Java Standard: Socket ServerSocket (java.net) UDP und TCP IP

Einleitung

Zur Erzeugung von Socket-Verbindungen stellt Java die beiden Klassen `java.net.Socket` und `java.net.ServerSocket` zur Verfügung.

Der folgende Code soll das Vorgehen verdeutlichen. Er enthält einen extrem primitiven Client und Server. Beim Server kann sich nur ein Client anmelden und nur einmal eine (kurze) Nachricht senden. Der Server sendet diese Nachricht dann zurück und beendet sich. Hat der Client die zurückgesendete Nachricht empfangen, beendet auch er sich. Hinweise für intelligentere Server und Clients finden sich weiter unten.

Ein primitiver Server

```
// Server.java

// import java.net.ServerSocket;
// import java.net.Socket;
import java.io.*;

public class Server {
   public static void main(String[] args) {
      Server server = new Server();
      try {
          server.test();
       } catch (IOException e) {
           e.printStackTrace();
       }
   }
   void test() throws IOException {
      int port = 11111;
      java.net.ServerSocket serverSocket = new
java.net.ServerSocket(port);
      java.net.Socket client = warteAufAnmeldung(serverSocket);
      String nachricht = leseNachricht(client);
      System.out.println(nachricht);
      schreibeNachricht(client, nachricht);
   }
   java.net.Socket warteAufAnmeldung(java.net.ServerSocket
serverSocket) throws IOException {
      java.net.Socket socket = serverSocket.accept(); // blockiert,
bis sich ein Client angemeldet hat
      return socket;
   }
   String leseNachricht(java.net.Socket socket) throws IOException {
       BufferedReader bufferedReader =
```

```
            new BufferedReader(
               new InputStreamReader(
                  socket.getInputStream()));
        char[] buffer = new char[200];
        int anzahlZeichen = bufferedReader.read(buffer, 0, 200); //
blockiert bis Nachricht empfangen
        String nachricht = new String(buffer, 0, anzahlZeichen);
        return nachricht;
    }
    void schreibeNachricht(java.net.Socket socket, String nachricht)
throws IOException {
        PrintWriter printWriter =
            new PrintWriter(
                new OutputStreamWriter(
                   socket.getOutputStream()));
        printWriter.print(nachricht);
        printWriter.flush();
    }
 }
```

Ein primitiver Client

```
 // Client.java

 // import java.net.Socket;
 import java.io.*;

 public class Client {
     public static void main(String[] args) {
        Client client = new Client();
        try {
            client.test();
        } catch (IOException e) {
            e.printStackTrace();
        }
     }
     void test() throws IOException {
        String ip = "127.0.0.1"; // localhost
        int port = 11111;
        java.net.Socket socket = new java.net.Socket(ip,port); //
verbindet sich mit Server
        String zuSendendeNachricht = "Hello, world!";
         schreibeNachricht(socket, zuSendendeNachricht);
        String empfangeneNachricht = leseNachricht(socket);
        System.out.println(empfangeneNachricht);
     }
     void schreibeNachricht(java.net.Socket socket, String nachricht)
throws IOException {
```

```
        PrintWriter printWriter =
           new PrintWriter(
              new OutputStreamWriter(
                 socket.getOutputStream()));
        printWriter.print(nachricht);
        printWriter.flush();
    }
    String leseNachricht(java.net.Socket socket) throws IOException {
        BufferedReader bufferedReader =
            new BufferedReader(
              new InputStreamReader(
                    socket.getInputStream()));
        char[] buffer = new char[200];
        int anzahlZeichen = bufferedReader.read(buffer, 0, 200); //
blockiert bis Nachricht empfangen
        String nachricht = new String(buffer, 0, anzahlZeichen);
        return nachricht;
    }
 }
```

Erklärung des Codes

Die Klasse `java.net.ServerSocket` dient in erster Linie zur Anmeldung von Clients. Sie hört nicht - wie man vielleicht erwarten könnte - auf die Nachrichten von Clients. Im Konstruktor wird der Port (im Beispiel 11111) übergeben (die IP-Adresse des Servers ist natürlich die IP-Adresse des Computers, auf dem der Server läuft). Die Methode `accept()` wartet so lange, bis sich ein Client verbunden hat. Dann gibt sie einen `java.net.Socket` zurück.

Die Klasse `java.net.Socket` hat zwei Funktionen:

- im Server dient sie dazu, sich den angemeldeten Client zu merken, auf seine Nachrichten zu hören und ihm zu antworten. Erzeugt wird eine Instanz durch die Methode `accept()`, wenn sich ein Client angemeldet hat.
- im Client erzeugt sie die Verbindung zum Server; dem Konstruktor wird die IP-Adresse des Servers und dessen Port übergeben. Mit dem Aufruf des Konstruktors im Client wird die Verbindung hergestellt - beim Server gibt `accept()` den verbundenen Client zurück.

Die Kommunikation läuft dann auf beiden Seiten gleich:
Mit `java.net.Socket#getInputSteam()` bzw. `java.net.Socket#getOutputSteam()` können die Nachrichten der anderen Seite gelesen werden bzw. dorthin gesendet werden. Die lesenden Methoden blockieren dabei so lange, bis eine Nachricht empfangen wurde.

Wie sähe ein intelligenter Server aus

Probleme bereiten die blockierenden Methoden; sowohl die Methode `java.net.ServerSocket#accept()` als auch die Methode `read()` (oder seine Verwandten) warten solange, bis sich ein Client angemeldet hat bzw. bis die Gegenseite eine Nachricht gesendet hat.

Auf der Seite des Servers wird es deshalb in der Regel einen eigenen Thread geben, der auf Anmeldungen von Clients wartet. Für jeden Client, der sich angemeldet hat, wird dann wiederum ein neuer Thread gestartet, der auf Nachrichten des Clients wartet. Daneben wird man in aller Regel den Client in einer Liste speichern, damit es möglich ist, allen angemeldeten Clients eine Nachricht zu senden.

Auch im Client gibt es einen eigenen Thread, der auf Nachrichten des Servers wartet. Während man es sonst bei Methodenaufrufen gewohnt ist, dass man auf den Rückgabewert (und sei es `void`) der Funktion wartet (synchron), verläuft die Kommunikation hier asynchron.

Eine weitere Schwierigkeit ist das Lesen von Nachrichten der Gegenseite. Im Beispiel werden Nachrichten der Länge 200 gelesen (das reicht für "Hello, world!"). Es hilft an dieser Stelle nicht wirklich, den Buffer deutlich größer zu machen (aus Performance-Gründen wird man ihn natürlich etwa auf 1024 Bytes vergrößern), Beim Lesen treten nämlich zwei Probleme auf:

- Es kommt vor, dass die Nachricht länger ist als der Buffer.
- Es kommt vor, dass mehrere Nachrichten (teilweise) im Buffer stehen.

Deshalb müssen Server und Client ein eigenes Protokoll vereinbaren. Denkbar (und üblich) sind dabei mehrere Varianten:

- Alle Nachrichten haben eine feste Länge
- Zu Beginn jeder Nachricht (z.B. in den ersten 8 Zeichen) wird die Länge der Nachricht angegeben; es werden dann zunächst diese 8 Zeichen eingelesen und ausgewertet und dann entsprechend viele Zeichen gelesen
- Jede Nachricht endet mit einem festen String, der sonst innerhalb der Nachrichten nicht vorkommt (z.B. "END"). Der InputStream wird solange gelesen, bis dieser String gefunden wurde.

Ein etwas komfortableres Client/Server-Beispiel

```
Server und Client verwenden das TCP-Protokoll.
- Server
  o Server-Threads
    1. main-Thread
    2. Thread zur Annahme von Client-Anforderungen.
    3. Pro Anforderung wird in Thread 2 ein neuer Thread aus einem Thread-Pool
       gestartet, der mit seinem Client kommuniziert und letztlich die
       Anforderung bearbeitet.
  o Threadpool
    Die Client-Threads werden in einem Pool verwaltet und bei neuen
    Anforderungen wiederverwendet.

- Client
  Client-Threads
    1. main-Thread
    2. Thread zur Kommunikation mit dem Server
    3. (irrelevanter) Thread zur Veranschaulichung der Nebenläufigkeit
  Im main-Thread (Methode 'worker') kann jederzeit abgefragt werden, ob die
  Antwort vom Server eingetroffen ist.
```

TCP-Server

Hinweis: Die Erklärung der Beispiele steht als Kommentar im Quelltext.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.io.*;
import java.net.*;
import java.util.Date;
import java.text.*;

public class ServerExample {
  /*
    - Server benutzt Port 3141,
      liefert zu einem Datum selbiges mit dem Wochentag an den Client
    - jede Client-Anforderung wird von einem Thread des erzeugten
Thread-Pools
      behandelt
    - Server-Socket kann mit Strg+C geschlossen werden oder vom Client
mit
      dem Wert 'Exit'.
  */
  public static void main(String[] args) throws IOException {
    final ExecutorService pool;
    final ServerSocket serverSocket;
    int port = 3141;
    String var = "C";
    String zusatz;
    if (args.length > 0 )
      var = args[0].toUpperCase();
    if (var == "C") {
      //Liefert einen Thread-Pool, dem bei Bedarf neue Threads
hinzugefügt
      //werden. Vorrangig werden jedoch vorhandene freie Threads
benutzt.
      pool = Executors.newCachedThreadPool();
      zusatz = "CachedThreadPool";
    } else {
      int poolSize = 4;
      //Liefert einen Thread-Pool für maximal poolSize Threads
      pool = Executors.newFixedThreadPool(poolSize);
      zusatz = "poolsize="+poolSize;
    }
    serverSocket = new ServerSocket(port);
    //Thread zur Behandlung der Client-Server-Kommunikation, der
Thread-
    //Parameter liefert das Runnable-Interface (also die run-Methode
für t1).
```

```
    Thread t1 = new Thread(new NetworkService(pool,serverSocket));
    System.out.println("Start NetworkService(Multiplikation), " +
zusatz +
                       ", Thread: "+Thread.currentThread());
    //Start der run-Methode von NetworkService: warten auf
Client-request
    t1.start();
//
    //reagiert auf Strg+C, der Thread(Parameter) darf nicht gestartet
sein
    Runtime.getRuntime().addShutdownHook(
      new Thread() {
        public void run() {
              System.out.println("Strg+C, pool.shutdown");
              pool.shutdown();  //keine Annahme von neuen Anforderungen
          try {
                //warte maximal 4 Sekunden auf Beendigung aller
Anforderungen
            pool.awaitTermination(4L, TimeUnit.SECONDS);
            if (!serverSocket.isClosed()) {
              System.out.println("ServerSocket close");
              serverSocket.close();
            }
          } catch ( IOException e ) { }
          catch ( InterruptedException ei ) { }
            }
      }
    );
//
  }
}

//Thread bzw. Runnable zur Entgegennahme der Client-Anforderungen
class NetworkService implements Runnable { //oder extends Thread
  private final ServerSocket serverSocket;
  private final ExecutorService pool;
  public NetworkService(ExecutorService pool,
                        ServerSocket serverSocket) {
    this.serverSocket = serverSocket;
    this.pool = pool;
  }
  public void run() { // run the service
    try {
      //Endlos-Schleife: warte auf Client-Anforderungen
      //Abbruch durch Strg+C oder Client-Anforderung 'Exit',
      //dadurch wird der ServerSocket beendet, was hier zu einer
IOException
```

```
      //führt und damit zum Ende der run-Methode mit vorheriger
Abarbeitung der
      //finally-Klausel.
      while ( true ) {
        /*
         Zunächst wird eine Client-Anforderung
entgegengenommen(accept-Methode).
         Der ExecutorService pool liefert einen Thread, dessen
run-Methode
         durch die run-Methode der Handler-Instanz realisiert wird.
         Dem Handler werden als Parameter übergeben:
         der ServerSocket und der Socket des anfordernden Clients.
        */
        Socket cs = serverSocket.accept();  //warten auf
Client-Anforderung

        //starte den Handler-Thread zur Realisierung der
Client-Anforderung
        pool.execute(new Handler(serverSocket,cs));
      }
    } catch (IOException ex) {
      System.out.println("--- Interrupt NetworkService-run");
    }
    finally {
      System.out.println("--- Ende NetworkService(pool.shutdown)");
      pool.shutdown();  //keine Annahme von neuen Anforderungen
      try {
            //warte maximal 4 Sekunden auf Beendigung aller
Anforderungen
        pool.awaitTermination(4L, TimeUnit.SECONDS);
        if ( !serverSocket.isClosed() ) {
          System.out.println("--- Ende NetworkService:ServerSocket
close");
          serverSocket.close();
        }
      } catch ( IOException e ) { }
      catch ( InterruptedException ei ) { }
    }
  }
}

//Thread bzw. Runnable zur Realisierung der Client-Anforderungen
class Handler implements Runnable {  //oder 'extends Thread'
  private final Socket client;
  private final ServerSocket serverSocket;
  Handler(ServerSocket serverSocket,Socket client) {
//Server/Client-Socket
```

```
    this.client = client;
    this.serverSocket = serverSocket;
  }
  public void run() {
    StringBuffer sb = new StringBuffer();
    PrintWriter out = null;
    try {
      // read and service request on client
      System.out.println( "running service, " + Thread.currentThread()
);
      out = new PrintWriter( client.getOutputStream(), true );
      BufferedReader bufferedReader =
        new BufferedReader(
          new InputStreamReader(
            client.getInputStream()));
      char[] buffer = new char[100];
      int anzahlZeichen = bufferedReader.read(buffer, 0, 100); //
blockiert bis Nachricht empfangen
      String nachricht = new String(buffer, 0, anzahlZeichen);
      String[] werte = nachricht.split("\\s");  //Trennzeichen:
whitespace
      if (werte[0].compareTo("Exit") == 0) {
        out.println("Server ended");
        if ( !serverSocket.isClosed() ) {
          System.out.println("--- Ende Handler:ServerSocket close");
          try {
            serverSocket.close();
          } catch ( IOException e ) { }
        }
      }      else {  //normale Client-Anforderung
        for (int i = 0; i < werte.length; i++) {
          String rt = getWday(werte[i]);  //ermittle den Wochentag
          sb.append(rt + "\n");
        }
        sb.deleteCharAt(sb.length()-1);
      }
    } catch (IOException e) {System.out.println("IOException,
Handler-run");}
    finally {
      out.println(sb);  //Rückgabe Ergebnis an den Client
      if ( !client.isClosed() ) {
        System.out.println("****** Handler:Client close");
        try {
          client.close();
        } catch ( IOException e ) { }
      }
    }
```

```
  }  //Ende run

  String getWday(String s) {  //Datum mit Wochentag
    SimpleDateFormat sdf = new SimpleDateFormat("EEEE, dd.MM.yyyy");
    String res="";
    try {
      //Parameter ist vom Typ Date
      res=sdf.format(DateFormat.getDateInstance().parse(s));
    } catch (ParseException p) {}
    return res;
  }
}
```

```
Es folgt ein passender Client. Aufrufbeispiel für 2 Datumsangaben:
java ClientExample 24.12.1941 9.11.1989
```

TCP-Client

Hinweis: Die Erklärung der Beispiele steht als Kommentar im Quelltext.

```
import java.io.*;
import java.net.*;
import java.util.concurrent.Callable;
import java.util.concurrent.FutureTask;

public class ClientExample {
  /*
    Im 'worker' des Hauptprogramms wird wie folgt verfahren:
    o Bilde Instanz von 'FutureTask', gib ihr als Parameter eine
Instanz von
      'ClientHandler' mit, die das Interface 'Callable' (ähnlich
'Runnable')
      implementiert.
    o Übergib die 'FutureTask' an einen neuen Thread und starte diesen.
      Im Thread wird nun die 'call'-Methode aus dem Interface
'Callable' des
      ClientHandlers abgearbeitet.
    o Dabei wird die komplette Kommunikation mit dem Server
durchgeführt.
      Die 'call'-Methode gibt nun das Ergebnis vom Server an die
'FutureTask'
      zurück, wo es im Hauptprogramm zur Verfügung steht. Hier kann
beliebig
      oft und an beliebigen Stellen abgefragt werden, ob das Ergebnis
bereits
      vorliegt.
  */
  String werte;
  public static void main(String[] args) {
```

```
    if (args.length == 0) {
      System.out.println("Datum-Parameter fehlen !");
      System.exit(1);
    }
    StringBuffer sb = new StringBuffer();
    //alle Parameter zusammenfassen, getrennt durch Leerzeichen
    for (int i = 0; i < args.length; i++) {
      sb.append(args[i] + ' ');
    }
    String werte = sb.toString().trim();
    try {
      //Irrelevanter Thread zur Illustrierung der Nebenläufigkeit der
Abarbeitung
      Thread t1 = new Thread(new AndererThread());
      t1.start();
      ClientExample cl = new ClientExample(werte);
      cl.worker();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
  public ClientExample(String werte) {
    this.werte = werte;
  }
  void worker() throws Exception {
    System.out.println("worker:" + Thread.currentThread());
    //Klasse die 'Callable' implementiert
    ClientHandler ch = new ClientHandler(werte);
    boolean weiter = false;
    do {  //2 Durchläufe
      int j = 0;
      //call-Methode 'ch' von ClientHandler wird mit 'FutureTask'
asynchron
      //abgearbeitet, das Ergebnis kann dann von der 'FutureTask'
abgeholt
      //werden.
      FutureTask<String> ft = new FutureTask<String>(ch);
      Thread tft = new Thread(ft);
      tft.start();

      //prüfe ob der Thread seine Arbeit getan hat
      while (!ft.isDone()) {
            j++;  //zähle die Thread-Wechsel
        Thread.yield();  //andere Threads (AndererThread) können
drankommen
      }
      System.out.println("not isDone:" + j);
```

```
      System.out.println(ft.get());  //Ergebnis ausgeben
      if (werte.compareTo("Exit") == 0)
        break;
      weiter = !weiter;
      if (weiter) {
        //2. Aufruf für Client-Anforderung, letzten Wert modifizieren
        ch.setWerte(werte.substring(0,werte.length()-4) + "1813");
      }
    } while (weiter);
  }
}

//Enthält die call-Methode für die FutureTask (entspricht run eines
Threads)
class ClientHandler implements Callable<String> {
  String ip = "127.0.0.1";  //localhost
  int port = 3141;
  String werte;

  public ClientHandler(String werte) {
    this.werte = werte;
  }
  void setWerte(String s) {
    werte = s;
  }
  public String call() throws Exception {  //run the service
    System.out.println("ClientHandler:" + Thread.currentThread());
    //verlängere künstlich die Bearbeitung der Anforderung, um das
Wechselspiel
    //der Threads zu verdeutlichen
    Thread.sleep(2000);
    return RequestServer(werte);
  }

  //Socket öffnen, Anforderung senden, Ergebnis empfangen, Socket
schliessen
  String RequestServer(String par) throws IOException {
    String empfangeneNachricht;
    String zuSendendeNachricht;

    Socket socket = new Socket(ip,port);  //verbindet sich mit Server
    zuSendendeNachricht = par;
    //Anforderung senden
    schreibeNachricht(socket, zuSendendeNachricht);
    //Ergebnis empfangen
    empfangeneNachricht = leseNachricht(socket);
    socket.close();
```

```
    return empfangeneNachricht;
  }
  void schreibeNachricht(Socket socket, String nachricht) throws
IOException {
    PrintWriter printWriter =
      new PrintWriter(
        new OutputStreamWriter(
          socket.getOutputStream()));
    printWriter.print(nachricht);
    printWriter.flush();
  }
  String leseNachricht(Socket socket) throws IOException {
    BufferedReader bufferedReader =
      new BufferedReader(
        new InputStreamReader(
          socket.getInputStream()));
    char[] buffer = new char[100];
    //blockiert bis Nachricht empfangen
    int anzahlZeichen = bufferedReader.read(buffer, 0, 100);
    String nachricht = new String(buffer, 0, anzahlZeichen);
    return nachricht;
  }
}

class AndererThread implements Runnable {
  public void run() {
    System.out.println("  AndererThread:" + Thread.currentThread());
    int n = 0;
    int w = 25000000;
    //hinreichend viel CPU-Zeit verbrauchen
    for (int i = 1; i <= 10; i++)
      for (int j = 1; j <= w; j++) {
        if (j % w == 0)
          System.out.println("  n=" + (++n));
      }
  }
}
```

Zurück zu Netzwerkprogrammierung | Hoch zum Java Inhaltsverzeichnis | Vor zu RMI

Quelle(n) und Bearbeiter des/der Artikel(s)

Java Standard: Socket ServerSocket (java.net) UDP und TCP IP *Quelle*: http://de.wikibooks.org/w/index.php?oldid=610127 *Bearbeiter*: Juetho, ThePacker, Utnovetur, WickiLissy, 11 anonyme Bearbeitungen

Quelle(n), Lizenz(en) und Autor(en) des Bildes

Bild:Go-previous.svg *Quelle*: http://de.wikibooks.org/w/index.php?title=Datei:Go-previous.svg *Lizenz*: unbekannt *Bearbeiter*: The people from the Tango! project

Bild:Go-up.svg *Quelle*: http://de.wikibooks.org/w/index.php?title=Datei:Go-up.svg *Lizenz*: unbekannt *Bearbeiter*: The people from the Tango! project

Bild:Go-next.svg *Quelle*: http://de.wikibooks.org/w/index.php?title=Datei:Go-next.svg *Lizenz*: unbekannt *Bearbeiter*: The people from the Tango! project

Lizenz