Fragmentierung der Pakete bei Netzwerkverbindungen 207

- 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 208

- 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 209

- 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.

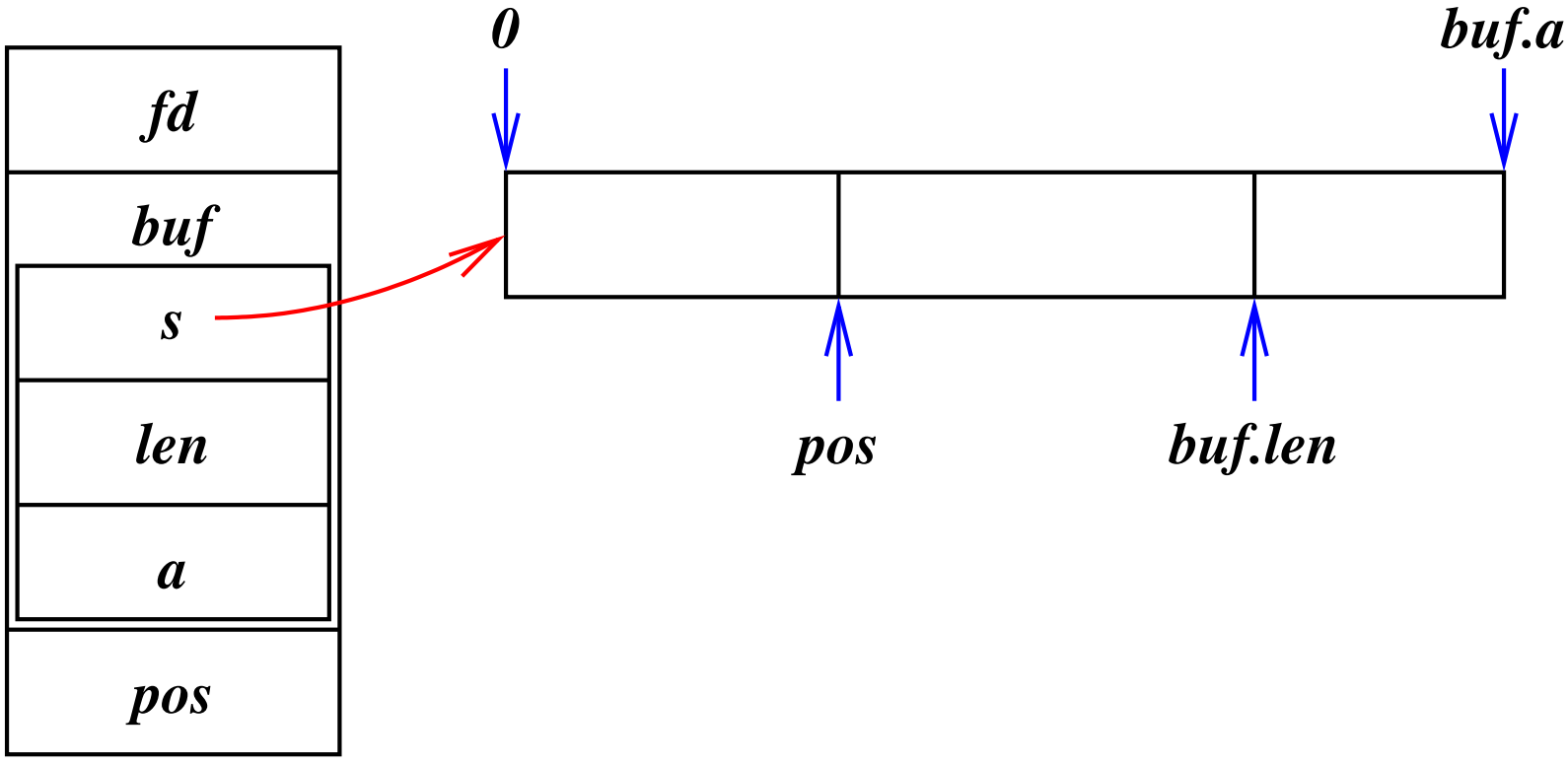
Diese Funktion kann die *stdio* jedoch nicht erfüllen:

- 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

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 */
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.

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 unter `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 einem 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);
}
```

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);
}
```