# Informatik B - Objektorientierte Programmierung in Java

## Vorlesung 23: Netzwerkprogrammierung/ Kommunikation 2





## Inhalt

- Netzwerkprogrammierung/ Kommunikation (Teil 2)
  - Übung: multithreaded EchoServer
  - Beispiel: Chat
  - UDP
  - Beispiel: Time-Service





## Verbindungen zu mehreren Clients

Wie könnte das bereits vorgestellte Programm des Echo-Servers (**`SimpleEchoServer.java`**) folgendermaßen erweitern werden?

- Der Server soll mehr als einen Client gleichzeitig bedienen können. Die Clients sollen zur besseren Unterscheidung durchnumeriert werden.
- Für jeden Client soll im Server ein eigener Thread angelegt werden.





## Übungsaufgabe: `EchoServer.java`

- Um diese Anforderungen zu erfüllen, verändert man das obige Programm folgendermaßen: Im Hauptprogramm wird nun nur noch der **`ServerSocket`** erzeugt und in einer Schleife jeweils mit **`accept`** auf einen Verbindungswunsch gewartet. Nach dem Verbindungsaufbau erfolgt die weitere Bearbeitung nicht mehr im Hauptprogramm, sondern es wird ein neuer Thread mit dem Verbindungs-Socket als Argument erzeugt. Dann wird der Thread gestartet und erledigt die gesamte Kommunikation mit dem Client. Beendet der Client die Verbindung, wird auch der zugehörige Thread beendet. Das Hauptprogramm braucht sich nur noch um den Verbindungsaufbau zu kümmern und ist von der eigentlichen Client-Kommunikation vollständig befreit.

# Beispiel: Chat-Server und -Clienten

- Der Chat-Server startet einen Thread für jeden angemeldeten Clienten; jeder Thread hält einen Socket. Der Server-Prozess verwaltet alle OutputStreams in einem **`static Vector outStreams`**, der als Monitor auch zur Synchronisation eingesetzt wird. Meldet sich ein neuer Client an oder ab, werden alle anderen vom Server darüber informiert: „frank joins/leaves“. Diese Meldung schreibt der Server in alle OutputStreams. Im laufenden Chat-Betrieb wird die Nachricht eines Clienten vom Server an alle anderen weitergegeben.





# Chat-Server: main()

```
public static void main(String[] args) {
   ServerSocket listenSocket;
   try {
     listenSocket = new ServerSocket(PORT);
     while (true) {
       Socket sock = listenSocket.accept();
       new ChatServer(sock);
     }
   } catch(IOException e)
              {e.printStackTrace();}
 }
```





# Chat-Server: run() (Ausschnitt)

```
name = in.readLine();          // liest Namen des neuen Chatters
synchronized (outStreams) {
  for (int i = 0; i < outStreams.size(); i++) {
    ((PrintStream)outStreams.get(i)).println(name + " joins");
  }                            // Meldung an alle: Neuer Chatter
  outStreams.add(out);         // OutputStream hinzufügen
}
while (true) {                 // Hauptschleife:
  line = in.readLine();        // vom Clienten lesen
  if (line == null) break;     // bis EOF
  synchronized (outStreams) { // an alle Clienten ausgeben
    for (int i = 0; i < outStreams.size(); i++) {
      ((PrintStream)outStreams.get(i)).println(name + ": " + line);
    }
} }
synchronized (outStreams) {
  outStreams.remove(out);      // entfernt den Chatter
  sock.close();                // schließt den Socket
  for (int i = 0; i < outStreams.size(); i++) {
    ((PrintStream)outStreams.get(i)).println(name + " leaves");
  }                            // Meldung an alle: Chatter ist weg
}
```





# Beispiel: Chat-Server und -Clienten

- Der Chat-Client wird mit dem Benutzernamen und dem Namen des Chat-Servers geöffnet:

**`java ChatClient frank localhost`**

Er startet eine Socketverbindung zum Server und einen zweiten Thread, der aus dem InputStream vom Server liest und dies auf der Console ausgibt, während der Haupt-Thread von der Console liest und dies in den OutputStream zum Server schreibt. Der Chat endet für einen Clienten mit Ctrl-Z bzw. Ctrl-D, dadurch endet die Eingabeschleife und der Client schließt den Socket.

## Beispiel: Chat-Server und -Clienten

- Das Schließen des Sockets führt dazu, dass der Server ein End-Of-File erhält, seine Schleife und seinen Thread beendet. Anschließend endet der Haupt-Thread des Clienten und weil der zweite Thread ein Dämon ist, wird auch dieser etwas unsanft beendet, obwohl er gerade bei **`readLine()`** auf Input vom Server wartet. Dies führt zu einer **`SocketException`**, die abgefangen und ignoriert wird, um den Client-Prozess möglichst schnell zu beenden, was der Benutzer mit seinem Ctrl-Z bzw. Ctrl-D ja gewünscht hat.





## Chat-Client: main() (Ausschnitt)

```
sock = new Socket(args[1], PORT);
out = new PrintStream(sock.getOutputStream());
in = new BufferedReader(
new InputStreamReader(sock.getInputStream()));
consoleIn = new BufferedReader(new InputStreamReader(System.in));
out.println(args[0]); // Name des neuen Chatters
new ChatClient(in);
while (true) {
  line = consoleIn.readLine(); // liest von Console
  if (line == null) break;     // Chat-Ende mit Ctrl-Z
  out.println(line);           // schickt an Server
}
sock.close();
```





## Chat-Client: run()

```
public void run() {
   String line;
   try {
     while (true) {
       line = in.readLine();     // liest vom Server
       if (line == null) break;  // Abbruch bei Ctrl-Z
       System.out.println(line); // schreibt auf Console
     }
   } catch(SocketException e) { /* bei Chat-Ende */ }
     catch(IOException e) {e.printStackTrace();}
}
```





## Wiederholung: TCP und UDP

- TCP ist ein *verbindungsorientiertes* Protokoll, das auf der Basis von IP eine sichere und fehlerfreie Punkt-zu-Punkt-Verbindung realisiert. (Analogie: Telefonieren)
- Daneben gibt es ein weiteres Protokoll oberhalb von IP, das als *UDP* (*User Datagram Protocol*) bezeichnet wird. Im Gegensatz zu TCP ist UDP ein *verbindungsloses* Protokoll, bei dem die Anwendung selbst dafür sorgen muss, dass die Pakete überhaupt und in der richtigen Reihenfolge beim Empfänger ankommen. Sein Vorteil ist die größere Geschwindigkeit gegenüber TCP. In Java wird UDP durch die Klassen **`DatagramPacket`** und **`DatagramSocket`** abgebildet. (Analogie: Brief)

# UDP

- ein Prozess lauscht auf Datenpakete
- ein anderer Prozess schickt dem ersten ein Datenpaket
- der empfangende Prozess erkennt am Datenpaket den Absender und kann somit Datenpakete an diesen adressieren
- Wird Fehlerkorrektur gewünscht, z.B. um verlorene Pakete zu erhalten, so müssen das die Prozesse selbst leisten.





# UDP

- Im Gegensatz zu TCP erzeugt UDP wesentlich weniger Netzbelastung
  - keine Verbindung, sondern einzelne Pakete
  - ungesichert
- UDP wird eingesetzt bei Kommunikation, die leicht von Client und Server selbst überwacht werden kann
  - falls Netzlast klein zu halten ist
  - wenn keine Gewähr für das Ankommen der Pakete übernommen werden muss
  - bei "Broadcast", d.h. einer sendet an viele





# UDP: `DatagramSocket`

- Die Klasse **`DatagramSocket`** stellt für UDP einen Verbindungspunkt zum Senden und Empfangen von Datenpaketen zur Verfügung.
- Konstruktoren:

  **`DatagramSocket()`** // am ersten freien Port
  **`DatagramSocket(int port)`** // am angegebenen Port
  **`DatagramSocket(int port, InetAddress iaddr)`**
  // am angegebenen Port und angegebener IP-Adresse, falls Rechner mehrere hat
- einige Methoden:
  **`send(datagrampacket)`**
  **`receive(datagrampacket)`**
  **`close()`**





# UDP: `DatagramPacket`

- Die Klasse **`DatagramPacket`** ermöglicht für UDP das Verschicken und Empfangen von Paketen
- Konstruktoren:

  **`DatagramPacket(byte buf[], int buflen)`** // Server-seitig

  **`DatagramPacket(byte buf[], int buflen, InetAddress iaddr, int port)`**
  // Client-seitig: Angabe des Empfängers

## UDP: `DatagramPacket`-Methoden

- einige Methoden von **`DatagramPacket`**:

```
getAddress()
getData()
getLength()
getPort()
setAddress(iaddr)
setData(bytebuffer)
setLength(int)
setPort(port)
```





## Beispiel: Time-Service mit UDP

- Der Server wartet an Port 4711. Der Server verschickt in einem Datenpaket mittels UDP an den Client, der ihm als Aufforderung ein Datenpaket geschickt hat, die aktuelle Zeit als formatierten String.





## Beispiel: UdpServer.java

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

public class UdpServer {
  private static final java.text.DateFormat DF =
                       java.text.DateFormat.getDateTimeInstance();
  private static String getTime() {
    return DF.format(new java.util.Date());
  }
  public static void main(String[] args) {
   try {
      DatagramSocket ds = new DatagramSocket(4711);
      DatagramPacket dp = new DatagramPacket(new byte[20], 20);
      while (true) {
        ds.receive(dp);
        System.out.println("Anforderung von Port " + dp.getPort());
        dp.setData(getTime().getBytes());
        ds.send(dp);
      }
    } catch (SocketException e) { e.printStackTrace();}
      catch (IOException e) { e.printStackTrace();}
} }
```





## Beispiel: Time-Service mit UDP

- Der Client verschickt ein Datenpaket mittels UDP an den Server (auf demselben Rechner, Port 4711) als Aufforderung, ihm die aktuelle Zeit zu schicken. Dazu benutzt der Client den ersten freien Port.
- Der Methodenaufruf **`send()`** kehrt sofort zurück, es erfolgt im Gegensatz zu TCP kein Warten auf das Zustandekommen einer Verbindung. Anschließend empfängt der Client die Zeit vom Server.

# Beispiel: UdpClient.java

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

public class UdpClient {
  public static void main(String[] args) {
    try {
      DatagramPacket dp = new DatagramPacket(new byte[20],
                    20, InetAddress.getLocalHost(), 4711);
      DatagramSocket ds = new DatagramSocket();
      ds.send(dp);
      ds.receive(dp);
      System.out.println(new String(dp.getData()));
   } catch (UnknownHostException e) { e.printStackTrace();}
     catch (SocketException e) { e.printStackTrace();}
     catch (IOException e) { e.printStackTrace();}
  }
}
```





# Beispiel UDP Time-Service starten

- Zuerst den UDP-Server starten:

**`java UdpServer`**

- Dann einen oder mehrere UDP-Clients starten:

**`java UdpClient`**

- Hinweis: Der Server kann mehrere Clients bedienen, ohne multithreaded zu sein.





# Zusammenfsg.: Netzwerkprogrammierung

- Grundlagen und Anwendungen der Netzwerkprogrammierung
- Grundlagen der Protokolle TCP und IP
- IP-Adressen, Netztypen und Domain-Namen
- Portnummern und Applikationen
- Das Konzept der Sockets
- Die Klasse **`InetAddress`** zur Adressierung von Socket-Verbindungen
- Die besonderen Adressen localhost und 127.0.0.1
- Aufbau einer Client-Socket-Verbindung mit Hilfe der Klasse **`Socket`** und dem streambasierten Zugriff auf die Daten
- Kommunikation mit dem Echo-Server
- Lesender Zugriff auf einen Web-Server
- Konstruktion von Serverdiensten mit der Klasse **`ServerSocket`**
- Konstruktion mehrbenutzerfähiger Server mit Threads (Bsp.: Chat)
- UDP – Beispiel: Time-Service