Schichtenmodell

- Komplexen Kommunikationsvorgänge werden in einzelne Schritte aufgegliedert und in mehreren Schichten übereinandergestapelt.
- Jede Schicht definiert bestimmte Aufgaben und Funktionen. Zur Lösung dieser Aufgaben existieren spezielle Verfahren und Protokolle.
- Jede Schicht verfügt jeweils über Schnittstellen zur darüber oder darunter liegenden Schicht.
- Die darunter liegenden Schichten sind „Übersetzer“ für darüber liegenden.
- Übertragungsweg, Protokolle und Anwendung genormt und genau spezifiziert.
- Referenzmodelle: OSI (Open System Interconnection model) und TCP/IP-Referenzmodell (DoD - Department of Defense)

Schichtenmodell

- Für das Internet und die Internetprotokollfamilie ist die Gliederung nach TCP/IP-Referenzmodell maßgebend.

OSI-Schicht	TCP/IP-Schicht	Beispiel
Anwendungen (7)	Anwendungen	HTTP, UDS, FTP, SMTP, POP, Telnet, DHCP, OPC UA
Darstellung (6)		
Sitzung (5)		SOCKS
Transport (4)	Transport	TCP, UDP, SCTP
Vermittlung (3)	Internet	IP (IPv4, IPv6), ICMP (über IP)
Sicherung (2)	Netzzugang	Ethernet, Token Bus, Token Ring, FDDI
Bitübertragung (1)		

Quelle: https://de.wikipedia.org/wiki/Internetprotokollfamilie#TCP/IP-Referenzmodell

TCP/IP Referenzmodell für Netzprotokolle

- Netzzugang (Link Layer)
  Techniken (mechanische und elektrische Mittel) der Datenübertragung von Punkt zu Punkt

- Internet (Internet Layer)
  Vermittlung von Datenpaketen über den gesamten Kommunikationsweg:
  - Routing (Wegsuche) zwischen den Netzknoten
  - eindeutige Adressierung durch IP (Internet Protocol)

- Transport (Transport Layer)
  Ende-zu-Ende-Kommunikation
  Adressierung wird durch „Kontaktpunkte“ (Portnummern) ergänzt
  Transportprotokolle
    UDP (User Datagram Protocol)
    TCP (Transmission Control Protocol)

- Anwendung (Application Layer)
  stellt alle Protokolle für die Anwendungen zur Verfügung, ist zuständig für Verbindung zu den unteren Schichten

Internet – Adressen

jede IP-Adresse ist weltweit eindeutig

- IPv4: 32 Bit, 4X8, (2 hoch 32 Adressen)
  Dezimalnotation
  0..255, d.d.d.d
  139.20.17.101

<244	einzelne Rechner (Unicast) – Adressen-Klassen A, B und C
224.. 239	D-Klasse: Multicast-Adressen, Gruppen-Adressen von 224.0.0.0 bis 239.255.255.255 (jede mit FF00::/8 beginnende Adr.)
>239	E-Klasse: reservierter Bereich

- IPv6: 128 Bit, 8X16, (2 hoch 128 Adressen) (seit 1998)
  Hexadezimalnotation, (0..ffff)
  2001:0db8:0000:0000:0000:ff00:0042:8329

- Namen (z.B. www.informatik.tu-freiberg.de) werden in IP-Adresse vom Domain Name System (DNS) umgesetzt

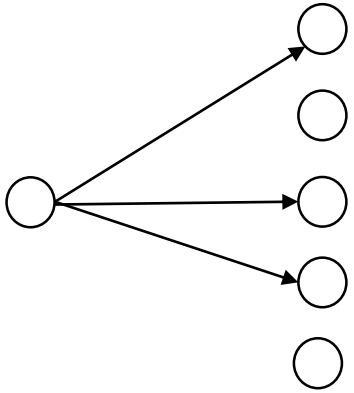
Kommunikationsformen

- Unicast

  Übertragung von Nachrichten zwischen einem Sender und einem einzigen Empfänger - genau ein Ziel

  Spezielle Adressen:

unspezifiziert	0.0.0.0	[::]
loopback (localhost)	127.0.0.1	[::1]



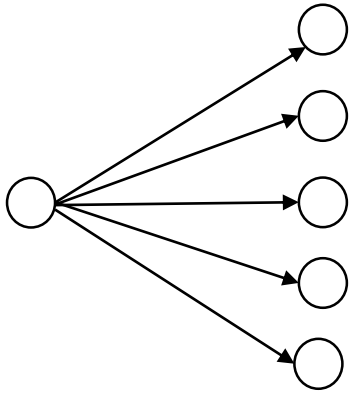
- Multicast

  eine Nachrichtenübertragung von einem Punkt zu einer Gruppe - mehrere Ziele

- Broadcast

  eine spezielle Form der Mehrpunktverbindung (alle Teilnehmer eines lokalen Netzwerks)

  eigenes lokales Netz: 255.255.255.255

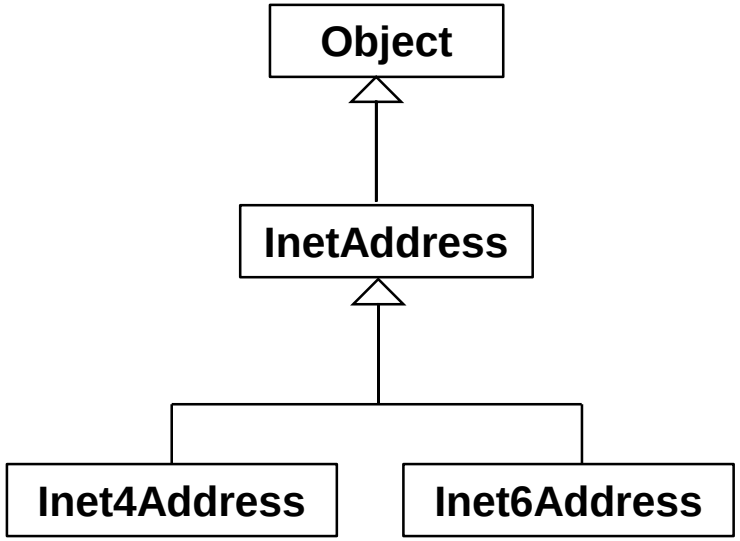


https://de.wikipedia.org/wiki/Multicast

Klasse

`InetAddress`

geeignet für beide IP-Adresstypen



Quelle: Prof. B.Steinbach, Vorlesung „Verteilte Software“

Klasse **InetAddress**

Anlegen des Objektes:

```
static InetAddress getLocalHost() throws UnknownHostException
static InetAddress getByName(String host) throws UnknownHostException
static InetAddress getByAddress(byte[] addr) throws UnknownHostException
```

```
InetAddress ia1 =
   InetAddress.getByName("pc1.informatik.tu-freiberg.de");
InetAddress ia2 = InetAddress.getLocalHost();
```

```
String getHostname()              //Rechnername
String getCanonicalHostName()     //qualifizierter Domainame
String getHostAddress()           //IP-Adresse
byte[] getAddress()               //IP-Adresse
```

Die Adresse allein reicht nicht zur Identifizierung einer Internet-Ressource (einer Webseite, Webservice, Email-Empfänger, Datei, …)!
Eindeutige Identifikation benötigt mehr Informationen, z.B. Art der Ressource ….

Uniform Resource Identifier (URI)

`scheme ":" authority-path [ "?" query ] [ "#" fragment ]`

- Schema oder Protokoll (Art der Ressource)
  z.B. https, ftp, mailto, tel, …)
- Anbieter oder Server (authority) inkl. Host (Name, IPv4, IPv6), Portnummer (Integer-Wert) und Pfad
- Abfrage – Query-String mit zusätzlichen Informationen (evtl.)
- Fragment (evtl.)

```
https://beispiel.de:8042/pfad/nochpfad?abfrage=text#stelle
tel:+49-111-111-111
mailto:anonymous@info.com
```

Uniform Resource Identifier

Klasse **URI** (mehrere Konstruktoren)

```
public URI(String scheme,
           String userInfo,
           String host,
           int port,
           String path,
           String query,
           String fragment)
                              throws URISyntaxException
```

- Bei URI geht es nur darum, die Resource zu identifizieren, nicht um den Zugriffsmechanismus!
- Uniform Resource Locator (URL) identifiziert und lokalisiert eine Ressource u.a. über die **Zugriffsmethode (Protokoll)**, erste und häufigste Art von URI

Uniform Resource Locator (URL)

- Klasse **`URL`**

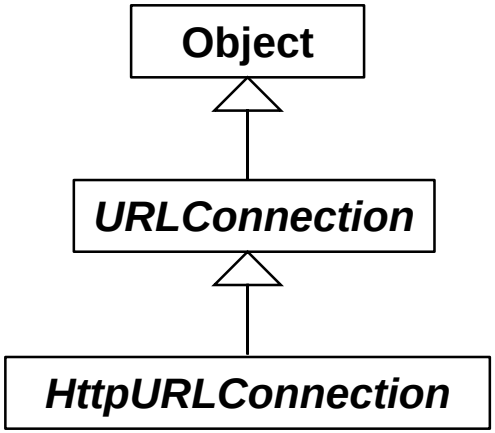
  mehrere Konstruktoren:

```
public URL(String url) throws MalformedURLException
public URL(String protocol,
           String host,
           String file) throws MalformedURLException
public URL(String protocol,
           String host,
           int port,
           String file) throws MalformedURLException
```

URL-Verbindung

- zur Kommunikation mit der URL-Ressource
- ist eine High-Level-Verbindung z.B. über Übertragungsprotokolle wie http(s)
- bauen auf Sockets auf

- abstrakte Klasse
  `URLConnection`




```
URLConnection URL::openConnection() throws IOException
void connect() throws IOException
Map<String, List<String>> getHeaderFields()
Object getContent() throws IOException
InputStream getInputStream() throws IOException
OutputStream getOutputStream() throws IOException
```

Zugriff auf Ressorce:

- Objekt erzeugen: **`openConnection()`**
- Eigenschaften festlegen: **`setXXX()`**, z.B. **`setUseCaches`**
- ggf. wegen geänderter Eigenschaften Verbindung herstellen: **`connect()`**

```
try
{
  URL url = new URL( "http://www.xyz.de/index.html" );
  URLConnection con = url.openConnection();
}
catch ( MalformedURLException e )    // new URL() ging daneben
{ ... }
catch ( IOException e )              // openConnection() schlug fehl
{ ... }
```

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

public class Show_URLText
{
  public static void main( String args[] )
  { URL url;
    URLConnection connection;
    try
      {   url = new URL("http://www..../datei.txt" );
          connection = url.openConnection();
          connection.setUseCaches( false );
          connection.connect();
          InputStream is=connection.getInputStream();
          BufferedReader in = new BufferedReader ( new InputStreamReader(is));
          while( (s = in.readLine() )!= null ) System.out.println(s);
      }
      catch (MalformedURLException e)
      { System.out.println("Url fehlerhaft");
      }
      catch (IOException e)
      { System.out.println("Ein/Ausgabefehler");
      }
    }
  }
```

Socket

- alle höhere Verbindungen bauen auf Socket auf
- Socket bildet eine Schnittstelle zwischen Transport- und Anwendungsschicht
- Socket ist ein Verbindungspunkt im Netz
  UDP: verbindungslos, TCP: verbindungsorientiert
- bekommt eine IP-Adresse und Port-Nummer
- jeder Kommunikationsbeteiligte (Client und Server) implementiert einen Socket

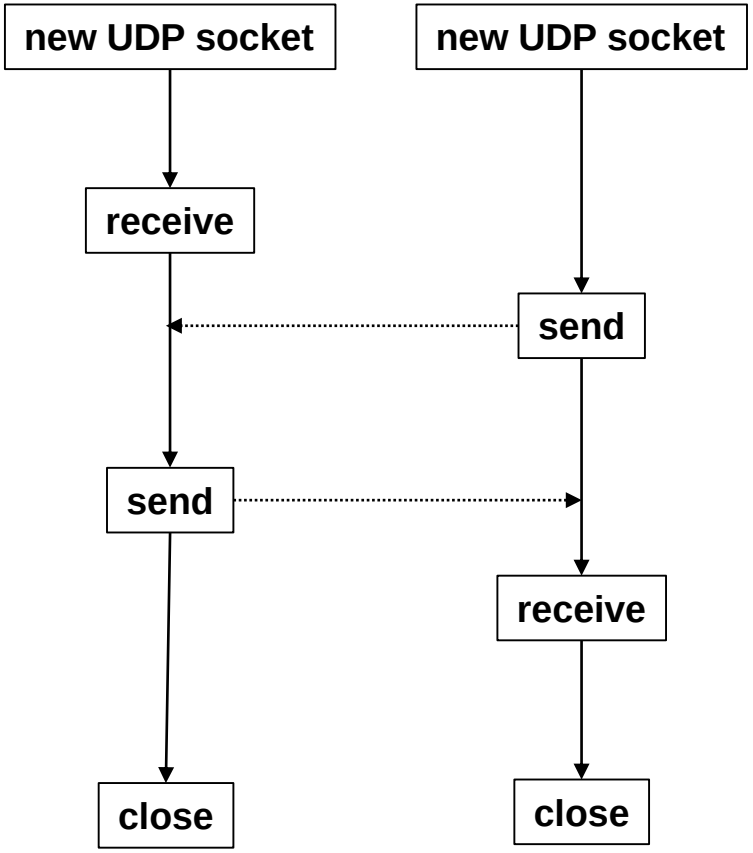
Port-Nummer

System-Ports: 0..1023
User-Ports: 1024..65535

z.B. 80 – Webserver, 20 - FTP

Verbindungslose Kommunikation – UDP (User Datagram Protocol)

- nur Operationen zum Senden und Empfangen
- Schnell aber unzuverlässig
- Empfänger muss empfangsbereit sein
- Sender kennt Empfänger (IP-Adresse und Port)
- Empfänger erfährt die Absenderadresse mit dem Empfang der Nachricht (Pakets)



Datenpaket

Daten
IP-Adresse des Ziels
Portnummer
Absenderadresse

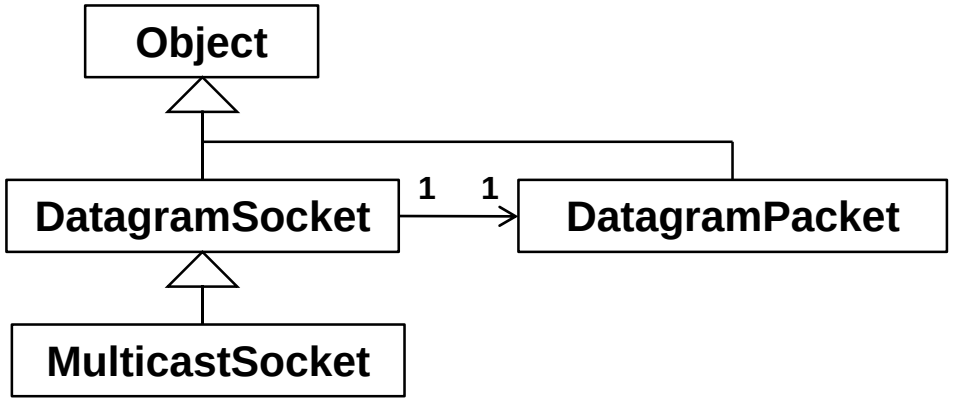
Verbindungslose Kommunikation – UDP

- Klasse **`DatagramPacket`**

```
DatagramPacket(byte[] buf, int offset, int length,
                       InetAddress address, int port)
DatagramPacket(byte[] buf, int length)

//aber auch set-Methoden
```

Verbindungslose Kommunikation – UDP

- Klassen
  `DatagramSocket`
  `MulticastSocket`




```
DatagramSocket() throws SocketException          //any port
                                                 //local host
DatagramSocket(int port, InetAddress laddr)      //spec. local host
                                        throws SocketException
MulticastSocket(int port) throws IOException
```

Quelle: Prof. B.Steinbach, Vorlesung „Verteilte Software“

Senden:

```
void send(DatagramPacket packet) throws IOException
```

zuvor das Packet vorbereiten mit Informationen über Empfänger

Empfangen:

```
void receive(DatagramPacket packet) throws IOException
```

zuvor Paket vorbereiten mit Datenpuffer
nur Pakete mit bekannter Portnummer werden empfangen

Senden:

```
InetAddress ia;
ia = InetAddress.getByName("sende.wohin.?? " );
int port = 4711;
String s = "Die Botschaft";
byte[] data = s.getBytes();
packet = new DatagramPacket( data, data.length, ia, port );
DatagramSocket socket = new DatagramSocket();
socket.send( packet );
```

Empfangen:

```
byte[] data = new byte[ 1024 ];
DatagramPacket packet = new DatagramPacket( data, data.length );
DatagramSocket socket = new DatagramSocket( 4711 );
socket.receive( packet );
```

```
/* sende ein UDP - Paket */
import java.io.*;
import java.net.*;

public class UDPSend
{
  public static void main( String args[] )
  {
    InetAddress iaddr;
    int port = 12345;
    String meldung="Die Meldung";
    try
    { DatagramSocket d_socket = new DatagramSocket();
      iaddr = InetAddress.getByName("server.informatik.tu-freiberg.de");
      byte[] data = meldung.getBytes();
      DatagramPacket packet;
      packet = new DatagramPacket(data, data.length, iaddr, port);
      d_socket.send(packet);
      d_socket.close();
    }
    catch (SocketException e)     {System.out.println(e);}
    catch (UnknownHostException e){System.out.println(e);}
    catch (IOException e)         {System.out.println(e);}
  }
}
```

```
/* empfange 3 UDP Pakete */
import java.io.*;
import java.net.*;

public class UDPReceive
{ public static void main( String args[] )
  {
   int port = 12345;
    byte[] buffer = new byte[1024];
    DatagramPacket packet;
    DatagramSocket d_socket;
    packet = new DatagramPacket(buffer, buffer.length);
    try
    { d_socket = new DatagramSocket(port);
      for (int i = 0; i < 3; i++)
      { d_socket.receive(packet);
        String s = new String( buffer, 0, packet.getLength() );
        System.out.println( "UDPReceive: from "
           + packet.getAddress().getHostName()
           + ":" + packet.getLength() + ":" + packet.getPort() + ":" + s );
        packet.setLength(buffer.length);
      }
      d_socket.close();
    }
    catch (SocketException e) {System.out.println(e);}
    catch (IOException e)     {System.out.println(e);}
  }
}
```

Multicast (mehrere Empfänger):

- Senden per UDP an Klasse D Adressen (224.0.0.0-239.255.255.255)
- Empfangen an Multicast-Adresse/Portnummer mit dem Multicast-Socket

```
MulticastSocket(int port)
```

- Beitreten einer Empfängergruppe

```
void joinGroup(InetAddress addr) throw IOException
```

- Empfangen mit `receive`

- Verlassen der Empfängergruppe

```
void leaveGroup(InetAddress addr) throw IOException
```

```
/* empfange 3 UDP Pakete als Multi-Cast*/
import java.io.*;
import java.net.*;
public class UDPMultiCast
{ public static void main( String args[] )
  {
     int port = 12345;
    byte[] buffer = new byte[1500];
    DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
    MulticastSocket mc_socket;
    InetAddress group;
    try
    { group = InetAddress.getByName("239.255.255.255");
      mc_socket = new MulticastSocket(port);
      mc_socket.joinGroup(group);
      for (int i = 0; i < 3; i++)
      { mc_socket.receive(packet);
        String s = new String( buffer, 0, packet.getLength() );
        System.out.println("MultiCastReceive: from "
           + packet.getAddress().getHostName()
           + ":" + packet.getLength() + ":" + packet.getPort() + ":" + s );
        packet.setLength(buffer.length);
      }
      mc_socket.leaveGroup(group);
      mc_socket.close();
    }
    catch (SocketException e | IOException e) {System.out.println(e);}
  }
}
```

Verbindungsorientierte Kommunikation – TCP (Transmission Control Protocol)

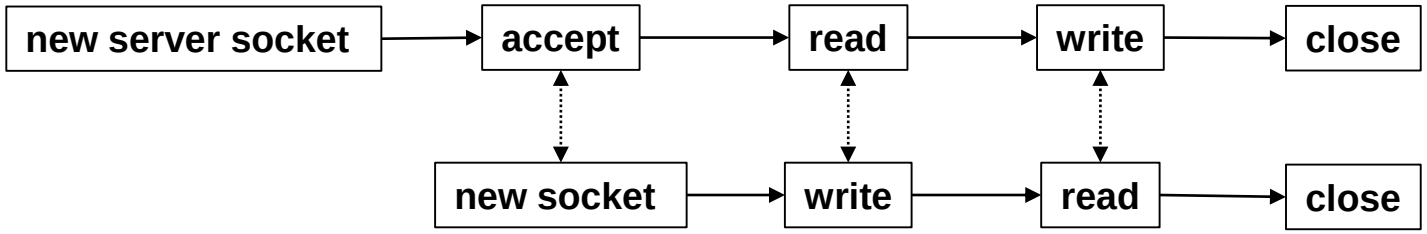
Verbindungsaufbau und Datenübertragung:

(1) Verbindungsaufbau (unterschiedlich für Client und Server):
Client initiiert die Verbindung zum laufenden Server.
Server wartet an einem speziellen Socket (Server-Socket), stellt einen weiteren Socket zum Kommunizieren zur Verfügung.
Server-Socket kann für weitere Verbindungsannahmen benutzt werden.

(2) Datenübertragung über Input- und Output-Streams

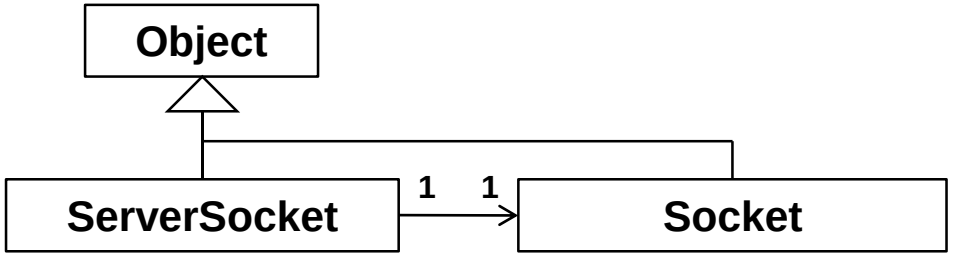
(3) Verbindungabbau, erst Client, dann Server

TCP Socket – Kommunikationsendpunkt mit zwei Datenströmen



Verbindungsorientierte Kommunikation – TCP

- Klassen
  `Socket`
  `ServerSocket`




```
Socket(InetAddress address, int port) throws IOException
void bind(SocketAddress bindpoint) throws IOException
void connect(SocketAddress endpoint) throws IOException
```

bind() ordnet dem Socket eine lokale Adresse zu
connect() stellt eine Verbindung zu einer entfernten (Server)-Adresse her

```
InputStream getInputStream() throws IOException
OutputStream getOutputStream() throws IOException

ServerSocket(int port) throws IOException
Socket accept() throws IOException
```

Quelle: Prof. B.Steinbach, Vorlesung „Verteilte Software“

Client:

- Zieladresse (Objekt) erstellen
- Socket anlegen (mit Zieladresse und Zielport)
- Datenströme vom Socket abfragen
- Kommunizieren über Datenströme mit Hilfe von read() und write()
- Datenströme und Socket mit close() schließen

Server:

- ServerSocket anlegen
- ServerSocket wartet auf Anfragen
- akzeptiert die Anfrage und stellt ein Socket zur Verfügung, ServerSocket kann weitere Anfragen bedienen
- Datenströme vom (Kommunikations-)Socket abfragen und Kommunizieren mit Hilfe von read() und write()
- Datenströme und Sockets (inkl. ServerSocket) mit close() schließen

Server braucht Klassen ServerSocket und Socket,
Client nur Socket

```
/* sende Nachricht über Socket zum TCP-Server und empfange von dort Antwort */
import java.io.*;
import java.net.*;

public class TCPClient
{
  public static void main(String[] args)
  {
    String anfrage= " Anfrage ";
    InetAddress host=InetAddress.getByName(" … ");
     int port=11111;
    try
    {
      Socket socket = new Socket(host, port);
      OutputStream zum_server = socket.getOutputStream();
      zum_server.write(anfrage.getBytes() );
      BufferedReader vom_server
            = new BufferedReader(new InputStreamReader(socket.getInputStream()));
      String antwort = from_server.readLine();
      System.out.println("Der Server meldet: " + antwort);
      zum_server.close();
      vom_server.close();
      socket.close();
    }
    catch (IOException e)
       { System.out.println("Kommunikationsfehler " + e.getMessage());  }
    }
}
```

```
/* empfange Nachricht über Socket vom TCP-Client und sende dorthin Antwort */
import java.io.*; import java.net.*;

public class TCPServer
{ public static void main(String[] args)
  { byte b[] = new byte[64];
    String antwort = "Das ist die Antwort vom Server";
    int port=11111;
    try
    { ServerSocket s_socket = new ServerSocket (port);
      Socket socket = s_socket.accept();
      System.out.println("verbunden mit: " + socket.getInetAddress().getHostName()
         + " Port: " + socket.getPort() + " local Port: " + socket.getLocalPort() );

      BufferedInputStream vom_client = new BufferedInputStream(socket.getInputStream());
      while (vom_client.available() == 0) ;
      vom_client.read(b);
      System.out.println("Message form client: " + new String(b));

      OutputStream zum_client = s_client.getOutputStream();
      zum_client.write(antwort.getBytes());

      vom_client.close();
      zum_client.close();
      socket.close();
      s_socket.close();
    }
    catch (IOException e) { System.out.println("Fehler bei Kommunikation: " + e); }
    }
}
```

Verbindungsorientierte Kommunikation – TCP

Nachteil der Single-Thread-Anwendung:

es kann die Anfrage nur eines Clients bearbeitet werden, andere Anfragen müssen warten

Lösung:

Kommunikationssocket an ein Thread oder ExecutorService weiter reichen

```
/* empfange 3 Nachrichten über Socket vom TCP-Clients und sende jeweils eine Antwort */
/* die Verarbeitung der Nachrichten erfolgt nebenläufig */

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

public class TCPMultiServer
{
    public static void main(String[] args)
    {
        int port=…;
        try
        {   ServerSocket s_socket = new ServerSocket (port);

            for(int anz = 0; anz < 3; anz++)
            {
                Socket socket = s_socket.accept();
                new ServerThread(socket, anz).start();
            }
            s_socket.close();
        }
        catch (IOException e)
        {
            System.out.println("Fehler bei Kommunikation: " + e);
        }
    }
}
```

```
class ServerThread extends Thread
{   Socket socket;
    public ServerThread( Socket socket, int nummer)
    {   this.socket = socket;
        System.out.println("ServerThread " + nummer + " verbunden mit: "
           + socket.getInetAddress().getHostName() + " Port: " + socket.getPort() );
    }
    public void run()
    {   int size;
        byte b[] = new byte[64];
        String antwort = "Serverantwort: *";
        try
        {  BufferedInputStream vom_client =
                new BufferedInputStream(socket.getInputStream());
            while ((size = vom_client.available()) == 0) ;
            vom_client.read(b);
            System.out.println("Message form client: " + new String(b));
            OutputStream zum_client = socket.getOutputStream();
            if (size <= b.length)
                for(int i = 0; i < size; i++)
                   { sleep(300); antwort += ((char)b[i] + "*"); }
            zum_client.write(antwort.getBytes());
            vom_client.close();
            zum_client.close();
            socket.close();
        }
        catch (Exception e) { e.printStackTrace(); }
    }
}
```

Quelle: Prof. B.Steinbach, Vorlesung „Verteilte Software“

```java
public class Server {
public static void main(String[] args) {
  ServerSocket servers;
  try {
   servers = new ServerSocket(5556);
   while (true){
    Socket socket=servers.accept();
    ServerThread st=new ServerThread(socket);
    st.start();
   }
  } catch (IOException e) { e.printStackTrace(); }
}
}
```

```java
public class ServerThread extends Thread {
Socket socket;

public ServerThread(Socket socket) {
  this.socket=socket;
}
public void run() {
  try {
    Scanner sca = new Scanner(socket.getInputStream());
    String s = sca.nextLine();
    System.out.println(s);

    PrintWriter pw= new PrintWriter(socket.getOutputStream(),true); //autoflush
    pw.println("Thank you for your message! I will get back to you later.");
    sca.close();
    pw.close();
    socket.close();
 }catch (IOException e) {e.printStackTrace();}
}
}
```

Text zu übertragen – das geht, was geht noch?

Versenden eines Objektes

- Serialisierung/Deserialisierung - persistent sichern und wiederherstellen. Übertragung übers Netz verwendet das gleiche Mechanismus
- Java Object Serialization (JOS): Sichern der Objektstruktur im binären Format
- rekursiver Ablauf unter Berücksichtigung doppelter Zugriffe und zyklischer Abhängigkeiten
- Voraussetzung: Implementierung von Interface Serializable (java.io.Seralizable)

- viele vordefinierte Klassen sind serialisierbar
- nicht serialisierbar:
  - z.B. Thread, Socket und viele Klassen des lang.io-Pakets
  - transient gekennzeichnete Variable
  - Klassenvariablen (static)

Interface: Serializable (marker interface)

Klassen:

ObjectOutputStream, ObjectInputStream

Serialisierung:

writeObject(daten)

falls Objekt nicht serialisierbar:
java.io.NotSerializableException

Deserialisierung:

readObject(): liefert Object

falls Klasse nicht verfügbar:
java.lang.ClassNotFoundException

Voraussetzungen:

- gleiche Klassen müssen vorhanden sein, SerialisierungsID müssen übereinstimmen
- VM muss sie gleich interpretieren können
- beim Lesen Typumwandlung notwendig

```
import java.io.Serializable;
import java.util.ArrayList;

public class Daten implements Serializable  {

   private static final long serialVersionUID = 1L;
   private ArrayList<String> inhalt;

   public Daten() {
      super();
      inhalt= new ArrayList<String>();
   }

   public ArrayList<String> getInhalt() {
      return inhalt;
   }
}
```

```
import java.io.IOException;
import java.io.NotSerializableException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;

public class ClientDaten {
public static void main(String[] args) {
   try {
            Daten daten=new Daten();
            vorbereite(daten);
            Socket socket = new Socket("localhost", 4445);
            ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());
            oos.writeObject(daten);
            ObjectInputStream ois = new ObjectInputStream(socket.getInputStream());
            Daten zurueck=(Daten)ois.readObject();
            for (int i=0;i<zurueck.getInhalt().size();i++)
                                    System.out.println(zurueck.getInhalt().get(i));
            ois.close();
            oos.close();
            socket.close();
   } catch (NotSerializableException e1) { e1.printStackTrace();
   } catch (ClassNotFoundException e2) { e2.printStackTrace();
   } catch (IOException e) { e.printStackTrace();
   }
}
public static void vorbereite(Daten daten){ /*…*/ }
}
```

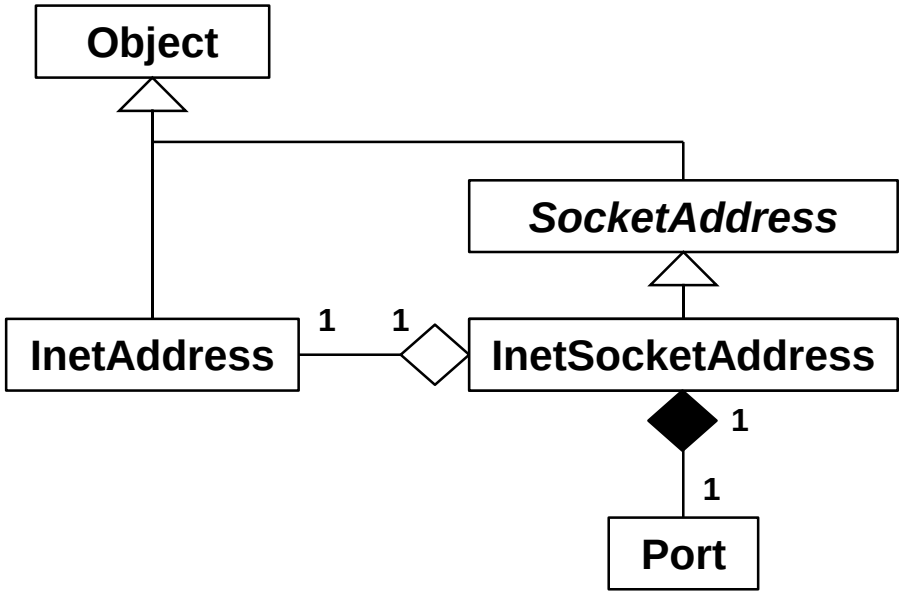
```
import java.net.ServerSocket;
import java.net.Socket;

public class ServerDaten {
public static void main(String[] args) {
   try {
           ServerSocket server = new ServerSocket(4445);
           while(true) {
             Socket socket = server.accept();
              new ServerThread(socket).start();
           }
    } catch (Exception e) {
           e.printStackTrace();
    }
}
}
```

```
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;

public class ServerThread extends Thread {
private Socket socket;

public ServerThread(Socket socket) {
           super();
           this.socket = socket;
}
@Override
public void run() {
           ObjectInputStream ois;
           try {
              ois = new ObjectInputStream(socket.getInputStream());
              Daten daten = (Daten) ois.readObject();
              veraendere(daten);
              ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());
              oos.writeObject(daten);
              ois.close();
              oos.close();
              socket.close();

          } catch (Exception e) {e.printStackTrace();}
}
public void veraendere(Daten daten){ /*…*/ }
}
```

Verbindungsdaten

- keine Methoden zum Verbindungsaufbau, nur Verbindungsdaten

- Klasse
  `InetSocketAddress`




```
InetSocketAddress(InetAddress addr, int port)
InetSocketAddress(int port)
InetSocketAddress(String hostname, int port)
```

Quelle: Prof. B.Steinbach, Vorlesung „Verteilte Software“

Verbindungsdaten

enthält keine Methoden zum Verbindungsaufbau, nur Verbindungsdaten, aber wozu?

- Objekte vom Typ InetSocketAddress sind serialisierbar, Socket nicht
- über SocketAddress-Objekte kann ein Timeout gesetzt werden

```
SocketAddress addr = new InetSocketAddress( host,
port );
Socket socket = new Socket();
socket.connect( addr, 100 ); //timeout in 100 ms
```

TCP vs UDP

TCP: Pakete werden nacheinander zugestellt, vollständig, in der richtigen Reihenfolge und nicht doppelt

UDP: Pakete werden zugestellt ohne Garantie, dass alle Pakete ankommen, in der richtigen Reihenfolge, jedes einmal. Ggf. Korrektur –, Sicherungsmaßnahmen notwendig

	TCP	UDP
Zuverlässigkeit	hoch	niedrig
Geschwindigkeit	Verzögerungen möglich	hoch
Fehlererkennung und -behebung	ja	nein
Empfangsbestätigung	ja	nein
Netzbelastung	höher	niedriger
Benutzung von anderen Protokollen	HTTP, HTTPS, FTP, SMTP, Telnet, …	DNS, TFTP, RTP (VoIP), SRTP, …

Synchronisation

TCP: verwendet Sequenznummern

Sequenznummern: triviale Form einer logischen Uhr

Funktionsschema:

- Jedem Paket wird eine Sequenznummer zugeordnet und die nächste Sequenznummer mit abgeschickt
- Empfänger kennt aus der Sequenznummer des vorangegangenes Pakets, welche Nachricht dran ist
- Wird die Nachricht mit der niedrigeren Nummer empfangen, so wird sie verworfen
- Die Nachricht mit der höheren Nummer wird empfangen und in den Zwischenspeicher abgelegt
- Wenn die Nachricht mit der richtigen Nummer angekommen ist, wird Zwischenspeicher abgearbeitet
- Kommt die Nachricht nicht, wird sie nach einer gewissen Zeit von Sender abgefordert
- Nach einer gewissen Anzahl erfolgslosen Rückfragen bricht die Kommunikation mit einer Fehlermeldung ab

Echtzeituhr:

- misst physikalische Zeit
- Synchronisation erfolgt, z.B. mittels Abfrage von Zeitservern und Umrechnungen der Verzögerung durch Übertragungszeit (verschiedene Algorithmen, Standard: Network Time Protokoll)

Logische Uhr:

- misst nicht die physikalische Zeit, gib den Ereignissen einen eindeutigen (logischen) Zeitstempel
- erzeugt streng monoton steigende Werte, um den Ereignissen eine Kasualordnung zuzuweisen

Verfahren zum Zuweisen von eindeutigen Zeitstempeln an Nachrichten:

Lamport-Uhr:

- Jeder Prozess hat einen Zähler (die *Uhr*)
- Der aktuelle Stand des Zählers wird an jede Nachricht als Zeitstempel angehängt
- Zähler wird bei jedem Ereignis (Senden und Empfangen) erhöht:
  wenn eine Nachricht empfangen wird, deren Zeitstempel größer oder gleich dem aktuellen Stand der eigenen Uhr ist, wird der Wert der Uhr um eins erhöht.
- Nachteil: unbekannt, welche Ereignisse *kausal unabhängig* (nebenläufig) sind

Vektoruhr:

- Berücksichtigt Nebenläufigkeit
- die Uhr jedes Prozesses besteht aus einem Vektor (Array), nicht nur einem Zähler
- Jeder Prozess merkt sich den Zählerstand aller anderen Prozesse
- Der aktuelle Stand der Uhr wird jeder gesendeten Nachricht angehängt
- Bei jedem Ereignis wird immer nur der *eigene* Zähler erhöht
- Wird eine Nachricht empfangen, werden aus dem aktuellen und dem empfangenen Vektor elementweise Maxima gebildet

```
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;

public class Client {

           public static void main(String[] args) {
                      Socket socket;
                      try{
                          socket= new Socket("139.20.16.xxx",5556);
                          PrintWriter pw= new PrintWriter(socket.getOutputStream());
                          pw.println("Greetings!");
                          pw.flush();
                          Scanner sca = new Scanner(socket.getInputStream());
                          String s = sca.nextLine();
                          System.out.println(s);
                          pw.close();
                          sca.close();
                          socket.close();
                       }catch(Exception e){
                          System.out.println(e);
                       }
           }
}
```

```
public class MainActivity extends AppCompatActivity {
 public class TCPKomm implements Callable<String>{
    String text=null;
    public TCPKomm(String message){
       text=message;
    }
    @Override
    public String call() throws Exception {
       Socket socket;
       try{
          socket=new Socket("10.0.2.2",5556);
                //("139.20.16.xxx",5556);
          PrintWriter pw=
                new PrintWriter(socket.getOutputStream(),true);
          pw.println(text);
          pw.flush();

          Scanner sca = new Scanner(socket.getInputStream());
          String s = sca.nextLine();
          pw.close();
          sca.close();
          socket.close();
          return s;
       }catch(Exception e){
          System.out.println(e);
          return null;
       }
    }
}
```

```
   EditText textedit=null;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);
       textedit=findViewById(R.id.textedit);
    }

    public void onButtonClick(View view) {
       String message=textedit.getText().toString();
       ExecutorService exec =
             Executors. newSingleThreadExecutor();
       Future<String> future =
             exec.submit(new TCPKomm(message));
       try{
          String s=future.get();
          textedit.setText(s);
       }
       catch (InterruptedException | ExecutionException ex )
       {
          ex.printStackTrace();
       }
       exec.shutdown();
    }
}
```