

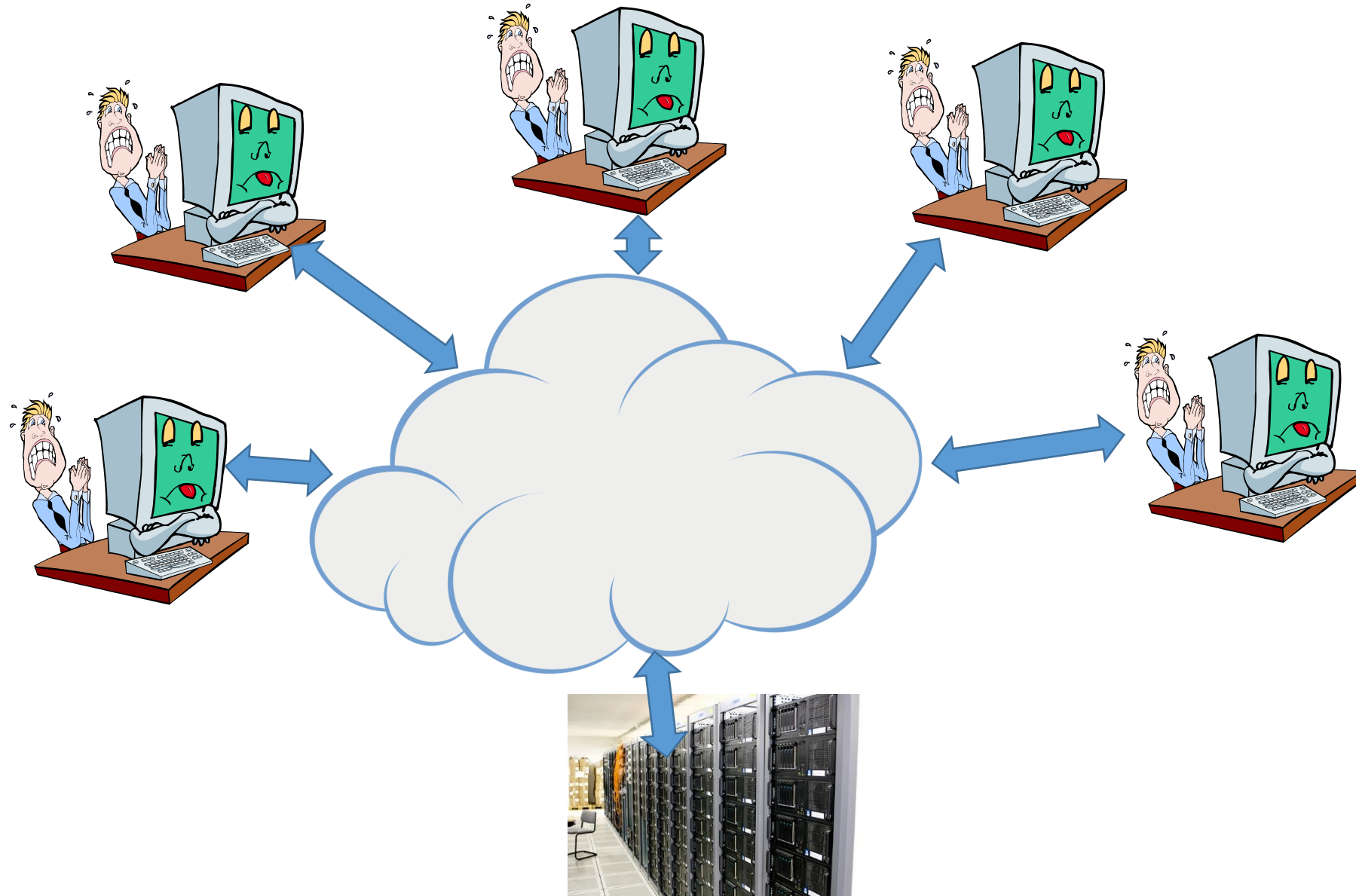
## Bachelor-Modul: Software Projekt

# Netzwerkkommunikation mit Java



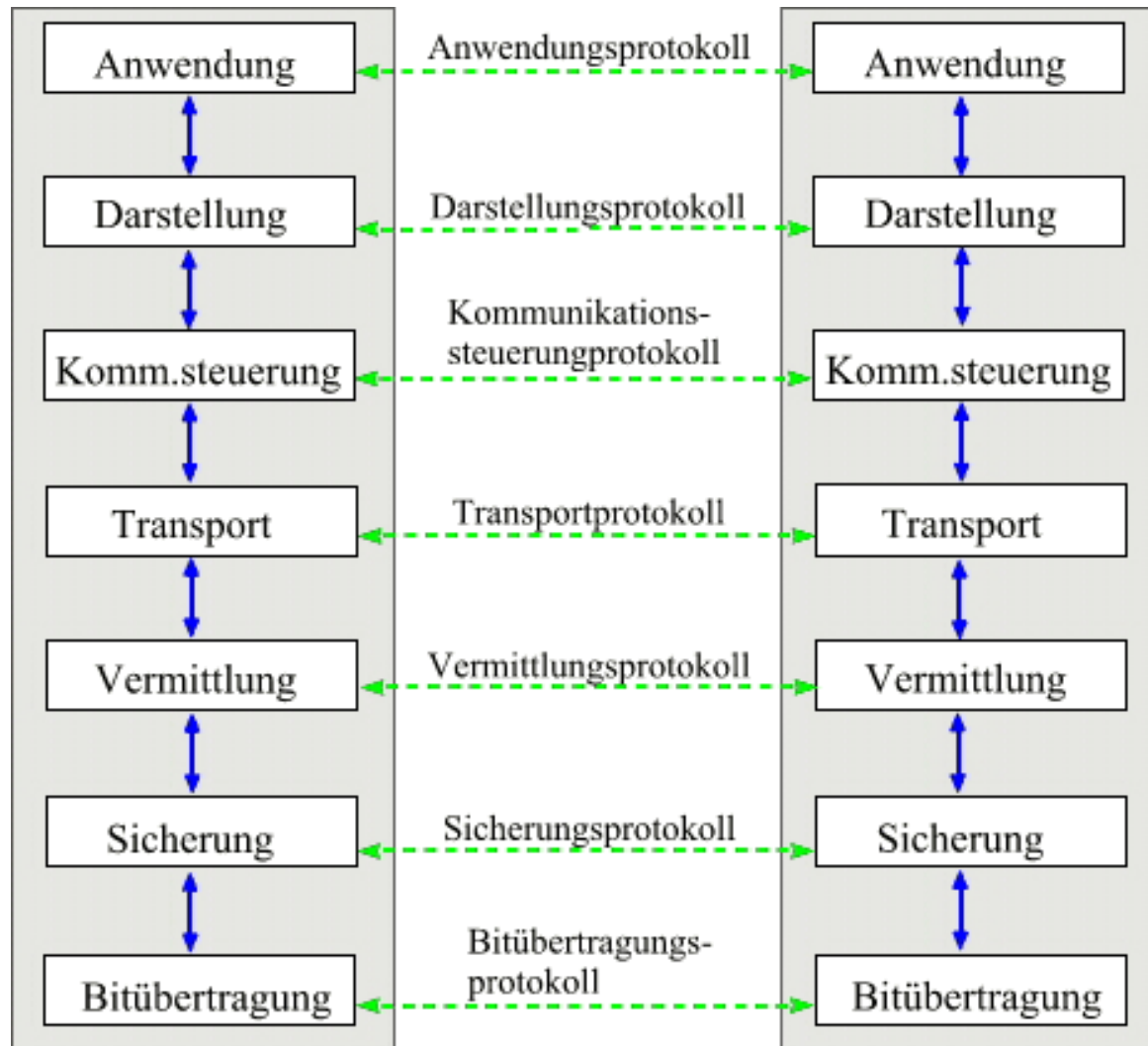
**Marco** Grawunder

- Netzwerk: Ports und Sockets
- Sockets in Java
- Beispiel: TimeEcho-Server mit Streams
- „Neuer“ Java Ansatz
- Netty
- Ausflug Ereignisgetriebene Ansätze
- Netty – Echo-Server, Grundlagen des „Bootstrapping“
- Beispiel für Client/Server-Kommunikation mit Netty und Nachrichten
- Literatur/Quellen:
  - Wilhelms/Kopp: Java Professional, MITP-Verlag, 1999
  - Bogner: Java in verteilten Systemen, dpunkt, 1999
  - Maurer/Wolfthal: Netty in Action, Manning, 2016

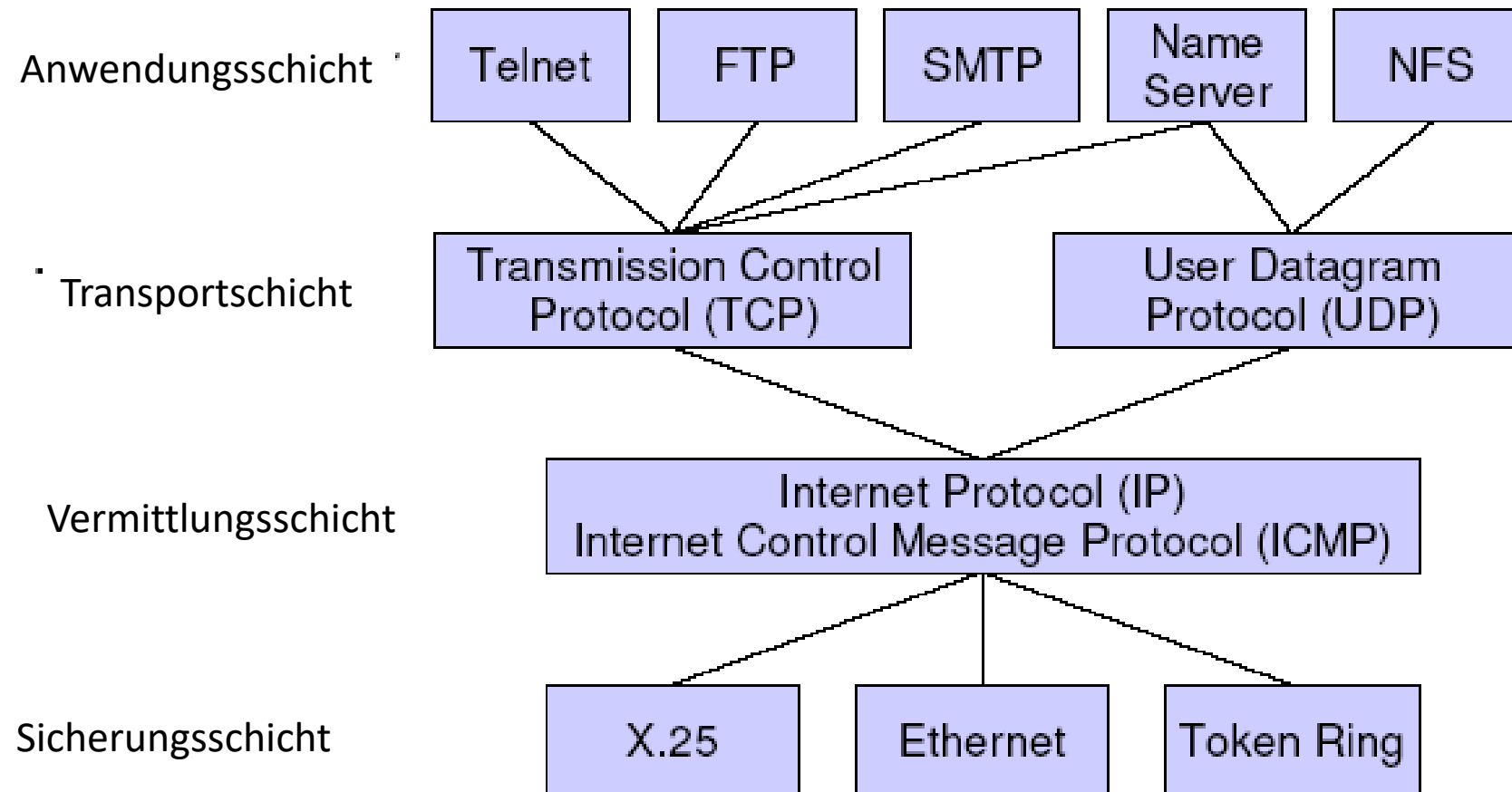


Software Projekt

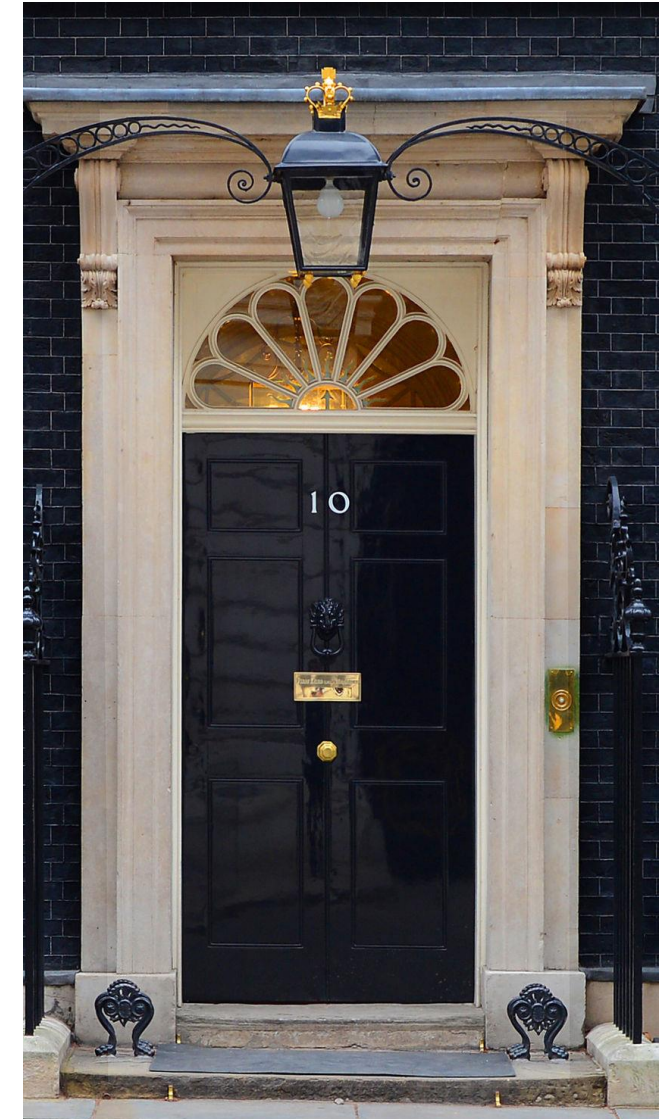
- Protokollfamilie zur Kommunikation
- Wurzeln: ARPANET, Vorgänger des Internets
- Ziel: sichere und zuverlässige Datenkommunikation auch im Krisenfall
  - **paketorientiertes** Netz
    - Unterteilung der Daten in kleine Portionen
    - unabhängige Übertragung über das Netz
  - Kommunikation wird bei Ausfall eines Knotens nicht zwingend unterbrochen, anderer Weg im Netzwerk möglich



- unterschiedliche Protokollschichten die unterschiedliche Aspekte der Datenübertragung verbergen
- eher theoretisch



- Rechner eines TCP/IP-Netzes werden eindeutig durch **IP-Adresse** (z.B: 134.106.52.240) identifiziert
- zu grob um einzelnen **Prozesse** auf Rechner ansprechen zu können
- **Portnummer** (16 Bit) ist die „Hausnummer“ eines Prozesses auf einem Rechnern
- wird vom Betriebssystem verwaltet und an Prozesse vergeben
- Viele „well know ports“, z.B: 80 für HTTP, 21 für FTP
- unter Unix: Port < 1024 sind Systemports und können i.d.R. nicht verwendet werden



- Ports stellen Kommunikationspunkte dar
- zusätzlich noch ein **Kommunikationskanal** notwendig
- wird durch **Socket** (engl. Sockel, Fassung) bereit gestellt
- genauer: Socket ist ein **Endpunkt** der **Kommunikationsverbindung** zweier Rechner → Client und Server haben Socket
- durch IP-Adresse und Portnummer identifiziert
- **Socket-Kommunikation stellt fundamentalsten Kommunikationsmechanismus dar**
- Ursprünglich für BSD-Unix entwickelt
- heute auf allen verbreiteten Plattformen verfügbar
- früher sehr schwierig:
  - Zugriff abhängig von BS
  - häufig sogar von Version zu Version unterschiedlich
- heute (u.a.) Java 😊



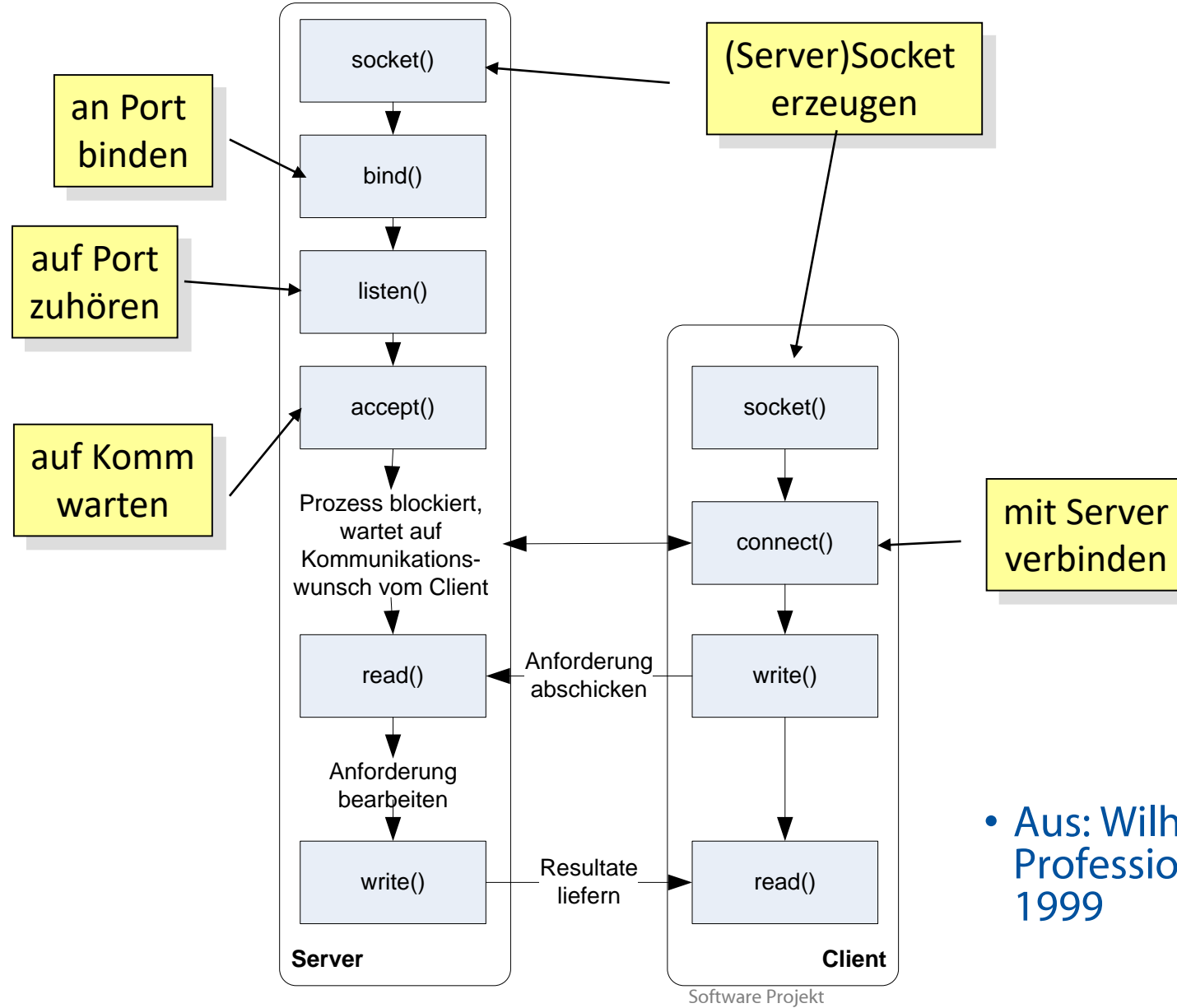


- Klassen im Paket `java.net.*`
- Abstrahiert vollkommen von darunter liegendem Betriebssystem
- Klasse `java.net.Socket` dient zum Erzeugen einer Socket
- Aufbau der Verbindung (zum Server) durch:
  - `Socket con = new Socket(ip, port)`
  - dort muss allerdings bereits eine (Server-)Socket eingerichtet sein
- Erzeugen einer **Server-Socket**
  - `ServerSocket server = ServerSocket(port)`



- bei Verbindungsaufbau zum Server, muss dieser reagieren und den Verbindungsaufbau akzeptieren (Methode `accept()`)
  - Durch `accept` wird **Socket auf Server** erzeugt
  - typisch:
    - Warten auf Verbindung in Endlosschleife
    - `accept()` blockiert bis Verbindungswunsch auftritt
- ```
while (true){  
    Socket client = server.accept();  
    ..  
}
```





- Aus: Wilhelms/Kopp: Java Professional, MITP-Verlag, 1999

- Kommunikation in Sockets über Streams
- Streams (Ströme) sind Abstraktionen für beliebige Datenströme:
  - der Konsole (`System.out`, `System.err`, `System.in`)
  - dem Filesystem
  - und natürlich auch von bzw. zu einem Socket
- in Package `java.io`
- Stream für Eingabe: `InputStream`
- Stream für Ausgabe: `OutputStream`
- Klassen sind abstrakt und werden durch konkrete Implementierungen umgesetzt

```
public class TimeServer {  
  
    public static void main(String[] args) throws IOException  
    {  
        int port = 1234;  
        ServerSocket server = new ServerSocket(port);  
        while (true){  
            System.out.println("Waiting for client ....");  
            Socket client = server.accept();  
            System.out.println("Client from "+client.getInetAddress()+" connected");  
            OutputStream out = client.getOutputStream();  
            Date date = new Date();  
            byte b[] = date.toString().getBytes();  
            out.write(b);  
        }  
    }  
}
```

Neue ServerSocket auf  
Port erstellen

Auf Client warten.  
Ergebnis ist ein Socket

OutputStream um zum  
Client zu schreiben

Nachricht zum Client  
schicken

Einfache Streams  
können nur **bytes**  
verarbeiten

```
public class TimeClient {  
  
    public static void main(String[] args)  
        throws UnknownHostException, IOException {  
        Socket server = new Socket("localhost", 1234);  
        System.out.println("Connected to " + server.getInetAddress());  
        InputStream in = server.getInputStream();  
        byte b[] = new byte[100];  
        int num = in.read(b);  
        String date = new String(b);  
        System.out.println("Server said: " + date.substring(0, num));  
    }  
}
```

Neue Socket-Verbindung zum  
Rechner auf Port erstellen

InputStream um vom Server  
zu lesen

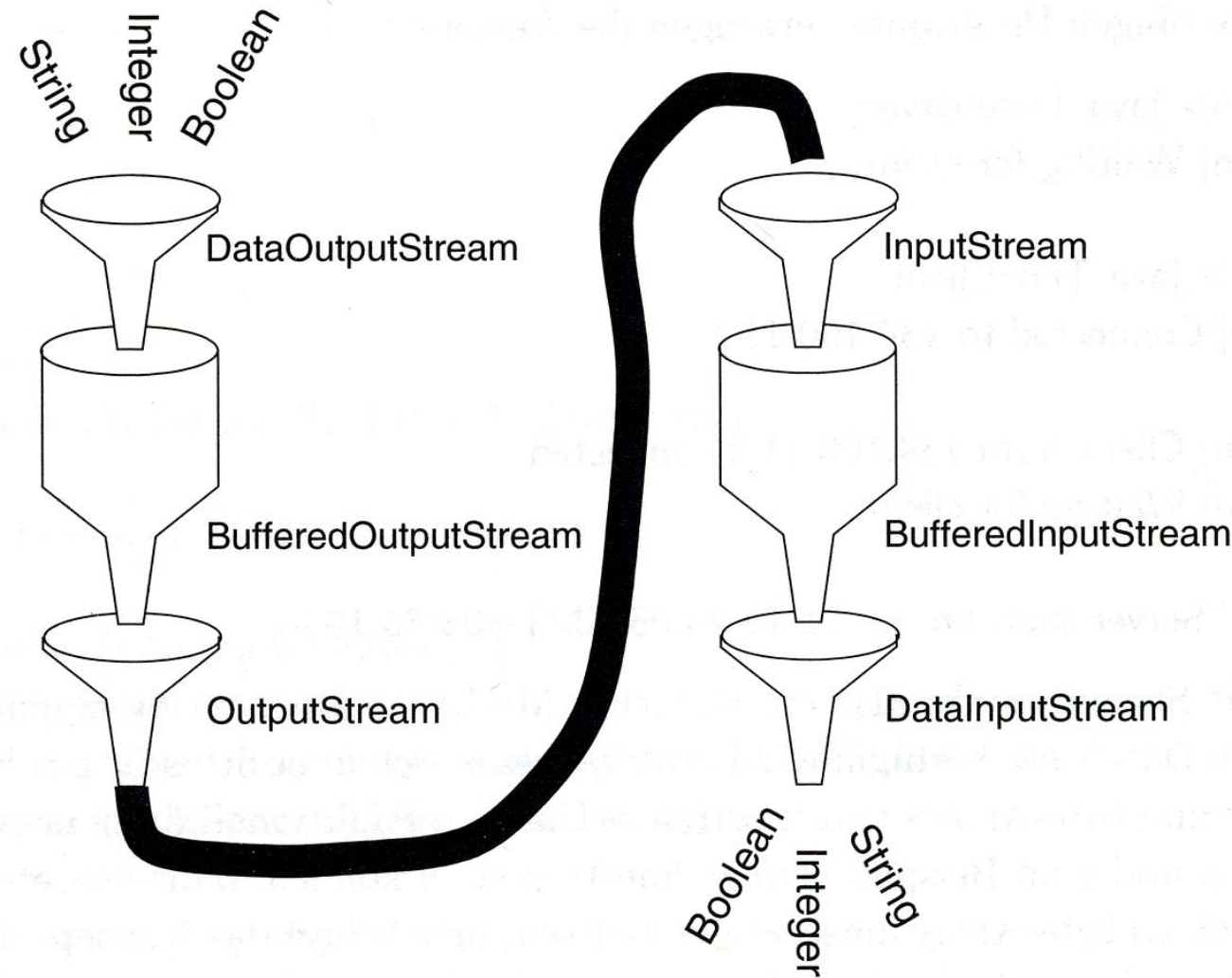
Lesen

Puffer groß genug wählen

- Streams bieten einfachen Mechanismus zur Übertragung von Daten
- es lassen sich allerdings nur `bytes` und `byte`-Arrays transportieren → **Paketorientiertes TCP/IP Protokoll!**
- Zur Erweiterung bietet Java Filter (`FilterInputStream` und `FilterOutputStream`)
- Lassen sich hintereinander schalten
- Vordefinierte Filter für einfache Datentypen:
  - `DataOutputStream`
    - **Methoden:** z.B. `writeInt()`, `writeChar()`, `writeUTF()`
  - `DataInputStream`
    - **Methoden:** z.B. `readInt()`, `readChar()`, `readUTF()`
  - kümmern sich im Wesentlichen um die korrekte **Umsetzung** bzw. **Rekonstruktion von einfachen Typen** in bzw. aus Byte-Folgen

- **Steigerung der Effizienz:**
  - nicht jedes Byte einzeln übertragen
  - erst wenn ein gewisse Menge angefallen ist, Übertragung starten
  - weniger Overhead
- **Klassen in Java**
  - `BufferedInputStream`
  - `BufferedOutputStream`
- **Buffer müssen nicht immer gleich groß sein**
  - Standard bei `BufferedOutputStream` 500 Bytes
  - Methode `flush()` versendet Buffer sofort





Aus: Bogner: Java in verteilten Systemen, dpunkt, 1999

```
public class TimeServer {  
    public static void main(String[] args) throws IOException {  
        int port = 1234;  
        ServerSocket server = new ServerSocket(port);  
        while (true){  
            System.out.println("Waiting for client ....");  
            Socket client = server.accept();  
            System.out.println("Client from "+client.getInetAddress()+" connected");  
            DataOutputStream out =  
                new DataOutputStream(  
                    new BufferedOutputStream(client.getOutputStream()));  
            Date date = new Date();  
            out.writeUTF(date.toString());  
            out.flush();  
        }  
    }  
}
```

```
public class TimeClient {  
    public static void main(String[] args)  
        throws UnknownHostException, IOException {  
        Socket server = new Socket("localhost",1234);  
        System.out.println("Connected to "+server.getInetAddress());  
        DataInputStream in =  
            new DataInputStream(  
                new BufferedInputStream(server.getInputStream()) );  
        System.out.println("Server said: "+in.readUTF());  
    }  
}
```

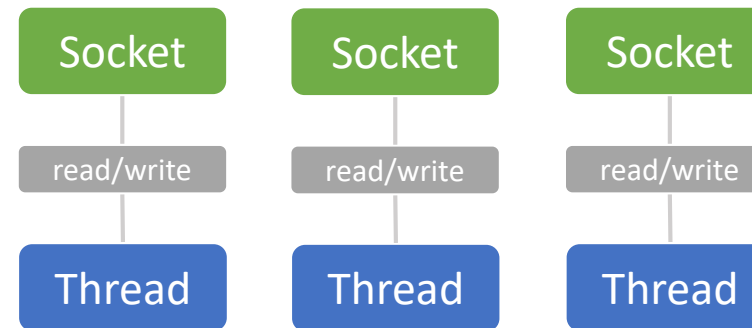
- Probleme:

- accept blockiert so lange, bis sich ein Client verbindet
- Man kann so naiv nur eine einzelne Verbindung gleichzeitig behandeln

```
ServerSocket server = new ServerSocket(port);  
while (true) {  
    System.out.println("Waiting for client ....");  
    Socket client = server.accept();  
}
```

- Ansatz:

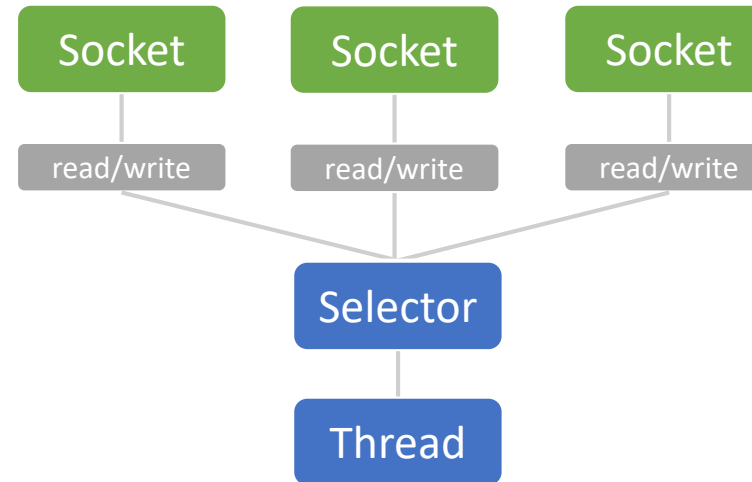
- Erzeuge für jede neue Socket-Verbindung einen neuen Thread



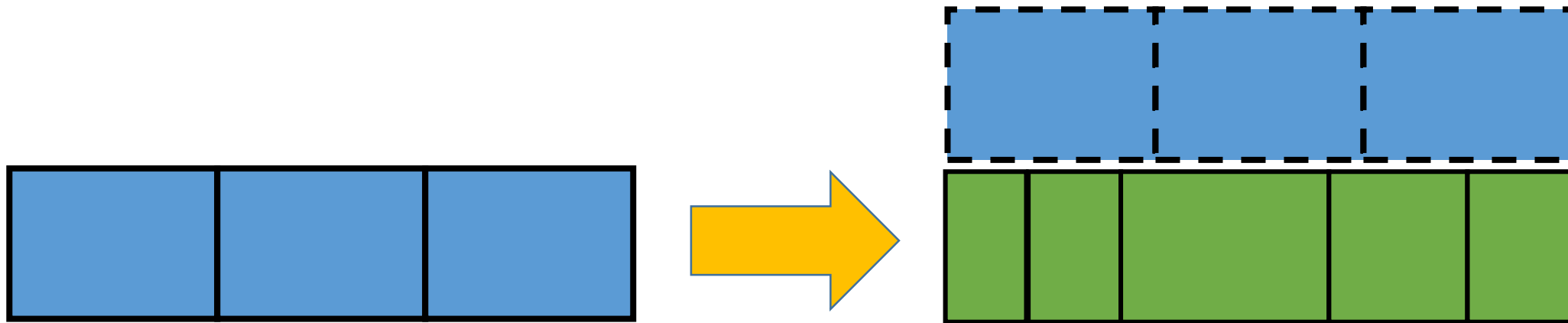
- Probleme:

- Viele der Threads warten nur auf Eingaben
- Jeder Thread benötigt einige Basisressourcen (bis zu 1MB)
- Thread-Anzahl ist beschränkt (... wenn auch sehr groß)

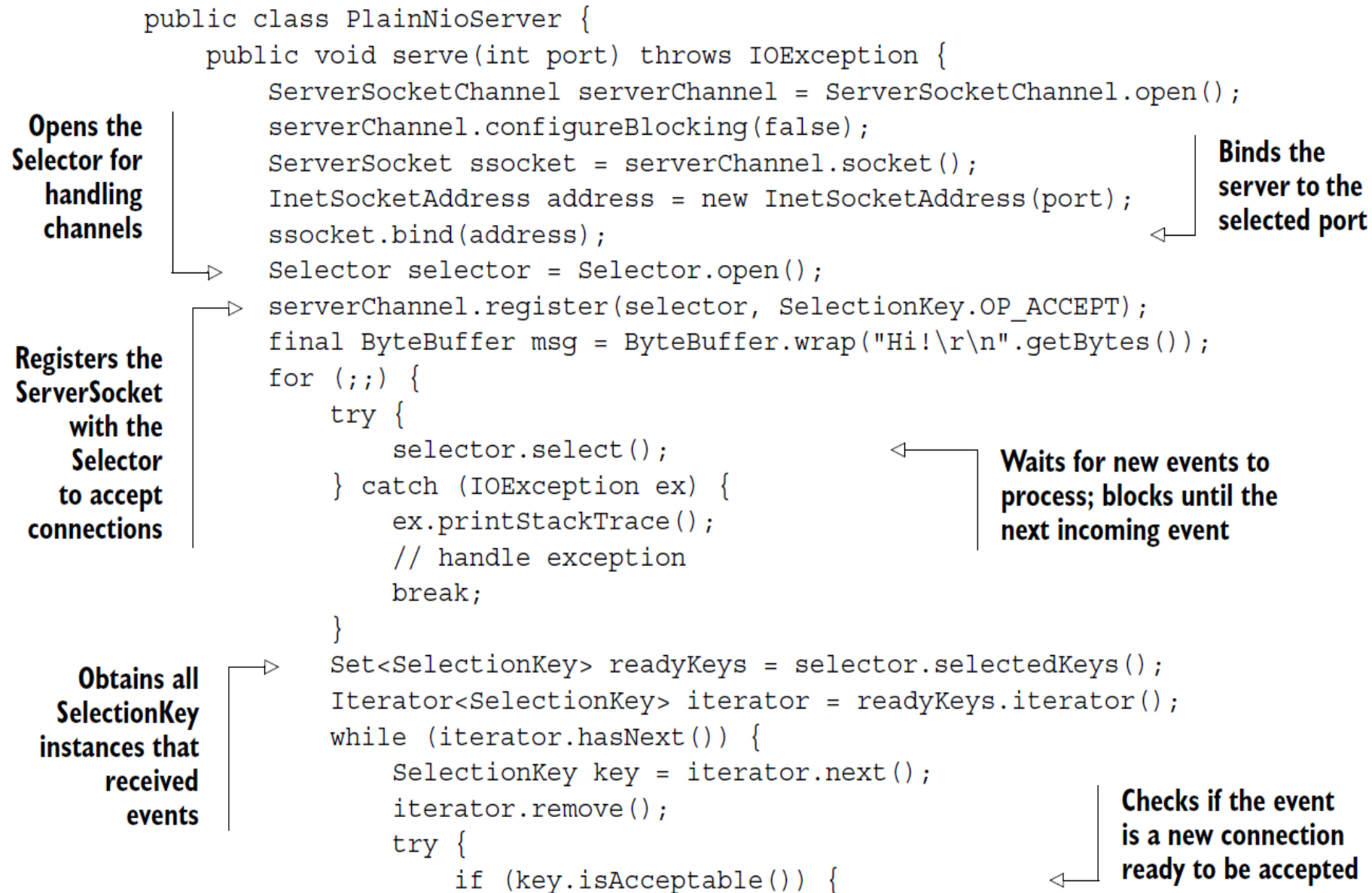
- „Neuer“ Ansatz (2002) JDK 1.4: Java NIO
- Damals **New** I/O ... heute eher **Nonblocking** I/O



- Selector und Event-Mechanismus
- Selector sagt Bescheid, wenn auf einem der Sockets was passiert (Neue Verbindung/lesen/schreiben/schließen...)
- Kommunikation auf ByteBuffer-Ebene :-o (→ insbes. auch „Rekonstruktion der empfangenen Nachrichten“ von Hand)



Was man auf deiner einen Seite hineinsteckt, kommt nicht unbedingt in der selben Art und Weise auf der anderen Seite wieder heraus, d.h Paket bleiben nicht erhalten → Information darüber was zusammengehört, muss mitgesendet werden (Protokoll)!  
Vor allem Problem, wenn Datenmenge größer!



```
        ServerSocketChannel server =
            (ServerSocketChannel)key.channel();
        SocketChannel client = server.accept();
        client.configureBlocking(false);
        client.register(selector, SelectionKey.OP_WRITE |
            SelectionKey.OP_READ, msg.duplicate());
        System.out.println(
            "Accepted connection from " + client);
    }
    if (key.isWritable()) {
        SocketChannel client =
            (SocketChannel)key.channel();
        ByteBuffer buffer =
            (ByteBuffer)key.attachment();
        while (buffer.hasRemaining()) {
            if (client.write(buffer) == 0) {
                break;
            }
        }
        client.close();
    }
} catch (IOException ex) {
    key.cancel();
    try {
        key.channel().close();
    } catch (IOException cex) {
        // ignore on close
    }
}
}
}
}
```

**Accepts client and registers it with the selector**

**Checks if the socket is ready for writing data**

**Writes data to the connected client**

**Closes the connection**



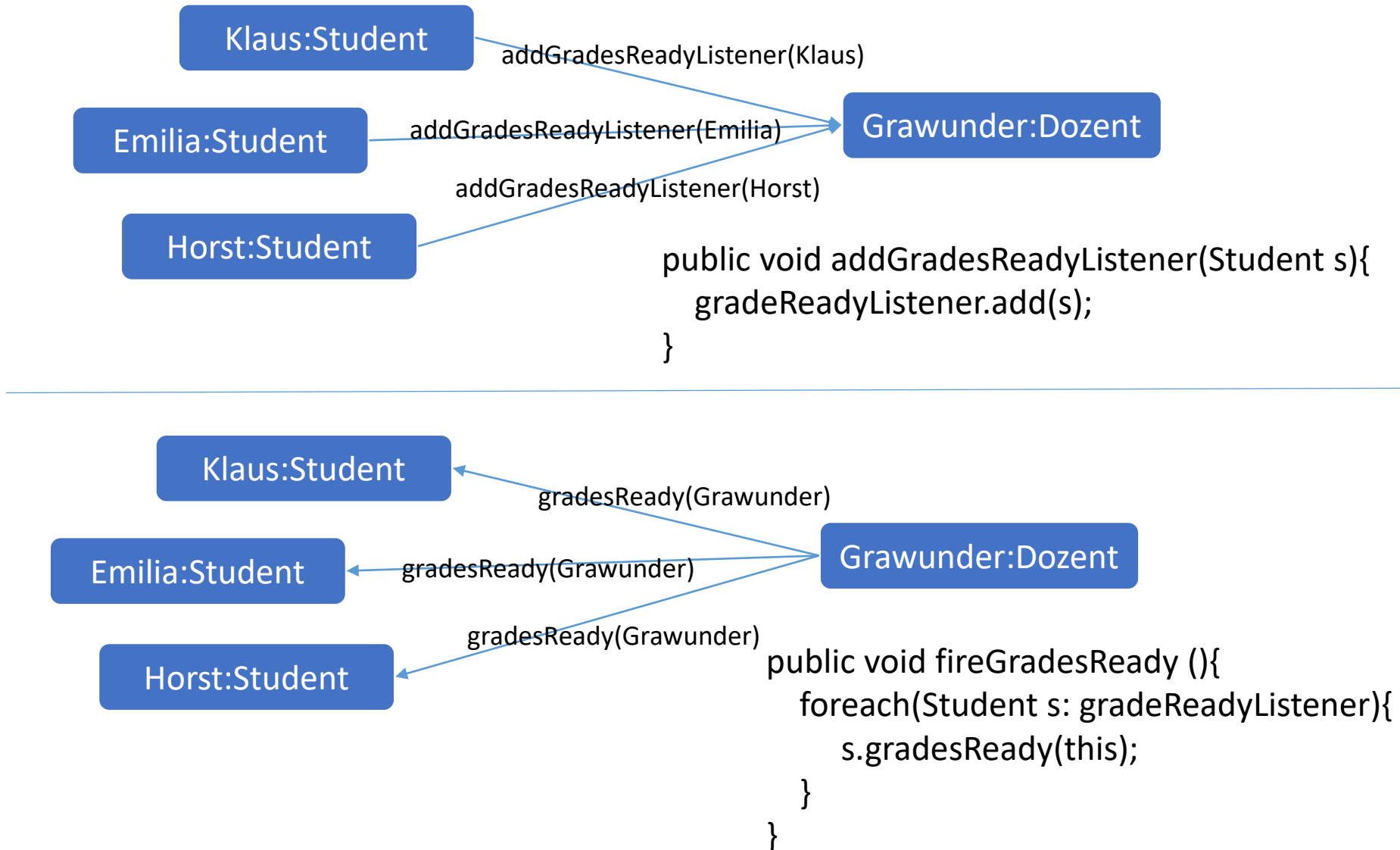


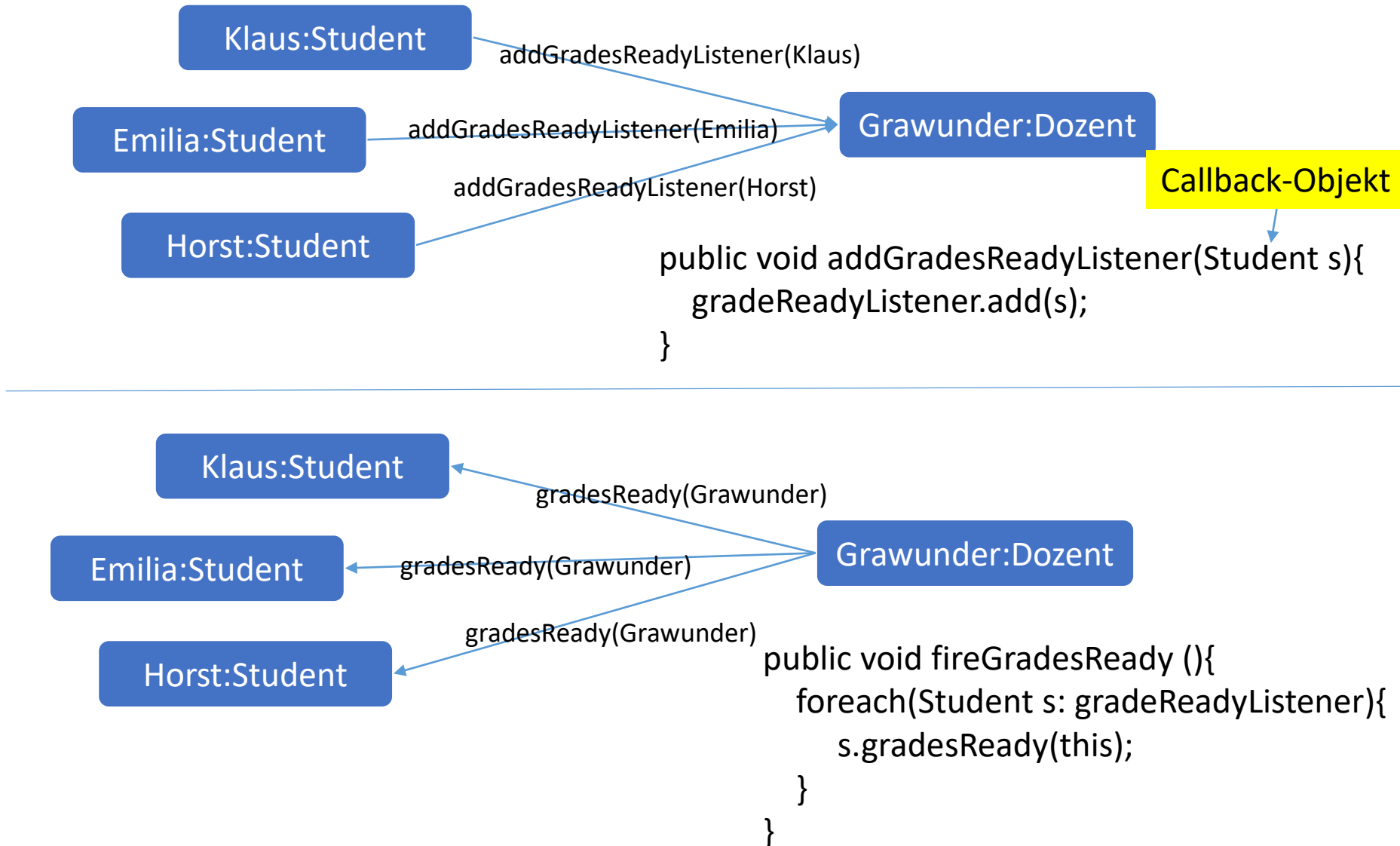
- Greift die Hauptidee von NIO auf, macht das Handling aber **viel einfacher**:
  - Asynchron
  - Ereignisgetrieben
- **Kernkomponenten von Netty**:
  - Channel:
    - Ein **Kanal** für eingehende oder ausgehende Kommunikation
    - Kann I/O-Operationen durchführen
    - Kann geöffnet, geschlossen, verbunden oder nicht verbunden sein
    - Kann auch eine Dateiverbindung sein
  - Callback
  - Futures/ChannelFutures
  - Events und Handler





- Ein Student hat **Klausur** geschrieben und möchte vom Dozenten informiert werden, wenn die **Ergebnisse** vorliegen
- **Ansatz:**
  - Student gibt Dozent seine E-Mail-Adresse mit dem Wunsch benachrichtigt zu werden, wenn das Ergebnis vorliegt
  - Prüfer sammelt alle E-Mail-Adressen
  - Prüfer korrigiert die Arbeit
  - Prüfer schickt an alle hinterlegten E-Mail-Adressen, dass die Ergebnisse vorliegen
  - Studenten holen die Ergebnisse ab
- **Asynchron** → Niemand „blockiert“ hier, kein „Busy Waiting“
- **Ereignisgetrieben** → Wenn es etwas gibt, erfolgt eine Nachricht
- In Java statt E-Mail Callback-Objekt bei dem Methode aufgerufen wird





```
1 import java.util.ArrayList;
2 import java.util.List;
3
4 public class Dozent {
5
6     List<Student> gradeReadyListener = new ArrayList<>();
7
8     public void addGradesReadyListener(Student s) {
9         this.gradeReadyListener.add(s);
10    }
11
12    public void work() {
13        //
14        fireGradesReady();
15    }
16
17    private void fireGradesReady() {
18        for(Student s:gradeReadyListener) {
19            s.gradeReady(this);
20        }
21    }
22 }
```

Student möchte informiert werden

Informieren aller Studenten

- Klasse Dozent bekommt public-Methode:
  - addGradesReadyListener(Student s)
- Klasse Student bekommt public-Methode:
  - gradesReady(Dozent d)
- Zwei **Callback-Objekte** Student (will informiert werden) und Dozent (welche Noten sind eigentlich fertig) werden jeweils übergeben
- Warum ist dies so noch keine „schöne“ Lösung?
- Was, wenn das P-Amt auch wissen möchte, wann die Noten vorliegen?
  - addGradesReadyListener(P-Amt p) ???

```
3
4 public class Dozent {
5
6     List<Student> studentGradeReadyListener = new ArrayList<>();
7     List<PAmt> pamtGradeReadyListener = new ArrayList<>();
8
9     public void addGradesReadyListener(Student s) {
10         this.studentGradeReadyListener.add(s);
11     }
12
13     public void addGradesReadyListener(PAmt s) {
14         this.pamtGradeReadyListener.add(s);
15     }
16
17
18     public void work() {
19         //
20         fireGradesReady();
21     }
22
23     private void fireGradesReady() {
24         for(Student s:studentGradeReadyListener) {
25             s.gradeReady(this);
26         }
27         for(PAmt p:pamtGradeReadyListener) {
28             p.gradeReady(this);
29         }
30     }
31 }
32
```



- und jetzt wollen auch noch die Eltern informiert werden ...



- ist (in diesem Beispiel) dem Dozent doch egal, wer sich für die Fertigstellung der Noten interessiert
- Ansatz: Interface einführen

```
1
2 public interface IGradeReadyListener {
3     void gradeReady(Dozent dozent);
4 }
5
6 public class Student implements IGradeReadyListener{
7
8     @Override
9     public void gradeReady(Dozent dozent) {
10
11     }
12 }
13
14 public class PAmt implements IGradeReadyListener {
15
16     @Override
17     public void gradeReady(Dozent dozent) {
18
19     }
20 }
```

```
public class Dozent {  
  
    List<IGradeReadyListener> gradeReadyListener = new ArrayList<>();  
  
    public void addGradesReadyListener(IGradeReadyListener s) {  
        this.gradeReadyListener.add(s);  
    }  
  
    public void work() {  
        //  
        fireGradesReady();  
    }  
  
    private void fireGradesReady() {  
        for (IGradeReadyListener l : gradeReadyListener) {  
            l.gradeReady(this);  
        }  
    }  
}
```

Kann Student, P-Amt ...  
oder wer auch immer sein

- Das Konzept nennt sich **Observer Pattern**
- Immer dann, wenn nicht aktiv gewartet werden soll/kann
- Oft im Kontext von GUIs, aber hier auch in der Netzwerkkommunikation



- Greift die Hauptidee von NIO auf, macht das Handling aber einfacher:
  - Asynchron
  - Ereignisgetrieben
- Kernkomponenten von Netty:
  - Channel:
    - Ein Kanal für eingehende oder ausgehende Kommunikation
    - Kann I/O-Operationen durchführen
    - Kann geöffnet, geschlossen, verbunden oder nicht verbunden sein
    - Kann auch eine Dateiverbindung sein
  - **Callback**
  - Futures/ChannelFutures
  - Events und Handler



- Ein **Callback** ist ein Mechanismus über einen bestimmten Weg eine Antwort zu bekommen
- I.d.R. wird in Java dafür ein Objekt übergeben, häufig wird dies auch mit **inneren Klassen** gemacht
- In Netty gibt es z.B. Interface ChannelHandler

```
public class ConnectHandler extends ChannelInboundHandlerAdapter {  
    @Override  
    public void channelActive(ChannelHandlerContext ctx)  
        throws Exception {  
        System.out.println(  
            "Client " + ctx.channel().remoteAddress() + " connected");  
        }  
}
```

**channelActive(ChannelHandlerContext) is  
called when a new connection is established.**

- **Anmerkung:**
  - \*Adapter ist typischer Java-Ansatz, wenn man nicht alle Methoden eines Interfaces überschreiben möchte
  - ChannelInboundHandlerAdapter hat Standard-Implementierungen für die einzelnen Methoden

- Seit Java5
- Ein **Future** ist eine Möglichkeit, eine Anwendung darüber zu informieren, dass eine Operation fertig ist
- **Platzhalter** für das Ergebnis einer **asynchronen** Operation
- Netty: ChannelFuture

```
Channel channel = ...;  
// Does not block  
ChannelFuture future = channel.connect(  
    new InetSocketAddress("192.168.0.1", 25));
```

← **Asynchronous  
connection to a  
remote peer**


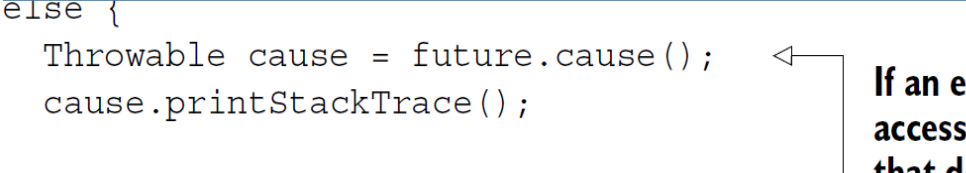
```
Channel channel = ...;  
// Does not block  
ChannelFuture future = channel.connect(  
    new InetSocketAddress("192.168.0.1", 25));  
future.addListener(new ChannelFutureListener() {
```

**Connects asynchronously  
to a remote peer.**

**Registers a ChannelFuture-  
Listener to be notified once**

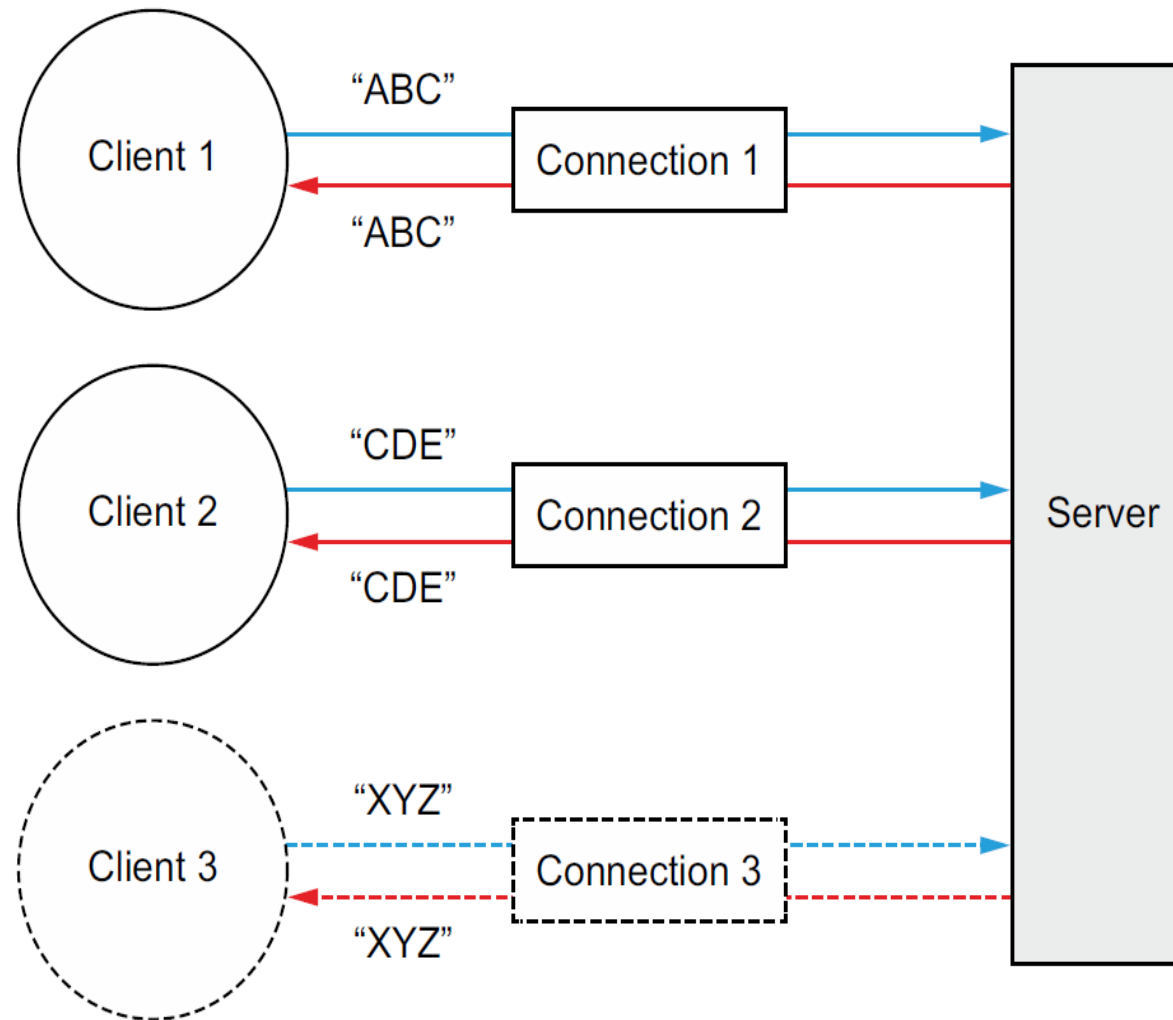
**Checks  
the status  
of the  
operation.**

1

```
    @Override  
    public void operationComplete(ChannelFuture future) {  
        if (future.isSuccess()) {  
              
        } else {  
              
            Throwable cause = future.cause();  
            cause.printStackTrace();  
        }  
    }  
});
```

**If an error occurred,  
accesses the Throwable  
that describes the cause.**

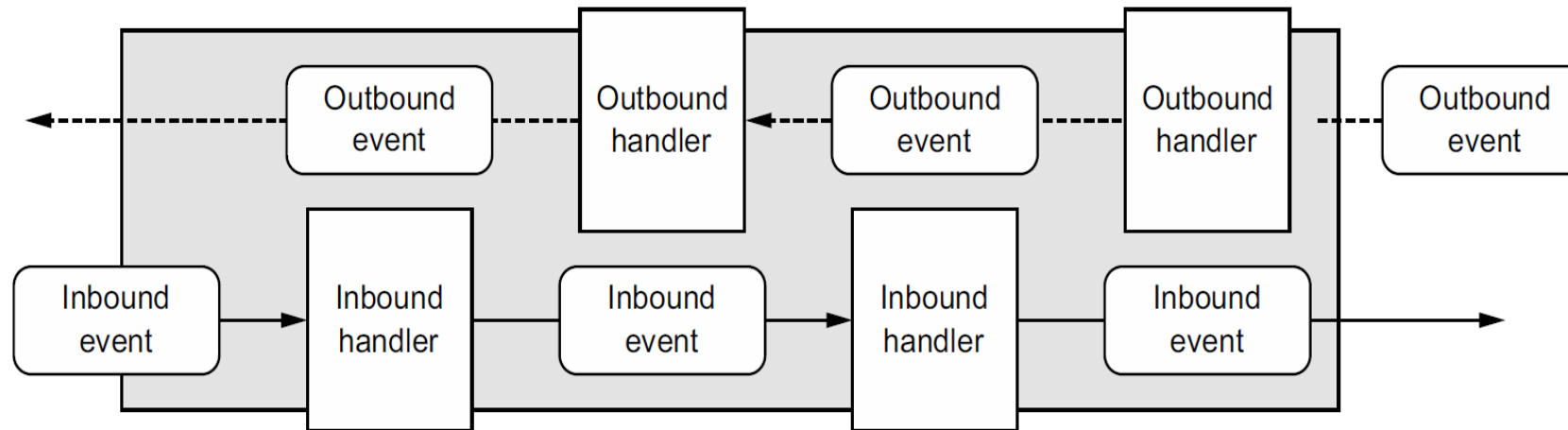
Hier ein Callback mit einer inneren Klasse



**Figure 2.1 Echo client and server**



- Man braucht pro Kommunikationsseite
  - (Mindestens) einen (Kanal)Handler → ChannelHandler
  - Einen Mechanismus, diese(n) Handler in Netty „einzutragen“ → Bootstrapping
- Netty nutzt Kette von ChannelHandlern
  - jeder übernimmt andere Aufgabe → Separation of Concerns
  - Event: z.B. ankommendes TCP-Paket



**Figure 1.3** Inbound and outbound events flowing through a chain of ChannelHandlers

```
@Sharable
public class EchoServerHandler extends ChannelInboundHandlerAdapter {

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        ByteBuf in = (ByteBuf) msg;
        System.out.println(
            "Server received: " + in.toString(CharsetUtil.UTF_8));
        ctx.write(in);
    }

    @Override
    public void channelReadComplete(ChannelHandlerContext ctx) {
        ctx.writeAndFlush(Unpooled.EMPTY_BUFFER)
            .addListener(ChannelFutureListener.CLOSE);
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx,
        Throwable cause) {
        cause.printStackTrace();
        ctx.close();
    }
}
```

**Indicates that a ChannelHandler can be safely shared by multiple channels**

**Logs the message to the console**

**Writes the received message to the sender without flushing the outbound messages**

**Flushes pending messages to the remote peer and closes the channel**

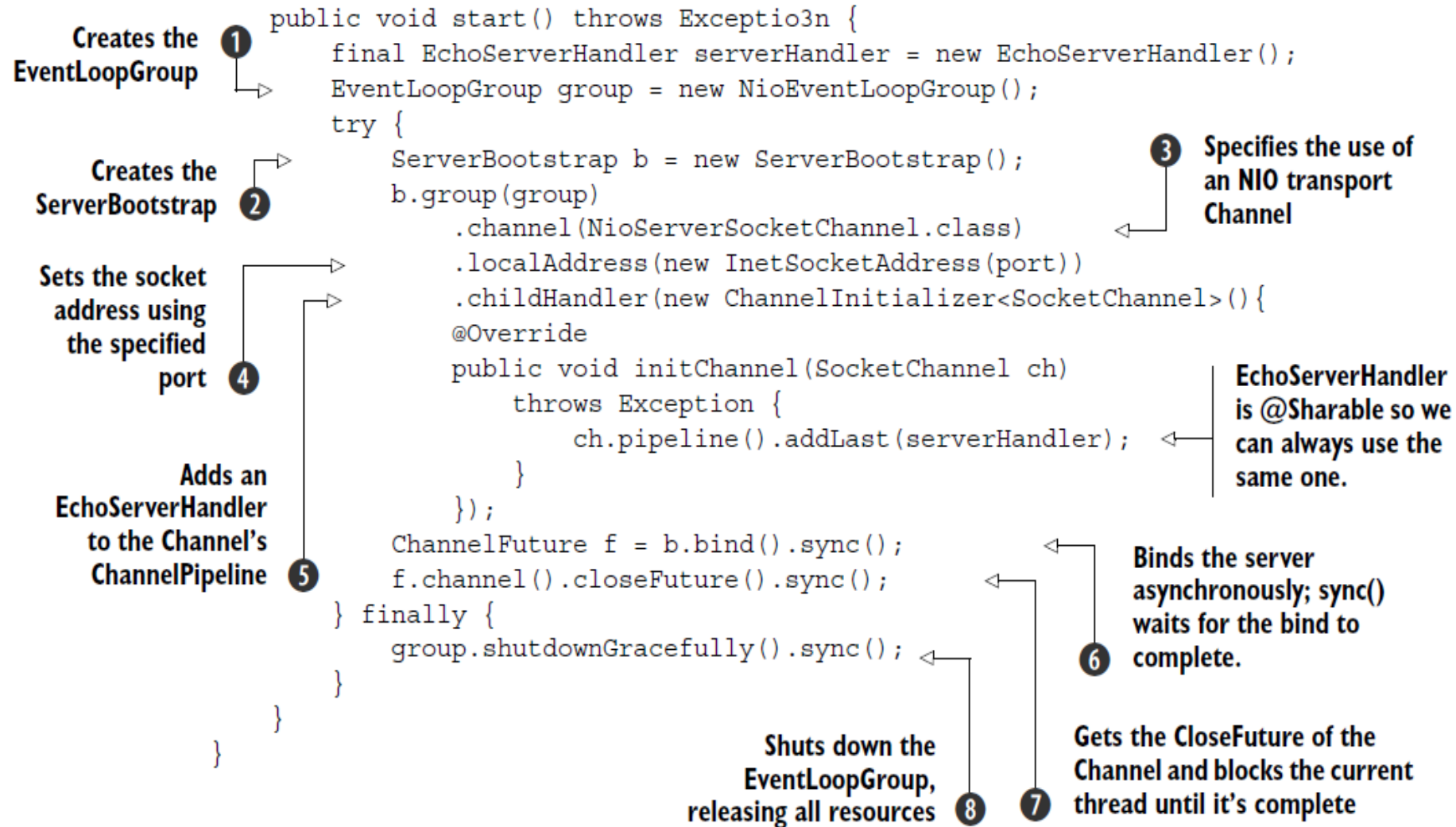
**Closes the channel**

**Prints the exception stack trace**

```
public class EchoServer {  
    private final int port;  
  
    public EchoServer(int port) {  
        this.port = port;  
    }  
  
    public static void main(String[] args) throws Exception {  
        if (args.length != 1) {  
            System.err.println(  
                "Usage: " + EchoServer.class.getSimpleName() +  
                " <port>");  
        }  
        int port = Integer.parseInt(args[0]);  
        new EchoServer(port).start();  
    }  
}
```

Sets the port  
value (throws a  
**NumberFormatException**  
if the  
port argument is  
malformed)

Calls the server's  
**start()** method



```
@Sharable
public class EchoClientHandler extends
    SimpleChannelInboundHandler<ByteBuf> {
    @Override
    public void channelActive(ChannelHandlerContext ctx) {
        ctx.writeAndFlush(Unpooled.copiedBuffer("Netty rocks!",
            CharsetUtil.UTF_8));
    }

    @Override
    public void channelRead0(ChannelHandlerContext ctx, ByteBuf in) {
        System.out.println(
            "Client received: " + in.toString(CharsetUtil.UTF_8));
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx,
        Throwable cause) {
        cause.printStackTrace();
        ctx.close();
    }
}
```

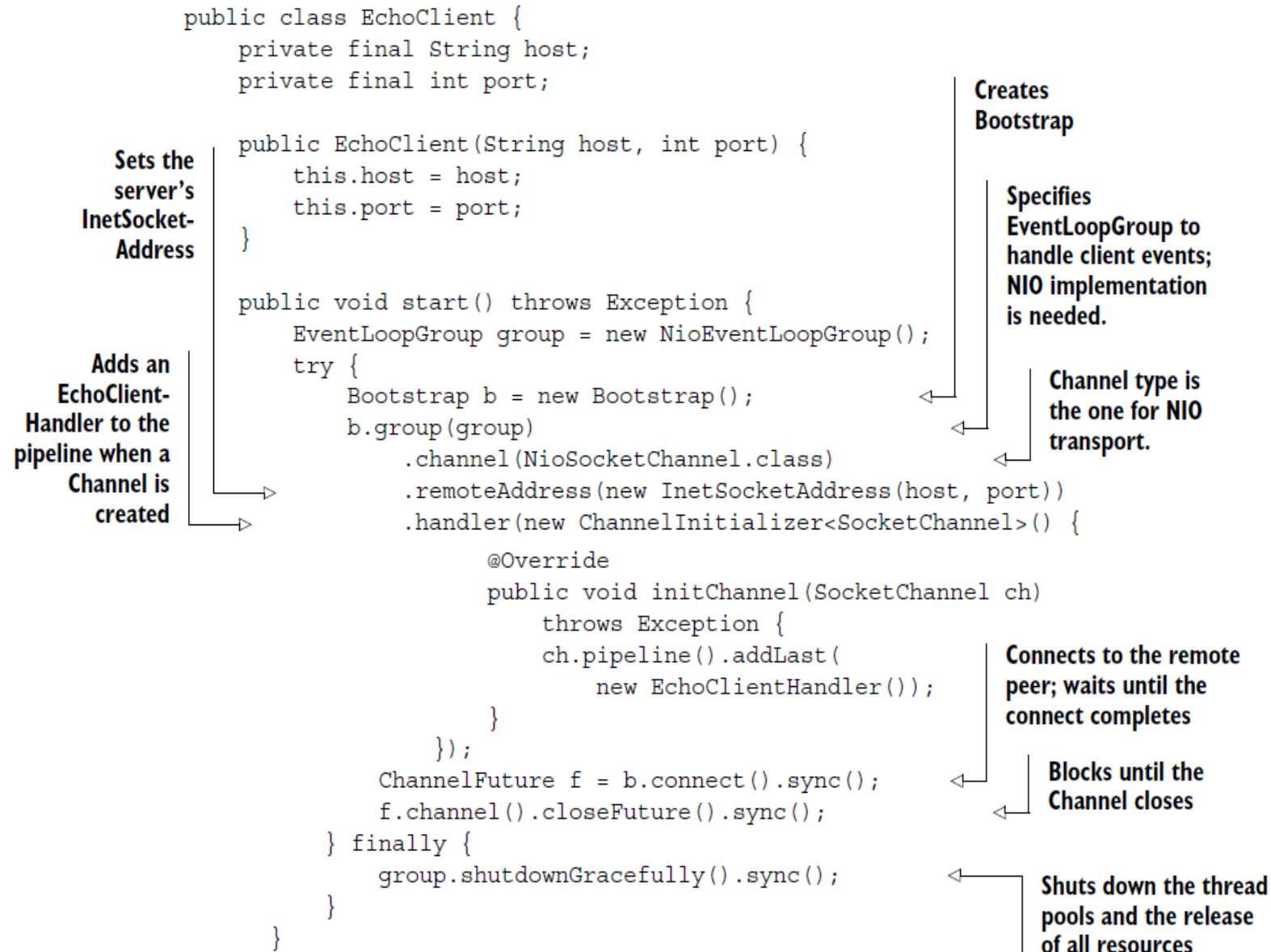
← Marks this class as one whose instances can be shared among channels

← When notified that the channel is active, sends a message

← Logs a dump of the received message

← On exception, logs the error and closes channel

```
public static void main(String[] args) throws Exception {  
    if (args.length != 2) {  
        System.err.println(  
            "Usage: " + EchoClient.class.getSimpleName() +  
            " <host> <port>");  
        return;  
    }  
  
    String host = args[0];  
    int port = Integer.parseInt(args[1]);  
    new EchoClient(host, port).start();  
}
```

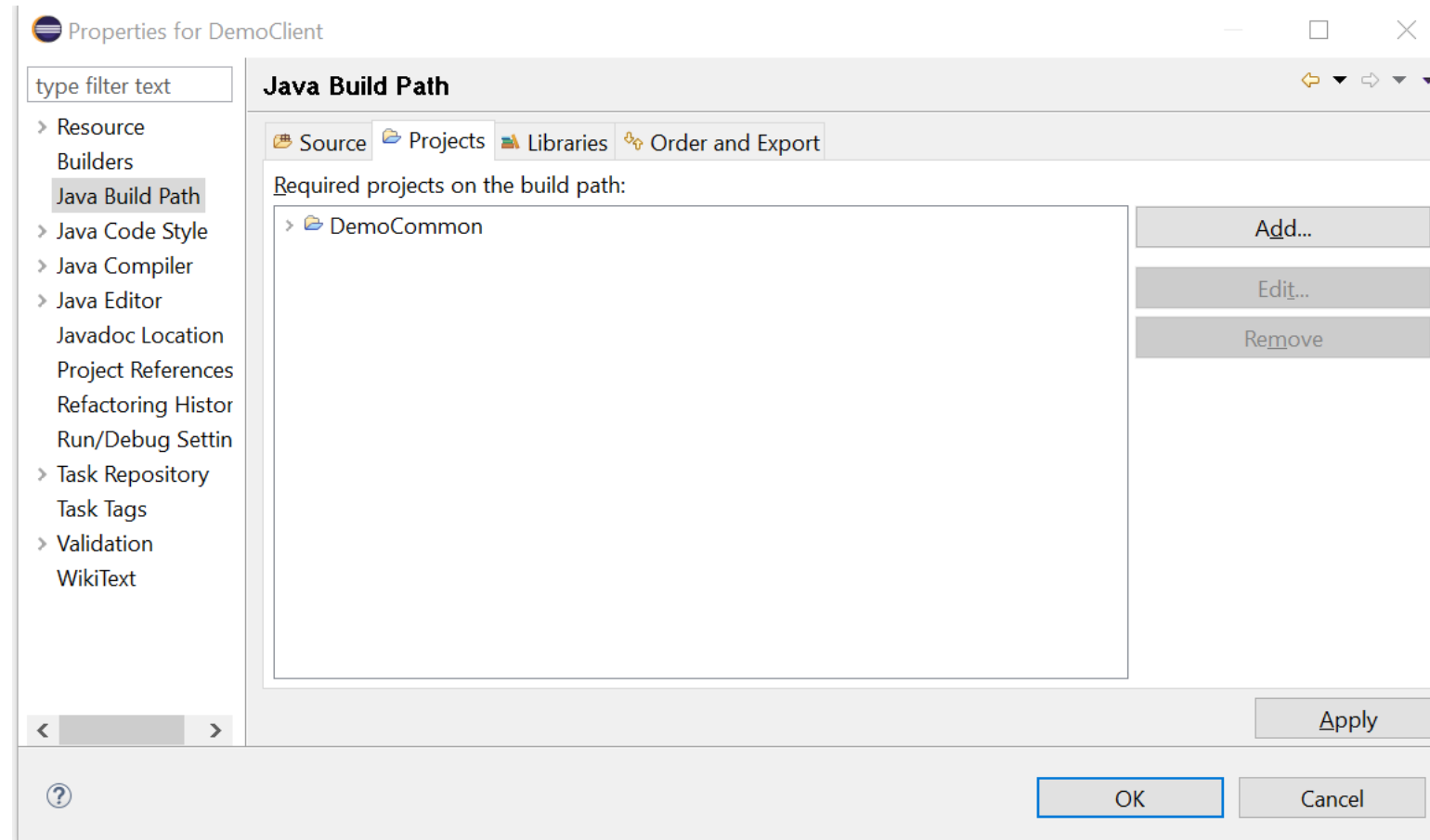




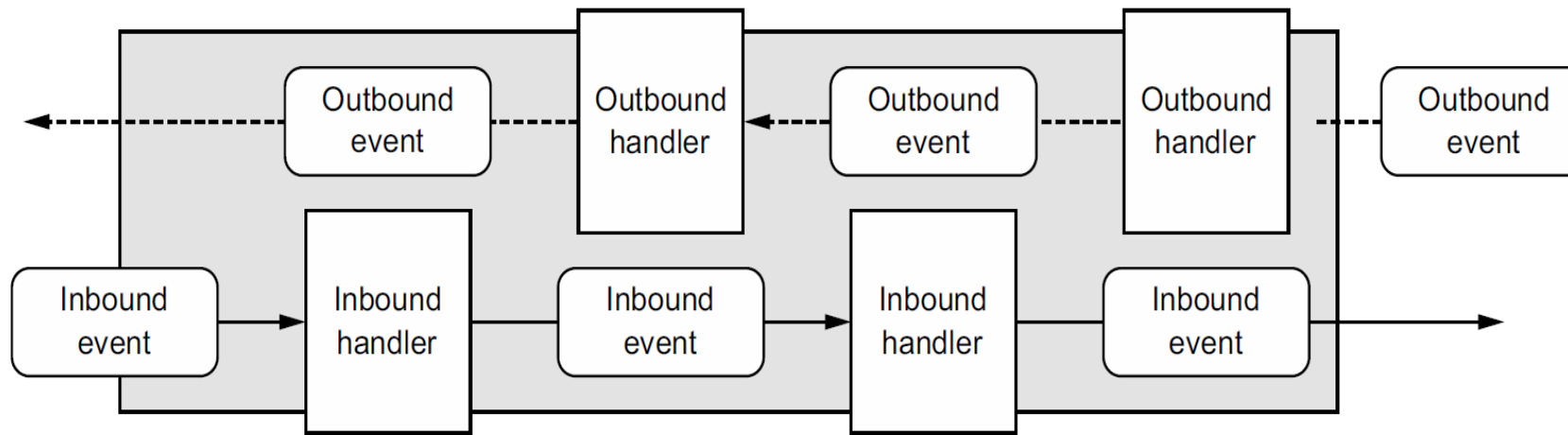


- **RMI (Remote Method Invocation):**
  - Kapselt sämtliche Kommunikationsmechanismen
  - Er werden sog. Stubs und Skeletons erstellt
  - Oft Probleme bei Firewalls
  - Simon soll Alternative sein, aber nur „Hobby“-Projekt
- **WebService**
  - Analog zu RMI, aber Kommunikation über HTTP
  - RMI auf jedem Fall vorzuziehen, da hier weniger Probleme mit Port
  - Web-Server Bibliothek gibt es „out of the box“ bei Apache (<https://hc.apache.org/>)
- **WebSockets**
  - Kommunikation über einen Socket-Kanal
  - Macht aber eigentlich nur Sinn, wenn man im Client eingeschränkt ist (z.B. in einem Browser)

- Nachrichtenaustausch mit **JSON-Objekten**
  - Vorteil:
    - Client und Server haben keinen direkten Abhängigkeiten
    - U.U. stabiler gegenüber Änderungen
  - Nachteil:
    - Manueller Serialisierungsaufwand (ist aber i.d.R. zu vernachlässigen)
    - Wartung u.U. komplexer
- **Objekt-Serialisierung**
  - ObjectInputStream/ObjectOutputStream oder auch mit Netty Handlern
  - Sowohl Client als auch Server benötigen die Klassen (in der selben Version!)
  - Ansatz:
    - Drei Projekte: Client, Common, Server
    - Client und Server nutzen jeweils auch das Common-Projekt (build path)



- Idee: Versende und empfangen Nachrichten-Objekte
- Z.B. : LoginMessage, Chat-Message, TurnMessage
- Die Objekte sind **serialisierbar**
- Der Empfänger interpretiert die Nachrichten und schickt sie entsprechend weiter (Google Guava bietet einen Nachrichtenbus (event bus), kann bei der Entkopplung der Komponenten helfen!)
- JSON-Ansatz analog, allerdings statt Objekt Feld im JSON-Objekt (z.B. MessageType = „Login“)



**Figure 1.3** Inbound and outbound events flowing through a chain of ChannelHandlers

- Nachrichten werden immer an den nächsten Handler in der Kette weiter geleitet
- Jeder Handler kann mit der Nachricht etwas machen
- Beispiel hier: Objekte **serialisieren** und **deserialisieren**

- **LoginMessage** (Nutzername und Passwort)
- Antwort: z.B. Nachricht mit SessionID wenn Nutzername und Passwort zusammenpassen
- Nachricht an alle anderen Clients: **UserLoggedInMessage** (mit dem Nutzernamen)
- Ganz wichtig: **Client und Server wissen nicht, wie die Kommunikation abläuft**
  - Definition eines Interfaces (z.B. IUserService) welches von der Kommunikationskomponente implementiert wird
- Achtung: Asynchrone Kommunikation über **Eventbus**
- Alternative: wait() und notify() Mechanismen verwenden, aber vor dem „Nutzer“ Clientanwendung verbergen

- Anpassung der Pipeline von Netty

## Server-Seite

```
public void start() throws Exception {
    final DemoServerHandler serverHandler = new DemoServerHandler(this);
    EventLoopGroup bossGroup = new NioEventLoopGroup(1);
    EventLoopGroup workerGroup = new NioEventLoopGroup();
    try {
        ServerBootstrap b = new ServerBootstrap();
        b.group(bossGroup, workerGroup).channel(NioServerSocketChannel.class)
        .localAddress(new InetSocketAddress(port)).childHandler(new ChannelInitializer<SocketChannel>() {

            @Override
            protected void initChannel(SocketChannel ch) throws Exception {
                ch.pipeline().addLast(new ObjectEncoder());
                ch.pipeline().addLast(new ObjectDecoder(ClassResolvers.cacheDisabled(null)));
                ch.pipeline().addLast(serverHandler);
            }

        });
        ChannelFuture f = b.bind().sync();
        f.channel().closeFuture().sync();
    } finally {
        bossGroup.shutdownGracefully().sync();
        workerGroup.shutdownGracefully().sync();
    }
}
```

Es gibt einen Handler für die Kommunikation

Zwei Gruppen für die Verarbeitung

Neu: ObjectEncoder und ObjectDecoder zum Serialisieren bzw. Deserialisieren

```
public void start() throws Exception {
    EventLoopGroup group = new NioEventLoopGroup();
    try {
        Bootstrap b = new Bootstrap();
        b.group(group).channel(NioSocketChannel.class).remoteAddress(new InetSocketAddress(host, port))
            .handler(new ChannelInitializer<SocketChannel>() {

                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    ch.pipeline().addLast(new ObjectEncoder());
                    ch.pipeline().addLast(new ObjectDecoder(ClassResolvers.cacheDisabled(null)));
                    ch.pipeline().addLast(new DemoClientHandler(DemoClient.this));
                }
            });
        ChannelFuture f = b.connect().sync();
        f.channel().closeFuture().sync();
    } finally {
        group.shutdownGracefully().sync();
    }
}
```

Neu: ObjectEncoder und  
ObjectDecoder zum Serialisieren  
bzw. Deserialisieren



```
    */
    @Sharable
    public class DemoServerHandler extends ChannelInboundHandlerAdapter {

        private ServerHandlerDelegate delegate;

        /**
         * Creates a new DemoServerHandler
         * @param delegate The ServerHandlerDelegate that should receive information about the connection
         */
        public DemoServerHandler(ServerHandlerDelegate delegate) {
            this.delegate = delegate;
        }

        @Override
        public void channelActive(ChannelHandlerContext ctx) throws Exception {
            delegate.newClientConnected(ctx);
        }

        @Override
        public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
            if (msg instanceof IMessage) {
                delegate.process(ctx, (IMessage) msg);
            } else {
                System.err.println("Illegal Object read from channel. Ignored!");
            }
        }

        @Override
        public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {
            if (ctx.channel().isActive()) {
                System.err.println("Exception caught " + cause);
            } else {
                delegate.clientDisconnected(ctx);
            }
        }
    }
}
```

Leitet wichtige Sachen  
einfach  
an den DemoServer  
„zurück“  
Da @Sharable hätte man  
auch  
DemoServer extends  
ChannelInboundHandler  
Adapter  
machen können.

```
public interface IUserService {  
  
    Session login(String username, String password) throws LoginException;  
    List<String> retrieveAllUsers();  
  
}
```

```
public class DemoApplication extends Thread implements IUserServiceListener, IConnectionListener {

    static Channel channel = null;
    IClientUserService userService;
    Session userSession = Session.invalid;

    @Override
    public void run() {
        // Wait for connection
        try {
            Thread.sleep(500);
        } catch (InterruptedException e2) {
        }

        System.out.println("Demo application started");
        userService = UserServiceFactory.getUserService();
        userService.addUserServiceListener(this);
        System.out.println("Calling login with wrong values");

        String userName = "egal";
        String password = "falsch";

        // In this simple example the service does not wait for a result and sends
        // return values, so just call the method. A more complex example should wait
        // for a server return value and deliver the session
        userService.login(userName, password);

        delay(1000);
        userService.retrieveAllUsers(userSession);

        delay(2000);
        // hashCode (so every client has its own username)
        userName = "test" + hashCode();
        password = "test" + hashCode();

        userService.login(userName, password);

        process();
        System.out.println("Demo application terminated");
    }
}
```

Demo-Anwendung darf nichts von  
Kommunikationsart wissen

Asynchron → Callback!

Login-Nachrichten zum Server  
Schicken (einmal falsche  
einmal korrekte Daten)

```
public class ObjectCommunication implements IClientUserService {  
  
    private List<IUserServiceListener> listener = new ArrayList<>();  
  
    Channel client;  
  
    public ObjectCommunication(Channel client) {  
        this.client = client;  
    }  
  
    @Override  
    public Session login(String username, String password) throws LoginExc  
        LoginMessage msg = new LoginMessage(username, password);  
        sendMessage(msg);  
        return null; // asynch call  
    }  
  
    @Override  
    public List<String> retrieveAllUsers() {  
        GenericCommand cmd = new GenericCommand(GenericCommands.RETRIEVE_USERS_LIST);  
        sendMessage(cmd);  
        return null; // asynch call  
    }  
  
    @Override  
    public void addUserServiceListener(IUserServiceListener userServiceListener) {  
        listener.add(userServiceListener);  
    }  
  
    private void sendMessage(Serializable msg) {  
        client.writeAndFlush(msg);  
    }  
}
```

Baue neue Login-Nachricht

Und versenden

Noch keine Antwort, deswegen null

Wegen Netty ObjectEncoder und  
ObjectDecoder: Einfach das Objekt schreiben

- Objekt welches vom Client zum Server gesendet wird (Serializable!!)
- Muss sowohl vom Client als auch vom Server verstanden werden

```
5 public class LoginMessage extends AbstractMessage {
6
7     private static final long serialVersionUID = 7793454958390539421L;
8     String username;
9     String password;
10
11     public LoginMessage() {
12         super(Session.invalid);
13     }
14
15     public LoginMessage(String username, String password) {
16         super(Session.invalid);
17         this.username = username;
18         this.password = password;
19     }
20
21     public void setUsername(String username) {}
22
23     public String getUsername() {}
24
25     public void setPassword(String password) {}
26
27     public String getPassword() {}
28
29 }
```

```
    */
    @Sharable
    public class DemoServerHandler extends ChannelInboundHandlerAdapter {

        private ServerHandlerDelegate delegate;

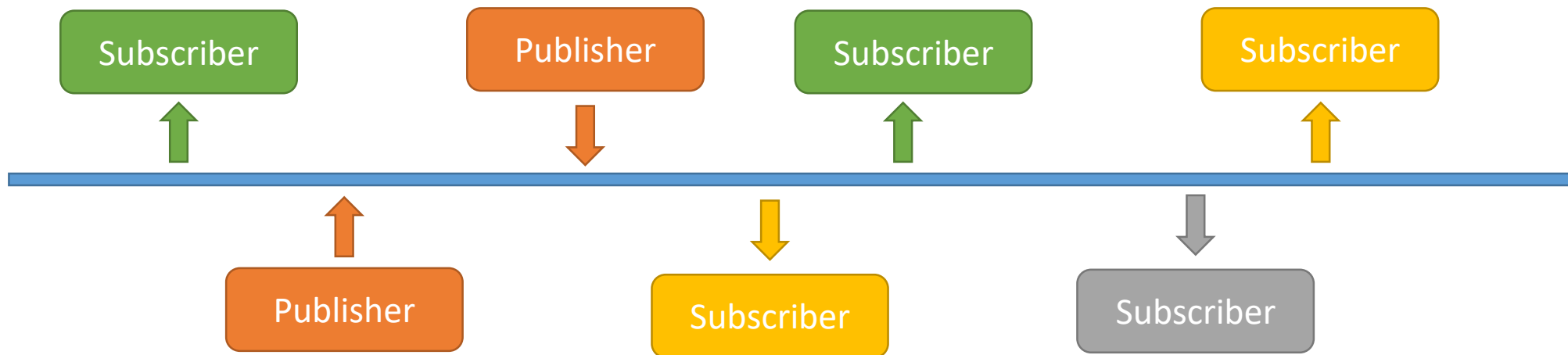
        /**
         * Creates a new DemoServerHandler
         * @param delegate The ServerHandlerDelegate that should receive information about the connection
         */
        public DemoServerHandler(ServerHandlerDelegate delegate) {
            this.delegate = delegate;
        }

        @Override
        public void channelActive(ChannelHandlerContext ctx) throws Exception {
            delegate.newClientConnected(ctx);
        }

        @Override
        public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
            if (msg instanceof IMessage) {
                delegate.process(ctx, (IMessage) msg);
            } else {
                System.err.println("Illegal Object read from channel. Ignored!");
            }
        }

        @Override
        public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {
            if (ctx.channel().isActive()) {
                System.err.println("Exception caught " + cause);
            } else {
                delegate.clientDisconnected(ctx);
            }
        }
    }
}
```

- **Problem:**
  - Wenn Server Nachricht empfängt, muss er wissen, für wen die Nachricht bestimmt ist
- **Alternative:**
  - Verwende einen EventBus
  - Schicke Nachrichten auf den EventBus
  - Mögliche Empfänger registrieren sich auf dem Bus (ähnlich wie beim Observer)
  - Nachricht wird an diese Empfänger zugestellt



```
// Called from DemoServerHandler
@Override
public void process(ChannelHandlerContext ctx, IMessage msg) {

    try {
        // Bind msg to ctx
        msg.setInfo(ctx);
        eventBus.post(msg);

    } catch (Exception e) {
        System.err.println("ServerException " + e.getClass().getName() + " " + e.getMessage())
        sendToClient(ctx, new ExceptionMessage(e));
    }
}
```

„Hack“

Es können beliebige Objekte auf den Bus „gelegt“ werden



- Bei dem selben EventBus auf dem gesendet wird registrieren
  - Es kann mehrere Busse geben
- Mit Annotation @Subscribe die Methode(n) kennzeichnen, die Events empfangen sollen
- Name der Methode ist egal
- Signatur der Methode: Nur ein Objekt
- Es können sich viele Objekte für Events registrieren (z.B.: UserManagement, ChatManagement, GameManagement, etc.)
- Wenn die Signatur der Methode passt (== das gesendete Event ist vom selben Typ wie das Element in der Signatur) wird die Methode aufgerufen

```

/**
 * For demo reasons the EventBus as part of this class
 */
final private EventBus eventBus = new EventBus();

```

```

65
66- /**
67  * Creates a new DemoServer Object and start listening on given port
68  *
69  * @param port
70  *         The port the server should listen for new connection
71  * @param userService
72  *         The userService that should be used for the server
73  */
74- public DemoServer(int port, IUserService userService) {
75     this.port = port;
76     this.userService = userService;
77     // TODO: Ping clients
78     eventBus.register(this);
79 }

```

Alle Bus-Nachrichten vom Typ LoginCommand führen hier zum Aufruf

@Subscribe

```
private void processLoginCommand(LoginCommand msg) {  
    if (msg.getInfo() instanceof ChannelHandlerContext) {  
        ChannelHandlerContext ctx = (ChannelHandlerContext) msg.getInfo();  
  
        System.out.println("Got new login message with " + msg.getUsername() + " " + msg.getPassword());  
  
        Session newSession = userService.login(msg.getUsername(), msg.getPassword());  
  
        if (newSession.isValid()) {  
            sendToClient(ctx, new LoginSuccessfulMessage(newSession, msg.getUsername()));  
            putSession(ctx, newSession);  
  
            // Send all clients information, that a new user is logged in  
            sendToAll(new UserLoggedInMessage(msg.getUsername()));  
        } else {  
            sendToClient(ctx, new ExceptionMessage(new LoginException()));  
        }  
    }  
}
```

Verbindung und Session verknüpfen

Nachricht an alle Clients

Alle Bus-Nachrichten vom Typ LogoutCommand führen hier zum Aufruf

@Subscribe

```
private void processLogoutCommand(LogoutCommand msg) {  
    if (msg.getInfo() instanceof ChannelHandlerContext) {  
        ChannelHandlerContext ctx = (ChannelHandlerContext) msg.getInfo();  
        System.out.println("Got new logout " + msg.getSession());  
        checkLogin(ctx, msg);  
        removeUser(ctx, msg.getSession());  
    }  
}
```

„Hack“

Beim echten User-Service auf Server-Seite login aufrufen

Antwort an Aufrufenden schicken

Default Empfänger, wenn niemand eine passende Methode hat

```
@Subscribe  
private void handleEventBusError(DeadEvent deadEvent){  
    System.err.println("DeadEvent detected "+deadEvent);  
}
```

Irgendwie sinnvoll behandeln ...

```
// -----  
// Handling of connected clients  
// -----  
@Override  
public void newClientConnected(ChannelHandlerContext ctx) {  
    System.err.println("New client " + ctx + " connected");  
    connectedClients.add(ctx);  
}  
  
@Override  
public void clientDisconnected(ChannelHandlerContext ctx) {  
    System.err.println("Client disconnected");  
    connectedClients.remove(ctx);  
}  
  
// -----  
// Session Management  
// -----  
private void putSession(ChannelHandlerContext ctx, Session newSession) {  
    // TODO: check if session is already bound to connection  
    activeSessions.put(ctx, newSession);  
}  
  
private Object getSession(ChannelHandlerContext ctx) {  
    return activeSessions.get(ctx);  
}  
  
// -----  
// Help methods: Send only objects of type IMessage  
// -----  
  
private void sendToClient(ChannelHandlerContext ctx, IMessage message) {  
    ctx.writeAndFlush(message);  
}  
  
private void sendToAll(IMessage msg) {  
    for (ChannelHandlerContext client : connectedClients) {  
        try {  
            client.writeAndFlush(msg);  
        } catch (Exception e) {  
            // TODO: Handle exception for unreachable clients  
            e.printStackTrace();  
        }  
    }  
}
```

Callback Methoden  
vom Handler aufgerufen

Session handling

Nachrichten versenden  
Da Netty mit Objekt-(De)Serialisierung  
sehr einfach

- Kommunikation über Netzwerkgrenzen ist nicht einfach
- Ansätze: Ports und Sockets
- Alter Java Ansatz blockierend
- Neuer Java Ansatz sehr komplex
- Netty kaspelt viele der Routine-Arbeiten mit Callbacks und Events
- Guava EventBus hilft Abhängigkeiten zu verringern
- Andere Ansätze mit RMI, Web-Services oder Web-Sockets auch möglich (RMI eher nicht)
- Zwei Gruppen probieren dieses Jahr NetCom2
  
- Quellcode zum Beispiel:
- <https://git.swl.informatik.uni-oldenburg.de/projects/SPD/repos/netzwerkdemo/browse>

# Fragen?

