Netzwerkprogrammierung

PROG 2: Einführung in die Programmierung für Wirtschaftsinformatiker

---

Steffen Helke, Marcus Mews
Technische Universität Berlin
Fachgebiet Softwaretechnik

17. Juni 2013

Übersicht

- Grundlagen Netzwerke
- Client-Server-Konzept
- Sockets
- UDP/TCP

Teil V der Vorlesung PROG 2

Netzwerkprogrammierung

Grundlagen

Quelle: *Inhalt & Gestaltung nach Vorlesungsfolien von Peter Pepper und Odej Kao, TU Berlin*
Methodische- und Praktische Grundlagen der Informatik 4 (MPGI 4), WS 2010/11 bzw. WS 2011/12

Adressierung von Rechnern

Idee

- Rechner in einem Rechnernetz benötigen eindeutige Namen, um Nachrichten an sie verschicken zu können

Umsetzung

- Festlegung einer eindeutigen IP-Adresse (*Internet Protocol*) für jeden Rechner
- Eindeutigkeit durch hierarchisch angelegten Adressraum

  ➜ jede untergeordnete Vergabestelle hat eigenen Adressraum

Beispiel

- Rechner *www.tu-berlin.de*
- IPv4-Adresse 130.149.7.201

Arten von IP-Adressen

Adressraumproblem

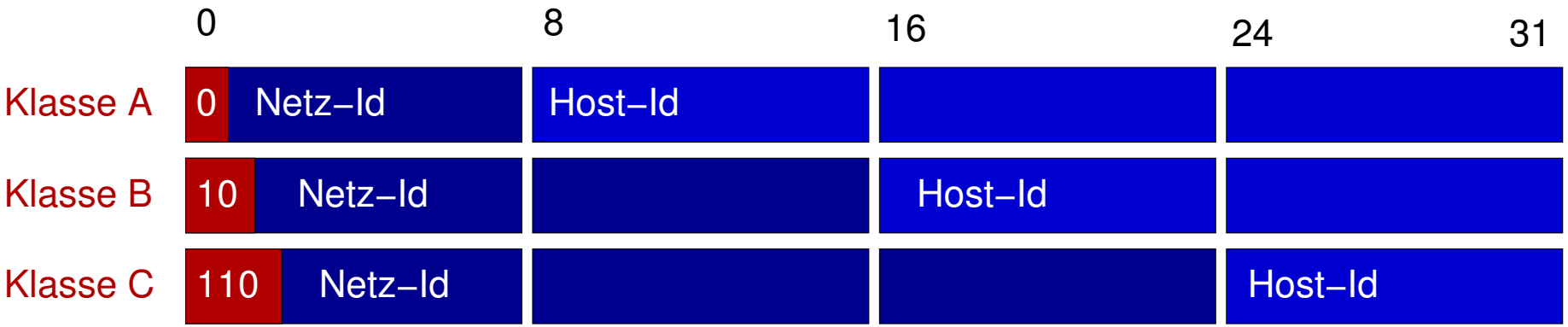
- nur *beschränkter Vorrat* an IP-Adressen
- älter IPv4-Standard auf 32 Bit beschränkt
- neuer IPv6-Standard auf 128 Bit erweitert

IPv4-Standard

- noch immer sehr verbreitet
- *Präfix:* identifiziert Netzwerk (*globale* Vergabe)
- *Suffix:* identifiziert Rechner (*lokale* Vergabe)
- Besonderheit: Größe des Präfix ist variabel

  ➜ Definition verschiedener IP-Adressklassen

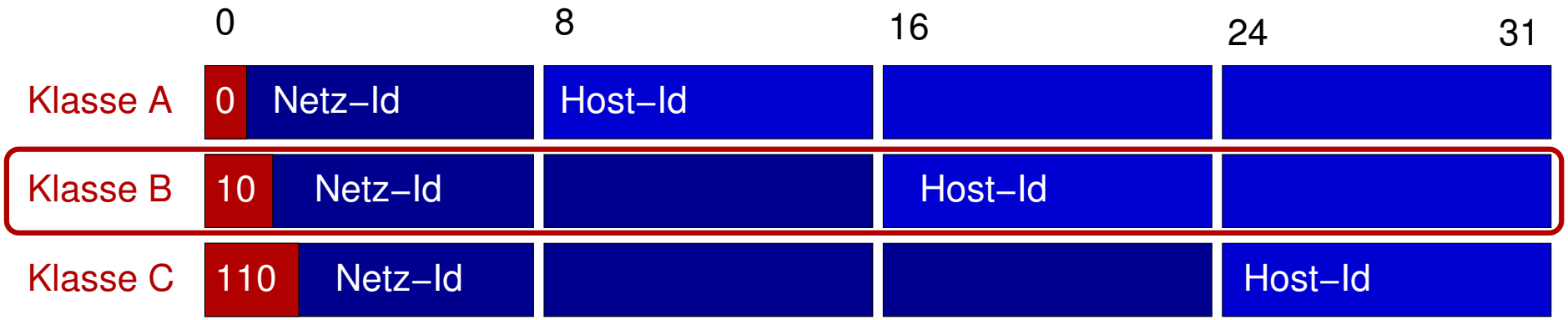
IP-Adressklassen im IPv4-Protokoll



Adressräume

- Klasse A: max. 126 Netz-Ids / max. 16.387.064 Host-Ids
- Klasse B: max. 16.256 Netz-Ids / max. 65.516 Host-Ids
- Klasse C: max. 2.064.512 Netz-Ids / max. 254 Host-Ids

Adressklasse eines TU-Rechners



Beispiel: TU-Rechner

- Adresse der Klasse B
- Aufteilung: 2-Bit Typ, 14 Bit Präfix, 16 Bit Suffix
- 130.149.7.201

  ➜ binär: 10 000010.10010101. 00000111.11001001

IP-Adressen in Java

- explizite Objekte zur Verwaltung von IP-Adressen
- Umwandlung von numerischen zu symbolischen Namen und umgekehrt mit DNS (*Domain Name Service*) möglich

```
import java.net.InetAddress;
import java.net.UnknownHostException;

public class Adressierung {
  public static void main (String[] args) {
    String name = "www.tu-berlin.de";
    // String name = "130.149.7.201";
     try {
      InetAddress addr = InetAddress.getByName(name);
      System.out.println(addr.getHostName());
      System.out.println(addr.getHostAddress());
     } catch (UnknownHostException e) { ... }
  }
}
```

Ports

Idee

- Kommunikationsendpunkte zur selektiven Annahme/Verarbeitung von Aufträgen
- Darstellung durch Integerzahl (16 Bit)

Aufteilung

- Port 0 bis 1023: Festlegung durch IANA (*Internet Assigned Numbers Authority*)
- Port 1024 bis 49151: Reservierung durch Hersteller
- Port 49152 bis 65535: keine Reservierung, variabler Einsatz

Vordefinierte Ports (Auswahl)

echo	7	Wiederholt jede Nachricht
ftp	21	Versenden und Empfangen von Dateien
ssh	22	Secure Shell
telnet	23	Interaktive Verbindung
smtp	25	E-Mail via smtp
time	37	Anzahl der Sekunden seit 1.1.1900
whois	43	Namensdienst
DNS	53	Auflösung von Domainnamen in IP-Adressen
finger	79	Informationen über aktive Benutzer
www	80	Web-Server
pop3	110	E-Mail via pop3
IMAP	143	Client-Zugriff für E-Mail-Server
nntp	119	News
snmp	161	Netzwerkmanagement
HTTPS	443	sicherer Webserver
rmi	1099	Entfernte Aufrufe
RDP	3389	Windows Remotedesktopzugriff, Terminal Services

Protokolle in der Datenkommunikation

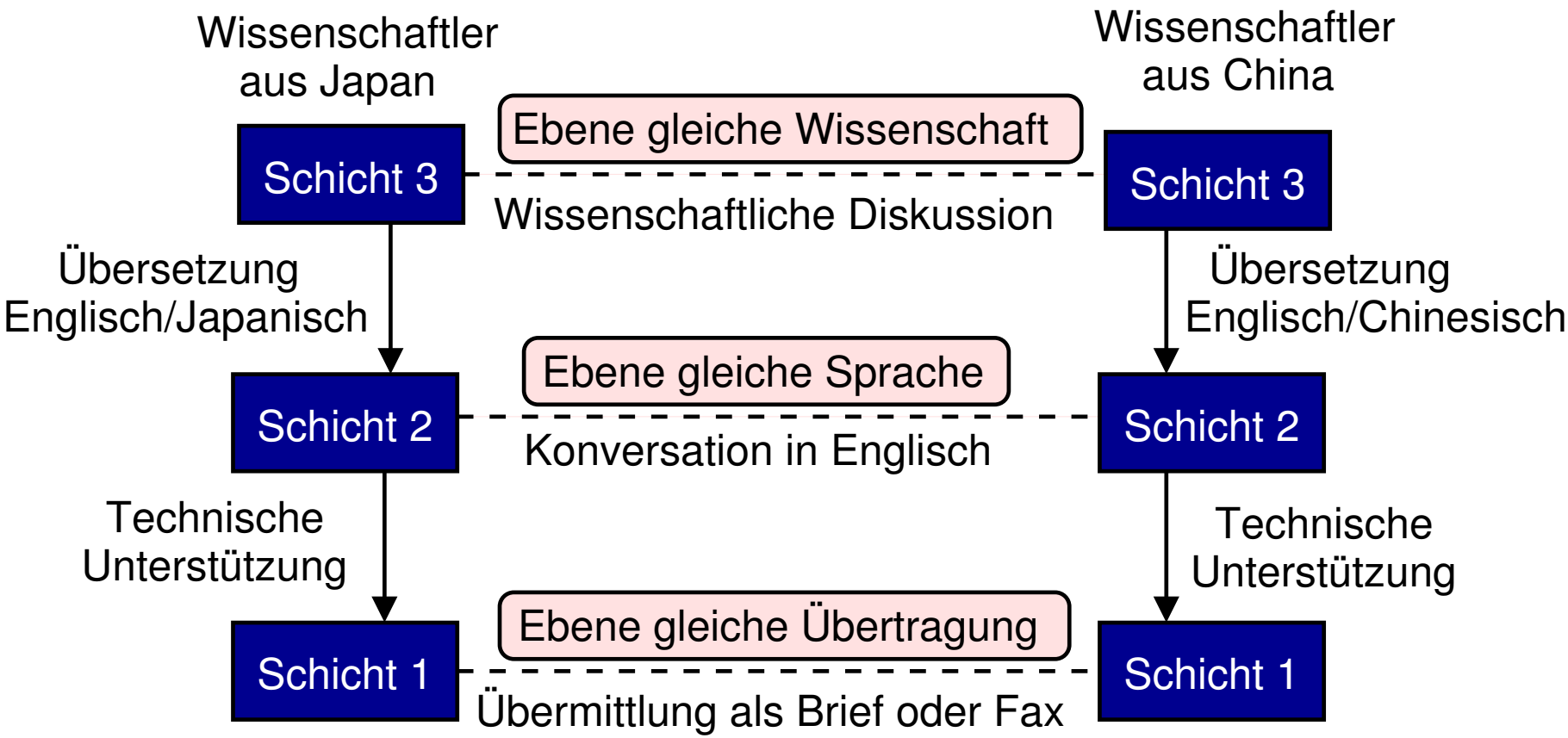
Idee

- Gliederung der Kommunikation in hierarchische Schichten
- jeder Schicht sind Kommunikationsfunktionen zugeordnet
- Kommunikationsfunktionen oberer Schichten nutzen zur Umsetzung Kommunikationsfunktionen unterer Schichten

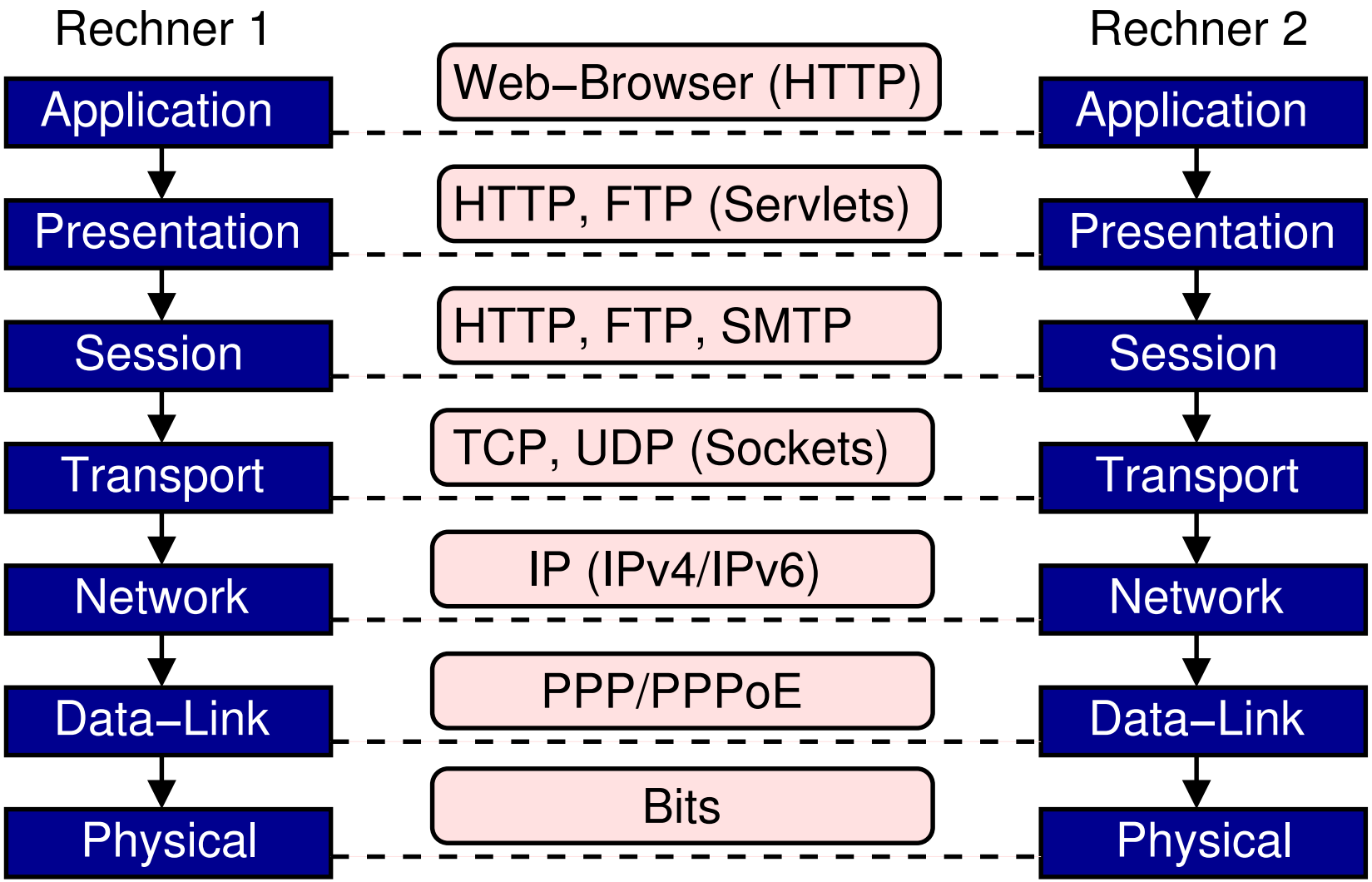
Umsetzung

- 7 Schichten im OSI-Modell *Open Systems Interconnection* (ISO-Standard)
- 4 Schichten im TSP/IP-Modell (De-facto-Standard)

Veranschaulichung OSI-Modell



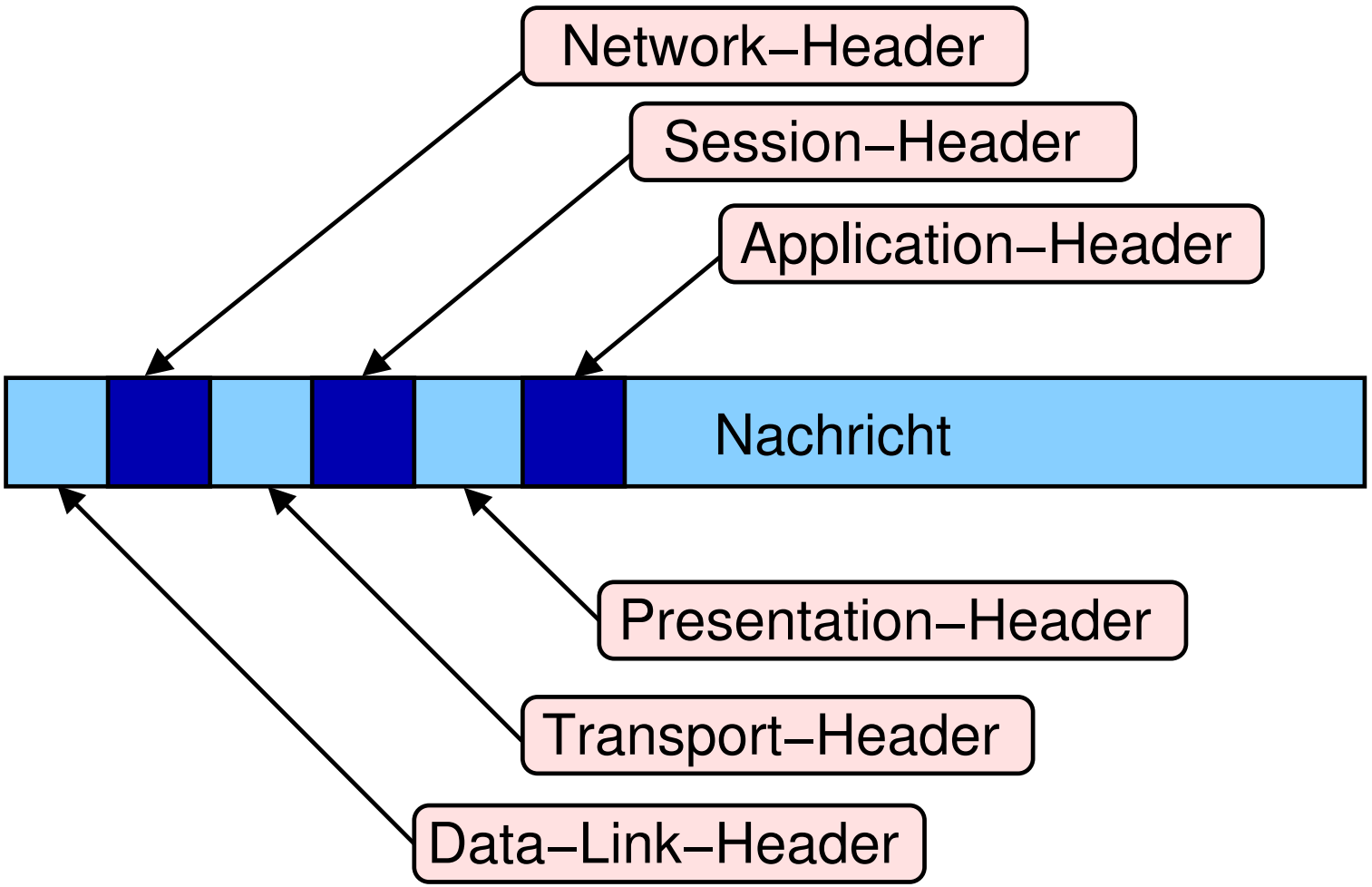
Schichten im OSI-Modell



Bedeutung einzelner OSI-Schichten

1. **Application:** Anwendungsschicht, dient der Kommunikation zwischen zwei Anwendungen
2. **Presentation:** Darstellungsschicht, dient der semantischen Interpretation von Bits
3. **Session:** Sitzungsschicht, dient der Kontrolle, welcher Teilnehmer gerade sendet
4. **Transport:** Übertragungsschicht, dient der verlustfreier Übertragung von Nachrichten
5. **Network:** Vermittlungsschicht, dient der Auswahl eines Weges durch das Netzwerk
6. **Data-Link:** Sicherungsschicht, dient der Gruppierung der Bits in Einheiten und kontrolliert deren Übertragung
7. **Physical:** Bitübertragungsschicht, dient der Bit-Übertragung

OSI-Header-Struktur in einer Nachricht



Teil V der Vorlesung PROG 2

Netzwerkprogrammierung

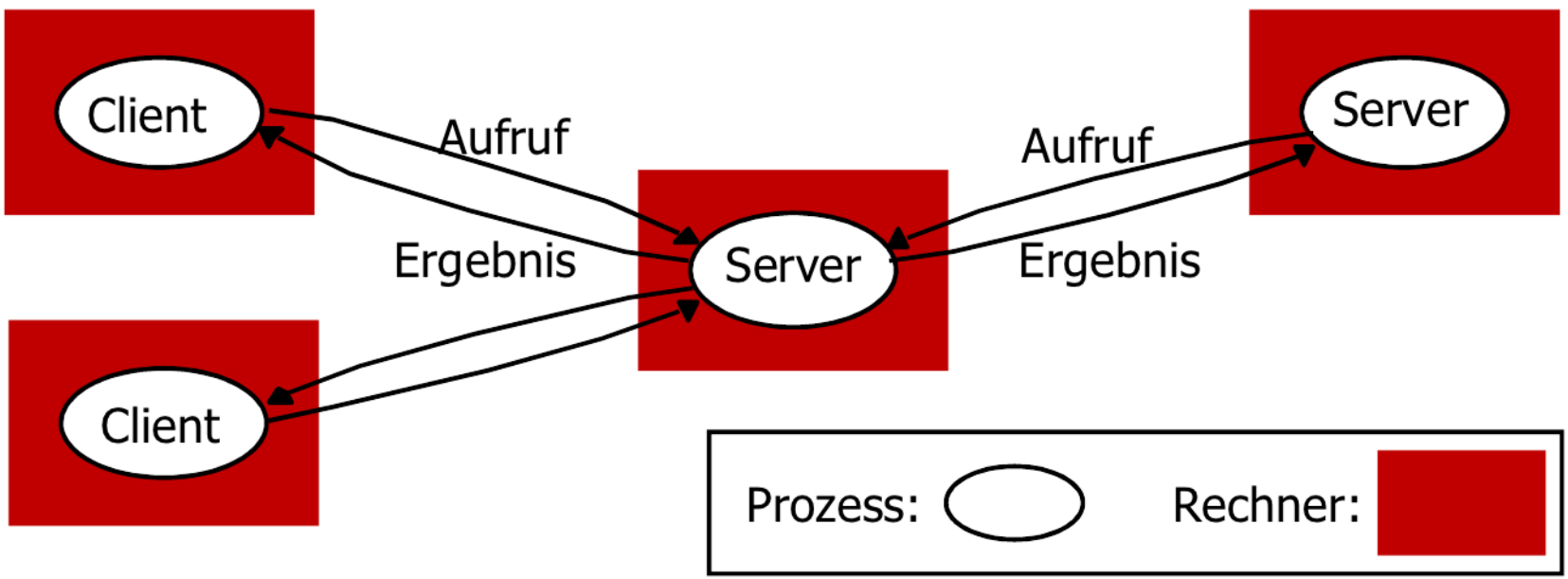
Client-Server-Konzept

Quelle: *Inhalt & Gestaltung nach Vorlesungsfolien von Peter Pepper und Odej Kao, TU Berlin*
Methodische- und Praktische Grundlagen der Informatik 4 (MPGI 4), WS 2010/11 bzw. WS 2011/12

Client-Server-Architektur

Grundprinzip

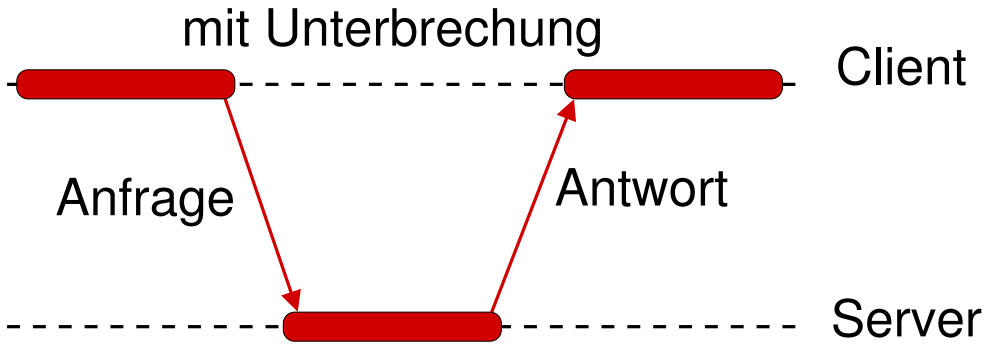
- mehrere Clients fordern bei einem Server Dienste an
- Server können Anfragen an andere Server weiterleiten
- auf Client und Server laufen unabhängige Prozesse



Kommunikationsarten

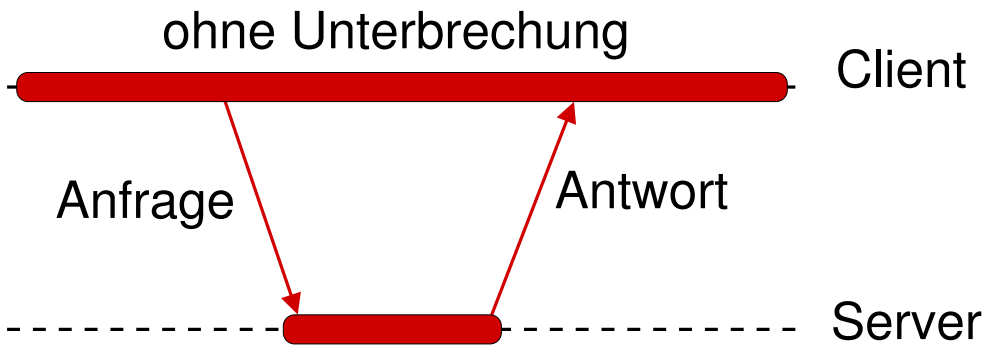
Synchrone Kommunikation

➜ z.B. Remote Method Invocation (RMI)



Asynchrone Kommunikation

➜ z.B. Email-Kommunikation (SMTP)



Teil V der Vorlesung PROG 2

Netzwerkprogrammierung

Socket-Programmierung

Quelle: *Inhalt & Gestaltung nach Vorlesungsfolien von Peter Pepper und Odej Kao, TU Berlin*
Methodische- und Praktische Grundlagen der Informatik 4 (MPGI 4), WS 2010/11 bzw. WS 2011/12

Netzwerkprogrammierung

Zielsetzung

- Kommunikation zwischen Prozessen über ein Netzwerk

Varianten

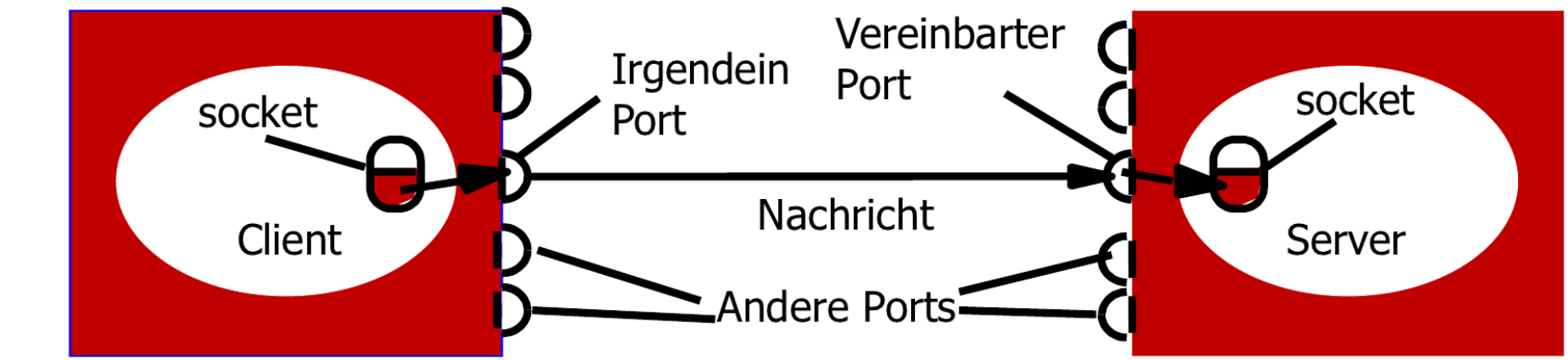
- TCP: *Transmission Control Protocol*
- UDP: *User Datagram Protocol*
- RMI: *Remote Method Invocation*
  ➜ nächste Vorlesung

Mittel

- *Socket-Abstraktion* ist bei TCP/UDP De-facto-Standard

Kommunikation über Sockets

- Sockets bilden *Kommunikationsendpunkte* in einem TCP/UDP-Protokollablauf
- Ursprung in BSD-Unix (*Berkeley Software Distribution*), aber auch in anderen Betriebssystemen unterstützt
- Sockets bestehen jeweils aus IP-Adresse und lokalem Port



Bestandteile einer Socket-Kommunikation

1. Protokoll (TCP, UDP oder RAW-IP)
2. lokaler Rechner (host)
3. lokaler Port
4. entfernter Rechner
5. entfernter Port

Socket-Aufteilung

- *Socket 1:* Protokoll, lokaler Rechner und Port
- *Socket 2:* Protokoll, entfernter Rechner und Port

Randbedingungen

- TCP-Server können mehrere Sockets auf einem Port haben
- Prozesse können mehrere Socket-Objekte haben, ein Socket-Objekt gehört aber nur zu einem Prozess

Socket-Typen

Datagram-Sockets

- *verbindungsloses Protokoll*, d.h. jede Nachricht enthält alle für Transport nötigen Informationen
- *unzuverlässig*, keine Wiederholung im Fehlerfall, keine Garantie für Erhalt der Reihenfolge

➜ implementiert *UDP-Protokoll*

Stream-Sockets

- *verbindungsorientiertes Protokoll*, d.h. Aufbau einer virtuellen Verbindung zwischen zwei Prozessen
- *viel zuverlässiger* als Datagram-Sockets, Garantie für Erhalt der Reihenfolge

➜ implementiert *TCP-Protokoll*

Welche Java-Klassen brauchen wir?

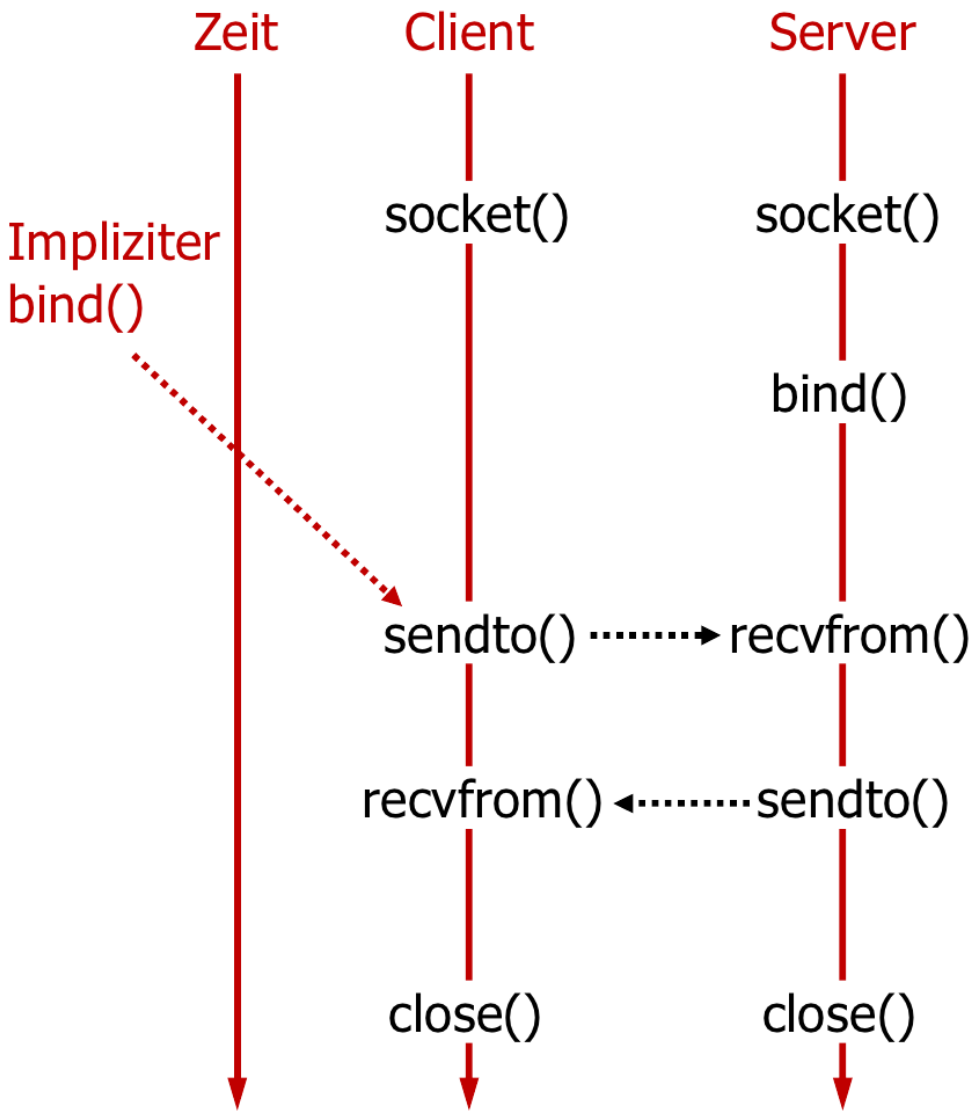
Einordnung

- *Stream-Sockets* können mit der Klasse *Socket* auf der Client-Seite und mit der Klasse *ServerSocket* auf der Server-Seite erzeugt werden
- *Datagram-Sockets* werden mit der Klasse *DatagramSocket* auf Client- und Serverseite erzeugt

Klasse	Protokoll	Richtung
Socket	TCP	ausgehend
ServerSocket	TCP	hereinkommend
DatagramSocket	UDP	ausgehend und hereinkommend

Datagram-Sockets für UDP

1. *Socket erzeugen:*
   Kommunikationsendpunkt und Protokoll festlegen
2. *Socket verbinden:*
   Server registriert Adresse, veranlasst Paketweiterleitung
3. *Datagrame senden:*
   Client definiert Server-Adresse und zu übertragene Daten
4. *Datagrame empfangen:*
   Server ermittelt Adresse des Senders und antwortet



Eigenschaften von Datagram-Sockets für UDP

Nachrichtengröße

- Empfängerprozess reserviert Byte-Array für Empfang der Nachricht
- falls Nachricht länger als Byte-Array, wird diese gekürzt
  ➜ Aufteilung längerer Nachrichten in Pakete nötig

Datenverlust

- bei Empfang durch Prüfsummentest erkennbar
- korrekte Reihenfolge kann verloren gehen

Latenzzeit

- *geringe Latenz* des Kommunikationskanals
- geeignet für Anwendungen, die geringe Paketverluste tolerieren können (z.B. VideoChat)

Java-Umsetzung für Datagram-Sockets

```
// Konstruktor
 public DatagramSocket()

// Konstruktor mit Port-Bindung an lokalem Host
// alle lokal verfügbaren Netzwerkinterfaces
 public DatagramSocket(in port)

// Konstruktor mit Port-Bindung von lokaler Adresse
 public DatagramSocket(in port, InetAdress laddr)

// Datagram-Socket-Objekt an lokale Adresse binden
 void bind(SocketAddress addr)

// Datagram-Socket-Objekt an entfernte Adresse binden
 void connect(InetAddress addr, int port)

// Datagram-Socket-Objekt schliessen
 void close()/disconnect()

// Ist Broadcasting erlaubt?
boolean getBroadcast()
```

Java-Umsetzung für Datagram-Pakete

Paketbestandteile

- Nachricht als Byte-Array und Nachrichtenlänge
- IP-Adresse und Port (zum Senden)

```
// Konstruktor zum Empfangen
 public DatagramPacket(byte[] buf, int length)

// Konstruktor zum Senden
 public DatagramPacket(byte[] buf, int length,
                       InetAddress a, int p)

// Konstruktor zum Senden mit Socket-Adresse
 public DatagramPacket(byte[] buf, int length,
                       SocketAddress sa)

// Abfragen/Setzen diverser Werte aus einem Datagram-Paket
 byte[] getData() / setData(byte[] buf)
 int getPort() / setPort(int p)
 ...
```

Kommunikation mit Paketen über Sockets

```
// Empfangen von Datagram-Paketen
 void receive(DatagramPacket p)

// Versenden von Datagram-Paketen
 void send(DatagramPacket p)

// Erlauben von Broadcasting
 void setBroadcast(boolean on)

// Puffergroesse lesen
 int getReceiveBufferSize()
 int getSendBufferSize()

// Puffergroesse setzen
 void setReceiveBufferSize(int size)
 void setSendBufferSize(int size)

// Abbruch, wenn nach Timeout bei receive()
// keine Datagramme eingetroffen
 void setSoTimeout(int timeout)
```

Beispiel: Datagram-Socket auf Client-Seite

```
import java.net.*;
import java.io.*;

public class Client_Echo {
 public static void main(String args[]) {
   DatagramSocket aSocket = null;
   String host = "localhost";
   String message = "UDP-Kommunikation";
   try {
     aSocket = new DatagramSocket();
     byte[] m = message.getBytes();
     InetAddress aHost = InetAddress.getByName(host);
     DatagramPacket request =
          new DatagramPacket(m, m.length, aHost, 6789);
     aSocket.send(request);
     byte[] buffer = new byte[1000];
     DatagramPacket reply =
          new DatagramPacket(buffer, buffer.length);
     aSocket.receive(reply); System.out.println( ... );
   } catch (SocketException e) { ... }
     catch (IOException e) { ... }
     finally { if(aSocket != null) aSocket.close();} }}
```

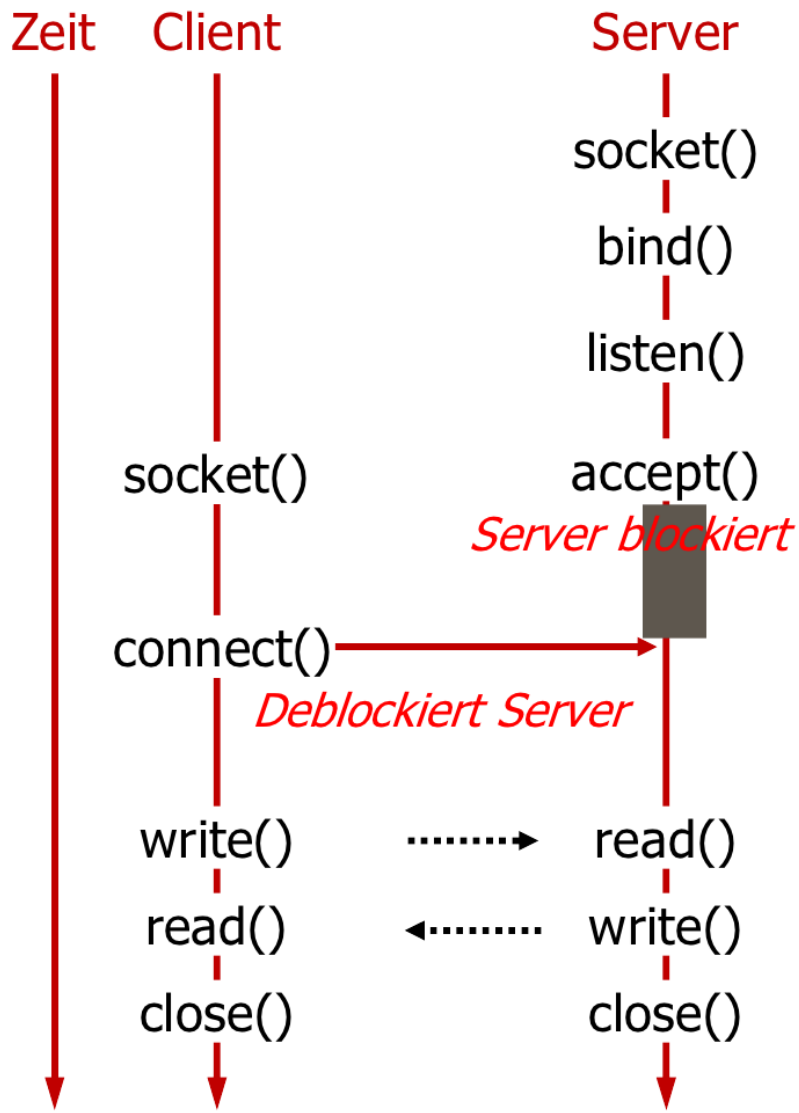
Beispiel: Datagram-Socket auf Server-Seite

```
import java.net.*;
import java.io.*;

public class Server_Echo {
 public static void main(String args[]) {
    DatagramSocket aSocket = null;
   try {
      aSocket = new DatagramSocket(6789);
     byte[] buffer = new byte[1000];
     while(true) {
        DatagramPacket request =
           new DatagramPacket(buffer,buffer.length);
        aSocket.receive(request);
        DatagramPacket reply =
           new DatagramPacket(request.getData(),
                              request.getLength(),
                              request.getAddress(),
                              request.getPort());}
        aSocket.send(reply);
   } catch (SocketException e) { ... }
     catch (IOException e) { ... }
     finally { if(aSocket != null) aSocket.close();} }}
```

Stream-Sockets für TCP

1. *Sockets erzeugen:* verbindungsorientierte Kommunikation
2. *Sockets verbinden und Horchen:* Server wartet mit *accept()* auf Aufforderung des Client
3. *Verbindungsaufbau* Aufforderung des Client mit *connect()*
4. *Unsymmetrisches Rendezvous:* Server muss *accept()* vor *connect()* des Clients ausführen



Eigenschaften von Stream-Sockets für TCP

Nachrichtengröße

- Anwendung legt fest, wie viele Daten geschrieben werden
- Stream-Socket-Klassen übernehmen die Aufteilung in Pakete

Datenverlust

- Empfangsbestätigung
- falls keine Bestätigung
  ➜ erneutes Senden nach Timeout

Duplizierte Nachrichten und Reihenfolge

- Datenpakete haben eindeutige Identifier
  ➜ *Duplikate oder falsche Reihenfolge werden aufgedeckt*
- Datenpakete brauchen keine Angaben über IP/Port

Java-Umsetzung von Stream-Sockets

Unterschiedliche Klassen für Client und Server

- Client: *Socket*
- Server: *ServerSocket*

```
// Konstruktor zur Erzeugung von Server-Sockets
// enthaelt implizite Warteschlange fuer 50
// wartende Verbindungen
 public ServerSocket(int port)

// Konstruktor zur Erzeugung von Server-Sockets
// mit backlog kann Warteschlange vergroessert werden
 public ServerSocket(int port, int backlog)

// Socket mit Port verbinden
 void bind(SocketAddress endpoint)

// Warten auf Connect-Anforderung vom Client
 Socket accept() throws IOException
```

Beispiel: Stream-Socket auf Client-Seite

```
import java.net.*;
import java.io.*;

public class TCPClient {
 public static void main(String args[]) {
   Socket s = null;
   String server = "localhost";
   String message = "TCP-Kommunikation";
   try {
     s = new Socket(server,7896);
     DataInputStream in =
          new DataInputStream(s.getInputStream());
     DataOutputStream out =
          new DataOutputStream(s.getOutputStream());
     out.writeUTF(message);
     String data = in.readUTF();
     System.out.println("Received: "+ data);
   } catch (UnknownHostException e) { ... }
     catch (IOException e) { ... }
     finally { if(s!= null) try {aSocket.close();}} ... }}
```

Beispiel: Stream-Socket auf Server-Seite

```
public class TCPServer {
 public static void main(String args[]) {
  try {
   ServerSocket listenSocket = new ServerSocket(7896);
   while(true) {
     Socket clientSocket = listenSocket.accept();
     Connection c = new Connection(clientSocket);
   }} catch (IOException e) { ... }}}
 class Connection extends Thread {
  DataInputStream in;
  DataOutputStream out;
  Socket clientSocket;
  public Connection (Socket aClientSocket) {
  try {
   clientSock = aClientSocket;
   in = new DataInputStream(clientSock.getInputStream());
   out = new DataOutputStream(clientSock.getOutputStream());
   this.start();
  } catch (IOException e) { ... }}
  public void run() { ... }}
```

Socket-Adressen mit InetSocketAddress

➜ Objekte der Klasse *InetSocketAddress* sind im Gegensatz zu „normalen" Socket-Objekten *serialisierbar* und unterstützen *Timeout*-Definitionen

```
// Socket-Adresse aus IP und Port
InetSocketAddress(InetAddress addr, int port)
// Socket-Adresse aus Hostname und Port
InetSocketAddress(String hostname, int port)

// Socket-Adresse ohne DNS-Test
static InetSocketAddress
          createUnresolved(String host, int port)
// Adresse aus Socket-Adresse ermitteln
InetAddress getAdress()
// Hostname aus Socket-Adresse ermitteln
String getHostName()
// Port aus Socket-Adresse ermitteln
int getPort()
// Adresse bereits getestet?
boolean isUnresolved()
```

Stream-Socket mit InetSocketAddress (Client)

```
import java.net.*;
import java.io.*;

public class TCPClient {
 public static void main(String args[]) {
   Socket s = null;
   String server = "localhost";
   String message = "TCP-Kommunikation";
   InetSocketAddress addr;
   try {
     addr = new InetSocketAddress(server,7896);
     s = new Socket();
     s.connect(addr);
     DataInputStream in =
          new DataInputStream(s.getInputStream());
     DataOutputStream out =
          new DataOutputStream(s.getOutputStream());
     out.writeUTF(message);
     String data = in.readUTF();
     System.out.println("Received: "+ data);
   } catch (...) { ... }
```