Technische Hochschule Rosenheim

Verteilte Verarbeitung

Kapitel 7

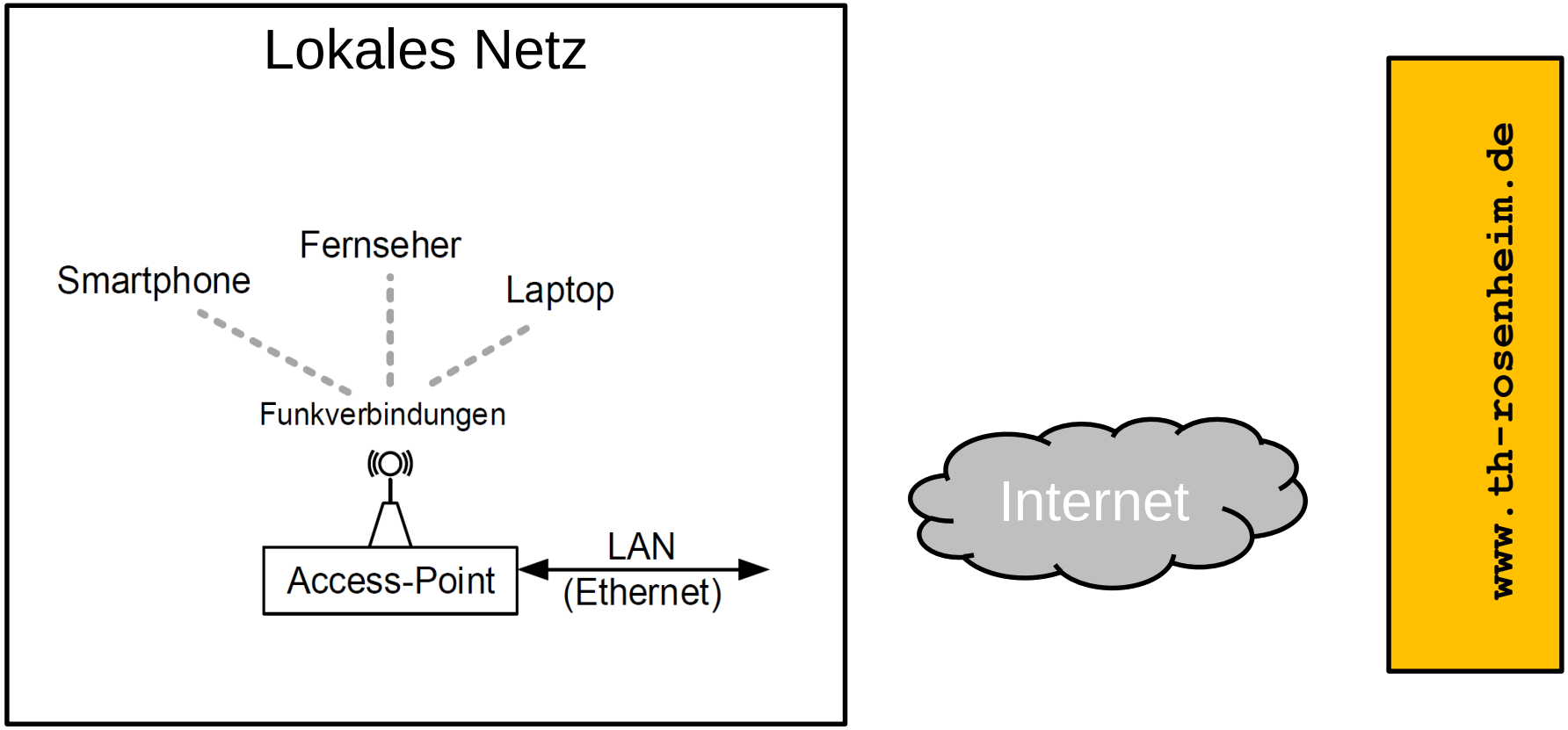
Netzwerkprogrammierung

Sockets, TCP/IP und UDP

Grundlagen aus RN …

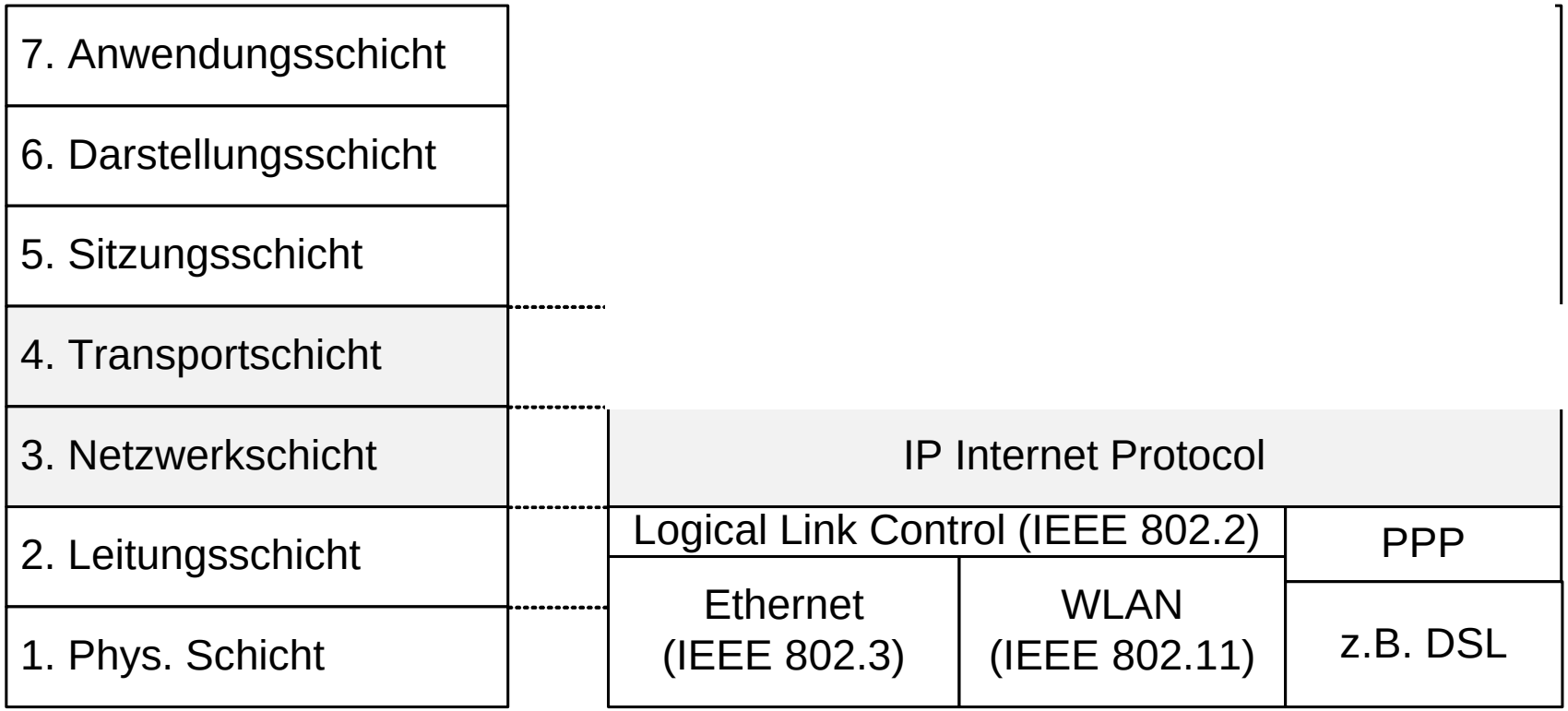
- ISO/OSI Schichtenmodell
- IP, TCP und UDP
- Internet = TCP/IP

Rechnernetz - WLAN



ISO / OSI – Schichtenmodell

(vgl. Vorlesung Rechnernetze)



Das IP - Protokoll

- IP-Protokoll = Internet Protocol (IEEE-Bericht, 1974, V.Cerf)
- Kommunikation von Rechner zu Rechner (IP-Adresse)
- ***Paketvermittelt:*** Datagramme werden ggf. auf verschiedenen Routen vom Sender zum Empfänger vermittelt
- ***Verbindungslos***
- ***Best-Effort Strategie***
  - Keine Garantie für die Reihenfolge der Pakete
  - Keine Garantie für das Ankommen der Pakete
    (Verloren gegangene Pakete bleiben unbemerkt)
  - IP-V4: Keine Garantien über Zustellzeiten oder Bandbreiten
    (= Problem z.B. bei Video/Audio Streaming), Trennung Intra- / Internet (NAT -> Grund: „Nur“ 4 Mrd. (= 2^32) Adressen möglich)
    - Beispiel für Adresse: 192.168.0.104 (private Adresse)
  - IP-V6: Wird immer noch eingeführt, QoS Garantien möglich,
    2^128 Adressen möglich (jedes Sandkorn adressieren)
    - Globale Adresse: 2a00:6020:19e9:9900:318d:1e5e:ef9f:e1c3
    - Lokale Adresse z.B.: fe80:0:0:0:fd08:916e:784b:d214

Klasse `InetAddress`

Die Klasse **`InetAddress`** repräsentiert IP-Adressen.

`static InetAddress getByName(String host)`
Ermittelt die InetAddress eines gegebenen Hosts

`static InetAddress getLocalHost()`
Ermittelt die InetAddress der aktuellen Maschine

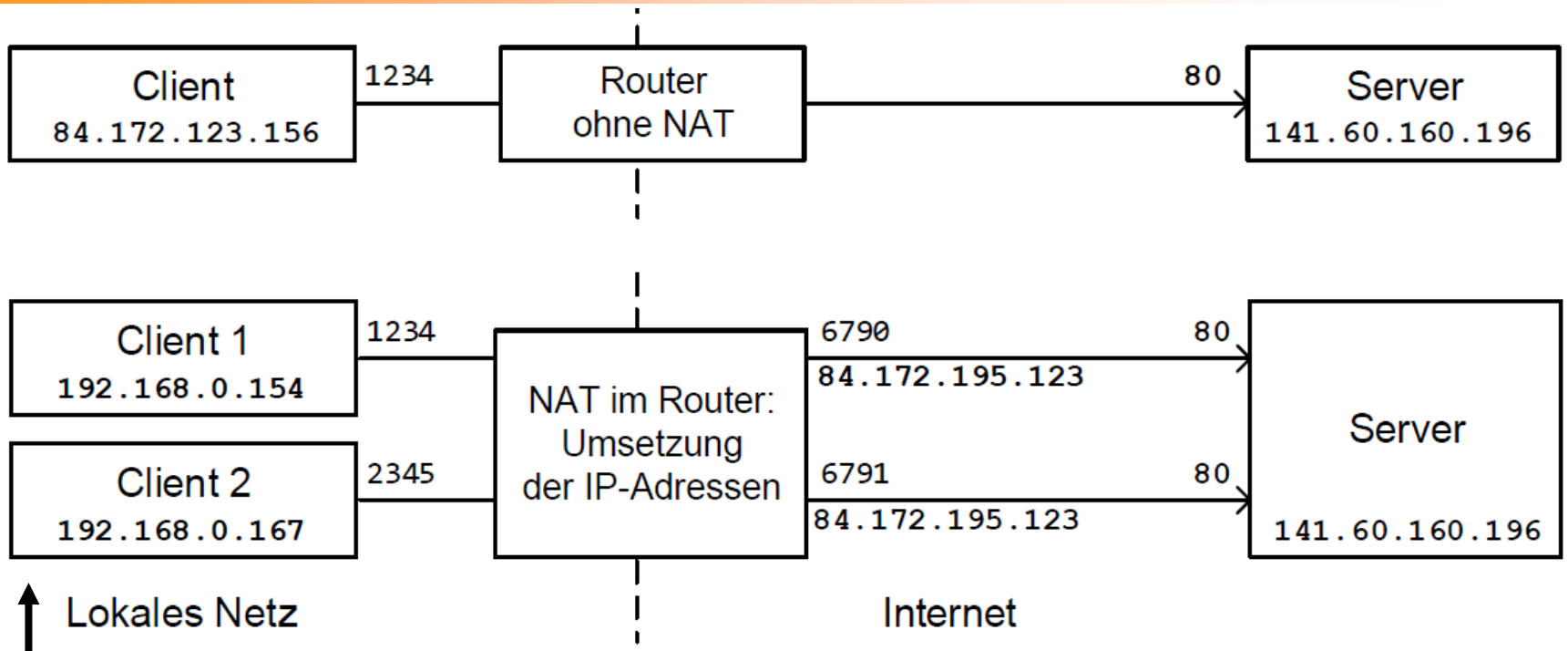
`String getHostName()`
Liefert den Rechnernamen zu der gegebenen IP-Adresse zurück

Subklassen: **`Inet4Address`** und **`Inet6Address`**

`InetAddress` Beispiel

```
public class WhoAmI {

    public static void main(String[] args) throws Exception {
      if (args.length != 1) {
        System.err.println("Usage: WhoAmI MachineName");
        System.exit(1);
      }

      InetAddress a = InetAddress.getByName(args[0]);
      System.out.println("I am =" + a);

      InetAddress localhost = InetAddress.getLocalHost();
      System.out.println("Localhost=" + localhost);
    }
}
```

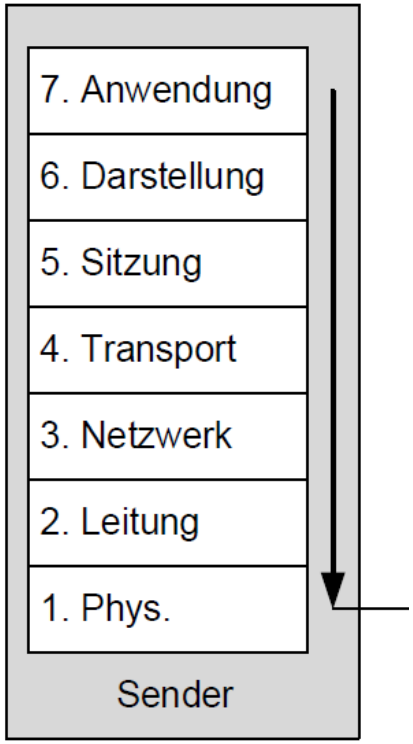
NAT für IPv4



Präfixe:

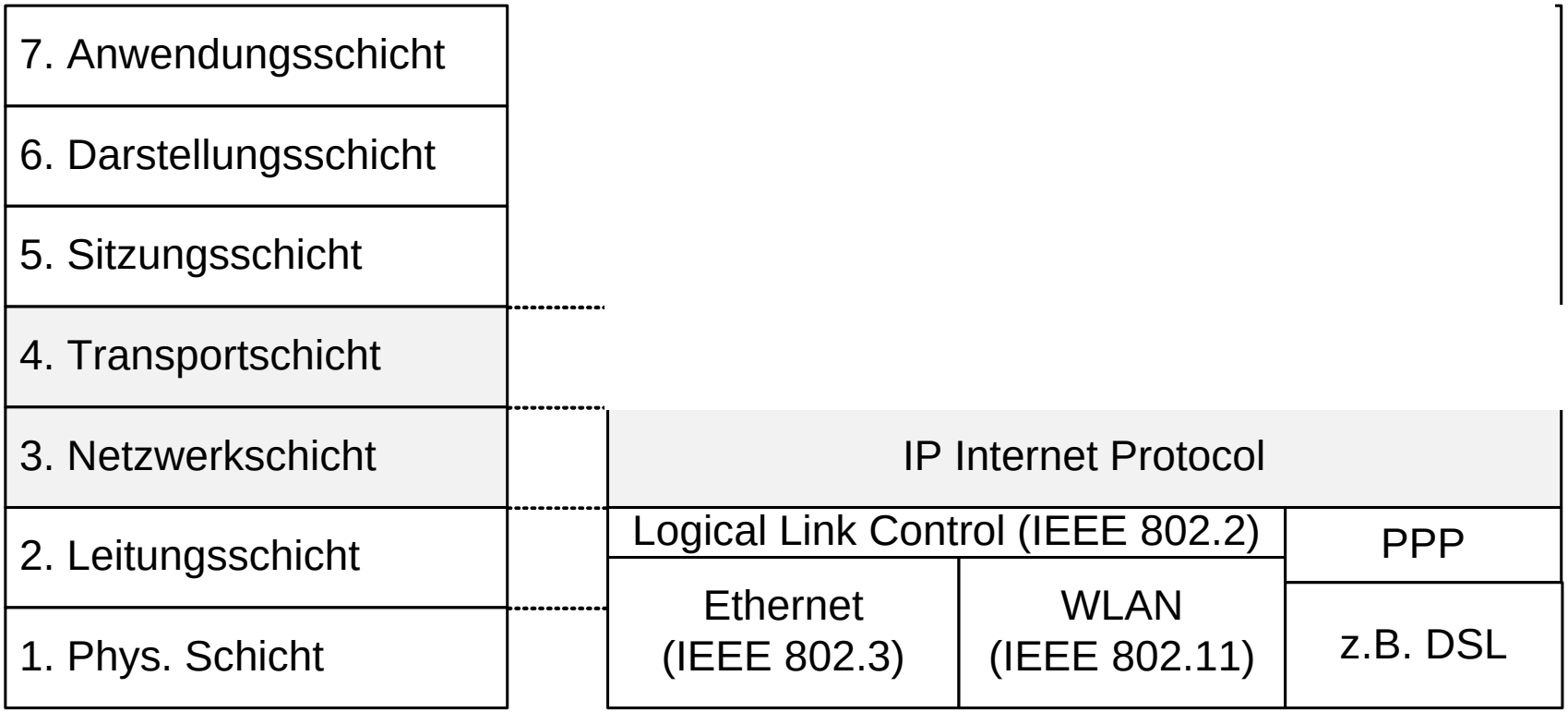
```
10.0.0.0      bis 10.255.255.255  also 10.0.0.0/8
176.16.0.0    bis 176.31.255.255  also 176.16.0.0/12
192.168.0.0   bis 192.168.255.255 also 192.168.0.0/16
```

Paketvermittlung über IP



ISO / OSI – Schichtenmodell

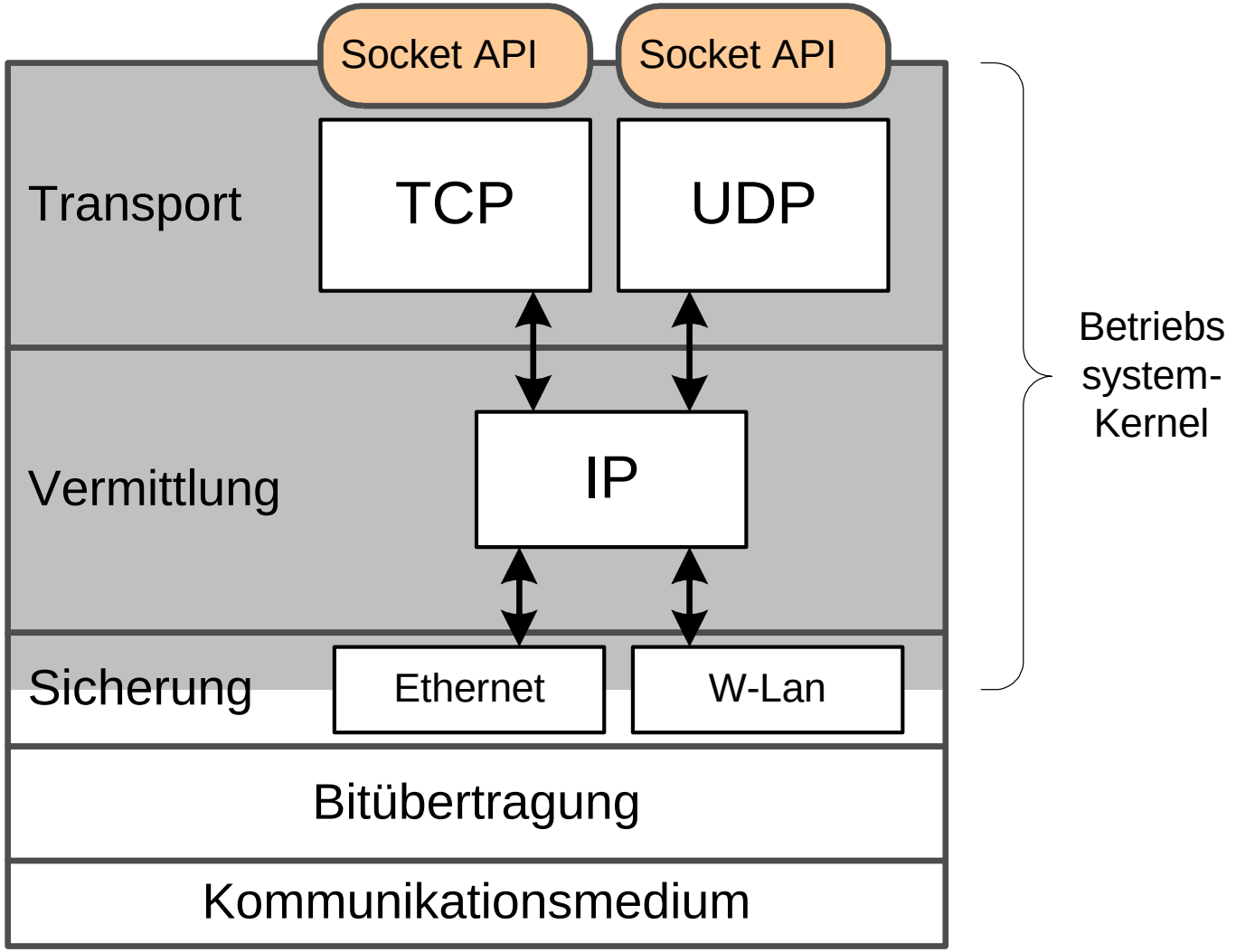
(vgl. Vorlesung Rechnernetze)



TCP und UDP

- Transportschicht = Kommunikation von Prozess zu Prozess
- Varianten: TCP (Zuverlässig) und UDP (Unzuverlässig)
- TCP
  - ***Garantiert Reihenfolge*** der Pakete
    -> Datenpakete sind nummeriert
  - ***Garantiert, dass Pakete ankommen*** oder eine ***Fehlermeldung***
    -> Empfänger sendet bei Empfang ACKnowledge
    -> Sender hat Timer, falls ACK ausbleibt
  - ***Verbindungsorientiert***, überträgt Byte-Ströme
- UDP
  - Gleiche Eigenschaften wie IP
  - ***Verbindungslos*** („Fire and Forget“)

Socket API


Socket API
Socket API
Transport
TCP
UDP
Vermittlung
IP
Sicherung
Ethernet
W-Lan
Bitübertragung
Kommunikationsmedium
Betriebs
system-
Kernel

Sockets

- ***Socket*** =

  „Kommunikationsendpunkt“, der ***mit Daten versorgt*** werden kann bzw. ***aus dem Daten herausgelesen*** werden kann (zwischen diesen Punkten liegt üblicherweise ein Netzwerk)

- Zuerst auf UNIX-Systemen (UNIX 4.3 BSD)
- Anfang/Mitte der 90er auf Windows (WinSockets)
- Baut auf TCP/UDP auf
- Serielle Übertragung von Informationen

Sockets /2

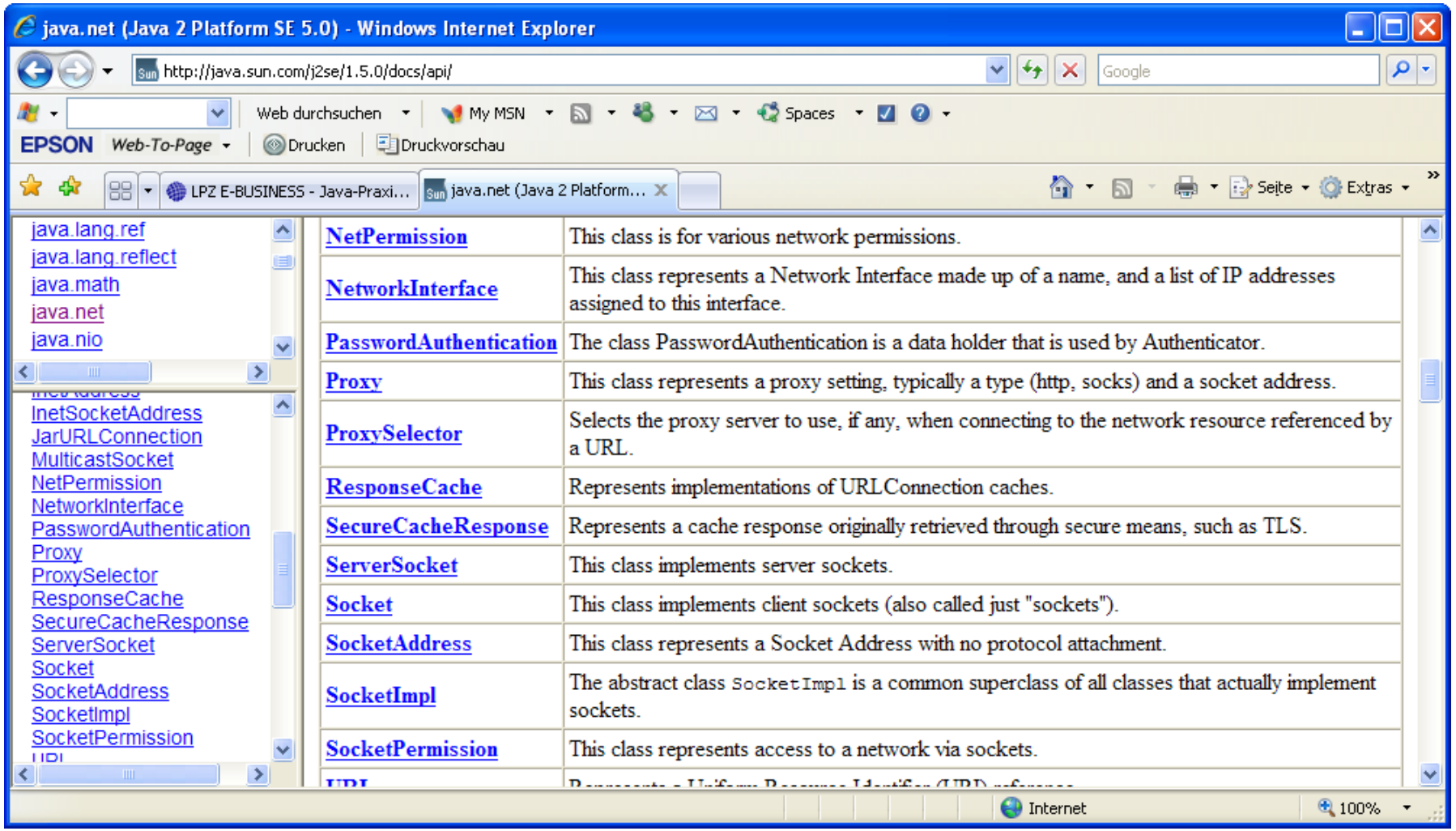
Vorteile von Sockets:

- Sockets gibt es auf allen gängigen Plattformen (Unix, Windows, IBM-Welt, …)
- Nutzung verschiedener Protokolle ist möglich (TCP/IP, UDP/IP)

Nachteile:

- Die Anwendung muss ein ***eigenes Protokoll*** implementieren
  - Jede Anwendung muss Datenpakete kodieren und dekodieren
  - Festlegen von Operationscodes und Datenstrukturen für Parameter ist mit spürbarem Aufwand verbunden (z.B. Protokoll-Automat)
- Socket-C-API auf jeder Plattform anders
- Socket-C-API ist fehlerträchtig
- Insbesondere Server-Programmierung ist aufwendig (Nebenläufigkeit, Ressourcen)

Socket API in Java



Java Sockets

Möglichkeiten der Kommunikation:

1. Verbindungsorientiert — Protokoll: TCP/IP
2. Paketorientiert, Verbindungslos — Protokoll: UDP/IP

Für die Kommunikation über ein Netzwerk wird das Java-Package
`java.net` benutzt

→ `import java.net.*;`

Wichtigste Klassen:

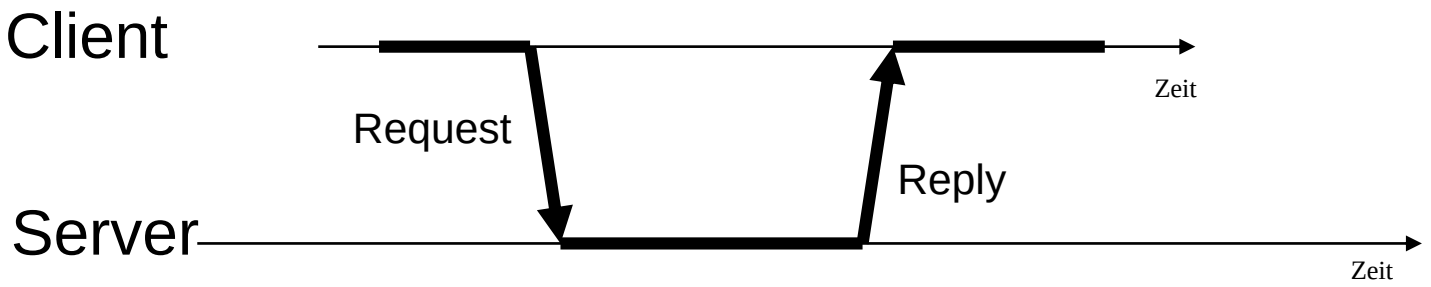
- `InetAddress`
- `Socket, ServerSocket` // Verbindungsorientiert
- `DatagramSocket, MulticastSocket` // Paketorientiert

TCP – Sockets

und die
Client/Server Architektur

Client / Server – Architektur (mit verbindungsorientierter Kommunikation)

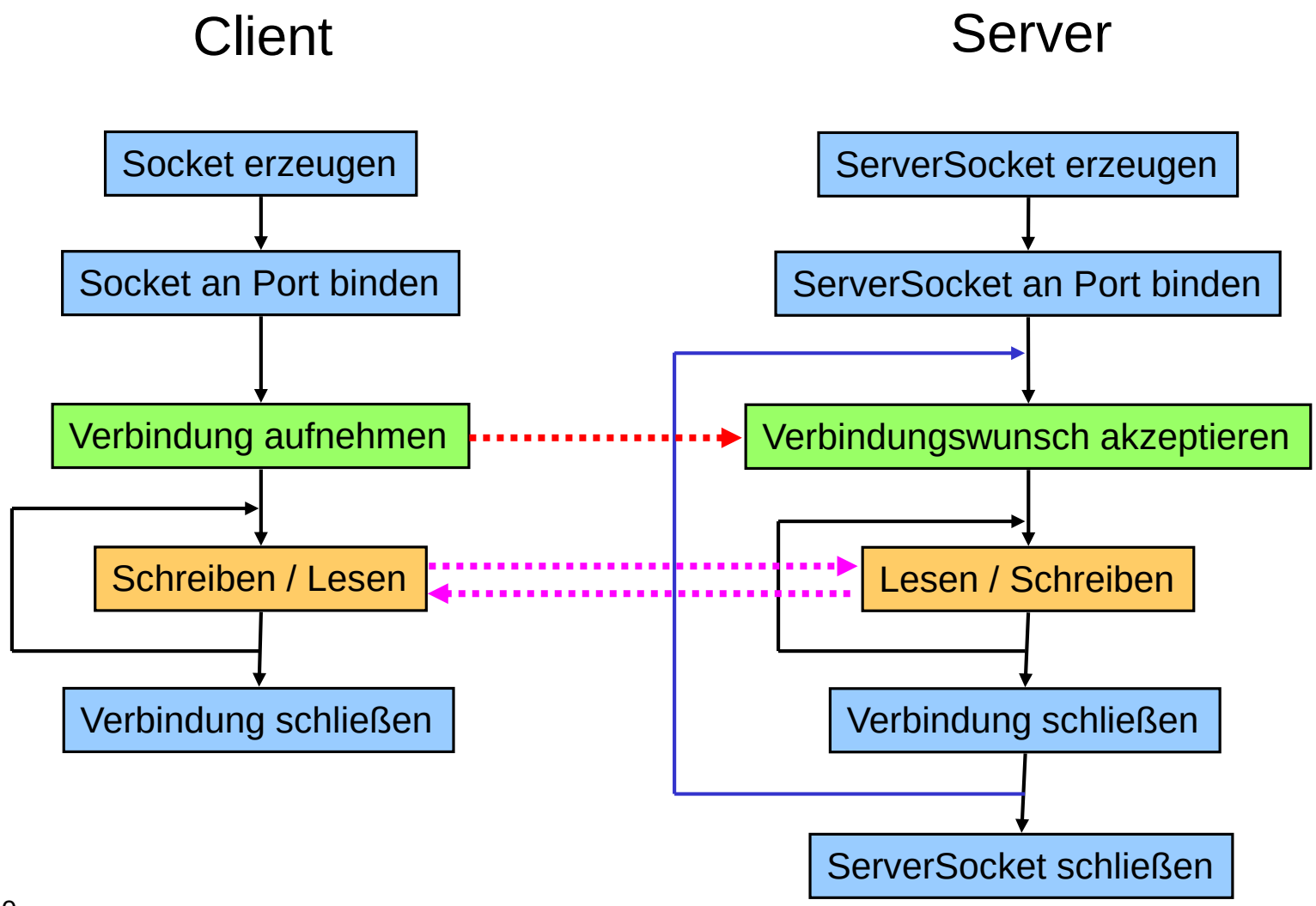
- Clients = aktiver Teil
  - Senden Anfragen
  - Verteilt räumlich / im Netzwerk
  - Typischerweise ein Client pro Benutzer, Typisch = „Fat“-Client
- Server = passiver Teil
  - Empfängt und verarbeitet Anfragen vieler Clients
  - Häufig zentral, etwa im RZ
  - Typischerweise einer oder wenige, Typisch = Datenbankserver
  - Typischerweise Multithreaded
- Beispiel: WebServer (z.B. Apache) + Browser-Client (z.B. Firefox)



Verbindungsorientierte Kommunikation – TCP

- Zwei Prozesse werden über ein Netzwerk miteinander verbunden
- Als Endpunkte werden Sockets auf jeder Seite benutzt
- Daten werden uninterpretiert ***binär*** ausgetauscht (= Byte-Strom)
- ***Keine Symmetrie***!
  - eine Seite explizit „Server“
  - die andere Seite „Client“
- Verbindung ist ***bidirektional***:
  Beide Seiten können schreiben und lesen
- ***Achtung: Beide Seiten müssen sich an das vom Programmierer/ Architekten vorgesehene Protokoll halten und auf Fehler reagieren***
  (Wer liest / schreibt wann was? Wie erkennt man einen Fehler?)

Verbindungsorientierte Kommunikation /2

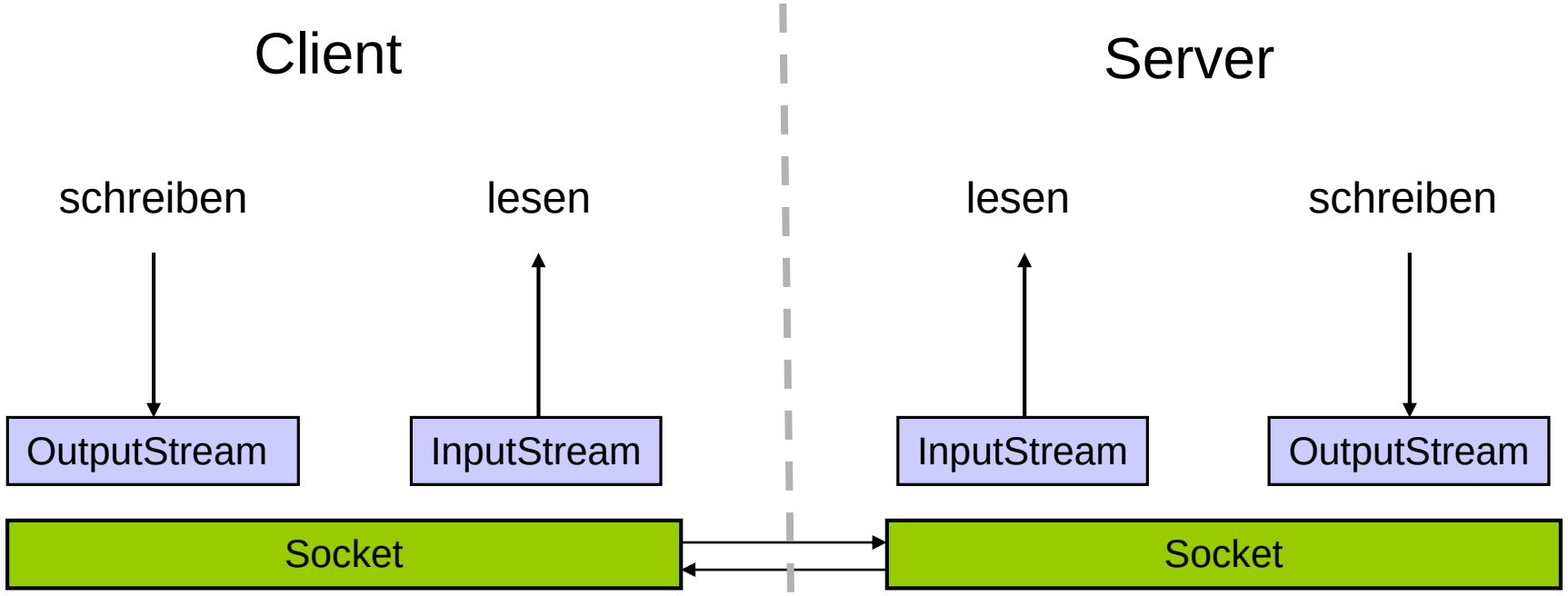


ServerSocket

Die Klasse **ServerSocket** ermöglicht die Verbindungsannahme im Server:

```
// Als Port wird 10013 angenommen, hier
// kann das bind() entfallen
ServerSocket ss= new ServerSocket(10013);

// Erzeugen eines Sockets, der für die
// Kommunikation verwendet werden kann
// Der Aufruf blockiert, bis ein Client
// eine Verbindung zu diesem Socket aufbaut
Socket s = ss.accept();

// Verbindungswunsch ist jetzt akzeptiert
// Der Server kann über s schreiben und lesen
...
```

Schreiben und Lesen über Sockets



Streams kann man dazu benutzen, Daten über Sockets auszutauschen:

```
Socket socket = …

InputStream in = socket.getInputStream();
OutputStream out = socket.getOutputStream();
```

(Client) **Socket**

```
// Erzeugen des Sockets
Socket s= new Socket("localhost", 10013);

// Ergebnis des folgenden Aufrufs ist z.B. 3152
System.out.println("Local Port="+s.getLocalPort());

// Erzeugen des Eingabe- bzw. Ausgabestroms
BufferedReader in= new BufferedReader(
    new InputStreamReader(s.getInputStream());
String message = in.readLine(); // Zeile Lesen

PrintWriter out = new PrintWriter(
    new OutputStreamWriter(s.getOutputStream()));
out.println(message); // Zeile Schreiben
```

Sockets und Ports

- Port (Anschluss) = Ressource des Betriebssystems
- Beispiel: Port 8080 oder Port 80 (HTTP)
- Socket wird an Port gebunden, wenn der Port noch nicht benutzt wird (Sonst: **`BindException`**)
- Port des Servers ist festgelegt (darauf „horcht“ er)

```
ServerSocket ss= new ServerSocket(10013); // am Server
Socket s= new Socket("localhost", 10013); // am Client
```

- Client verwendet irgendeinen Port

```
Socket s= new Socket("localhost", 10013); // am Client
 // Ergebnis des folgenden Aufrufs ist z.B. 3152
System.out.println("Local Port="+s.getLocalPort());
```

Bekannte Ports

- Port Nummern
  - Typischerweise belegt: Port 0 .. 1023
  - Verwendbar: Port 1024 .. 65535
- Beispiele (Server-Dienste)
  - 7 Echo service
  - 21 FTP
  - 22 SSH Remote Login Protocol
  - 23 telnet Interactive Session
  - 25 smtp Simple Mail Transfer
  - 80 HTTP World Wide Web

Vgl. (http://www.iana.org/assignments/port-numbers)

Anzeigen der verwendeten Sockets (Windows)

Programm: **`netstat`** (auf der Konsole)

Optionen:

- **`-a`** alle Verbindungen anzeigen
- **`-n`** Adressen und Portnummern numerisch
- **`-o`** Prozesskennung (PID) = Programm, dass die Sockets verwendet

Über Taskmanager Programm zur PID finden

Beispiel: **Echo-Server**

```
try (ServerSocket serversocket = new ServerSocket(10014)) {

   while (true) {
      try (Socket socket = serversocket.accept()) {
         handleClient(socket);
      } catch (IOException ex) {
         System.out.println("[Error] " + ex.getMessage());
         break;
      }
   }
} catch (IOException ex) {
   ex.printStackTrace(System.err);
}
```

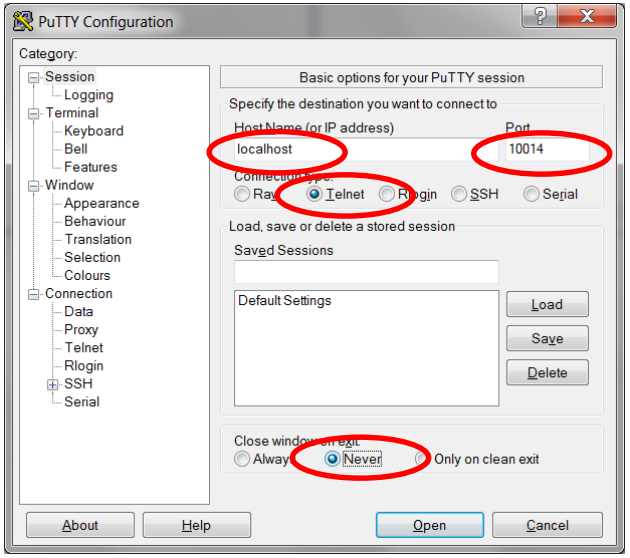
Beispiel: **Echo-Server**

```
private static void handleClient(Socket socket) {
   try (InputStream in = socket.getInputStream();
        OutputStream out = socket.getOutputStream()) {

        int zeichen = 0;
        while ((zeichen = in.read()) != EOF) {
            out.write((byte)zeichen);
            out.flush();
        }
    } catch (IOException ex) {
        ex.printStackTrace(System.err);
    }
}
```

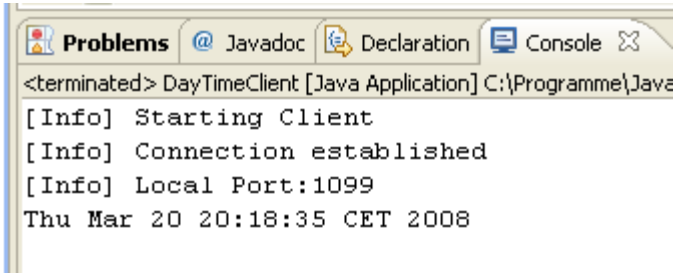
Beispiel: Client, der eine Zeile liest

```
public class DayTimeClient
{
    public static void main(String[] args)
    {
        BufferedReader in;     // Zum Einlesen vom Server
        Socket server;         // Verbindung zum Server

        try
        {
            // Aufbau der Verbindung
                server =
                   new Socket(InetAddress.getLocalHost(),10014);

                // Anlegen von Ein- und Ausgabestream
                in = new BufferedReader(
                   new InputStreamReader(server.getInputStream()));
                String text = in.readLine();
                System.out.println(text);
        }
        catch (Exception e) { System.err.println(e);}
    }
}
```

Ausgabe des Beispiels



Telnet als Client



DayTimeClient

Server-Design: Multi - Threaded

1.) Iterativ (single-threaded):

Zu jeder Zeit kann nur eine Anfrage am Server bearbeitet werden

- Es werden nur geringe Socket-Ressourcen benötigt
- Geeignet für kurze Anfragen, da Server blockiert

```
while(true) {
      Socket t = ss.accept();
      // Verarbeite nun die Client-Anfrage über Socket t
      ...
 }
```

2.) Parallel (multi-threaded):

Beliebig viele Anfragen können „gleichzeitig“ bearbeitet werden

- N+1 Sockets belegt: Einer für „listen“ und je einer pro Client-Anfrage
- Geeignet für länger andauernde Anfragen, z.B. Datenbankzugriff etc.

```
while(true){
      Socket t = ss.accept();
      createThread(t);  // Client bekommt eigenen Thread
                        // accept für nächsten Client sofort möglich
 }
```

→ ***Der erste Client merkt keinen Unterschied***

Multi-Threaded Server /1

```
public class MultiThreadedServer
{
  private Executor exe = …;
  public void start()
  {
    try
    {
        ServerSocket s = new ServerSocket(10014);
        while (true)
        {
            Socket t = s.accept();
            Application a = new Application(t);
            exe.execute(a);
        }
        s.close();
    }
    catch (java.io.IOException e) {... }
  }
}
```

Multi-Threaded Server /2

```
public class Application implements Runnable {

  private Socket t = null;

  public Application(Socket t) {
      this.t = t;
  }

  public void run() {
      OutputStream os = null; InputStream is = null;
      try
      {
          os = t.getOutputStream();
          is = t.getInputStream();

          ... // Lies Parameter, Op-Code
          ... // eigentliche Arbeit
          os.write( ... );  // Schreibe Ergebnis auf Stream
      }
      catch (IOException e) {}
      finally { try {is.close(); os.close(); t.close();} catch … }
  }
}
```

Diskussion Multithreaded Server

- Lesen ist bei „normalen“ Sockets immer blockierend

```
Socket t = ss.accept(); // Blockiert
t.getInputStream().read(); // Blockiert
t.getOutputStream().write(…); // Blockiert
```

- Lösung bei Multithreaded Servern: Pro Client ein Thread (aus einem Thread Pool)
  - Nachteil: Client bindet den Thread aus dem Pool
  - Poolgröße bestimmt damit die maximale Zahl an Clients
  - Threads werden nicht effizient genutzt

- Alternative: Nichtblockierendes Lesen mit java.nio
  - Klassen: ServerSocketChannel, SocketChannel und Selector
  - Selector „meldet“ wenn sich Client angemeldet hat, bzw. wenn ein Client Daten sendet

UDP – Sockets

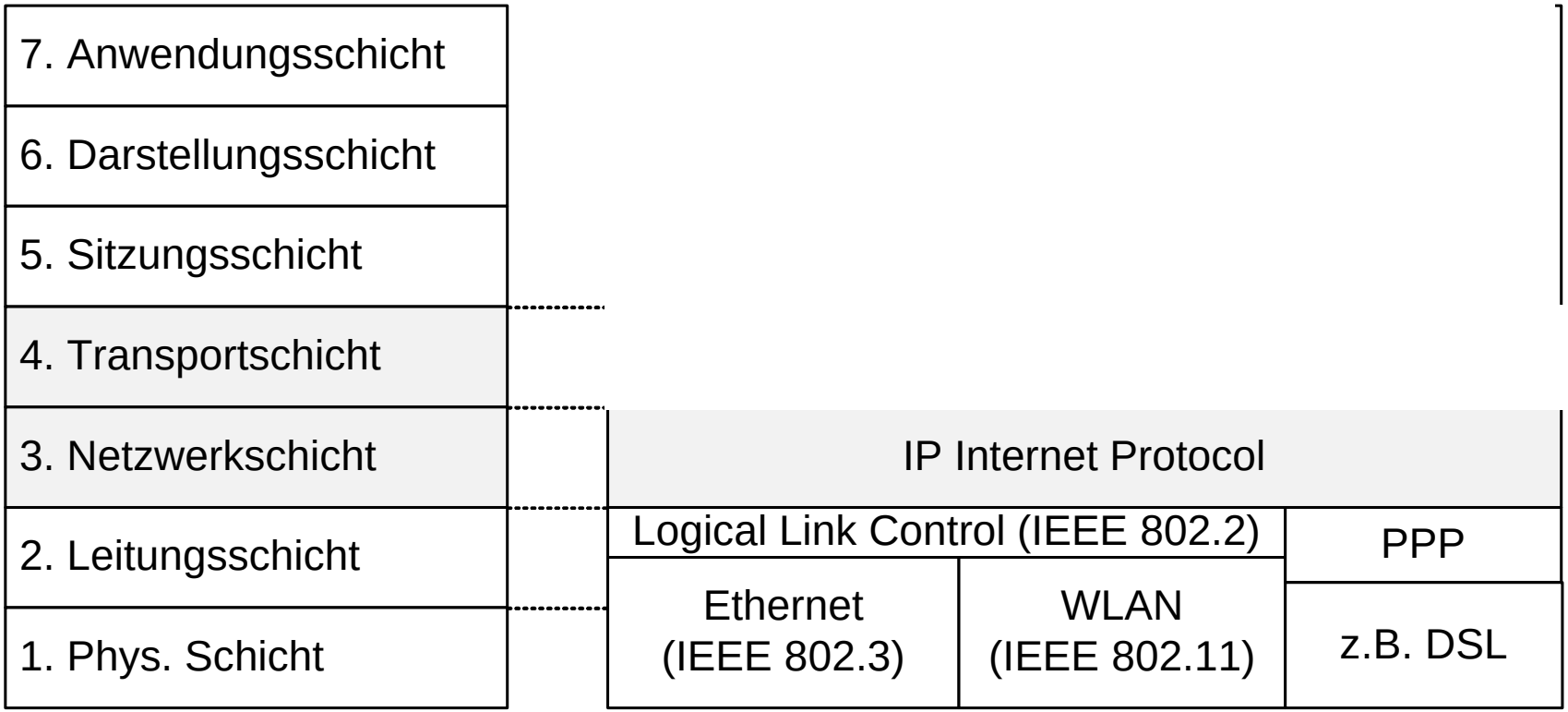
und die

Peer-To-Peer Architektur

Das behandeln wir später ausführlich!

ISO / OSI – Schichtenmodell

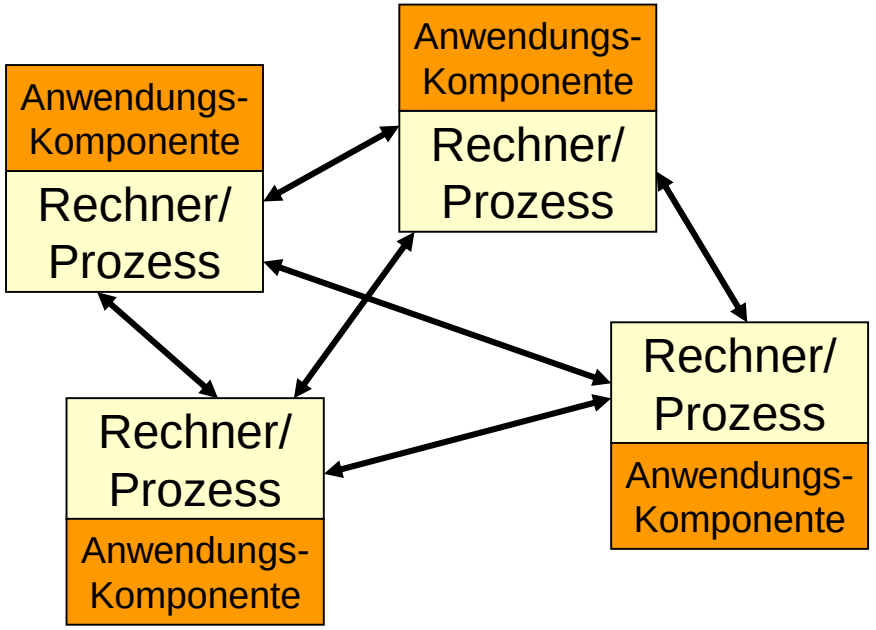
(vgl. Vorlesung Rechnernetze)



Peer-to-Peer – Architekturen

(Hier mit paketorientierter / verbindungsloser Kommunikation)

- ***Gleichberechtigte Prozesse*** (Peers) interagieren
- Prozess kann sowohl als Client- als auch als Serverprozess sein (aktiv und passiv)
- Ziel: Unabhängigkeit von einem zentralen Server
- Häufig: Eigenes „logisches“ Netzwerk über dem Internet
- Beispiele:
  - FileSharing (Napster, eDonkey, Gnutella, BitTorrent)
  - Skype, Insatant Messaging
  - Ad-Hoc-Netze
  - ***Bitcoin und andere Block-Chains***



Paketorientierte Kommunikation /1

- ***Keine Verbindung*** zwischen Prozessen!
- Client verschickt ***Datenpakete*** („datagrams“)
  Die Empfängeradresse ist Bestandteil jedes Pakets.
  Keine Garantien bzgl. der Reihenfolge!
- ***Symmetrische*** Kommunikation:
  Kein Unterschied zwischen Client und Server!
- Bidirektionale Kommunikation:
  - Beide Seiten können schreiben und lesen
  - Beide Seiten müssen sich an das vorgesehene Protokoll halten (wer liest / schreibt wann was ?)

Paketorientierte Kommunikation /2

Die Klasse **DatagramSocket** implementiert einen Socket für die verbindungslose Kommunikation über UDP:

```
// Erzeuge einen Socket mit der angegebenen Portnummer
DatagramSocket d= new DatagramSocket(8888);

// Erzeuge ein Datagram-Paket
// p wird die Empfängeradresse und die Nutzdaten enthalten
DatagramPacket p= new DatagramPacket(new byte[20], 20,
   InetAddress.getLocalhost(), 10014)

// Schicke das Datagram-Paket weg
d.send(p);
-------------------------------------------------
// Empfange ein Datagram-Paket an diesen Socket
// Die Methode blockiert, bis ein Paket ankommt
// Nach Ausführung des Aufrufs ist p mit Daten gefüllt.
d.receive(p);
```

Paketorientierte Kommunikation /3

Die Klasse `DatagramPacket` implementiert ein Datenpaket für die ***verbindungslose*** Kommunikation mittels DatagramSockets:

- Konstruktor `DatagramPacket(byte[] buf, int length, InetAdress adress, int port)`:
  Erzeugt ein Paket zum ***Senden*** an die angegebene Adresse
- Konstruktor `DatagramPacket(byte[] buf, int length)`:
  Erzeugt ein Paket zum ***Empfangen*** mit der angegebenen Länge
- `byte[] getData(), int getLength()`:
  - Liefert die empfangenen Daten bzw. deren Länge zurück
- `void setData(), void setLength(int l)`:
  - Setzt die zu sendenden Daten bzw. deren Länge

Beachte:
Pakete, die nacheinander an die gleiche Adresse gesendet werden, kommen nicht notwendigerweise in dieser Reihenfolge beim Empfänger an!

DateTimeServerUDP

```
DatagramSocket ds = new DatagramSocket(10014);

DatagramPacket packet = new DatagramPacket(new byte[20], 20);

while (true) {
    ds.receive(packet);
    packet.setData(getTime().getBytes());
    ds.send(packet);
}

ds.close();


// Hier die verwendeten Methoden
    private static final DateFormat DATEFORMAT =
        DateFormat.getDateInstance();
    private static String getTime() {
        return DATEFORMAT.format(new Date());
    }
```

DateTimeClientUDP

```
DatagramSocket ds = new DatagramSocket();

DatagramPacket packet =
   new DatagramPacket(new byte[20], 20,
            InetAddress.getLocalHost(), 10014);
ds.send(packet);

ds.receive(packet);

System.out.println("[Info] Empfangen:" +
    new String(packet.getData()));

ds.close();
```

Fehlerbehandlung

Fehlerbehandlung

- Mögliche Probleme sind:
  - Client
    - stürzt ab (Server muss das merken)
    - meldet sich regulär ab (Server muss das merken)
    - hat falschen Server konfiguriert, hat kein Passwort o.Ä.
  - Server
    - stürzt ab (Clients müssen informiert werden)
    - fährt herunter (Clients müssen informiert werden)
    - nicht (mehr) erreichbar, Verbindung bricht zusammen
    - ist überlastet (-> Graceful Degradation)
- Wichtig: Bauen Sie Mechanismen ein, um
  - ***Fehler erkennen*** zu können
  - ***Auf Fehler robust zu reagieren***

Fehlerbehandlung: ***Immer*** Timeout setzen

- Verbindung bricht ab oder Server ist nicht mehr erreichbar, Server zu langsam
  -> Timeout am Client:

```
Socket s = new Socket(InetAddress.getLocalHost(), 10013);
s.setSoTimeout(5000); // SocketTimeoutException
```

- Wenn Timeout, dann SocketTimeoutException, sonst wartet der Client eventuell ewig auf eine Reaktion des Servers
- Reaktion auf Timeout-Exception
  - Automatisches Retry [vgl. „Sicherheitsfassade“ aus Prog. 3] (Vorsicht: das kann einen neustartenden Server überlasten!)
  - Manuelles Retry über Meldung an Benutzer
  - Fehler dem Nutzer melden und Client beenden

Fehlerbehandlung - Verbindungsabbruch

- Jeder Zugriff auf einen Stream kann fehlschlagen, falls Client / Server die Verbindung beendet hat

```
Socket server = …;
BufferedReader in = new BufferedReader(
     new InputStreamReader(server.getInputStream()));
String text = in.readLine(); // Hier eventuell null
if (null == text) … // Reaktion auf Verbindungsabbruch
```

- Achtung: Implementieren Sie eine eigene Logik/Protokoll um,
  - Clients über den Shutdown des Servers zu informieren (TCP)
  - Clients beim Server abzumelden (TCP)
  - Peers, die offline gehen (UDP)
- Denkbar zusätzlich: ***Heart Beat*** (= regelmäßiges „Ping“)

Fehlerbehandlung bei Sockets

- Vorteil von java.net.: Gute Fehlerbehandlung möglich
- Exceptions beim Aufbau von Verbindungen:
  - `BindException // idR. Socket wird schon verwendet`
  - `ConnectException // idR. Server verweigert Zugriff`
  - `UnknownHostException // IP-Adresse/Host unbekannt`
  - `SocketException  // Fehler grundlegender Protokolle (TCP)`
  - `IOException // Vater der Exceptions in java.net`
- Reaktion auf diese Exceptions
  - Zentrales „catch“ am Client (vgl. Vorlesung zu Threads und Prg. 3)
  - Abbruch des Clients mit Meldung an den Benutzer (z.B. Konfigurationsfehler, Sicherheitsfehler, …)