ISA
Institut für
SoftwareArchitektur

TECHNISCHE HOCHSCHULE MITTELHESSEN

Nebenläufige und verteilte Programme CS2301

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

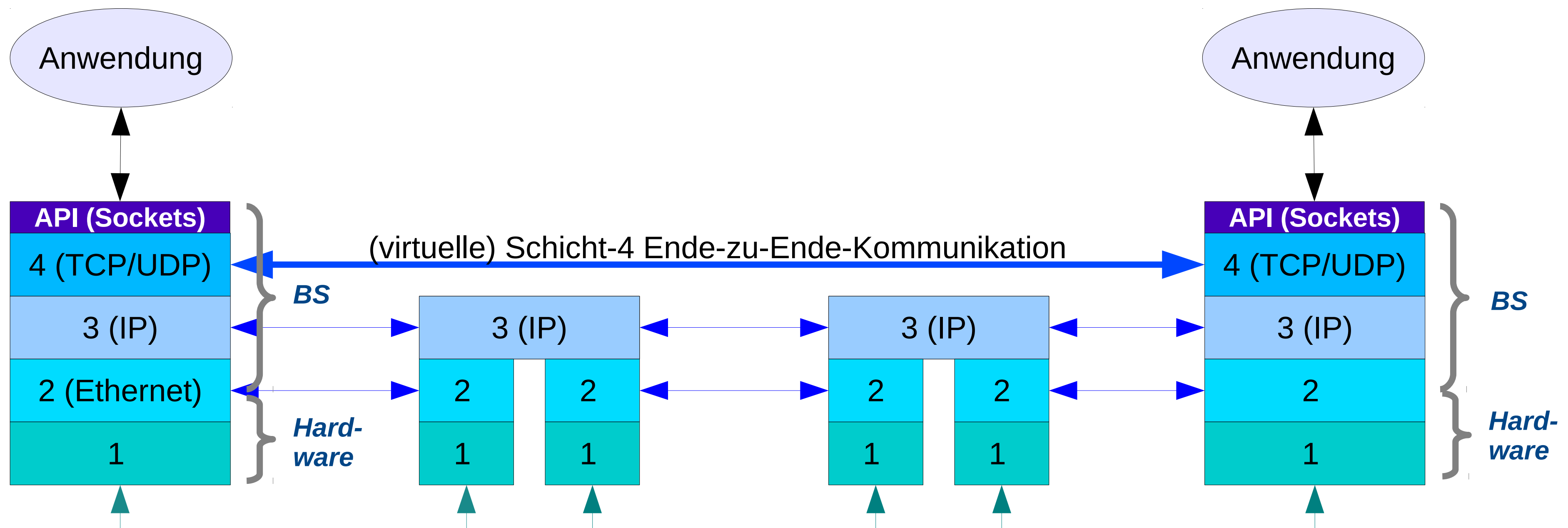
Klassische Netzwerkprogrammierung

- Socket-I/O
- TCP-Kommunikation
- UDP-Kommunikation
- Server-Architektur: Nebenläufigkeit im Server

Socket-Programmierung

Sockets im Betriebssystem

- standardisierte Betriebssystem-Schnittstelle für Kommunikation
- ursprünglich nur eine von mehreren APIs und nur in Unix-Varianten verfügbar
  (Verteilte Anwendungen bestanden bis dahin auch aus Treiber-Programmen für Netzadapter)
- nach dem Siegeszug des Internets allgemeine verfügbare und weitgehend standardisierte BS-API für Netz-Kommunikation auf TCP-/UDP-Ebene



Socket-Programmierung

Sockets in Programmiersprachen

- Adapter-Bibliotheken, speziell für C++, vereinfachten ab Ende der 1980 den Umgang mit der Socket-API.

  (C-Programmierer bevorzug(t)en den Umgang mit der BS-API)
- Beginnend mit Java wurden sprachspezifische Adapter-Bibliotheken der BS-Socket-API in die Standard-Bibliotheken von Mainstream-Programmiersprachen übernommen.

Netzwerk-Programmierung / Socket-Programmierung

- „Netzwerk-Programmierung“ = Programmierung auf der (BS- / Sprach-) Socket-API

  gelegentlich auch: Client-Server-Programmierung
- Älteste Art der Entwicklung verteilter Anwendung
- Immer noch wichtig und die Basis aller verteilten Anwendungen
- Für die Anwendungsentwicklung werden zunehmend auf Technologien mit einer höheren Abstraktionsstufe entwickelt.

TCP/IP

IP : Netzprotokoll

- Ungesicherte paket-orientierte Kommunikation
- duplex
- Endpunkt-Identifikation: IP-Adresse

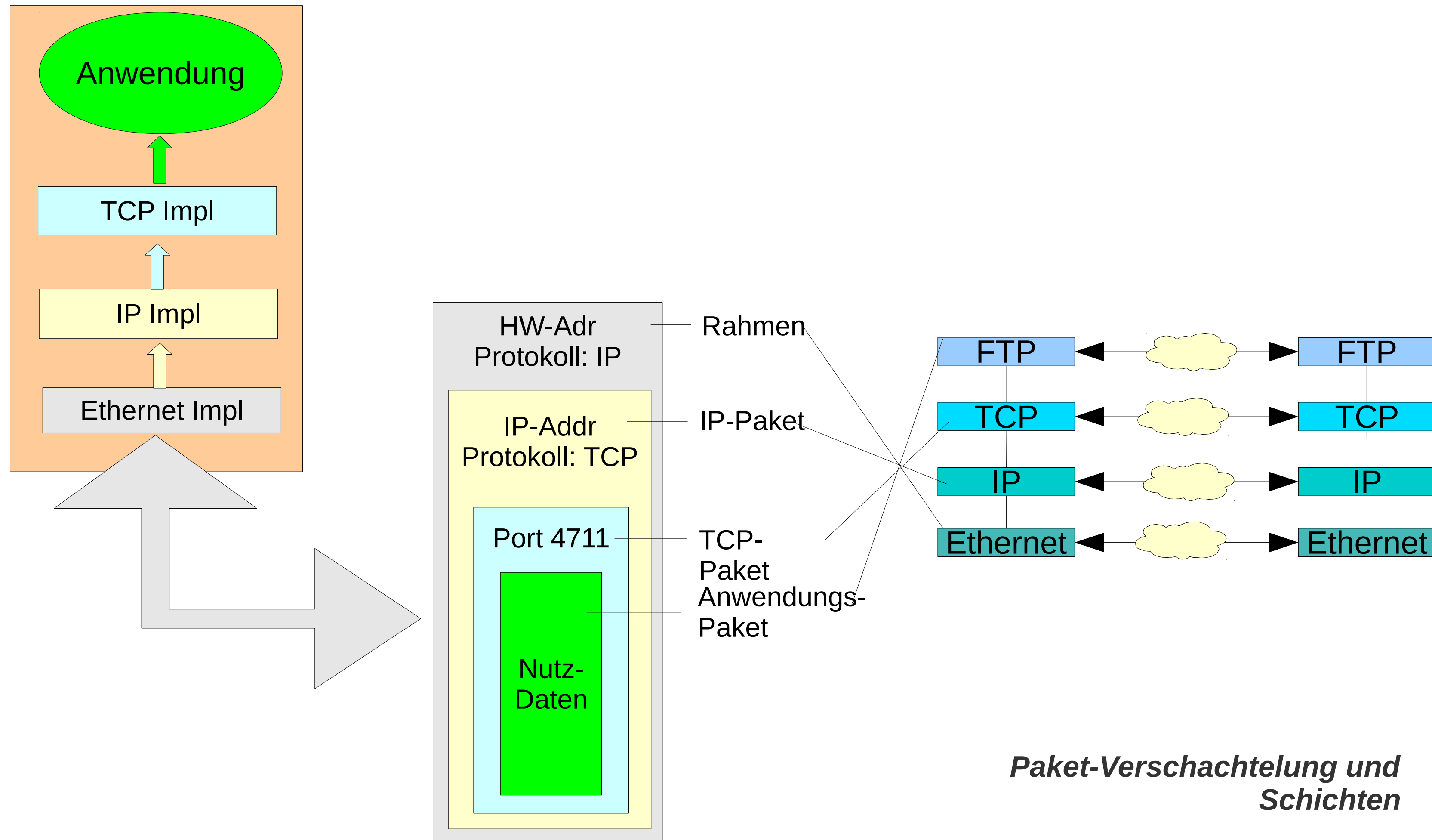
UDP: Ende-zu-Ende-Protokoll

- Ungesicherte Paket-Kommunikation
- Anwender-Schnittstelle zu IP
- Endpunkt-Identifikation: IP-Adresse + UDP-Port

TCP : Ende-zu-Ende-Protokoll

- Daten-Strom-Kommunikation
- Gesichert
- Endpunkt-Identifikation: IP-Adresse + TCP-Port

TCP/IP

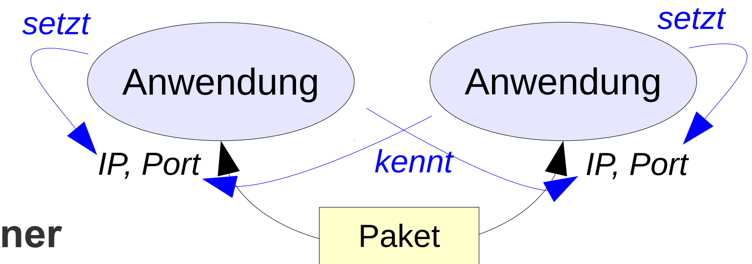


Paket-Verschachtelung und Schichten

TCP/IP

UDP-Kommunikation

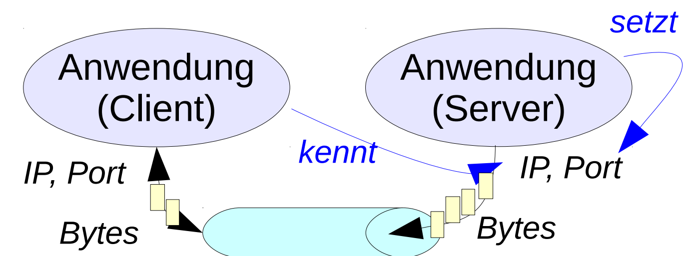
- Datenpakete zu IP-Adresse/Port senden, von IP-Adresse/Port empfangen
- Kommunikation symmetrisch (Die Kommuniaktions-Partner können die gleichen Operationen ausführen)



UDP- Kommunikation: Pakete packen und versenden

TCP : Ende-zu-Ende-Protokoll

- Verbindung von IP-Adress/Port zu IP-Adress/Port herstellen
- Verbindung als bidirektionalen Bytestrom nutzen
- Kommunikation asymetrisch
  - Ein „passiver“ Partner erwartet Verbindungswünsche
  - Ein „aktiver“ Partner stellt Verbindung her
  - Senden / empfangen ist symtrisch.



TCP- Kommunikation: Verbindungen aufbauen Bytes senden

Socket

SAP *(Service Access Point)* Verbindungsendpunkt der Schicht-4:

SAP ~ Socketadresse = IP-Adresse + Port-Nummer

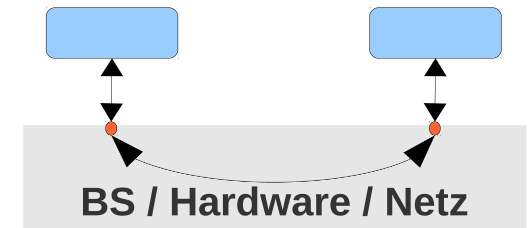
entsprechend der Konzeption von TCP und UDP

eigenen SAP bereitstellen

- Socket als SAP erzeugen
- Socket an Socketadresse binden
- Kommunikation / Verbindungswünsche an Socket erwarten

fremden SAP nutzen

- Socket erzeugen
- Daten via eigenen Socket an fremde Socketadresse senden
- eigenen Socket mit fremder Socketadresse verbinden



Klassen der Socket-API

Socket

Kommunikations-Schnittstelle bei verbindungsorientierter Kommunikation (TCP)

ServerSocket

Schnittstelle für die Annahme von Verbindungswünschen (TCP)

DatagrammSocket

Kommunikations-Schnittstelle bei verbindungsloser Kommunikation (UDP)

Diverse Hilfsklassen

– DatagramPacket Datenpaket bei UDP (verbindungsloser) Komm.
– InetAddress Umschlag-Klasse für IP-Adressen
– SocketAddress IP-Adress + Port
– ...

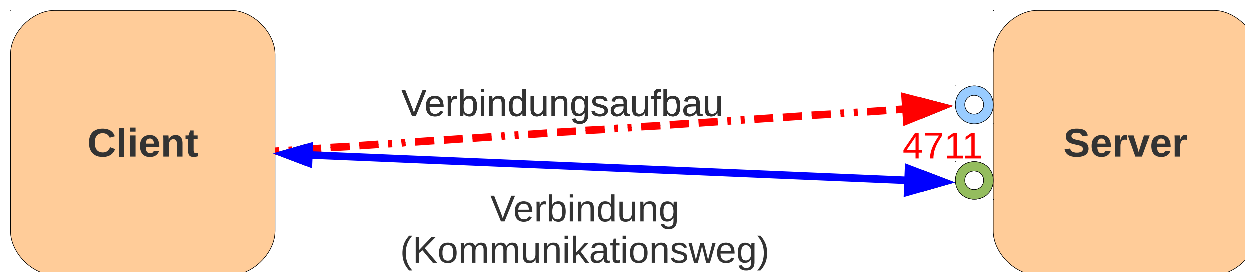
Verbindungsorientierte Kommunikation

Server: Stelle Port zur Verfügung, Akzeptiere Verbindung vom Client

- *Server*: Erzeuge Server-Socket
```
ServerSocket serverSock = new ServerSocket(4711);
```
eigene Adresse setzen (IP/Port, IP-Adr. meist implizit)

- *Server*: Erwarte Verbindungswunsch
```
Socket sock = serverSock.accept();
```

- *Client*: verbindet sich
```
Socket sock = new Socket(“127.0.0.1“, 4711);
```
fremde Adresse (IP/Port) angeben



4711 : Port an dem Verbindungswünsche angenommen werden.

: Server-seitiger Endpunkt für Verbindungswünsche

: Server-seitiger Endpunkt der aufgebauten Verbindung.

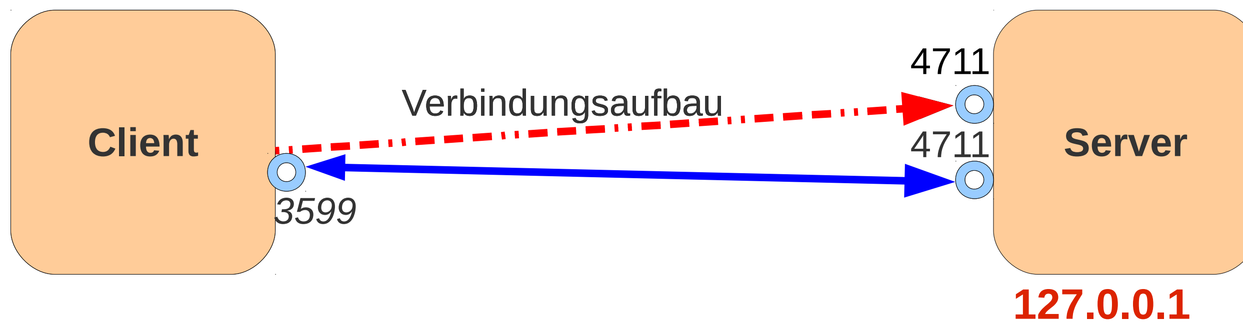
Verbindungsorientierte Kommunikation

Client: Stelle Verbindung zum Server her

- Client verbindet sich

```
Socket socket = new Socket("127.0.0.1", 4711);
```

fremde Adresse setzen (IP/Port), eigene Adresse meist implizit)



Der Client gibt die Serveradresse an.
Die eigene Adresse wird implizit gesetzt.
Beim Verbindungsaufbau wird die Client-Adresse dem Server bekannt gegeben.
Die Anwendung muss sich nicht um die Adressen der Clientseite (z.B.: 3599) kümmen.

Verbindungsorientierte Kommunikation

Client: Empfängt Daten

– Stelle Verbindung zum Server her

```
Socket sock = new Socket("127.0.0.1", 4711);
```

– Erzeuge InputStreamReader aus dem Eingabestrom der Verbindung

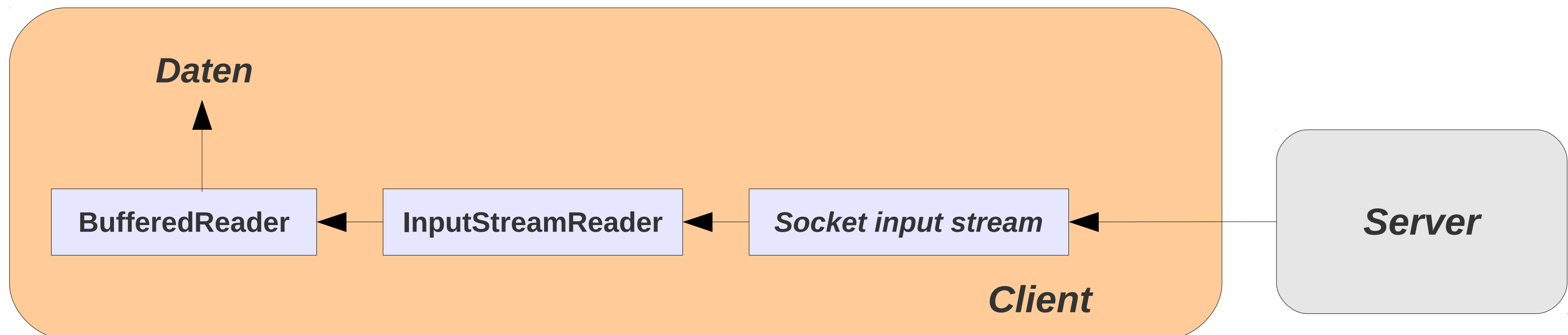
```
InputStreamReader stream = new InputStreamReader(sock.getInputStream());
```

– Erzeuge einen BufferedReader

```
BufferedReader reader = new BufferedReader(stream);
```

– Lies

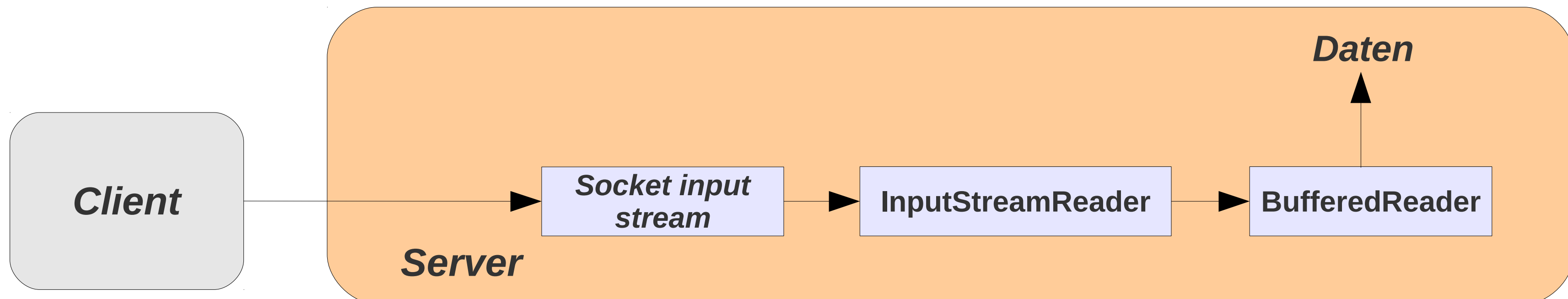
```
String msg = reader.readLine();
```



Verbindungsorientierte Kommunikation

Server: Empfängt Daten

- Verbindungsaufbau durch Client
```
Socket serverSock = new ServerSocket(4711);
Socket sock = serverSock.accept();
```
- Erzeuge InputStreamReader aus dem Eingabestrom der Verbindung
```
InputStreamReader stream = new InputStreamReader(sock.getInputStream());
```
- Erzeuge einen BufferedReader
```
BufferedReader reader = new BufferedReader(stream);
```
- Lies
```
String msg = reader.readLine();
```



Verbindungsorientierte Kommunikation

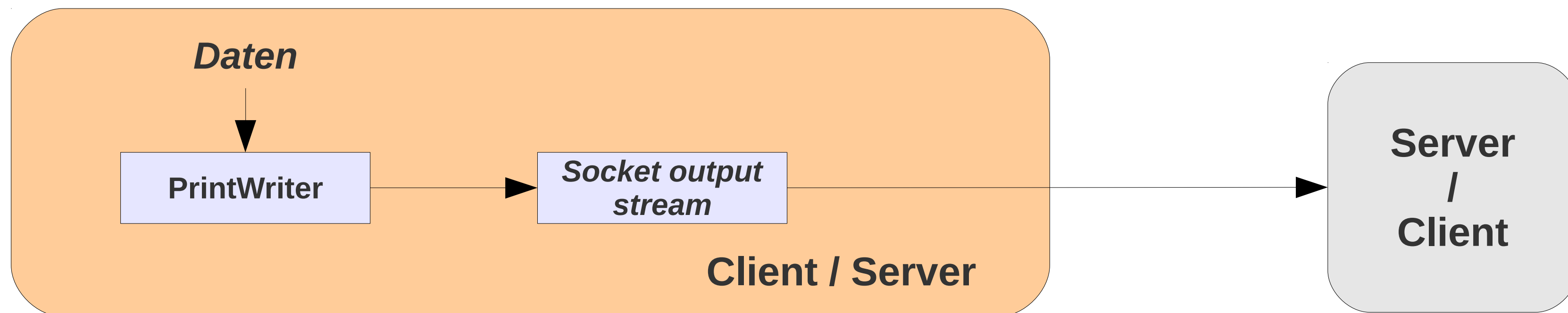
Client / Server: Sendet Daten

– Erzeuge Printwriter aus dem Ausgabestrom der Verbindung

```
PrintWriter writer = new PrintWriter(sock.getOutputStream());
```

– Schreibe (Sende)

```
writer.println(''Hallo Du da, alles klar?'')
```



Sockets erzeugen

Mit Konstruktoren

```
public Socket(String host, int port) throws UnknownHostException,IOException
public Socket(InetAddress address, int port) throws IOException
public Socket(String host, int port, InetAddress localAddr,int localPort)
        throws IOException
public Socket(InetAddress address,int port,
              InetAddress localAddr, int localPort) throws IOException
```

Der entfernte Rechner und der entfernte Port müssen mindestens angegeben werden

Der entfernte (oder eigene) Rechner kann angegeben werden

- **als String (Rechner-Name oder *dotted-decimal* IP-Adresse) oder**
- **als Objekt vom Typ InetAddress**

Der Port ist Bereich 1 bis 65535 (1 bis 1023 auf Unix für *root* reserviert)

Der entfernte Rechner muss am angegebenen Port auf Verbindungen warten

Ein Socket kann immer nur mit einem anderen verbunden sein

Sockets erzeugen

Binden

```
public ServerSocket(int port) throws IOException
public ServerSocket(int port, int backlog) throws IOException
public ServerSocket(int port, int backlog, InetAddress bindAddr) throws IOException
```

Der eigene Port muss mindestens angegeben werden (0 ~ irgendein freier Port)

Der eigene Rechner kann angegeben werden

- **als Objekt vom Typ InetAddress**
- **wird keine Adresse angegeben dann bindet sich der Socket an den Port bei allen verfügbaren IP-Adressen (Schnittstellen) des Rechners**

Ankommende Verbindungswünsche werden bei Bedarf gepuffert, backlog setzt Puffergröße

Über einen ServerSocket können sich viele Clients mit einem Server verbinden

An einen Port (pro IP-Adresse) kann nur ein ServerSocket gebunden sein

accept blockiert bis ein Verbindungswunsch eintrifft und liefert dann ein Socket-Objekt das zur Kommunikation genutzt werden kann

Verbindungsorientierte Kommunikation

Beispiel: Echo-Server

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.io.IOException

import scala.util.{ Try, Success, Failure }

object EchoServer_Main extends App {

  val ECHO_PORT = 4713;

  val serverSockTry = Try[ServerSocket] {
    new ServerSocket(ECHO_PORT)
  }

  serverSockTry match {
    case Failure(e) => e.printStackTrace()
    case Success(serverSocket) => TCPServer(serverSocket)
  }
}
```

```
object TCPServer {
  def apply(serverSock: ServerSocket): Unit = {
    while (true) {
      try {
        val sock = serverSock.accept();
        val pw = new PrintWriter(sock.getOutputStream());
        val stream = new InputStreamReader(sock.getInputStream());
        val reader = new BufferedReader(stream);

        val msg = reader.readLine();
        val response = "Echo: "+ msg;
        pw.println(response);
        pw.flush();
        reader.close();
        pw.close();
        sock.close();
      } catch {
        case e: IOException => /* continue */
      }
    }
  }
}
```

Verbindungsorientierte Kommunikation

Beispiel: Echo-Client

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

import scala.util.{ Try, Success, Failure }

object EchoCient_Main extends App {
  val ECHO_PORT = 4713;
  val ECHO_HOST = "127.0.0.1"

  val sockTry = Try[Socket] {
    new Socket(ECHO_HOST, ECHO_PORT)
  }

  sockTry match {
    case Failure(e) => e.printStackTrace()
    case Success(socket) => TCPClient(socket)
  }
}
```

```
object TCPClient {
  def apply(sock: Socket): Unit = {
    try {
      val pw = new PrintWriter(sock.getOutputStream());
      val stream = new InputStreamReader(sock.getInputStream());
      val reader = new BufferedReader(stream);

      pw.println("Hallo");
      pw.flush();

      val response = reader.readLine();
      println("Received from Server: "+ response);

      reader.close();
      pw.close();
      sock.close();
    } catch {
       case e: IOException => e.printStackTrace()
    }
  }
}
```

UDP-Kommunikation

User Datagram Protocol

- UDP verbindungsloses IP-Transport-Protokoll (Schicht-4)
- UDP ist paketorientiert, Paket = Datagramm ~ IP-Paket
- keine Fehler- / Fluss-Kontrolle
- schneller / einfacher als TCP
- im LAN oft ausreichend sicher

UDP Nutzung via Socket-API

- Paket-orientierte Kommunikation (statt Strom-orientiert)
- Pakete senden / empfangen (statt Bytes schreiben / lesen)
- Symmetrische Struktur (statt Client vs. Server)

UDP-Unterstützung in Java

Klasse DatagrammPaket

- Zum Senden und Empfangen von Datenpaketen
- beinhaltet Daten (Byte-Array)
  - theoretische maximale Paketgröße: 65536 Bytes
  - realistische (Netz-unterstützte) Größe: 8192 Bytes
- beinhaltet Socket-Adresse (IP + Port)
- Konstuktor auf der Empfangsseite (Adresse ergibt sich empfangenem Paket):
  - DatagramPacket (byte [] data, int length)
- Konstruktor auf der Sendeseite
  - DatagramPacket (byte [] data, int length, InetAddress address, int port)

Klasse DatagramSocket

- kann zum Senden und Empfangen verwendet werden

Verbindungslose Kommunikation

Beispiel: Echo-Server

```
import java.io.IOException
import java.net.DatagramPacket
import java.net.DatagramSocket
import java.net.InetAddress

import scala.util.{ Try, Success, Failure }

object UDPEchoServer_Main extends App {
  val ECHO_PORT = 4713

  val dtgrmSocketTry = Try[DatagramSocket] {
    new DatagramSocket(ECHO_PORT)
  }

  dtgrmSocketTry match {
    case Failure(e) => e.printStackTrace()
    case Success(dtgrmSocket) => UDPServer(dtgrmSocket)
  }

}
```

```
object UDPServer {
 def apply(dtgrmSocket: DatagramSocket): Unit = {

   try {
     println("Server ready")
     // data: receive and send:
     val buf = new Array[Byte](256)
     val rcvpkt = new DatagramPacket(buf, buf.length)

     while (true) {
           dtgrmSocket.receive(rcvpkt);
           println( "Server got " +
                   new String(rcvpkt.getData(), 0,
                   rcvpkt.getLength()));
           val clientAdr = rcvpkt.getAddress();
           val clientPort       = rcvpkt.getPort();
           println("From host " +
                   clientAdr.getHostAddress() +
                   " at Port "+clientPort);
           val sndpkt =  new DatagramPacket(
                   buf,
                   buf.length,
                   clientAdr,
                   clientPort);
           dtgrmSocket.send(sndpkt);
       }
   } catch {
     case e: IOException => e.printStackTrace()
   }
 }
}
```

Verbindungslose Kommunikation

Beispiel: Echo-Client

```scala
import java.net.DatagramPacket
import java.net.DatagramSocket
import java.net.InetAddress
import java.net.SocketException

import scala.util.{ Try, Success, Failure }

object UDPEchoClient_Main extends App {
  val ECHO_PORT = 4713
  val host = "127.0.0.1";
  val port = ECHO_PORT;

  val dtgrmSocket = new DatagramSocket()
  val serverAddress = InetAddress.getByName(host);

  val msg = "Hello, is someone out there?";
  val buf = msg.getBytes();
  val pkt = new DatagramPacket(buf, buf.length, serverAddress, port);
  dtgrmSocket.send(pkt);

  dtgrmSocket.receive(pkt);
  println("Client received: " +new String(pkt.getData()));
}
```

UDP-Kommunikation

Klassen

java.net.DatagramPacket

- Umschlag um zu sendende / zu empfangende Daten als Byte-Array
- Enthält Adresse (IP + Port) des Ziels der Sendung / des Senders

java.net.DatagramSocket

- Verbindung mit lokaler Adresse (IP + Port)
- Via DatagramSocket kann zu unterschiedlichen Zielen gesendet werden (Ziel-Adress-Info steckt im Paket nicht im Socket)
- UDP-Ports im Bereich 1 bis 65,536, völlig unabhängig von TCP-Ports.

java.net.InetAddress

- Umschlag-Klasse für IP-Adresse
- Enthält Routinen zur Adressauflösung, z.B.:

```
public static InetAddress getByName(String host) throws UnknownHostException
```

DatagramPacket Konstruktoren

- Empfängerseite:

```
public DatagramPacket( byte[] data, int length)
```

- Senderseite:

```
public DatagramPacket( byte[] data, int length,
                       InetAddress iaddr, int iport)
```

Datagramme senden

Daten in Bytes konvertieren und in Byte-Array speichern

Erzeuge `DatagramPacket` aus:

- Byte-Array
- Anzahl der zu sendenden Bytes (i.d.R. Länge des Arrays)
- IP-Adresse als `InetAddress`-Objekt
- Port-Nr

Erzeuge `DatagramSocket`

aktiviere dessen `send`-Methode mit DatagramPacket als Argument

Datagramme empfangen

Erzeuge `DatagramSocket`-Objekt mit lokalem Port (+ eventuell IP-Adresse) über den (die) empfangen werden soll

receive aufrufen mit leerem `DatagramPacket`-Objekt blockiert bis Daten-Paket eintrifft

Leeres `DatagramPacket`-Objekt wird bei Empfang gefüllt

Mit den `DatagramPacket`-Methoden :

- `getPort()`
- `getAddress()`

kann festgestellt werden woher die Daten kommen und

- `getData()`
- `getLength()`

liefern die empfangenen Daten als Byte-Array und dessen Länge

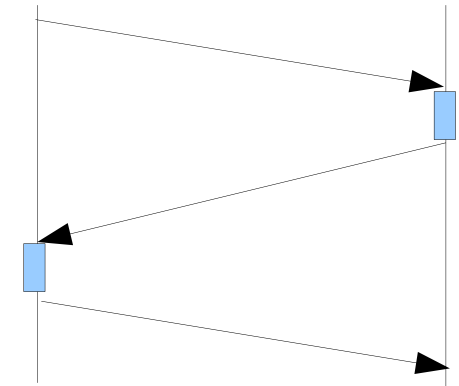
Synchrone / Asynchrone Kommunikation

- **synchrones *Request-Response* Protokoll**

  Neue Anfragen erst nach Antwort auf die vorhergehende

  Einsatz

  - **Neue Anfragen hängen von der Antwort ab**
  - **Kurze Bearbeitungszeit + geringe Netz-Latenz (LAN)**
  - **Einfachheit vor Performanz**



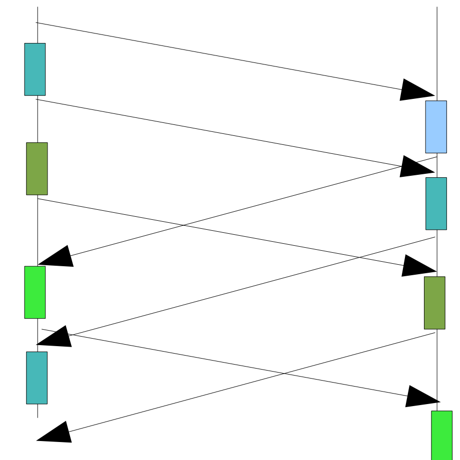
synchrones Protokoll

- **asynchrones *Request-Response* Protokoll**

  Anfragen und Antwort entkoppelt

  Einsatz

  - **Anfragen und Antworten sind nicht eng gekoppelt (z.B. HTTP-Gets)**
  - **Latenz im Netz groß (WAN)**
  - **Performanz vor Einfachheit**



asynchrones Protokoll

Verbindungen und Sitzungen

Verbindung (Connection)

- Schicht-4 Konzept
- Fehler-/Flusskontrolle erfordert Verbindung

  Verbindung = Information über den Zustand der Kommunikation
  z.B. Welches Datenpaket wurde gesendet ist aber noch noch nicht bestätigt

Sitzung (Session)

- Anwendungskonzept
- „logische" Verbindung der Kommunikationspartner für die Dauer einer Interaktion

  Stand der Diskussion der Partner auf Anwendungsebene
  z.B. Mit welchem Account hat sich der Partner identifiziert
- OSI-Modell: Session-Layer (so praktisch nie realisiert, Sitzungen werden von der Anwendung verwaltet)

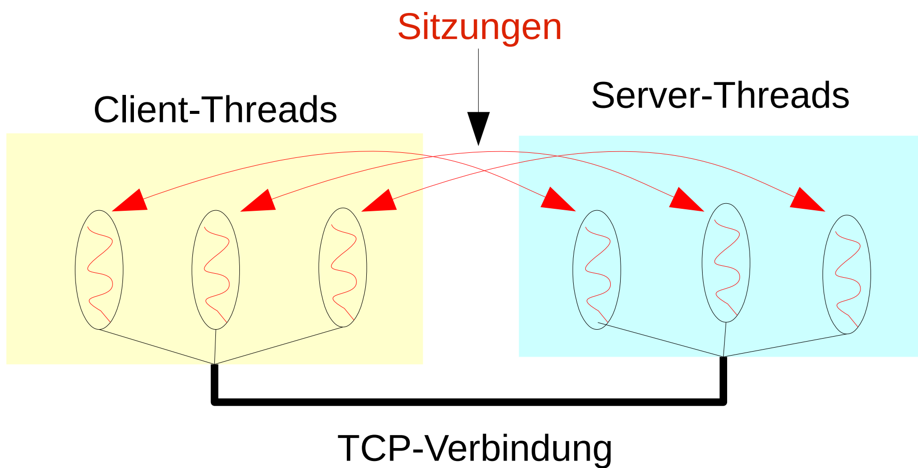
Verbindungen und Sitzungen

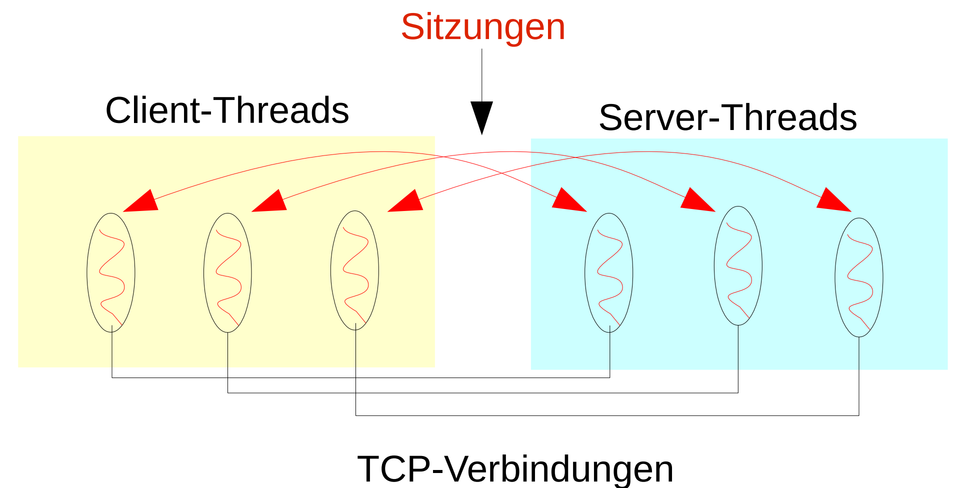
ohne Multiplexing der Sitzungen

Jede Beziehung *Client (Thread) – Service-Provider (Thread) (Session / Sitzung)* hat ihre eigene Verbindung

mit Multiplexing der Sitzungen

Eine gemeinsame Verbindung für jede Beziehung *Client – Service-Provider*





Dienste

zustandsloser Dienst *(stateless service)*

- Dienst wird ohne Zustandsinformation im Server ausgeführt, jede Anfrage enthält alle Informationen, die notwendig sind, um sie auszuführen
- Beispiel:
  - einfache Dienste,
  - NFS
  - HTTP (ohne Cookies, etc.)

zustandsbehafteter Dienst *(stateful service)*

- Anfragen werden in Zustand ausgeführt,
- Zustand
  - Verbindungs-/Sitzungs -Zustand: Protokoll wickelt FSM (endlichen Automat) ab
  - dauerhafter Zustand: Zustand dauerhaft (über die Dauer einer Sitzung hinaus)
  - Zustand Absturz-resistent
- Beispiel:
  - Telnet / FTP: Sitzungszustand
  - Naming-Services: Absturz-resistent

Nebenläufigkeit – *Concurrency Model*

Threads und Nachrichten-Verabeitung

- iterativer Server: behandelt eine Nachricht nach der anderen
- nebenläufiger Server: behandelt mehrere Nachrichten gleichzeitig
  einfacher mit Threads zu realisieren

Threads und Verbindungen

- Ein Thread für viele Verbindungen: möglich aber in Java lange Zeit (d.h. bis ca 2005) komplex und unüblich
- pro Verbindung ein Thread: üblich (bei Hobbyprogrammen) und einfach

Threads und Sitzungen

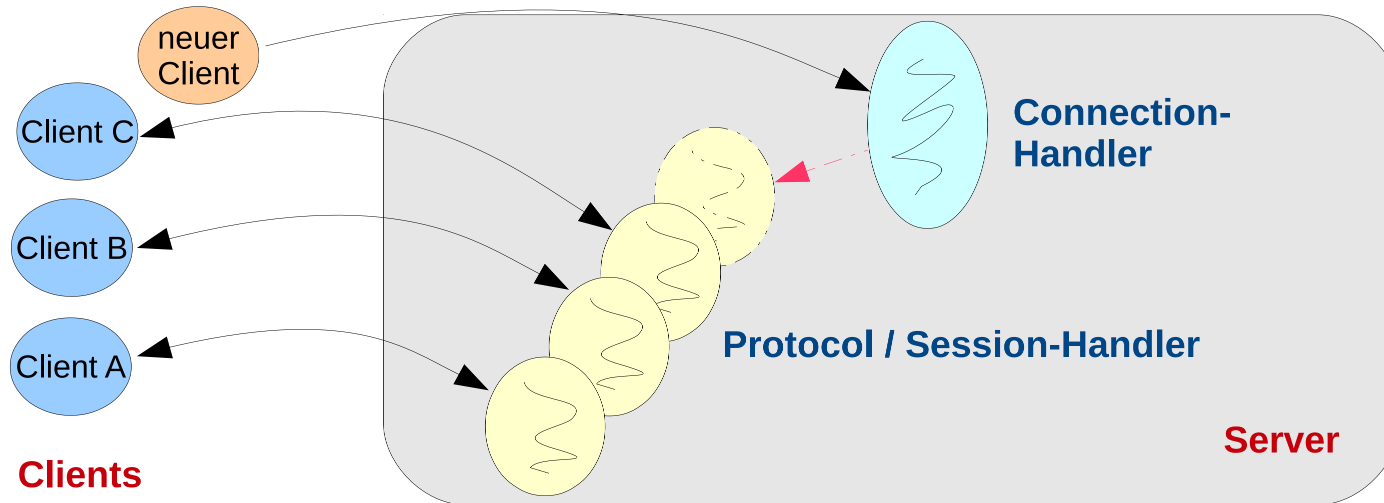
- Wenn jede Sitzung einen zugeordneten Thread hat, dann kann dieser die Zustandsinformationen der Sitzung verwalten.
- Oft – aber nicht immer – ist jeder Sitzung eine Verbindung zugeordnet. Aber es kann Sitzungen ohne Verbindungen und Verbindungen ohne Sitzungen geben

Nebenläufigkeit – *Concurrency Model*

Ziele

- Skalierbarkeit: Fähigkeit des Servers sich an zunehmende Zahl von Clients anzupassen
- Geringe Latenz:
  - Kurze Zeit bis zur Annahme eines Verbindungswunschs
  - Kurze Zeit bis zur Reaktion auf eine Nachricht
- Effizienz
  - geringer Ressourcenverbrauch:
    CPU, Speicher, Threads, ...

Beispiel



übliches Szenario der Implementierung es zustandsbehafteten Protokolls und / oder eines auf Verbindungen basierenden Protokolls mit Threads. Ein Thread behandelt eine Sitzung und/oder eine Verbindung.

```
do for ever {
  handle = accept new connection;
  thread = get SessionThread(handle);
  thread.start(handle);
}
```

Connection-Handler

```
while ( ! finished ) {
  msg = handle.getMsg();
  answer = preform service
  handle.send(answer)
}
```

Session-Handler

Iterativer Server

behandelt eine Anfrage komplett ab bevor die nächste betrachtet wird

- WS-Strategie
  - Warteschlange für eintreffende Anfragen
  - Ignorieren

geeignet für

- Dienste mit kurzen Bearbeitungszeiten
- unregelmäßig und eher selten angefragte Dienste

Vorteil

- einfach
- kein Thread- / Prozess-Overhead

Nachteil

- eventuell schlechte Nutzung der Plattform-Fähigkeiten
  mehrere CPU's, asynchroner DMA-Transfer, ...
- schlechte Antwortzeiten / Verlust von Anfragen
- eventuell Sende-Wiederholung bei Time-out -> Problemverschärfung

Struktur

```
do forever:
        retrieve request
        perform service
        send response
```

Server-Architektur: Nebenläufigkeit im Server

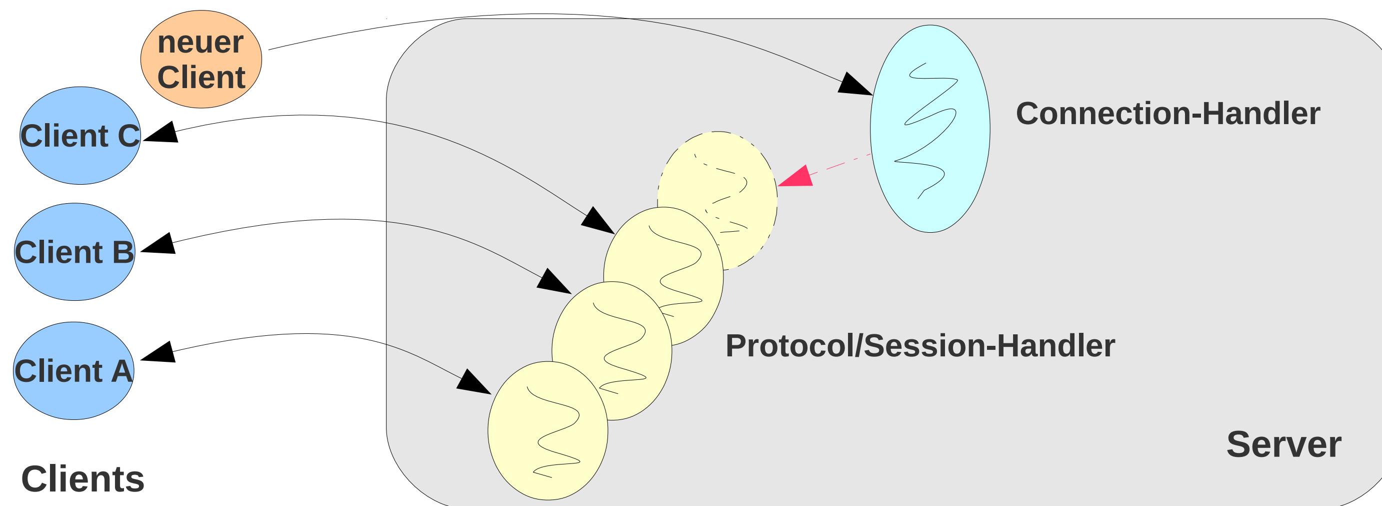
Nebenläufiger Server: Thread pro Session / Verbindung

```
do forever {
     handle = accept new connection;
     thread = getSessionThread(handle);
     thread.start();
}
```

Connection-Handler

```
while ( ! finished ) {
     msg = handle.getMsg();
     answer = preform service
     handle.send(answer)
}
```

Verbindungs- /Protokoll-Handler



Server-Architektur: Nebenläufigkeit im Server

```
object FactorizationServer_Main extends App {

  val serverPort = 4713; // port at which the server accepts requests
  val serverSocket = new DatagramSocket(serverPort); // my socket

  val buf = new Array[Byte](256) // receive buffer
  val rcvpkt = new DatagramPacket(buf, buf.length) // receive packet

  while (true) {

    println("server waits for msg")

    serverSocket.receive(rcvpkt)

    val msg        = new String(rcvpkt.getData(), 0, rcvpkt.getLength(), "UTF-8");
    val clientAdr  = rcvpkt.getAddress()
    val clientPort = rcvpkt.getPort()

    println(s"server received msg $msgEphemeral from $clientAdrEphemeral at port $clientPortEphemeral")

    Future {

      val value = msg.toLong
      val result = Factorization.factors(value)

      val resultS = new StringBuilder
      result.foreach { factor => resultS.append("" + factor + " ") }
      val buf = resultS.toString().getBytes();

      // create packet and send it:
      val sndpkt = new DatagramPacket(buf, buf.length, clientAdr, clientPort);
      serverSocket.send(sndpkt);
    } onComplete {
      case Success(_) => println("request was successfully processed")
      case Failure(t) => println(s"processing request failed because of $t")
    }

  }

}
```

Nebenläufiger Server:

Beispiel Faktorisierungs-Server (UDP)
zustandsloses Request-Response Protokoll
Session = [Abfrage + Bearbeitung + Antwort]
Verbindungslos

arbeitet mit Threadpool: Threads werden im Voraus erzeugt.

Achte auf konkurrierende Zugriffe im Main- und Handler-Thread!

```
import java.net.{ DatagramPacket, DatagramSocket, InetAddress }

import scala.concurrent.{Future, ExecutionContext}
import scala.util.{Success, Failure}
import ExecutionContext.Implicits.global
```

Threads erzeugen / aufspannen *(Thread Spawning)*

Erzeugungs-Strategien

- Im Voraus erzeugen *(eager spawning)*
- Bei Bedarf erzeugen *(on demand spawning)*

Im Voraus erzeugte Threads

- bei Start erzeugen und in einem Pool platzieren
- vermeidet Erzeugungs-/Vernichtungs-Overhead
- erfordert Thread-Verwaltung
- Größe des Thread-Pools
  - ~ Zahl der CPUs
  - ~ Last
  - ~ dynamisch variabel
    - ~ Last
    - ~ aktuelle Länge der Warteschlange der Anfragen

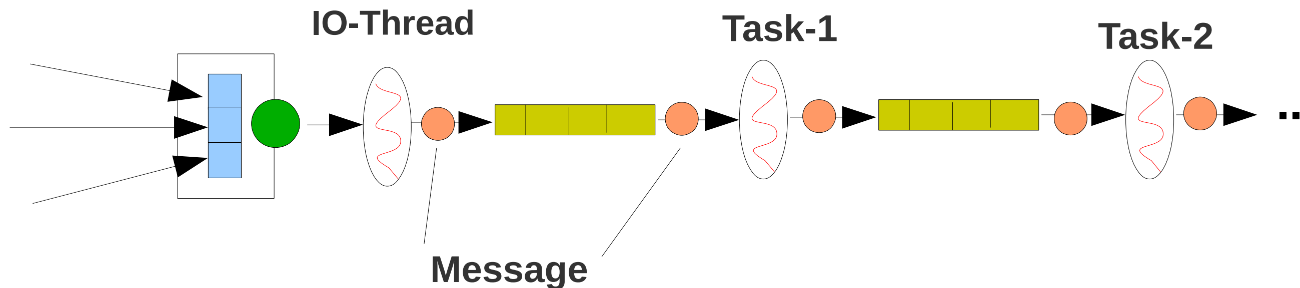
Threads erzeugen / aufspannen *(Thread Spawning)*

Bei Bedarf erzeugen *(on demand spawning)*

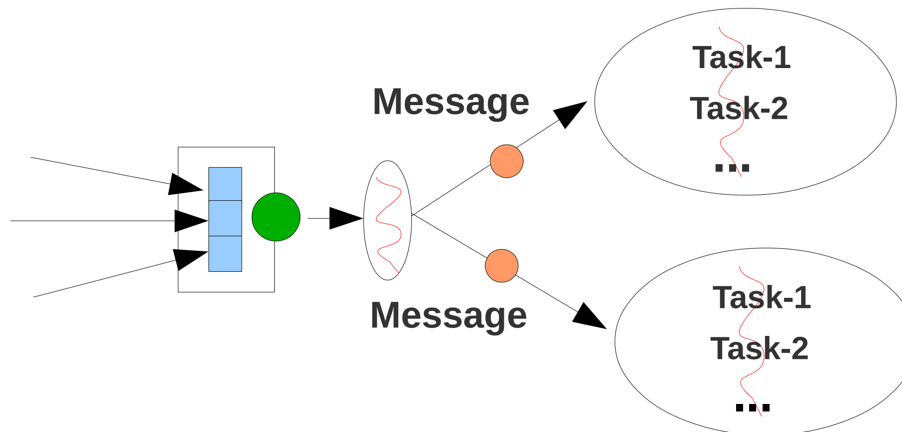
- Vorteil
  - Ressourcen-Schonung
- Nachteil
  - Schlecht konfigurierbar
  - Performance Probleme
    Denial-of-Service Attacken leicht möglich
  - längere Reaktionszeit

Threads erzeugen / aufspannen *(Thread Spawning)*

Task-basierte Thread-Zuteilung: pro Aufgabe ein Thread



Nachrichten-basierte Thread-Zuteilung: pro Nachricht / Session / Verbindung ein Thread



Server-Architektur: Nebenläufigkeit im Server

Threading-Strategie

ein Thread

- Ereignisse annehmen
- Bedienen der Ereignisse
- andere Aktivitäten

Schlecht: skaliert nicht

zwei Threads

- Ereignisse annehmen / Ereignisse bedienen
- andere Aktivitäten

OK bei 1-Prozessor System (welcher Server ist das noch?)

Thread-Pool

- Thread: Ereignisse annehmen
- Thread-Pool: Ereignisse bedienen

Anpassbar an System

Mehrere Threads / Thread-Pools

- Thread: Ereignisse annehmen
- Thread(-Pool) A: Ereignisklasse A
- Thread(-Pool) B: Ereignisklasse B
- etc.

Ereignisbehandlung konfigurierbar (Wichtigkeit, ...)

Thread-Erzeugung und Thread-Aufgaben

Thread-Erzeugung

– **Wann / Wie werden Threads erzeugt**

Thread vs. Task

– **Welche Aufgaben werden den Threads zugeteilt**

Thread vs. I/O Ereignis

– **welche(r) Thread(s) bearbeiten I/O-Ereignisse**