# Nebenläufige und verteilte Programme CS2301

**Th. Letschert**

**TH Mittelhessen Gießen**

***University of Applied Sciences***

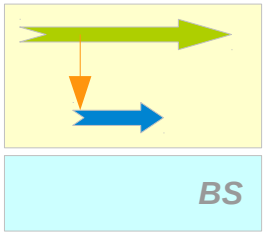
## New I/O, Asynchrone Kommunikation, Reaktive Server

- Asynchrone I/O und *non-Blocking*-I/O
- NIO 1 Reaktive I/O: Selector
- NIO 2 Asynchronous File Channel
- NIO-Frameworks
- Reaktive Architekturen: *Reactor-Pattern*, *Staged Event-Handling*

# Synchron – Asynchron: Wer agiert (a-)synchron zu wem?

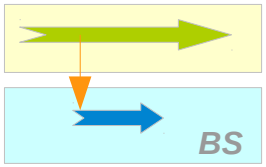
## Asynchron auf der Anwendungsebene

- verwendet synchrone (blockierende) I/O-Operationen der Systeme-Ebene
- Demultiplexer / Dispacher-Thread erwartet die Ereignisse und leitet sie an Handler-Threads weiter
- Diese operieren dann asynchron zum Rest der Anwendung




## Asynchron auf der Systemebene

- verwendet asynchrone (nicht blockierende) I/O-Operationen der Systeme-Ebene
- Diese operieren dann asynchron zur gesamten Anwendung



# Asynchrone I/O

**Meist (nicht immer) verstanden als**

- **nebenläufige Abarbeitung von I/O-Operationen und anderen Aktionen**
- **ohne den Einsatz von Threads (der Anwendung)**
- **also nebenläufig**
  - **zum kompletten Rest der Anwendung**
  - **zu allen Anwendung / ohne Einsatz der CPU**

**also: asynchron auf System-Ebene**

**I/O-Operationen werden dabei**

- **vom Programm beim Betriebssystem registriert / angestoßen,**
- **nach Abschluss der Anwendung gemeldet**

**Basis: Betriebssystem**

- **Asynchrone I/O ist sinnvoll / möglich bei Unterstützung durch das Betriebssystem**
- **Falls die Plattform-Unterstützung für asynchrone I/O nicht vorhanden ist, kann die Funktionalität mit Threads emuliert werden. (Ist dann aber nicht mehr wirklich asynchrone I/O)**

**Nicht-blockierende I/O**

- **Asynchrone I/O ist nicht blockierend**

  **Blockiert den Thread nicht, der sie initiiert**
- **Die asynchronen I/O-Operationen sollten vom Betriebssystem geliefert werden**

# Asynchron: Verarbeitung, I/O, Kommunikation

## Asynchrone Verarbeitung

Aufgaben in einem speziellen eigenen Handlungsfaden bearbeiten

- **Technische Frage** wie: Threadpool / Event-Queue, ….
- **SW-Technik-Frage** Strukturiertes Programmieren / Kombination asynchroner Aktivitäten

## Asynchrone I/O

I/O – eine Aktivität des Betriebssystems – asynchron ausführen

- **Technische Frage** Wie BS-Unterstützung nutzen (NIO / NIO2)
- **SW-Technik-Frage** Strukturiertes Programmieren / Kombination asynchroner I/O-Aktivitäten mit dem Rest der Verarbeitung

## Asynchrone Kommunikation

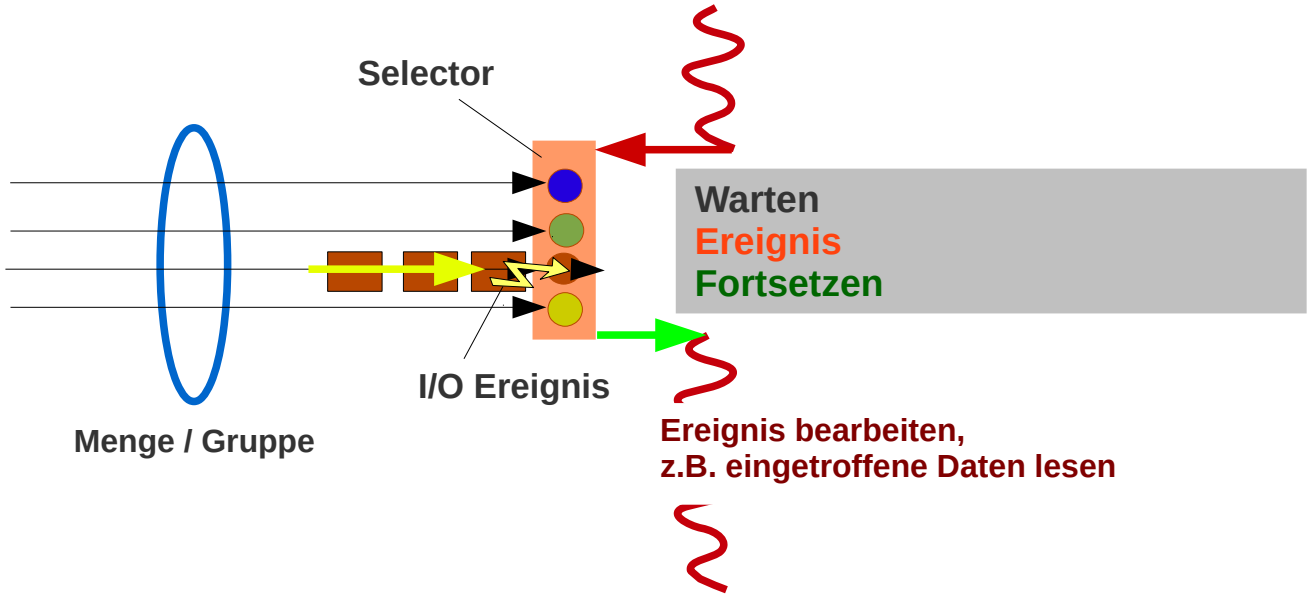
Sende- und Empfangsoperation erfolgen nicht gleichzeitig, bzw. sind entkoppelt

- **Technische Frage** Wer kommuniziert mit welchen (BS-) Mechanismen z.B. Interprozess-Kommunikation mit UDP
- **SW-Technik-Frage** Anwendungen basierend auf asynchroner Kommunikation gestalten (üblich, synchron ist unüblich)

# Selector: Die Urform der asynchronen I/O

## Prinzip

- Gruppiere einige I/O-Kanäle zu einer Menge
- Warte darauf, dass an einem oder mehreren ein Ereignis auftritt (etwas passiert)
- Bearbeite das Ereignis




## Implementierung

- Bestandteil der Socket-Schnittstelle (BS-übergreifender Quasi-Standard)
- Erfordert BS-Unterstützung (select-Systemaufruf)
- Erstmals eingeführt in BSD-Unix (1982) (unverändert aktuell)

## Selector / Selectable Channels: Asynchrone I/O mit NIO.1-Mitteln

**Klasse *java.nio.channels*.Selector (ab Java 1.4)**

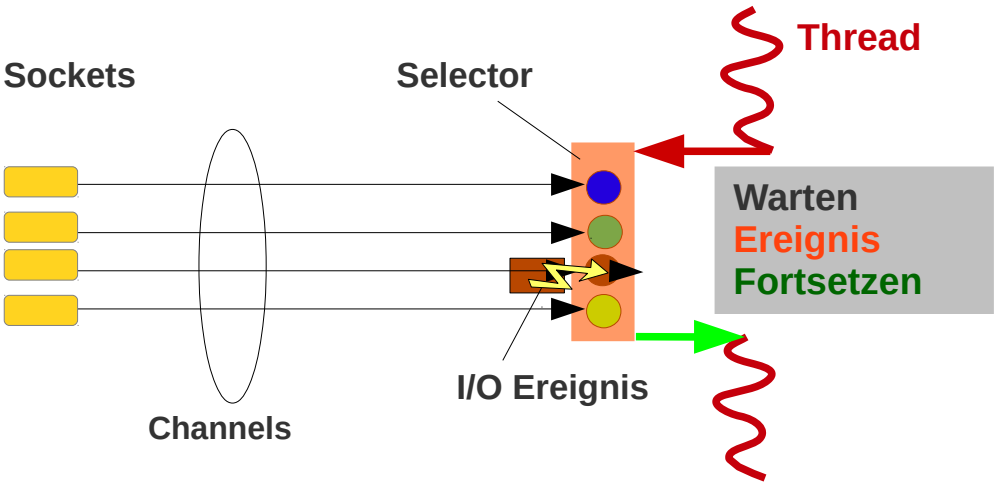
Funktionalität wie select-Systemaufruf:
Multiplexer / Dispatcher für eintreffende Ereignisse die an SelectableChannels weitergegeben werden

**Klasse *java.nio.channels*.SelectableChannel**

Unterklassen (u. A.): DatagramChannel, ServerSocketChannel, SocketChannel

**Asynchrone I/O im „alten Stil“**

SelectableCannels im nicht-blockierenden Modus
+ Selector




- Channels werden beim Selector registriert,
- Selector wartet auf ein Ereignis bei irgendeinem der Channels
- Selector de-blockiert: Thread stellt fest an welchem Channel welches Ereignis anliegt und behandelt es.

*Viele Sockets können gleichzeitig überwacht werden, ohne dass dafür jeweils ein Thread verwendet werden muss.*



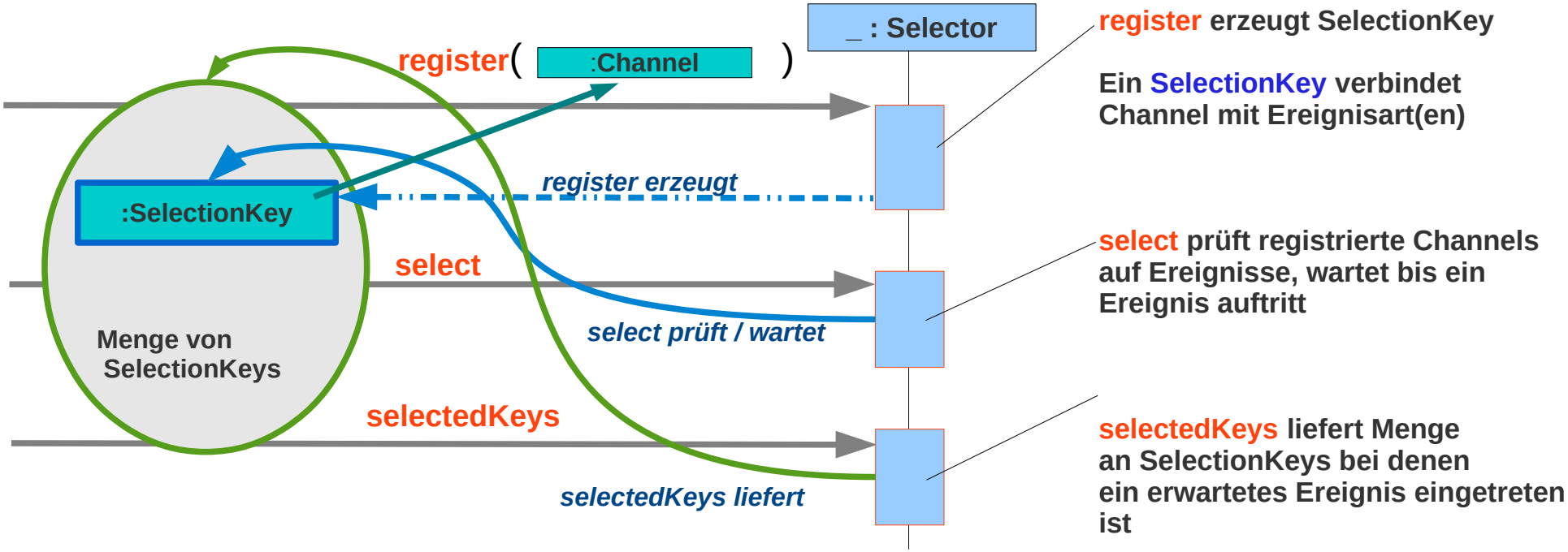
*Der Sytemaufruf select wird von allen (relevanten) Betriebssystemen unterstützt, wenn auch nicht unbedingt unter diesem Namen.*

# Selector / Selectable Channels

## Klasse java.nio.channels.SelectionKey

Mechanismus von das Dispatchen von Ereignissen mit Select zu realisieren

Repräsentiert einen bei Select registrierten Channel




*Sequenzdiagramm*

# Selector / Selectable Channels

## Vorgehen – Übersicht

- Kanäle (SelectableChannels) werden bei einem Selector registriert
- Multiplexing der Kanäle:
  Auswahl der I/O-bereiten Kanäle mit (bockierendem) selector.select();
- SelectionKey ist eine Beziehung zwischen selector und Channel+Ereignisart
- Eine Teilmenge von I/O-bereiten (*ready)* Kanälen wird ausgewählt *(select)* aus der Menge registrierten Kanäle (selector.select())
  - selector.selectedKeys() : Selected Key-Set
    enthält jetzt die Keys mit nicht-leerem Ready Sets
- Jeder SelectionKey hat ein Interest-Set and a Ready-Set
  - Interest-Set (woran bin ich interessiert) z.B.: *accept, read, write, connect*
  - Ready-Set (was ist bereit): Teilmenge des Interest-Set

***Entspricht im Wesentlichen dem Vorgehen mit dem Systemaufruf select in Unix.***

# Selector / Selectable Channels

## Vorgehen – Details

- Erzeuge Selector-Objekt und registriere Kanäle
  register() Methode der Klasse SelectableChannel
  erzeugt SelectionKey
- rufe select() der Selector-Klasse
- Hole Selected-Set von Keys vom Selector
  Selected set: Registrierte Keys mit nicht-leeren Ready-Sets
  keys = selector.selectedKeys()
- Iteriere über Selected-Set
  - Prüfe Ready-Set (Ausführbare Operationen seit dem letzten select() )
  - Entferne den key aus dem Selected-Set ( iterator.remove() )
  - Bediene den Kanal key.channel() geeignet ( read, write, etc. )
- Bekunde Interesse an dem Ereignis “Bereitschaft für eine *Operation*”:
  - Neuer Key:
    SelectionKey key = channel.register(selector, SelectionKey.*Operation*);
  - Key existiert bereits:
    key.interestOps(SelectionKey.*Operation*);

# Selector / Selectable Channels

## SelectionKey

- **SelectionKey–Attachment**

  **mit einem Selection-Key kann ein beliebiges Objekt verbunden werden**

  ```
  SelectionKey aKey = aChannel.register(selector, ...);
  aKey.attach(anObject);
  ```

- **das Objekt kann später (nach einem select) wieder geholt werden**

  ```
  AClass anObject = (AClass) key.attachment();
  ```

  **das Attachment kann beispielsweise der Buffer sein, über den die Kommunikation über den Channel abgewickelt wird.**

- **Das Attachment vereinfacht die Arbeit mit Selektoren:**

  **beliebige anwendungsspezifische Informationen können mit einem**

  **SelectionKey** ***also mit*** **Kanal + Ereignisart / Ereignis**

  **verbunden werden**

## Beispiel TCP-Echo-Server (1)

```
import java.io.IOException;
import java.net.{ InetSocketAddress, ServerSocket }
import java.nio.{ ByteBuffer, CharBuffer }
import java.nio.channels.{ SelectionKey, Selector, ServerSocketChannel, SocketChannel }
import java.nio.charset.Charset;

import scala.collection.JavaConversions._

object TcpSelectServer_Main extends App {
  val ECHO_PORT = 4713

  val serverChannel = ServerSocketChannel.open()
  val serverSocket   = serverChannel.socket()
  val address    = new InetSocketAddress(ECHO_PORT)
  serverSocket.bind(address)

  serverChannel.configureBlocking(false); // async !

  val selector = Selector.open();    // create selector
  val acceptkey = serverChannel.register(selector, SelectionKey.OP_ACCEPT) // register channel

  while (true) {
    val num = selector.select();  // wait for an event
    val readyKeys = selector.selectedKeys(); // an event occurred

          . . . Ereignis verarbeiten . . .


  }
}
```

## Beispiel TCP-Echo-Server (2)

*. . . Ereignis verarbeiten . . .*

```
for (key <- readyKeys) { // key represents channel with event

  readyKeys.remove(key);  // event is processed: remove it
  if (key.isValid()) {
    if (key.isAcceptable()) { // channel is ready for accept

          Accecpt-Bereitschaft verarbeiten

    }

    if (key.isReadable()) {   // channel is ready for read

          Lese-Bereitschaft verarbeiten

    }
    if (key.isWritable()) {   // channel is ready for write

          Schreib-Bereitschaft verarbeiten

    }
  }
```

## Beispiel TCP-Echo-Server (3)

```
if (key.isAcceptable()) { // channel is ready for accept
  val server : ServerSocketChannel = key.channel().asInstanceOf[ServerSocketChannel]
  val client = server.accept()
  println("Accepted connection from " + client)
  client.configureBlocking(false) // asynv !
  val clientKey = client.register(selector, SelectionKey.OP_READ);
  val buffer = ByteBuffer.allocate(100);
  clientKey.attach(buffer);
}
```

***Accecpt-Bereitschaft verarbeiten:***
*accept ausführen und dabei neuen Channel erzeugen,*
*den Channel als lese-bereit registrieren,*
*Buffer erzeugen und als Attachment an den Key hängen*

```
if (key.isWritable()) {   // channel is ready for write
  val client = key.channel().asInstanceOf[SocketChannel]
  val output = key.attachment().asInstanceOf[ByteBuffer]
  output.flip();
  client.write(output);
  output.clear();
  key.interestOps(SelectionKey.OP_READ);
}
```

***Schreib-Bereitschaft verarbeiten:***
*Buffer ist Key-Attachment.*
*Buffer-Inhalt auf Channel schreiben.*
*Channel jetzt auf Lesebereit beobachten.*

## Beispiel TCP-Echo-Server (4)

```
if (key.isReadable()) {   // channel is ready for read

  val client: SocketChannel = key.channel().asInstanceOf[SocketChannel]
  val output = key.attachment().asInstanceOf[ByteBuffer]
  output.clear();
  val numRead = client.read(output);
  println(s"server received $numRead bytes")

  if (numRead <= 0) {
     println("Client closed connection");
     key.channel().close();
     key.cancel();

  } else {
     output.flip();
     val charBuffer = Charset.defaultCharset().newDecoder().decode(output);
     output.clear();
     output.put((charBuffer.toString()+"\n").getBytes());
     key.interestOps(SelectionKey.OP_WRITE);
  }
}
```

***Lese-Bereitschaft verarbeiten:***
***Buffer zurücksetzen, und Daten aus Channel in Buffer lesen.***
***Falls nichts lesbar: Partner hat Verbindung geschlossen: Channel schließen, Key entfernen.***
***Sonst: Buffer auslesen, Daten verabeiten, Buffer füllen und Schreibreitschaft des Channels beobachten.***

## Selector Bewertung

- Asynchrones Warten auf BS-Ebene auf eines von vielen möglichen Ereignissen
- Bearbeitung des Ereignisses (synchron oder asynchron) wird auf Anwendungsebene organisiert
- Prinzip: *low-level* Form der Ereignisverarbeitung
  - Registrieren von Datenstrukturen,
  - Warten,
  - Untersuchen von Datenstrukturen und verarbeiten der Ereignis-Effekte

# Asynchrone Kommunikation mit NIO.2

## NIO.2 (ab Java 7) – Neue Interfaces und Klassen:

- Interface java.nio.AsynchronousChannel

  Ein Channel der Zugang gibt zu asynchronen I/O-Operationen, entweder auf Dateien, oder auf Sockets

- Interface java.nio.NetworkChannel

  Ein Channel der Zugang zu einem Socket bietet

- Klassen die NetworkChannel und AsynchronousChannel implementieren:
  - java.nio.AsynchronousServerSocketChannel
  - java.nio.AsynchronousSocketChannel
  - java.nio.AsynchronousByteChannel

  Diese Klassen sind keine Unterklassen der entsprechenden nicht asynchronen Klassen

- Klasse java.nio.AsynchronousChannelGroup

  Eine Gruppe von Channels.

## Asynchrone I/O in Java: NIO.1 oder NIO.2

2 Möglichkeiten:

- NIO.1: (nicht-asynchrone) Channels im nicht-blockierenden Modus + Selector
- NIO.2: Asynchrone Channels

# NIO.2: Asynchrone I/O mit *AsynchronousXY*

**unterstützt mit den Klassen:**

**AsynchronousXY, (AsynchronousChannel, ...)**

**nur TCP-Kommunikation**

**UDP-Kommunikation wird nicht unterstützt**

- **auch nicht von AsynchronousSocketChannel**
- **es gab kurzzeitig in einigen *Pre-Release* Java-7 Versionen einen AsynchronousDatagramChannel,**
  **dieser wurde aber wieder zurück gezogen**
  **und ist in Java 8 nicht mehr vorhanden**

## NIO.2 Asynchrone I/O in zwei Formen

- **Future Style**

  **nutzt** java.util.concurrent.**Future**

- **Callback Style**

  **nutzt** java.nio.channels.**CompletionHandler**

# Asynchrone I/O *Future-Style: Future-Style*

## Beispiel: Datei asynchron lesen – Future Style

```
import java.nio.ByteBuffer
import java.nio.channels.AsynchronousFileChannel
import java.nio.charset.Charset;
import java.nio.file.{ Path, Paths }
import java.nio.file.StandardOpenOption

import java.util.concurrent.Future
import scala.collection.JavaConversions._

object AsyncFileRead_FutureStyle_Main extends App {
  val buffer = ByteBuffer.allocate(1024);
  val filePath = Paths.get("/Users/thomasletschert/Desktop/Test.m");

  val asynchronousFileChannel
        = AsynchronousFileChannel.open(filePath, StandardOpenOption.READ)
  val result: Future[Int] = asynchronousFileChannel.read(buffer, 0).asInstanceOf[Future[Int]]

   //Do something else while reading:
   Thread.sleep(100);

   while (!result.isDone()) { // check whether read completed
      //Do something while still reading:
      Thread.sleep(100);
   }
   println("Read done: " + result.isDone());
   println("Bytes read: " + result.get());

   buffer.flip();
   print(Charset.forName(System.getProperty("file.encoding")).decode(buffer));
   buffer.clear();
}
```

**Future „des Systems“, kein Threadpool involviert! - Auch nicht implizit.**

**Synchrone Weiterverabeitung der Daten**

# Asynchrone I/O*: Future-Style*

## Beispiel: Datei lesen, Future-Stil mit Hilfe von *CompletableFuture*

```
import java.nio.ByteBuffer
import java.nio.channels.AsynchronousFileChannel
import java.nio.charset.Charset
import java.nio.file.{ Path, Paths }
import java.nio.file.StandardOpenOption
import java.util.concurrent.{ Future, CompletableFuture }

import scala.collection.JavaConversions._

object AsyncFileRead_CompletionStyle_Main extends App {

  val buffer = ByteBuffer.allocate(1024);
  val filePath = Paths.get("/Users/thomasletschert/Desktop/test.m");

  val asynchronousFileChannel = AsynchronousFileChannel.open(filePath, StandardOpenOption.READ)

  val result: Future[Int] = asynchronousFileChannel.read(buffer, 0).asInstanceOf[Future[Int]]

  CompletableFuture.runAsync(new Runnable {
      override def run(): Unit = {
        System.out.println("Bytes read: " + result.get());
        buffer.flip();
        print(Charset.forName(System.getProperty("file.encoding")).decode(buffer));
        buffer.clear();
      }
  });

  Thread.sleep(2000); // do something else
}
```

**Komplett asynchron: asynchrone Weiterverarbeitung der gelesenen Daten**

# Asynchrone I/O: *Future-Style / CompletableFuture*

## Java 8: CompletableFuture

- Mit Java 8 eingeführte Klasse
- Entspricht Scala  Promise + Future
- Erlaubt die Verkettung von Futures

  z.B. mit runAsync

| | |
|---|---|
| `static CompletableFuture<Void>` | **`runAsync`**`(`**`Runnable`** `runnable)`<br>Returns a new CompletableFuture that is asynchronously completed by a task running in the **`ForkJoinPool.commonPool()`** after it runs the given action. |

# Asynchrone I/O: Java-Future => Scala-Future

**Um in Scala-Stil mit Futures arbeiten zu können, müssen Java-Futures in Scala-Futures umgewandelt werden**

**Beispiel (1):**

```
import java.nio.ByteBuffer
import java.nio.channels.AsynchronousFileChannel
import java.nio.charset.Charset
import java.nio.file.{ Path, Paths }
import java.nio.file.StandardOpenOption

import java.util.concurrent.{ Future => JavaFuture }

import scala.concurrent.{ Future => ScalaFuture, Promise, ExecutionContext }

import scala.util.{ Success, Failure }
import ExecutionContext.Implicits.global
import scala.collection.JavaConversions._

object AsyncFileRead_FutureStyle_B_Main extends App {

  val buffer = ByteBuffer.allocate(1024);
  val filePath = Paths.get("/path/to/some/file.m");

  val asynchronousFileChannel
        = AsynchronousFileChannel.open(filePath, StandardOpenOption.READ)
```

# Asynchrone I/O: Java-Future => Scala-Future

**Um in Scala-Stil mit Futures arbeiten zu können, müssen Java-Futures in Scala-Futures umgewandelt werden**

**Beispiel (2):**

Promise erzeugen

```
val promise = Promise[Int]

ScalaFuture {
  promise.complete({
      try {
        Success(
            asynchronousFileChannel.read(buffer, 0).asInstanceOf[JavaFuture[Int]].get()
        )
      } catch {
        case t: Throwable => Failure(t)
      }
  })
}
```

Scala Future erzeugen

```
val scalaFuture = promise.future
```

Scala übliche Verkettung verwenden (statt runAsync von Java 8)

```
ScalaFuture. foreach ( result => {
      println("Bytes read: " + result)
      buffer.flip();
      print(Charset.forName(System.getProperty("file.encoding")).decode(buffer));
      buffer.clear();
  }
)

Thread.sleep(2000); // do something else

}
```

# Asynchrone I/O *Callback-Style*

## Beispiel: Datei lesen – 2: Callback-Stil mit *CompletableFuture*

```
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousFileChannel;
import java.nio.channels.CompletionHandler;
import java.nio.charset.Charset;
import java.nio.file.Path;
import java.nio.file.Paths;

object AsyncFileRead_CallbackStile_Main extends App {
  val buffer = ByteBuffer.allocate(1024);
  val filePath = Paths.get("/Users/some/path/test.m");
  val asynchronousFileChannel = AsynchronousFileChannel.open(filePath, StandardOpenOption.READ)
  asynchronousFileChannel.read(
        buffer, // destination
        0,      // position
        buffer, // attachment: to be used in completionHandler
        new CompletionHandler[Integer, ByteBuffer]{ // callback: what to do after a read
            override def completed(result: Integer, attachment: ByteBuffer): Unit = {
                println("Nr of Bytes read: " + result);
                buffer.flip();
                println(Charset.forName(System.getProperty("file.encoding")).decode(attachment));
                buffer.clear();
             }

             override def failed(exc:Throwable, attachment: ByteBuffer): Unit = {
                        println("Operation failed"); exc.printStackTrace();
                    }
        })

        Thread.sleep(10000)

}
```

# Asynchrone I/O

## Future vs Callback-Stil

```
Future<V> future  = asyncChannel.operation(...)

asyncChannel.operation(...
                A attachment,
                CompletionHandler<V,? super A> callback)
```

**Das Attachment wird an die complete-Methode des Handlers ausgeliefert**

**Ein beliebiges Objekt kann übergeben werden, z.B. der buffer**

```
CompletionHandler<V, A> completionHandler= new CompletionHandler<V, A>() {

   @Override
   public void completed(V result, A attachment) {
     result:   das Ergebnis der read-Operation
     attachment: das Attachment
   }

   @Override
   public void failed(Throwable exc, A attachment) {
     exc:           die Exception der Operation
     attachment: das Attachment
   }

};
```

## Asynchrone Netz-I/O / Beispiel: TCP-Echo-Server im Future-Stil (1)

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.{ ByteBuffer, CharBuffer }
import java.nio.channels.{ AsynchronousServerSocketChannel, AsynchronousSocketChannel }
import java.nio.charset.Charset;

import scala.concurrent._
import ExecutionContext.Implicits.global

object TcpFutureServer_Main extends App {
  val ECHO_PORT = 4713

  val asynchronousServerSocketChannel = AsynchronousServerSocketChannel.open()

  if (! asynchronousServerSocketChannel.isOpen()) { System.err.println("can to open channel"); System.exit(-1); }

  asynchronousServerSocketChannel.bind(new InetSocketAddress("127.0.0.1", ECHO_PORT));
  println("Server ready Waiting for connections");

  while (true) {
    val asynchronousSocketChannelFuture = asynchronousServerSocketChannel.accept();
    val asynchronousSocketChannel = asynchronousSocketChannelFuture.get();


          . . . Client-Verbindung  bedienen  . . .


    }
  }
```

# Asynchrone Netz-I/O / Beispiel TCP-Echo-Server im Future-Stil (2)

*Client-Verbindung bedienen*

```
Future {
  val host = asynchronousSocketChannel.getRemoteAddress().toString();
  println("Incoming connection from: " + host)
  val buffer = ByteBuffer.allocateDirect(32)
  var i: Int = 0
  var loop = true
  while (loop) {
     i = i+1
     val fnr = asynchronousSocketChannel.read(buffer)
     val bytesRead = fnr.get()
     if (bytesRead < 0) {
        println("Read " + bytesRead + " Bytes: STOP")
        loop = false
     } else {
        println("Server has read " + bytesRead + " Bytes")
        buffer.flip();
        val charBuffer = Charset.defaultCharset().newDecoder().decode(buffer)
        println("Server received from Client : " + charBuffer.toString());
        if (buffer.hasRemaining()) {
          buffer.compact();
        } else {
          buffer.clear();
        }
        println("Server will send back to Client: " + charBuffer.toString());
        buffer.put((charBuffer.toString()+"\n").getBytes());
        val fnw = asynchronousSocketChannel.write(ByteBuffer.wrap((charBuffer.toString()+"\n").getBytes()));
        println("Server send back " + fnw.get() + " Bytes");
        println("Server try again reading");
     }
  }
  println("Server: Connection to " + host + " will be closed");
  asynchronousSocketChannel.close();
}
```

# Asynchrone Netz-I/O

## TCP-Echo-Server im Callback-Stil

### Übungsaufgabe für Interessierte

*„Netty is an asynchronous event-driven network application framework*
*for rapid development of maintainable high performance protocol servers & clients.“*
[http://netty.io/]

## NIO Frameworks

**NIO / NIO.2 ist in der Anwendung komplex und gelegentlich un-intuitiv**

**Ernsthafte Serveranwendungen nutzen NIO/NIO.2**

- **Allerdings i.d.R. über ein Framework**
- **Bekanntestes Framework: Netty**

## Netty

- **Java Client/Server Networking Framework**
- **Weit verbreitet**
  **Kern vieler Java-Web-Frameworks – auch „leichtgewichtiger“ (= Non-JEE) Frameworks**
- **Projekt**
  **http://netty.io/**
- **Kennzeichen**
  **extensive Nutzung von asynchroner / nicht-blockiernder IO**
  **bietet API zum einfachen Umgang mit NIO-Features**

# NIO Frameworks und Reaktive Server-Architekturen

**NIO ist die Basis für reaktive Server-Architekturen**

## Reaktive Server-Architekturen – Basis

- **Reactor-Pattern** ~ 1995

  D. C. Schmidt: *Reactor, an Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events*

  http://www.cs.wustl.edu/~schmidt/PDF/reactor-siemens.pdf

  siehe auch Acceptor-Connector Pattern:

  D. Schmidt: *Acceptor-Connector: An Object Creational Pattern for Connecting and Initializing Communication Service*s

  http://www.dre.vanderbilt.edu/~schmidt/PDF/Acc-Con.pdf

- **Staged Event Handling** ~ 2005

  M. Welsh: *An Architecture for Highly Concurrent, Well-Conditioned Internet Services*

  http://www.mdw.la/papers/mdw-phdthesis.pdf

**NIO-Frameworks basieren auf diesen Prinzipien (auch heute noch, auch Netty)**

## Reactor-Pattern

**Muster zur Implementierung des Szenarios:**

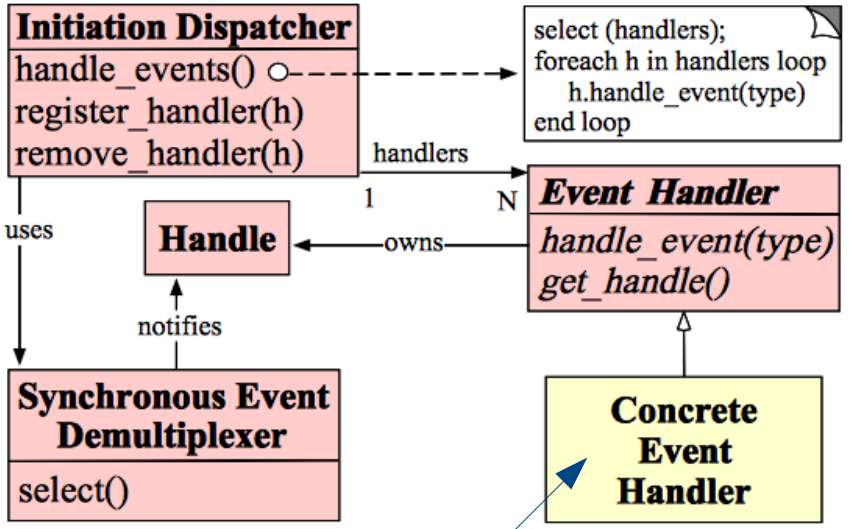
- ein Thread führt alle I/O Operationen aus
- I/O-Ereignisse müssen einer zustandsbehafteten Verbindung / Sitzung zugeordnet und behandelt werden

**Bestandteile**

- Event Demultiplexer / Dispatcher nehmen alle I/O-Ereignisse an und leiten sie an den zuständigen Event-Handler weiter
- Event-Handler (Endlicher Automat / FSM) verarbeiten die Ereignisse und gehen dabei in einen neuen Zustand über

**Beispiel**

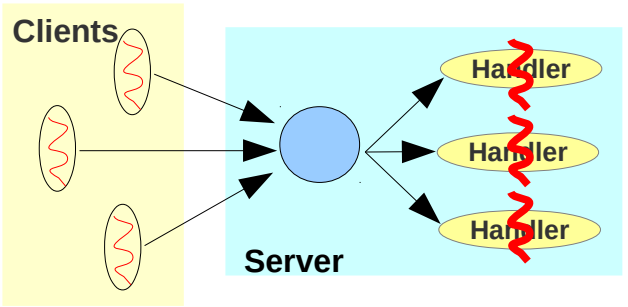
- siehe Selector-Beispiel oben



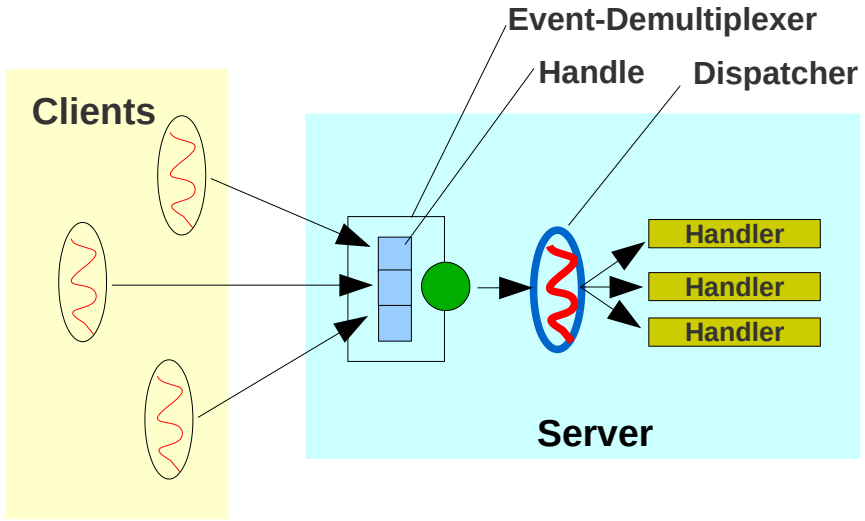

aus:
http://www.cs.wustl.edu/~schmidt/PDF/reactor-siemens.pdf

ein Automat (FSM)

## Reactor-Pattern




*„klassische“ Struktur*




*Reactor-Struktur*
*Grundform*

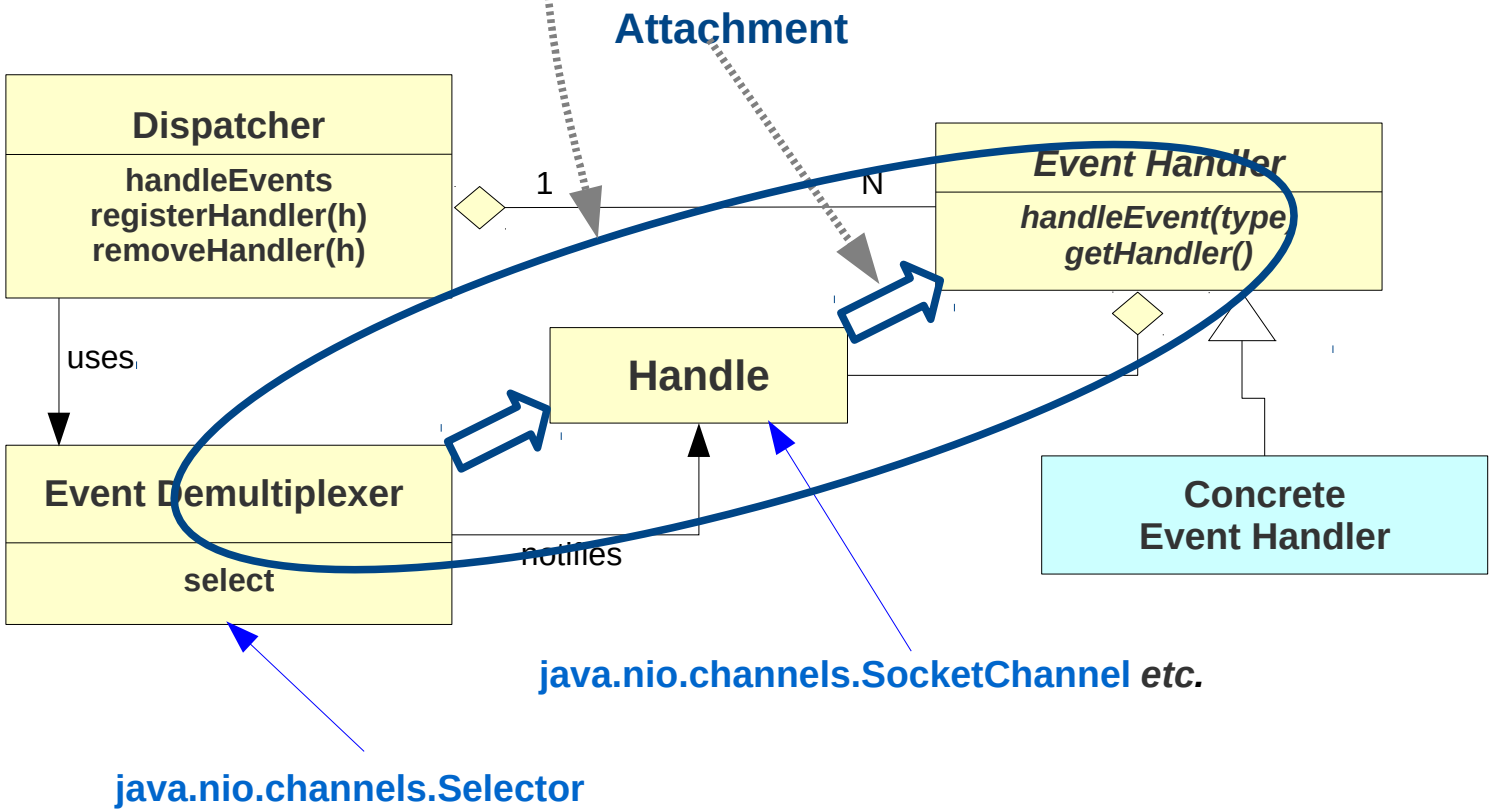
## Reactor-Pattern

**Implementierung mit**

- **Selector** als Demultiplexer
- **Channels** als Handles
- **SelectionKeys** als Verbindung

**SelectionKey**
entsteht bei / repräsentiert Registrierung
- eines Handles (Channels)
- bei einem Selector

kann beliebiges Attachment verwalten,
z.B. den Handler der zu aktivieren ist, wenn das Ereignis eintrat, auf das Registrierung erfolgte

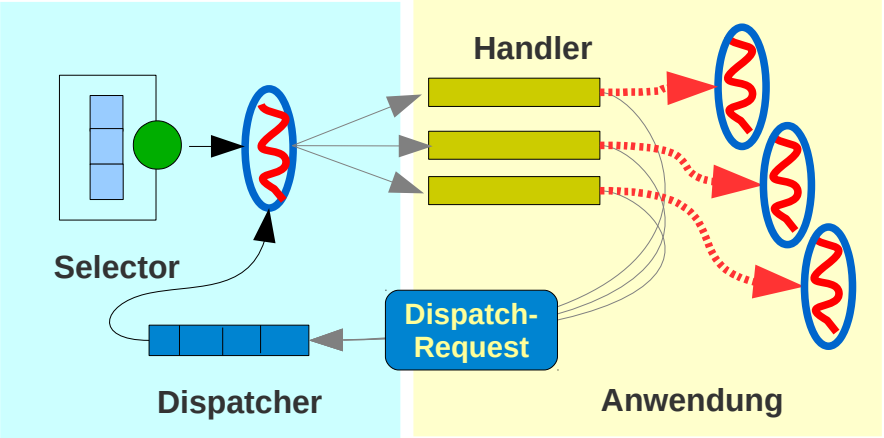


# Reactor-Pattern

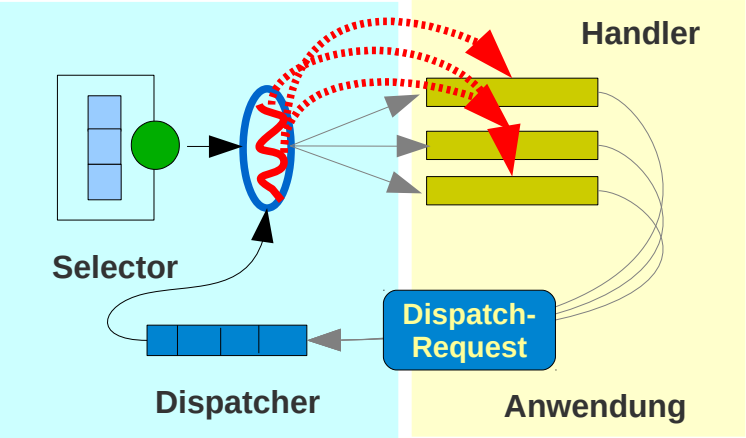
## Implementierung

Der *Dispatcher* nimmt Registierungs- und Veränderungswünsche der *Handler* an.

Diese werden in einer Queue gespeichert und von der *run*-Methode zu passenden Zeitpunkten abgearbeitet. (Vergleiche „invokeLater“ in Swing)




„klassische“ Implementierung:
Jeder Handler läuft in seinem Thread.




Einfache Reactor-konforme Implementierung:
Alle Aktionen laufen im Kontext des Dispatcher-Threads

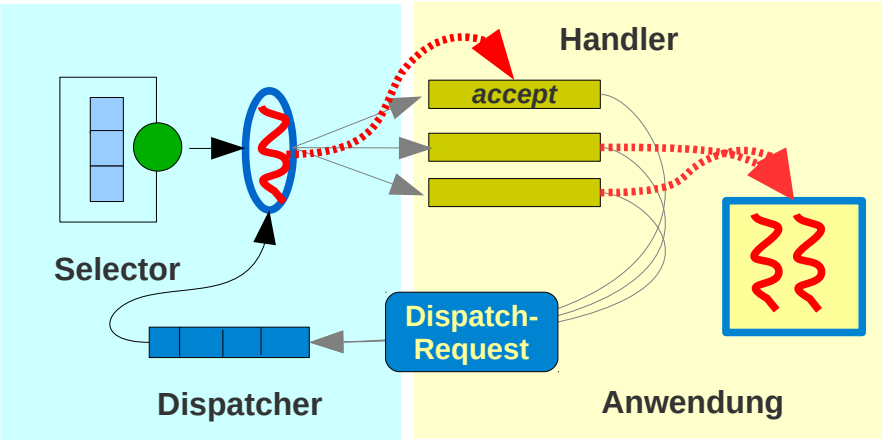
# Reactor-Pattern

## Implementierung

Die Abwicklung der Handler kann auf unterschiedliche Arten Threads zugeordnet werden

Eine naheliegende Lösung ist die Verwendung eines Pools

Das entspricht der Intention des Musters: dynamische und ungebundene Thread-Erzeugung wird verhindert



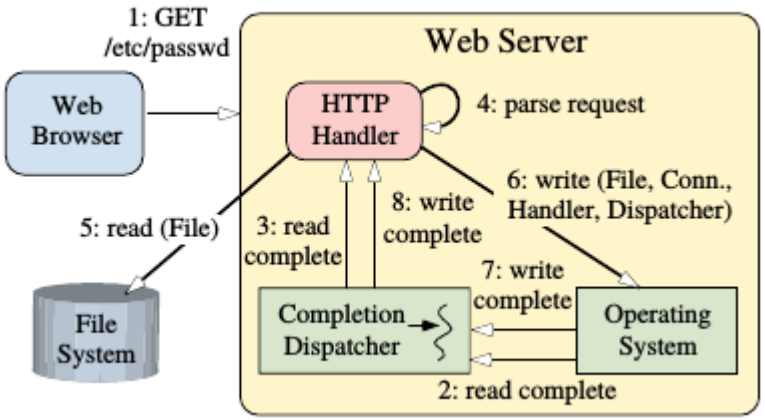

**Typische Concurrency-Implementierung des Reaktor-Musters:**
Accept-Handler agiert im Kontext des Dispatcher-Threads
alle anderen Handler agieren im Kontext eines Threads aus einem Pool.

## Proactor-Muster – Variante des Reactor-Musters

**Steigerung der Asynchronität:**

**Asynchrone vom BS ausgeführte Aktivitäten melden ihre Beendigung mit Completion-Events einem Completion-Dispatcher, der dann die Weiterbearbeitung an einen Completion-Handler delegiert.**

**vergl. NIO-2: CompletionHandler !**




**Webserver nach dem Proactor-Muster**

**aus: Irfan Pyarali, Tim Harrison, Douglas C. Schmidt, Thomas D. Jordan : *Proactor, An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events***

# Staged Event Handling Architektur

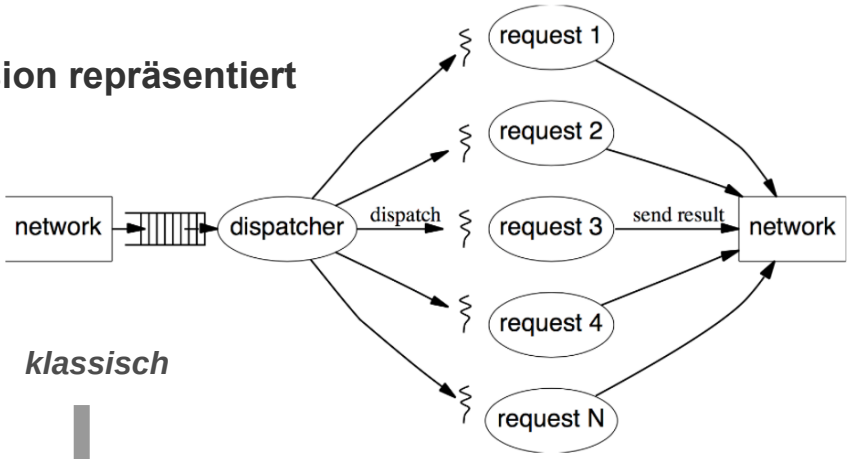
**Ersetze den Zustandsautomat, der eine Verbindung / Session repräsentiert**

**durch eine Verarbeitungskette aus**

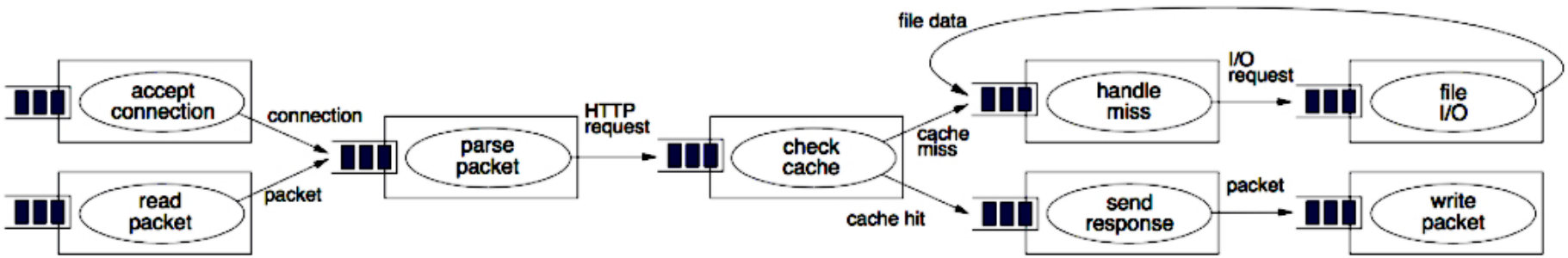
- **Stages die via**
- **Queues verbunden sind**

**Jede Stage hat ihren eigenen Thread-Pool**

**Basis-Arbeit: Matt Welsh (jetzt Google)**




*klassisch*

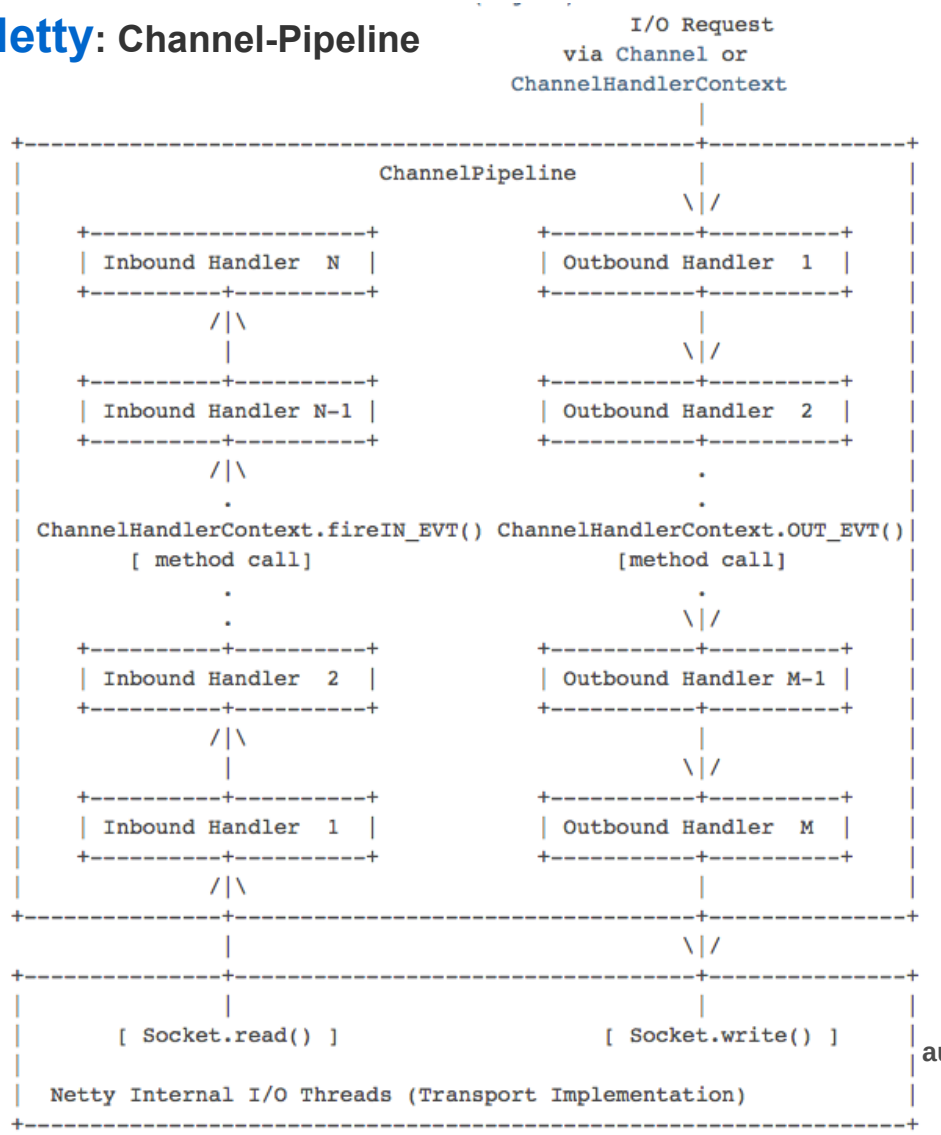



*staged*

**Graphiken aus:**
**https://github.com/mdwelsh/mdwelsh.github.io/blob/master/talks/seda-talk-amazon.pdf**
**https://github.com/mdwelsh/mdwelsh.github.io/blob/master/talks/seda-sosp01-talk.pdf**

## Staged Event Handling Architektur

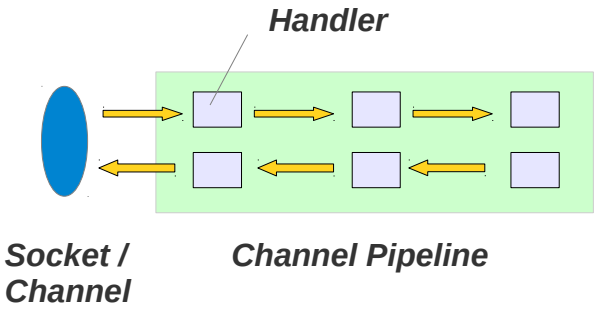
### Netty: Channel-Pipeline




aus: https://netty.io/4.1/api/io/netty/channel/ChannelPipeline.html

# Staged Event Handling Architektur

## Netty: Channel-Pipeline und Channel-Handler




### Channel Pipeline

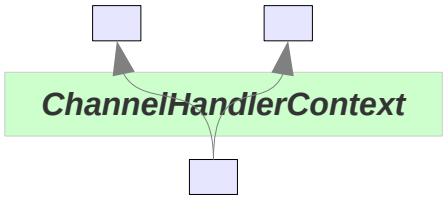
- Jeder Channel ist mit einer Channel-Pipeline verbunden
- Die Pipeline wird automatisch mit jedem Channel erzeugt
- Die Channel-Pipeline enthält Handler
- Die Handler verarbeiten die Events in einer Kette

### ChannelHandler

- Interface eines Handlers
- Die Channel Pipeline enthält Handler
- Die Handler verarbeiten die Events in einer Kette und
- interagieren über den ChannelHandlerContext

### ChannelHandlerAdapter

- Rahmenimplementierung von ChannelHandler
- i.A. Basis für eigene Implementierungen




### ChannelHandlerContext

- repräsentiert die Channel-Pipeline in der sich ein Handler befindet
- erlaubt Interaktion mit anderen Handlern in der Pipeline

# Staged Event Handling Architektur

## Netty

### Channel Handler: Inbound und Outbound

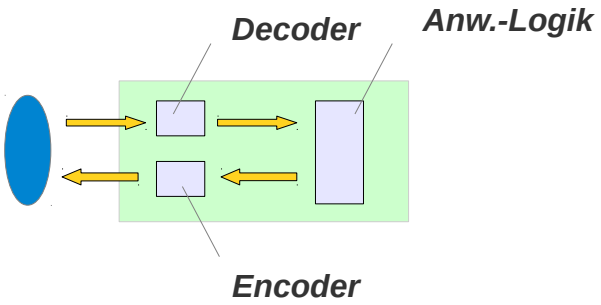
- Ein ChannelHandler kann Nachrichten verarbeiten die in beide Richtungen fließen
- Ältere Netty-Versionen unterscheiden
  - ChannelInboundHandler und
  - ChannelOutboundHandler

Diese Interfaces sind *deprecated.* Alle Handler sind sowohl Inbound als auch OutBound

### Standard Channel-Pipeline

- enthält

  Msg-Decoder / Msg-Encoder / Anwendungslogik




### Vordefinierte Decoder / Encoder

- Einige Decoder und Encoder sind vordefiniert
- z.B. StringDecoder und StringEncoder

### SimpleChannelInputHandler

Einfache Basisklasse für Anwendunglogik, die Nachrichten von einem bestimmten Typ verarbeite

## Reaktive Server mit und ohne NIO

### NIO

**Java-spezifische Variante um BS-Features zur reaktiven Ereignis-Verarbeitung der Anwendung zugänglich zu machen.**

### Betriebsystem

**Entscheidend sind die Fähigkeiten des Betriebssystems und wie gut auf sie zugegriffen werden kann – sie der Anwendung zur Verfügung gestellt werden können.**

**Der direkte Weg wären die entsprechenden Systemaufrufe (in C!):**

**select, poll, epoll, kqueue, …**

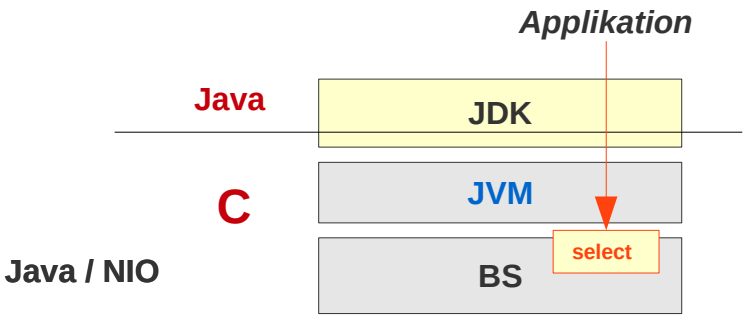
**Java: Steckt (plattform-spezifisch) in JDK / JVM**

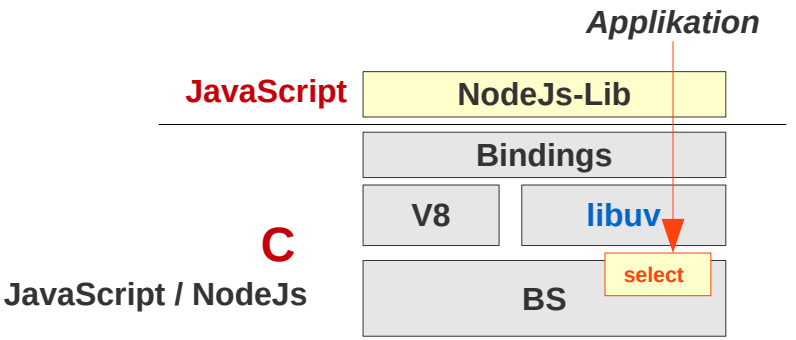
**Alternative, z.B. libuv (Kommunikationskern Kern NodeJs)*:**

**C/C++-Bibliothek (plattformspezifisch compiliert)**

**mit Hochsprachen-Interface**

**(JavaScript vor allem, aber auch andere)**






* https://github.com/libuv

# Literatur

***Netzwerkprogrammierung ist und war von Anfang an ein zentrales und wichtiges Anwendungsgebiet von Java.***

***Leider ist die Literaturlage bei aktuellen Einführungen in diese interessante und wichtige Thematik eher dünn.***

**Neben der (Java-8) API-Dokumentation beruht dieser Foliensatz im Wesentlichen auf:**

**Anghel Leonard: *Pro Java 7 NIO.2* , Apress, 2011**

**Netty wird beschrieben in:**

**Norman Maurer: *Netty in Action*, Manning, 2015**