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 Verf€gung.

Der folgende Code soll das Vorgehen verdeutlichen. Er enth•lt 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 zur€ck und beendet sich. Hat der Client die zur€ckgesendete Nachricht empfangen, beendet auch er sich. Hinweise f€r intelligentere Server und Clients finden sich weiter unten.

Ein primitiver Server

```
 // Server.java

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

                    {
                  void main(String[] args) {
       Server server =     Server();
           {
           server.test();
        }       (IOException e) {
            e.printStackTrace();
        }
    }
    void test()        IOException {
       int port = 11111;
       java.net.ServerSocket serverSocket =
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)        IOException {
       java.net.Socket socket = serverSocket.accept(); // blockiert,
bis sich ein Client angemeldet hat
             socket;
    }
    String leseNachricht(java.net.Socket socket)        IOException {
        BufferedReader bufferedReader =
```

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

Ein primitiver Client

```
 // Client.java

 // import java.net.Socket;
                ;


                   {
                 void main(String[] args) {
       Client client =     Client();
           {
           client.test();
       }       (IOException e) {
           e.printStackTrace();
       }
     }
     void test()        IOException {
       String ip = "127.0.0.1"; // localhost
       int port = 11111;
       java.net.Socket socket =     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)
       IOException {
```

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

Erkl€rung des Codes

Die Klasse `java.net.ServerSocket` dient in erster Linie zur Anmeldung von Clients. Sie h,rt nicht - wie man vielleicht erwarten k,nnte - auf die Nachrichten von Clients. Im Konstruktor wird der Port (im Beispiel 11111) €bergeben (die IP-Adresse des Servers ist nat€rlich die IP-Adresse des Computers, auf dem der Server l•uft). Die Methode `accept()` wartet so lange, bis sich ein Client verbunden hat. Dann gibt sie einen `java.net.Socket` zur€ck.

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

- im Server dient sie dazu, sich den angemeldeten Client zu merken, auf seine Nachrichten zu h,ren 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 €bergeben. Mit dem Aufruf des Konstruktors im Client wird die Verbindung hergestellt - beim Server gibt `accept()` den verbundenen Client zur€ck.

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

Wie s€he 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. F€r 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 m‚glich ist, allen angemeldeten Clients eine Nachricht zu senden.

Auch im Client gibt es einen eigenen Thread, der auf Nachrichten des Servers wartet. W•hrend man es sonst bei Methodenaufrufen gewohnt ist, dass man auf den R€ckgabewert (und sei es `void`) der Funktion wartet (synchron), verl•uft die Kommunikation hier asynchron.

Eine weitere Schwierigkeit ist das Lesen von Nachrichten der Gegenseite. Im Beispiel werden Nachrichten der L•nge 200 gelesen (das reicht f€r "Hello, world!"). Es hilft an dieser Stelle nicht wirklich, den Buffer deutlich gr„er zu machen (aus Performance-Gr€nden wird man ihn nat€rlich etwa auf 1024 Bytes vergr„ern), Beim Lesen treten n•mlich zwei Probleme auf:

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

Deshalb m€ssen Server und Client ein eigenes Protokoll vereinbaren. Denkbar (und €blich) sind dabei mehrere Varianten:

- Alle Nachrichten haben eine feste L•nge
- Zu Beginn jeder Nachricht (z.B. in den ersten 8 Zeichen) wird die L•nge der Nachricht angegeben; es werden dann zun•chst 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 Nebenl€ufigkeit
  Im main-Thread (Methode 'worker') kann jederzeit abgefragt werden, ob die
  Antwort vom Server eingetroffen ist.
```

TCP-Server

Hinweis: Die Erkl•rung der Beispiele steht als Kommentar im Quelltext.

```
                                                ;
                                            ;
                                           ;
                    ;
                     ;
                         ;
                       ;


                                {
  /*
    - 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'.
  */
                 void main(String[] args)        IOException {
          ExecutorService pool;
          ServerSocket serverSocket;
    int port = 3141;
    String var = "C";
    String zusatz;
       (args.length > 0 )
      var = args[0].toUpperCase();
       (var == "C") {
      //Liefert einen Thread-Pool, dem bei Bedarf neue Threads
hinzugef•gt
      //werden. Vorrangig werden jedoch vorhandene freie Threads
benutzt.
      pool = Executors.newCachedThreadPool();
      zusatz = "CachedThreadPool";
    }      {
      int poolSize = 4;
      //Liefert einen Thread-Pool f€r maximal poolSize Threads
      pool = Executors.newFixedThreadPool(poolSize);
      zusatz = "poolsize="+poolSize;
    }
    serverSocket =     ServerSocket(port);
    //Thread zur Behandlung der Client-Server-Kommunikation, der
Thread-
    //Parameter liefert das Runnable-Interface (also die run-Methode
f•r t1).
```

```
    Thread t1 =     Thread(    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(
          Thread() {
               void run() {
              System.out.println("Strg+C, pool.shutdown");
              pool.shutdown();  //keine Annahme von neuen Anforderungen
              {
                //warte maximal 4 Sekunden auf Beendigung aller
Anforderungen
            pool.awaitTermination(4L, TimeUnit.SECONDS);
               (!serverSocket.isClosed()) {
              System.out.println("ServerSocket close");
              serverSocket.close();
            }
          }       ( IOException e ) { }
                ( InterruptedException ei ) { }
            }
      }
    );
//
  }
}

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

```
      //f€hrt und damit zum Ende der run-Methode mit vorheriger
Abarbeitung der
      //finally-Klausel.
            (      ) {
        /*
         Zun•chst 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 €bergeben:
         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(    Handler(serverSocket,cs));
      }
    }       (IOException ex) {
      System.out.println("--- Interrupt NetworkService-run");
    }
            {
      System.out.println("--- Ende NetworkService(pool.shutdown)");
      pool.shutdown();  //keine Annahme von neuen Anforderungen
          {
            //warte maximal 4 Sekunden auf Beendigung aller
Anforderungen
        pool.awaitTermination(4L, TimeUnit.SECONDS);
           ( !serverSocket.isClosed() ) {
          System.out.println("--- Ende NetworkService:ServerSocket
close");
          serverSocket.close();
        }
      }       ( IOException e ) { }
            ( InterruptedException ei ) { }
    }
  }
}

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

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

```
  }  //Ende run

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

```
Es folgt ein passender Client. Aufrufbeispiel f•r 2 Datumsangaben:
java ClientExample 24.12.1941 9.11.1989
```

TCP-Client

Hinweis: Die Erkl•rung der Beispiele steht als Kommentar im Quelltext.

```
                ;
                 ;
                                ;
                                  ;


                           {
  /*
    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' (•hnlich
'Runnable')
      implementiert.
    o ‚bergib 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
durchgef€hrt.
      Die 'call'-Methode gibt nun das Ergebnis vom Server an die
'FutureTask'
      zur€ck, wo es im Hauptprogramm zur Verf€gung steht. Hier kann
beliebig
      oft und an beliebigen Stellen abgefragt werden, ob das Ergebnis
bereits
      vorliegt.
  */
  String werte;
               void main(String[] args) {
```

```
       (args.length == 0) {
      System.out.println("Datum-Parameter fehlen !");
      System.exit(1);
    }
    StringBuffer sb =     StringBuffer();
    //alle Parameter zusammenfassen, getrennt durch Leerzeichen
        (int i = 0; i < args.length; i++) {
      sb.append(args[i] + ' ');
    }
    String werte = sb.toString().trim();
        {
      //Irrelevanter Thread zur Illustrierung der Nebenl•ufigkeit der
Abarbeitung
      Thread t1 =     Thread(    AndererThread());
      t1.start();
      ClientExample cl =     ClientExample(werte);
      cl.worker();
    }       (Exception e) {
      e.printStackTrace();
    }
  }
         ClientExample(String werte) {
        .werte = werte;
  }
  void worker()        Exception {
    System.out.println("worker:" + Thread.currentThread());
    //Klasse die 'Callable' implementiert
    ClientHandler ch =     ClientHandler(werte);
    boolean weiter =      ;
       {  //2 Durchl•ufe
      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 =     FutureTask<String>(ch);
      Thread tft =     Thread(ft);
      tft.start();

      //pr€fe ob der Thread seine Arbeit getan hat
            (!ft.isDone()) {
            j++;  //z•hle die Thread-Wechsel
        Thread.yield();  //andere Threads (AndererThread) kfnnen
drankommen
      }
      System.out.println("not isDone:" + j);
```

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

//Enth•lt die call-Methode f€r die FutureTask (entspricht run eines
Threads)
                            Callable<String> {
  String ip = "127.0.0.1";  //localhost
  int port = 3141;
  String werte;

        ClientHandler(String werte) {
       .werte = werte;
  }
  void setWerte(String s) {
    werte = s;
  }
        String call()        Exception {  //run the service
    System.out.println("ClientHandler:" + Thread.currentThread());
    //verl•ngere k€nstlich die Bearbeitung der Anforderung, um das
Wechselspiel
    //der Threads zu verdeutlichen
    Thread.sleep(2000);
          RequestServer(werte);
  }

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

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

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

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

Zur€ck 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