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

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

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;
   int 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.

Fragmentierung der Pakete bei Netzwerkverbindungen 196

- 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 197

- 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 198

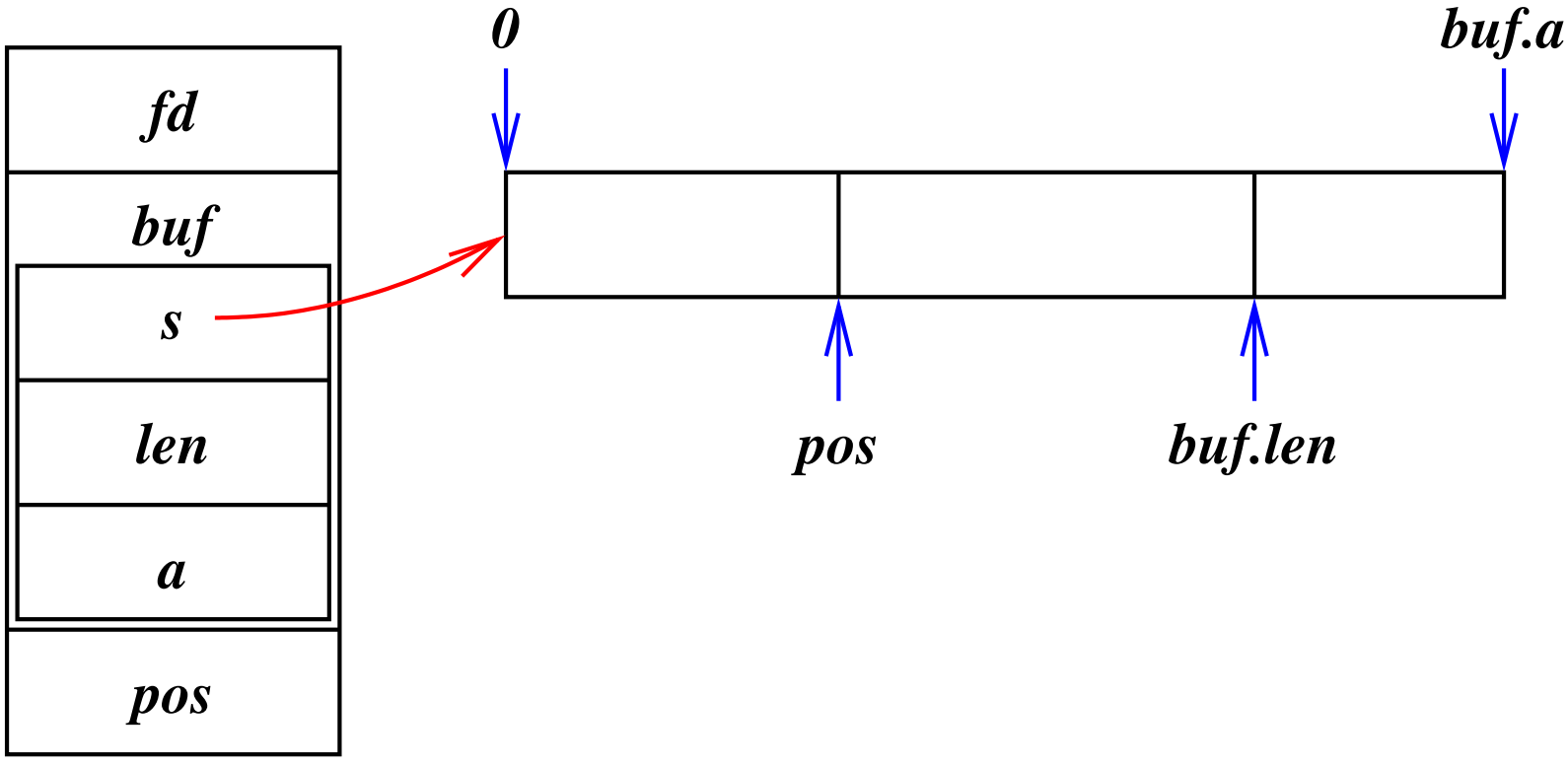
- Diese Vorgehensweise ist jedoch außerordentlich ineffizient, weil Systemaufrufe wie *read*() zu einem Kontextwechsel zwischen dem aufrufenden Prozess und dem Betriebssystem führen.
- Wenn ein Kontextwechsel für jedes einzulesende Byte initiiert wird, dann ist der betroffene Rechner mehr mit Kontextwechseln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

Implementierung für eine gepufferte Eingabe

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;
}
```

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);
}
```