Referat:
Netzwerkprogrammierung in Java

1.)ISO/OSI und Internet Referenzmodell

ISO/OSI 7 Schichtenmodell (1-Bitübertragung, 2-Sicherung, 3-Netzwerk, 4-Transport, 5-Sitzung, 6-Darstellung, 7-Anwendung)

- vereinfachtes Internet-Referenzmodell aus 4 Schichten:
  - Anwendungsschicht (Schicht 4)
  - Transportschicht (Schicht 3)
  - Internetschicht (Schicht 2)
  - Netzwerkinterfaceschicht (Schicht 1)

- Verwendung von Protokollen für die Kommunikation zwischen verschiedenen Schichten, die beide Partner kennen müssen (ähnlich einer gemeinsamen Sprache)
- Kommunikation zwischen den Schichten:
  - vertikale (reale) Kommunikation zwischen überinanderliegenden Schichten
  - horizontale (virtuelle) Kommunikation zwischen zwei Schichten gleichen Typs in unterschiedlichen Endsystemen

(1) Netzwerkinterfaceschicht:
- Umwandlung von Daten in Signale
- Funktion abhängig vom Medium (Funk, Lichtleiter, Kupferkabel)
- Auflösung nach MAC-Adressen
- Flußsteuerung
- Fehlererkennung und Korrektur

(2) Internetschicht:
- Wegewahl im Netz (Routing)
- IP Protokoll (Paketversand)
- IP Adressen (Klasse A/B/C- Adressen nach Längen und somit Anzahl bestimmt)
  wichtig: IP verbindungslos und unzuverlässig, Adresse ist eindeutig
- Staukontrolle
- Quality of Service (Garantien für Zuverlässigkeit und Sicherheit) nicht befriedigend möglich

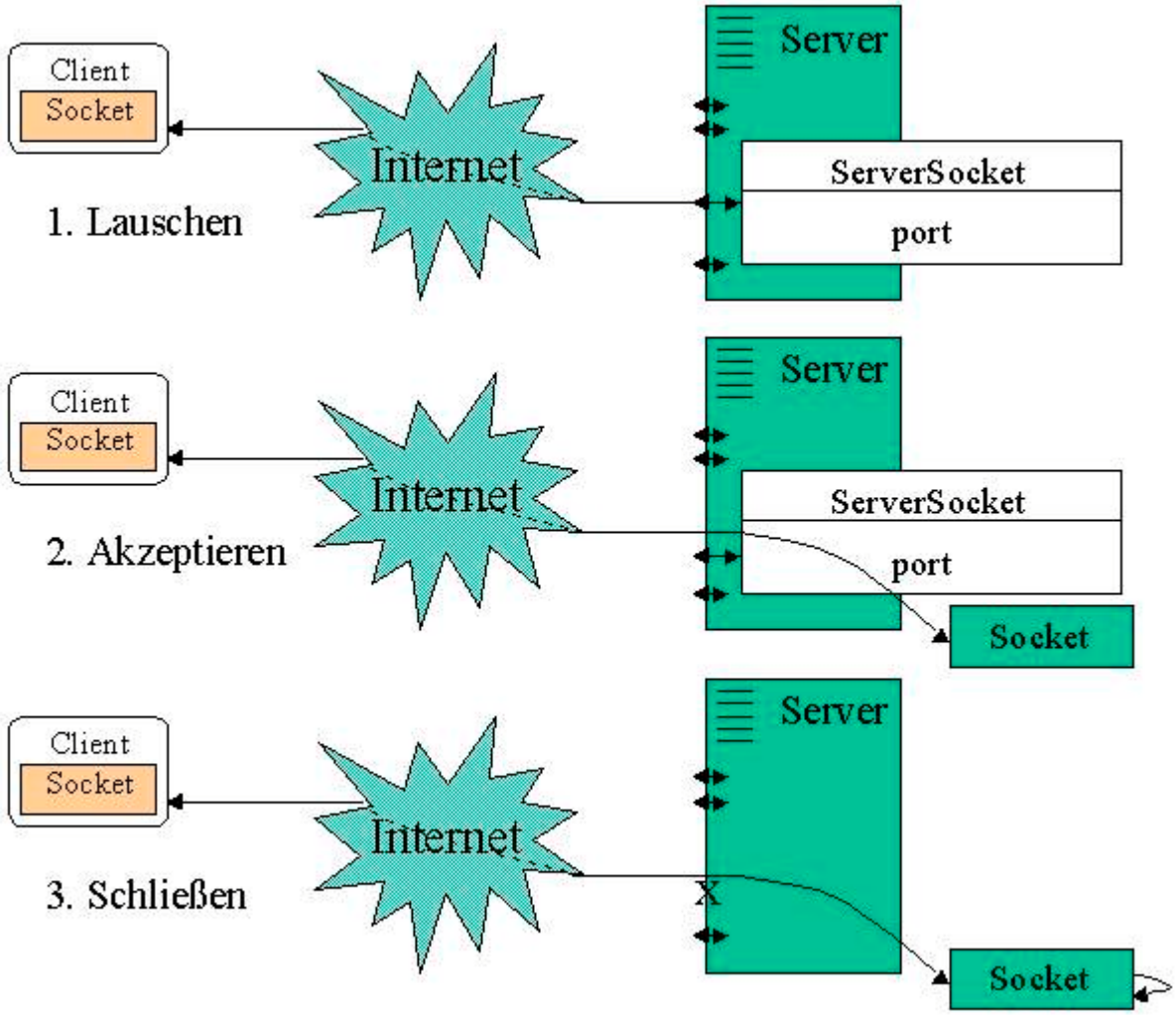
(3) Transportschicht:
- TCP Protokoll (verbindungsorientiert, setzt auf IP eine sichere, fehlerfreie Punkt-zu-Punkt-Verbindung auf)
- Portnummern dienen der Unterscheidung mehrerer Server
  - 0 bis 1023 für well-known Ports reserviert, darüber bis 65535 frei verfügbar
  - echo 7
  - ftp 21
  - telnet 23
  - smtp 25
  - www 80
  - pop3 110
  - ursuppe-server 14195 oder 14196
- QoS-Bereitsstellung auf den unbeeinflußbaren Medien der Internetschicht via IP
- Segmentierung/Reassemblierung von Paketen (Teilen und Zusammenschließen)

(4) Anwendungsschicht:
- Plattform für diverse Anwendungen, u. a. Ursuppe Clienten
- horizontale Kommunikation mit der Server-Anwendung

Verwendung von TCP/IP zur Steuerung der Kommunikation des Clients und des Servers im Projekt

2.)Client – Server – Architektur in Java

Es gibt einen Server, der an einem bestimmten Port auf eine Verbindungsanfrage wartet. Dies tut er, indem der Server einen Socket öffnet. Der Client öffnet auf seiner Seite einen Socket mit der Anfrage nach einer Verbindung. Nun können Daten ausgetauscht werden. Danach wird die Verbindung geschlossen, in dem beide Sockets geschlossen werden.



- Sockets sind streambasierte Netzwerkschnittstellen zur Kommunikation zweier Rechner in einem TCP/IP – Netz (jeweils auf Client- und Serverseite vorhanden)
- sie sind im Paket Java.net enthalten
- es gibt in Java Clientsockets und Serversockets
- zur Adressierung von Rechnern wird das Objekt *InetAddress (*enthält IP und URL des Rechners) benutzt => IP-Adresse wird mit *String getHostAddress*() oder *byte[] getAddress()*, die URL mit *String getHostName()* abgefragt
- die Methode *public static InetAddress getLocalHost() throws UnknownHostException()* liefert die Adresse des eigenen Rechners

ClientSockets in Java:

- versendet Connection-Request an Serversocket
- die drei Phasen der Verbindung sehen in Java so aus:

  - Wunsch des Verbindungsaufbaus => Erzeugen eines neuen Socket – Objekts:
    Socket sock = new Socket (InetAddress IpAdresse, int PortNummer)
    oder
    Socket sock = new Socket (String Host, int Portnummer)
  - Datenaustausch über Streams =>
    InputStreams in = sock.getInputStraem()
    OutputStream out = sock.getOutputStream()
  - Verbindungsabbau => Streams und Socket schließen
    in.close()
    out.close()
    sock.close()

Bsp.: Zugriff auf einen Web-Server

```
[1]001 /* Listing4504.java */
002
003 import java.net.*;
004 import java.io.*;
005
006 public class Listing4504
007 {
008   public static void main(String[] args)
009   {
010     if (args.length != 2) {
011       System.err.println(
012         "Usage: java Listing4504 <host> <file>"
013       );
014       System.exit(1);
015     }
016     try {
017       Socket sock = new Socket(args[0], 80);
018       OutputStream out = sock.getOutputStream();
019       InputStream in = sock.getInputStream();
020       //GET-Kommando senden
021       String s = "GET " + args[1] + " HTTP/1.0" + "\r\n\r\n";
022       out.write(s.getBytes());
023       //Ausgabe lesen und anzeigen
024       int len;
025       byte[] b = new byte[100];
026       while ((len = in.read(b)) != -1) {
027         System.out.write(b, 0, len);
028       }
029       //Programm beenden
030       in.close();
031       out.close();
032       sock.close();
033     } catch (IOException e) {
034       System.err.println(e.toString());
035       System.exit(1);
036     }
037   }
038 }[2]
```

aus dem Buch „GoTo Java 2 – Handbuch der Java-Programmierung“, Addison-Welsley Verlag

ServerSocket in Java:

- mittels Erzeugen eines Objekts *ServerSocket echod = new Serversocket(int portNummer)* lauscht der Server am übergebenem Port und wartet auf Request
- er akzeptiert eine Anfrage mit: *Socket sock = new echod.accept()* => Verbindung hergestellt
- Verbindungsabbau mittels *sock.close()* und *echod.close()*

Bsp.: Ein Echo-Server für Port 7

```
[3]001 /* SimpleEchoServer.java */
002
003 import java.net.*;
004 import java.io.*;
005
006 public class SimpleEchoServer
007 {
008   public static void main(String[] args)
009   {
010     try {
011       System.out.println("Warte auf Verbindung auf Port 7...");
012       ServerSocket echod = new ServerSocket(7);
013       Socket socket = echod.accept();
014       System.out.println("Verbindung hergestellt");
015       InputStream in = socket.getInputStream();
016       OutputStream out = socket.getOutputStream();
017       int c;
018       while ((c = in.read()) != -1) {
019         out.write((char)c);
020         System.out.print((char)c);
021       }
022       System.out.println("Verbindung beenden");
023       socket.close();
024       echod.close();
025     } catch (IOException e) {
026       System.err.println(e.toString());
027       System.exit(1);
028     }
029   }
030 }[4]
```

aus dem Buch „GoTo Java 2 – Handbuch der Java-Programmierung“, Addison-Welsley Verlag

Streams:

- zunächst abstraktes Konstrukt, das Zeichen auf imaginäres Ausgabegerät schreibt, und von diesem liest
- erst konkrete Unterklassen binden Zugriffsroutinen an echte Ein-und Ausgabegeräte (wie Dateien, Strings, Kommunikationskanäle im Netzwerk)
- Unterscheidung zwischen InputStream und OutputStreams
- eingehende Daten => Input, ausgehende Daten => Output
- in Java gibt es Characterstreams (Unicode – 16 Bit) und Bytestreams (1 Byte), Umwandlung ineinander möglich (wir arbeiten bei der Netzwerkprogrammierung mit Byte-Streams!)
- Klasse OutputStream benutzt folgende Methoden:

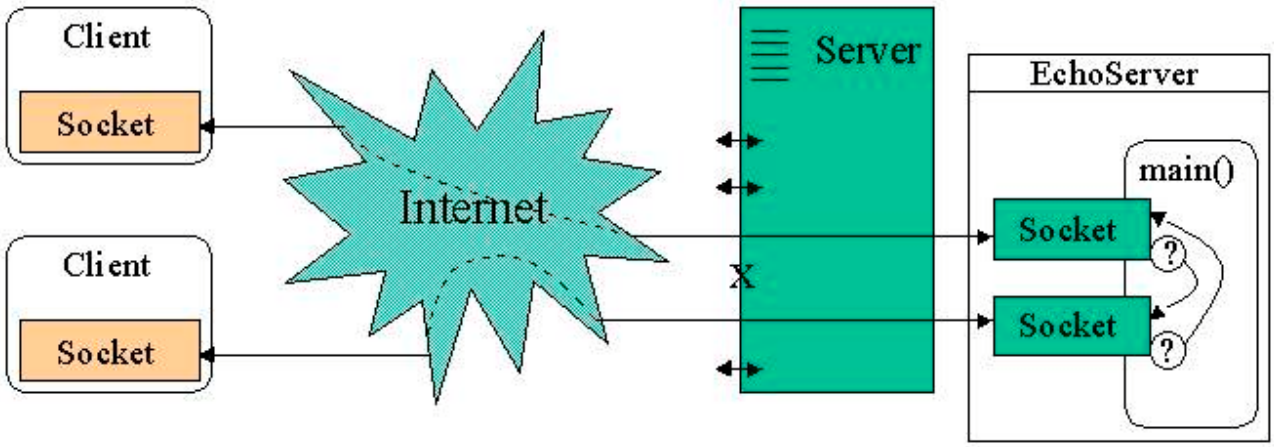
  protected OutputStream() *public void flush()*
  public void close() *public void write (int b)*
  public void write (byte[] b) *public void write (byte[] b, int offs, int len)*

- *close* schließt den OutputStream, *flush* schreibt gepufferten Daten aufs Ausgabegerät und leert Puffer, *write* erwartet Bytes oder Byte-Array als Daten
- Klasse InputStream benutzt folgende Methoden:

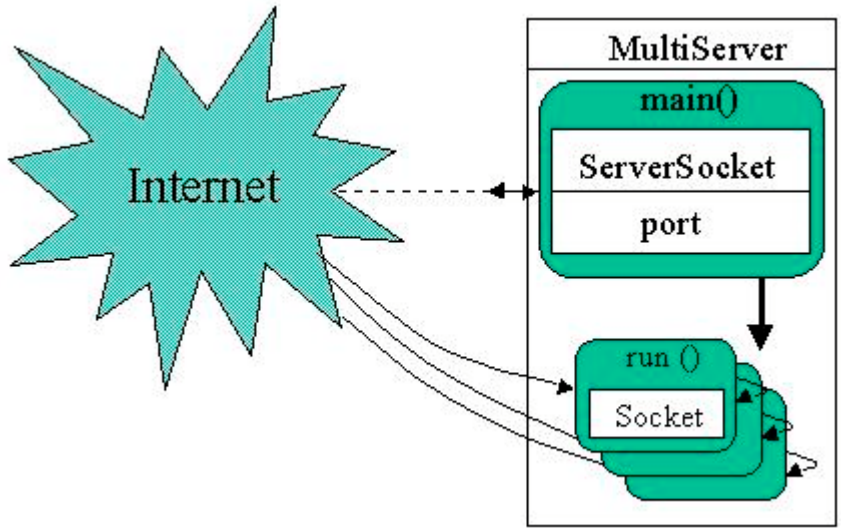
public long skip (long n) throws IOException	überspringt Bytes
public int available () throws IOException	lesbare Bytes ohne Blockieren
public void close() throws IOException	schließt Stream
public void mark (int readlimit)	markiert aktuelle Position
public long reset() throws IOException	springt zu mark zurück
public boolean marksupported()	wird Markieren unterstützt?
public int read() throws IOException	liest Daten ein
public int read(byte[] b) throws IOException	
public int read(byte[] b, int offs, int len) throws IOException	

3.)Probleme:

- nur ein Client kann sich mit dem Server verbinden, denn ein Socket verbindet immer nur zwei Rechner
- **Lösung1:** Erstellen von mehreren Sockets, Wechsel zwischen den Sockets in der Main (jedoch immer nur ein aktiver Socket, die anderen blockieren dann)



- **Lösung 2:** Multithreading (für jede eingehende Verbindung wird ein neuer Thread eröffnet und nummeriert, um einzeln ansprechbar zu sein)



- jeder Client erhält bei Öffnen des Thread eine Begrüßungsmeldung
- bei Beenden der Verbindung wird eine entsprechende Meldung ausgegeben

```
[5]001 /* EchoServer.java */
002
003 import java.net.*;
004 import java.io.*;
005
006 public class EchoServer
007 {
008   public static void main(String[] args)
009   {
010     int cnt = 0;
011     try {
012       System.out.println("Warte auf Verbindungen auf Port 7...");
013       ServerSocket echod = new ServerSocket(7);
014       while (true) {
015         Socket socket = echod.accept();
016         (new EchoClientThread(++cnt, socket)).start();
017       }
018     } catch (IOException e) {
019       System.err.println(e.toString());
020       System.exit(1);
021     }
022   }
023 }
024
025 class EchoClientThread
026 extends Thread
027 {
028   private int    name;
029   private Socket socket;
030
031   public EchoClientThread(int name, Socket socket)
032   {
033     this.name   = name;
034     this.socket = socket;
035   }
036
037   public void run()
038   {
039     String msg = "EchoServer: Verbindung " + name;
040     System.out.println(msg + " hergestellt");
041     try {
042       InputStream in = socket.getInputStream();
043       OutputStream out = socket.getOutputStream();
044       out.write((msg + "\r\n").getBytes());
045       int c;
046       while ((c = in.read()) != -1) {
047         out.write((char)c);
048         System.out.print((char)c);
049       }
050       System.out.println("Verbindung " + name + " wird beendet");
051       socket.close();
052     } catch (IOException e) {
053       System.err.println(e.toString());
054     }
055   }
056 }[6]
```