- Die von Christopher M. Kohlhoff seit 2003 entwickelte Bibliothek bietet eine für C++ geeignete Schnittstelle auf Basis der BSD-Sockets.
- Das bedeutet, dass prinzipiell auch alle Operationen der BSD-Socket-Ebene „durchscheinen“ – es ist aber auch möglich, mehrere der Aufrufe der BSD-Socket-Ebene implizit ausführen zu lassen.
- Alle Operationen können synchron durchgeführt werden oder asynchron auf Basis des Proactor-Patterns.
- Prinzipiell kann die Bibliothek ohne Threads verwendet werden.

timeclient.cpp

```
#include <boost/asio.hpp>
#include <iostream>
#include <cstdlib>
#include <exception>

using namespace std;
using boost::asio::ip::tcp;

char* cmdname;
void usage() {
   cerr << "Usage: " << cmdname << " server port" << endl; exit(1);
}

int main(int argc, char* argv[]) {
   // ...
}
```

- Die Header-Datei <*boost/asio.hpp*> wird benötigt und die zweite **using**-Direktive vermeidet ansonsten unpraktisch lange Namen.
- Beim Übersetzen sollte „-lboost_system“ angegeben werden, unter Solaris auch noch „-library=stlport4“ und „-lsocket“.

timeclient.cpp

```
boost::asio::io_service io_service;
```

- Die boost::asio-Bibliothek tut nichts ohne ein Objekt des Typs *boost*::*asio*::*io_service*, über das alle BSD-Socket-Operationen abgewickelt werden.
- Von der prinzipiellen Vorgehensweise wäre das Objekt nur notwendig bei asynchronen Zugriffen, aber die Bibliothek wickelt auch die synchronen Operationen darüber ab.
- Im Normalfall wird nur ein solches Objekt benötigt. Es können darüber beliebig viele Operationen parallel abgewickelt werden.
- Prinzipiell dürfen auch mehrere Threads konkurrierend das gleiche *io_service*-Objekt verwenden.

timeclient.cpp

```
// process command line arguments
cmdname = *argv++; --argc;
if (argc != 2) usage();
std::string server(argv[0]);
std::string port(argv[1]);

// connect to given server
boost::asio::io_service io_service;
tcp::resolver resolver(io_service);
tcp::resolver::query query(server, port);
```

- Während in der vorherigen C-basierten Lösung *gethostbyname* verwendet wurde, das selbst direkt die BSD-Socket-Schnittstelle nutzt, können bei boost::asio solche Abfragen auch über ein *io_service*-Objekt abgewickelt werden.
- *tcp*::*resolver*::*query* repräsentiert eine DNS-Anfrage, wobei nur *server* an den DNS-Server in der Anfrage weitergeleitet wird. Der Port wird hier nur aus Gründen der Einfachheit integriert, damit anschließend vollständige kontaktierbare Adressen zurückgeliefert werden können, zu der eben auch der Port gehört.

timeclient.cpp

```
tcp::socket socket(io_service);
tcp::resolver::iterator end;
bool opened = false;
for (tcp::resolver::iterator it = resolver.resolve(query);
      it != end; ++it) {
   boost::system::error_code error;
   socket.connect(*it, error);
   if (!error) {
      opened = true; break;
   }
   socket.close();
}
```

- Bei einer DNS-Anfrage können mehrere Antworten zurückkommen.
- Die Methode *resolve* eines *tcp*::*resolver*-Objekts liefert einen Iterator zurück, der auf die Ergebnisliste verweist.
- Prinzipiell können das sowohl IPv4- oder IPv6-Adressen sein.
- Mit *socket.connect* wird jeweils eine Verbindungsaufnahme durchprobiert.

timeclient.cpp

```
try {
   // read input (under the assumption that we get just one packet)
   boost::array<char, 128> timebuf;
   size_t nbytes = socket.read_some(boost::asio::buffer(timebuf));
   std::cout.write(timebuf.data(), nbytes);
} catch (std::exception& e) {
   cout << cmdname << ": server failure: " << e.what() << endl;
   exit(1);
}
```

- Ein Objekt des Typs *boost*::*asio*::*buffer* kann eine Sequenz einzelner normaler Buffer aufnehmen. Damit ist eine sogenannte *scatter*/*gather*-Übertragung möglich, die Kopieroperationen zu vermeiden hilft und die auch auf der Ebene der Systemaufrufe zur Verfügung steht.
- Hier wird nur ein einzelnes Array in eine solche Sequenz aufgenommen.
- Ein *boost*::*array* ist festdimensioniert, kennt seine Größe und ist sicher gegen Überläufe.

timeserver.cpp

```
boost::asio::io_service io_service;
tcp::acceptor acceptor(io_service, tcp::endpoint(tcp::v4(), port));
acceptor.set_option(
   boost::asio::ip::tcp::acceptor::reuse_address(true));
```

- Ein *tcp*::*acceptor*-Objekt erlaubt es, Verbindungen mit Klienten aufzunehmen.
- Ein *acceptor*-Objekt ist mit einer festen Adresse und einem festen Port verbunden. (Hier IPv4, weil *tcp*::*v4* angegeben wurde.)
- Analog wie im C-basierten Beispiel wird hier ebenfalls sinnvollerweise die Socket-Option *SO_REUSEADDR* gesetzt.

timeserver.cpp

```
try {
   for(;;) {
      tcp::socket socket(io_service); acceptor.accept(socket);
      try {
         char timebuf[32]; time_t clock; time(&clock);
         ctime_r(&clock, timebuf, sizeof timebuf);
         std::string msg(timebuf);
         boost::asio::write(socket, boost::asio::buffer(msg),
            boost::asio::transfer_all());
      } catch (exception& e) {
         cerr << cmdname << ": " << e.what() << endl;
      }
   }
} catch (exception& e) {
   cerr << cmdname << ": " << e.what() << endl;
}
```

- Die Methode *accept* blockiert, bis sich ein Klient meldet.
- Die Sitzung wird hier in nicht parallelisierter Form abgewickelt.
- Wenn ein *write*-Systemaufruf nicht ausreicht, um alles zu schreiben, dann erzwingt der *boost*::*asio*::*transfer_all*-Handler eine Schleife, die *write* mit dem noch nicht übersandten Text aufruft.

Netzwerkdienste sollten sinnvollerweise parallel laufende Sitzungen ermöglichen. Hierzu gibt es vier prinzipielle Ansätze:

1. Für jede neue Sitzung wird mit Hilfe von *fork*() ein neuer Prozess erzeugt, der sich um die Verbindung zu genau einem Klienten kümmert.
2. Für jede neue Sitzung wird ein neuer Thread gestartet.
3. Sämtliche Ein- und Ausgabe-Operationen werden asynchron abgewickelt mit Hilfe von *aio_read*, *aio_write* und dem *SIGIO*-Signal.
4. 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.

Die boost::asio-Bibliothek bzw. das Proactor-Pattern unterstützt die vierte Variante bei den asynchronen Operationen. Prinzipiell ist natürlich auch die zweite Variante möglich – auch dann kann ein *boost*::*asio*::*io_service*-Objekt gemeinsam verwendet werden.

- Zu jedem *io_service*-Objekt gehört eine Warteschlange mit durchzuführenden Operationen.
- Wenn asynchrone *boost*::*asio*-Operationen abgesetzt werden, dann füllen die zunächst nur die Warteschlange.
- Zu jeder asynchronen Operation gehört auch die Angabe eines Behandlers, der bei Fertigstellung der Operation aufzurufen ist.
- Wenn die *run*-Methode aufgerufen wird, dann werden die in der Warteschlange befindlichen Operationen alle gleichzeitig in die Wege geleitet (mit *poll*) und dann die Ereignisse abgearbeitet.
- Die *run*-Methode endet, wenn die Warteschlange leer geworden ist oder das *io_service*-Objekt mit *stop* gebeten wurde, aufzuhören.
- In diesem Pattern haben wir Initiatoren (Aufrufer der asynchronen Operationen) und den Proactor (Implementierung der *run*-Methode).

Hauptprogramm im Proactor-Pattern

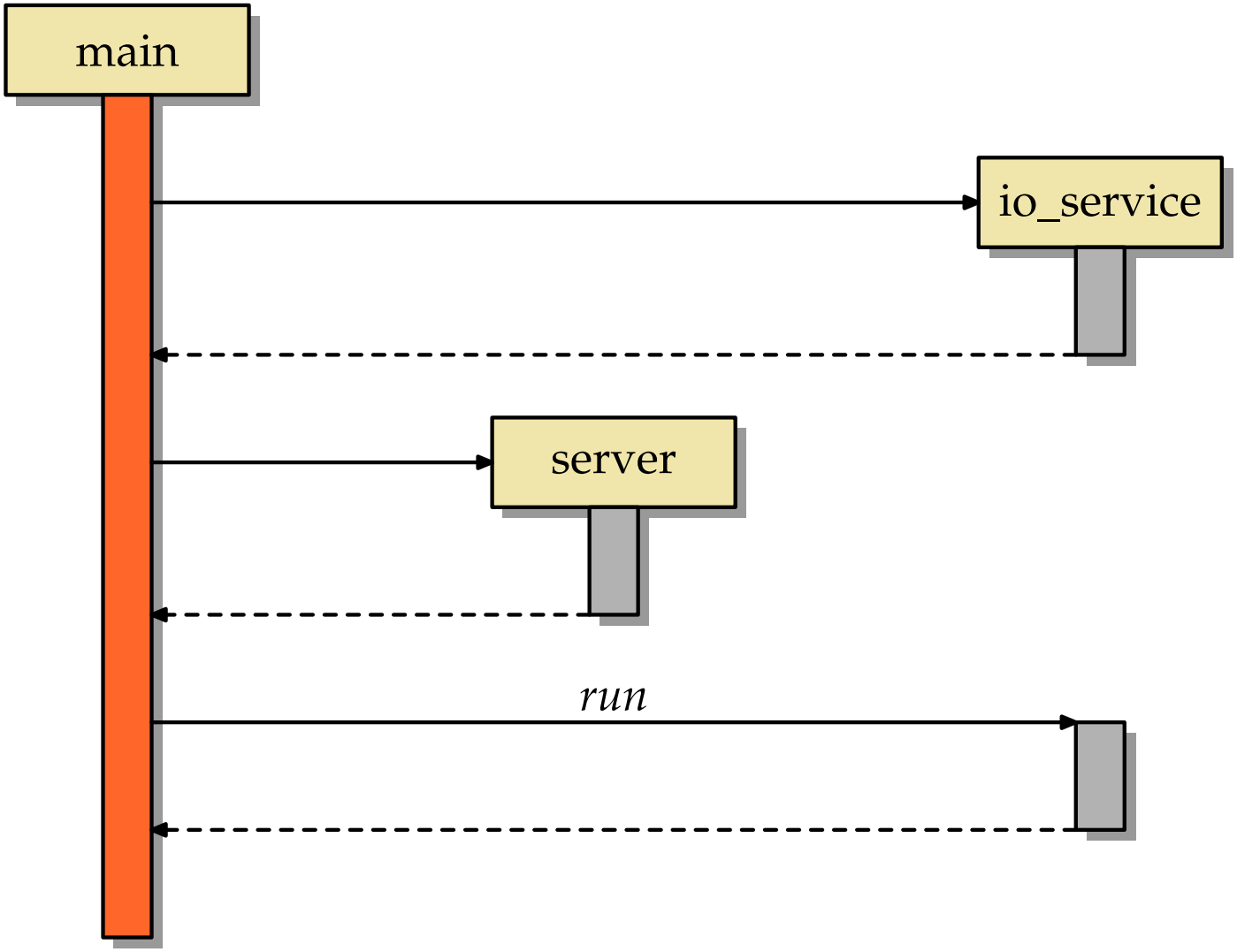
async-timeserver.cpp

```
int main(int argc, char* argv[]) {
   // process command line arguments
   cmdname = *argv++; --argc;
   if (argc != 1) usage();
   unsigned int port(atoi(argv[0]));

   try {
      boost::asio::io_service io_service;
      Server<TimeServerSession> server(io_service,
            tcp::endpoint(tcp::v4(), port));
      io_service.run();
   } catch (exception& e) {
      cerr << cmdname << ": " << e.what() << endl;
   }
}
```

- Die *main*-Funktion erzeugt nur ein *Server*-Objekt und überlässt dann den Rest dem *io_service.run*(), das normalerweise hier nie zurückkehrt.

Hauptprogramm im Proactor-Pattern



Server im Proactor-Pattern

async-timeserver.cpp

```
template<typename Session>
class Server {
   public:
      template<typename Endpoint>
      Server(boost::asio::io_service& io_service, Endpoint endpoint) :
            acceptor(io_service, endpoint) {
         acceptor.set_option(
            boost::asio::ip::tcp::acceptor::reuse_address(true));
         start_accept();
      }
   private:
      typedef boost::shared_ptr<Session> SessionPtr;

      void start_accept() {
         SessionPtr session = Session::create(acceptor.get_io_service());
         acceptor.async_accept(session->socket(),
            boost::bind(&Server<Session>::handle_accept, this,
               session, boost::asio::placeholders::error));
      }

      void handle_accept(SessionPtr session,
            const boost::system::error_code& error) {
         if (!error) {
            session->start();
            start_accept();
         }
      }

      tcp::acceptor acceptor;
};
```

async-timeserver.cpp

```
template<typename Endpoint>
Server(boost::asio::io_service& io_service, Endpoint endpoint) :
      acceptor(io_service, endpoint) {
   acceptor.set_option(
      boost::asio::ip::tcp::acceptor::reuse_address(true));
   start_accept();
}
```

- Der Konstruktor ist als Template ausgeführt, damit er unabhängig ist von dem Datentyp *Endpoint*.
- Die private *start_accept*-Methode führt die Registrierung beim *io_service*-Objekt durch.

async-timeserver.cpp

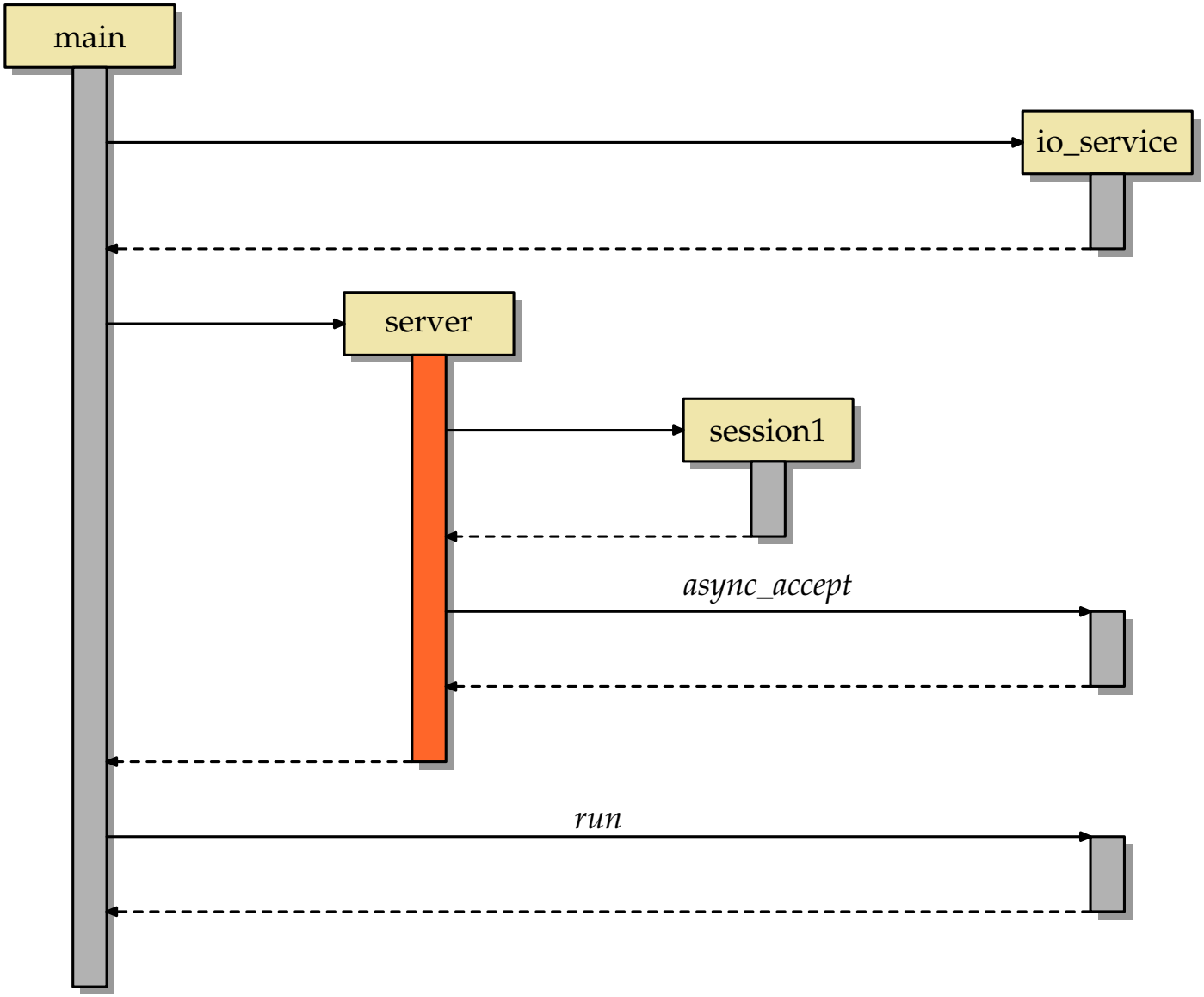
```
tcp::acceptor acceptor;
```

async-timeserver.cpp

```
void start_accept() {
   SessionPtr session = Session::create(acceptor.get_io_service());
   acceptor.async_accept(session->socket(),
      boost::bind(&Server<Session>::handle_accept, this,
         session, boost::asio::placeholders::error));
}
```

- *start_accept* erzeugt ein neues Sitzungs-Objekt und sorgt dafür, dass dieses beim Aufruf des Bearbeiters *handle_accept* gestartet wird.
- *SessionPtr* ist ein *boost*::*shared_ptr* auf *Session* damit die Freigabe automatisiert wird.
- *boost*::*bind* erzeugt ein anonymes Funktionsobjekt, das die Funktion (hier ein Methodenzeiger) mit den weiteren Parametern aufruft.
- Bei Methodenzeigern muss hier als extra Parameter noch **this** angegeben werden.

Erzeugung des Servers im Sequenz-Diagramm



Erzeugung einer Sitzung

async-timeserver.cpp

```
class TimeServerSession:
      public boost::enable_shared_from_this<TimeServerSession> {
   public:
      static TimeServerSessionPtr create(boost::asio::io_service& io_service) {
         return TimeServerSessionPtr(new TimeServerSession(io_service));
      }

      tcp::socket& socket() {
         return my_socket;
      }

      // ...

   private:
      TimeServerSession(boost::asio::io_service& io_service) :
            my_socket(io_service) {
      }

      // ...

      tcp::socket my_socket;
      std::string message;
};
```

async-timeserver.cpp

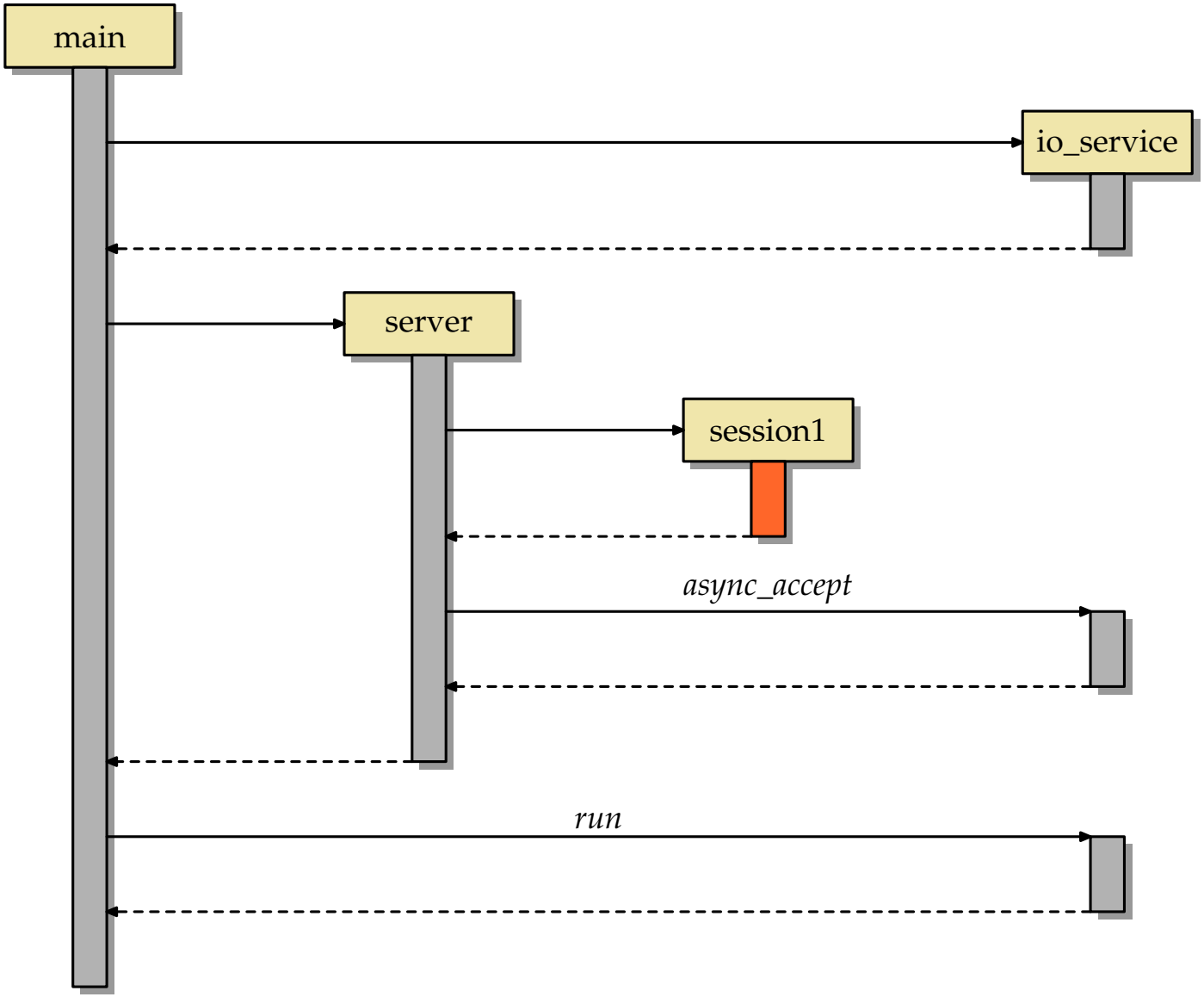
```
public:
   static TimeServerSessionPtr create(boost::asio::io_service& io_service) {
      return TimeServerSessionPtr(new TimeServerSession(io_service));
   }

   tcp::socket& socket() {
      return my_socket;
   }
   // ...

private:
   TimeServerSession(boost::asio::io_service& io_service) :
         my_socket(io_service) {
   }
```

- Da Sitzungen nur über *TimeServerSessionPtr* (*boost*::*shared_ptr*) referenziert werden sollen, kann dies durch eine statische *create*-Methode sichergestellt werden.
- Sitzungen werden jeweils auf Vorrat erzeugt. Damit das *my_socket*-Objekt später von *start_accept* mit Hilfe von *async_accept* fertig konfiguriert werden kann, wird die *socket*-Methode benötigt, die das Objekt zur Verfügung stellt.

Erzeugung einer Sitzung im Sequenz-Diagramm

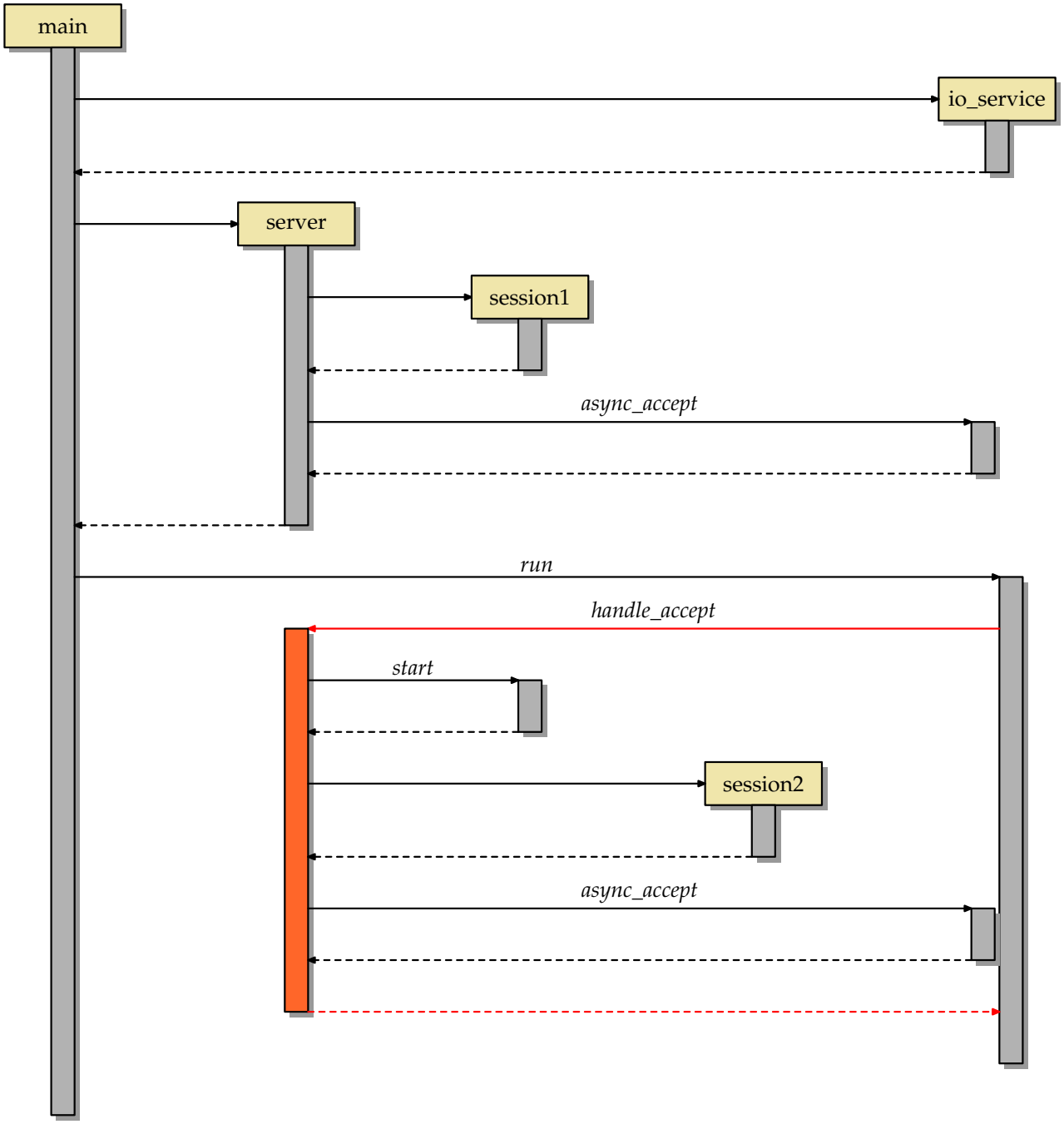


async-timeserver.cpp

```
void handle_accept(SessionPtr session,
      const boost::system::error_code& error) {
   if (!error) {
      session->start();
      start_accept();
   }
}
```

- Diese Methode wurde von *start_accept* an *async_accept* übergeben.
- Sobald die *accept*-Operation abgeschlossen ist und somit eine neue Verbindung aufgenommen wurde, wird dieser Bearbeiter aufgerufen.
- Das bereits zuvor erzeugte Sitzungsobjekt wird dann gestartet und mit Hilfe von *start_accept* wird *async_accept* erneut aufgerufen.
- Bei Fehlern gibt es hier nichts weiter zu tun.

Neue Verbindung im Sequenz-Diagramm



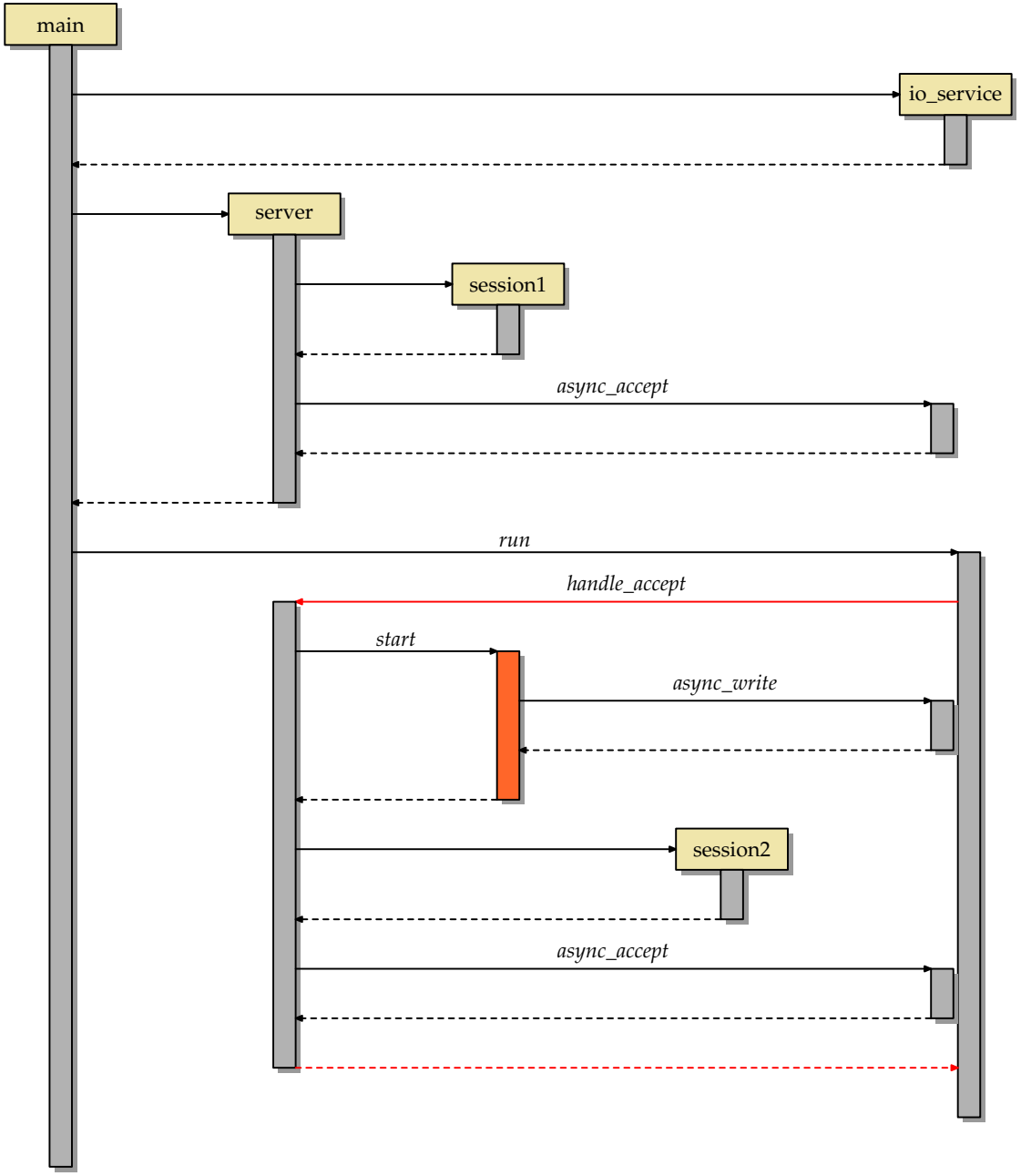
async-timeserver.cpp

```
void start() {
   char timebuf[32]; time_t clock; time(&clock);
   ctime_r(&clock, timebuf);
   message = timebuf;

   boost::asio::async_write(my_socket, boost::asio::buffer(message),
      boost::bind(&TimeServerSession::handle_write,
         shared_from_this(),
         boost::asio::placeholders::error,
         boost::asio::placeholders::bytes_transferred));
}
```

- Alle I/O-Operationen müssen in einer Sitzung asynchron über die entsprechenden asynchronen Operationen des *io_service*-Objekts initiiert werden.
- *async_write* erhält als Parameter *my_socket*, das zuvor mit *io_service* verknüpft wurde. Entsprechend führt dies intern zu einem Aufruf einer Methode des *io_service*-Objekts, das diese Operation in seine Warteschlange übernimmt.
- Deswegen wird das auf dem Sequenz-Diagramm vereinfacht als Aufruf des *io_service*-Objekts dargestellt.

Start einer Sitzung im Sequenz-Diagramm

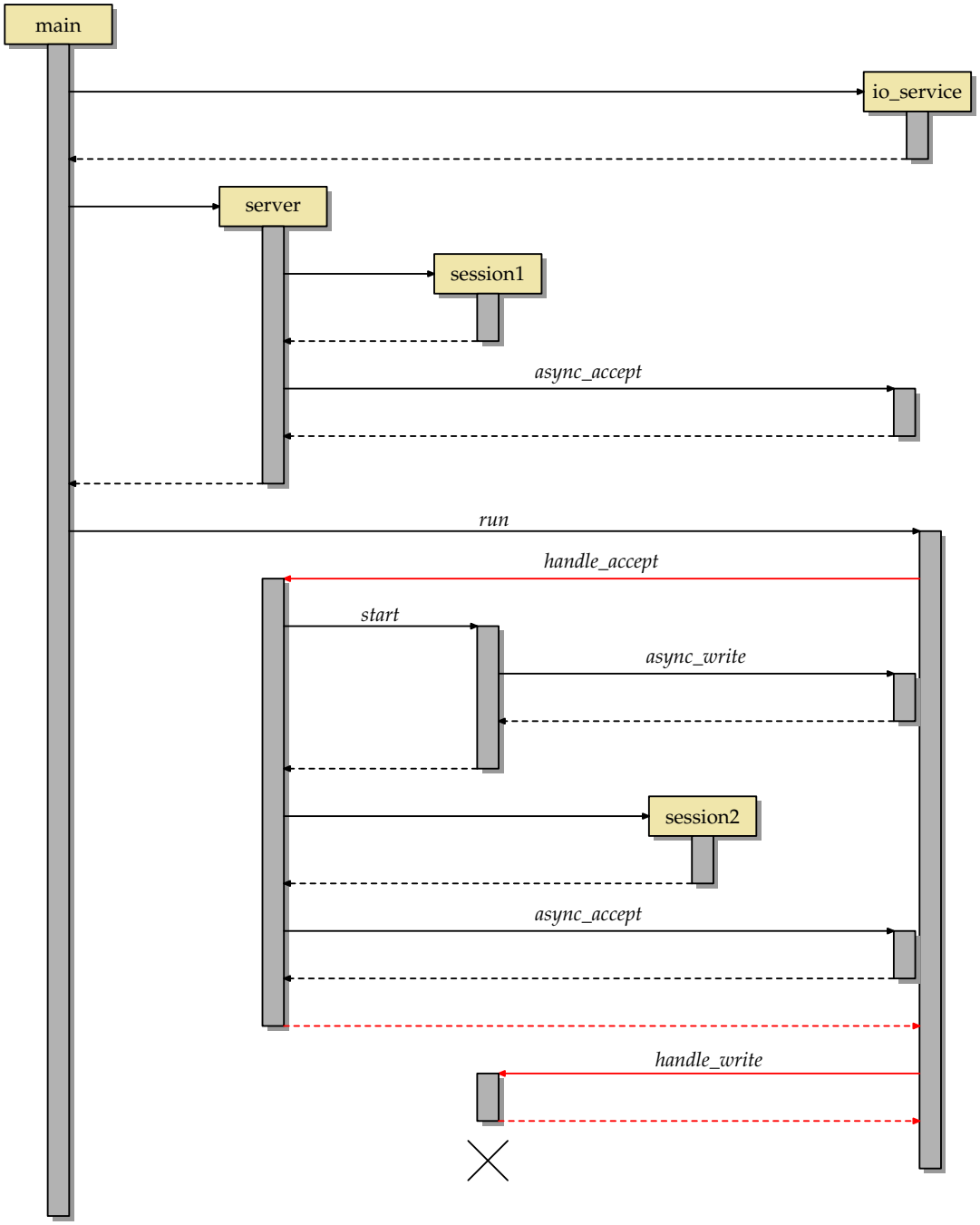


async-timeserver.cpp

```
void handle_write(const boost::system::error_code& error,
      size_t nbytes) {
}
```

- Wenn es nach dem Abschluss der Schreib-Operation noch etwas zu tun gäbe – dann müsste das hier ebenfalls wieder asynchron in die Wege geleitet werden.
- Da hier jedoch nichts passiert, kommt auch kein dieses Sitzungs-Objekt referenzierendes Element in die Warteschlange des *io_service*-Objekts.
- Da das Sitzungs-Objekt sonst nirgendwo mehr über einen *boost*::*shared_ptr* referenziert wird, wird es unmittelbar nach Rückkehr dieser Methode beendet werden.
- (Bis dahin wurde es von dem von *boost*::*bind* erzeugten Funktionsobjekt referenziert, das wiederum in der Warteschlange des *io_service*-Objekts lag.)

Sequenz-Diagramm



- Auf dem Proactor-Pattern basierende Lösung sind sehr viel ressourcenschonender in Bezug auf den Kernel (Threads) und durch die nicht benötigte Synchronisierung von Threads auch sehr viel effizienter.
- Dieser Vorteil ist insbesondere dann von Gewicht, wenn mit einer sehr hohen Zahl konkurrierender Sitzungen gerechnet wird.
- Nachteilig ist der schwer lesbare Programmtext, da zusammengehörende Code-Sequenzen völlig zerstückelt werden.
- Ein Ausweg wäre mit Hilfe von Koroutinen möglich, die aber bislang noch nicht von C++ unterstützt werden. Eine entsprechende Boost-Bibliothek befindet sich in Entwicklung.