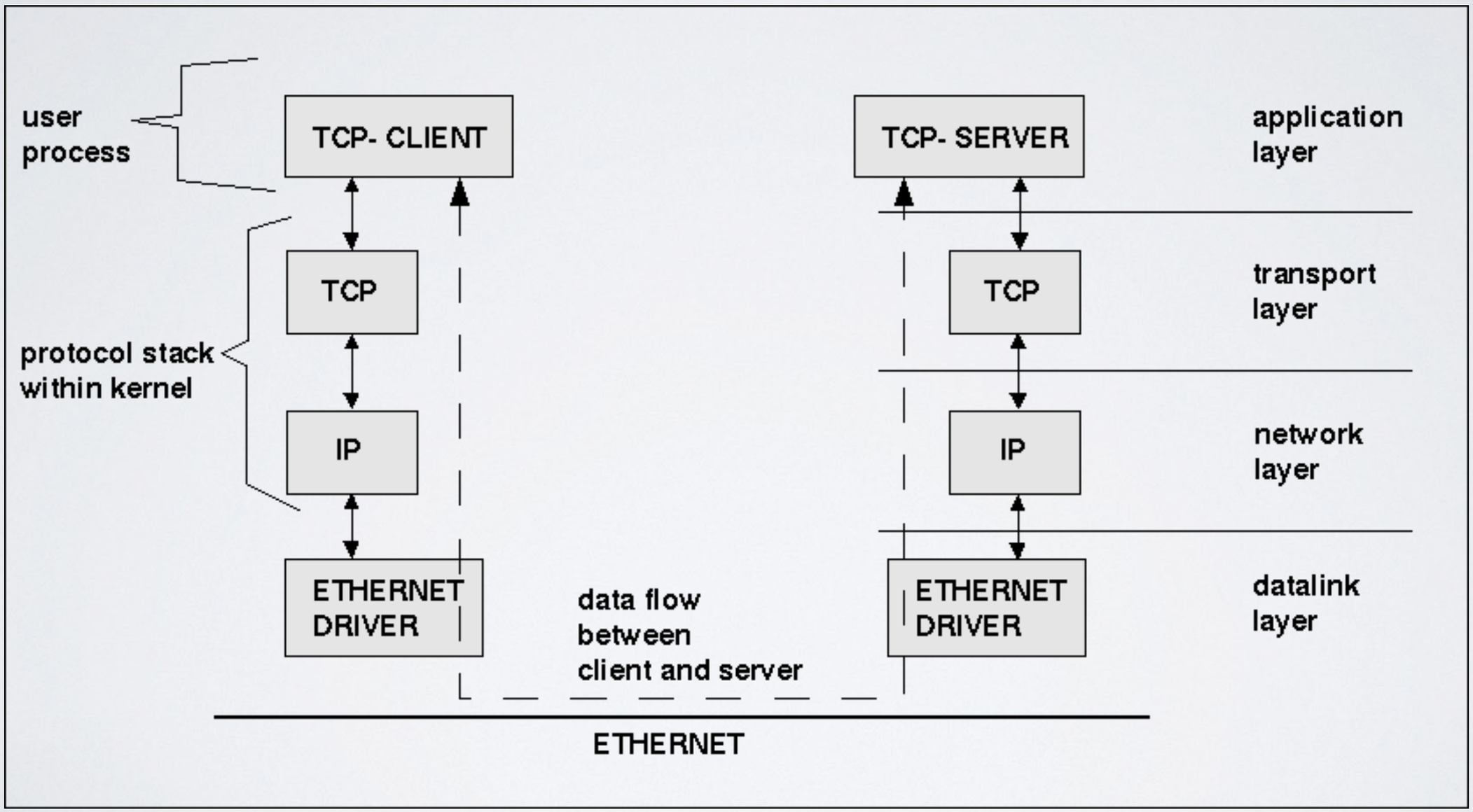
PROGRAMMIEREN II

DHBW Stuttgart Campus Horb INF2018

AGENDA

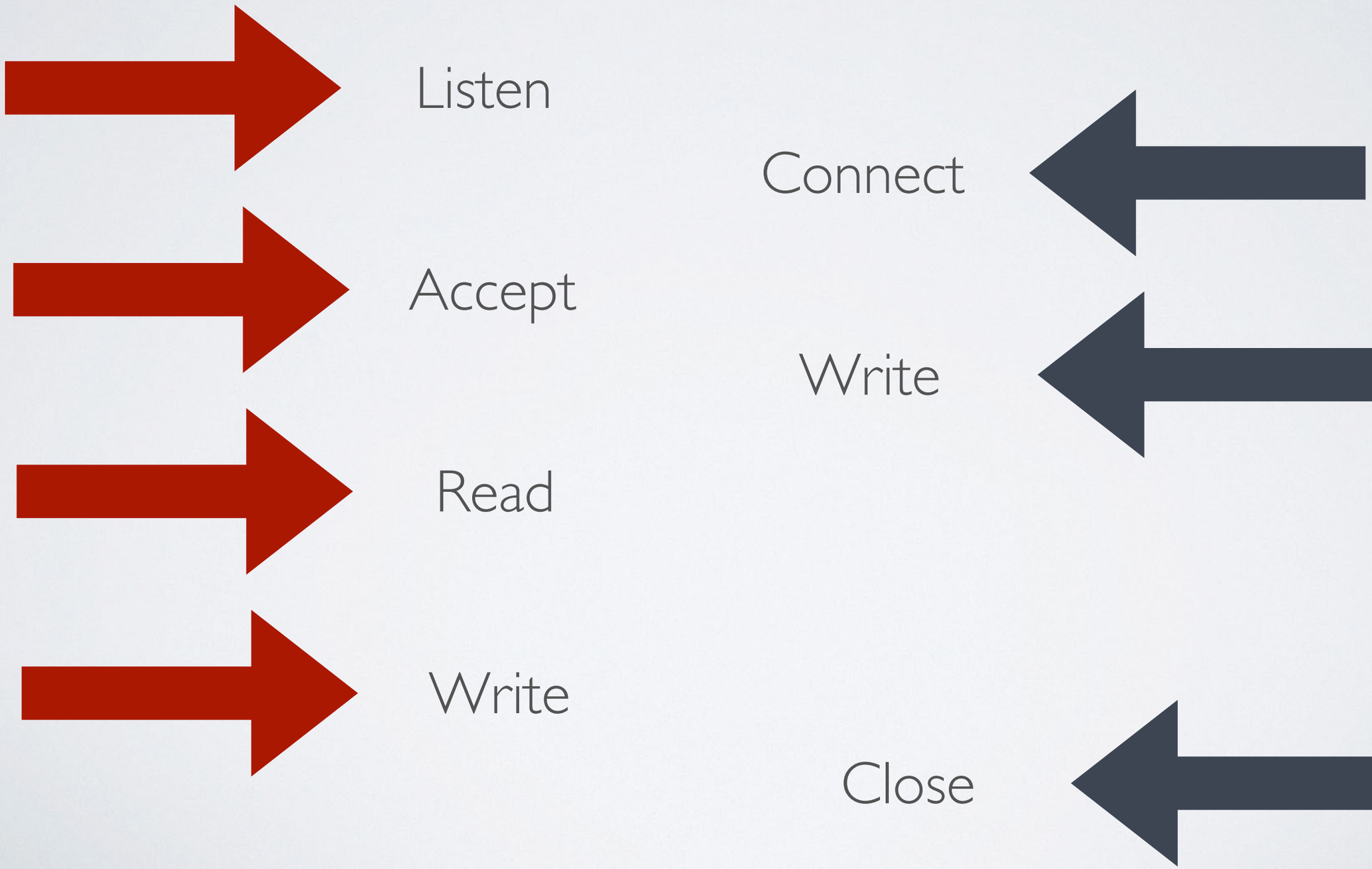
- Netzwerken mit Boost anhand von Beispielen
- Letzte Fragen zur Projektarbeit
- Rückblick und Feedback


user
process
TCP- CLIENT
TCP
IP
protocol stack
within kernel
ETHERNET
DRIVER
data flow
between
client and server
ETHERNET
TCP- SERVER
TCP
IP
ETHERNET
DRIVER
application
layer
transport
layer
network
layer
datalink
layer

WIEDERHOLUNG WAS IST TCP?

- Kommunikationsprotokol mit Empfangsgarantie (vs. UDP)
- Kommunikation über Sockets (zu deutsch Muffen)
- Synchrone Lese und Schreibeaktionen
- Networking liegt in Betriebssystemschicht

TYPISCHER TCP FLOW



WICHTIG DABEI

- Beide Seite müssen wissen, welche Nachrichtenformen sie versenden

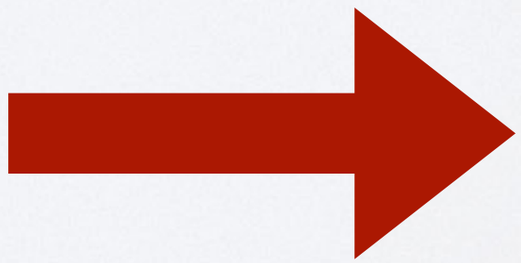
- Beide Seiten müssen immer lesend sein, um über Kommunikationsabbrüche Bescheid zu wissen

ZUNÄCHST DER SERVER

- Einfaches Beispiel eines Servers, der einfach nur eine 100 Byte lange Nachricht bekommt, reversed und zurücksendet

SCHRITT EINS:

```
// io_service ist die wichtigste Komponente
// sie regelt alles rund um das asynchrone
// eventhandling. --> Erstmal Blackbox
boost::asio::io_service my_io_service;
// das ip Protokoll sowie der Port
tcp::endpoint endpoint{tcp::v4(), 8999};
// Komponente zum akzeptieren von Verbindungen
tcp::acceptor acceptor{my_io_service};
acceptor.open(endpoint.protocol());
// das festlegen auf den Port, ist dieser Port bereits
// belegt, gibt es eine exception
acceptor.bind(endpoint);
// lausche auf dem Port nach neuen Verbindungen
acceptor.listen();
```



Listen

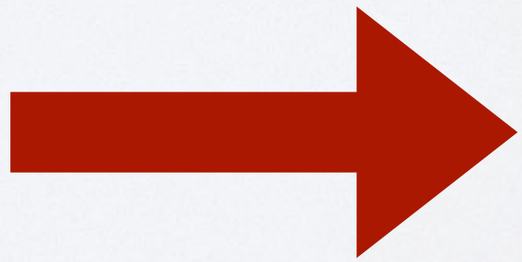
DATENSTRUKTUR ZUM VERWALTEN EINER VERBINDUNG

```
typedef struct connection
{
    tcp::socket m_sock;
    char *buf;
    connection(boost::asio::io_service &io_service)
: m_sock(io_service), buf(new char[100])
    {
    }
    ~connection()
    {
        delete[] buf;
        m_sock.close();
    }

} connection_t;
```

SCHRITT 2:

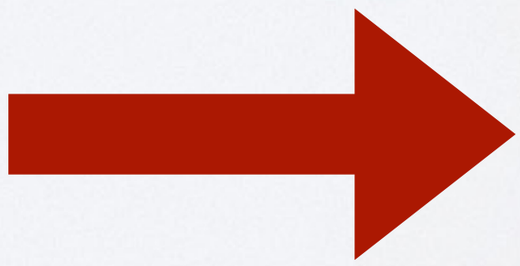
```
    acceptor.async_accept(con->m_sock, [con](const
error_code_t &ec) {
  //… Handler function
        }
    });
```



Accept

SCHRITT 2,5:

```
    acceptor.async_accept(con->m_sock, [con](const error_code_t &ec) {
        if (!ec)
        {
            auto boost_buf = boost::asio::buffer(con->buf, 100);
            boost::asio::async_read(con->m_sock, boost_buf, [con]
(error_code_t ec, size_t len) {
                // Handler function
        }
    });
```



Read

BOOST BUFFER

```
// buffer ist der Boost Way um einen block mit fester
// größe zum lesen zu erzeugen. Benötigt einen Speicher-
// bereich und die gewünschte Menge an bytes, die gelesen
// werden sollen
auto boost_buf = boost::asio::buffer(con->buf, 100);
```

DIE ASYNC_READ FUNKTION

```
// liest so lange asynchron von diesem Socket bis
// der buffer voll ist, oder die Verbindung abgebrochen wird
// wir geben die Connection immer mit, damit wir wissen, dass
// das Object zu 100% gültig ist, wenn die Funktion ausgeführt wird
boost::asio::async_read(con->m_sock, boost_buf, [con](error_code_t
ec, size_t len) {…}
```

FRAGE

- Warum fangen wir direkt mit dem lesen an?

Interaktion mit dem Client

Mitbekommen von Verbindungsabbrüchen

DIE READ_HANDLER FUNKTION UND ASYNC_WRITE

```
[con](error_code_t ec, size_t len) {
                std::cout << con->buf;
                std::reverse(con->buf, con->buf + 100);
                auto buf = boost::asio::buffer(con->buf, 100);
                // async_write ruft die handler Funktion auf, sobald es den
                // kompletten buffer in das Socket geschrieben hat
                boost::asio::async_write(con->m_sock, buf,
                    [con](error_code_t ec, size_t len) {
                        std::cout << std::endl << "i did my job" << std::endl; });
            });
```

FRAGE

- Was ist problematisch an dem Code?

Error Code wird nicht ausgewertet!

FRAGE

- Was muss man bei einer Handler Funktion beachten?

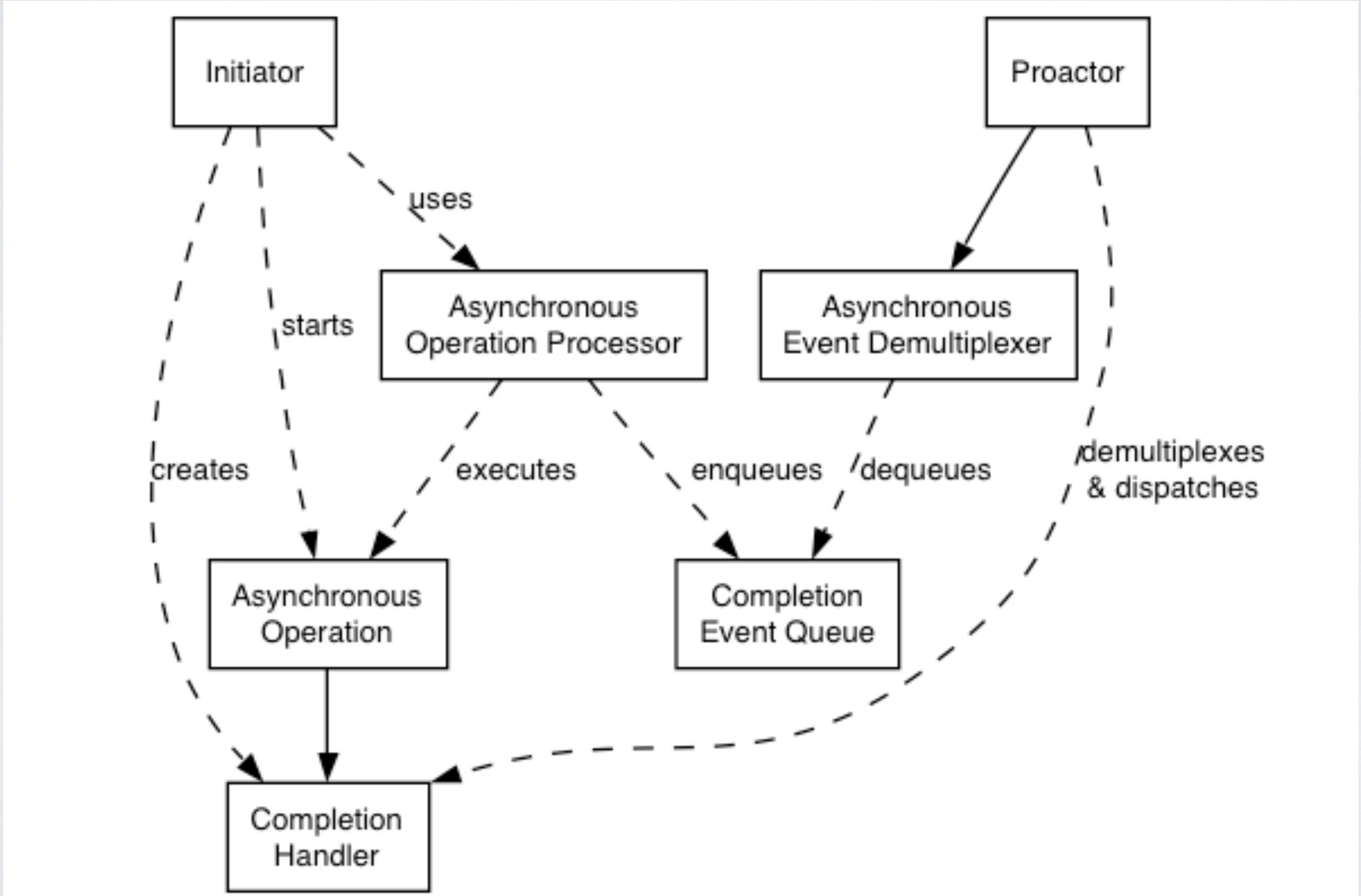
Keine langen blockenden (synchronen) Aufrufe

Am Ende immer wieder asynchron lesen!

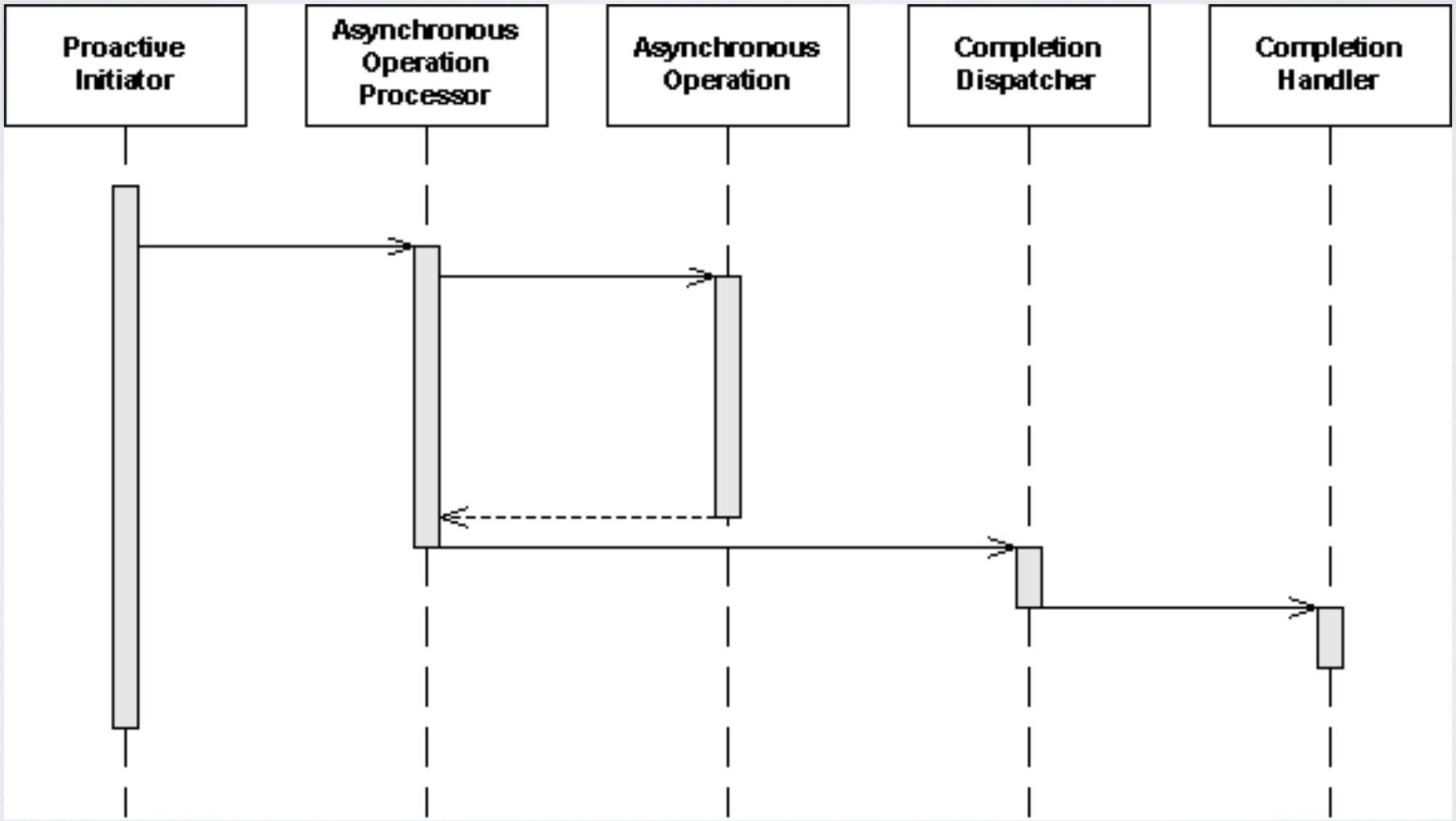
STARTEN DER EIGENTLICHEN ACTION

```
    // Erstellen eines threads in dem der io_service gestartet wird
    // Der Thread endet erst wenn der Event-Loop leer ist,
    // also keine asynchrone Aktion mehr ausgeführt wird
    // das geschieht bei einem Server in der Regel erst wenn er
    // beendet wird
    auto thread_func = [&my_io_service]() { my_io_service.run(); };
    std::thread t(thread_func);
    // Der Server soll solange laufen bis der Event-Loop leer ist
    t.join();
```

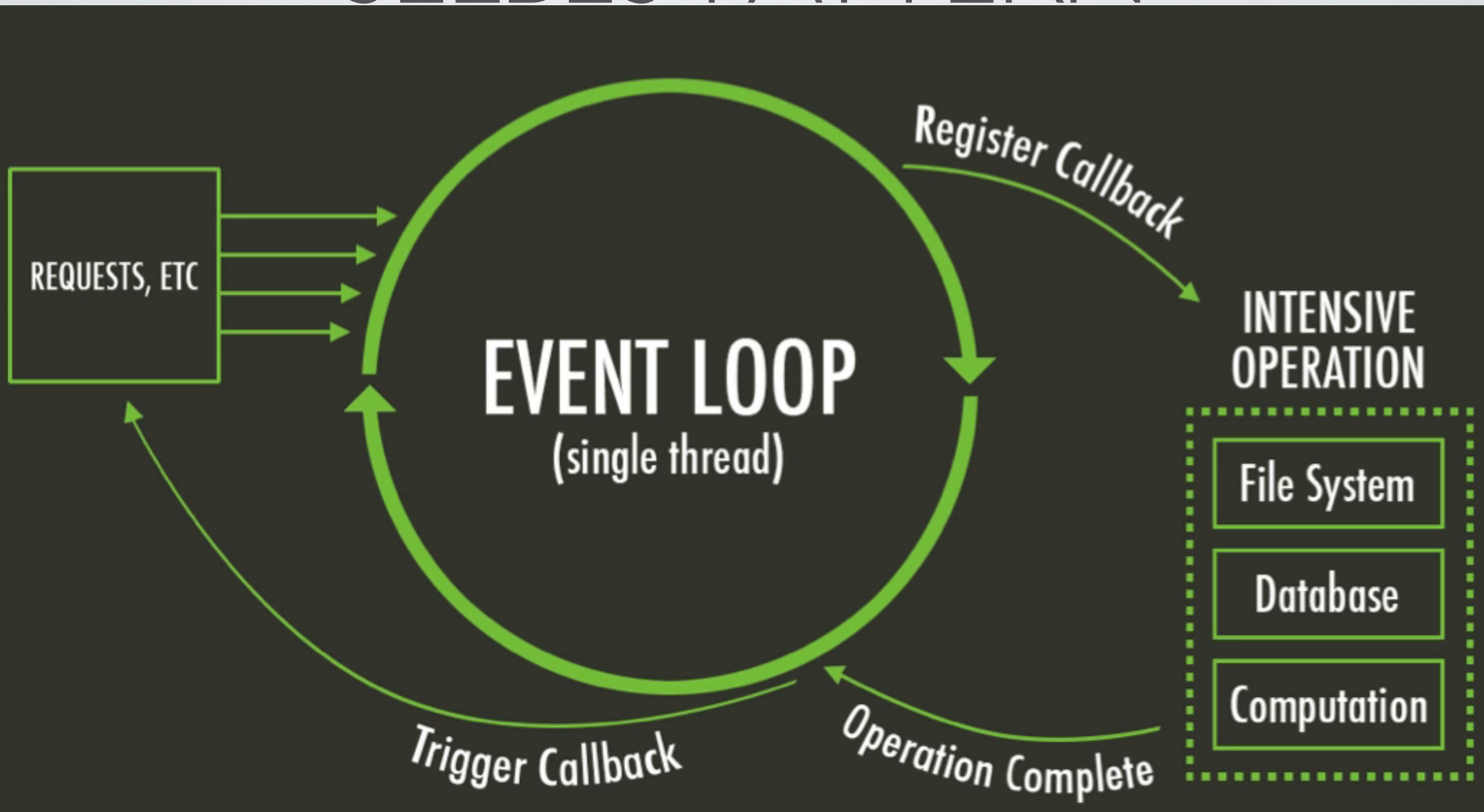
ETWAS THEORIE



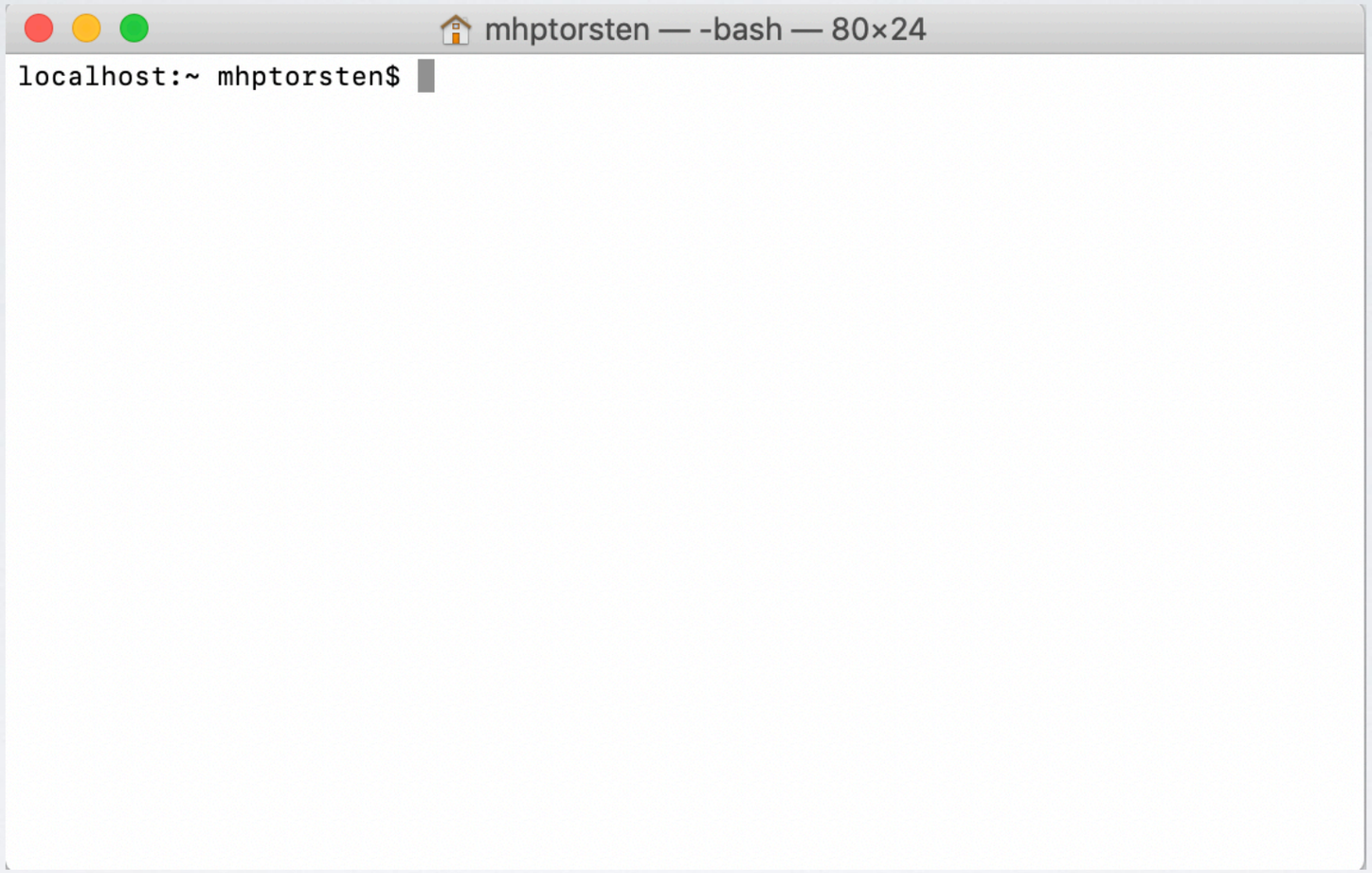
ETWAS VERSTÄNDLICHER



LEICHTERES BEISPIEL NODE.JS SELBES PATTERN



LIVE DEMO



FRAGE

- Warum beendet sich der Server nach dem Antworten?

FRAGE

- Live Session zum Thema Server & Client?

- Oder: Wir schauen uns das fertige einfach mal an

RÜCKBLICK / FEEDBACK

- Wie hat es euch gefallen?
- Habt ihr das Gefühl ihr könnt jetzt C++?
- Was hat euch gefehlt?
- Wie fandet ihr die Übungen?
- War es zu einfach oder eher zu schwer?
- Allgemeines Feedback