
static int setup_polls(multiplexor* mpx) {
   int len = mpx->count;
   if (mpx->socketok) ++len;
   if (len == 0) return 0;
   /* weed out links which have been closed
      and where our output queue is empty */
   connection* link = mpx->head;
   while (link) {
      connection* next = link->next;
      if (link->eof && link->oqhead == 0) remove_link(mpx, link);
      link = next;
   }
   /* allocate or enlarge pollfds, if necessary */
   if (mpx->pollfdslen < len) {
      mpx->pollfds = realloc(mpx->pollfds, sizeof(struct pollfd) * len);
      if (mpx->pollfds == 0) return 0;
      mpx->pollcs = realloc(mpx->pollcs, sizeof(connection*) * len);
      if (mpx->pollcs == 0) return 0;
      mpx->pollfdslen = len;
   }

   /* ... */
}
```

Vorbereitung der Datenstruktur für *poll*

multiplexor.c

```
/* prepare fields pollfds and pollfdslen in mpx in
   dependence of the current set of connections */
static int setup_polls(multiplexor* mpx) {
   /* ... */

   int index = 0;
   /* look for new network connections as long accept()
      returned no errors so far */
   if (mpx->socketok) {
      mpx->pollcs[index] = 0;
      mpx->pollfds[index++] = (struct pollfd) {mpx->socket, POLLIN};
   }
   /* look for incoming network connections and
      check whether we can write any pending output packets
      without blocking */
   link = mpx->head;
   while (link) {
      short events = 0;
      if (!link->eof) events |= POLLIN;
      if (link->oqhead) events |= POLLOUT;
      mpx->pollcs[index] = link;
      mpx->pollfds[index++] = (struct pollfd) {link->fd, events};
      link = link->next;
   }
   return index;
}
```

multiplexor.c

```
static bool add_connection(multiplexor* mpx) {
   int newfd;
   if ((newfd = accept(mpx->socket, 0, 0)) < 0) {
      mpx->socketok = false; return true;
   }
   connection* link = malloc(sizeof(connection));
   if (link == 0) return false;
   *link = (connection) {
      .fd = newfd, .handle = 0, .mpx = mpx, .mpx_handle = mpx->mpx_handle,
      .eof = false, .oqhead = 0, .oqtail = 0, .next = 0, .prev = mpx->tail,
   };
   if (mpx->tail) {
      mpx->tail->next = link;
   } else {
      mpx->head = link;
   }
   mpx->tail = link; ++mpx->count;
   if (mpx->ohandler) (*mpx->ohandler)(link);
   return true;
}
```

Entfernung einer Verbindung

multiplexor.c

```
/* remove a connection from the double-linked linear
   list of connections
*/
static void remove_link(multiplexor* mpx, connection* link) {
   close(link->fd);
   if (link->prev) {
      link->prev->next = link->next;
   } else {
      mpx->head = link->next;
   }
   if (link->next) {
      link->next->prev = link->prev;
   } else {
      mpx->tail = link->prev;
   }
   if (mpx->chandler) (*mpx->chandler)(link);
   free(link);
   --mpx->count;
}
```

multiplexor.c

```
/* read one input packet from the given network connection */
ssize_t read_from_link(connection* link, char* buf, unsigned int len) {
   if (link->eof) return 0;
   ssize_t nbytes = read(link->fd, buf, len);
   if (nbytes <= 0) {
      link->eof = true;
      if (link->oqhead == 0) remove_link((multiplexor*)link->mpx, link);
   }
   return nbytes;
}
```

- Wenn *poll* signalisiert hat, dass wir von einer Verbindung einlesen dürfen, dann wird der entsprechende Behandler aufgerufen, der wiederum *read_from_link* aufruft, um die Eingabe in den eigenen Puffer einzulesen.

multiplexor.c

```
/* write one pending output packet to the given network connection */
static void write_to_socket(multiplexor* mpx, connection* link) {
   ssize_t nbytes = write(link->fd,
      link->oqhead->buf + link->oqhead->pos,
      link->oqhead->len - link->oqhead->pos);
   if (nbytes <= 0) {
      remove_link(mpx, link);
   } else {
      link->oqhead->pos += nbytes;
      if (link->oqhead->pos == link->oqhead->len) {
         output_queue_member* old = link->oqhead;
         link->oqhead = old->next;
         if (link->oqhead == 0) {
            link->oqtail = 0;
         }
         free(old->buf); free(old);
         if (link->oqhead == 0 && link->eof) {
            remove_link(mpx, link);
         }
      }
   }
}
```

Schreiben auf eine Verbindung

multiplexor.c

```
bool write_to_link(connection* link, char* buf, unsigned int len) {
   assert(len >= 0);
   if (len == 0) {
      free(buf); return true;
   }
   output_queue_member* member = malloc(sizeof(output_queue_member));
   if (!member) return false;
   member->buf = buf; member->len = len; member->pos = 0;
   member->next = 0;
   if (link->oqtail) {
      link->oqtail->next = member;
   } else {
      link->oqhead = member;
   }
   link->oqtail = member;
   return true;
}
```

- Diese Funktion ist von den Behandlern aufzurufen, wenn etwas auf eine der Netzwerkverbindungen auszugeben ist.
- Der Ausgabepuffer wird dann in die entsprechende Warteschlange eingereiht.

multiplexor.c

```
void close_link(connection* link) {
   link->eof = 1;
   shutdown(link->fd, SHUT_RD);
}
```

- Bei bidirektionalen Netzwerkverbindungen ist es möglich, nur eine Seite zu schließen.
- Dies geht nicht mit *close*, das sofort beide Seiten schließen würde, sondern mit *shutdown*, mit dem eine spezifizierte Seite geschlossen werden kann.
- Hier wird aus der Sicht des Aufrufers die lesendende Seite geschlossen, also die Verbindung vom Klienten zum Dienst. Danach können keine weiteren Anfragen mehr eintreffen, aber die Warteschlange der abzuarbeitenden Ausgabe-Puffer kann noch abgearbeitet werden.
- Erst wenn die Warteschlange ganz leer ist, dann wird (von *remove_link*) die Verbindung vollständig geschlossen.

multiplexor.c

```
void run_multiplexor(int socket, multiplexor_handler open_handler,
      multiplexor_handler input_handler, multiplexor_handler close_handler,
      void* mpx_handle) {
   multiplexor mpx = {
      .socket = socket, .ohandler = open_handler,
      .ihandler = input_handler, .chandler = close_handler,
      .mpx_handle = mpx_handle, .socketok = true,
   };
   int count;
   while ((count = setup_polls(&mpx)) > 0) {
      if (poll(mpx.pollfds, count, -1) <= 0) return;
      for (int index = 0; index < count; ++index) {
         if (mpx.pollfds[index].revents == 0) continue;
         int fd = mpx.pollfds[index].fd;
         if (fd == mpx.socket) {
            if (!add_connection(&mpx)) return;
         } else {
            connection* link = mpx.pollcs[index]; assert(link);
            if (mpx.pollfds[index].revents & POLLIN) {
               (*mpx.ihandler)(link);
            }
            if (mpx.pollfds[index].revents & POLLOUT) {
               write_to_socket(&mpx, link);
            }
         }
      }
   }
}
```

- Der *input_handler* wird für jedes eingehende Paket aufgerufen.
- Da Pakete fragmentiert sein können, sind dies möglicherweise Bruchstücke einer Anfrage oder auch Teile mehrerer Anfragen.
- Entsprechend muss die Eingabe wieder gepuffert und zerlegt werden, da normalerweise eine Reaktion erst bei einer vollständig übermittelten Anfrage erfolgen sollte.
- Eine ereignisgesteuerte Behandlung wäre daher aus Anwendungssicht leichter zu programmieren, wenn sie auf vollständigen Anfragen beruhen würde.
- Die Erkennung einer vollständigen Anfrage ist im allgemeinen Fall nicht ganz trivial zu spezifizieren. Im folgenden wird eine Lösung auf Basis regulärer Ausdrücke vorgestellt, die für textbasierte Protokolle gut geeignet ist.

mpx_session.h

```
typedef void (*mpx_handler)(session* s);

int mpx_session_scan(session* s, ...);
int mpx_session_printf(session* s, const char* restrict format, ...);
void close_session(session* s);

void run_mpx_service(hostport* hp, const char* regexp,
   mpx_handler ohandler, mpx_handler rhandler, mpx_handler hhandler,
   void* global_handle);
```

- *run_mpx_service* erhält einen regulären Ausdruck, der eine Anfrage spezifiziert.
- Dieser reguläre Ausdruck darf mit Hilfe runder Klammern beliebig viele Elemente der Anfrage herausgreifen – analog zu *inbuf_scan*.
- Der *rhandler* (*request handler*) wird dann für jede vollständig vorliegende Anfrage aufgerufen und kann dann mit *mpx_session_scan* die herausgegriffenen Elemente in *stralloc*-Objekte hineinkopieren lassen.

mxprequest.h

```
#define MXP_REQUEST_RE "([a-z]+) (.*?)\r?\n"
```

mutexd.c

```
   run_mpx_service(&hp, MXP_REQUEST_RE,
      mxp_session_open, mxp_session_read, mxp_session_hangup,
      locks);
```

- Beim Aufruf von *run_mpx_service* wird der reguläre Ausdruck zum Erkennen einer Anfrage mitgegeben.

mxpsession.c

```
void mxp_session_read(session* s) {
   struct mxp_session* ms = s->handle; assert(ms);
   if (!read_mxp_request(s, &ms->request)) {
      close_session(s); return;
   }
   /* ... process request and generate response ... */
   if (!write_mxp_response(s, &ms->response)) {
      close_session(s);
   }
}
```

- Der Behandler *mxp_session_read* wird jetzt nur aufgerufen, wenn eine vollständige Anfrage vorliegt. Entsprechend sollte *read_mxp_request* eine Anfrage einlesen können.

mxprequest.c

```
bool read_mxp_request(session* s, mxp_request* request) {
   return
      mpx_session_scan(s, &request->keyword, &request->parameter) == 2;
}
```

mxresponse.c

```
/* write one (possibly partial) response to */
bool write_mxp_response(session* s, mxp_response* response) {
   return mpx_session_printf(s, "%c%.*s\r\n", response->status,
      response->message.len, response->message.s) > 0;
}
```

- Die Einlese-Operation für Anfragen und die Ausgabe-Operation für Antworten verwenden hier die entsprechenden Funktionen aus *mpx_session.h*