- 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.

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, size_t len);
ssize_t read_from_link(connection* link, char* buf, size_t 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, size_t len);
ssize_t read_from_link(connection* link, char* buf, size_t 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 */
   /* private fields */
   void* mpx; /* internal handle */
   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

Datenstruktur für die Warteschlange 275

multiplexor.c

```
typedef struct output_queue_member {
   char* buf;
   size_t len;
   size_t pos;
   struct output_queue_member* next;
} output_queue_member;
// ...
bool write_to_link(connection* link, char* buf, size_t 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 */
   size_t count; /* number of connections */
   struct pollfd* pollfds; /* parameter for poll() */
   size_t 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 size_t setup_polls(multiplexor* mpx) {
   size_t 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 size_t setup_polls(multiplexor* mpx) {
   /* ... */

   size_t 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 = true;
   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) {
   /* ignore SIGPIPE as we might receive this signal
      on writing to connections which were already
      closed by our client */
   struct sigaction sigact = {.sa_handler = SIG_IGN};
   struct sigaction old_sigact = {0};
   if (sigaction(SIGPIPE, &sigact, &old_sigact) < 0) return;

   /* ... */

   /* restore previous SIGPIPE handler */
   sigaction(SIGPIPE, &old_sigact, 0);
}
```

- *SIGPIPE* kann uns unerwartet treffen, wenn wir in eine Netzwerkverbindung schreiben, die von der Gegenseite bereits geschlossen ist.
- Entsprechend müssen wir uns dagegen wappnen und Verbindungen bei Schreibfehlern umgehend fallen lassen.

multiplexor.c

```
multiplexor mpx = {
   .socket = socket, .ohandler = open_handler,
   .ihandler = input_handler, .chandler = close_handler,
   .mpx_handle = mpx_handle, .socketok = true,
};
size_t count;
while ((count = setup_polls(&mpx)) > 0) {
   if (poll(mpx.pollfds, count, -1) <= 0) return;
   for (size_t 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*