In diesem Kapitel:
- *Was ist ein UDP Datagramm*
- *Die DatagrammPacket Klasse*
- *Die DatagrammSocket Klasse*
- *Höhere Abstraktionslevel*
- *Anwendungsbeispiele*

9

UDP Datagramme und Sockets

9.1. Einführung

In den beiden vorher gehenden Kapiteln haben wir Beispiele kennen gelernt, die alle auf TCP aufbauen. TCP wurde für die stabile Verbindungen geplant und implementiert. Den Preis den man dafür bezahlt, ist die reduzierte Übertragungsgeschwindigkeit.

Falls Datagramme ausserhalb der ursprünglichen Sequenz ankommen, sorgt TCP dafür, dass die Reihenfolge korrigiert wird.

TCP hat eine "billige Cousine", UDP : hier werden die Daten schnell aber eher unzuverlässig übertragen. Es kann vorkommen, dass Datagramme verloren gehen.

Die Frage ist, was man mit einem Protokoll anfangen kann, welches unzuverlässig ist? Die Antwort ist heute sicher nicht mehr schwierig zu finden, da im Web viele Applikationen existieren, bei denen die Übertragung nicht 100% zuverlässig erfolgen nuss, zum Beispiel Streaming Applikationen, wie Video oder Audio Streaming.

Ein anderes Beispiel ist DNS (Domain Name System), bei dem periodisch eine Abstimmung der verschieden Datenbanken erfolgt. Falls die Abstimmung nicht funktioniert, stört das höchstens kurzfristig, da nach kurzer Zeit die Daten erneut übermittelt werden.

Ein bekanntes Dateisystem, NFS (Network File System) von Sun Microsystem beruht ebenfalls auf UDP.

UDP und TCP können etwa so verglichen werden:
- UDP ist wie eine Radio Kommunikation
  Die Radiowellen werden einfach in den Äther hinaus gesandt.
- TCP ist wie eine Telefonverbindung
  Eine Verbindung zum Gesprächspartner wird aufgebaut und zuverlässig aufrecht erhalten

Beide Systeme haben ihre Vorteile und Nacheile.

Im Rahmen von `java.net` werden die UDP spezifischen Teile in zwei Klassen eingeteilt:
- `DatagrammSocket` mit der Klasse `DatagrammSocketImpl`
- `DatagrammPacket`

die beide direkt von der Klasse `Object` abhängen, im Package `java.net.`

Wenn man Daten mit Hilfe von UDP versenden möchte, packt man die Daten in ein `DatagrammPacket` und versendet sie mit Hilfe eines `DatagramSockets`.
Beim Empfangen geht man umgekehrt vor : es wird ein `DatagrammPacket` über ein `DatagramSocket` empfangen und anschliessend werden die Daten aus dem `DatagramPacket` gelesen.

Die DatagramPackets enthalten die Zieladresse als Teil der Headerinformation.

Die DatagramSockets sind sehr einfach aufgebaut:
der Socket muss nur den Port kennen, auf den der Socket hören oder senden muss.

Die Unterschiede zwischen UDP und TCP lassen sich wie folgt zusammen fassen:

- UDP kennt keine Unterscheidung zwischen Socket und ServerSocket.
- in TCP besteht die Verbindung zwischen Client und Server aus Input und Output Streams, die Reihenfolge der Pakete ist klar definiert;
  in UDP werden Datagramme zusammen gestellt und weggeschickt, die Reihenfolge der Pakete ist nicht definiert
- ein TCP Socket hört genau auf einen Port und kommuniziert mit einem wohl definierten Host; ein UDP Socket kann gleichzeitig auf viele Hosts hören, er hört auf einzelne Pakete und filtert die für ihn bestimmten Pakete heraus.

9.2. Globale Lernziele

- Sie kennen den Aufbau eines Datagramms.
- Sie kennen die wichtigsten Konstruktoren und Methoden der DatagramPacket und der DatagramSocket Klassen.
- Sie kennen mindestens zwei Techniken, um Daten aus einer Anwendung in ein Datagramm zu packen und aus einem Datagramm zu lesen

9.3. Aufbau

In der Einleitung haben wir bereits kurz erklärt, was die Unterschiede zwischen TCP und UDP sind. Als nächstes werden wir uns mit dem Aufbau eines Datagrammes befassen, die Datagramm Konstruktoren kennen lernen und einzelne Methoden anschauen. Zudem lernen Sie, wie Daten aus einer Applikation an ein Datagramm weitergegeben werden können und umgekehrt. Wesentlich in diesem Zusammenhang ist die Umwandlung der Daten von und in Byte Arrays.
Die DatagramSocket Klasse gestattet es, Verbindungen mit Hilfe von Sockets unter der Verwendung des UDP Protokolls aufzubauen.
Wie üblich werden wir abschliessend einige komplexere Beispiele anschauen, um uns mit der Technik und den darunter legenden Konzepten vertraut zu machen.

9.4. java.net Class DatagramPacket

```
java.lang.Object
   |
   +--java.net.DatagramPacket
```

9.4.1. Lernziele

- Sie kennen die Methoden und Konstruktoren der Datagramm Klasse.
- Sie wissen, welche Faktoren mitbestimmend sind für die Wahl der optimalen Paketgrösse.
- Sie kennen den Aufbau eines Datagramm Headers
- Sie können Daten einem Datagramm zuordnen und aus einem Datagramm lesen
- Sie können Daten, die aus einem Datagramm stammen umwandeln
  Sie können Daten, die ein Datagramm transportieren soll, in das Byte Array des Datagrammes umwandeln.

9.4.2. public final class **DatagramPacket** extends Object

Diese Klasse repäsentiert ein Datagramm Paket.
Datagramm Pakete werden eingesetzt, um verbindungslose Lieferdienste zu implementieren. Jede Meldung wird von einer Maschine zur andern geleitet, auf Grund von Informationen, die im Paket enthalten sind. Mehrere Pakete, die von einer Maschine zur andern gesandt werden, können unterschiedliche Routen verwenden und können auch in einer beliebigen Reihenfolge ankommen.

Seit:

JDK1.0

9.4.3. Konstruktor Übersicht

DatagramPacket(byte[] buf, int length) Konstruiert ein Datagram Paket, welches Pakete der Länge length aufnehmen kann.	
DatagramPacket(byte[] buf, int length, InetAddress address, int port) Konstruiert ein Datagram Paket zum Versenden von Paketen der Länge length, an einen Port port auf dem spezifizierten Rechner address.	
DatagramPacket(byte[] buf, int offset, int length) Konstruiert ein DatagramPacket, um Pakete der Länge length auf zu nehmen und zwar ab Position offset des Buffers.	
DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port) Konstruiert ein Datagram Paket zum Versenden von Paketen der Länge length mit Offset ioffsetto zum Port port auf dem Host address.	

9.4.4. Methoden Übersicht

InetAddress	**getAddress**() Liefert die IP Adresse der Maschine, an die dieses Datagramm geschickt wird, oder von dem dieses Datagramm empfangen wurde.
byte[]	**getData**() Liefert die Daten, die empfangen oder gesandt wurden..
int	**getLength**() Liefert die Länge der Daten, die empfangen oder gesandt werden.
int	**getOffset**() Liefert die Startposition der Daten, die übermittelt werden, oder die empfangen wurden.
int	**getPort**() Liefert die Port Nummer des remote Hosts, an den dieses Datagramm gesendet werden oder von dem sie empfangen wurden.
void	**setAddress**(InetAddress iaddr)
void	**setData**(byte[] buf) Setzt den Daten Buffer für dieses Paket.
void	**setData**(byte[] buf, int offset, int length) Setzt den Daten Buffer für dieses Paket.
void	**setLength**(int length) Setzt die Länge für dieses Paket.
void	**setPort**(int iport)

Betrachten wir nun einzelne Konstruktoren und Methoden etwas genauer:

9.4.5. Konstruktoren

Ein Datagramm Konstruktor muss primär die Daten aufnehmen. Im eher unüblichen Fall, dass das Datagramm gezielt an einen Port eines Hosts gesandt wird, steht auch dieser Konstruktor zur Verfügung.

9.4.5.1. public DatagramPackage(byte buffer[ ], int length)

Dieser Konstruktor kreiert ein DatagrammPacket, welches Daten empfängt. buffer[ ] ist ein Bytearray, in welchem die Daten gespeichert werden. Wann immer ein Datagramm empfangen wird, werden die Daten in diesem Buffer abgespeichert, beginnend mit buffer[0], bis alle Daten gespeichert sind, oder die maximale Länge des Buffers, length, erreicht ist.

Falls diese Länge überschritten wird, dann wird eine IllegalArgumentException ausgelöst, die das Laufzeitsystem betrifft und im Programm nicht abgefangen werden muss.

Die maximale Länge des Buffers beträgt 64 Kilobyte (65'535 Bytes). Davon verbraucht der Header 8 Bytes für die Adresse, den UDP Header, und mindestens 20 Bytes (bis maximal 60 Bytes) für den IP Header. Das grösste Datenpaket kann also maximal 65'507 Bytes lang sein.

9.4.5.1.1. Optimale Paket Grösse

Es gibt unterschiedliche Faktoren, die bei der Wahl der Buffer Grösse berücksichtigt werden müssen:

- falls die Verbindung eher unzuverlässig ist,
  dann sollten die Pakete eher klein sein (es geht weniger verloren)
- falls die Kommunikation innerhalb eines eher zuverlässigen LAN's stattfindet,
  dann sollten die Pakete möglichst gross gewählt werden

Im Rahmen der Gigabit Ethernet Technologie wurde die Buffer Grösse über die 64 KB erweitert, um die hohen Übertragungsgeschwindigkeiten bewerkstelligen zu können.

Ein gängiger Kompromiss sind etwa 8 KB oder 8192 Bytes.

9.4.5.1.2. Programmbeispiel

```
//Titel:       public DatagramPacket(byte buffer[ ]¨, int length)
//Version:
//Copyright:   Copyright (c) 1999
//Autor:       J.M.Joller
//Firma:
//Beschreibung:  Konstruktion eines Datagramms
package Beispiel9_1;
import java.net.*;

public class KlasseBeispiel9_1 {

  public static void main(String args[]) {
    System.out.println("Kapitel 9 UDP Datagrams Beispiel 9_1");
    byte[] buffer = new byte[8096];
    DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
    System.out.println("Ende Kapitel");
  }
}
```

9.4.5.1.3. Selbsttest Aufgabe

Konstruieren Sie ein Datagramm, welches länger als 64 KB lang ist.
Welche Meldung produziert Ihr Programm? Vergleichen Sie dies mit den obigen Bemerkungen zur Paketgrösse.
Konstruieren Sie ein Datagramm, welches eine negative Länge oder die Länge Null hat.

9.4.5.2. public DatagrammPacket(byte buffer[ ], int length, InetAddress ia, int port)

Mit diesem Konstruktor kann man Pakete, Datagramme, von einem UDP Server zu andern Hosts, Clients, senden.

Das Datagramm wird als Byte Array gefüllt und an den Konstruktor übergeben. Die Anzahl Bytes, die gesendet werden sollen, stehen im Parameter length. Falls der Parameter grösser als die Länge des Buffer ist, wird eine Ausnahme geworfen, weil der Buffer dann nicht alle Daten aufnehmen könnte. Man darf aber DatagrammPakete bauen,die kürzer sind als der Buffer. In diesem Fall werden allerdings nur die ersten length Bytes über das Netzwerk versandt.

Die Port Adresse port ist der Port der Zieladresse.

Die InetAddress Adresse beschreibt den Zielhost.

Die Grösse des Datagramms haben wir im vorhergehenden Abschnitt bereits beschrieben (maximal 64 KB, einige Bytes gehen für Header Informationen verloren).

Hätte man Zugriff auf die IP Optionen, was man in Java nicht hat, dann könnten auch detailliertere Optionen der Paketbildung genutzt werden. Diese stehen aber in Java, das über IP aufsetzt, nicht zur Verfügung.

Die Daten im Buffer können auch modifiziert werden, nachdem das Datagramm Paket kreiert wurde.

Im folgenden Beispiel treffen wir auf eines der Grundprobleme des Zeichensatzes : die Zeichen werden im Unicode eingegeben, Java muss sie intern in Bytes umwandeln, in den Buffer stellen und schliesslich versenden.

9.4.5.2.1. Programmbeispiel

```
//Titel:       public DatagramPacket(byte buffer[ ]¨, int length, InetAddress ia, int port);
//Version:
//Copyright:   Copyright (c) 1999
//Autor:       J.M.Joller
//Firma:
//Beschreibung:  Beispiel 9_2
//Umwandlung einer Zeichenkette in ein Byte Array und
//senden dieses Arrays an einen fix einprogrammierten Host
package Beispiel9_2;
import java.net.*;
public class KlasseBeispiel9_2 {
  public static void main(String args[]) {
    System.out.println("Begin Beispiel 9_2 : DatagramPacket");
    String host = "localhost";
    int port = 7;
    if (args.length > 0) {
```

```
      host = args[0];
    } else System.out.println("Dem Programm kann man zwei Parameter uebergeben : host und
port");
    if (args.length > 1) {  // erstes Argument ist der Host; zweites der Port
      port = Integer.parseInt(args[1]);
    }
    System.out.println("Parameter : host="+host+" und port="+port);
    String s = "Das ist ein Test.";
    byte[] data = new byte[s.length()];
    // übertragen der Daten (String) in den Byte Buffer data
    s.getBytes(0, s.length(), data, 0);
    try {
      InetAddress ia = InetAddress.getByName(host);
      DatagramPacket dp = new DatagramPacket(data, data.length, ia, port);
    }    catch (UnknownHostException e) {}
    System.out.println("Ende Beispiel9_2");
  }
}
```

9.4.5.2.2. Ausgabe

```
Begin Beispiel 9_2 : DatagramPacket
Dem Programm kann man zwei Parameter uebergeben : host und port
Parameter : host=localhost und port=7
Ende Beispiel9_2
```

9.4.6. Die Methoden

Mit Hilfe der Methoden kann man die Datenfelder des Datagrammes abfragen. Die Methoden sind also vom "GET" Typus : sie lesen nur Informationen, ohne diese Setzen zu können.

Was konkret gelesen werden kann, ergibt sich aus dem Aufbau des Datagrammes.

- Jedes Datagramm startet mit einem IP Header. Dieser enthält die Adresse des Rechners, von dem das Datagramm stammt, sowie die Adresse, an die das Paket adressiert ist.

- Dem IP Header folgt ein UDP Header. Dieser ist 8 Bytes lang und im RFC768 beschrieben.
  1. Die ersten zwei Bytes sind (positive) ganze Zahlen (unsigned Integer), mit der Port Adresse, von der das Datagramm gesendet wurde (Senderport).
  2. Die zweiten zwei Bytes (wieder unsigned Integer) enthalten den Port, an den das Datagramm gesendet werden (Receiverport).
  3. Die dritten zwei Bytes (wieder unsigned Integer) enthalten die Länge des Datagramms, inklusive Header.
  4. die vierten zwei Bytes (wieder unsigned Integer), die letzten zwei Bytes des UDP Headers, enthalten eine Prüfsumme (Checksumme), mit denen die Übermittlungsqualität geprüft werden kann. Oft sind diese zwei Bytes aber auf Null gesetzt!
- nach den zwei Headern folgt der eigentliche Datagramm Inhalt.

Soweit der Aufbau des Datagramms.
Und nun zurück zu den Methoden :

9.4.6.1. public InetAddress getAddress()

Das InetAddress Objekt ist die Adresse des remote Hosts.

- Falls das Datagramm vom Internet empfangen wurde, dann handelt es sich um die Adresse der Maschine, die das Datagramm versandt hat.
- Falls das Datagramm lokal kreiert wurde, um an einen remote Host geschickt zu werden, dann handelt es sich um die Adresse, an den das Datagramm adressiert ist.

Die Methode kann also zwei unterschiedliche Enden der Kommunikation beschreiben. Um welches es sich handelt, ergibt sich aus dem Kontext.

In der Regel wird man die Methode einsetzen, um fest zu stellen, von welchem Host das Datagramm stammt, um diesem Host eine Antwort senden zu können.

9.4.6.2. public int getPort()

Der Port, den diese Methode zurück gibt, ist der Port des remote Hosts, vom dem das Datagramm stammt.

- falls das Datagramm vom Internet stammt, dann handelt es sich um den Port des Hosts, der das Datagramm versandt hat.
- falls das Datagramm lokal kreiert wurde, um an einen remote Host geschickt zu werden, dann handelt es sich um den Port, an den das Datagramm beim remote Host gesandt werden soll.

9.4.6.3. public byte[ ] getData()

Die Methode liefert die Daten des Datagrammes, in Form eines Byte Arrays. Normalerweise müssen dann die Daten konvertiert werden, zum Beispiel in eine Zeichenkette.

Die kann man zum Beispiel mit Hilfe des Konstruktors erledigen:

9.4.6.3.1. Programm Fragment

```
...
public String(byte[ ] buffer, int high_byte, int start, int num_byte)
...
```

Das erste Argument, der Buffer, ist ein Datenfeld von Bytes, welches die Daten des Datagramms enthält.

Dieser Konstruktor setzt voraus, dass die Daten ASCII Zeichen enthält, also jedes Byte ein ASCII Zeichen darstellt.

Da in Java alle Zeichen in Unicode dargestellt werden, muss ein extra Byte pro Zeichen hinzu gefügt werden. Dies ist das high_byte Argument. Es wird einfach auf Null (0) gesetzt.

Die Startposition, also die Position im Byte Array ab der konvertiert werden soll, wird als Integer start angegeben. Alle Bytes vor array[start] werden ignoriert

Um zum Schluss der num_byte Parameter : dieser gibt die Anzahl Bytes an, die konvertiert werden sollen.

Wie viele Daten im Datagramm stehen, kann man mit der Methode getLength feststellen, wie wir gleich sehen werden.

Da ein Datagramm dg nicht "gefüllt" sein muss, ist es ratsam, jeweils bei der Konvertierung die Länge zu bestimmen. Ein Programm könnte dann etwa folgenden Programm Code enthalten:

```
String s = new String( dg.getData(), 0, 0, dg.getLength() );
```

Falls die Zeichenkette einen speziellen Aufbau hat, dann kann sie mit Hilfe des Tokenizers analysiert werden.

Falls die Daten im Byte Buffer keine Zeichenkette ist, dann ist die Konvertierung inJava etwas aufwendiger.

Hier eine mögliche Methode:

1. konvertieren des Byte Arrays in einen ByteArrayInputStream

   ```
   public ByteArrayInputStream( byte[ ] buffer, int offset, int num_bytes)
   ```

   wobei mit offset die Start Position und mit num_bytes die Anzahl Bytes, die berücksichtigt werden sollen, spezifiziert werden.

   Hier ein Beispiel mit Feststellen der Länge des Buffers:

   ```
   ByteArrayInputStream bis = new ByteArrayInputStream(dg.getData(), 0, dg.getLength());
   ```

2. Umwandlung in einen DataInputStream

   ```
   DataInputStream dis = new DataInputStream(bis);
   ```

   Damit kann man die Daten mit Hilfe der DataInput Methoden readInt(), readLong(), readChar() , readLine() und andern Methoden die Daten lesen.

Dabei muss allerdings vorausgesetzt werden, dass die Datenübermittlung vom Sender zum Empfänger in einem Standard Datenformat erfolgt (ASCII, Unicode, ...).

9.4.6.4. public int getLength()

Die Methode liefert die Anzahl Bytes im Datagramm. Falls das Datagramm aus dem Netzt stammt, dann sollte die Länge, die mit Hilfe von getLength() bestimmt wird, identisch sein mit der Länge des Byte Arrays, also getData().length.

Falls das Datagramm lokal kreiert wurde, dann können die zwei Längen unterschiedlich sein.

9.4.6.4.1. Programmbeispiel

```
//Titel:       Beispiel zu den get Methoden der UDP Datagramm Klassen
//Version:
//Copyright:   Copyright (c) 1999
//Autor:       J.M.Joller
//Firma:
//Beschreibung:  Definition eines einfachen Datagrammes
//Zuordnung einer Zeichenkette an das Datgramm
//"Versenden" des Datagrammes an einen Host und Port, welcher als Parameter übergeben wird
//Analyse des Datagrammes mit Hilfe der get Methoden
package Beispiel9_3;
import java.net.*;

public class KlasseBeispiel9_3 {

  public static void main(String args[]) {
    System.out.println("Beginn Kapitel 9 Beispiel 9_3");
    String s = "Hier stehen die Daten.";
    // Argumente
    String host = "localhost";
    int port = 7;
    if (args.length > 0) {
      host = args[0];
    } else System.out.println("Dem Programm kann man zwei Parameter uebergeben : host und
port");
    if (args.length > 1) {  // erstes Argument ist der Host; zweites der Port
      port = Integer.parseInt(args[1]);
    }
    System.out.println("Parameter : host="+host+" und port="+port);
    // Byte Array für das Datagramm
    byte[] data = new byte[s.length()];
    s.getBytes(0, s.length(), data, 0); // jetzt steht s im Byte Array
    try {
      InetAddress ia = InetAddress.getByName(host);
      DatagramPacket dp = new DatagramPacket(data, data.length, ia, port);
      System.out.println("Dieses Paket ist adressiert an " + dp.getAddress() + " on port " +
dp.getPort());
      System.out.println("Es sind " + dp.getLength() + " Daten Bytes im Paket");
      System.out.print(" ... und hier die Rückübersetzung der Datagramm Daten:");
```

```
      System.out.println(new String(dp.getData(), 0, 0, dp.getLength()));
    }
    catch (UnknownHostException e) {
      System.err.println(e);
    }
    System.out.println("Ende Beispiel 9_3");
  }
}
```

9.4.6.4.2. Ausgabe

```
Beginn Kapitel 9 Beispiel 9_3
Dem Programm kann man zwei Parameter uebergeben : host und port
Parameter : host=localhost und port=7
Dieses Paket ist adressiert an localhost/127.0.0.1 on port 7
Es sind 22 Daten Bytes im Paket
 ... und hier die Rückübersetzung der Datagramm Daten:Hier stehen die Daten.
Ende Beispiel 9_3
```

Die Ausgabe muss man kaum interpretieren. Die Zeichenkette hat die Länge 22, genau wie der Byte Buffer. Die Konversionen Zeichenkette -> Byte Array und Byte Array -> Zeichenkette waren offensichtlich erfolgreich, das heisst die Unicode - > ASCII -> Unicode Umwandlung scheint zu funktionieren.

Die getBytes gilt als veraltet, weil sie Plattform abhängig ist. Neu würde man die Methode getBytes(String enc) verwenden. Daher auch die Warnung in JBuilder. Das Gleiche gilt für die andere Form der Umwandlung von Bytes in eine Zeichenkette.

Ich werde mich wohl demnächst mit der Internationalisierung, dem Encoding und all diesem Kleinkram beschäftigen müssen.

Im Java Tutorial und in der JDK1.2 Dokumentation finden Sie Hinweise auf Konvertierungshilfen und Werkzeuge.

9.5. Die DatagramSocket Klasse

9.5.1. Lernziele

- Sie kennen die Methoden und Konstruktoren der DatagramSocket Klasse.
- Sie kennen die Systemparameter der Socket Klasse
- Sie können Daten mit Hilfe eines DatagramSockets versenden und empfangen.

9.5.2. Einführung

Damit man ein DatagramPacket senden oder empfangen kann, muss man einen DatagramSocket öffnen. Diesen muss man natürlich zuerst als Objekt, als Instanz der Klasse DatagramSocket kreieren.

Im Gegensatz zu TCP Sockets unterscheidet man nicht Client und Server Sockets.
Falls man einen Datagramm Client konstruiert, dann braucht man sich keine Gedanken über Ports zu machen. Das System kann sich einen freien Port aussuchen. Der Socket ist anonym.

Falls man einen Datagramm Server konstruiert, dann sollte man den Port explizit angeben, damit man dem Client mitteilen kann, auf welchem Port er mit dem Server kommunizieren kann.

Schauen wir uns diese Klasse genauer an.

9.5.3. java.net Class DatagramSocket

```
java.lang.Object
   |
   +--java.net.DatagramSocket
```

Direkt bekannte Unterklassen:
MulticastSocket

---

public class **DatagramSocket**
extends Object
Diese Klasse repräsentiert einen Socket für das Senden und Empfangen von Datagramm Paketen. Ein Datagramm Socket ist ein Start- oder Zielpunkt für eine Paketvermittlung. Jedes gesendete oder empfangene Paket wird individuell adressiert und geroutet. Die Wege der Pakete von einer Maschine zur andern Maschine können sich von einem zum nächsten Paket unterscheiden. Die Pakete können auch in einer falschen Reihenfolge ankommen. UDP Broadcasting Senden und Empfangen ist immer enabled bei einem DatagramSocket.

Seit:
JDK1.0

Siehe Auch:
`DatagramPacket`

9.5.3.1. Konstruktor Übersicht

DatagramSocket() Konstruiert einen Datagramm Socket und bindet ihn an irgend einen verfügbaren Port auf der lokalen Host Maschine.
DatagramSocket(int port) Konstruiert einen Datagramm Socket und bindet ihn an den spezifizierten Port auf der lokalen Host Maschine.
DatagramSocket(int port, InetAddress laddr) Kreiert einen Datagramm Socket und bindet ihn an die spezifizierte lokale Adresse.

9.5.3.2. Methoden Übersicht

void	**close**() Schliesst diesen Datagramm Socket.
void	**connect**(InetAddress address, int port) Verbindet den Socket mit einer remote Adresse für diesen Socket.
void	**disconnect**() Bricht die Verbindung des Sockets ab.
InetAddress	**getInetAddress**() Liefert die Adresse mit der dieser Socket verbunden ist.
InetAddress	**getLocalAddress**() Liefert die lokale Adresse an die der Socket gebunden ist.
int	**getLocalPort**() Liefert die Port Nummer auf dem lokalen Host an den dieser Socket gebunden ist.
int	**getPort**() Liefert den Port für diesen Socket.
int	**getReceiveBufferSize**() Liefert den Wert der SO_RCVBUF Option für diesen Socket, die Empfangs-Buffer Grösse für diesen Socket auf dieser Plattform.
int	**getSendBufferSize**() Liefert den Wert der SO_SNDBUF Option für diesen Socket, die Sende-Buffer Grösse für diesen Socket auf dieser Platform.
int	**getSoTimeout**() Liefert den Wert der SO_TIMEOUT Option.
void	**receive**(DatagramPacket p) Empfängt ein Datagramm Paket von diesem Socket.
void	**send**(DatagramPacket p) Sendet ein Datagramm Paket von diesem Socket.
void	**setReceiveBufferSize**(int size) Setzt die SO_RCVBUF Option auf den spezifizierten Wert für diesen DatagramSocket.
void	**setSendBufferSize**(int size) Setzt die SO_SNDBUF Option auf den spezifizierten Wert für diesen DatagramSocket.
void	**setSoTimeout**(int timeout) Enable/disable SO_TIMEOUT mit der spezifizierten timeout, in

	Millisekunden.

Schauen wir uns nun die Konstruktoren im Detail an.

9.5.3.3. public DatagramSocket() throws SocketException

Dieser Konstruktor kreiert einen DatagrammSocket, der an einen anonymen Port gebunden ist. Der Zielport ist Teil des Datagrammes, nicht des Sockets.

9.5.3.3.1. Anwendungsfall

Dieser Konstruktor wird als Client eingesetzt. Damit kann eine Kommunikation mit einem Server initialisiert werden. Es spielt keine Rolle. über welchen Port die Kommunikation aufgebaut wird. Der Server wird den Port, über den er erreichbar ist in seinem Datagramm festhalten und mitsenden. Den eigenen Port kann man mit getPort() erfragen.

Der selbe Socket wird auch eingesetzt, um Daten zu empfangen.

Falls der Socket nicht kreiert werden kann, dann wird eine SocketException geworfen. Das dürfte aber kaum je passieren, da ja der Port systemseitig frei wählbar ist.

Schauen wir uns zur Illustration ein Beispiel an.

9.5.3.3.2. Programmbeispiel

```
//Titel:       public DatagramSocket() throws SocketException
//Version:
//Copyright:   Copyright (c) 1999
//Autor:       J.M.Joller
//Firma:
//Beschreibung:  dies ist ein einfacher Test des
//anonymen Konstruktors der DatagramSocker Klasse
package Beispiel9_4;
import java.net.*;
public class KlasseBeispiel9_4 {
  public static void main(String[] args) {
    System.out.println("Beginn Programm Beispiel 9_4");
    try {
      DatagramSocket theClient = new DatagramSocket();
    }
    catch (SocketException e) {
       System.err.println("Beim Kreieren des anonymen DatagramSockets ist ein Fehler aufgetreten
: "+e);
    }
    System.out.println("Ende Beispiel 9_4");
  }
}
```

9.5.3.3.3. Ausgabe

```
Beginn Programm Beispiel 9_4
Ende Beispiel 9_4
```

9.5.3.4. public DatagramSocket(int port) throws SocketException

Dieser Konstruktor kreiert einen DatagramSocket, der auf ankommende Datagramme hört, an einem spezifischen Port port.

Mit diesem Konstruktor können Datagramm Server geschrieben werden, da der Port bekannt ist.

Ausnahmen können geworfen werden, weil:
1. der Port bereits besetzt ist
2. Sie versuchen, den Socket an einen Port < 1024 zu binden, ohne die nötigen Privilegien zu besitzen.

TCP und UDP Ports hängen nicht direkt zusammen. Es ist also möglich, gleichzeitig einen UDP Server und einen TCP Server an Port 2012 zu binden. Die unterschiedlichen Protokolle reichen für eine Differenzierung aus.

Schauen wir uns einmal einen UDP Port Scanner an, analog zum Beispiel zum TCP Port Scanner.

9.5.3.4.1. Programmbeispiel

```
//Titel:       public DatagramSocket(int port) throws SocketException
//Version:
//Copyright:    Copyright (c) 1999
//Autor:       J.M.Joller
//Firma:
//Beschreibung:  UDP Port Scanner
//Kapitel 9 Beispiel 5
package Beispiel9_5;
import java.net.*;

public class KlasseBeispiel9_5 {
  public static void main(String[] args) {
    System.out.println("Start Beispiel 9_5");
    DatagramSocket udpServer;
    for (int i = 0; i <= 1024; i++) {
      try {
        // falls der port besetzt ist, tritt eine Exception auf
        udpServer = new DatagramSocket(i);
        udpServer.close();
        //System.out.println("Port "+i+" ist noch frei für UDP.");
      }
      catch (SocketException e) {
        System.out.println("Port " + i + " ist schon besetzt.");
      } // end try
    } // end for
    System.out.println("Ende Beispiel 9_5");
  }
}
```

9.5.3.4.2. Ausgabe

```
Start Beispiel 9_5
Port 135 ist schon besetzt.
Ende Beispiel 9_5
```

beziehungsweise, falls die Kommentarzeile aktiviert wird:

```
Start Beispiel 9_5
Port 0 ist noch frei für UDP.
Port 1 ist noch frei für UDP.
Port 2 ist noch frei für UDP.
Port 3 ist noch frei für UDP.
Port 4 ist noch frei für UDP.
Port 5 ist noch frei für UDP.
Port 6 ist noch frei für UDP.
Port 7 ist noch frei für UDP.
Port 8 ist noch frei für UDP.
Port 9 ist noch frei für UDP.
Port 10 ist noch frei für UDP.
Port 11 ist noch frei für UDP.
Port 12 ist noch frei für UDP.
Port 13 ist noch frei für UDP.
Port 14 ist noch frei für UDP.
...
```

Sie werden unterschiedliche Ports aktiviert sehen, falls Sie das Programm auf Unix, Linux, ... laufen lassen, da zum Beispiel das Sun Network Filesystem NFS mit UDP arbeitet und normalerweise auf den Solaris Rechnern installiert ist.

9.5.3.5. public DatagramSocket(int port, InetAddress intf) throws SocketException

Mit diesem Konstruktor kann man auf Rechnern, die mehrere Web Sites verwalten (Multihomed Hosts) die Datenpakete besser zuordnen:

intf ist ein InetAddress Objekt, welches zu diesem Multihomes Host gehört.

Die Ausnahme wird geworfen, falls der Port besetzt ist, oder der Benutzer nicht das Recht hat, auf diesem Port zu kommunizieren.

9.5.4. Senden und Empfangen von Datagrammen

Nachdem wir gelernt haben, wie man Datagramme lesen und schreiben kann, wie man DatagramSockets kreieren kann, ist es an der Zeit, Pakete zu senden und zu empfangen!

Die Socket Klasse verfügt über eine send() Methode, mit deren Hilfe die zuvor konstruierten und "geladenen" Datagramme versendet werden können.

9.5.4.1. Lernziele

- Sie können einfache Datagramme senden und empfangen
- Sie können einfache UDP Clients und Server schreiben, testen und einsetzen

9.5.4.2. public void send(DatagramPacket dp) throws IOException

Die Methode gehört zur Klasse DatagramSocket! Sie gestattet das Senden von vorher konstruierten Datagrammen mit Hilfe eines DatagramSockets mit Hilfe des UDP Protokolls.

9.5.4.2.1. Programm Fragment

```
udpSocket.send(dgPaket);
```

Schauen wir uns ein etwas umfangreicheres Beispiel an. In diesem Beispiel werden alle ankommenden Datagramme ignoriert. Es wird nur geschrieben!

9.5.4.2.2. Programmbeispiel

```
//Titel:       public void send(DatagramPaket dp) throws IOException
//Version:
//Copyright:   Copyright (c) 1999
//Autor:       J.M.Joller
//Firma:
//Beschreibung:  ein einfacher UDP Client / Server
//Lesen der Daten ab System.in (Konsole)
//Ausgabe der Daten als Datagramm über UDP.
package Beispiel9_6;
import java.net.*;
import java.io.*;

public class KlasseBeispiel9_6 {
  public static void main(String[] args) {
    System.out.println("Start Beispiel 9_6");
    String host;
    int port;

    if (args.length > 0) {
      host = args[0];
    }
    else {
      System.out.println("Das Programm hätte zwei Parameter : host und port");
```

```
      host = "localhost";
    }
    if (args.length > 1) {
      port = Integer.parseInt(args[1]);
    }
    else {
      port = 2048;
    }
    System.out.println("Host = "+host+"; Port = "+port);
    System.out.println("Geben Sie die zu uebermittelnden Daten ein\nAbbruch : durch Eingabe von
'.'");

    try {

      String strEingabe;
      DatagramPacket dpOutput;

      InetAddress iaServer = InetAddress.getByName(host);
      DataInputStream userInput = new DataInputStream( System.in);
      DatagramSocket dsSocket = new DatagramSocket();
      while (true) {
        strEingabe = userInput.readLine();
        if (strEingabe.equals(".")) break; //  ich versuch's nochmal
        byte[] data = new byte[strEingabe.length()]; // Datagramm
        strEingabe.getBytes(0, strEingabe.length(), data, 0);
        dpOutput = new DatagramPacket(data, data.length, iaServer, port);
        dsSocket.send(dpOutput);
        System.out.println("Die Eingabedaten '"+strEingabe+"' wurden per UDP versandt! Abbruch :
'.' eingeben");
      }  // end while
    }  // end try
    catch (UnknownHostException e) {
      System.err.println(e);
    }
    catch (SocketException se) {
      System.err.println(se);
    }
    catch (IOException e) {
      System.err.println(e);
    }
    System.out.println("Ende Beispiel 9_7");
  }  // end main
}
```

9.5.4.2.3. Ausgabe

Die hängt von Ihnen ab : Sie müssen das Programm ohne Umleitung der Ausgabe ins Ausführungsprotokoll starten.

9.5.4.3. public void receive(DatagrammPacket dp) throws IOException

Jetzt wollen wir auch mal Daten empfangen, lesen und analysieren!
Die receive Methode liefert uns diese Funktion. Sie funktioniert ähnlich wie die accept() Methode der ServerSocket Klasse, das heisst, sie wartet einfach unendlich lange, bis endlich ein Datagramm angeflogen kommt, welches für diesen Host bestimmt ist. Sinnvollerweise würden wir also auch hier wieder mit Threads arbeiten.

Damit die ankommenden Daten auch sicher Platz finden im Buffer, müssen wir diesen gross genug wählen. NFS, das Network File System der Sun, verwendet Buffer der Grösse 8192 Bytes. Die maximale Grösse des Buffers ist 65'507 = 65'535 - 8 Byte UDP Header - 20 Byte IP Header.

Falls die Übertragung schief läuft, kann eine IOException geworfen werden.

Im folgenden Beispiel empfängt der Socket die Daten (als UDP Server) und schreibt sie auf die Konsole, wobei hoffentlich nur lesbare Daten empfangen werden. Das könnte man natürlich noch prüfen....

9.5.4.3.1. Programmbeispiel

```
//Titel:       public void receive(DatagrammPacket dp) throws IOException
//Version:
//Copyright:   Copyright (c) 1999
//Autor:       J.M.Joller
//Firma:
//Beschreibung:  Dieser "discard" Server hört an einem UDP Port
//(Default Port = 9)
//und schreibt die empfangenen Daten auf die Konsole.
package Beispiel9_7;
import java.net.*;
import java.io.*;

public class KlasseBeispiel9_7 {
  static byte[] buffer = new byte[65507];

  public static void main(String[] args) {
    System.out.println("Start Beispiel 9_7");
    String host;
    int port;

    if (args.length > 0) {
      host = args[0];
    }
    else {
      System.out.println("Das Programm hätte zwei Parameter : host und port");
      host = "localhost";
    }
```

```
    if (args.length > 1) {
      port = Integer.parseInt(args[1]);
    }
    else {
      port = 2048;
    }
    System.out.println("Host = "+host+"; Port = "+port);
    System.out.println("Der Server wartet auf Daten");

    try {
      DatagramSocket ds = new DatagramSocket(port);
      while (true) {
        try {
          DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
          ds.receive(dp);
          String s = new String(dp.getData(), 0, 0, dp.getLength());
          System.out.println(dp.getAddress() + " am Port " + dp.getPort() + " sagt " + s);
        }
        catch (IOException e) {
          System.err.println(e);
        }
       } // end while
    }  // end try
    catch (SocketException se) {
      System.err.println(se);
    }  // end catch
    System.out.println("Ende Beispiel 9_7");
  }  // end main
}
```

9.5.4.3.2. Ausgabe

Diese hängt vom Umfeld ab: was empfangen wird, wird auch angezeigt. Hier der Start des Programmes :

```
Start Beispiel 9_7
Das Programm hätte zwei Parameter : host und port
Host = localhost; Port = 2048
Der Server wartet auf Daten
```

9.5.4.3.3. Verbesserungsvorschlag

Das Ausdrucken des Echo's verbraucht sehr viel Zeit. Ein effizienter Server würde auf diese Funktion sicher verzichten!

9.5.4.3.4. Selbsttestaufgabe

Testen Sie (zu zweit oder zu mehreren) den Server, in dem Sie in der Gruppe einen Server und mehrere Clients starten.
Versuchen Sie auch mehrere Server und mehrere Clients zu starten.

9.5.4.4. public int getLocalPort()

Diese DtagramSocket Methode liefert den Port, über den der Socket kommuniziert. Diese Methode benötigt man speziell, falls man anonyme Ports definiert, also das System den ersten freien Port wählen lässt.

9.5.4.4.1. Programmbeispiel

```
//Titel:       public getLocalPort()
//Version:
//Copyright:   Copyright (c) 1999
//Autor:       J.M.Joller
//Firma:
//Beschreibung:  Liefert den lokalen Port eines DatagramSocket
package Beispiel9_8;
import java.net.*;

public class KlasseBeispiel9_8 {
  public static void main(String[] args) {
    System.out.println("Start Beispiel 9_8");
    try {
      DatagramSocket ds = new DatagramSocket();
      System.out.println("Der DatagramSocket ist an Port " + ds.getLocalPort());
    }
    catch (SocketException e) {
    }
    System.out.println("Ende Beispiel 9_8");
  }
}
```

9.5.4.4.2. Ausgabe

Hier eine der vielen möglichen Ausgaben (der Port wird zufällig aus den freien Ports zugeteilt):

```
Start Beispiel 9_8
Der DatagramSocket ist an Port 1038
Ende Beispiel 9_8
```

9.5.4.5. public synchronized void close()

Mit dieser Methode kann ein DatagramSocket wieder geschlossen werden, keine schlechte Idee!

9.5.4.5.1. Programm Fragment

```
...
try {
        DatagramSocket dgSocket = new DatagramSocket();
        ...
        dgSocket.close();
        ...
} ...
```

9.5.5. Socket Optionen

Die UDP / DatagramSocket kennen die gleichen Socket Optionen, wie die TCP Sockets.
Diese Parameter haben wir bereits besprochen. Hier nur zur Wiederholung:

1. SO_TIMEOUT :
   so lange wartet receive() auf Pakete;
   falls kein Paket eintrifft, wird eine InterruptedIOException geworfen.
   falls SO_TIMEOUT = 0 ist, wartet receive() beliebig lange.
2. andere Socket Parameter:
   diese werden über die DatagramSocketImpl Klasse implementiert, wie man aus folgendem Diagramm ersehen kann.

java.net Class DatagramSocketImpl :

```
java.lang.Object
  |
  +--java.net.DatagramSocketImpl
```

public abstract class DatagramSocketImpl
extends Object
implements SocketOptions

java.net Interface SocketOptions
kennt folgende Optionen / Felder :

```
static int      IP_MULTICAST_IF
static int      SO_BINDADDR
static int      SO_LINGER
static int      SO_RCVBUF
static int      SO_REUSEADDR
static int      SO_SNDBUF
static int      SO_TIMEOUT
static int      TCP_NODELAY
```

Die Parameter sind im Einzelnen in der Dokumentation von JDK beschrieben, und Sie eine Beschreibung und Diskussion der obigen Parameter im Kapitel über ServerSockets.

9.6. Einige komplexere Applikationen

Jetzt wollen wir einige komplexere Clients und Server bauen, und mit Datagrammen kommunizieren.

Viele einfache Internet Protokolle besitzen TCP und UDP Implementationen. Immer wenn ein IP Paket bei einem Host ankommt, bestimmt der Host auf Grund des Headers, ob es sich um TCP Paket oder ein UDP Datagramm handelt. Da sich UDP und TCP grundlegend unterscheiden, können beide die Ports teilen, ohne dass ein Konflikt eintritt. Sie erkennen dies auch in der Liste der Ports und der dort verfügbaren Protokolle und Services (separates Dokument).

Zum Beispiel:

```
...
daytime             13/tcp
daytime             13/udp
...
qotd                17/tcp    quote  //quote of the day
qotd                17/udp    quote
...
```

Es gibt also offensichtlich einen UDP und einen TCP daytime Service, jeweils an Port 13.

9.6.1. Ein einfacher UDP Client

Die Protokolle Daytime, Quote of the Day, Chargen funktionieren vereinfacht gesagt auf folgende Weise:

- der Server sendet an die Adresse des Clients (Port und Adresse) bestimmte Informationen
- der Server ignoriert alle Daten, die an ihn gesandt werden

- der Client sendet ein Datagramm an den Server
- der Server antwortet, indem er eine Standardantwort sendet
- der Client liest die Antwort

Da diverse Protokolle nach diesem Schema funktionieren, könnte man eine allgemeine Klasse definieren und die unterschiedlichen Clients als Unterklassen implementieren.

Als Parameter, protected, damit sie in Unterklassen überschrieben werden können, wählen wir den Port und die Bufferlänge (8192 Bytes).

9.6.1.1. Die Klasse

```
//Titel:       UDP Klasse
//Version:
//Copyright:   Copyright (c) 1999
//Autor:       J.M.Joller
//Firma:
//Beschreibung:  ein allgemeiner Client
package Beispiel9_10;
import java.net.*;
import java.io.*;
```

```
public class KlasseBeispiel9_10 {
  protected static int defaultPort = 0;   // Standard Port : anonym
  protected static int bufferLength = 8192; // NFS abgeschaut
  public static void main(String[] args) {
    System.out.println("Start Beispiel 9_10");
    String hostname;
    int port;
    int len;
    if (args.length > 0) {
      hostname = args[0];
    }
    else {
      hostname = "localhost";
      port = defaultPort;
      len = bufferLength;
    }
    try {       // zur Abwechselung mal eine neue Variante
      port = Integer.parseInt(args[1]);
    }
    catch (Exception e) {
      port = defaultPort;
    }
    try {
      len = Integer.parseInt(args[2]);
    }
    catch (Exception e) {
      len = bufferLength;
    }
    try {
      DatagramSocket ds = new DatagramSocket(0);
      InetAddress ia = InetAddress.getByName(hostname);
      DatagramPacket outgoing = new DatagramPacket(new byte[512], 1, ia, port);
      DatagramPacket incoming = new DatagramPacket(new byte[len], len);
      ds.send(outgoing);
      ds.receive(incoming);
      System.out.println(new String(incoming.getData(), 0, 0, incoming.getLength()));
    }  // end try
    catch (UnknownHostException e) {
      System.err.println(e);
    }  // end catch
    catch (SocketException e) {
      System.err.println(e);
    }  // end catch
    catch (IOException e) {
      System.err.println(e);
    }  // end catch
    System.out.println("Ende Beispiel 9_10");
  }  // end main
}
```

9.6.1.2. Ein Daytime Client

Jetzt verwenden wir die allgemeine Klasse, um einen Daytime Client zu konstruieren.

9.6.1.2.1. Programmbeispiel

```
//Titel:       Daytimen Client
//Version:
//Copyright:   Copyright (c) 1999
//Autor:       J.M.Joller
//Firma:
//Beschreibung:  extends ... client
package Beispiel9_11;

public class KlasseBeispiel9_11 extends KlasseBeispiel9_10 {
  protected static int defaultport = 13;
}
```

9.6.2. Ein UDP Server

Nachdem die Client Seite objektorientiert entwickelt werden konnte, folgt nun das Gleiche auf der Server Seite.

9.6.2.1. Die UDP Server Klasse

```
//Titel:       UDP Server Klasse
//Version:
//Copyright:   Copyright (c) 1999
//Autor:       J.M.Joller
//Firma:
//Beschreibung:  allgemeine Klasse für die Implementation von UDP Servern
package Beispiel9_12;
import java.net.*;
import java.io.*;

public class KlasseBeispiel9_12 {

  protected static int defaultPort = 0;
  protected static int defaultBufferLength = 65507;

  public static void main(String[] args) {
    System.out.println("Start Beispiel 9_12");
    DatagramPacket incoming;

    int port;
    int len;
```

```
    try {
      port = Integer.parseInt(args[0]);
    }
    catch (Exception e) {
      port = defaultPort;
    }
    try {
      len = Integer.parseInt(args[1]);
    }
    catch (Exception e) {
      len = defaultBufferLength;
    }

    try {
      DatagramSocket ds = new DatagramSocket(port);
      byte[] buffer = new byte[len];
      while (true) {
        incoming = new DatagramPacket(buffer, buffer.length);
        try {
          ds.receive(incoming);
          respond(ds, incoming);
          System.out.println("Datagramm empfangen und beantwortet");
        }
        catch (IOException e) {
          System.err.println(e);
        }
      } // end while
    }  // end try
    catch (SocketException se) {
      System.err.println(se);
    }  // end catch
    System.out.println("Ende Beispiel 9_12");
  }  // end main

  public static void respond(DatagramSocket ds, DatagramPacket dp) {
    ; // da müssen wir noch was hineinschreiben, damit wir einen Server erhalten
  }
}
```

9.6.2.2. Ein UDP Server

```
//Titel:        UDP Server
//Version:
//Copyright:    Copyright (c) 1999
//Autor:       J.M.Joller
//Firma:
//Beschreibung:  extends UDP Server Class
package Beispiel9_13;
```

```
public class KlasseBeispiel9_13 extends KlasseBeispiel9_12{
  protected static int defaultport = 9;
}
```

9.6.2.3. Ein UDP Server mit Ausgabe

```
//Titel:       ein UDP Server, der Datagramme auf die Konsole schreibt
//Version:
//Copyright:    Copyright (c) 1999
//Autor:       J.M.Joller
//Firma:
//Beschreibung:  extends UDP Server
package Beispiel9_14;
import java.net.*;

public class KlasseBeispiel9_14 extends KlasseBeispiel9_12 {

  static { defaultPort = 9; }

  public static void respond(DatagramSocket ds, DatagramPacket dp) {

    String s = new String(dp.getData(), 0, 0, dp.getLength());
    System.out.println(dp.getAddress() + " an Port " + dp.getPort() + " sagt " + s);

  }
}
```

9.6.2.4. Ein UDP Echo Server

```
//Titel:       UDP echo Server
//Version:
//Copyright:    Copyright (c) 1999
//Autor:       J.M.Joller
//Firma:
//Beschreibung:  extents KlasseBeispiel9_12
package Beispiel9_15;
import java.net.*;
import java.io.*;

public class KlasseBeispiel9_15 extends KlasseBeispiel9_12 {
  static { defaultPort = 7; }
  public static void respond(DatagramSocket ds, DatagramPacket dp) {
    DatagramPacket outgoing;
    try {
      outgoing = new DatagramPacket(dp.getData(), dp.getLength(),
                                    dp.getAddress(), dp.getPort());
      ds.send(outgoing);
    }
```

```
    catch (IOException e) {    System.err.println(e);    }
  }
}
```

9.6.2.5. Ein UDP Daytime Server

```
//Titel:        Daytime Server
//Version:
//Copyright:    Copyright (c) 1999
//Autor:        J.M.Joller
//Firma:
//Beschreibung:  extends KlasseBeispiel9_12
package Beispiel9_16;
import java.net.*;
import java.io.*;
import java.util.Date;

public class KlasseBeispiel9_16 extends KlasseBeispiel9_12 {

  static { defaultPort = 13; }

  public static void respond(DatagramSocket ds, DatagramPacket dp) {

    DatagramPacket outgoing;
    Date now = new Date();
    String s = now.toString();
    byte[] data = new byte[s.length()];
    s.getBytes(0, s.length(), data, 0);
    try {
      outgoing = new DatagramPacket(data, data.length, dp.getAddress(), dp.getPort());
      ds.send(outgoing);
    }
    catch (IOException e) {
      System.err.println(e);
    }
  }
}
```

9.6.3. UDP Client mit mehreren Threads

Im Gegensatz zu einem TCP Client kann der UDP Client nicht eine fixe Verbindung zu seinem Server aufbauen. Die Verbindung ist "unzuverlässig".

Unter Umständen müssen Daten erneut übermittelt werden. Dies geschieht beim UDP Protokoll nicht automatisch, wie beim TCP. Beim UDP Protokoll liegt es in der Verantwortung der Kommunikations-Partner dafür zu sorgen, dass die Kommunikation zuverlässig wird.

Eine übliche Variante dieses Problem anzugehen, besteht darin, je einen Thread für Eingabe und Ausgabe zu konstruieren.

Die Architektur eines solchen Clients besteht also aus drei Klassen:

- Eingabe Thread Klasse
- Ausgabe Thread Klasse
- eigentliche Client Klasse

9.6.3.1. Programmbeispiel

9.6.3.1.1. Der Multithreading echoClient

```
//Titel:        Multi Threading UDP Client
//Version:
//Copyright:    Copyright (c) 1999
//Autor:       J.M.Joller
//Firma:
//Beschreibung:  EIngabe Thread
//Ausgabe Thread
//Main
package Beispiel9_17;
import java.net.*;
import java.io.*;

public class KlasseBeispiel9_17 {
  public static void main(String[] args) {
    System.out.println("Start Beispiel 9_17");
    String hostname="localhost";
    int echoPort = 7;

    if (args.length > 0) {
      hostname = args[0];
      if (args.length > 1) {
        echoPort =  Integer.parseInt(args[2]);
      }
      System.out.println("Programm Parameter : Host = "+hostname+"; Port = "+echoPort);
    }
    else {
```

```
      System.out.println("Das Programm hat zwei Parameter : Host und Port");
    }

    try {
       InetAddress ia = InetAddress.getByName(hostname);
       DatagramSocket theSocket = new DatagramSocket();
       echoEingabeThread eit = new echoEingabeThread(ia, theSocket);
       eit.start();
       echoAusgabeThread eot = new echoAusgabeThread(theSocket);
       eot.start();
    }
    catch (UnknownHostException e) {
      System.err.println(e);
    }
    catch (SocketException se) {
      System.err.println(se);
    }
    System.out.println("Ende Beispiel 9_17");
  }  // end main
}
```

9.6.3.1.2. Der Eingabe Thread

```
//Titel:       Input Thread
//Version:
//Copyright:    Copyright (c) 1999
//Autor:       J.M.Joller
//Firma:
//Beschreibung:  Multithreading echo Server :
//Eingabe Thread
//Beispiel 9_18 (gehört zu 9_17)
package Beispiel9_18;
import java.net.*;
import java.io.*;

public class KlasseBeispiel9_18 extends Thread {

  InetAddress server;
  int echoPort = 7;
  DatagramSocket theSocket;

  public KlasseBeispiel9_18(InetAddress ia, DatagramSocket ds) {

    server = ia;
    theSocket = ds;

  }
```

```
  public void run() {

    DataInputStream userInput;
    String theLine;
    DatagramPacket theOutput;

    try {
      userInput = new DataInputStream(System.in);
      while (true) {
        theLine = userInput.readLine();
        if (theLine.equals(".")) break;
        byte[] data = new byte[theLine.length()];
        theLine.getBytes(0, theLine.length(), data, 0);
        theOutput = new DatagramPacket(data, data.length, server, echoPort);
        theSocket.send(theOutput);
        Thread.yield();
      }

    }  // end try
    catch (IOException e) {
      System.err.println(e);
    }

  }  // end run
}
```

9.6.3.1.3. Der Ausgabe Thread

```
//Titel:        echoOutputThread
//Version:
//Copyright:    Copyright (c) 1999
//Autor:       J.M.Joller
//Firma:
//Beschreibung:  Ausgabe Thread des
//Multithreading echoServers
package Beispiel9_19;
import java.net.*;
import java.io.*;

public class KlasseBeispiel9_19 extends Thread {

  DatagramSocket theSocket;
  protected DatagramPacket dp;

  public KlasseBeispiel9_19(DatagramSocket s) {
    theSocket = s;
```

```
    byte[] buffer = new byte[65507];
    dp = new DatagramPacket(buffer, buffer.length);
  }

  public void run() {

    while (true) {
      try {
        theSocket.receive(dp);
        String s = new String(dp.getData(), 0, 0, dp.getLength());
        System.out.println(s);
        Thread.yield();
      }
      catch (IOException e) {
        System.err.println(e);
      }
    }
  }
}
```